Journal of Network and Computer Applications 208 (2022) 103494

Contents lists available at ScienceDirect

Journal of Network and Computer Applications

journal homepage: www.elsevier.com/locate/jnca

# *iContainer*: Consecutive checkpointing with rapid resilience for immortal container-based services

Sang-Hoon Choi [a], Ki-Woong Park [b,*]

[a] SysCore Lab, Sejong University, Seoul 05006, South Korea
[b] Department of Information Security, and Convergence Engineering for Intelligent Drone, Sejong University, Seoul 05006, South Korea

ARTICLE INFO

*Keywords:*
Cloud computing
Container
Mission-critical system
Failover
Memory snapshot

ABSTRACT

Container-based cloud services that can achieve scalability at a low cost by dividing a complex system into instances, functions, or applications are essential for the operation of mission-critical industrial systems. Mission assurance and survivability are required as core elements and unique functions for these services to be the basic environment of major systems. In particular, a mission-critical system operating environment must guarantee service resilience that can provide stable services even in a situation where it is impossible because of cyberattacks or service failures. To solve this problem, we propose *iContainer*, which stands for Immortal Container. It provides stable services by quickly returning to the point desired by a user when a failure occurs by continuously recording container services. If efficient checkpoints are available, the lifecycles of containers are recorded and services are rolled back to a previous point desired by a user when a critical event occurs. *iContainer* has three contributions. First, it minimizes checkpointing operations through checkpoint zoning. We remove unnecessary checkpointing operations through a semantic-aware hot/cold container classification scheme for zones where data changes rarely occur. Second, rapid checkpointing is achieved through dirty-page tracking. We minimize checkpoint data (read/write) operations by efficiently tracking the memory area. Consequently, *iContainer* reduces the checkpoint execution time by 3.27 times compared to the conventional checkpointing scheme, and the size of the data generated by repetitive checkpointing is reduced by 69.2%. Third, *iContainer* includes rapid checkpoint restoration and flexible restore points. We designed the software-defined checkpoint/restore (SDCR) tool, which enables the rapid restoration and flexible selection of checkpoints and restore points. Experimental results show that it takes 337 ms on average from the detection of a service failure until stable service operation. Thus, the rollback time of SDCR is approximately 1.93 times faster than the conventional checkpointing tool, checkpoint/restore in userspace (CRIU). The experiment was conducted in an environment where a web service was operated. Moreover, *iContainer* can be utilized for service error restoration and as data for identifying the causes of accidents in the event of an attack or security accident because it records the lifecycle of containers through checkpoints.

## 1. Introduction

Various mission-critical systems used in autonomous vehicles, aircrafts, and industrial fields require cloud computing services (Dutta and Dutta, 2019; Hyseni and Ibrahimi, 2017; Pierleoni et al., 2019; Rajan, 2018; Elazhary, 2019). Furthermore, movement to the cloud is accelerating in areas such as administration, public agencies, and national defense (Khan, 2016; Sun, 2020; Yan et al., 2015; Joshi et al., 2012; Bamiah and Brohi, 2011). In particular, container-based services have attracted attention because they have advantages in cost, maintenance, and operations (Kavis, 2014; Pahl, 2015; Van Eyk et al., 2017). Container technology is widely used in cloud computing because it allows the operation of instances with fewer resources compared to hypervisor-based services (Bernstein, 2014; Chung et al., 2016; Ruan et al., 2016; Desai et al., 2013; Jiang et al., 2019). The advantages of containers are demonstrated by various services, used by several leading vendors such as AWS, MS, IBM, and Google, as business models (Cloud, 2011; Copeland et al., 2015; Zhu et al., 2009; Garraghan et al., 2013). However, the potential threats in cloud computing pose concerns about the introduction of the cloud in major industries (Singh and Chatterjee, 2017). Cloud security alliance has identified various security threats about cloud that prevent the use of common cloud to store sensitive data by such organizations as financial agencies, medical institutions, federal and state governments, and national defense agencies (Alliance, 2021; Kumar and Goyal, 2021). Therefore,

* Corresponding author.
*E-mail addresses:* csh0052@gmail.com (S.-H. Choi), woongbak@sejong.ac.kr (K.-W. Park).

https://doi.org/10.1016/j.jnca.2022.103494
Received 5 February 2022; Received in revised form 6 June 2022; Accepted 7 August 2022
Available online 24 September 2022

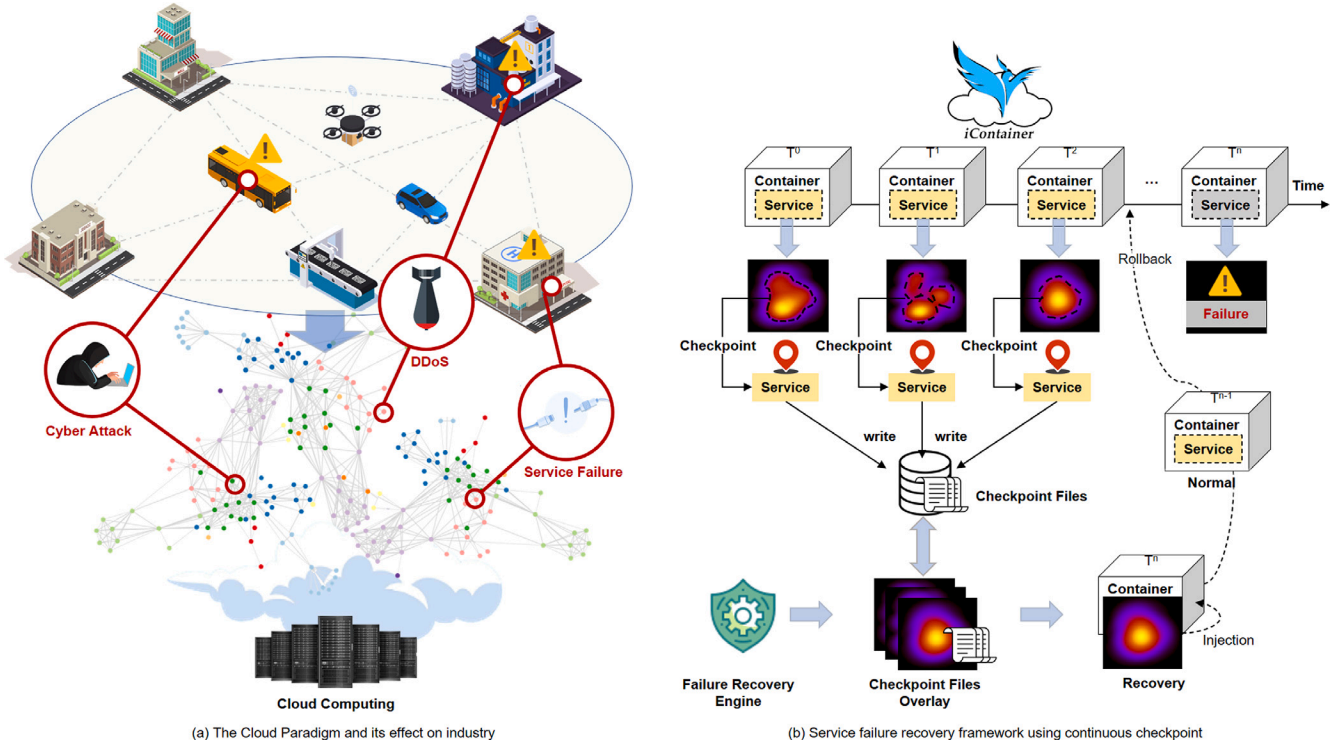


**Fig. 1.** Overview of cloud service failure recovery with *iContainer*.

the utilization of container-based cloud services in industrial systems require countermeasures against cyber security threats, service failure, and damage. Particularly, mission-critical systems must guarantee rapid system recovery and resilience because they must operate without interruption because one trouble inside a mission-critical system can lead to a mission failure or disaster. An example is the recent oil supply interruption accident in the eastern US due to an attack on the colonial pipeline (Analytica, 2021b,a). The unexpected oil supply interruption made citizens panic. It took six days for the oil pipeline to be restored to stable condition. A fast response could have been possible if a solution for recovery from service failure such as cyberattack were prepared. Fig. 1(a) shows that resilience for service failure must be guaranteed in a situation where the operations of industries, including mission-critical systems, should be migrated to the cloud in the future.

Various methods have been proposed for detecting and restoring service failures (Matos et al., 2018, 2021; Venkatesh et al., 2019a; Amoon et al., 2019). However, in most of them, the recoverable services are limited or user intervention is required. Furthermore, because service restoration is time-intensive, it is difficult to apply these methods to mission-critical systems that require immediate service restoration. To achieve resilience to cloud service failure, this study proposes *iContainer*, which continuously records containers and quickly returns to a point desired by the user, to provide stable services (Pickartz et al., 2016; Widjajarto et al., 2021; Stoyanov and Kollingbaum, 2018; Mirkin et al., 2008). The proposed framework checkpoints the state of container services in operation using only a few operations and quickly restores the stable state when a container service failure occurs. Moreover, as *iContainer* secures various rollback points through consecutive checkpoints, it can quickly respond to unexpected service failures. Fig. 1(b) shows the design of *iContainer*. This study makes three major contributions.

First, unnecessary operations are removed through checkpoint zoning. Methods to checkpoint containers have been studiedFirst, unnecessary operations are removed through checkpoint zoning. Although various methods for checkpointing containers have been proposed, they do not consider the design of repetitive checkpoints because their checkpoints are designed for live migration (Ahmed et al., 2020; Chen, 2015; Laadan and Hallyn, 2010). *iContainer* quickly secures various rollback points through consecutive container checkpoints. Therefore, it is crucial to optimize checkpoint operations. The data produced when creating consecutive checkpoints contains duplicate data, for which scanning and saving in storage constitute unnecessary operations. By efficiently removing duplicate data, the operations caused by the checkpoints can be minimized. For example, as set or installed environmental variables or libraries do not change frequently, they are not needed at every checkpoint. This study proposes a hot/cold classification scheme that is friendly to container-repetitive checkpoints. The proposed scheme minimizes unnecessary checkpoint operations and duplicate data, classifying checkpoint zones as hot or cold by scanning containers in operation. Cold zones contain data such as library allocation, environmental variables, and network information, which are not easily changed once they are set. By contrast, hot zones contain data such as file system and memory information, which are continuously changed. *iContainer* performs checkpointing predominantly in the hot zone; checkpointing is performed in the cold zone only when a file system event occurs. This method effectively removes unnecessary search operations and duplicate data from the cold zone. Nevertheless, *iContainer* has a limitation in that it can only be used in the Docker Engine environment. This is because the information required to separate the checkpoint zones of the container is collected through Docker's inspection function and union filesystem analysis (Dua et al., 2016).

Second, rapid checkpointing can be implemented using dirty-page tracking. We divided the checkpoint zones into hot and cold zones to remove unnecessary search and duplication operations that occur in checkpoints. The hot zone is characterized by frequent updates. In particular, in the memory corresponding to the hot zone, data modulations continuously occur. The memory is characterized by locality in which only specific areas are repeatedly changed instead of uniform changes over the entire area. We selected Ubuntu as the base image of the container, activated the Apache2 service, and dumped the memory 100 times with a 2 s cycle. Subsequently, we compared the memories and discovered that approximately 78% or more of the

memory areas were duplicated. Therefore, we concluded that partial checkpointing by tracking only the changed areas is more effective than checkpointing the entire area of the hot zone. This is because the memory has spatiotemporal locality, a characteristic that is actively utilized in the live migration field (Pagani et al., 2019; Mao et al., 2015; Hu et al., 2011; Zheng et al., 2011). This method involves tracking and transmitting memory change information, thereby minimizing service downtime that may occur during migration. *iContainer* uses dirty-page tracking to track the memory changes without the duplication removal operation of the memory. If the allocated memory area is processed by dirty bits and managed as a bitmap, a page fault is caused whenever a memory write operation occurs. Therefore, the changed memory data can be stored without duplication removal operation if the memory area in which a page fault occurred is tracked and checkpointed. Additionally, we discovered that most performance delays in the checkpointing operation process occur in write. Hence, we used some space of the host DRAM for checkpointing to minimize any performance overhead due to the data write operation caused by checkpointing. It took 290 ms on average to generate the checkpoint of container with a 128 MB memory. This performance is approximately 3.27 times faster than the conventional checkpoint/restore in userspace (CRIU) tool (Virtuozzo, 2011). Furthermore, when checkpoints were consecutively generated 100 times, the cumulative data size decreased by 69.2%. We compared iContainer with CRIU because the latter will be used for the checkpointing function of Docker Engine in the future. CRIU can currently be used in the experimental mode of Docker Engine.

Third, random-access performance for checkpoint restoration is guaranteed and user-defined checkpoint/restore is supported. *iContainer* restores the state to facilitate the provision of stable services by rolling back to the optimal point that was checkpointed when the container service could not work normally. The phenomenon that services in a container are stopped can occur suddenly at any time; hence, it is critical to efficiently checkpoint and quickly restore the services. We proposed a scheme to classify the checkpoint zones into hot/cold and to record changes by tracking them to quickly perform checkpoints. Because our scheme partially performs checkpointing for hot and cold zones, a post-processing operation called "checkpoint reprocessing" is needed for the checkpoints that are necessary in the rollback. To generate a complete checkpoint that can be rolled back to, $\oplus$ operations must be repeatedly performed from the first checkpoint to the checkpoint to which the user wishes to roll back. Although *iContainer* secures many restore points by optimizing the checkpointing operations, as the size of the partial checkpoints accumulates, the operations required for restoration increase proportionally, thereby increasing the rollback time. Therefore, the time required for checkpoint restoration increases in proportion to the checkpoint accumulated. To solve this problem, we designed the concept of save point. SDCR inserts a complete checkpoint between incomplete checkpoints through background threads separated into control groups (cgroups), which helps to maintain a constant checkpoint restoration time. SDCR predicts the time required for the restoration operation based on the size of the data and generates a save point that reflects the partial checkpoints up to the current point at which the checkpoints accumulated over a certain size. Creating a save point can solve the problem of having to perform $\oplus$ operations from the first checkpoint to the rollback point, and because the resources this uses are separate from the checkpoint operations, it does not produce noise in the target service and checkpoint performance. Furthermore, it provides a performance advantage when restoring the container by storing the last save point and checkpoint used for restoration in the dram space of the host zone. It took 337 ms on average to restore a stable container state from a service unavailable state. This implies that it became approximately 1.93 times faster than restoring the container using CRIU. Furthermore, we designed the SDCR to enable users to freely select checkpoints and restore points. Checkpointing and restore point selection are critical because, in the case of checkpoints, the information lost is different depending on the creation criteria, and creating unnecessary checkpoints causes computational and spatial overheads. SDCR freely designs the checkpoint and restore point. For example, when a service error occurs, users can freely move to a desired point through the timeline. However, incorrect restore point selection causes an infinite loop and delays service error restoration time. *iContainer* allows the flexible definition of checkpoint creation and restore points. The checkpoint creation criteria and utility of the restore point selection are explained through usage scenarios.

The remainder of this paper is organized as follows. Section 2 introduces related works. Section 3 describes the design of *iContainer* in detail. Section 4 evaluates the performance of *iContainer*. Section 5 explains various scenarios for using *iContainer*. Finally, Section 6 summarizes the findings of this study and suggests future study topics.

## 2. Related work

Cloud service availability attracted the interest of many researchers even before the activation of cloud services. Consequently, several studies have proposed methods to quickly respond to service interruptions due to malicious attacks or defects of the service. These studies are traditionally classified into log-based or checkpoint-based restoration (Treaster, 2005). In this section, the limitations of conventional service restoration methods are discussed.

### 2.1. Studies on log-based service restoration

The log-based service restoration technology has been used for a long time in the database field. The log-based restoration technology analyzes collected log information and enables return to stable service state by performing rollback for abnormal work behaviors (Passerini et al., 2009; Pecchia et al., 2011). Taser is a system that performs restoration based on logs when a system is infected with malware (Goel et al., 2005). Because the system is restored based on logs, all the behaviors of files, processes, and networks stored in the system are recorded and analyzed. However, users must decide in a situation where the system is infected with malware and service is unavailable. Moreover, once the file system is damaged, it cannot be restored. The back to the future is also a study on log-based system restoration. This study secures data required for restoration by distinguishing unreliable and reliable software and recording logs only for reliable software. Therefore, restoration is impossible if the reliable software is locked and user intervention is necessary to collect the log data required for restoration. In 2018, Matos et al. proposed RockFS as a method to strengthen storage security in the cloud environment (Matos et al., 2018). RockFS logs file system activities and uses them for recovery in the event of an attack. However, because storage recovery takes approximately 40 s, it is difficult to apply RockFS to real-time service environments. Consequently, Matos et al. also recently proposed Sanare, a system that can effectively recover when a web application is attacked (Matos et al., 2021). Sanare has the advantage of being able to self-recover from a service failure within 4 s of a web service attack. However, the services that Sanare can recover are limited to those that perform HTTP requests. Most log-based service recovery solutions require user intervention or have long recovery times because a log analysis process is necessary. Moreover, if the error occurs in a log category that cannot be collected or is not specified by the user, recovery becomes impossible. Therefore, this method has limitations in a cloud service that must ensure availability through immediate restoration.

### 2.2. Studies on snapshot-based service restoration

Recently, hypervisor/container technologies have been widely used with the increasing scope of use and interest in cloud platforms (Khan et al., 2020). In the virtualization-based service environment the entire area of target services is snapshot more easily without contamination

compared to the traditional system structure. Owing to such an advantage, snapshot-based service restoration has been actively studied in recent years. TimeVM stores data by snapshotting the same VM as the VM in which a service is operated and generates a slight delay in the processing of traffic requested to the service VM (Elbadawi and Al-Shaer, 2009). Subsequently, it delivers the snapshot to the cloned VM by migration. When the VM running the service becomes inoperable because of malicious traffic, TimeVM restores stable service by moving malicious traffic to the cloned VM and replaying and migrating the remaining traffic. However, this method only responds to known threatening traffic and takes more than 30 s for service restoration. In a hypervisor environment, the scope of targets to be snapshot becomes large because each service uses fully independent kernel and memory area. Therefore, it is difficult to use hypervisor-based snapshot restoration methods in a distributed service environment that requires immediate restoration because it requires many operations and time for snapshot and restoration. The current trend is to use container engine, which is a relatively light virtualization technology, as an alternative to hypervisor. Consequently, several research results about checkpoint/restore for container instance have been reported. One project output that is widely used in the checkpoint/restore operation for processes such as instance/container is CRIU (Virtuozzo, 2011). CRIU can save the current state of a running process as an image file through checkpoints, and it can use the saved image to recover the process. Currently, the checkpoint function of CRIU is officially provided in Docker Engine's experimental mode. As containers operate as processes, if efficient container checkpoints are possible, then they can be used in the security and service recovery fields. Considering this fact, numerous studies have been conducted with the aim of strengthening the fault tolerance of services using CRIU and improving the performance of CRIU. Araujo et al. used CRIU as the base technology for the honey-patches of services in operation. A bait service is created through the honey-patch, which solves the availability problem that may occur from a service attack. In 2010, Goiri et al. proposed a checkpoint-based fault-tolerant infrastructure for cloud service providers (CSPs). However, the infrastructure has the limitation of only being applicable to a service environment using Another Union File System (AUFS) (Goiri et al., 2010). Karhula et al. utilized checkpoints to solve the fault tolerance and service availability problems that may arise when the Function as a Service (FaaS) model is applied in an Internet of Things (IoT) network (Karhula et al., 2019). In 2022, Mudassar et al. proposed a method that uses CRIU to ensure the fault tolerance of nodes in edge computing environments and improve availability (Mudassar et al., 2022). Specifically, their method uses CRIU as a strategy for the distributed backup of nodes in resource-limited edge computing environments. In 2018, Ashton Webster proposed a method for restoring a system infected with malware, using checkpoints (Webster et al., 2018). In that study, it took 2.8 s on average to restore to the stable state through checkpoints. This is more than ten times higher performance compared to that of the hypervisor-based restoration technology. However, restoration is impossible when attacks such as ransomware are launched because checkpointing is performed after the system is infected with malware and when the process is already damaged by malware. Furthermore, when it is applied to a distributed application environment, one service container is locked down for 2.8 s or more. Hence, if a response code awaiting processing is not prepared by different services that are organically interconnected, the total service may be locked down for approximately 2.8 s or more. This is because CRIU was mainly designed for complete checkpointing rather than for performance. Consequently, various methods for minimizing the checkpoint/restore time required by CRIU have been proposed (Venkatesh et al., 2019a; Webster et al., 2018). In 2019, Amoon et al. proposed a method that ensures the fault tolerance of services by flexibly controlling the checkpoint performance intervals (Amoon et al., 2019). Their proposed method dynamically controls the checkpoint interval according to the service failure rate. However, failure rate-based post-processing methods cannot respond when a sudden service failure arises, and because checkpointing is performed according to the failure rate, important checkpoint data may be lost. Awalia et al. proposed a method that improves service resilience via checkpoints in a parallel computing environment. Their proposed method creates periodic checkpoints to secure restoration points for the service. However, the method does not consider the checkpoint performance cycle, checkpoint operation optimization, and data similarity (Putri et al., 2020). Venkatesh et al. proposed a method to rapidly checkpoint/restore containers through the CRIU running on the Docker Engine (Venkatesh et al., 2019a). This study optimized the read/write operations that occur during checkpointing using in-memory and copy-on-write (COW) techniques. This reduced the checkpointing time by 10%–200% and restoration time up to 42.3% compared to the CRIU. Because that study optimized operations for CRIU from the simple checkpointing perspective, it did not consider the homogeneity of containers, checkpoint cycle, and management and restoration points. Therefore, appropriate requirements and factors for a specific environment should be reflected in the design for these research results to be used for special purposes in cloud services.

## 3. Design of *iContainer*

We propose *iContainer*, which stands for Immortal Container. It provides stable services by quickly returning to the point desired by a user when a failure occurs, by continuously recording container services. In this section, the overview and design of the *iContainer* are discussed.

### 3.1. iContainer overview

*iContainer* quickly restores the stable state using a checkpoint when a service failure occurs. Consecutive checkpoints are required for restoration from service failure. Therefore, we devised methods to efficiently perform consecutive checkpointing. To use *iContainer* for restoration from container service failures, the following requirements are considered. First, the time required for generating the checkpoints of container must be minimized. Second, the size of the checkpoint data must be minimized. Finally, it must be possible to restore the system from service errors within a short time. A few assumptions are required to achieve this goal. First, it is assumed that the service running in the container is not specialized for memory changes. In a database service that causes continuous memory changes, a performance delay can occur during checkpoint creation and the generated checkpoint data may increase exponentially. Second, it is assumed that the container does not provide teletypewriter (TTY). The connection of TTY cannot be maintained because the transmission control protocol (TCP) connection is temporarily lost in the process of restoring the service from the checkpoint. We established these requirements and designed *iContainer* as shown in Fig. 2.

*iContainer* has three contributions. First, checkpoint operations are minimized through checkpoint zoning. We divide the checkpoint target zones into hot and cold zones to minimize the data write operations that are duplicated in the repetitive checkpoint operations. The cold zone is a checkpoint target with duplicate data because data changes rarely occur. The hot zone is a checkpoint target in which data changes frequently occur. An example is the memory area. We extract libraries, environmental variables, and network information by analyzing the containers in operation and manage the containers by a cold mapping table. The *iContainer* performs checkpointing only when a file system event occurs for the zones registered in the cold mapping table and mainly for the hot zone. The reason for this design is to remove unnecessary checkpoints because data changes occur in the cold zone only when a file system event occurs. Second, checkpoint operations are accelerated through dirty-page tracking. In the cold zone, checkpointing is performed for changed data when a file system event occurs. However, the hot zone, where data changes occur frequently, requires

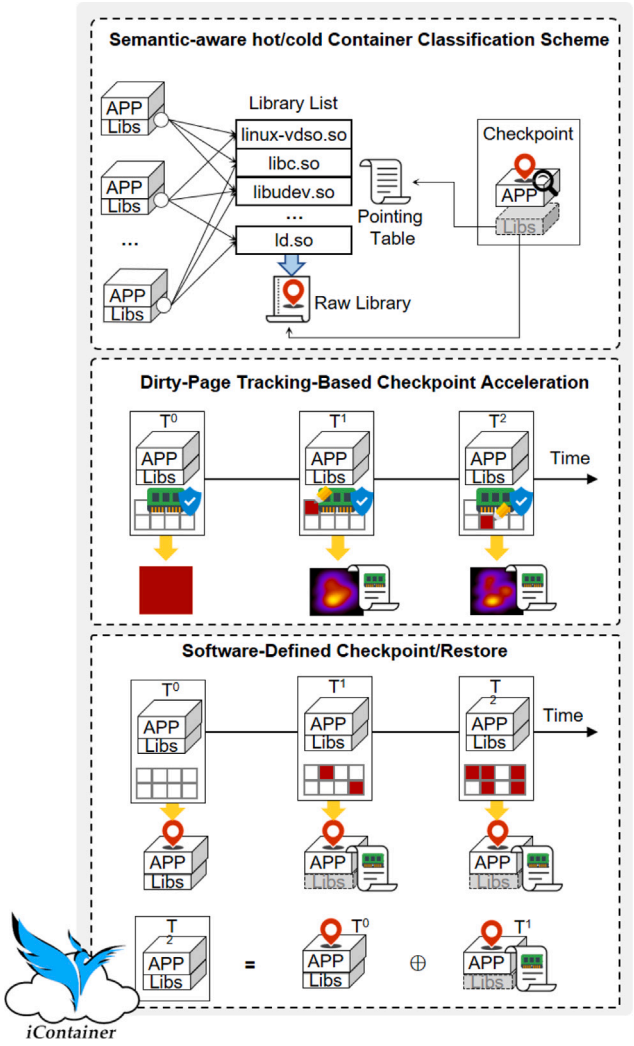


**Fig. 2.** Overview of the Immortal Container Framework.

an efficient checkpointing method. Therefore, we devised a method to efficiently track data changes by managing the container memory area corresponding to the hot zone by write-protect. When write-protect is specified for the memory area, the write performance may become low; however, the changes of the memory can be tracked by detecting page-fault events. Third, checkpoint random-access performance can be guaranteed for service failure recovery. *iContainer* restores the stable state using a checkpoint when a service failure occurs. In this process, it backs up the stable container environment through repetitive checkpoint creations. We divide the checkpoint targets into hot and cold zones and create partial checkpoints by tracking changed data for efficient checkpoint creation. However, an additional operation to convert to a complete checkpoint is required to use partial checkpoints for service failure recovery. The transformation to a complete checkpoint is achieved through repetitive ⊕ operations from the first checkpoint to a specific checkpoint. Thus, the time required for checkpoint transformation increases proportionally with the accumulation of incomplete checkpoints. To solve this problem, we actualized the concept of save point. A save point is a complete checkpoint that is inserted between incomplete checkpoints to accelerate the checkpoint restoration speed. A thread in the background generates and inserts a save point between incomplete checkpoints based on the user rules, thus, keeping the restoration time constant. The proposed framework is loaded in the container environment where Docker Engine is running.

**Table 1**
Frequency of syscalls called at checkpoint.

| Symbol | Calls | Percentage |
|---|---|---|
| memcpy_orig | iter_file_splice_write<br>vfs_iter_write<br>do_iter_write<br>do_iter_readv_writev<br>ext4_file_write_iter<br>_generic_file_write_iter<br>iov_iter_copy_from_user_atomic | 86% |
| _lock_text_start | arch_ptrace<br>ptrace_request<br>ptrace_resume<br>wake_up_state<br>try_to_wake_up | 6.5% |
| SYSC_kill | kill_pid_info<br>group_send_sig_info<br>do_send_sig_info | 6.5% |
| Etc. | – | 1% |

### 3.2. Semantic-aware hot/cold container classification scheme

*iContainer* is a solution that generates checkpoints for previous points by repeatedly taking snapshots of container states and returns to a stable checkpoint when an abnormal behavior is detected. However, service recovery methods based on snapshots or checkpoints are limited by service downtime and low availability. This is because a large overhead is generated by checkpoint operations. In particular, most checkpointing overhead occurs in the write operation (Venkatesh et al., 2019b).

Table 1 shows the ratio of syscalls called when checkpointing is performed. The read-to-write operations must be performed repeatedly to store the container memory and file system data as one file. In the process, the number of write operation calls increases in proportion to the size of the resource to be copied to the storage. To reduce the number of write operation calls, we devise a scheme to minimize unnecessary checkpoint operations and duplicate data by classifying the checkpoint zones into hot/cold through container scanning. A container instance is composed of core internal elements of the host operating system kernel required for operation and external elements such as libraries, environmental variables, and software sets. Therefore, the service-type containers that are operated in one physical environment use the same internal elements and are operated by the unique individual external elements of each container. The internal and external elements can be separated into the cold and hot zones, respectively. In the cold zone, libraries are core resources that are commonly used by several containers. Several libraries are used in a duplicated manner in an environment where multiple containers are operated simultaneously. Therefore, in a large-scale container operation environment, it is advantageous to exclude duplicate areas from checkpointing by efficiently managing the library area. We scan containers using the same library and simply point to the corresponding area during checkpointing to effectively manage relatively large libraries.

Fig. 3 shows the process of combining and managing checkpoint libraries. ① Because containers are managed by a process in the operating system (OS), the library information used in the containers can be quickly obtained by scanning a partial area of the process. When the lib scanning engine is called, the library information that is being used by the containers in operation is extracted. ② All libraries are created as a library checkpoint file by overlaying the library information of all containers in operation as an image; the library information for each container is created and managed as a cold pointing table. ③ When a checkpoint event occurs in a specific container, the resource registered in the cold pointing table is excluded from the checkpoint target. Because checkpoint operations for the library area registered in the table do not occur any more, the duplicate track and write operations

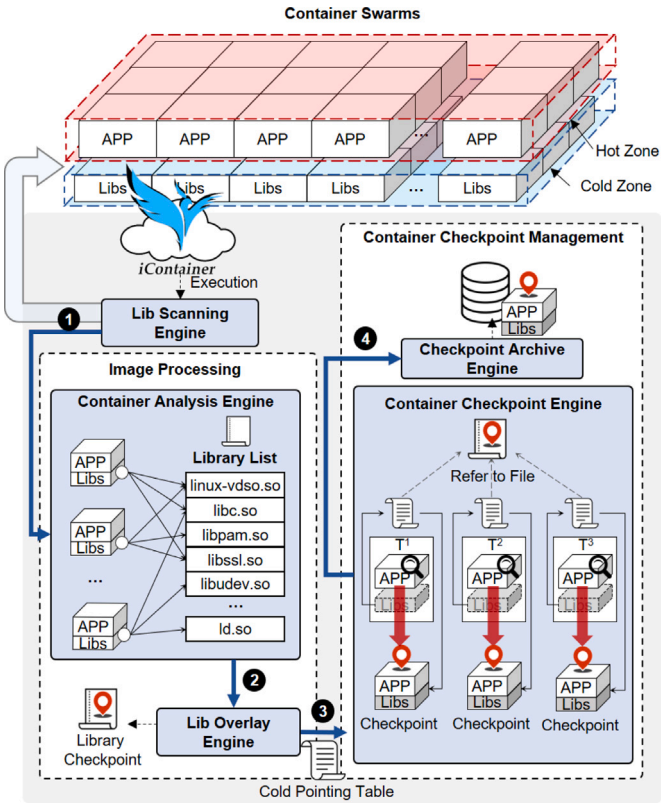


**Fig. 3.** Minimization of container checkpoint operations through container image analysis.

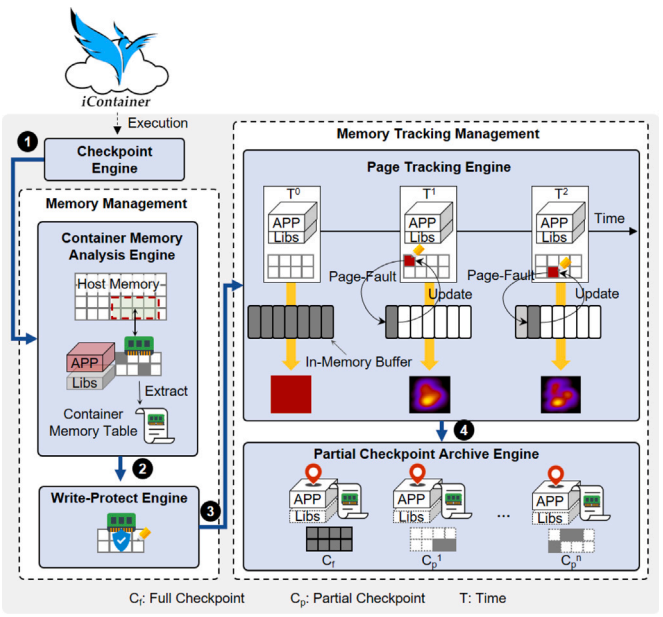


**Fig. 4.** Tracking container memory change information for checkpoint acceleration.

from repetitive checkpoints is removed. Therefore, we focus only on the change information of the hot zone and allocated memory to track and record the container states, excluding unnecessary duplicate areas. ④ Once the checkpointing operation for the hot zone is completed, the cold pointing table information is combined with the hot zone and stored. Additionally, *iContainer* provides a file system tracking feature to track cold zone. For efficient detection of file system event, it was implemented using fcntl-inotify. Inotify is a linux kernel subsystem that efficiently notifies events specified by user through file system monitoring. We designed it to monitor only creation, removal, and modification operations among the various file system events. The file system event detection is used to efficiently track the change information of the cold zone. As Inotify, which we implemented in *iContainer*, is an event-driven mechanism, it is possible to track the change information of the file system only by detecting events without duplication removal operations. Thus, file system event detection can be used to efficiently extract duplicate cold zone data. Furthermore, the file system event detection feature is used in the user-defined checkpoint creation rules. Examples of utilization are provided in the case study.

### 3.3. Dirty-page tracking-based checkpoint acceleration

The tracking and recording of the memory area require the most operations when performing checkpointing for the hot zone area. The memory is classified as a hot zone because it continuously changes. To minimize memory recording operations, the scope of the memory to track must be minimized. Existing container checkpointing mechanisms do not consider consecutive checkpoints; consequently, each time a checkpoint is created, the entire memory area of the container's process is saved to storage. Experimental results show that more than 78% of the memory area was duplicate, when the memory of multiple containers, in which the same services are operated was dumped 100 times with 2 s cycle. As the purpose of *iContainer* is to continuously record checkpoints, storing duplicate memory is unnecessary. We devised a method to track and record the change information of the memory without duplication removal operations. We seek to track and record the change information of the memory because the memory has spatiotemporal locality. By focusing exclusively on the memory area that changes rather than the entire memory area, we solve the problem of checkpoint performance delay caused by duplicate memory storage. Moreover, by simplifying the checkpointing operation, it becomes possible to minimize the memory potentially lost during the checkpointing process. Fig. 4 shows the detailed process of tracking and recording the memory change information of the container. ① *iContainer* generates the container memory table by analyzing the container memory allocation information to map the container memory, which is the checkpoint target, to the host memory. ② To effectively track changes in the memory of the process, the write permission for the memory area is protected. The first checkpointing operation freezes the container and stores the entire memory area of the container. The data generated here is $C_f$. A process to generate $C_f$ is necessary because it is used as seed data to process the partially saved checkpoints into complete checkpoints. When $C_f$ creation is completed, it disables the write operation of the container with reference to the container memory table. ③ When the write permission is removed from the memory area that is allocated to and used by a container, a page fault occurs with force whenever a memory change occurs. The page fault is tracked and the data and position information that were used as write arguments are intercepted and stored in a temporary checkpoint buffer. Subsequently, changes in the information corresponding to the page fault are updated in the container memory. Saving only the memory area where the write operation occurred through page fault tracking has the same effect as removing duplication between the previous memory and current point. The proposed framework is used as a checkpoint buffer area by allocating part of the host memory. A performance delay in checkpoint operations in a hot zone is caused by the bandwidth of the memory and file system. Therefore, if the in-memory area is used as a dedicated area for checkpointing, the read/write throughput increases and this is advantageous for performance in container checkpoint and restoration operations. ④ If the temporary checkpoint buffer becomes full or the condition specified by the user is satisfied, the changed memory information is generated in the storage as one checkpoint. The

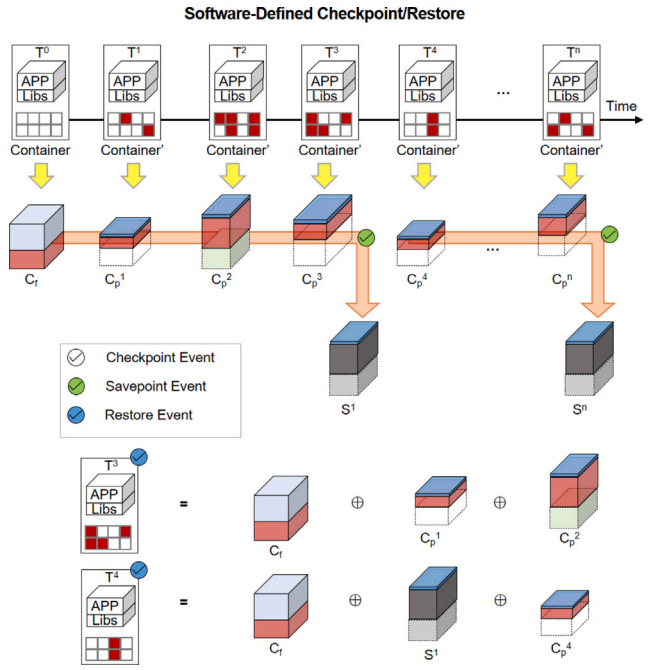


**Fig. 5.** Save point for creating partial and complete checkpoints for checkpoint acceleration.

data generated here is $C_p$. This process is the same as tracking only the changed memory between the previous and current memory and saving it to storage. However, because the $C_p$ created here contains only the memory offset and raw data where the write operation occurred, it cannot be used for rollback. The solution to this is discussed in the next section.

### 3.4. Software-defined checkpoint/restore

Our goal is to return to a state where stable services can be provided by rolling back to the checkpointed optimal point when the container service cannot be operated normally. Therefore, when a situation in which a container cannot provide stable services is detected, it must be quickly restored to the optimal point. In this section, we present the design of the software-defined checkpoint/restore (SDCR) scheme that efficiently manages and utilizes checkpoints. The SDCR Scheme has the following two features. The first feature is management of the collected checkpoints. Rollback through checkpoint generates a large amount data over time because it stores and manages the checkpoint file in the file system. Hence, effective management of checkpoints is a problem that must be solved in terms of storage cost. We combined libraries, tracked changed memories, and partially recorded them to remove duplicate checkpoint data. The proposed method provides a considerable advantage in terms of storage cost because most duplicate data can be removed. However, complete checkpoint data are required to rollback through checkpoints when a service error occurs. Fig. 5 shows the process of generating checkpoints through SDCR. In our proposed framework, partial checkpoints are created only for the changed container data. The data created here cannot be immediately used for rollback. As such, a post-processing operation is needed to recreate the complete checkpoint required for rollback. To generate the checkpoint used for rollback, $\oplus$ is sequentially calculated from point $C_f$, which has the form of a complete checkpoint, to the user-specified point $C_p$. This design quickly converts the previous checkpoint to a complete checkpoint when stable service recovery through the checkpoint is created immediately before the service error. The second feature is the guarantee of random-access performance for checkpoint restoration. *iContainer* must perform the $\oplus$ operation to create a complete checkpoint that can be rolled back. If the point where the service error is detected is far from the point of the stable checkpoint, then many $\oplus$ operations must be performed. The restoration time linearly increases in proportion to the number and size of checkpoints for which the $\oplus$ operation must be performed. Furthermore, the time it takes to create a complete checkpoint significantly varies when the user randomly selects a restoration checkpoint. We introduce the save point concept to solve this problem. Fig. 5 shows the process of creating a save point through SDCR. SDCR can freely specify the save point creation condition by user rules. For example, when the user sets the maximum time required for restoration, the restoration time is inferred by calculating the sizes of the checkpoint and save points are generated between checkpoints that exceed the time specified by the user. Another example is the generation of save points at the time a file system event occurs. Most of the situations where stable services cannot operate normally occur when the file system is changed by code generation, removal, or modification by an administrator or infection by malware; these are highly likely to be a restore point. Because save points are complete checkpoint data of specific points, the $\oplus$ operation must be performed only for the checkpoints after the generation of a save point. Therefore, the time required for checkpoint restoration is reduced.

### 3.5. Seamless recovery

Availability is a prerequisite that must be guaranteed by every computing system and service. Specifically, availability must be ensured more in a distributed service environment in which many containers operate organically. For example, if one container does not work normally, it can have an adverse effect on all the other containers. Therefore, availability must be guaranteed even if a service failure occurs. We devised a method of buffering the packets generated after a checkpoint in a queue to respond to requests that entered in a situation where a service failure has occurred after self-recovery. Service requests from clients can occur during a service failure or self-recovery. To prevent loss of the packets requested in a situation where stable service is impossible, it is designed to buffer the packets of incoming traffic in the queue through the netfilter-NFQUEUE library. When the NFQUEUE is used, the traffic entering the host can be stored in the packet queue by the packet filter and requested again. The scope of network packets stored in the queue comprises all incoming packets to the service port after the last checkpoint. In this process, packets for which the user or service requests were processed are excluded from the queue. Thus, the only packets remaining in the queue are those for which a service processing request was received but the user or service was not responded to when the checkpoint was created. This design is used because a service error may occur while the service requested by the user is being processed. It is also used to process service requests after service recovery for all packets requested when stable service could not be provided. However, all methods that do not require a response are stored in the queue. To solve this problem, the user must manage the methods that do not require a response using a whitelist. However, for methods recorded in the whitelist, only packets requested when stable service was unavailable can be processed. We filter packets based on checkpoints because we cannot predict the point of service failure. *iContainer* ensures seamless service by retransmitting the packets buffered in the queue to the target container service after the container service is restored through rollback. However, one structural limitation is that normal processing cannot be performed if the service rollback time exceeds the user-specified timeout period. This limitation can be addressed by configuring the save point creation condition of SDCR to be less than the timeout period.

**Algorithm 1** Performing a tracing-based checkpoint

**Input:** Request for container checkpoint
1: $C_f \leftarrow Checkpoint\ in\ Complete\ File$
2: $C_p \leftarrow Checkpoint\ in\ Partial\ File$
3: $M \leftarrow A\ file\ containing\ the\ location\ of\ the\ trace\ information$
4: $Init\_Flag \leftarrow False$
5: **if** $Init\_Flag = False$ **then**
6: Create a $C_f$
7: $Init\_Flag \leftarrow True$
8: **end if**
9: **for** $iteration = 1, 2, \ldots$ **do**
10: **if** $\Delta Page \parallel \Delta Filesystem$ **then**
11: Create a ${C_p}^n = 1, 2, \ldots, N$
12: Create a $M^n = 1, 2, \ldots, N$
13: **end if**
14: **end for**
**Output:** $C_f \| {C_p}^n, M^n$

### 3.6. Algorithms of iContainer

To accelerate checkpoints, the proposed *iContainer* creates incomplete checkpoints through hot/cold classification. Additionally, we propose a method to restore incomplete checkpoints to complete checkpoints. This section explains the checkpoint and rapid checkpoint restoration algorithm used by *iContainer*.

• Event trigger-based checkpoint: *iContainer* creates partial checkpoints by separating hot and cold zones and tracking the memory, which is a hot zone. Algorithm 1 shows the checkpointing process. *iContainer* performs checkpointing for all zones, be it hot or cold, when the first checkpoint creation request by a user or an event is received. $C_f$ is generated through this process. Subsequently, the background thread monitors the hot and cold zones of the container. If a file system event is detected in a cold zone, the event entity and offset information are recorded in $M$ and the $C_p$ is generated by inserting data into it. At this point, $C_p$ is not a complete checkpoint; however, it stores partial data. If a dirty page is detected by the memory tracker in a hot zone, the event entity and offset information is recorded in metadata and data are input to $C_p$. If $C_p$ is generated whenever a dirty page occurs, numerous metadata and $C_p$ are generated. This causes file I/O overhead and difficulty in checkpoint management. Therefore, $C_p$ is generated only according to a user-specified cycle or when a user-defined trigger occurs.

• Rapid checkpoint restoration algorithm: *iContainer* creates incomplete checkpoints to accelerate the checkpointing speed. Because the proposed method performs partial checkpoints, it is impossible to perform rollback using one checkpoint. Therefore, a process for creating a new checkpoint using complete and incomplete checkpoints is required. Algorithm 2 shows the basic checkpoint restoration method. When the user selects a random $N$, a complete checkpoint is created through the init checkpoint and $\oplus$ processes. At this point, $M$ is used for the offset information for which $\oplus$ must be performed and $C_p$ is used for the data. $C_f$ is created through this process. Subsequently, the $C_f$ at the random point $N$ selected by a user is obtained by repeatedly performing the $C_f \oplus C_p$ operations. However, such checkpoint restoration operations increase in proportion to the $\oplus$ operation as the $N$ point becomes farther from the init point; this exponentially increases the time required for checkpoint restoration. The checkpoint restoration time issue is explained through an experiment in Section 4.4. The problem is that the service error rollback time increases with the checkpoint restoration time. We propose save point to solve this problem.

A save point is a complete checkpoint that is inserted between incomplete checkpoints to accelerate the recovery speed of checkpointing. The save points are generated adaptively through user-defined rules. Algorithm 3 shows the process for restoring a checkpoint using a save point. When the user requests the creation of a random ${C_f}^n$, SDCR selects a save point that is closest before $N$ among the generated save points. The last $C_p$ of $C_s$ is indicated as $S$. If a random $N$ is at the same position as $S$, the restoration process is unnecessary. However, if $N$ is larger than S, the $C_f$ at point $N$ is acquired by repeatedly performing the $C_f \oplus C_p$ operation until the point $N$. This design solves the problem of exponentially increasing the checkpoint execution time. However, the indiscriminate save point generation condition wastes the storage space and intermittent save point generation increases the container service error restoration time. In this study, an experiment is conducted for a case study that uses the maximum checkpoint restoration time for the save point creation condition. The experiment on the save point creation is explained in Section 4.4.

**Algorithm 2** Process for restoring an incomplete checkpoint

**Input:** Select $C_pN$
1: $C_f \leftarrow Checkpoint\ in\ Complete\ File$
2: $C_i \leftarrow Initial\ Checkpoint\ File$
3: $C_p \leftarrow Checkpoint\ in\ Partial\ File$
4: $M \leftarrow A\ file\ containing\ the\ location\ of\ the\ trace\ information$
5: $Offset \leftarrow Address\ value\ for\ trace\ data$
6: **for** ${C_p}^n = 1, 2, \ldots, N$ **do**
7: **if** $N = 1$ **then**
8: **for all** $M^n$ **do**
9: $C_f \leftarrow C_i[Offset]\ \delta\ {C_p}^1[Offset]$
10: **end for**
11: **end if**
12: **if** $N$ **then**
13: **for all** $M^n$ **do**
14: $C_f \leftarrow C_f[Offset]\ \delta\ {C_p}^n[Offset]$
15: **end for**
16: **end if**
17: **end for**
**Output:** $C_f$

**Algorithm 3** Using save point to restore an incomplete checkpoint

**Input:** Select $C_pN$
1: $C_s \leftarrow Nearest\ Complete\ Checkpoint\ Before\ N\ (Savepoint)$
2: $C_f \leftarrow Checkpoint\ in\ Complete\ File$
3: $C_p \leftarrow Checkpoint\ in\ Partial\ File$
4: $S \leftarrow N\ After\ C_s$
5: $M \leftarrow A\ file\ containing\ the\ location\ of\ the\ trace\ information$
6: $Offset \leftarrow Address\ value\ for\ trace\ data$
7: **for** ${C_p}^n = 1, 2, \ldots, N$ **do**
8: **if** $N = S$ **then**
9: $C_f \leftarrow C_s$
10: **end if**
11: **if** $S < N$ **then**
12: **for all** $M^n$ **do**
13: $C_f \leftarrow C_f[Offset]\ \delta\ {C_p}^n[Offset]$
14: **end for**
15: **end if**
16: **end for**
**Output:** $C_f$

## 4. Evaluation

The three most important elements of the proposed system are the time required for checkpointing, data size, and time it takes to restore the stable state from a service unavailable state. We evaluated the usability of *iContainer* through three experiments, in which we

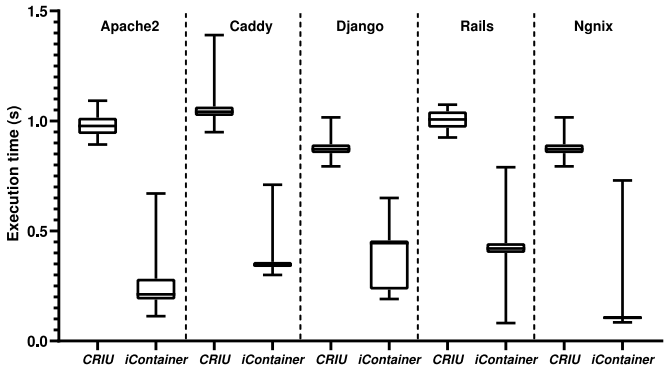


**Fig. 6.** Comparison measurement of checkpoint execution time (CRIU vs. *iContainer*).

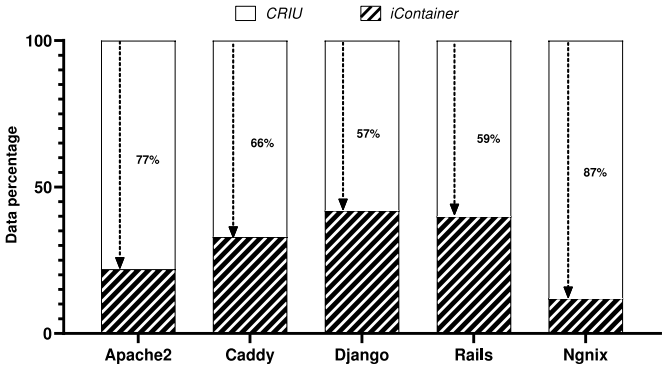


**Fig. 7.** Comparison of data size generated by checkpoints (CRIU vs. *iContainer*).

compared it with CRIU. We compared *iContainer* with CRIU because the latter will be integrated into the checkpoint function of Docker Engine. Furthermore, *iContainer* also runs in the Docker Engine environment. First, we measured the time it takes for checkpoint. To measure the potential downtime and performance overhead caused by checkpoints, an apache2, caddy, django, rails, nginx web service environment was built through an Ubuntu image uploaded to Docker Hub (apache, 1995; ZeroSSL, 2014; Hansson, 2004; Sysoev, 2004). An Ubuntu image was used in order to consistently build all web services in the form of packages. As the checkpointing operation processes the container at the process level, all container images can be used. We performed a stress test using the apache2 benchmark installed on the web server. For comparison of CRIU and *iContainer*, we generated 1000 checkpoints in 1 s cycles during the stress test. We measured the average checkpointing time during the generation of 1000 checkpoints. Second, we compared the size of the data generated by the container checkpoints. CRIU checkpoints the entire container, whereas *iContainer* checkpoints only the area that has changed after the first checkpoint was created. For comparison of CRIU and *iContainer*, we generated 1000 checkpoints in 5 s cycles during the stress test. Subsequently, we measured the storage space consumed by the generation of 1000 checkpoints. Third, we measured the time required for service recovery. The proposed framework creates partial checkpoints to reduce the checkpointing time and the size of the generated data. However, checkpoint restoration is required for rollback because partial checkpoints are not complete checkpoints. In this experiment, the restoration times for checkpoints created through CRIU and SDCR were measured and compared.

### 4.1. Experimental environments

The experimental setup for this study was as follows. Intel(R) Xeon(R) Silver 4114 CPU @ 2.20 GHz with 20 Cores, 384 MB DDR RAM, and 1 TB SSD storage were used. Ubuntu 16.04 (kernel 5.3.0-59) was the host OS, and Docker Engine version 18.09.7 was used. Ubuntu was used as the base image for the containers. The maximum memory size used by the containers was limited to 128 MB.

### 4.2. Comparison of checkpoint execution time

In this experiment, the time it takes for checkpoint creation was measured. We performed checkpointing in 1000 ms cycles for the web service container that was built using *iContainer*. A memory of 128 MB was allocated to the web service container. To demonstrate the execution time reduced by the proposed service recovery-friendly checkpoints, the result was compared with that of CRIU. Fig. 6 shows the experimental results. The $x$-axis represents the checkpoint scheme and workload and the $y$-axis represents the checkpoint execution time. For CRIU, it took 950 ms on average for checkpoint creation. For *iContainer*, the first checkpoint took 712 ms on average; however, it decreased gradually from the next checkpoint and converged to 290 ms on average. This indicates that the checkpoint execution time was reduced by more than 3.27 times compared to CRIU. Two reasons account for the reduction in the checkpoint execution time. The first is the change in the checkpoint structure. CRIU was designed for single checkpointing. Therefore, when a checkpoint is requested, the entire area of the target process is checkpointed. Thus, duplicate operations data are included because the entire area of the process is checkpointed repeatedly whenever a checkpoint is requested for the same target process. By contrast, the proposed framework was designed with consideration for repetitive checkpoints. Therefore, the first checkpoint is performed for the entire area, excluding the library area of the process; however, the ensuing checkpoints are created only with the changed data, thereby minimizing duplicate operations and data. The second reason is the in-memory checkpoint buffer. Most checkpointing operations consist of read–write repetitions. Therefore, when checkpoints are generated in a file system such as HDD and SSD, the checkpointing time increases owing to the difference in bandwidth between the memory and file system. To minimize such a bandwidth difference, we used some memory from the host as a buffer area for checkpoints.

### 4.3. Comparison of data size generated by checkpoints

In this experiment, we measured the size of the data generated in storage for container checkpoints created by CRIU and *iContainer*. CRIU checkpoints and stores the entire container, whereas *iContainer* stores only the area that has changed after the first complete checkpoint. We built five web-server environments (Apache2, Caddy, Django, Redis, Nginx) to reproduce the environment in which services are executed. The maximum size of memory that could be used by a web-server container was limited to 128 MB. For the workload of the server, the Apache2 benchmark was used. In this experiment, 1000 checkpoints were generated in 5000 ms cycles. Fig. 7 shows the experimental results. The *iContainer* saved the cumulative data size of the generated checkpoints by 69.2% on average compared to CRIU. Nginx showed a storage reduction effect of up to 87%. This proved that the separation of hot and cold zones and tracking-based checkpoints designed in *iContainer* have a storage reduction effect. In this experiment, only one web server was operated for checkpoints; however, the storage reduction effect will increase if multiple containers are operated. The difference in checkpoint capacity is caused by the removal of duplicate data. CRIU was designed for a single checkpoint, but *iContainer* was designed to remove duplication from repetitive checkpoints.

### 4.4. Comparison of time taken to restore checkpoint

In this experiment, the time taken to restore the stable service when the service stopped owing to an error was measured. The *iContainer* uses part of the memory of the host OS as a checkpoint buffer for fast

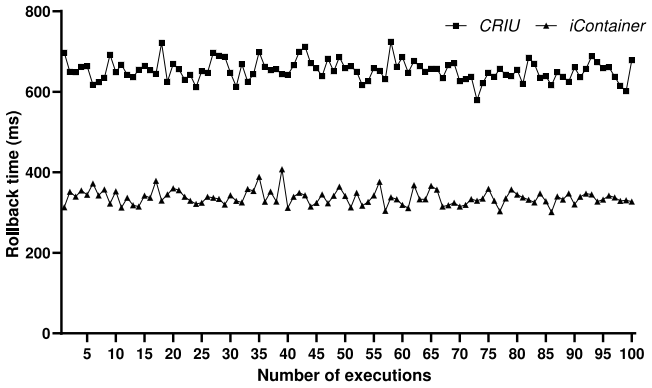


**Fig. 8.** Comparison of checkpoint rollback times.

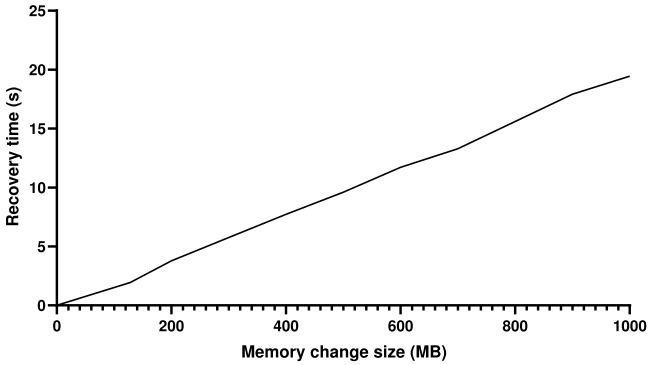


**Fig. 9.** Measurement of time taken to restore checkpoints according to capacity.

rollback in the event of a container recovery request. First, the rollback time is compared with CRIU in Fig. 8. This experiment used one container that has a memory of 128 MB and runs the apache2 server. CRIU took 653 ms on average to perform rollback, whereas the SDCR took 337 ms on average. The experimental results showed that the rollback performance of the proposed framework was approximately 1.93 times faster than that of CRIU. The rollback time was reduced because most of the containers in operation have duplicate cold zones and the data and operations to be rolled back decreased.

To minimize storage costs and improve checkpointing speed, *iContainer* creates partial checkpoints by tracking and saving only the change information of the container area. This structure requires an additional process called checkpoint restoration to use checkpoints for rollback. We measured the time taken to restore checkpoints. The $\oplus$ operation must be performed to create complete checkpoints using partial checkpoints. Thus, the problem of increasing $\oplus$ operations with the accumulation of checkpoints may occur. Fig. 9 shows the measurement of the time taken for checkpoint restoration as the $\oplus$ operations increase. The $x$-axis represents the capacity required for operation and $y$-axis represents the time required for checkpoint restoration. The experimental results show that, as the checkpoints accumulated, the time required for restoration increased in proportion. Therefore, when checkpoints are accumulated, immediate rollback becomes impossible in the event of a service error.

To solve the problem above, we introduced the save point concept. Save point reduces the rollback time by inserting complete checkpoints at specific points defined by the user. The criteria for generating save points is freely defined by the user. In this experiment, when the user sets the maximum time required for checkpoint restoration, SDCR calculates the cumulative size of checkpoints, infers the required restoration time, and generates save points in the background. Fig. 10 shows the experimental results. The $x$-axis represents the number of

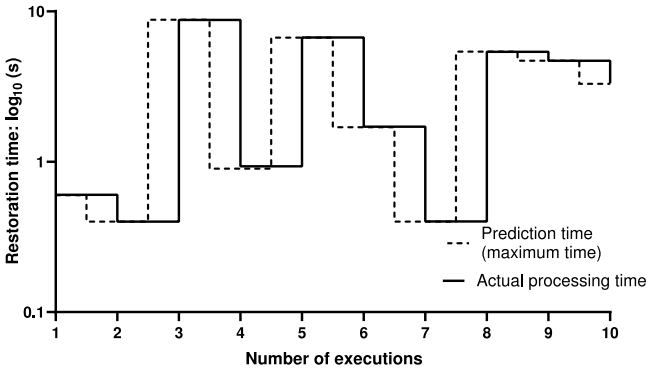


**Fig. 10.** Measurement of checkpoint restoration time through save points.

times and $y$-axis represents the maximum restoration time and actual time taken for restoration. We verified if the maximum restoration time could be guaranteed through save points when it was set to a random value between 0.1 s and 10 s. The experimental results proved that the save points guaranteed the maximum restoration time by 99% or more. The checkpoint execution time was not increased by the save points because they are separately generated in the background from checkpoint operations.

## 5. Usage scenario

In this section, the usage scenarios of the proposed *iContainer* are discussed.

### 5.1. Scenario 1: Checkpoint creation based on file system event

The criteria for creating checkpoints are crucial. Because the lost information varies by the checkpoint creation criteria, it is ideal to perform checkpoints in a fast cycle to minimize the information lost in the event of a service error. However, efficient criteria for checkpoint creation are important because unnecessary checkpoint creation can cause computational/spatial overheads. Most of the situations where stable services cannot operate normally are caused by a change in the file system such as code generation, removal, or modification by an administrator or infection with malware (Analytica, 2021a). Therefore, it is expedient to select the conditions for creating checkpoints based on the generation, removal, and modification events of files through the detection of file system events. *iContainer* is designed to efficiently monitor file system events using Inotify. Therefore, users can generate conditions to create checkpoints when a file system event occurs. However, users should be cautious because memory-resident malware may not generate a file system event.

### 5.2. Scenario 2: Checkpoint creation based on resource usage

If users desire to leave many parts of containers as checkpoints, they can set the checkpoint based on resource usage. The hot zone data can be lost for checkpoints that are performed based on a cycle or a file system event. To minimize data loss for hot zone, adaptive checkpoint creation based on resource usage is required. Adaptive checkpoint creation generates checkpoints based on the change of hot zone. Thus, a fine-grained checkpoint is created if the memory usage is large and a coarse checkpoint is created if the memory usage is small. This method is useful when closely monitoring the behavior of a container because active checkpoint creation can preserve data. However, if the container operates a memory-intensive service, a performance delay may be caused by numerous checkpoints and memory tracking.

### 5.3. Scenario 3: Service error detection and self-service rollback

The health check engine inside the *iContainer* can be used to determine the service state of the container in operation. The health check engine normally self-diagnoses the state of services through two tests. First, it verifies the operation through resource usage monitoring for a process inside the container. The container service uses a resource to process requested jobs when a request is received from the outside. If the resource usage of the container does not increase when a normal request is received, it is considered that the service does not work normally. Second, it generates the health API through requested packet monitoring. Health checker generates the health API that determines whether a service operates normally by monitoring the packet that enters the container. However, because the generated health API can cause service overload or affect the service, the generated health API does not have parameters and replicates the packet whose method is obtained by monitoring. An example of this method is the API that requests a server time. *iContainer* can quickly and efficiently determine the service state using the two tests above. The health API has different service interruption detection times depending on the request cycle. When we performed health checks with 1 s cycle, the service interruption could be detected within 1.01 s on average. Furthermore, the health check engine can be used for service error recovery and self-recovery because it can be used to verify whether the rollback point provides a stable service. However, detecting service errors only using the health check engine was left for a case study because it can cause false positive and false negative.

## 6. Conclusion

This study proposed *iContainer*, a framework that can quickly restore a cloud-based system to a point desired by the user and restore the stable service state when a failure occurs by continuously recording container services to secure service resilience. *iContainer* has three contributions. First, it minimized checkpoint operations through checkpoint zoning. It reduced the storage cost by removing duplicate checkpoints and reduced the checkpoint execution time. We classified the checkpoint zones of the container into hot and cold zones using the semantic-aware hot/cold container classification scheme. This scheme minimized the unnecessary checkpoint operations for the cold zone. Second, it minimized the time required for checkpoint operations. We performed duplication removal operations through memory tracking for the hot zone. Consequently, the checkpoint execution time of the *iContainer* became faster by approximately 3.27 times compared to that of CRIU and the storage cost was reduced by 69.2%. Finally, a case study was conducted for rapid checkpoint restoration and selection of checkpoint/restore points. We proposed the save point concept to improve the performance for additional checkpoint restoration operation, which is a limitation of *iContainer*. It was proven through an experiment that 99% or more of the checkpoint restoration time specified by the user can be guaranteed using save points. Consequently, it took 337 ms on average to restore the container to the same checkpoint in the event of a service error. Comparison with the conventional checkpoint tool CRIU shows that the restoration time increased by approximately 1.93 times. Moreover, the user can freely select the checkpoints and restore points through SDCR. The *iContainer* proposed in this study was designed for services with small memory operations, such as web services. Therefore, the checkpoint time may be delayed and the operation performance of the container may decrease in services that require memory-intensive operations such as database, machine learning, and storage services. This limitation is common to all checkpointing-based restoration solutions. It can be solved through checkpoint creation by user intervention or long-term checkpoint creation. *iContainer* can be applied as a basic technology in forensics, introspection, and abnormal behavior detection because it stores the lifecycles of containers as checkpoints, which can be used as data to identify the cause of attacks or security accidents. Furthermore, the proposed framework can be used in a variety of areas because it can freely specify checkpoints and restore points according to user requirements. In future work, we will conduct studies to solve the execution time delay and service interruption problems in checkpoint creation for memory-intensive containers and apply efficient checkpointing to the security field.

## CRediT authorship contribution statement

**Sang-Hoon Choi:** Conceptualization, Methodology/Study design, Software, Validation, Formal analysis, Investigation, Data curation, Writing – original draft, Writing – review & editing, Visualization. **Ki-Woong Park:** Conceptualization, Methodology/Study design, Validation, Formal analysis, Investigation, Resources, Writing – original draft, Writing – review & editing, Visualization, Supervision, Project administration, Funding acquisition.

## Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

## Data availability

No data was used for the research described in the article.

## Acknowledgments

This work was supported by the Institute of Information & Communications Technology Planning & Evaluation (IITP) , South Korea(Project No. RS-2022-00165794, Development of a Multi-Faceted Collection-Analysis-Response Platform for Proactive Response to Ransomware Incidents, 30%, and Project No. 2019-0-00426, 10%), the ICT R&D Program of MSIT/IITP, South Korea (Project No. 2021-0-01816, A Research on Core Technology of Autonomous Twins for Metaverse, South Korea, 10%), and a National Research Foundation of Korea (NRF), South Korea grant funded by the Korean government (Project No. NRF-2020R1A2C4002737, 50%).

## References

Ahmed, A., Mohan, A., Cooperman, G., Pierre, G., 2020. Docker container deployment in distributed fog infrastructures with checkpoint/restart. In: 2020 8th IEEE International Conference on Mobile Cloud Computing, Services, and Engineering. MobileCloud, IEEE, pp. 55–62.

Alliance, C.S., 2021. Top threats to cloud computing: Egregious eleven deep div. URL https://cloudsecurityalliance.org.

Amoon, M., El-Bahnasawy, N., Sadi, S., Wagdi, M., 2019. On the design of reactive approach with flexible checkpoint interval to tolerate faults in cloud computing systems. J. Ambient Intell. Humaniz. Comput. 10 (11), 4567–4577.

Analytica, O., 2021a. Efforts to curb ransomware crimes face limits. Emerald Expert Brief. (oxan-db).

Analytica, O., 2021b. US pipeline hack signals critical infrastructure risks. Emerald Expert Brief. (oxan-es).

apache, 1995. The apache HTTP server project. URL https://httpd.apache.org.

Bamiah, M.A., Brohi, S.N., 2011. Seven deadly threats and vulnerabilities in cloud computing. Int. J. Adv. Eng. Sci. Technol. 9 (1), 87–90.

Bernstein, D., 2014. Containers and cloud: From lxc to docker to kubernetes. IEEE Cloud Comput. 1 (3), 81–84.

Chen, Y., 2015. Checkpoint and restore of micro-service in docker containers. In: 2015 3rd International Conference on Mechatronics and Industrial Informatics (ICMII 2015). Atlantis Press, pp. 915–918.

Chung, M.T., Quang-Hung, N., Nguyen, M.-T., Thoai, N., 2016. Using docker in high performance computing applications. In: 2016 IEEE Sixth International Conference on Communications and Electronics. ICCE, IEEE, pp. 52–57.

Cloud, A.E.C., 2011. Amazon web services. Retrieved November 9 (2011), 2011.

Copeland, M., Soh, J., Puca, A., Manning, M., Gollob, D., 2015. Microsoft Azure. Springer, New York, NY, USA:: Apress.

Desai, A., Oza, R., Sharma, P., Patel, B., 2013. Hypervisor: A survey on concepts and taxonomy. Int. J. Innov. Technol. Explor. Eng. 2 (3), 222–225.

Dua, R., Kohli, V., Patil, S., Patil, S., 2016. Performance analysis of union and cow file systems with docker. In: 2016 International Conference on Computing, Analytics and Security Trends. CAST, IEEE, pp. 550–555.

Dutta, P., Dutta, P., 2019. Comparative study of cloud services offered by Amazon, Microsoft & Google. Int. J. Trend Sci. Res. Dev. 3 (3), 981–985. http://dx.doi.org/10.1371/journal.pcbi.1002503.

Elazhary, H., 2019. Internet of things (IoT), mobile cloud, cloudlet, mobile IoT, IoT cloud, fog, mobile edge, and edge emerging computing paradigms: Disambiguation and research directions. J. Netw. Comput. Appl. 128, 105–140.

Elbadawi, K., Al-Shaer, E., 2009. TimeVM: A framework for online intrusion mitigation and fast recovery using multi-time-lag traffic replay. In: Proceedings of the 4th International Symposium on Information, Computer, and Communications Security. ASIACCS '09, Association for Computing Machinery, New York, NY, USA, pp. 135–145.

Garraghan, P., Townend, P., Xu, J., 2013. An analysis of the server characteristics and resource utilization in google cloud. In: 2013 IEEE International Conference on Cloud Engineering. IC2E, IEEE, pp. 124–131.

Goel, A., Po, K., Farhadi, K., Li, Z., de Lara, E., 2005. The taser intrusion recovery system. In: Proceedings of the Twentieth ACM Symposium on Operating Systems Principles. SOSP '05, Association for Computing Machinery, New York, NY, USA, pp. 163–176.

Goiri, Í., Julia, F., Guitart, J., Torres, J., 2010. Checkpoint-based fault-tolerant infrastructure for virtualized service providers. In: 2010 IEEE Network Operations and Management Symposium. NOMS 2010, IEEE, pp. 455–462.

Hansson, D.H., 2004. Ruby on rails. URL https://rubyonrails.org.

Hu, B., Lei, Z., Lei, Y., Xu, D., Li, J., 2011. A time-series based precopy approach for live migration of virtual machines. In: 2011 IEEE 17th International Conference on Parallel and Distributed Systems. IEEE, pp. 947–952.

Hyseni, L.N., Ibrahimi, A., 2017. Comparison of the cloud computing platforms provided by Amazon and Google. In: 2017 Computing Conference. IEEE, pp. 236–243.

Jiang, C., Wang, Y., Ou, D., Li, Y., Zhang, J., Wan, J., Luo, B., Shi, W., 2019. Energy efficiency comparison of hypervisors. Sustain. Comput.: Inform. Syst. 22, 311–321.

Joshi, B., Vijayan, A.S., Joshi, B.K., 2012. Securing cloud computing environment against DDoS attacks. In: 2012 International Conference on Computer Communication and Informatics. IEEE, pp. 1–5.

Karhula, P., Janak, J., Schulzrinne, H., 2019. Checkpointing and migration of IoT edge functions. In: Proceedings of the 2nd International Workshop on Edge Systems, Analytics and Networking. pp. 60–65.

Kavis, M.J., 2014. Architecting the Cloud: Design Decisions for Cloud Computing Service Models (SaaS, PaaS, and IaaS). John Wiley & Sons.

Khan, M.A., 2016. A survey of security issues for cloud computing. J. Netw. Comput. Appl. 71, 11–29.

Khan, A.A., Zakarya, M., Khan, R., Rahman, I.U., Khan, M., ur Rehman Khan, A., 2020. An energy, performance efficient resource consolidation scheme for heterogeneous cloud datacenters. J. Netw. Comput. Appl. 150, 102497.

Kumar, R., Goyal, R., 2021. Top threats to cloud: A three-dimensional model of cloud security assurance. In: Computer Networks and Inventive Communication Technologies. Springer, pp. 683–705.

Laadan, O., Hallyn, S.E., 2010. Linux-CR: Transparent application checkpoint-restart in Linux. In: Linux Symposium, vol. 159, Citeseer.

Mao, B., Jiang, H., Wu, S., Tian, L., 2015. Leveraging data deduplication to improve the performance of primary storage systems in the cloud. IEEE Trans. Comput. 65 (6), 1775–1788.

Matos, D.R., Pardal, M.L., Carle, G., Correia, M., 2018. Rockfs: Cloud-backed file system resilience to client-side attacks. In: Proceedings of the 19th International Middleware Conference. pp. 107–119.

Matos, D., Pardal, M., Correia, M., 2021. Sanare: Pluggable intrusion recovery for web applications. IEEE Trans. Dependable Secure Comput..

Mirkin, A., Kuznetsov, A., Kolyshkin, K., 2008. Containers checkpointing and live migration. In: Proceedings of the Linux Symposium, vol. 2, pp. 85–90.

Mudassar, M., Zhai, Y., Lejian, L., 2022. Adaptive fault-tolerant strategy for latency aware IoT application executing in edge computing environment. IEEE Internet Things J..

Pagani, F., Fedorov, O., Balzarotti, D., 2019. Introducing the temporal dimension to memory forensics. ACM Trans. Priv. Secur. 22 (2), 1–21.

Pahl, C., 2015. Containerization and the paas cloud. IEEE Cloud Comput. 2 (3), 24–31.

Passerini, E., Paleari, R., Martignoni, L., 2009. How good are malware detectors at remediating infected systems? In: International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment. Springer, pp. 21–37.

Pecchia, A., Cotroneo, D., Kalbarczyk, Z., Iyer, R.K., 2011. Improving log-based field failure data analysis of multi-node computing systems. In: 2011 IEEE/IFIP 41st International Conference on Dependable Systems Networks. DSN, pp. 97–108.

Pickartz, S., Eiling, N., Lankes, S., Razik, L., Monti, A., 2016. Migrating linux containers using CRIU. In: International Conference on High Performance Computing. Springer, pp. 674–684.

Pierleoni, P., Concetti, R., Belli, A., Palma, L., 2019. Amazon, Google and Microsoft solutions for IoT: Architectures and a performance comparison. IEEE Access 8, 5455–5470.

Putri, R.A., Winamo, I., Yuwono, W., Utomo, A.P., 2020. Implementation of resilience as a service for parallel computing. In: 2020 International Electronics Symposium. IES, IEEE, pp. 626–630.

Rajan, R.A.P., 2018. Serverless architecture-a revolution in cloud computing. In: 2018 Tenth International Conference on Advanced Computing. ICoAC, IEEE, pp. 88–93.

Ruan, B., Huang, H., Wu, S., Jin, H., 2016. A performance study of containers in cloud environment. In: Asia-Pacific Services Computing Conference. Springer, pp. 343–356.

Singh, A., Chatterjee, K., 2017. Cloud security issues and challenges: A survey. J. Netw. Comput. Appl. 79, 88–115.

Stoyanov, R., Kollingbaum, M.J., 2018. Efficient live migration of linux containers. In: International Conference on High Performance Computing. Springer, pp. 184–193.

Sun, P., 2020. Security and privacy protection in cloud computing: Discussions and challenges. J. Netw. Comput. Appl. 160, 102642.

Sysoev, I., 2004. Nginx. URL https://www.nginx.com.

Treaster, M., 2005. A survey of fault-tolerance and fault-recovery techniques in parallel systems. arXiv Preprint cs/0501002.

Van Eyk, E., Iosup, A., Seif, S., Thömmes, M., 2017. The SPEC cloud group's research vision on FaaS and serverless architectures. In: Proceedings of the 2nd International Workshop on Serverless Computing. pp. 1–4.

Venkatesh, R.S., Smejkal, T., Milojicic, D.S., Gavrilovska, A., 2019a. Fast in-memory CRIU for docker containers. In: Proceedings of the International Symposium on Memory Systems. MEMSYS '19, Association for Computing Machinery, New York, NY, USA, pp. 53–65.

Venkatesh, R.S., Smejkal, T., Milojicic, D.S., Gavrilovska, A., 2019b. Fast in-memory CRIU for docker containers. In: Proceedings of the International Symposium on Memory Systems. pp. 53–65.

Virtuozzo, 2011. Checkpoint/restore in userspace. URL http://criu.org.

Webster, A., Eckenrod, R., Purtilo, J., 2018. Fast and service-preserving recovery from malware infections using CRIU. In: 27th USENIX Security Symposium. USENIX Security 18, USENIX Association, Baltimore, MD, pp. 1199–1211.

Widjajarto, A., Jacob, D.W., Lubis, M., 2021. Live migration using checkpoint and restore in userspace (CRIU): Usage analysis of network, memory and CPU. Bull. Electr. Eng. Inf. 10 (2), 837–847.

Yan, Q., Yu, F.R., Gong, Q., Li, J., 2015. Software-defined networking (SDN) and distributed denial of service (DDoS) attacks in cloud computing environments: A survey, some research issues, and challenges. IEEE Commun. Surv. Tutor. 18 (1), 602–622.

ZeroSSL, 2014. The ultimate server with automatic HTTPS. URL https://caddyserver.com.

Zheng, J., Ng, T.S.E., Sripanidkulchai, K., 2011. Workload-aware live storage migration for clouds. In: Proceedings of the 7th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments. pp. 133–144.

Zhu, J., Fang, X., Guo, Z., Niu, M.H., Cao, F., Yue, S., Liu, Q.Y., 2009. IBM cloud computing powering a smarter planet. In: IEEE International Conference on Cloud Computing. Springer, pp. 621–625.

**Sang-Hoon Choi** received the B.S. and M.S. degree in computer and information security from Daejeon University, South Korea, in 2016. He is currently working toward the Ph.D. degree in the Department of Computer and Information Security, University of Sejong, South Korea. His research interests include cloud computing, virtualization, system memory and machine learning.

**Ki-Woong Park** received the B.S. degree in computer science from Yonsei University, South Korea, in 2005, the M.S. degree in electrical engineering from the Korea Advanced Institute of Science and Technology (KAIST) in 2007, and the Ph.D. degree in electrical engineering from KAIST in 2012. He received a 2009–2010 Microsoft Graduate Research Fellowship. He worked for National Security Research Institute as a senior researcher. He has been a professor in the department of computer and information security at Sejong University. His research interests include security issues for cloud and mobile computing systems as well as the actual system implementation and subsequent evaluation in a real computing system.