Replayable Execution Optimized for Page Sharing for a Managed Runtime Environment

Kai-Ting Amy Wang
Huawei Canada Research Centre
kai.ting.wang@huawei.com

Rayson Ho
Huawei Canada Research Centre
Rayson.Ho@huawei.com

Peng Wu
Huawei American Research Centre
Peng.PengWu@huawei.com

Abstract

We present Replayable Execution, a system for improving the efficiency of Function-as-a-Service (FaaS) frameworks. It takes advantage of standard kernel features to reduce memory usage and accelerate cold startup speed without changes to the OS kernel, language runtimes, and the surrounding FaaS deployment environment. Replayable Execution exploits the intensive-deflated execution characteristics of the majority of target applications. It uses checkpointing to save an image of an application, allowing this image to be shared across containers and resulting in speedy restoration at service startup. We apply Replayable Execution to a representative FaaS Java framework to create a *ReplayableJVM execution*, which together with benefits from deterministic execution of a warmed up runtime, offers 2X memory footprint reduction, and over 10X startup time improvement.

CCS Concepts • **Computer systems organization** → **Cloud computing**;

Keywords Cloud Computing, Operating Systems, Programming Languages and Runtimes

ACM Reference Format:
Kai-Ting Amy Wang, Rayson Ho, and Peng Wu. 2019. Replayable Execution Optimized for Page Sharing for a Managed Runtime Environment. In *Fourteenth EuroSys Conference 2019 (EuroSys '19), March 25–28, 2019, Dresden, Germany*. ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3302424.3303978

1 Introduction

Function-as-a-Service (FaaS) [39] is an emerging programming and application deployment paradigm that is adopted by all major cloud providers [8, 15, 16, 19, 22, 23, 28, 30], where FaaS functions are deployed using Linux containers [20, 33] on computing resources. However, there are two performance problems that impact the economic bottom-line of FaaS [12]. The first is the bloat in the memory footprint and usage of system resources by containers. The second is the long start-up times, caused by the time to initialize a container for each FaaS function. Existing work [4, 40, 42, 43, 52, 60] attempts to address the problems by running multiple microservices in a single JVM, or by keeping a pool of warmed JVMs/containers alive for reuse [17, 33, 55]. They require either extensive modifications to the JVM, or non-trivial extensions to the container deployment environment.

In this paper, we address the two aforementioned problems with a novel approach called *Replayable Execution*. It is based on the observation that an application undergoes different phases during the lifetime of its execution. It first goes through a memory *intensive* framework initialization phase then followed by a *deflated* low memory usage phase where the application simply sits in an event loop listening to user requests. The phased execution characteristic is especially pronounced for applications that are built on complex frameworks.

Replayable Execution exploits the phased behaviour described above using a checkpoint and restore process. Live but cold data in the intensive phase is checkpointed and evicted from memory. The evicted data is only brought into memory on-demand during the low memory usage phase and is also made shared with other FaaS invocation instances. We do so in a way that requires no changes to the OS kernel, the language runtime or the surrounding FaaS environment. However, checkpointing and correctly restoring a unique FaaS instance in the non-privileged, non-root container environment, and taking into account practical cloud considerations, poses a set of unique challenges. We address these challenges in the context of an industrial FaaS framework that uses the Java Virtual Machine (JVM). We show that Replayable Execution leads to faster container startup time and a reduction in memory footprint.

We utilize Huawei's FaaS framework to implement Replayable Execution. Its Java framework currently uses Oracle HotSpot 64-Bit Server VM version 1.8.0_151 with a set of JVM flags optimized for memory footprint and FaaS style execution. We transform the JVM execution inside the framework into a Replayable Execution, which we call *ReplayableJVM*.

EuroSys '19, March 25–28, 2019, Dresden, Germany

ACM ISBN 978-1-4503-6281-8/19/03…$15.00
https://doi.org/10.1145/3302424.3303978

We experimentally evaluate the effectiveness of ReplayableJVM and demonstrate that it outperforms Oracle's Application Class Data Sharing technology (AppCDS) [6]. It achieves a 2x memory footprint reduction bringing the framework footprint from 49M to 25M, and a 10x start-up time reduction. Further, Replayable Execution brings down the best FaaS start-up time of 473ms using OracleJVM with AppCDS to only 54ms. We also experimentally show that the phased memory behaviour and the benefit of Replayable Execution extend to other FaaS runtime systems such as JavaScript and Python. We focus our evaluation on the JVM because of its popularity and also because it has the most significant overheads [59].

In summary, our work makes the following contributions:

- It introduces a novel technique called Replayable Execution that reduces FaaS start-up time and memory resource usage without changes to the OS kernel, language runtimes, and the surrounding FaaS deployment environment.
- It extends a popular checkpoint and restore implementation to restore memory pages, overcoming restrictions of restoring in a non-privileged, non-root container environment with techniques for PID translation, syscall replacement, and in-place restore, and addressing practical cloud deployment considerations.
- It experimentally demonstrates performance improvements, achieved with no modifications to the runtime, the OS nor the deployment mechanism, in the context of Huawei's JVM based FaaS framework. It also shows that the performance improvements also extend to Huawei's Python and JavaScript language runtimes with 2-4x startup and 1.16-1.4X memory reductions.

The remainder of the paper is organized as follows. Section 2 describes the working principles behind Replayable Execution. Section 3 describes transforming a workload into Replayable Execution. Section 4 describes our memory implementation and section 5 dives into the challenges of restoring inside a container. Section 6 elaborates on deployment challenges. We evaluate ReplayableJVM in Section 7. We survey the related work in Section 8 before we conclude.

2 Replayable Execution

This section details the working principles and challenges behind Replayable Execution.

Intensive-Deflated Phase Execution Characteristic To exploit the intensive-deflated phase execution characteristic, we explain two insights using Figure 1. First, at time t, during *Phase I*, the application holds a large working set [44] in its memory. Precisely speaking, the large amount of live data of the process is held in anonymous memory that has no backing store. When the application is in *Phase D* at time s, the working set drops drastically. Live data used in *Phase I* is no longer actively used in *Phase D*. Despite so, memory is still needed to hold on to the live data. Thus, the first insight

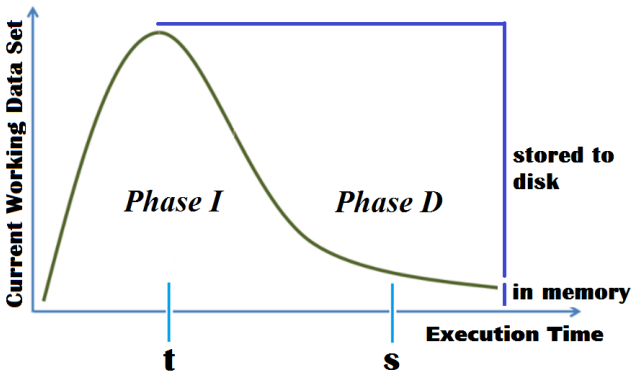

Figure 1. Application Phased Behaviour

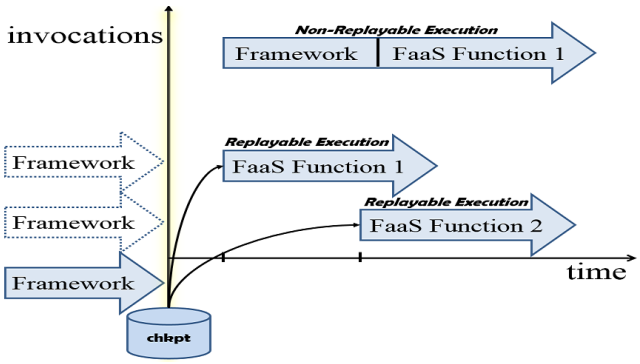

Figure 2. Jump starting of FaaS execution

is that we can store away these live data to a file, and only keep data that are in the current working set in memory. The live data that is stored on disk can be on-demand paged into the memory. This can result in a drastic reduction in memory usage by the application.

The second insight is that the live data is typically very similar between applications that share a lot of common codebase (e.g., applications built on top of the same framework such as `Spring Cloud/Boot` [9]). The live data is identical for running multiple copies of an application up to an identical execution point in time (e.g., time s). Consequently, storing live data to disk at time s in Figure 1 captures a common image. If we can *jump start* all the running programs on the same system using this common image, then the kernel can naturally share memory pages amongst all the running copies. Each copy, or process, will rely on the copy-on-write (COW) mechanics to grow its own working set from time s onward.

As shown in Figure 2, Replayable Execution is able to jump start FaaS function 1 and 2 skipping computation needed for framework initialization. The framework data is backed by a file and shared by the two invocations. Replayable Execution requires page sharing of the jump started processes, which we describe next.

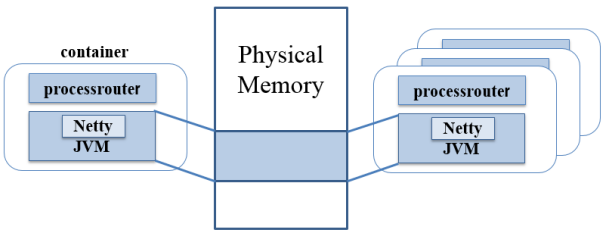

Figure 3. Memory page sharing

Private Anonymous Memory Sharing Across Containers Anonymous memory is private to a process; the kernel transparently makes the address space of a parent process share with the child in a COW fashion. However, the container execution model makes it hard for child processes created outside of a container to enter one. Replayable Execution makes anonymous memory shareable without modifying the Linux kernel, nor the container deployment model, nor using Kernel Same-page Merging (KSM). It achieves so by populating an address space using memory mapping a checkpoint file, and thus subsequent processes constructed this way would benefit from kernel's page sharing mechanism. Figure 3 depicts the page sharing concept implemented by Replayable Execution in a container environment for JVMs.

Replayable Execution Mechanism To realize the jump start and anonymous memory saving effect, we borrow an existing functionality well known in the HPC community: checkpoint and restore (C&R) [46]. Essentially, we identify an advantageous execution time point (e.g., *s* in Figure 1) and checkpoint the running process to a file. The checkpoint file thus contains all the needed information in order to reconstruct, or in other words, restore a single or multiple copies of the same process to jump start executing from the exact time point *s* onward.

The use of memory mapping also benefits from on-demand paging. This allows us to exploit our first insight described above and bring only a small amount of the working set at time *s* into physical memory.

Challenges There are two technical challenges that need to be overcome to make Replayable Execution possible and profitable. The first challenge is to ensure a workload stays in an ideal low working set phase for as long as possible. This is because if the restored process moves into a different memory phase quickly where it rewrites a large portion of its memory, the COW mechanism would have to allocate new private pages for the process, thus losing the memory saving. We solve this challenge by the judicious selection of a checkpoint location, described in Section 3. The second challenge is to ensure the anonymous memory sharing benefit remains for the jump started processes that share a common execution path up to time *s*, even if their execution paths diverge after *s*. We solve this challenge by handling the runtime divergence requirement correctly and carefully preventing the processes from altering the common codebase which may result in memory divergence that diminishes the saving.

3 Workload Transformation

We first provide evidence that phased execution is exhibited in our target FaaS framework. We then briefly describe the framework and then detail the changes needed to support Replayable Execution in it.

3.1 Phased Execution

To determine whether an application exhibits the intensive-deflated phase characteristic and thus can benefit from Replayable Execution, we develop two systematic methods to calculate the number of dirtied pages at each point of execution.

Our first tool injects a small utility function at different points of an application. The function forks a child process that blocks indefinitely using the `pause` `libc` call, while the parent continues execution. Since the child process shares the address space of the parent in a COW fashion, any dirtying of the shared address space causes new pages to be allocated. The decrease in the `Shared_Dirty` field in the `smaps` file of the child process in `procfs` indicates divergence of the working set size as execution continues since time of the fork. While this tool requires modification to the application, it gives the ability to pinpoint a precise location to begin the measurement. If one does not have the luxury to modify the application, the second tool can be useful.

Our second tool measures the number of pages dirtied during the different phases of a program execution by using the soft dirty bits exposed by the Linux kernel [5]. To calculate the number of pages dirtied between two time points during a process execution, the tool first clears all the soft dirty bits in the PTE (Page Table Entry) which makes the pages read-only at the first time point. As such, page faults are generated when the processor writes to the pages. At the second time point, the tool extracts the soft-dirty and present pages from the pagemap file and count the number of pages. The tool then computes a *deflate ratio* with respect to the process' peak resident set size (RSS) memory. In essence, if the tool cannot find points of execution with low deflate ratio, the application does not exhibit intensive-deflated phase execution characteristic.

With the tools, we determine that Huawei's Java FaaS framework exhibits intensive-deflated phase characteristic with a working set that is a fraction of the RSS as shown in Table 1.

Table 1. FaaS framework: pages dirtied after one `http` request

Total RSS	Shared_Dirty	soft-dirty size
95 MB	10 MB	10 MB

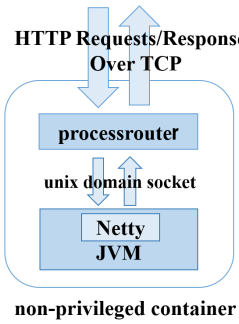

Figure 4. Netty based FaaS Framework

```
1  callbackMethodEntry()
2  {
3    // JVMTI_EVENT_METHOD_ENTRY callback
4    (*jvmti)->GetMethodName(&methodName, &methodSignature);
5    log(methodName, methodSignature);
6  }
7
8  Agent_OnLoad()
9  {
10   // At JVM start, register callback function
11   cb.MethodEntry = callbackMethodEntry;
12   (*jvmti)->SetEventCallbacks(jvmti, &cb, ...);
13 }
14
15 Agent_OnUnload(JavaVM *vm)
16 {
17   // At JVM exit, find method calls that lead to divergence
18   processLog(logfile);
19 }
```

Figure 5. Pseudo code for JVMTI tool to detect blacklisted functions

3.2 Introducing the FaaS Framework

In the FaaS programming paradigm, external events (such as REST requests from the web or writes to object storage) trigger the FaaS framework to on-demand deploy and launch user supplied functions to handle them. As an optimization to hide container startup latency, most FaaS platforms implement container reuse [18], i.e. after serving a request, the container is put in an idle state and as another request comes in, the container is reused to process it. Container reuse eliminates the entire overhead of bringing up a fresh JVM. However, aggressive container caching is not economical to the cloud provider as the idle containers are not billed but still occupy memory, and thus after a period of idle activity the containers are killed.

Huawei's Java FaaS Framework At the core of the framework, as shown in Figure 4, is a Netty[1] server communicating through unix domain socket [26] and a `processrouter` that acts as a request proxy fronting the JVM, both running inside a non-privileged Linux container. The `processrouter` accepts `http` requests from the internet and in turn, passes the requests to the JVM through a unix domain socket. Each FaaS invocation runs serially in its own container, and a container is reused if the next request arrives before the recycling of the idle container.

3.3 Checkpointing Locations in the FaaS Framework

By checkpointing the JVM at the very last stage of FaaS launch, Replayable Execution reduces the amount of computation remaining to serve a request after restore. Obviously it is too late to checkpoint the JVM after the user function is loaded, as it is not possible to know ahead of time which user function is used at checkpoint time. Further, loading of an external function is just one example of irreversible changes to the address space of the JVM. Similar to other Record and Replay implementations [56], we scan the application against a blacklist of functions that are known to introduce irreversible changes in the address space of a process, and thus checkpointing must be made before any of such calls. We detect occurrences of any blacklisted function using the JVMTI callback mechanism, shown in Figure 5, such that a callback is made when a blacklisted function is encountered. For the FaaS framework, we identify three possible irreversible places:

- Although the `accept` system call does not change the address space of the JVM process, it changes the kernel state as kernel resources are needed for establishing new TCP connections.
- The FaaS framework updates its internal bookkeeping data structure containing instance metadata uploaded by the cloud infrastructure (e.g., metadata such as the container's memory and CPU limits, ID information). This data structure is passed into the FaaS user function as a context argument.
- The `Class.forName` classloader loads the FaaS function into the address space of the JVM and triggers the JVM to perform `Loading`, `Linking` and `Initializing` steps required by class loading [54].

As `accept` is blocking, it is a natural location for checkpointing without the need to introduce other means of blocking or signalling into the framework. By ensuring no established connection exists during the time of checkpoint (e.g., no request in flight, `processrouter` persistent connection torn down), we prevent irreversible changes to JVM's address space and kernel state from being checkpointed.

[1]Open-source reference implementation can be readily found online, for instance [7].

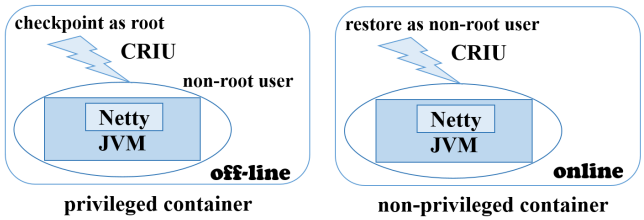

Figure 6. Checkpoint and Restore of a Netty based FaaS Framework

3.4 Transforming the FaaS Framework

The first step in the Replayable Execution transformation is the checkpoint step, which checkpoints a JVM process in an *off-line* privileged container, as shown in Figure 6. *Off-line* means this is done during framework build process, and only one checkpoint is needed per framework code revision. First, the `processrouter` is started, which in turn starts a JVM running the Netty server as a non-root user inside a privileged container. We then **warm up** or **train** the JVM using a training FaaS function and run through a predetermined number of `http` requests. The purpose of training is to ensure that the normal execution path through the framework is Just-In-Time (JIT) compiled. We further ensure that we trigger GC right after training in order to compact the heap, before we perform the checkpoint. This reduces the start-up time for restore as live objects are closer to each other in a compact heap, and thus fewer page faults are needed to bring them into memory.

Since the framework takes the same execution path (e.g., classloading, class/method locating, `http`/`json` message processing, logging, input parameters extracting, etc.) to launch every FaaS user function, one training function is sufficient. After that, we checkpoint the JVM process using CRIU [21], as shown in Figure 6. We perform the checkpoint step in a privileged container since checkpointing requires the use of `ptrace`. As checkpointing is done off-line, it is safe to be performed as `root`, under which the kernel facilities can be more easily accessed.

The second step is the restore step which is performed whenever a fresh FaaS container is invoked; instead of starting a JVM process, the processrouter starts the Replayable Execution command line tool (`RECLT`), described in section 5, which restores the checkpointed JVM. As FaaS containers come in different memory configurations, Replayable Execution needs to be able to handle the differences, and finally load in the user FaaS function.

3.5 Ensuring Correctness of Diverging Execution Paths after Restore

Environment Variable Handling via Reconstructor A restored process has exactly the same environment variables as the process that was checkpointed. Incorrect behavior can be introduced as new environment variables cannot be passed from the parent process. As Replayable Execution does not work in the context of the checkpointed process and cannot call `malloc` on its behalf, normal means of updating environment variables by `libc` functions such as `setenv` and `unsetenv`, or manipulating the `__environ` array directly cannot be used. We solve this problem by pre-allocating environment variables with sizes that are large enough for our purpose in the address space of the JVM, and injecting the new values directly into it.

Another challenge to updating the environment variables in Java is that the `HotSpot VM` takes a snapshot of environment variables at JVM bootup time, and the values are cached inside the Java heap in an unmodifiable map. While this speeds up access to the environment variables in Java, it defeats our environment variable updating mechanism. To create a complete solution to this problem, we implement a *reconstructor* method in the FaaS framework. When the framework detects that itself is a restored process, the reconstructor method is executed once to reset the environment map in the Java heap.

Leveraging Container's Virtualization Ability We take advantage of Linux container technologies for isolation and environment virtualization. Specifically, we rely on the PID and network namespaces, together with `chroot` to give each restored JVM process the illusion that it is running in its own system, with its own TCP port namespace and the loopback interface as if each restored JVM process can reoccupy the same system resources. Without namespaces, multiple restored JVM processes would be binding to the same endpoints (e.g., Unix domain listening sockets or TCP ports).

4 Private Anonymous Memory Sharing Across Containers

Replayable Execution leverages `mmap` to restore address space of the JVM such that both memory usage and startup time of the JVM are reduced. The first memory reduction comes from the on-demand paging nature of modern OS virtual memory subsystem; specifically only pages that are touched by the restored JVM are brought into the address space from the checkpoint file. As the JVM puts internal bookkeeping data in its address space, only a small portion of the internal JVM data is actually required to be in memory during execution of the user Java program [57]. Effectively, live but cold data is kept in storage. While other language runtime approaches return dirty pages that contain no live data back to OS, they cannot evict live but cold data

The second benefit comes from sharing pages across multiple JVM processes. While the Huawei serverless framework runs each FaaS function invocation in an independent container, the Linux page cache is aware of the fact that each JVM process has the same memory mappings from the file, and thus only a single copy is brought into memory. The final

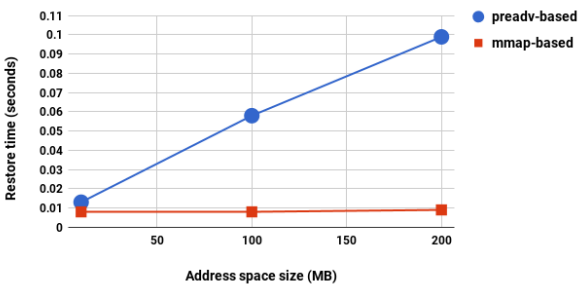

Figure 7. Restore Time Microbenchmark - mmap-based and preadv-based comparison

benefit that mmap offers is speed; data copying is needed when read is used, whereas only a few page table manipulations are needed to associate the memory mapping. Figure 7 depicts the speed difference between a read-based and an mmap-based implementation.

In order to capture all three benefits without modifying the Huawei FaaS platform, Replayable Execution creates private copy-on-write (COW) mappings of the checkpoint file in each container by passing the MAP_PRIVATE flag to mmap. When a JVM modifies its address space, new memory pages are allocated in a on-demand COW manner. This way, our design achieves data and JIT-code sharing without compromising security, and without changes to the JVM to add interprocess communication and synchronization to coordinate write to the shared memory segments as required by earlier systems [41], which further require the JIT compiler to generate position independent code (PIC) to allow different JVMs to load the JIT-code into each potentially unique address space. Overlayfs is needed to achieve page cache sharing so that the checkpoint files are brought into memory once only. In constrast, earlier layered filesystems do not offer such benefits [13].

Memory Map Algorithm While implementing a complex feature such as this inside CRIU can be non-trivial, we demonstrate it is possible to make a centralized and modular change to CRIU with the mmap algorithm, shown in Figure 8, as a *drop-in replacement* for the default preadv implementation.

While the default implementation uses preadv to read in multiple virtual memory area (VMA) contents all at once from the pages file, our mmap algorithm expands that out into a for loop iterating over the VMAs at line 8. Our algorithm needs to judiciously determine which VMA's content can be put back via mmap and which cannot. For instance, vsyscall area, which is not VMA_AREA_REGULAR, would require the usage of pread. MAP_GROWSDOWN indicates a stack area and would require pread as well, otherwise, it loses the ability to auto-grow. Since our algorithm is a drop-in replacement for the default implementation, as such, we need to first munmap the original buffer that was allocated for preadv at

```
1  #ifdef DEFAULT_IMPLEMENTATION
2
3   sys_preadv(args->vma_ios_fd, iovs, nr, rio->off);
4
5  #else
6
7  /* iterating through all the io vectors */
8  for (j = 0; j < nr; j++) {
9    void *p = (void *)iovs[j].iov_base;
10   unsigned long end = (unsigned long)p + iovs[j].iov_len;
11
12   /* for each IO vector, handle the memory mapping ranges */
13   while ((unsigned long)p < end) {
14
15     VmaEntry *vma_entry = lookup(p, args);
16     unsigned long io_size = min(vma_entry->end, end) - p;
17     ssize_t local_r;
18
19     if (!vma_entry_is(vma_entry, VMA_AREA_REGULAR) ||
20         (vma_entry->flags & MAP_GROWSDOWN)) {
21       local_r = sys_pread(args->vma_ios_fd, p, io_size, offset);
22     }
23     else {
24       sys_munmap(p, io_size);
25       addr = sys_mmap(p, io_size, vma_entry->prot | PROT_WRITE,
           (vma_entry->flags & ~MAP_ANONYMOUS) | MAP_FIXED |
           MAP_PRIVATE, args->vma_ios_fd, offset);
26
27       local_r = io_size;
28     }
29     p += local_r;
30     offset += local_r;
31   }
32 }
33
34 #endif
```

Figure 8. Memory Map Algorithm in Replayable Execution

line 24. Then at line 25, we use mmap to register a VMA's mapping to the checkpoint file. Additionally we force the protection bits to PROT_WRITE, which temporarily makes all pages writable. This is needed as the VDSO of the kernel may have changed on the machine where the JVM process is restored, and thus we rely on CRIU's vdso_proxify function to generate stub functions to VDSO functions. An additional fix-up pass is required to adjust the page permissions by removing PROT_WRITE for pages that are not originally writable.

Security Implications Since Replayable Execution restores every single JVM process from the same checkpoint, the base addresses of shared libraries are the same as well, which seems to defeat ASLR (Address space layout randomization) offered by the Linux kernel. We believe it is not a security risk as the JVM offers a secure sandbox that protects the integrity of the VM, and is in general immutable to buffer overflow attacks because of the array bound checks. Further, the processrouter acts as a firewall, fronting the network traffic that is sent from attackers, and invalid requests are filtered before they can even reach the JVM.

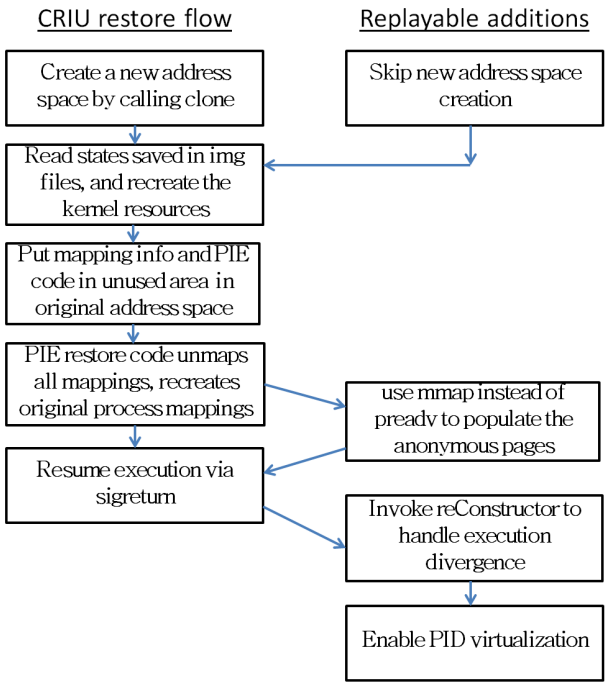

Figure 9. Modifications to CRIU's Restore Process

5 Overcoming Constraints of Process Restore Inside a Container

The Replayable Execution command line tool (RECLT) is a drop-in replacement for the Oracle JVM. RECLT accepts all Oracle JVM command line arguments, and does not rely on system calls that are restricted in container environments. We package RECLT together with the checkpoint files and the Java 8 Runtime Environment into a single container image, which is deployed onto each compute server. When a FaaS container is launched, RECLT is started in place of the Oracle JVM, and restores a JVM from the checkpoint. If restore fails for some reason, then the Oracle JVM is invoked. For security reasons, the FaaS framework runs inside a non-privileged container and the JVM is started as a non-root user as described in Section 3.4. Hence, RECLT needs to restore a process under the same condition. As such, we describe how we overcome the constraints when the needed system calls are restricted and the consequence of restoring a JVM to a different PID than what is checkpointed.

***ReplayableJVM: In-place Restore of* RECLT** Since the RECLT is drop-in replacement for the Oracle JVM in the way it is deployed, it is important to preserve the same parent-child process relationship as the default Oracle JVM. The existing CRIU restore mechanism creates a brand new process for the restoree which breaks this relationship. As such, we need to modify CRIU's restore flow as depicted in Figure 9, where the key modification is to skip new address space creation. Thus, the restore is made directly into the address space of the RECLT later on.

Afterwards, kernel resources are recreated by making system calls. The necessary state is placed at a memory range that is not used by the original checkpointed JVM, and control is transferred to the Position Independent Executable (PIE) [10]. The PIE code unmaps all memory mappings in the address space of RECLT except the code and data areas used by the PIE code itself. Once the address space is vacant, the JVM's memory pages are `preadv` or `memory map` back to RECLT's address space. At this point, the RECLT becomes the checkpointed JVM.

Instead of reconnecting the `stdin/out/err` of the restored process to the original terminal (or files in the case of the I/O redirection), as in the checkpoint environment, the restored JVM simply inherits them from the process that invokes RECLT. This way, RECLT becomes a drop-in replacement for the Oracle JVM.

5.1 Overcoming Prohibited System Calls

Containers use `seccomp` filters to block certain system calls to minimize the attack surface of the kernel. They include `ptrace`, `brk` and `prctl`.

`ptrace` `ptrace` is blocked in non-privileged containers but is used heavily by CRIU during restore for error detection, stage synchronization and PIE code removal. We replace each ptrace call with a sensible alternative so to preserve equivalent functionality under the constrained environment.

Originally for error detection, CRIU uses `ptrace` on each restoree (a restoree is a restored process). Should any restore error occur, CRIU is notified via `SIGCHLD` and the error is reported to the user. In the RECLT, when an error is detected, control is passed to a fallback path, at which point the Oracle JVM is invoked.

For stage synchronization, CRIU ensures that all threads within a process arrive at the synchronized restoration points at the same time to prevent some threads from running too far ahead, or signals become restored and delivered before signal masks are restored, causing loss of signal delivery. In the RECLT, we replace `ptraces` with `futexes`.

For PIE code removal, when the restore is complete, CRIU detaches from the restoree and `unmaps` the PIE code. With the removal of `ptrace` in the RECLT, the PIE code requires another technique in order to `unmap` itself, which we outline in the future work section. For now, the PIE code is left inside the restored JVM's address space.

`brk` *and* `prctl` The `brk` system call extends and shrinks the program break, located at the end of the uninitialized data segment. Address space randomization introduces randomness into the locations where the program break is placed. Restoring as a non-root user prevents the usage of `prctl` `PR_SET_MM` which modifies the kernel memory map descriptor fields to adjust program break. Without `prctl`, the only

Table 2. microbenchmark: PID translation overhead with 10 million invocations

Function	standard	with libpid	overhead
kill	2.46 s	2.53 s	2.8%
unlink	4.36 s	4.37 s	0.2%

alternative is using brk. However, it lacks the ability to restore to a program break that is at a lower address than that of the RECLT process, because the kernel prevents lowering break below a certain watermark. We solve the problem by raising the JVM's program break during the checkpoint step. This way, at restore step, only raising the program break through brk is required.

5.2 PID Translation

In a FaaS container, the processrouter always takes PID 1 of the PID namespace [2]. As the child of the processrouter, the RECLT can take any PID other than 1. While a process can be created with the desired PID by writing (desired PID-1) to the ns_last_pid file before its creation, the problem arises when /proc is read only in a container.

Allowing the JVM to be restored at a PID randomly chosen by the kernel can introduce undesired behaviour as the PID is cached inside its address space and the old value is used after restore. glibc caches the PID and TID in thread local variables, which we update with the corresponding values at the restore step. HotSpot's JVM Tool Interface (JVMTI) on the other hand, caches the PID in the Java heap which we do not have a safe way to modify the value. Since JVMTI is used by agents such as debuggers and profilers to interact with the JVM, we introduce a PID translation mechanism to offer compatibility.

We intercept system calls that take PID arguments and substitute the original PID with the new PID. Examples of system calls that take PID as arguments are kill and wait4. A more sophisticated approach is used for JVMTI, which, in the HotSpot implementation, encodes PID in some filesystem objects and through Unix Domain Sockets to uniquely identify the target JVM. We intercept the related system calls and substitute the arguments with endpoints encoded with the new PIDs. Table 2 shows insignificant overhead introduced by libpid.so which is the PID translation layer we developed.

6 Deployment Challenges

Two challenges are encountered when deploying ReplayableJVM inside Huawei's container based production cloud. The first challenge comes from how memory control group (cgroup), which is used by Docker, accounts for shared memory usages. The second challenge comes from adapting ReplayableJVM to different FaaS container capacities despite the lack of dynamic maximum heap size control mechanism with Oracle's JVM.

Fairness of Memory Accounting According to [1], the cgroup that first touches a page and thus triggering a page fault, is accounted for the page. The problem is the first JVM process is accounted for all the shareable pages, including the pages mapped from the checkpoint file, while subsequently launched JVM processes are not accounted for the pages made shareable by the first JVM because page faults do not occur. Memory accounting becomes even more unpredictable as shareable pages get evicted and brought back in again to memory. To overcome this problem, we employ the DaemonSet capability of Kubernetes which runs a small program to bring the checkpointed JVM image into memory via mmap MAP_LOCKED ahead of time. This way, each JVM is only charged for its private anonymous pages

JVM Memory Ballooning Memory capacity of FaaS containers ranges from 128MB to 1.5GB, and for each memory size, the JVM is started with the corresponding -XX:MaxHeapSize setting. The problem is ReplayableJVM gets a fixed MaxHeapSize at the time of checkpoint with no way to dynamically adjust the maximum heap size after restore. This is due to the lack of dynamic heap size control in Oracle's JVM. Modifying and maintaining a custom JVM with this capability introduces extra operational cost. An alternative is to provide a different checkpoint image for each FaaS container size but such alternative would defeat the purpose of memory sharing across different container sizes.

ReplayableJVM solves this problem by using memory balloons [50, 58, 62] to dynamically control the maximum size of the heap available to an FaaS function. A balloon module implemented via JNI first allocates memory from the Java heap and returns the underlying memory pages to the kernel via madvise prior to checkpointing. Then as part of the reconstructor method described in Section 3.5, the effective heap size is increased by releasing the appropriate number of Java references for the balloons after restore. We use SerialGC which pre-tenures large objects into the beginning of the old generation directly. As such, balloon memory will not be touched in the future even during GC's compaction process.

7 Evaluation

In this section, we evaluate Replayable Execution's performance of Huawei's Java FaaS framework as, described in Section 3. Our experiments focus on the following dimensions: (1) JVM memory usage over handling a large number of requests; (2) start-up time. Oracle HotSpot 64-Bit Server VM version "1.8.0_151" is used. The baseline OracleJVM refers to the default non-Replayable Execution of Netty whereas ReplayableJVM is the RECLT drop-in replacement for OracleJVM.

We also compare ReplayableJVM to OracleJVM with AppCDS. AppCDS is a commercial feature extending the system class data sharing (CDS) mechanism, which is enabled by default in OracleJVM, to include application (App) class data sharing. It is based on a similar approach where the JVM is first instructed to dump out the class metadata into a *jsa* file and subsequently, the *jsa* file is memory mapped into the JVM's address space. In so doing, AppCDS makes class metadata shareable across JVMs running the same framework. While AppCDS can only share class metadata and cannot be trained to produce JIT code like ReplayableJVM can, we nonetheless compare to it to demonstrate the advantages of ReplayableJVM.

We finally show that Replayable Execution is beneficial for frameworks other than JVM, including JavaScript, Python and Facebook's Nailgun Java server [25].

7.1 Experimental setup

The experiments are performed on an Intel Xeon CPU E5-2687W, which is a SandyBridge EP @ 3.0 GHz machine with 12 cores and with HyperThreading enabled. It has 30M of L3 cache and 256G of memory. Ubuntu 16.04 is used as the base OS together with Docker 1.12.6.

JVM baseline flags Huawei's FaaS framework already employs a competitive set of JVM flags which optimizes for memory footprint. A different maximum heap flag setting is used according to container's memory cgroup setting. For our experiments, we target 128M memory cgroup container, which is found to be most frequently used by Huawei's cloud operators. So the full set of flags used is: `-Xmx128m -XX:+UseSerialGC -XX:-TieredCompilation -Dio.netty.eventLoopThreads=4`. We showcase that the ReplayableJVM is able to further save memory on top of this competitive baseline.

Workload Description We use two FaaS functions: a training function for training our FaaS framework prior to checkpointing, and a Network Time Protocol (NTP) function as an actual function uploaded by an online FaaS user. The training function, shown in Figure 10, echoes the input *name* to the output and accesses the logger from the context data structure of the cloud framework. The NTP function also accesses the logger and instantiates a `Date` object by invoking its constructor, i.e. `Date objDate = new Date()`, and returns `objDate.toString()`. The NTP function, despite of its simplicity, is representative of the FaaS programming paradigm [39]. We also present results for a thumbnail creation FaaS function in section 7.5. As training is done offline, we are not interested in its performance. We only present the *online* performance evaluation using the NTP function. Given we are only interested in the start-up time and memory consumed by the FaaS framework alone, the complexity of the user function does not matter. A complex function

```
1  public class MyLambda {
2    public String myHandler(String name, Context context) {
3      RuntimeLogger runtimeLog = context.getLogger();
4      runtimeLog.log(name);
5      return String.format("Hello␣\%s.", name);
6    }
7  }
```

Figure 10. Training FaaS Function

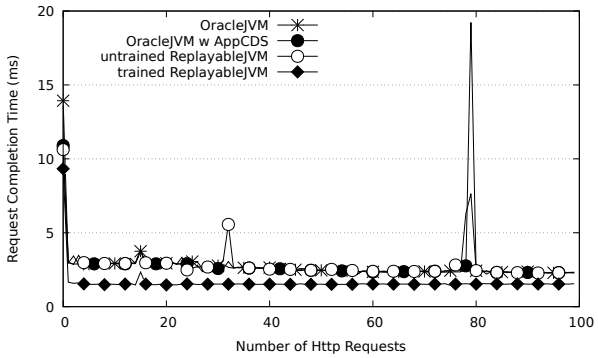

Figure 11. Request Completion Time Comparison for the First 100 Requests

may consume more heap space, which must be excluded in order to obtain the memory consumption of the framework.

7.2 Effect of JVM Training

We *warm up* or *train* the JVM by sending 10K requests to the FaaS framework running the training function and afterwards, a checkpoint is taken. Restoring from this checkpoint produces results for the *trained ReplayableJVM* in Figure 11. The 10K request point is determined experimentally as we observe from the JIT log that most of the framework functions in the common execution paths are JIT compiled by that point. The *untrained ReplayableJVM* would be one that restores from a checkpoint that is taken right after the Netty server is started, but before having received any request. This means, common execution paths where classloading of the user FaaS function, class/method locating, `http/json` message processing, logging, input parameter extracting, and etc. have not been exercised. In a trained ReplayableJVM, these common paths have been exercised 10K number of times. Another important role of the `reconstructor`, other than resetting the environment map as described in section 3.5, is that it is run only once immediately after restore and it creates a new classloader to load in the actual user FaaS function (i.e. `jar` file) at effectively the $(10K+1)^{th}$ request point. In contrast, the `reconstructor` runs at the 1^{st} request point after restore for the untrained ReplayableJVM. We demonstrate both the NTP and thumbnail creation Faas functions 7.5 running using the same trained (or untrained) ReplayableJVM checkpoint image.

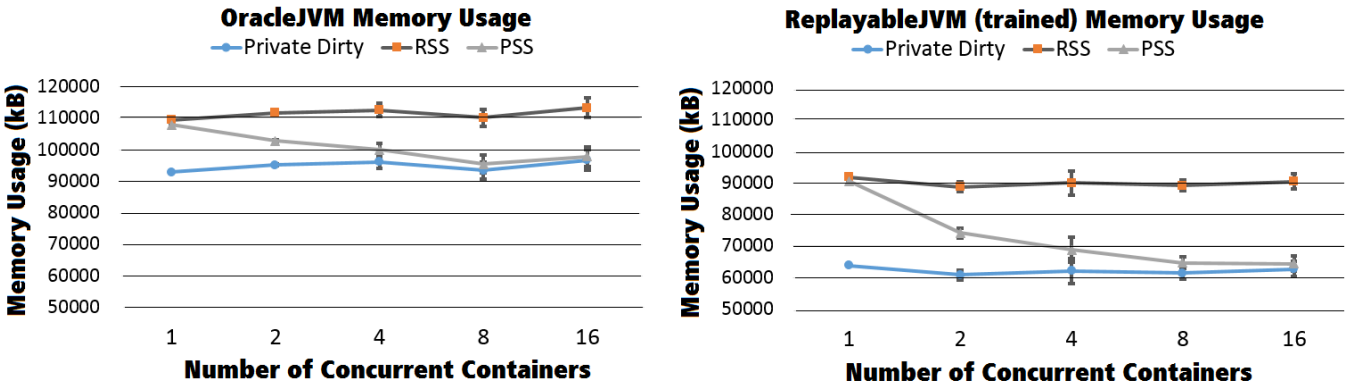

Figure 12. Memory Usages (after 50K Requests) for Concurrent Container Runs

Excluding the cold start-up spike, Figure 11 shows the trained ReplayableJVM is able to handle the first 100 NTP requests almost two times faster (3ms versus 1.5ms) than the untrained ReplayableJVM or OracleJVM with or without AppCDS. Further, it is able to avoid the JIT introduced latency spike at 79 request point. Each point in the figure is an average of 10 runs removing the highest and lowest outlier runs.

Through checkpointing a trained Netty framework, native code generated by the JIT compiler into the code cache is backed by a file and becomes shareable. At restore, on-demand paging ensures only the native code is pulled into memory, not the other JVM areas [57]. The 2X improvement in request completion time is a good indication that native code is being executed right from the start. Nonetheless, should an exception path through the framework be taken, the JVM must pull in the exception handler bytecode and other areas, and COW divergence will make Replayable Execution ineffective. An exception-triggering test (i.e. json parse error) found ReplayableJVM using equivalent amount of memory for exception handling as that of OracleJVM (i.e. worst case) when Java's exception handling class is pulled into memory. Thus, ReplayableJVM performs no worse than OracleJVM for these rarely executed exceptions.

7.3 Memory Usage Evaluation

Multi-Container Runs Figure 12 shows memory usages of typical deployment scenarios where n containers are running at the same time, using trained ReplayableJVM and OracleJVM respectively, for `n = 1,2,4,8,16`. Each graph shows three lines: resident set size (RSS), proportional set size (PSS), and private dirty memory measurements. The measurements are taken after 50K requests as a plateau has been reached at that point. Each point shows the average and standard deviation of memory usages over the n containers. In both graphs, private dirty and RSS memory measurements remain somewhat constant across different runs. PSS starts out at the same value as RSS when n=1 and approaches the

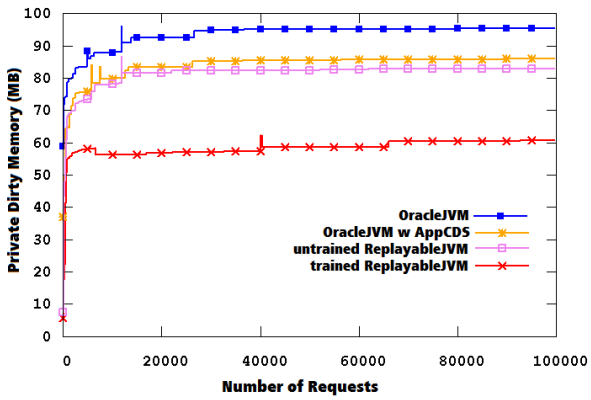

Figure 13. Private Dirty Memory Usages

private dirty memory measurements as n reaches 16. This is expected as PSS takes sharing of the *clean* portion of RSS by the operating system into account. Equation 1 shows that when the number of containers n increases, memory usage of a JVM process is dominated by its private dirty memory, or the anonymous portion of the RSS memory.

$$\texttt{memory usage} \propto \frac{clean}{n} + dirty \tag{1}$$

A comparison of the two graphs shows that ReplayableJVM achieves lower RSS and private dirty memory usages comparing to OracleJVM. Also, the gap between these two readings for ReplayableJVM is much wider than that of OracleJVM, by over 10MB. These are clear benefits of on-demand paging and improved code sharing, described in Section 7.2 for ReplayableJVM.

Single Container Runs Figure 13 shows the private dirty memory usages for running a single JVM instance over the course of 50K requests. A measurement is taken every 100 requests. We capture four runs: OracleJVM, OracleJVM with AppCDS [6], and ReplayableJVM with and without training.

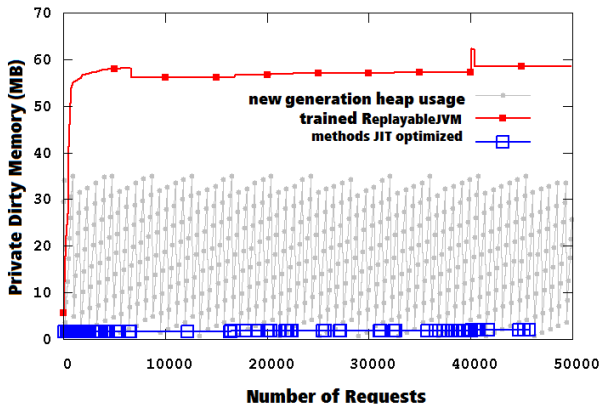

Figure 14. JIT Activities versus trained ReplayableJVM's Memory Usage

As shown in Figure 13, the steady-state memory consumption of an OracleJVM instance is around 94M. OracleJVM with AppCDS reaches steady-state at 84M. There are 2496 classes in the *classlist* file and AppCDS' *jsa* file is 18M in size. As not all classes are loaded at runtime, only 10MB of class metadata is brought into memory by AppCDS.

The ReplayableJVM run without training reaches a steady-state memory consumption at 81M. This slightly outperforms the AppCDS run, which indicates that ReplayableJVM is capable of capturing the needed class metadata sharing and more. The trained ReplayableJVM run reaches a steady-state of 60M quicker than the other runs because it has gone through 10K training requests prior to restore.

The gap of 21M between the trained and untrained ReplayableJVM is a result of memory used by the JIT compiler. The JIT compiler uses memory to perform its job (e.g. building intermediate representation, performing extensive transformations, etc.). Memory usage rises as more JIT optimizations occur. The trained run has significantly less JIT activities than the untrained run as most of the framework has been JIT-compiled prior to checkpointing. Further, the framework bytecode is not brought into the memory.

To gain understanding of the breakdown of 60M used by the trained run, we examine usage due to JIT activity and transient garbage.

JIT Activities: We correlate the JIT compiler's activities with memory usage increases of the trained ReplayableJVM run. Since a new classloader is used to load the user function JAR, the JVM naturally triggers JIT compilation to optimize this new class. This leads to frequent JIT activities as requests are made. We depict the number of methods being JIT optimized in Figure 14 with the bottom-most line with boxes. Most JIT activities occur within the first 5K requests of executing the user function which correlates well with the initial rapid rise of ReplayableJVM's private dirty memory.

Table 3. Start-up Time Comparison

Runs	Startup time	Std. dev.
OracleJVM	780ms	8ms
OracleJVM w AppCDS	474ms	8ms
untrained ReplayableJVM	54ms	2ms
trained ReplayableJVM	54ms	2ms

Transient Garbage: We also monitor memory usage of the new and tenured generations of the heap gathered via `jstat` [24]. With maximum heap size setting of 128M, the JVM internally configures the eden space to 35M and tenure generation to 87M. While the NTP function does not allocate memory explicitly, transient allocation still occurs as Netty allocates memory to handle each http request. Transient objects are allocated in the young generation of the heap. As these objects never make their way into the tenured generation and are garbage collected, the old generation usage remains constant and at less than 5%. In Figure 14, we only plot the new generation usage which resembles the shape of a saw-tooth. The memory usage rises as requests are handled. When the new generation memory usage reaches 35M, garbage collection kicks in and the usage drops nearly back down to 0M. However, since Serial GC does not return the dirty anonymous memory back to the OS, `smaps` indicates a constant usage of 35M of anonymous memory. Thus, during the first 5K requests of Figure 14, the memory usage is expected to rise up to at least 35M as requests are being handled.

In order to assess the memory use of ReplayableJVM, we subtract the 35M of young generation caused by transient garbage. Thus, the trained ReplayableJVM is able to contain the framework in 25M (i.e. 60M-35M=25M) of memory which is 2x reduction from what OracleJVM achieves (i.e., 84M-35M=49M).

7.4 Start-up Time Evaluation

Inside the FaaS framework, start-up time measurement for the Netty server is instrumented as shown in Figure 15. A clock reading is taken before the shell script that starts Netty. Then Netty undergoes socket `create`, `bind`, `listen` and finally waits in `accept` for incoming connections. The `processrouter`, which behaves as a third party observer, attempts to connect to Netty in a loop and upon success, sends a message to Netty and reads the reply. After this point, another clock reading is taken, and the delta gives us the start-up time. Table 3 shows the average and standard deviation of Netty start-up time over 10 runs. While AppCDS is able to achieve 1.65x start-up reduction, the Replayable technology is able to achieve 14x.

The start-up time measuring code path shown in Figure 15 uses the `/healthcheck` REST endpoint which is not exercised by the training requests because training requests go

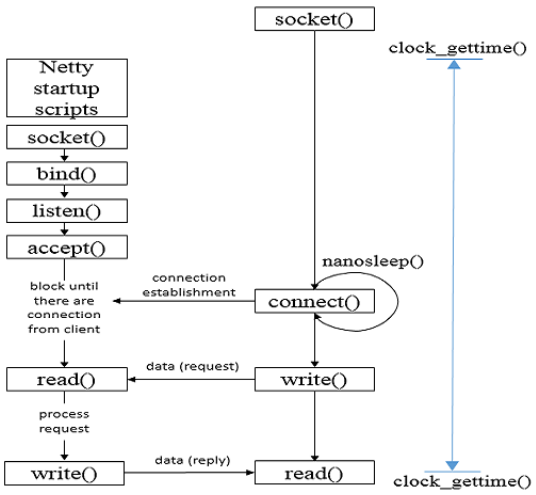

Figure 15. Start-up Time Measurement

Table 4. Processing Times for Different Image Sizes

Original Image Size	Thumbnail Size	Processing Time
75K	3.5K	120ms
750K	2.9K	540ms
1436K	2.5K	2150ms
4143K	2.5K	2376ms

through a different REST endpoint. As the `/healthcheck` path is not frequently executed, we did not train for it. Consequently, the same code for start-up executes for ReplayableJVM for both the train and untrained versions. As such, the trained and untrained ReplayableJVM achieve identical start-up time of 54ms.

We also measure the reduction in CPU time as a result of Replayable Execution via tick counters in `stat` [3]. With ReplayableJVM, the framework blocks at `accept`, ready to serve incoming requests, in 1 tick. When comparing to OracleJVM's 45 ticks, ReplayableJVM achieves significant amount of CPU cycle reduction.

7.5 Evaluation of Image Processing FaaS Function

In the absence of standard Java FaaS benchmarks, we present an evaluation of Replayable Execution for a FaaS function on Amazon Web Services (AWS). When a picture is uploaded to AWS Single Cloud Storage Services (S3), a FaaS function is triggered to create the corresponding thumbnail.

A simple Java thumbnail function is presented in Figure 16. We assume pictures reside locally within the container to avoid measuring the latency of downloading them from a remote storage server. Input images with varying sizes are taken from [34].

Figure 17 shows the memory usages (i.e. private dirty) for processing http requests to create thumbnails for four input image sizes: 75KB, 750KB, 1436KB, and 4143KB. And for each input image size, we capture the memory usages for creating 50 thumbnails. We observe the processing of 75KB image size follows similar early memory usage pattern as the NTP FaaS function shown in Figure 13 while for 750KB and above, the memory usage shoots above 120M within the first few requests.

Table 4 shows the time needed to create thumbnails with varying input image sizes. To create a thumbnail for a 4143KB image, 2376ms is needed. Thus, depending on the image size and the corresponding processing time, the JVM cold-start time may or may not be a significant contributor to overall execution time. For example, JVM cold-start up time of 790ms (w/o AppCDS) is 33% of 2376ms but may be a smaller contributor if larger image sizes, which takes longer time, are processed.

This, in turn, affects the extent of the benefit of Replayable Execution. The same can be observed in Figure 17, a memory saving of 15M-20M is more significant for the 75KB input image size than for larger sizes.

Table 5. Language Runtime Start-up Time Comparison

Languge Runtimes	Default	Replayable Execution
Python	102ms	52ms
JavaScript	209ms	52ms
Java (w/o AppCDS)	780ms	54ms

Table 6. Language Runtime Memory Usage (after 50K Requests) Comparison

Languge Runtimes	Default	Replayable Execution
Python	16 MB	11 MB
JavaScript	28 MB	24 MB
Java (w/o AppCDS)	94 MB	81 MB

7.6 Workloads with Intensive-Deflated Phase Execution Behaviour

We show that our Replayable Execution benefits workloads that exhibit intensive-deflated memory behaviour for Huawei's JavaScript and Python FaaS frameworks and for Facebook's Nailgun server. These results demonstrate that our approach extends to and is applicable for runtimes other than JVMs.

Python and JavaScript Faas Frameworks: Tables 5 and 6 show the start-up time and memory savings of Huawei's Python and JavaScript FaaS frameworks.

Huawei's Python FaaS framework uses Python 2.7 with the reference CPython implementation which does not support JIT. The framework uses tornado [32], an open-source http server, to bind to the unix domain socket for communicating with the `processrouter`. The JavaScript FaaS framework uses Node.js v6.10.0 with V8 JavaScript engine 5.1

```
1  public class MyLambda {
2      public String myHandler(String name, Context context) {
3          BufferedImage img = new BufferedImage(100, 100, BufferedImage.TYPE_INT_RGB);
4          BufferedImage temp = ImageIO.read(new File("/scratch/original/"+name+".jpg"));
5          img.createGraphics().drawImage(temp.getScaledInstance(100, 100, BufferedImage.SCALE_SMOOTH),0,0,null);
6          ImageIO.write(img, "jpg", new File("/scratch/thumbnail/"+name+".jpg"));
7
8          return String.format("Thumbnail_successfully_created");
9      }
10 }
```

Figure 16. Image Processing FaaS Function

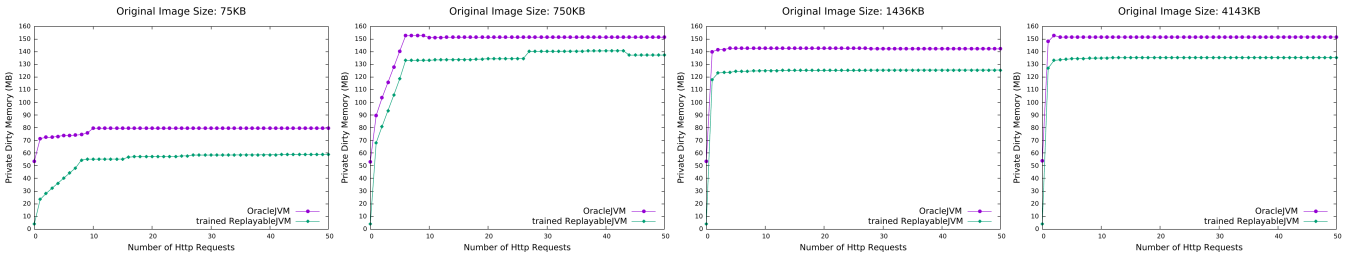

Figure 17. Private Dirty Memory Usages (first 50 requests) for Different Image Sizes

which supports JIT. The framework uses Node.js http module [27] for creation of an http server and binding to the unix domain socket. Both Python and JavaScript are managed runtime languages where memory is managed via reference counting or GC.

The results presented are gathered *without* training since the purpose of training is for triggering JIT and CPython does not support JIT. This means, the framework is checkpointed without handling any http request. And with replayable execution, a runtime is restored from image files instead of starting from scratch. Upon receiving a request, the restored runtime then loads in the user FaaS function (e.g. via the `require` module, or `imp.load_source`) and executes it.

The FaaS function used for the experiments is equivalent to the *echo* lambda shown in Figure 10 but written in JavaScript and Python respectively.

As shown in Table 5, all runtimes benefit from significant start-up time reduction ranging from Python's 2x to Java's 14x. Memory usage reduction ranges from 4M to 13M. The memory readings reported are gathered after 50K requests post-restore, which follows the same methodology described in Section 7.3. One interesting observation is that the Python framework remains at 11M from the first request, throughout the 50Kth requests while the other two runtimes' memory usages climb up and plateau at 50K request point. We believe this is due to the combination of reasons from Python's lack of JIT and the effectiveness of reference counting mechanism which marks memory `blocks` as free to be reused as soon as the FaaS function returns. That is, it does not accumulate transient garbage until the next GC cycle.

Ultimately, the significant start-up time reduction for the JVM and JVM being the heaviest memory consumer motivate a detailed study of Replayable Execution as presented in the paper.

Nailgun Server: We use the `Heartbeat.java` client example to send continuous requests to the nailgun server. The server consumes a steady-state 122M of memory at the time of checkpoint. After restore, the server resumes to respond to the `Heartbeat` requests at a steady-state 39M memory usage.

8 Related Work

Running multiple microservices in a single JVM, or a multi-tasking JVM (MVM) approach, enables maximum sharing amongst the microservices, but is challenged by the extensive changes to the JVM needed to guarantee isolation [35, 45, 47] and to allocate resources in a fair way [14, 53]. Work that also leverages OS level isolation and page sharing for microservices includes AppCDS, and JIT code as a shared library [6, 41, 61]. Similar to Replayable Execution, AppCDS uses memory mapping to achieve page sharing, but is limited to only sharing of class metadata generated in a trial run. Both AppCDS and JIT code sharing require JVM modification, in particular making JIT code position independent (PIC) requires a large amount of engineering effort with minimal return. IBM J9 VM's Shared Class Cache (SCC) is a memory mapped file which also stores ahead-of-time (AOT) compiled PIC code [36]. Replayable Execution places pages back to exactly the same virtual addresses for all instances of JVM, allowing code sharing to be naturally achieved without

such extensive modifications, and subsuming all benefits from AppCDS and JIT/AOT code sharing.

Container reuse or pre-warming is another effective approach to improve start-up time but requires additional complexity to the scheduler for keeping track of reusable containers (e.g., those that loads similar classes). In contrast, a ReplayableJVM is deployed/launched identically as a stock JVM and requires no extra operation costs for keeping resources alive.

Cloneable JVM [49] keeps a master JVM alive and uses fork to create children JVMs. However, it requires significant changes to the J9 VM to recreate kernel states for the children JVMs. In Replayable Execution, process restore is done externally and can be easily applied to different language runtimes.

ALMA [38] uses CRIU with no modifications for C&R to migrate a JVM but modifies JVM/GC so that only live objects are checkpointed. The purpose is to minimize the image size for lowering network traffic. Pages are restored via the default `preadv` mechanism in CRIU. However, the work does not exploit page sharing. ReplayableJVM invokes GC but the purpose is to improve live object locality. Since SerialGC does not return dirty memory (containing no live objects) back to the OS, the image size is not minimized. This is not a problem because Docker image is pulled in once and stored locally on the server. On the device side, checkpointing has been implemented on Android for quick app recovery [51], and boot time optimization [48], [29].

We also compare restore techniques implemented by DMTCP [11] to our extensions to CRIU. While we also use the LD_PRELOAD mechanism to implement PID translation, DMTCP uses it to intercept more than 30 key system calls, e.g., `getpid`, and `clone`, in order to manage the process tree properly and to perform PID virtualization. This is more involved than our thin translation layer. DMTCP 2.5.0 version onward supports a FAST_RST_VIA_MMAP path where memory pages are `mmap`ped into the restored processes. However, the driving purpose is for a faster startup time, not for anonymous page sharing. Overall, the coordinator process introduces such heavy overhead that restoring our FaaS framework using DMTCP is slower than starting a stock JVM.

9 Conclusion

We have demonstrated that the *Replayable Execution* approach is effective in minimizing memory footprint of a certain class of workloads that exhibits *phased characteristics*. We detailed our systematic approach to determine and transform Huawei's Java FaaS framework into a *Replayable Execution* and along the way, building non-trivial extensions in and around a traditional checkpoint and restore tool to overcome limitations imposed by the containerized deployment environment.

Our work also ensure a common execution path and presents techniques for correctly handling divergent execution. These techniques can be useful for alternative approaches such as VM, or container level checkpointing.

Future Work Future work can explore a *ROMization* approach to disable JIT optimizations from happening in the common codebase in combination with other warm-up strategies [31]. The goal is to introduce a clear optimization boundary between the common codebase that is *ROMized* and the diverging execution path that continues to require JIT optimizations. Evaluation using larger, or more complex user FaaS functions can also be useful. Future work can explore a `sigreturn` oriented method such as the one described in [37] to remove PIE code.

10 Acknowledgements

We would like to thank Huawei's project management and product delivery team members: Haichuan Wang, Yaoqing Gao, Sendao Yan, and Peng Li, Jianhao Huang, and Guanshun Lu, for making this work publicly available on Huawei' FunctionStage Offering. We greatly appreciate the insightful feedback from the anonymous reviewers, and our colleague Abraham Chan. Finally, we want to express our deepest gratitude to Professor Tarek Abdelrahman for his mentorship, reviews and feedback so generously given to the paper.

References

[1] [n. d.]. Memory Accounting. https://www.kernel.org/doc/Documentation/cgroup-v1/memory.txt.
[2] [n. d.]. pid_namespaces(7) Linux Manual Page.
[3] [n. d.]. proc(5) Linux Manual Page.
[4] 2005. Multi-tasking Virtual Machine. http://www.javalobby.org/java/forums/t19780.html?start=150.
[5] 2013. SOFT-DIRTY PTEs. https://www.kernel.org/doc/Documentation/vm/soft-dirty.txt.
[6] 2016. Leveraging AppCDS to Optimize Application Startup and Memory Footprint in the Cloud. https://www.youtube.com/watch?v=dRw77QDSL-A.
[7] 2016. Netty Unix Domain Socket Server. https://github.com/dandaso/netty_uds/.
[8] 2016. OpenStack Functions-as-a-Service (Picasso). https://launchpad.net/picasso.
[9] 2016. Spring Cloud Microservice Example. https://github.com/kbastani/spring-cloud-microservice-example.
[10] 2017. CRIU: Restorer Context. https://criu.org/Restorer_context.
[11] 2017. dmtcp. http://dmtcp.sourceforge.net/downloads.html.
[12] 2017. Optimizing Enterprise Economics with Serverless Architectures. https://d0.awsstatic.com/whitepapers/optimizing-enterprise-economics-serverless-architectures.pdf.
[13] 2017. Use the OverlayFS storage driver. https://docs.docker.com/storage/storagedriver/overlayfs-driver/.
[14] 2018. Apache Tomcat 9. http://tomcat.apache.org/tomcat-9.0-doc/introduction.html.
[15] 2018. AWS Lambda. https://aws.amazon.com/lambda/.
[16] 2018. Azure Functions. https://azure.microsoft.com/en-us/services/functions/.
[17] 2018. Cloud Functions Cold Boot Time. https://www.youtube.com/watch?v=IOXrwFqR6kY.

[18] 2018. Cloud Functions Cold Boot Time. https://www.youtube.com/watch?v=IOXrwFqR6kY.

[19] 2018. Cloudflare Workers. https://www.cloudflare.com/products/cloudflare-workers/.

[20] 2018. Containers At Google. https://cloud.google.com/containers/.

[21] 2018. criu. https://criu.org/Main_Page.

[22] 2018. Google Cloud Functions. https://cloud.google.com/functions/.

[23] 2018. IBM Cloud Functions. https://www.ibm.com/cloud/functions.

[24] 2018. Java Platform, Standard Edition Tools Reference. https://docs.oracle.com/javase/8/docs/technotes/tools/unix/jstat.html/. jstat.

[25] 2018. nailgun. https://github.com/facebook/nailgun/.

[26] 2018. Netty Project. https://netty.io/.

[27] 2018. nodejs. https://nodejs.org/en/blog/release/v0.6.10/.

[28] 2018. Oracle FN. http://fnproject.io/.

[29] 2018. Overview of Android Memory Management. https://developer.android.com/topic/performance/memory-overview.html.

[30] 2018. Qinling. https://github.com/openstack/qinling.

[31] 2018. ReadyNow! Solving the Java warm-up issue in low-latency systems by Azul Systems. https://www.azul.com/products/zing/readynow-technology-for-zing/.

[32] 2018. tornado. https://https://www.tornadoweb.org/en/stable/.

[33] 2018. Understanding Container Reuse in AWS Lambda. https://aws.amazon.com/blogs/compute/container-reuse-in-lambda/.

[34] 2018. Unsplash - Photos for everyone. https://unsplash.com/search/photos/high-resolution/.

[35] Godmar Back, Wilson C. Hsieh, and Jay Lepreau. 2000. Processes in KaffeOS: Isolation, Resource Management, and Sharing in Java. In *In Proceedings of the 4th Symposium on Operating Systems Design and Implementation*. 333–346.

[36] D. Bhattacharya, K. B. Kent, E. Aubanel, D. Heidinga, P. Shipton, and A. Micic. 2017. Improving the performance of JVM startup using the shared class cache. In *2017 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing (PACRIM)*. 1–6. https://doi.org/10.1109/PACRIM.2017.8121911

[37] Erik Bosman and Herbert Bos. 2014. Framing Signals - A Return to Portable Shellcode. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy (SP '14)*. IEEE Computer Society, Washington, DC, USA, 243–258. https://doi.org/10.1109/SP.2014.23

[38] Rodrigo Bruno and Paulo Ferreira. 2016. ALMA: GC-assisted JVM Live Migration for Java Server Applications. In *Proceedings of the 17th International Middleware Conference (Middleware '16)*. ACM, New York, NY, USA, Article 5, 14 pages. https://doi.org/10.1145/2988336.2988341

[39] Adrian Cockcroft. 2017. Shrinking Microservices to Functions. (2017). https://www.youtube.com/watch?v=ZgxZCXouBkY microXchg 2017.

[40] Grzegorz Czajkowski and Laurent Daynès. 2001. Multitasking without Compromise: A Virtual Machine Evolution. In *Proceedings of the 2001 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 2001, Tampa, Florida, USA, October 14-18, 2001.*, Linda M. Northrop and John M. Vlissides (Eds.). ACM, 125–138. https://doi.org/10.1145/504282.504292

[41] Grzegorz Czajkowski, Laurent Daynès, and Nathaniel Nystrom. 2002. Code Sharing Among Virtual Machines. In *Proceedings of the 16th European Conference on Object-Oriented Programming (ECOOP '02)*. Springer-Verlag, London, UK, UK, 155–177. http://dl.acm.org/citation.cfm?id=646159.758681

[42] Grzegorz Czajkowski, Stephen Hahn, Glenn Skinner, Pete Soper, and Ciarán Bryce. 2003. *A Resource Management Interface for the Java Platform*. Technical Report SMLI TR-2003-124. Mountain View, CA, USA.

[43] Nigel Daniels. 2014. Securing the JVM. *JVM Language Summit* 2014 (2014).

[44] Peter J. Denning. 1967. The Working Set Model for Program Behavior. In *Proceedings of the First ACM Symposium on Operating System Principles (SOSP '67)*. ACM, New York, NY, USA, 15.1–15.12. https://doi.org/10.1145/800001.811670

[45] Nicolas Geoffray, Gaël Thomas, Gilles Muller, Pierre Parrend, Stéphane Frénot, and Bertil Folliot. 2009. I-JVM: a Java Virtual Machine for component isolation in OSGi. *2009 IEEE/IFIP International Conference on Dependable Systems and Networks* (2009), 544–553.

[46] Paul H Hargrove and Jason C Duell. 2006. Berkeley lab checkpoint/restart (BLCR) for Linux clusters. *Journal of Physics: Conference Series* 46, 1 (2006), 494. http://stacks.iop.org/1742-6596/46/i=1/a=067

[47] Chris Hawblitzel, Chi-Chao Chang, Grzegorz Czajkowski, Deyu Hu, and Thorsten von Eicken. 1998. Implementing Multiple Protection Domains in Java. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference (ATEC '98)*. USENIX Association, Berkeley, CA, USA, 22–22. http://dl.acm.org/citation.cfm?id=1268256.1268278

[48] Jim Huang and Kito Cheng. 2012. Implement Checkpointing for Android. In *Embedded Linux Conference Europe*. Las Vegas, NV. https://elinux.org/images/1/1c/Implement_Checkpointing_for_Android.pdf

[49] Kiyokuni Kawachiya, Kazunori Ogata, Daniel Silva, Tamiya Onodera, Hideaki Komatsu, and Toshio Nakatani. 2007. Cloneable JVM: A New Approach to Start Isolated Java Applications Faster. In *Proceedings of the 3rd International Conference on Virtual Execution Environments (VEE '07)*. ACM, New York, NY, USA, 1–11. https://doi.org/10.1145/1254810.1254812

[50] Avi Kivity, Dor Laor, Glauber Costa, Pekka Enberg, Nadav Har'El, Don Marti, and Vlad Zolotarov. 2014. OSv: Optimizing the Operating System for Virtual Machines. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference (USENIX ATC'14)*. USENIX Association, Berkeley, CA, USA, 61–72. http://dl.acm.org/citation.cfm?id=2643634.2643642

[51] Li Li, Yunhao Bai, Xiaorui Wang, Mai Zheng, and Feng Qin. 2017. Selective checkpointing for minimizing recovery energy and efforts of smartphone apps. In *Eighth International Green and Sustainable Computing Conference, IGSC 2017, Orlando, FL, USA, October 23-25, 2017*. IEEE, 1–8. https://doi.org/10.1109/IGCC.2017.8323571

[52] San Hong Li. 2017. Optimizing JVM at Alibaba, for e-commerce apps running on 100,000+ servers. *JVM Language Summit* 2017 (2017).

[53] Sheng Liang and Gilad Bracha. 1998. Dynamic Class Loading in the Java Virtual Machine. In *Proceedings of the 13th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '98)*. ACM, New York, NY, USA, 36–44. https://doi.org/10.1145/286936.286945

[54] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. 2014. *The Java Virtual Machine Specification, Java SE 8 Edition* (1st ed.). Addison-Wesley Professional.

[55] David Lion, Adrian Chiu, Hailong Sun, Xin Zhuang, Nikola Grcevski, and Ding Yuan. 2016. Don't Get Caught in the Cold, Warm-up Your JVM: Understand and Eliminate JVM Warm-up Overhead in Data-Parallel Systems. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 383–400. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/lion

[56] Robert O'Callahan, Chris Jones, Nathan Froyd, Kyle Huey, Albert Noll, and Nimrod Partush. 2017. Engineering Record and Replay for Deployability. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. USENIX Association, Santa Clara, CA, 377–389. https://www.usenix.org/conference/atc17/technical-sessions/presentation/ocallahan

[57] Kazunori Ogata, Dai Mikurube, Kiyokuni Kawachiya, Scott Trent, and Tamiya Onodera. 2010. A study of Java's non-Java memory.. In *OOPSLA*, William R. Cook and Martin C. Clarke, SiobhÃąand Rinard (Eds.). ACM, 191–204. http://dblp.uni-trier.de/db/conf/oopsla/oopsla2010.html#OgataMKTO10

[58] Carl A. Waldspurger. 2002. Memory Resource Management in VMware ESX Server. *SIGOPS Oper. Syst. Rev.* 36, SI (Dec. 2002), 181–194. https://doi.org/10.1145/844128.844146

[59] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. 2018. Peeking Behind the Curtains of Serverless Platforms. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 133–146. https://www.usenix.org/conference/atc18/presentation/wang-liang

[60] Peter Whitehead. 2014. Multitenancy and the Multi Tenant Java Virtual Machine. In *Impact2014*. Las Vegas, NV. file:///C:/Users/k00375917/Downloads/Impact2014_MT_2014040.pdf

[61] Bernard Wong, Grzegorz Czajkowski, and Laurent DaynÃÍ. [n. d.]. Dynamically Loaded Classes as Shared Libraries: an Approach to Improving Virtual Machine Scalability.

[62] Weiming Zhao and Zhenlin Wang. 2009. Dynamic Memory Balancing for Virtual Machines. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '09)*. ACM, New York, NY, USA, 21–30. https://doi.org/10.1145/1508293.1508297