System-Level Checkpointing of Verification Tools

Wäinö Kotilainen

School of Science

Thesis submitted for examination for the degree of Master of Science in Technology.

Espoo 22.10.2018

Supervisor

Senior University Lecturer
Vesa Hirvisalo

Advisor

M.Sc. Jussi Judin

Aalto University
School of Science

Aalto University
School of Science

Aalto University, P.O. BOX 11000, 00076 AALTO
www.aalto.fi
Abstract of the master's thesis

Author Wäinö Kotilainen

Title System-Level Checkpointing of Verification Tools

Degree programme Computer, Communication and Information Sciences

Major Computer Science **Code of major** SCI3042

Supervisor Senior University Lecturer Vesa Hirvisalo

Advisor M.Sc. Jussi Judin

Date 22.10.2018 **Number of pages** 62 **Language** English

Abstract

In modern software development many kinds of verification is performed to prevent regressions and to ensure robustness of the software. Execution of verification tasks is usually automated with continuous delivery (CD) systems built on CD-platforms.

Currently available CD-platforms (Jenkins, Concourse, GoCD) are essentially job schedulers based on traditional job scheduling model. They execute tasks to completion in order of arrival. This model is known to cause user dissatisfaction due to long wait-times when the variation in task execution times is high. It's also known to exhibit low resource utilization. This prevents integration of new kinds of verification, reduces cost-effectiveness and decreases developer productivity.

Preemption, that is task-switching, enables much more flexibility to scheduling. It greatly improves the system's responsiveness by reducing wait-times. It solves the problem of short tasks having to wait extendedly for long tasks to complete. By enabling time-slicing of resources it increases their utilization. The result is interactive service for developers, supporting more kinds of verification in CD and enabling more value to be extracted of available compute resources.

Implementation of preemption requires ability to suspend and resume the execution of verification tools. We evaluate system-level checkpointing, a technique used for preemption in high performance computing, that does not require modification of the verification tools. We selected Checkpoint and Restore in Userspace (CRIU) as the checkpointing utility to be evaluated. We evaluated CRIU's capability to checkpoint verification tools and measured checkpoint creation time and checkpoint image size. We selected AFL, AddressSanitizer, Valgrind and Android Emulator as the tools to be tested.

Our results show CRIU is not yet capable of preempting arbitrary verification tools as only AFL and Valgrind were checkpointable. Checkpoint creation was fast making it feasible for interactive use in a CD-system. Checkpoint image's size was found to depend on the verification tool's memory size, as expected, meaning most tools would be feasible for preemption to network storage in a cluster.

Keywords Continuous Delivery, Scheduling, Preemption, Checkpointing, Verification Tools

Aalto University
School of Science

Aalto-yliopisto, PL 11000, 00076 AALTO
www.aalto.fi
Diplomityön tiivistelmä

Tekijä Wäinö Kotilainen

Työn nimi Laadunvarmistustyökalujen varmistusvedostus järjestelmätasolla

Koulutusohjelma Computer, Communication and Information Sciences

Pääaine Computer Science **Pääaineen koodi** SCI3042

Työn valvoja Vanhempi Yliopistolehtori Vesa Hirvisalo

Työn ohjaaja DI Jussi Judin

Päivämäärä 22.10.2018 **Sivumäärä** 62 **Kieli** Englanti

Tiivistelmä

Nykypäivän ohjelmistokehityksessä käytetään monenlaisia laadunvarmistusmenetelmiä regressioiden estämiseen ja ohjelmistojen vikasietoisuuden takaamiseksi. Tällaisten tehtävien suoritus yleensä automatisoidaan jatkuvan toimituksen (CD) järjestelmillä, jotka on rakennettu jollekin CD-alustalle.

Saatavilla olevat CD-alustat (Jenkins, Concourse, GoCD) ovat pääpiirteissään perinteiseen ryväslaskennan vuoronnusmalliin pohjautuvia tehtävävuorontajia. Ne suorittavat tehtäviä saapumisjärjestyksessä alusta loppuun. Tehtävien keston vaihdellessa odotusajat kasvavat pitkiksi, joten mallin käyttökokemus on huono. Resursseja ei myöskään hyödynnetä tehokkaasti. Nämä estävät uusien varmistusmenetelmien käytön sekä heikentävät kustannustehokkuutta ja ohjelmistokehittäjien tuottavuutta.

Tehtävien vuorottelu tekee vuoronnuksesta joustavaa. Se lyhentää odotusaikoja huomattavasti. Lyhyet tehtävät eivät enää joudu odottamaan pitkäkestoisten tehtävien päättymistä ja resursseja hyödynnetään tehokkaammin. Näillä saavutetaan ohjelmistokehittäjille vuorovaikutteinen käyttökokemus, uudenlaisia varmistusmenetelmiä voidaan ottaa käyttöön ja laskentaresursseista saadaan parempi hyöty.

Vuorottelun toteuttamiseksi laadunvarmistustyökaluiden suoritus täytyy olla keskeytettävissä. Työssä arvioimme järjestelmätason varmistusvedostusta, joka on suurteholaskennassa käytetty menetelmä tehtävien vuorotteluun. Menetelmä ei vaadi muutoksia työkaluihin. Tarkastelemme Checkpoint and Restore in Userspace (CRIU)-varmistusvedostustyökalua, sen kykyä laadunvarmistustyökalujen vuorotteluun sekä vedoksen luontiin kuluvaa aikaa ja vedoksen kokoa. Kokeiltuja laadunvarmistustyökaluja olivat AFL, AddressSanitizer, Valgrind sekä Android Emulator.

Ilmeni, että CRIU ei vielä kykene kaikkien laadunvarmistustyökalujen vuorotteluun sillä kokeilluista työkaluista vain AFL ja Valgrind voitiin vedostaa. Vedoksen luonti oli nopeaa, mikä tekee varmistusvedostuksesta käyttökelpoisen vuorovaikutteisissa CD-järjestelmissä. Kuten oletettiin, vedoksen koko riippui laadunvarmistustyökalun muistin koosta, joten yleisimpien työkalujen vuorottelu verkkotallennusta käyttävissä laskentaryppäissä olisi mahdollista.

Avainsanat Jatkuva toimitus, Vuoronnus, Varmistusvedostus, Laadunvarmistustyökalut

Acknowledgements

I would like to thank Senior University Lecturer Vesa Hirvisalo for the highly professional and helpful supervising of this thesis that made this seemingly complex project very straightforward and pleasant. I would also like to express my sincere gratitude to M.Sc. Jussi Judin for the eager and effortful advising of this work and for all the support from the start to the finish line. In addition, I would like to thank Ericsson for facilitating this thesis and for the commendable flexibility enabling me to focus on finishing my degree. My Ericsson colleagues and `!nakkilan_pojat` I would like to thank warmly for the support on my way to becoming an engineer and for making me challenge myself.

Espoo, 22.10.2018

Wäinö Kotilainen

Contents

1 Introduction

In modern software development the use of continuous delivery (CD) with automated verification keeps increasing. Verification is utilized for a range of purposes such as detecting regressions, finding defects, analyzing stability and measuring performance. Tools used in verification are complex and computationally intensive applications that require highly varying amounts of execution time to complete. In continuous delivery this verification is performed every time the software is changed.

How verification tasks are scheduled for execution has a significant impact on developers' experience of the continuous delivery process, in turn affecting their productivity. Scheduling affects also how cost-effectively the available compute resources can be utilized. By introducing the possibility to take checkpoints of the progress of verification tools we enable them to be preempted, that means the actively executing task can be switched when necessary. This makes scheduling of verification tasks flexible leading to a number of benefits, such as short waiting times, interactive service and high compute resource utilization. Preemption also enables introducing new kinds of verification tools that require an extended execution time.

Previous research on checkpointing has focused on its use for fault-tolerance and high availability where the primary interest is in retaining the state of a critical application in case of the failure of the host computer. This is important for computational applications in high-performance computing (HPC) and service applications, such as database management systems, in enterprises. We explore the limits of checkpointing by utilizing it for the preemption of verification tools. The challenges in this scenario are the complexity of the applications to be checkpointed and the performance impact of the checkpointing procedure.

In practice, verification is automated with CD-systems that execute the build, verification and delivery steps of the development workflow. CD-systems are built on CD-platforms that provide task scheduling, execution, monitoring and development tool integration. Developers interact with these systems many times a day meaning interactive user experience and high performance of these systems is critical for the velocity of the software development process.

Popular CD-platforms we looked into (Jenkins, Concourse, GoCD) rely on traditional job scheduling model. Tasks are queued in order of arrival and executed to completion. This model is known to suffer of a number of problems. First, when there is high variation in tasks' length waiting time for short tasks will increase. Second, it exhibits low resource utilization due to resource fragmentation. In practice, verification tools that require a very long execution time (eg. fuzz-testers) cannot be executed on these platforms. To overcome the limitations of the traditional job scheduling model the literature recommends preemption, that is achievable by checkpointing.

To implement checkpointing of verification tools requires a method to save the progress of their execution. Verification tools rarely support this and implementing state-saving retroactively is very difficult. However, in system-level checkpointing the operating system saves the state of the requested processes. This means modification of the applications to be checkpointed is not necessary. On the other hand, this

method may suffer of a high performance overhead.

The use of checkpointing raises several questions: is system-level checkpointing the optimal choice and is this method capable of capturing the diverse kinds of resources used by verification tools? In addition, for good developer experience CD-systems should process tasks interactively and without delay so we would like to know how much execution time penalty preemption by checkpointing causes. Lastly, because some verification tools can require substantial resources we would like to know how large the resulting checkpoint images will be. In the following problem statement we describe in more detail how we plan to answer these questions in this thesis.

1.1 Problem Statement

Our main problem is that current continuous delivery platforms use inflexible and inefficient scheduling methods. This leads to long waiting, uninteractive service, low resource utilization and difficulty integrating more kinds of verification tools. Preemption is suggested in literature to make job scheduling more flexible and efficient and checkpointing has been proposed as one method of performing it. We want to find out whether verification tools can be preempted by checkpointing.

Checkpointing of verification tools is not expected to be easy due to their complexity. The problem is that these tools utilize many system resources in uncommon ways which is likely to make their checkpointing difficult. The selected checkpointing method needs to be able to record the state of these resources. In addition, verification tasks often consist of multiple parallel processes and their state needs to be recorded consistently together.

The final aspect of the problem is the performance of preemption. Because one of our reasons for enabling preemption is to increase the interactivity of scheduling in CD-platforms preemption itself should not cause too much execution time overhead. Since CD-platforms are cluster computing systems where network attached storage is used the data associated with the checkpoints must not be too large.

The goal of this thesis is to evaluate the capabilities and performance of checkpoint- and restore for saving the state of software verification tools. We focus on system-level checkpointing because it doesn't require verification tools to be modified. Currently the most actively developed system-level checkpointing utility for Linux is Checkpoint and Restore in Userspace (CRIU) which we will evaluate.

To evaluate the selected utility we will empirically test its capabilities by checkpointing several verification tools. We have chosen the tools so that they present a sample of functionality used in software verification that is commonplace and likely to expose issues and limitations of the checkpointing utility.

We report the capabilities of CRIU in checkpointing the resources utilized by the tools. We also report our findings regarding the steps required for making the tools checkpointable. We will also measure the size of the resulting checkpoint images and the time taken by the checkpointing to determine the suitability of checkpoint- and restore for interactive use.

1.2 Research Questions

In this thesis we focus on answering the following questions:

- Can checkpoint and restore be used for saving the state of software verification tools?
- Can all resources used by verification tools be saved by the checkpointing method? Does the selected checkpointing utility support all of these resources?
- Is creating a checkpoint fast enough to be usable in an interactive continuous delivery system? Does the size of the checkpoint files permit the use of network-attached storage?

1.3 Contribution

This thesis provides the following contribution:

- Literature review of continuous delivery practice, the evolution of continuous delivery platforms, modern cluster management systems and basic cluster scheduling practices. Discussion of the benefits of enabling task preemption on CD-platforms.
- Analysis of the feasibility of process checkpointing as a task preemption mechanism for CD-platforms, focusing on its use with software verification tools.
- Evaluation of the feasibility of Checkpoint/Restore in Userspace (CRIU) for preempting common verification tools. Implemented by testing the checkpointing of American Fuzzy Lop (AFL), Valgrind, AddressSanitizer and Android Emulator.
- Cursory evaluation of the performance of preempting AFL and Valgrind with CRIU.

1.4 Outline

The rest of the thesis is structured as follows: In section 2 we will describe the background of the thesis in more detail and present the relevant concepts discussed in the thesis. In section 3 we will describe the methods we used for answering the research questions. The implementation of our evaluation of the checkpointing system is described in section 4. In section 5 we present the results of our evaluation. In section 6 we discuss the results and in section 7 we conclude the work and our findings.

2 Background

In this section we present the context of this thesis and its motivation. We present the scheduling problems in continuous delivery that led us to investigate preemption and process checkpointing. We provide an introduction to the concepts relevant for understanding the contribution of this thesis and throughout the text address the current literature of the field. We also look at the current state of available CD-platforms, cluster management and checkpointing methods.

We begin in section 2.1 with an introduction to continuous integration (CI) and continuous delivery (CD) followed by a discussion about their importance in modern software development. In section 2.2 we discuss continuous delivery platforms that are used for the practical implementation of automated CD-systems. We compare a number of different platforms ranging from dated to state-of-the-art.

Because continuous delivery at any significant scale requires cluster computing we will in section 2.3 present how general purpose cluster management systems are used for managing compute resources and simplifying application development. With that knowledge we will in section 2.4 return to CD-platforms and discuss how they schedule work to a cluster. We note several deficiencies in this scheduling for which preemption, discussed in section 2.5, is proposed as an improvement.

We conclude in section 2.6 by introducing process checkpointing and the major techniques of implementing it. We also discuss the state of currently available checkpointing utilities with more focus on the CRIU-utility we evaluate in this thesis.

2.1 Continuous Integration and Delivery

Continuous integration (CI) is an agile way of doing software development. In continuous integration software developers submit their source code changes frequently to a common code repository managed with a version control system (VCS). This automatically triggers a build of the software and the execution of tests and other software verification tasks in a CI-system making any integration problems visible at the earliest opportunity [18].

Continuous delivery (CD) extends this automation system to cover the additional steps required for delivering the software product to its users. These steps can include user acceptance testing, performance benchmarking, security scans, software packaging in addition to the final deployment to staging or production environments [3, 11]. The CD-workflow is illustrated in figure 1. From now on we shall refer to this complete automation system as just the *CD-system*.

Continuous integration and continuous delivery are significant contributors to the efficiency and speed of modern software development [3, 22]. Hilton et. al. have found that, among other benefits, the use of CI has made software projects release twice as often compared to projects not using CI [22]. The rapid integration of changes increases the productivity of developers and decreases the time to market [3]. CI and CD enable a short feedback loop leading to faster iteration cycles which perfectly supports the iterative model of agile development.

According to Fowler, the main characteristics of continuous integration are the

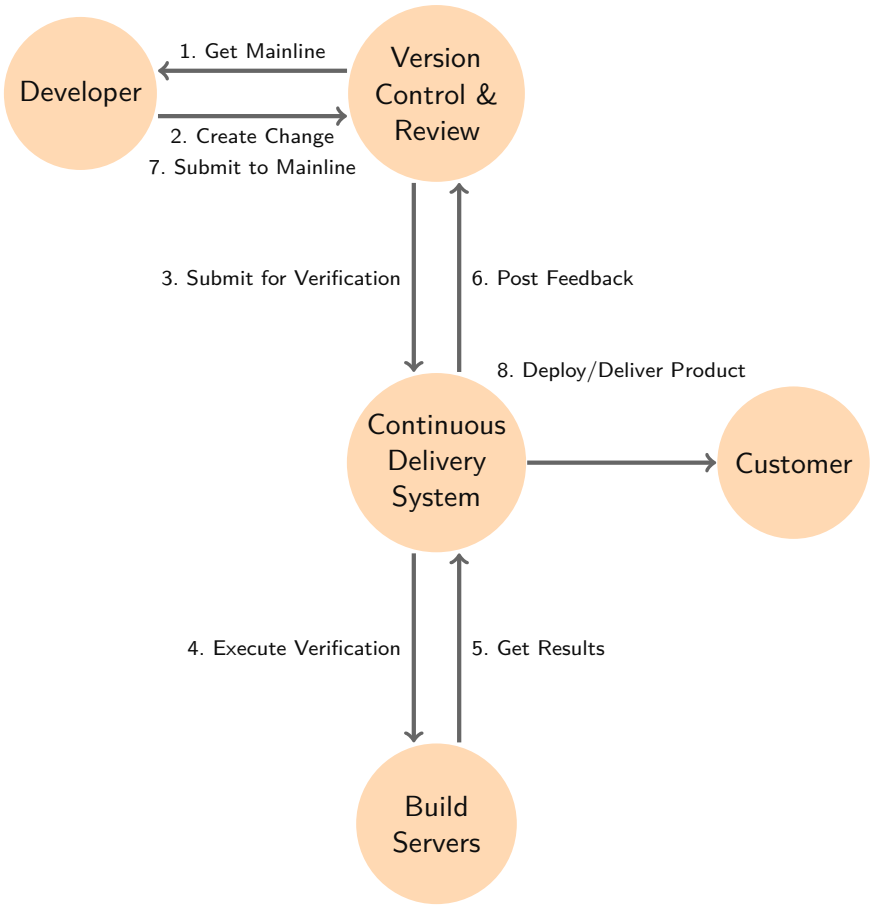


Figure 1: Continuous delivery workflow.

use of a version control system, automation of builds and tests and the frequent integration of changes back to the mainline. The CI way of working is supported by a continuous attention to maintaining the build in working order and keeping it fast. Confidence is gained in the quality of the software by having the test environment mirror the production environment as closely as possible. To make the CI workflow transparent it is common to have build monitors that display the status of the CI [18]. We will now discuss the main characteristics in more detail.

The basis for continuous integration is in the VCS. It is used to keep a master copy of the source code known as *the mainline*. With the VCS it is simple for developers to get a working copy of the latest source code and to maintain the local changes they have made. In addition, the VCS is used for integrating the source code changes back to the mainline and hence is often accompanied by a code review system. The VCS also enables the use of branches that can be used to manage pending code changes during their progression from development to production and to maintain fixes to released versions of the software, though the use of branches is better avoided to not complicate the development process [18].

Automating the build and test process saves time and makes the integration faster. It reduces the probability that integration errors go unnoticed. Recording

the dependencies of the software makes the build results reproducible and enables developers easily build the complete software in their development environments. This automation is important to give developers the ability to build and test the software to reveal integration problems in their changes before submitting them to the mainline. In addition, it is recommended that for every change the build and tests are executed automatically in an integration server before the changes are accepted to the mainline [18].

Frequent integration of changes is necessary to detect integration problems as soon as possible. Fowler states that frequent integration is also a form of communication that makes it visible if multiple developers are working on conflicting code. Since developers are expected to integrate their changes to mainline at least daily others will soon know if their changes will not work with the latest mainline [18].

We have now discussed what continuous delivery is and what its benefits are. We will next have a look at how continuous delivery is implemented in practice with continuous delivery platforms.

2.2 Continuous Delivery Platforms

We discussed earlier how changes are built and verified before they are integrated into the mainline. In continuous delivery there is also a multitude of other tasks to execute before every delivery, such as packaging and additional forms of verification. The CI literature refers to the environment where these tasks are executed as the integration server. In practice, however, these environments can be much larger than a singular server. In addition, software is required for building the automation for executing and monitoring the tasks. We call as *continuous delivery platform* the software that enables developers to build an automated CD-system for their project.

Some of the first systems for automating continuous integration were CruiseControl[1], Hudson[2] and TeamCity[3]. These systems were initially not much more than simple batch task executors. Triggered by a schedule or a change submitted to the VCS they would execute a configured list of tasks and notify developers of the results [25, 37]. In the beginning all tasks were executed locally, but support for remote execution was quickly added to Hudson and it is nowadays also supported by TeamCity [25].

From Hudson spun off the now very popular Jenkins project[4]. The recent versions of Jenkins allow the continuous deployment process to be defined as a pipeline of consecutive jobs, each job consisting of one or more tasks to be executed. Parallel execution of tasks is also supported. There are a large number of plugins for Jenkins that enable additional functionality and integration to many development tools.

The task scheduling model in Jenkins remains similar to Hudson at its core. Jobs are placed to a queue and executed to completion in first-in first-out (FIFO) order. Jobs can be executed in parallel if multiple Jenkins executor instances are available.

[1] CruiseControl, `http://cruisecontrol.sourceforge.net`
[2] Hudson, `http://hudson-ci.org/`
[3] TeamCity, `http://www.jetbrains.com/teamcity/`
[4] Jenkins, `https://jenkins.io/`

With a plugin it is possible to assign different priorities for jobs that affects their place in the queue, but once started jobs are executed to completion regardless of their priority.

In recent years there has been a proliferation of platform-as-a-service (PaaS) cloud services for continuous delivery, including TravisCI[5], Appveyor[6] and CircleCI[7]. These provide a set of features mostly comparable to Jenkins, but are not as customizable. Their greatest advantage is that since they are offered as services they are very easy to take into use. Their task scheduling models have some minor differences, but generally compared to Jenkins are more streamlined. TravisCI and Appveyor provide a fixed pipeline of steps that execute user-defined tasks, whereas CircleCI allows defining more complex pipelines similar to Jenkins.

In addition to the PaaS-services, there are also some new competitors for self-hosted continuous delivery platforms. Concourse[8], GoCD[9] and Drone[10] are some of the most recently introduced. They promise good usability, such as a clear ways of defining the pipelines, powerful visualizations and easier debugging.

A great advantage of the newer platforms Concourse, GoCD and Drone is the easy modeling of CD processes with clear sequential pipelines and steps. However, the disadvantage of all of these is that their task scheduling model is designed for batch execution and more flexible scheduling schemes are not supported. This limits the kind of tasks that can be executed in these systems. In addition, all of these systems require their own environments for executing the tasks, which means that a dedicated cluster is required for the system.

In literature we can find some descriptions of state-of-the-art continuous delivery platforms developed at Google and Microsoft. First we have the Test Automation Platform (TAP) and its closely related build system Bazel developed at Google [32]. The second is CloudBuild and its successor Concord developed at Microsoft [39]. What is common in them is that they have been designed for large-scale use to enable a single distributed CD platform to be used at the scale of the whole enterprise. For Google TAP, Memon reports that the system executes almost one million build tasks and more than 150 million verification tasks daily [32]. Esfahani reports that Microsoft's CloudBuild employs 10 000 machines, has more than 4 000 users with the daily number of executed build tasks exceeding 20 000 [15].

The problem TAP and CloudBuild try to solve is to scale continuous delivery for thousands of developers. The main issue is how to build and verify all the changes submitted at a high rate to the companies' monolithic code bases. Simply keeping up with the high rate of changes is a problem in itself, but the high computation cost of supporting CD at this scale and providing developers fast feedback are other equally important concerns [15, 32].

TAP and CloudBuild solve these problems mainly by tight integration with the

[5]TravisCI, `https://travis-ci.org/`
[6]Appveyor, `https://www.appveyor.com/`
[7]CircleCI, `https://circleci.com/`
[8]Concourse, `https://concourse-ci.org/`
[9]GoCD, `https://www.gocd.org/`
[10]Drone, `https://drone.io/`

novel build systems they are accompanied with and their data-parallel approach to task scheduling. Granular dependency information maintained by the build system enables the effective distribution and massive parallelization of build, verification and other CD tasks. In addition, striving for strict determinism in the build and verification steps enables aggressive caching of build outputs and verification results enabling a significant amount of computational work to be avoided [15, 32].

The current continuous delivery platforms Jenkins, GoCD, Concord and the others presented all perform simple batch scheduling of tasks, which naturally follows from the way these systems represent CD-pipelines as consecutive jobs of tasks. However, TAP and CloudBuild use task scheduling models similar to those used in data-parallel computing frameworks. Data-parallelism means the data set to be processed, in this case consisting of source code, build artifacts and development tools, is distributed to a cluster of compute nodes where it is processed in parallel by multiple nodes [44].

Esfahani has stated that the design of CloudBuild was influenced by Apache Tez, which is a DAG-scheduling[11] framework, and Apache YARN, which is a cluster resource management framework [15, 45]. From dependency information CloudBuild computes what tasks need to be executed (and what can be reused from cache) and then uses DAG-scheduling to schedule the execution of the tasks in the cluster [15].

An emerging theme in the CD-platforms we discussed is the use of cluster computing to enable the platforms to scale out to support a higher workload. The use of data-parallel computing makes the modern CD-platforms increasingly similar to systems used in scientific computing and big-data processing. An advantage of this is we can utilize the learnings made in the research of those fields to increase the efficiency and flexibility of CD-platforms. Utilizing data-parallel cluster computing for CD has its challenges as well, such as the SDK-problem[12] of distributing development tools that take time to install.

We have now discussed the characteristics of continuous delivery platforms. We also presented some history of the software that implements such a platform and described several current- and state-of-the-art solutions. Because at large scale continuous delivery relies on cluster computing and the trend in CD is to utilize data-parallel techniques we will next take a moment to discuss how clusters are managed and how they execute work.

2.3 Cluster Management Systems

We will now introduce the main features of cluster management systems. Most importantly, we will discuss work scheduling that is the primary function of these systems. With this knowledge we are able to better describe the work scheduling models used by the CD-platforms we presented in section 2.2 and to see their advantages and limitations. By understanding continuous delivery platforms as applications of cluster computing systems we can see how more efficient and responsive task execution is achievable if more flexible scheduling models are available.

[11] directed acyclic graph
[12] software development kit

Developing applications that utilize a cluster of computers is a problem for organizations in many fields, notably scientific computing and Internet service enterprises [23]. For this reason there exists a number of competing solutions to make the application development easier and to maximize the efficiency of cluster resource utilization. These can be categorized to cluster management systems and cluster computing frameworks. Older solutions are generally monolithic systems that couple together the application programming paradigm and resource management, which means these groups are partially overlapping [23, 45].

Cluster computing frameworks, like MapReduce and Pregel are designed for solving particular kinds of data processing problems. They tightly couple the compute resource management with the programming model [23]. Hadoop YARN is an attempt to decouple the programming model from the compute resource management enabling many programming models to be used concurrently in one cluster [45]. Mesos is a similar effort, that uses two-level scheduling to divide the resource management responsibility between the cluster management system and the application to enable multiple cluster computing frameworks to share a cluster [23].

Cluster management systems provide a platform for executing tasks of many kinds. Apollo is a DAG-scheduler that is intended for tasks that can be represented as an acyclic graph of dependent jobs [7]. Borg and Omega were designed for executing a mixed workload of batch- and service jobs [41, 46]. Kubernetes is similarly designed for a mixed workload, but with focus on easy building of distributed services [10].

The purpose of a cluster management system is similar to an operating system kernel, but at the level of a cluster of machines. The cluster management system manages tasks that are executed in the cluster, isolates them from each other and makes sure resources are shared according to configured policies. But most importantly, the cluster management system operates the *cluster scheduler* that decides the *placement* of tasks in the cluster [40].

Scheduling is a resource management problem where a number of tasks compete for a set of shared resources. Solving the scheduling problem requires finding a sequence of the tasks where each gets to use the resources on their turn. In addition, there is usually an objective to be maximized and a cost to be minimized, and possibly constraints on the assignment of jobs to resources. The resulting sequence of task to resource assignments is the placement [2].

The scheduling problem has a number of different forms in cluster computing. Scheduling is performed locally within each host and globally at the cluster level. The tasks to be scheduled can be dependent (ie. DAG-scheduling) or independent (job scheduling). Lastly, the placement can be decided in advance before execution begins (static scheduling) or during execution (dynamic scheduling) [2].

The traditional model of job scheduling is based on a fixed partitioning of the machines in the cluster. A set of work queues is assigned to each partition. When there is idle capacity within a partition more work is pulled from one of the assigned queues. Jobs are assigned to a specific queue according to the job's characteristics. The job queues themselves can be assigned varying priorities, enabling giving priority to certain kinds of jobs. Within a queue jobs are ordered in a strict first come first served (FCFS) ordering [17]. This model is illustrated in figure 2.

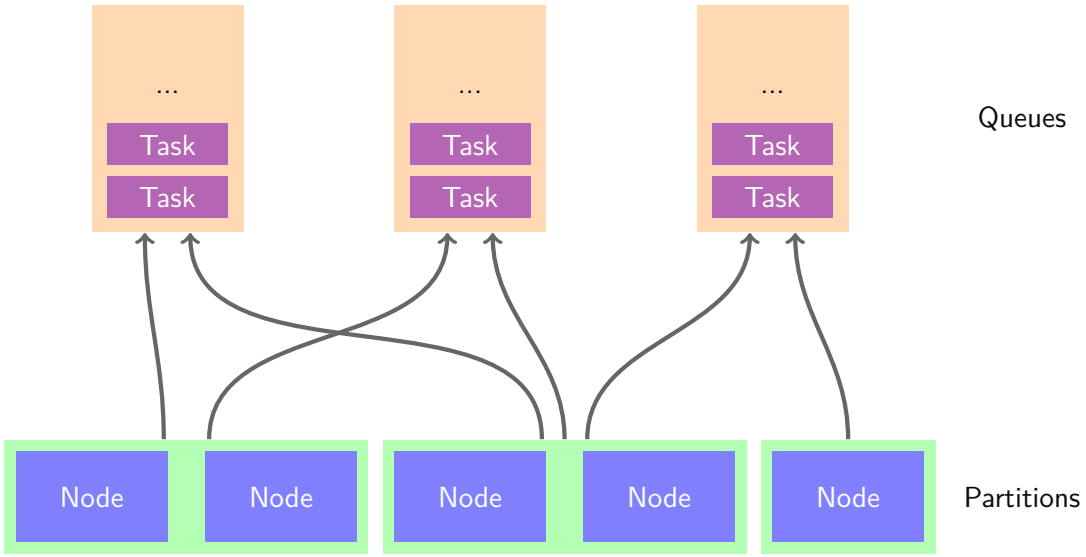


Figure 2: Example of the traditional cluster scheduling model.

Feitelson has described several shortcomings of the traditional model caused by its inflexibility and inefficiency. First, in this model jobs are always executed to completion. This results in poor responsiveness because the execution of long jobs blocks the execution of short jobs leading to extended waiting and thereby uninteractive user experience. Second, the sizes of the resource partitions are fixed and do not accommodate jobs of varying sizes or jobs that change size over their lifetime. Equipping this model with variable resource partitioning results in the additional problem that, since the scheduler cannot see the future it will make unoptimal choices in which jobs to start leading to fragmentation of the resources [16, 17].

To increase the flexibility and the efficiency of job scheduling Feitelson has listed some better scheduling techniques that could be used. Dynamic partitioning of resources would lead to their more efficient utilization by better accommodating jobs of varying sizes. Job preemption would enable gang-scheduling and time-slicing of resources leading to shorter response times and better user experience. Time-slicing solves the problem of having to execute an *a posteriori* unoptimal choice of jobs to completion therefore increasing resource utilization [16, 17]. Preemption is also the focus of this thesis.

We have now discussed the way cluster resources are managed, how work is scheduled in a cluster and how choices in scheduling affect a system's flexibility and efficiency. Equipped with this information we can return to continuous delivery platforms and evaluate them from the point of view of their work scheduling practices. We will do so in the next section.

2.4 Scheduling in Continuous Delivery Platforms

The increasing adoption of continuous delivery and the advances in software verification put more pressure on the CD-platforms that execute the jobs that build, verify

and deliver software products. When the number of users and jobs in a CD-system grows and the job variety increases the rudimentary scheduling models used by most of the current CD-platforms will show their limitations. Job execution becomes slow and uninteractive and increasing the system's capacity is increasingly costly. This is clearly a scalability problem.

To increase the scale of continuous delivery we need to be able to utilize the full capacity of the available computation infrastructure. But there are also times when the full capacity of the infrastructure is not needed. For example, an enterprise's software development activities may be localized to certain neighboring time zones which means the infrastructure will be underutilized outside the regular office hours and on weekends. For cost-effective operation we would like to get as much out of the resources as possible.

Our goals of making a CD-platform get the most capacity out of cluster resources and maximizing their utilization are the same goals cluster management systems and workload management systems, discussed in section 2.3, have tried to meet. We can therefore use the research and learnings regarding those systems to evaluate the current state of CD-platforms. We will focus on characterizing the platform's scheduling models.

As the most widely used CD-platform our primary interest is in Jenkins. The scheduling model in Jenkins is very similar to the traditional job scheduling model described by Feitelson [17]. A pipeline in Jenkins defines a sequence of stages. When the pipeline is triggered the stages are placed in a queue and executed in order by free executors[13]. In scheduling vocabulary the Jenkins stages correspond to jobs and executors to partitions of the available compute resources.

Jenkins allows stages and executors to be assigned labels. The labels enable certain kinds of stages to be executed on specific executors. This achieves effectively the same as the queues-to-partitions mapping in the traditional job scheduling model. However, since Jenkins uses only a single queue all pending stages will be blocked if there is no free executor satisfying the requirements of the stage at the front of the queue.

In Jenkins the stages are executed to completion and preemption is not supported. This means that scheduling in Jenkins shares the deficiencies of the traditional job scheduling model, most significantly long waiting times when there is high variability in the execution times of jobs. This limits the types of tasks we are able to perform as part of CD and harms users' satisfaction with the system.

With a plugin it is possible to assign stages static priorities[14]. However, since preemption is not supported there is no way to perform priority preemption that is preempting running jobs to free up resources for pending jobs of higher priority. Therefore the effectiveness of prioritization in Jenkins is limited.

In GoCD a pipeline defines a set of jobs. When triggered, the GoCD Server schedules the jobs by assigning them to available GoCD Agents that match the jobs'

[13]Jenkins – Defining Execution Environments. `https://jenkins.io/doc/pipeline/tour/agents/`

[14]Jenkins – Priority Sorter Plugin. `https://wiki.jenkins.io/display/JENKINS/Priority+Sorter+Plugin`

requirements[15]. The agents periodically poll the server to know when new jobs have been assigned to them and if the agents are free they will begin executing them.

The job scheduling model of GoCD is not documented very extensively. Moreover, in the user interface of GoCD there is currently no way to see what jobs are pending[16]. These make it difficult to characterize the scheduling model of GoCD. However, based on what we can see the model appears similar to the traditional job scheduling model, likely sharing the same limitations as Jenkins.

The other available CD-platforms are not significantly different from the two we just discussed. Greater differences can be found in the state-of-the-art solutions TAP and CloudBuild that are based on DAG-scheduling. They make use of the tight integration with their related build systems to extract a graph of jobs to be scheduled and utilize an external cluster management system for the job scheduling and execution.

The modern CD-platforms (eg. GoCD) are beginning to support execution in shared clusters managed by external cluster management systems (eg. Kubernetes). This enables CD-systems to share computing resources with other cluster computing frameworks increasing resource utilization. It also enables the CD-platform to benefit of the more flexible scheduling model provided by the underlying cluster management system. However, currently available CD-platforms don't seem to make use of this yet.

We have now seen that the current scheduling practices in CD-platforms don't quite enable us to reach our goals of interactivity and utilization. What we believe is the greatest deficiency in the scheduling models of the platforms is that they focus on sequential queuing of tasks and execution to completion. They do not allow tasks to be paused and resumed or migrated. Prioritization of tasks only affects their queue positions but once tasks are running they will execute to completion regardless of pending tasks of higher priority.

The consequence of these poor scheduling practices is that we don't get the full capacity of the cluster resources and the platform becomes non-interactive when there are tasks that take a long time to execute. Since the resource partitioning is fixed and tasks cannot be migrated the options for task placement are limited leading to suboptimal placements. Because tasks cannot be preempted the prioritization of tasks is ineffective and long running tasks cause short tasks to experience a long latency.

One solution to improve the scheduling in CD-platforms is to enable task preemption. Task preemption would enable migration of tasks enabling better placements through rescheduling. It would also allow more effective prioritization by priority preemption. In the next section we will present preemption in more detail and discuss how it can help in reaching our goals.

[15] GoCD Developer Documentation. `https://developer.gocd.org/current/4/4.2.html`

[16] `https://github.com/gocd/gocd/issues/14`

2.5 Preemption

Preemption means an active task is involuntarily paused to give another task the turn to execute. In preemptive scheduling the scheduler is allowed to switch active tasks to meet its objectives. This is in contrast to co-operative scheduling, where the scheduler can only start another task after the first one finishes or yields control voluntarily. Preemptive scheduling of tasks at the cluster level is also known as gang scheduling [16].

Preemption makes scheduling much more flexible. The main benefit is that it enables resources to be time-shared between competing tasks. Time-sharing leads to higher utilization of compute resources [16]. Combined with task migration preemption enables rescheduling to achieve more efficient placements [19]. It enables effective prioritization of tasks and short response times even when the tasks are of highly varying lengths [16].

Time-sharing improves the utilization of compute resources because it reduces the effects of their fragmentation. Feitelson has argued that because the length of tasks is usually not well known in advance and because the scheduler cannot know what future tasks will arrive this will lead to fragmentation of compute resources. Fragmentation causes low utilization of the resources because tasks of different sizes may not fit for execution at the same time. Time-sharing overcomes the problem of fragmentation and therefore leads to higher utilization [16].

Another advantage of time-slicing is that it enables the system to be used for interactive work. With time-slicing long running tasks will no longer cause short tasks to experience a high latency. As Feitelson has stated, users expect short tasks to finish in a short time [17]. This means time-slicing makes the system more responsive. This also enables the system to execute tasks of greater variety without harming the system's interactivity.

With support for task migration preemption enables the scheduler to move running tasks to different nodes. This can result in better placements because existing tasks can be rescheduled. Rescheduling means the scheduler considers all of the tasks in the system, including those that are already executing, when deciding placement [19]. This enables the scheduler to spread, consolidate and arrange tasks in such way that the efficiency of their execution is maximized, reducing resource fragmentation [17].

Preemption enables tasks to be effectively prioritized, since it allows low priority tasks to be evicted to execute a task of higher priority. This is called priority preemption. This reduces the waiting time high priority tasks experience until the start of their execution. Without priority preemption the task prioritization would not be as effective since the tasks would have to wait for free resources before execution can begin [29].

The advantages of preemption are interesting for a CD-platform. Time-slicing would alleviate the problem of long-running verification tasks blocking the execution of short tasks. It would also increase the utilization of the CD resources increasing their cost-effectiveness, compounded by prioritization that enables utilizing the idle capacity of the system. In addition, better placements would increase the performance of CD-systems overall.

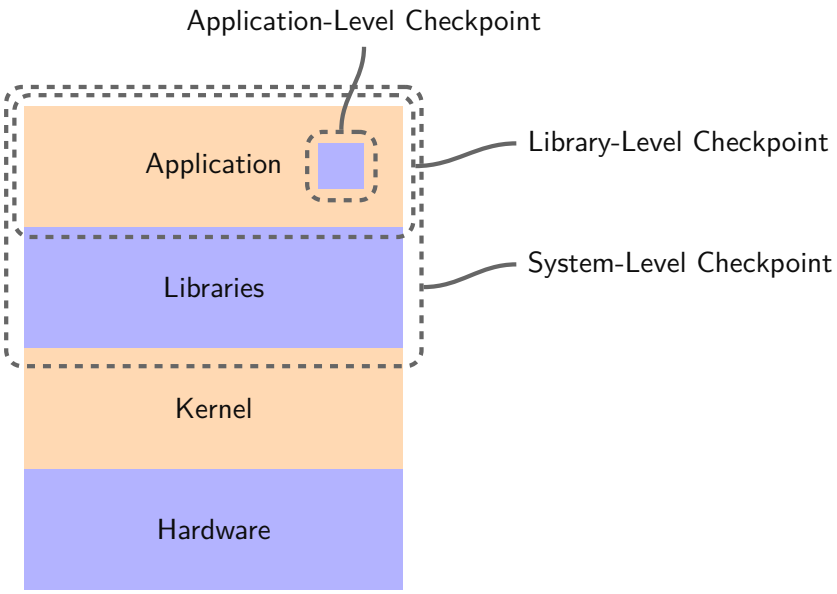


Figure 3: State captured by each checkpointing method.

Of these advantages, the increased responsiveness has an additional twofold benefit for CD-systems: it would improve the responsiveness of existing systems and enable us to comfortably execute a larger variety of tasks of different execution times. This would make it possible to integrate new kinds of verification tools such as fuzz-testers and load testing into CD-systems.

We have presented what preemption is and what kind of benefits preemptive scheduling would have for CD-platforms. However, to actually implement it a mechanism to preempt tasks, that are a group of one to many processes executing in a node of the cluster, is needed. Since our focus is in CD-platforms we are interested in how this works with verification tools specifically. One way to implement task preemption is process checkpointing which we introduce in the next section.

2.6 Process Checkpointing

Process checkpointing means saving the state of a process to a file. This enables the execution of the process to be stopped and later resumed. The main difference to process preemption of the operating system is that a checkpoint enables the process to be terminated and its resources freed when it's not executing. Some checkpointing methods allow the checkpoint to be taken of a group of processes together.

Checkpointing has been studied for many purposes. It has mainly been used for increasing the fault tolerance of large computation jobs in scientific high performance computing (HPC) applications [9, 27, 38, 47]. It can also be used to increase service availability and to implement load balancing [27]. In this thesis, however, we will focus on using it for enabling task preemption. This use has been discussed earlier by eg. Roman, Arora and Zhong [5, 38, 47].

There are a number of methods to implement checkpointing at different levels of the system. These include application-level checkpointing (ALC), library-level

checkpointing (LLC), system-level checkpointing (SLC) and compiler-assisted checkpointing. Each method has its own advantages and disadvantages depending on the intended use and tool availability for utilizing these methods varies by platform. We have illustrated the state each method captures in figure 3. Next, we will shortly introduce the methods.

The most basic checkpointing method is ALC. In ALC an application simply saves its own progress and exits. To resume the checkpoint the application loads the state of its previous progress and starts execution from that point. This is what many ordinary applications already support. However, implementing this can be difficult and time-consuming for many applications and even more so to retrofit it to existing applications. Since ALC-checkpointing is built-in and unique for each application this method is not universal [4].

The greatest advantage of ALC is that it produces the smallest checkpoints since each application only has to save what state is necessary to reconstruct the progress at the checkpoint. This means that ALC is also the fastest method. ALC produces portable checkpoints, since only the progress of the application's work is saved [4].

Compiler-assisted checkpointing works in similar way as ALC, but the code to record the state of the application's work is automatically added by a special compiler. This method is therefore comparable to ALC in performance. The automatic code generation makes this method easier to take into use than ALC, but is limited by the availability of such compilers. Silva has noted that this method cannot support arbitrary applications, since the state of external resources (eg. communication channels) cannot be captured by the application [43].

In LLC an application is started with a wrapper library that intercepts and tracks the application's system calls enabling the state of the application's resources to be checkpointed. This requires the wrapper library to implement a considerable amount of functionality to support all system calls of the kernel. A result of this is that not all system calls may be supported by an LLC-system [38].

An advantage of LLC is that it can be implemented in user-space. Duell has stated that for some use-cases this can be a disadvantage too, since it may not be possible to modify all process' resources from user-space. Duell states therefore that LLC cannot restore all types of resources limiting its application support [14]. However, Duell has listed as examples of such resources process IDs, which for instance in current versions of Linux can be specified by user-space. We gather that the research on LLC is not fully up to date, but the argument may be valid for certain resources in other operating systems.

SLC is checkpointing implemented in the operating system. In SLC the operating system saves the state of all of the application's resources. SLC's main advantage is that it is transparent for the application to be checkpointed and does not require wrappers or other preparation of the application. A disadvantage that results from this ease of use is that SLC-checkpoints will be large since they consist of the state of the whole application, which may contain unnecessary or redundant data [43].

We have introduced checkpointing and some methods for performing it. This concludes our review of literature and practices. In the next section we will choose a checkpointing method for use with verification tools and present how we evaluate it.

3 Methods

In this section we describe what methods we have chosen for answering our research questions. We will weigh the advantages and deficiencies of the methods and discuss how they affect the validity of our results. We will also describe in detail the empirical tests we are about to perform.

We will first describe how we approach our primary research question of the feasibility of preemption by checkpointing. We explain how we selected a checkpointing technique and a checkpointing utility that would be suitable for preempting verification tools. We will then discuss why there is a need to perform empirical tests of the utility to reveal its usability.

We continue with a discussion on how we performed the empirical tests of the checkpointing utility we selected (CRIU). We also present the measurements we used for evaluating the interactivity of the utility. This involves measurements of the checkpoint creation time and size. Since we perform empirical tests of the checkpointing utility this section is concluded by a detailed description of the four verification tools we selected for testing.

3.1 Finding out the Feasibility of Checkpointing

Our hypothesis for the primary research question is that checkpointing is a feasible technique for preempting verification tools. To verify the hypothesis we need to show that there exists a checkpointing utility capable of saving and resuming the execution of verification tools. Since we cannot test all verification tools that exist we will focus on several selected tools that we consider to be representative of common functionality used in software verification. Because of the many kinds of verification tools and the different ways to configure them our results may not apply to all of them and under all configurations.

Processes can be checkpointed on different levels of the system. The first step of verifying the hypothesis is to choose a checkpointing technique that best suits our needs. Our use case of checkpointing verification tools in a continuous delivery system places some requirements on the checkpointing method. We will now list the criteria we had for choosing the method.

Verification tools use many kinds of system resources, including uncommon ones, and in ways that are not common for ordinary applications. The main requirement for the checkpointing method is that it needs to be able to capture all of these resources. The second requirement is that the checkpointing technique must not require modification of the verification tools since that can be very difficult. Lastly, to maintain the interactivity of the CD-system the checkpointing should not cause too much performance overhead.

In addition to these general requirements the practical implementation of checkpointing places some more criteria on the choice of the method. First of all, we need an actively maintained checkpointing utility for our x86-64 Linux platform to perform the checkpointing. To ease the integration to CD-platforms the checkpointing method should not require a nonstandard kernel or additional kernel modules. Overall, we

Advantages	Disadvantages
Application-level	
• Small checkpoints • Fastest • Tool decides when to checkpoint	• Tool-specific • Hard to implement in existing tools
Library-level	
• Tool may decide when to checkpoint	• Limited resource support • May change tool behavior • May require tool modification
System-level	
• Can capture all resources • No tool modification • Actively maintained utility exists	• Checkpoints may not be portable • Large checkpoints • Slowest
Compiler-assisted	
• Fast	• Not available for our platform • Requires tool's source code

Table 1: Comparison of checkpointing techniques for use with verification tools.

prefer the checkpointing method that is the easiest to introduce.

We will next compare checkpointing methods according to the criteria we defined. The methods of Application-level, Library-level, System-level and Compiler-assisted checkpointing are compared. The comparison is also presented in table 1.

In application-level checkpointing (ALC) the verification tool itself would be responsible for saving its execution state. Unfortunately our verification tools mostly do not support saving their state and we also believe that generally few verification tools have that feature. Implementing ALC support into the verification tools is very likely a complicated task requiring lots of effort, so we do not expect the tools to feature it soon [5, 38, 43]. Thereby we reject ALC and have to look at the other options.

The second option is compiler-assisted checkpointing. In compiler-assisted checkpointing the verification tool is compiled with a specialized compiler that automatically inserts state-saving code into the application [43]. Since we are not aware of any actively maintained projects offering such a compiler for our x86-64 Linux-platform we reject this technique. Another reason to reject this technique is that it is not usable without access to the source code of the verification tool which we may not have for proprietary tools.

The third option is library-level checkpointing (LLC). Depending on the LLC solution used it can require the target application to be modified. Because LLC works by tracking the system calls of the target application it can be limited in its capability of checkpointing all system resources required by verification tasks. It can also affect the behavior of the target program. We will therefore not consider LLC

in this thesis, but it can be a feasible solution in some cases.

Our final option is system-level checkpointing (SLC), which means that the operating system is able to checkpoint the state of processes without any support from the processes themself [5, 14, 38, 1, 43]. Because it does not require any modification of the target processes we therefore choose to focus on system-level checkpointing. In the next section we will discuss it in more detail.

3.2 System-Level Checkpointing

In this thesis we focus on system-level checkpointing. We chose this checkpointing technique primarily because it is transparent to the processes being checkpointed. It does not require any support from the tools that are checkpointed [5, 43]. The verification tools used in our CD-system are from multiple providers and have not been designed to support checkpointing. Therefore SLC is the easiest checkpointing technique to take into use.

Another reason to choose SLC is that the operating system kernel has the most complete view of system resources utilized by the verification tools and therefore has the best capability to checkpoint them. This is in contrast to LLC, that depends on capturing system calls made by the target application and may not support checkpointing all the system resources it uses [14, 38]. Capturing system calls would also cause an execution performance penalty that does not happen with SLC.

System-level checkpointing is also able to fully restore the state of all system resources. User space solutions are unable to restore some resources, such as process- and session IDs [14].

Our final motivation for choosing SLC is because there exist actively developed SLC utilities for our x86-64 Linux platform. There is also recently published research about these tools.

System-level checkpointing does have some disadvantages for use in preemption. Out of the available options it is known to have the highest performance penalty and produce the largest checkpoint images [43]. This is because the checkpointing system does not know which resources of the target processes are necessary to save and which are not. The large size of the checkpoint image causes the checkpoint creation to take more time and to require more data transfer than the alternatives [5].

Another disadvantage of SLC is that due to the hardware and kernel specific attributes of processes such as available CPU instruction sets, system-level checkpoints are dependent on the host system and may not be portable to different hosts [43]. This means that using SLC checkpointing can require using homogeneous hosts to execute the verification tools.

Because in this thesis we choose to focus on SLC it means that if some verification tool is not checkpointable with SLC-checkpointing it could still be checkpointable with some other checkpointing technique like LLC and likely with ALC. Therefore the choice of SLC necessarily limits the validity of our results.

Since we chose to use system-level checkpointing we needed an SLC checkpointing utility for x86-64 Linux. Currently there exist two options: Berkeley Lab Check-

point/Restart[17] (BLCR) and Checkpoint- and Restore in Userspace (CRIU).

BLCR has been designed for use in scientific computing with the primary purpose of enabling gang scheduling and migration of processes. BLCR consists of a kernel module and a user space library the target application must be loaded with [14]. CRIU, however, only requires a recent kernel of version 3.11 or later[18]. The CRIU-project also seems more actively developed since the last software release from the BLCR-project is from year 2013.

The ability to use an unmodified kernel and to not require the target applications to be launched with additional libraries would make the integration of checkpointing into our CD system much easier and therefore we choose to evaluate CRIU in this thesis. There are also recently published articles reporting successful use of CRIU which further motivated our choice [24, 1].

3.3 Evaluating CRIU

To evaluate whether CRIU is capable of checkpointing verification tools we will test it with several selected tools. Although the system resources CRIU is able to checkpoint are documented quite thoroughly it is difficult to determine the checkpointability of the verification tools based on the documentation alone. Documentation can be incorrect or incomplete and we must verify the actual capabilities of the tool. More importantly, we do not have sufficient documentation of what resources the verification tools require. Finding out that information would require extensive studying of the source code of the verification tools and may easily omit something. In addition, since there can be limitations in how system resources can be utilized while still allowing CRIU to checkpoint them we consider empirical testing to be necessary to reveal the practical effectiveness of CRIU.

With empirical testing of the CRIU utility there is a possibility that the verification tools we test are checkpointable under our configuration, but may not be under some other configuration we did not test. This is a threat to the validity of our results. However, we attempt to reduce this effect by following the instructions of the verification tools and testing them in a recommended configuration. The opposite is also possible, that the verification tools are not checkpointable under our configuration, but are in some other configuration. To get valid results we need to thoroughly investigate the root cause of checkpointing failures to rule out configuration mistakes.

Our first test of CRIU is to execute each verification tool and to checkpoint the entire process tree of the tool while it executes. This shows that CRIU is capable of checkpointing all processes started by the verification tool and the system resources they use. If CRIU is unable to checkpoint a tool we will investigate if it is possible with a reasonable amount of work to configure the tool such that it is checkpointable.

Because the state of the verification tool changes during its execution we do not know what resources are in use at the exact time we perform the checkpoint. Therefore

[17] Berkeley Lab Checkpoint/Restart (BLCR) for Linux. `http://crd.lbl.gov/departments/computer-science/CLaSS/research/BLCR/`

[18] CRIU, Installation. `https://criu.org/Installation`

strictly speaking we do not have a complete certainty of the checkpointability of the tools from a single trial. However, our subsequent tests of CRIU will include repeated checkpointing of the verification tools and we assume that any issues would be revealed then. Since we try to model a scenario where the verification tool is preempted by the checkpoint we always create full checkpoints and not incremental checkpoints[19] which could be used for speeding up successive checkpoints.

After a successful checkpoint we will resume the tool to verify that it is able to continue execution. This is because some resources including system time and process statistics[20] change during the checkpoint- and resume procedure and we want to verify this doesn't cause the verification tool to fail continue execution. The way we determine the execution of the tool continues successfully is specific to each verification tool and we will discuss this more when we present the individual tools.

3.4 Interactivity of CRIU

After we have verified the tools can be successfully checkpointed and resumed we proceed to measure the time taken by creating a checkpoint that is our second test. Because continuous deployment systems are used in an interactive way and it is commonly preferred by software developers that build- and verification tasks execute in less than 10 minutes, no more than 2 minutes being preferable, the system should always be responsive [28]. Therefore when a new important task is submitted to the system it should not take too long to preempt a currently executing less important task to free resources for the new task. Measuring the time it takes to create a checkpoint tells us whether checkpointing with CRIU can be used for preempting verification tasks in a CD-system allowing us to use it for a greater variety of tasks of different execution time requirements and priorities without losing the interactivity of the system.

The measurement of the checkpoint creation time is performed by starting the verification tool and repeatedly creating a checkpoint of it. The verification tool is not reset between measurements and CRIU is configured to leave the tool running after the checkpoint is created. Performing the measurement this way gives us information if (and how) the checkpoint creation time changes over the lifetime of the verification tool.

Since the verification tools we are interested in are expected to require long execution times measuring the checkpoint creation time at different points in the lifetime of the tool yields us more describing results. In addition, since the verification tools may require some time after their start up to ramp up the actual work, due to eg. preprocessing of test input data, resetting the tool in between measurements would not yield realistic results. We repeat the measurement 10000 times with 10 seconds between each measurement. This results in taking measurements of the checkpoint creation time of the verification tool for over 24 hours.

Our third test of CRIU is to measure the size of the checkpoint images. A checkpoint image is the state of the running processes written to disk. The checkpoint

[19] CRIU, Incremental dumps. `https://criu.org/Incremental_dumps`

[20] CRIU, What can change after C/R. `https://criu.org/What_can_change_after_C/R`

image contains the virtual memory of the processes including program text, data segments (static, heap) and stack. The CPU execution state including the contents of all registers is also saved to the checkpoint image. In addition, the state of all of the processes' associated kernel resources, such as file descriptors, signal handlers, IPC channels, timers and sockets are also saved to the image [30, 38].

Because we are interested in distributed continuous deployment systems where the checkpoint images would possibly be stored in network attached storage it is important that the checkpoint images are not so large as to take a long time to transfer over network. In addition, since storage space in these systems is often limited and is always a cost we would prefer the checkpoint images to be as small as possible. Therefore the size of the checkpoint images directly affects the feasibility of checkpointing in an interactive CD system. Because the virtual memory segments of the verification tools can be substantial and as Li has shown the memory size of the processes correlates with the image size we would like to know how large the resulting checkpoint images will be [29].

Since some of verification tools use virtual memory in unconventional ways, which we will later discuss in more detail, we are interested in seeing whether it has an adverse effect on the size of the resulting checkpoint images. We would also like to know how the different kinds of virtual memory segments, such as shared memory, file-backed memory or unreserved memory are handled by CRIU and whether they have an effect on the image size.

CRIU saves the checkpoint image as a collection of files into an image directory we specify so the size of the checkpoint image is essentially the total size of the files in the directory. We measure the checkpoint image size by calculating a simple total size of the contents of the image directory using the standard `du` utility.

For the same reasons as with the checkpoint time measurement, we will measure the image size at different points in the lifetime of the verification tasks by measuring it repeatedly with 10 second intervals for 10000 samples. In practice we combined this measurement with the checkpoint time measurement to allow both to be performed with a single test run.

We have discussed so far our general methods of evaluating the capabilities and performance of CRIU. We will now present the sample verification tools we use as the target processes to be checkpointed with CRIU.

3.5 Short-Lived Processes

We begin the evaluation of CRIU with a verification tool that rapidly creates new processes that execute only a short time before terminating. We are interested in testing the checkpointing of this kind of short-lived processes because they commonly occur when executing software robustness tests. For example in fuzz-testing a target application is repeatedly executed in parallel with different inputs and the execution of the processes can be very short and often ends with a crash.

This kind of verification tool can reveal deficiencies in the checkpointing utility if it does not properly guarantee consistency of the checkpoint image when processes change rapidly and concurrently. Problems can be caused by race conditions due

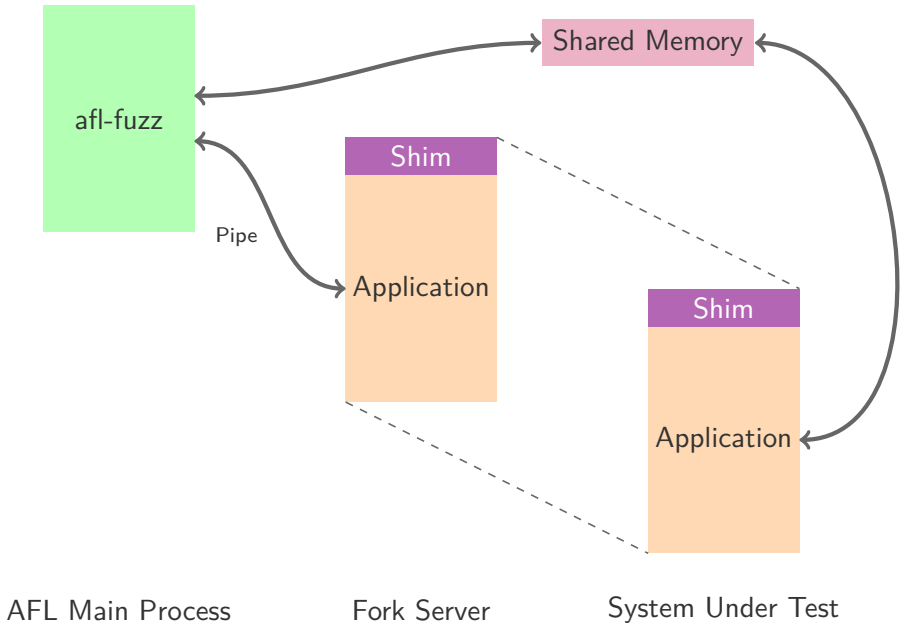


Figure 4: AFL process hierarchy and communication.

to the process tree changing or the state of the processes or their system resources changing during the checkpoint creation or restoration. Consistency of the processes is maintained by freezing their execution for the duration of the checkpoint creation and restoration and we want to see how well CRIU performs this [27].

To test CRIU's behavior with short-lived processes we tried checkpointing the American Fuzzy Lop[21] (AFL) fuzz-tester. AFL is a popular, easy to use and freely available fuzz-testing tool. It repeatedly executes a target application with different generated inputs and attempts to make the application take execution paths that lead to crashes or hangs. To speed up the repeated execution AFL is usually launched with several parallel instances. Rapid process creation and concurrency make AFL well suited for testing the consistency of the CRIU checkpoints.

We also needed to choose an application to use as the fuzzing target of AFL. For this purpose we chose the Opus Interactive Audio Codec[22]. We chose Opus because it's open source which enables us to compile it with the instrumentation required by AFL. What makes Opus good for our purposes is that it has already been fuzz tested extensively. Because discovering bugs in the target application is out of the scope of this thesis it's more worthwhile to use an already well fuzz-tested application as a target.

Testing an application with AFL involves compiling the target application with AFL-specific instrumentation. AFL uses the instrumentation to guide the test input generation that makes it highly efficient in finding interesting execution paths [20].

To speed up repeated execution of the target application AFL uses a scheme

[21] American fuzzy lop. `http://lcamtuf.coredump.cx/afl/`
[22] Opus Interactive Audio Codec. `https://opus-codec.org/`

called execve fork server[23]. AFL makes a small modification to the target application that causes it to pause after its initialization procedure and to wait for commands from AFL. AFL executes the modified target application as a child process (which is the fork server) and communicates with it over pipes. AFL commands the fork server to create additional copies of itself that will be the processes that are used for the actual fuzzing. By forking the target application from an already initialized state AFL is able to significantly speed up its repeated execution. Hierarchy of the processes AFL uses is shown in figure 4.

To record the branching decisions executed by the application under test AFL uses System V shared memory[24]. The AFL process allocates an area of shared memory where the instrumentation in the target application records counts of unique branching decisions. After execution of the target application is completed AFL analyzes the shared memory for new interesting execution paths.

3.6 Large Virtual Memory Allocation

Memory error detection is a common form of software verification. In memory error detection the memory accesses of a target program are analyzed for problems such as out of bounds access, use after free and read of uninitialized values. Because of the shadow memory used by many memory error detection tools the size of the program's allocated virtual memory can be significant which presents a challenge for checkpointing tools that attempt to save the state of the process' virtual memory areas (VMAs).

Shadow memory is an area of memory used for keeping metadata of the program's data. It is commonly used by memory error detection tools like AddressSanitizer to keep track of the program's memory. In AddressSanitizer shadow memory is used for determining the addressability of each byte of the program's memory. For this purpose AddressSanitizer maps one eighth of the program's address space as shadow memory [42].

AddressSanitizer is not a stand-alone tool. Instead it is implemented with the help of compile-time instrumentation and an accompanying run-time library. AddressSanitizer can detect out-of-bounds accesses in heap, stack and global data. It can also detect memory use-after-free issues. AddressSanitizer works by mapping every 8 byte sequence of the application's memory to the shadow memory area and with the instrumentation it checks all memory accesses for the addressability of the requested address. Accesses to stack are bounds-checked using red zones inserted at compile time. The way AddressSanitizer maps memory addresses to shadow memory is represented in figure 5 [42].

Because the amount of virtual memory AddressSanitizer maps as shadow memory is much larger than a typical host has physical memory (exceeding 20TB on our 64-bit host) this kind of verification tool could easily reveal limitations in the checkpointing utility. For this reason we chose to test CRIU checkpointing with AddressSanitizer.

[23] lcamtuf's blog, Fuzzing random programs without execve(). `https://lcamtuf.blogspot.com/2014/10/fuzzing-binaries-without-execve.html`

[24] American fuzzy lop 2.52b source code, `afl-fuzz.c` line 1351.

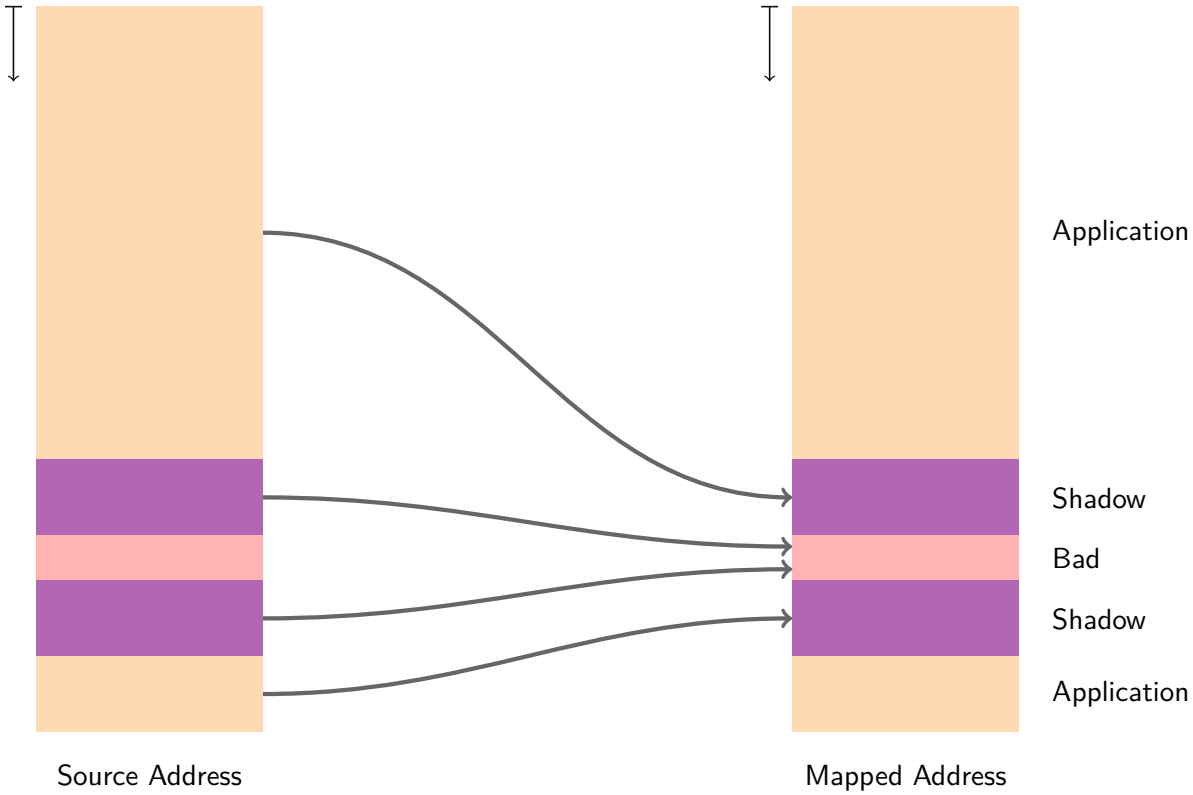


Figure 5: AddressSanitizer's address mapping as presented by Serebryany. Regular areas are mapped to shadow addresses. Addresses in the shadow areas are mapped to the bad area [42].

We want to know primarily whether CRIU is able to checkpoint this kind of verification tool at all, but would also like to see how the large size of the VMAs affects the checkpoint creation time and the checkpoint image size.

AddressSanitizer is used by compiling a target application with the sanitizer instrumentation. As a target application for AddressSanitizer we used Stress-ng[25] which is a freely available stress testing tool for Linux. With Stress-ng we create a workload that repeatedly allocates, modifies and frees memory. This workload resembles the activity AddressSanitizer is designed to analyze. Testing the checkpointability of AddressSanitizer is performed by checkpointing the Stress-ng process executing with the AddressSanitizer instrumentation.

Because of the way AddressSanitizer is implemented as instrumentation of a target application we are by definition unable to test it in isolation. It has to be tested with a target application which may impose its own limitations for checkpointing. If CRIU is unable to checkpoint a target application instrumented with AddressSanitizer we verify the validity of this result by attempting to checkpoint the target application without the instrumentation to see if the result is the same.

[25]Stress-ng. `http://kernel.ubuntu.com/~cking/stress-ng/`

3.7 Valgrind Virtual Machine

We will now present the third verification tool which is Memcheck of the Valgrind suite of tools. Memcheck is a very popular memory management analyzer implemented with the Valgrind framework. The popularity of Memcheck combined with the complexity and performance intensity of the virtual machine of Valgrind make this an interesting target to evaluate CRIU with.

Valgrind[26] is a dynamic analysis framework for building tools that analyze the behavior of programs at run-time. Valgrind enables tools to add analysis code to the target program's machine code during execution and to record shadow values representing interesting metadata of the program's execution state. Modifying the program's code by dynamic binary instrumentation enables Valgrind tools to be used with any program without access to its source code and with no modification which makes it easy to take into use [34, 35].

The use of shadow values enables Valgrind tools to perform many kinds of in-depth analyses of a program's behavior. The tools can track accesses to both memory and registers during the program's execution. One of the Valgrind tools is Memcheck, which analyzes the memory management of a program revealing memory leaks, use of uninitialized values and other defects. Other Valgrind tools exist for purposes such as data-race detection and heap profiling. We focus on Memcheck because it is the most popular [35].

Valgrind operates as a just-in-time compiling (JIT) virtual machine. It disassembles the machine code of the executing program block-by-block into an intermediate representation (IR), lets the analysis tool instrument the IR-code, applies several optimization passes and finally translates the IR back to native machine code to be executed. Valgrind wraps system calls in order to enable the analysis tool to track how the system calls access the program's memory and registers. Valgrind also intercepts signal handlers to keep the target program under Valgrind's control. Execution of parallel threads is serialized by Valgrind to make the shadow value updates atomic [35].

The compilation steps and the use of shadow values for memory and registers adds considerable overhead to the program's execution. In addition, the serialization of parallel execution makes only a single thread of the program execute at a time. This means executing a program in Valgrind has a significant performance penalty. Therefore Valgrind is suitable for implementing in-depth analysis tools providing results valuable enough to outweigh the performance impact. For example, for the Memcheck tool Nethercote states the average execution slowdown factor to be 22.2 [35].

In addition to the significant performance impact it has Valgrind is also a very complex application. Because of the way it executes the analyzed program within its own address space, performs JIT-compilation, captures system calls and signals, has its own internal C-library (to not clash with the one used by the target program) and schedules thread execution it behaves in many ways differently than common user applications [35]. This makes Valgrind especially worthwhile to be tested

[26] Valgrind. `http://valgrind.org/`

with checkpointing, since there are many possibilities it can reveal edge-cases and limitations in the checkpointing utility.

We evaluate CRIU's capabilities to checkpoint Valgrind by executing the Stress-ng load generator tool under the Memcheck analyzer. Similarly to our test of AddressSanitizer we will use a Stress-ng workload that repeatedly allocates, modifies and frees memory. However, unlike in the AddressSanitizer test we now use the Stress-ng tool compiled in its default configuration since Valgrind is based on run-time instrumentation and does not require modification of the target program.

3.8 Android System Emulation

System emulation is used when there is a need to execute software built for a hardware platform different from the host platform. The differences may be in hardware architecture, type of processor and what peripheral devices are available. One use case for system emulation is to enable the verification of software for embedded and mobile devices on the commodity x86 machines used for the software's development. This is why we have chosen to test the checkpointing of the Android Emulator[27], a system emulator for devices using the Android operating system.

Android is a widely used operating system for mobile devices. The Android Emulator simulates most of the hardware capabilities of these devices and provides software images of the many versions of the operating system. It also provides CPU emulation of the ARM processors a majority of mobile devices are using. This enables fast, convenient and accurate verification of Android applications compiled for the target hardware. A screen capture of the emulator emulating a smartphone is shown in figure 6.

The Android Emulator is an extension of the open source QEMU machine emulator. QEMU consists of CPU and device emulators, a debugger and user interface components. The CPU emulator in QEMU is based on dynamic translation of the target instructions to the host CPU instructions. The device emulators emulate various PC peripheral devices using the generic devices provided by the host operating system [6]. The Android Emulator extends these by providing emulators for hardware found in mobile devices, such as cameras and touchscreens and a user interface for operating them.

Since the Android Emulator is based on QEMU the advanced features of QEMU are available. This includes hardware-assisted virtualization when the target and host architectures are the same and supported by the host machine. Also included is live migration of the guest software to another instance of QEMU. This kind of complex features make the Android Emulator an interesting target to be tested regarding its checkpointability. Especially the live migration feature can cause problems for checkpointing since this overlaps with the functionality provided by checkpointing tools.

To test the checkpointability of Android Emulator we will start it in headless mode with an Android system image for an ARM CPU and attempt to take checkpoints of

[27] Android Developers, Run apps on the Android Emulator. `https://developer.android.com/studio/run/emulator`

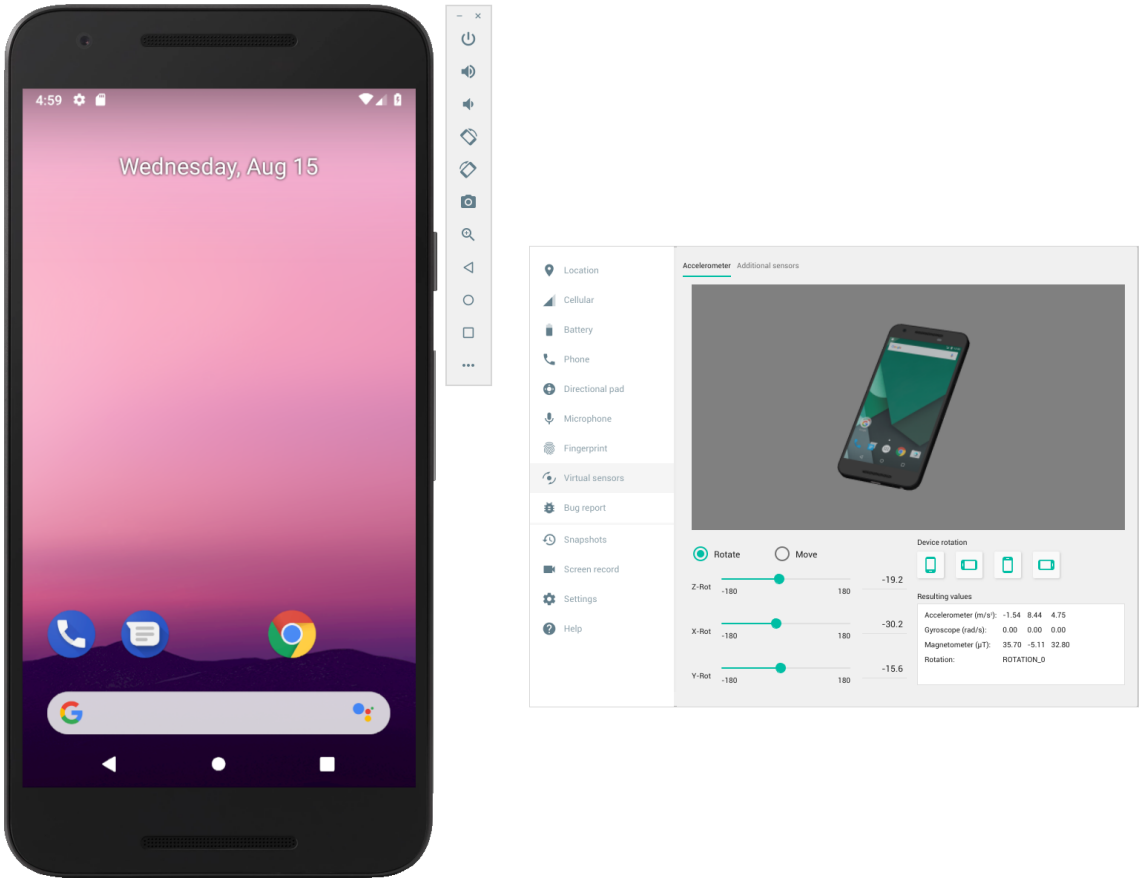


Figure 6: Screen capture of the Android Emulator emulating a smartphone.

the running system. This simulates preempting the emulator if it was used as part of a CD-system for verifying mobile applications. It reveals us what resources the emulator uses and whether they can be checkpointed. Since emulation is performance intensive, we will also measure checkpoint creation time and checkpoint image size similar to the other verification tools.

4 Test Implementation

In this section we present the implementation of our evaluation of the capabilities and performance of CRIU for preempting verification tools. We first discuss on a general level how we used CRIU to create checkpoints. We will then discuss the practical details of checkpointing each of the verification tools. After that we will describe our test environment and how we performed the measurements of the checkpointing time and checkpoint image size.

4.1 Usage of CRIU

Checkpoints are created with the command `criu dump`. With the `--tree`-parameter we specify the PID of the root process of the process tree to be checkpointed, that in our case is a shell script executing the verification task as child processes. By default the checkpoint image is written to the current directory but with the `--images-dir` parameter we specify a separate directory so that we can later easily measure the size of the checkpoint image. CRIU also provides an option `--shell-job` for conveniently disconnecting the session, controlling terminal and pseudo terminal of the target processes, which we will now discuss in more detail.

A process started from a shell session is usually coupled to the shell in a few ways. Because it is sharing resources with the shell CRIU will not dump the process. Processes started from a shell session by default belong to the session of the shell and are connected to its controlling terminal. Unless otherwise specified a process also has its standard I/O streams connected to the same terminal. In order to be checkpointed CRIU requires that the root process is a session leader. The root process must also have no controlling terminal and no pseudo terminals attached.

We can satisfy these requirements by making the root process of the verification task a session leader by starting it in a new session using the `setsid` command from the `util-linux` package. This also disconnects the task from the controlling terminal of its parent shell. By closing or redirecting the standard I/O streams we disconnect the processes from the shell's pseudo terminal. The `--shell-job` option of CRIU could relieve us of these steps, which makes dumping interactive tasks more convenient, but requires CRIU to become the parent of the process tree when the task is restored. Since we are checkpointing non-interactive verification tasks we chose to perform these steps ourselves and keep the restored process tree identical to the original.

Checkpoints are restored with the `criu restore` command. Similarly to the dump command, we use `--images-dir` to specify where the checkpoint image is located. Another important argument for this command is `--restore-detached`, which makes CRIU detach from the restored process tree. By default CRIU stays attached and terminates when the restored processes terminate. We enable this option to have the restored state of the verification task resemble its original state as closely as possible.

4.2 Checkpointing American Fuzzy Lop

In our early tests we found out that checkpointing the AFL verification task requires additional steps due to the System V shared memory AFL uses. Because System V IPC objects lack ownership semantics CRIU does not know which shared memory segments should be included in the checkpoint of the processes. In addition, because the identifiers of the shared memory segments are global in the system CRIU will not be able to restore them.

However, if we execute the target processes in a Linux IPC namespace CRIU can simply dump all shared memory segments within that namespace. When the checkpoint is resumed CRIU can use a namespace to recreate the shared memory segments with their original identifiers. Therefore by starting the AFL verification task in an IPC namespace we are able to checkpoint it.

We execute the verification task in an IPC namespace using `unshare`, also available in the `util-linux` package, with the option `--ipc`. Since creating a namespace requires the `CAP_SYS_ADMIN` capability[28] we must execute the command with superuser privileges. However, we do not want to execute the verification task itself with superuser privileges so we have to change user ID after the namespace has been created. In figure 7 we show how we executed AFL in an IPC namespace including changing the access privileges and the earlier discussed session setup and I/O redirection.

```
# TASK=./run-afl.sh
# AS_USER=user
# unshare </dev/null &>/dev/null --ipc \
      su --command "setsid $TASK" "$AS_USER"
```

Figure 7: Executing AFL in an IPC namespace in a new session.

We configured AFL following the recommendations in its instructions. To speed up the execution of the fuzzing it is recommended to use parallelism[29] and for this purpose we used a shell script `run-afl.sh` to execute as many parallel AFL instances as there are CPU cores in the host machine. In our test environment we had 8 parallel instances.

We also set the CPU frequency scaling governor[30] to `performance` as suggested by AFL. The version of the AFL we used was 2.52b. Prior to executing the tests we compiled the Opus encoder `opusenc` with instrumentation according to the AFL instructions.

Checkpointing and resuming the AFL verification task is performed using the same CRIU `dump` and `restore` commands we presented earlier. The shell script we

[28] The Linux kernel user-space API guide, unshare system call. `https://www.kernel.org/doc/html/v4.14/userspace-api/unshare.html`

[29] American Fuzzy Lop 2.52b documentation. `parallel_fuzzing.txt`

[30] Linux CPUFreq Governors. `https://www.kernel.org/doc/Documentation/cpu-freq/governors.txt`

use to to start the AFL instances also writes its own PID to a file. This allows us to easily read the verification task root process PID from the file when we want to checkpoint the process tree.

To verify the task can be successfully resumed from the checkpoint we check the status of the restored AFL instances using the `afl-whatsup` tool. Given the AFL master–slave instance synchronization directory (`syncdir`) as an argument the tool shows whether the instances are running and reports some metrics of their performance. We inspect that none of the instances report as dead and follow the performance counters for a moment and verify they are changing to be assured the instances were restored successfully.

4.3 Checkpointing AddressSanitizer

Tests of checkpointing AddressSanitizer were performed by compiling the Stress-ng stress-testing tool with the AddressSanitizer instrumentation. AddressSanitizer itself did not require any specific configuration. The Stress-ng tool was started with `vm` workers with no execution timeout. We then attempted to checkpoint the running task.

The `vm`[31] workers of Stress-ng repeatedly allocate, modify and free memory. We chose this type of workload because it resembles the memory allocation and addressing that is the kind of activity AddressSanitizer is intended to track. Similar to the previous test of AFL the number of worker processes was equal to the number of CPU cores on the host system.

Since our main focus is in AddressSanitizer, which is implemented as instrumentation added to the target application, we needed to verify that the Stress-ng tool does not affect the checkpointability of the task. For this reason we also tested checkpointing the tool under the same configuration but compiled without the AddressSanitizer instrumentation.

We noted in our early tests that the Stress-ng tool maintains a UNIX datagram socket to send log entries to the system log daemon. The socket is created even when logging to system log is not enabled. This is a problem for CRIU checkpointing because the other end of the UNIX socket is maintained by the system log daemon which is not part of the checkpoint. For CRIU this is known as an external UNIX socket[32]. By default CRIU will not checkpoint processes that have such sockets because it cannot guarantee the availability and state of the endpoint outside the checkpoint at the time the checkpoint is resumed.

However, since the socket is in this case used for sending log entries we assume the communication to be unidirectional towards the logging daemon. In addition, we expect the logging daemon to be always available. The Stress-ng tool also wasn't our main focus in this test so its functioning after checkpoint restoration was not a significant concern to us. With these assumptions we considered it to be safe to

[31] Stress-ng, General Commands Manual. `http://kernel.ubuntu.com/~cking/stress-ng/stress-ng.pdf`

[32] CRIU, External UNIX socket. `https://criu.org/External_UNIX_socket`

instruct CRIU to checkpoint the task regardless of leaving one end of the UNIX socket outside the checkpoint.

CRIU provides a command line argument `--external` for defining filters for external resources that shall be ignored when the checkpoint is created. Using a filter like `unix[<stress_sock_id>]` we can ignore the UNIX socket of Stress-ng. The ID of the UNIX socket `stress_sock_id` is found by parsing the open files of the main Stress-ng process with the help of the `lsof` command.

Stress-ng was compiled with the AddressSanitizer instrumentation using the GCC compiler. Since AddressSanitizer is part of the compiler package the version of AddressSanitizer tested is therefore the version included with the compiler.

4.4 Checkpointing Valgrind

We tested the checkpointing of Valgrind by running the Stress-ng load generator under the Memcheck tool of Valgrind.

Stress-ng was configured to use the same `vm` workers that we used with AddressSanitizer as described in section 4.3. Because the memory size of the processes is the most significant factor on the checkpoint size and the creation time [14, 33, 29] we configured the workers to use the minimum 4 kB memory size in attempt to get results that describe the behavior of CRIU and not only the I/O-performance of the host.

For this test an unmodified version of Stress-ng was used because Valgrind relies on runtime instrumentation. Required by Stress-ng, the same external UNIX-socket filter was also used with CRIU as in the previously described AddressSanitizer test.

4.5 Checkpointing Android Emulator

Checkpointing of Android Emulator was tested by starting the emulator from a snapshot and attempting to take checkpoints of the emulator while it's running. The emulator was executed in a new session similarly to the other verification tools. To simulate use in a CD-system the emulator was executed in headless mode. To reflect its use for verifying mobile applications compiled for the target architecture we used an Android system image for ARM architecture.

Snapshots record the state of the emulator such that its execution can be resumed later from the same state. The reason for using them is because the cold-boot of the Android operating system in the emulator requires about 6 minutes on our host before an idle state is reached. This is partly due to the slowdown caused by the performance intensive CPU emulation. Snapshots enable us to cold-boot the Android system once and run tests starting from this state. Booting the Android system is therefore not included in the tests or measurements.

Headless mode means the emulator executes without initializing its graphical user interface. By default the emulator displays this interface to enable the user to operate the simulated device and provide inputs for its sensors and other peripherals. However, we are interested in the emulator's use in automated CD-systems where verification tools are executed without interactive user input. In addition, a graphical

user interface would complicate checkpointing since the emulator would require additional resources for connecting to the user's display compositor and potentially a graphics acceleration device. For these reasons the emulator was executed with the user interface and audio output disabled.

The Android operating system to be executed in the emulator was chosen to be Android 24 since this is available as the default version that has full user privileges available[33] and is available for the ARMv7 mobile CPU.

4.6 Measurements

We performed our measurements in an x86-64 PC running Debian Buster GNU/Linux operating system. Complete specification of the test environment is presented in appendix A.

To measure the time elapsed to create a checkpoint we started the verification tools and created checkpoints of them during their execution. We used CRIU's `--leave-running` option to create the checkpoints without terminating the tool. To get measurements at different times over the execution of the verification tools the measurement was repeated for 10000 times with 10 seconds between each trial. We used the default mode of CRIU to create a full checkpoint each time.

We measured the time elapsed for creating a checkpoint with the built-in `time` command of the Bash shell. With this command we get the real elapsed time of the `criu dump` command execution in millisecond precision. This measurement includes the time taken by the creation of the checkpoint and the time required to load and execute the CRIU tool itself. This gives us a practical figure of the time required to create a checkpoint but will be subject to various sources of variation.

There are several factors that add variation to execution time measured this way. Most significant are the effects of the operating system's I/O caching. Because of the caching the measurement will not include the complete time required to write the checkpoint image to storage. When the checkpointing tool exits the checkpoint image may still be in the I/O cache of the operating system and not fully written. Also because of caching the loading time of the CRIU tool will likely be shorter on subsequent executions than the first one adding outliers to measurements. Other factors that cause variation include process scheduling and other activity in the system.

Despite the variation we believe this way of measuring the execution time to yield meaningful results since it describes the real elapsed time of the checkpoint creation. Our large number of measurement samples should enable us to see the scale and direction of change of the time over our measurement period. Spurious outliers resulting from the way of measurement should be discernible.

We measured the checkpoint image size by simply calculating the size of the image directory of each checkpoint created. This tells us the size of the data that is needed to restore the execution of the preempted processes but does not include the

[33] Android Studio, Create and manage virtual devices. `https://developer.android.com/studio/run/managing-avds#system-image`

external data of the tested application. Thus the real-world size of the checkpoint would be larger.

The image directory contains the saved state of the checkpointed processes system resources including the contents of its memory pages. Notably, it does not contain the files the processes are accessing during their execution. This means that data files, named pipes and file-backed shared memory are excluded from the measurement.

This concludes the discussion of our test implementation and the environment we used. In the following section we will present the results of the tests we performed.

5 Results

In this section we report our findings of checkpointing the verification tools we introduced in the previous section. We will first discuss how CRIU succeeded in checkpointing the tools overall and what we discovered. We will then proceed to the performance analysis and describe our results of the checkpoint creation time and checkpoint image size measurements.

Our tests revealed that CRIU can successfully checkpoint and restore the AFL task. Checkpointing and restoring the Valgrind task was successful after a minor modification of the CRIU tool to not require restored processes to have a vDSO segment mapped. Checkpointing the AddressSanitizer task was not successful and requires more significant changes to how CRIU processes VMA areas. The latter is a known issue in CRIU[34] and the former we reported to the developers[35].

We discovered that all of the tasks require some preparation when they are executed in order to either contain or ignore certain resources that prevent the task from being checkpointed by CRIU. To make sure checkpointed processes can be successfully restored CRIU will not checkpoint tasks that share resources with processes outside the checkpoint. CRIU will also not checkpoint processes that use resources with no ownership.

In the applications we tested resources such as System V shared memory segments, UNIX sockets, I/O streams and terminals required measures to enable processes using them to be checkpointed. With the help of Linux namespaces and some commonly available utilities this was not difficult to achieve but the required operations make it necessary to have higher privileges to start the tasks.

We also found reliable restoration of the checkpointed processes to require consideration. The first problem is the possibility of a PID clash unless the processes are restored within a PID namespace. The second and more difficult problem is that CRIU cannot ensure the consistency of the file system when the processes are restored which is an issue for tasks that access files during their execution. We will next discuss results of the individual verification tasks and their checkpointability in more detail.

5.1 American Fuzzy Lop

The AFL tool was successfully checkpointed and restored. Checkpointing of the tool required a few steps to contain the System V shared memory segment it uses but this was not difficult. Restoration of the checkpoint was reliable and the tool continued execution without interruption. Challenges in the preemption of this tool are in the high privileges required for creating namespaces and because files accessed by the tool need to be available in the same location when execution is resumed.

We tested the tool as described in section 4.2. As mentioned, we noted that the

[34]CRIU Issue: CRIU can't dump huge mappings. `https://github.com/checkpoint-restore/criu/issues/392`

[35]CRIU Issue: Unable to restore processes with vDSO unmapped. `https://github.com/checkpoint-restore/criu/issues/488`

tool utilizes a System V shared memory segment to enable communication between the tool and the system under test. To enable the shared memory segment to be checkpointed the tool was executed in a Linux IPC-namespace. This enabled the tool to be successfully checkpointed.

After a checkpoint was successfully created we tested its restoration. The restoration was without trouble and the tool continued execution. We used the `afl-whatsup` utility to verify the liveliness of the AFL instances after the restoration and observed the performance counters to be changing. We concluded that all instances continued without interruption and the preemption and restoration of AFL was successful.

We noted two main challenges in the checkpointing of this tool. The first is that creating the IPC-namespace needed requires high privileges. The same applies for a PID-namespace required for reliable restoration of the checkpoint to avoid a clash of PIDs. The former implies that the tool needs to be started by a privileged process. Since high privileges are already required for restoring checkpoints the latter issue is not as significant.

The second challenge is to ensure the files accessed by the tool during its execution are available in the same location when execution is resumed from the checkpoint. Since AFL accesses files during its execution to record information of the fuzzing inputs these files need to be present when the checkpoint is restored. These files are located in the working directory of the tool and need to be available in the same location after restoration.

5.2 AddressSanitizer

Checkpointing of an AddressSanitizer instrumented application was not successful. AddressSanitizer utilizes sophisticated features of the kernel's memory mapping API to be able to allocate a large part of the application's address space as shadow memory. At this time CRIU does not take into account this kind of memory allocations and fails to process the excessively large shadow memory areas. We found this to be a known limitation of CRIU.

We executed the Stress-ng tool with AddressSanitizer instrumentation as described in section 4.3. Attempting to checkpoint the running process failed with an error from CRIU as shown in figure 8. The error states that CRIU has attempted to allocate an exceedingly large amount of memory. We investigated the cause of this kind of error from the source code of CRIU and determined that this memory is required to process metadata of AddressSanitizer's shadow memory areas.

```
Error (criu/pagemap-cache.c:54): pagemap-cache: pmc_init: ↩
Can't allocate 30064246792 bytes
```

Figure 8: CRIU unable to checkpoint AddressSanitizer.

AddressSanitizer allocates the shadow memory in such way that the memory areas are not reserved physical memory. This in conjunction with memory overcommitting[36]

[36] As documented in `proc(5)` of Linux man-pages, version 4.04

enables the shadow memory to exceed the size of physical memory in the host but does not guarantee how much of the memory can be written to. In our system the largest of these areas was 15TB in size while the total size of the instrumented application's VMAs was 22TB.

To verify how the shadow memory segments are allocated we looked into the source code of the AddressSanitizer run-time library. We found that the shadow memory segments are allocated with the `MAP_NORESERVE`[37] option of `mmap(2)` that instructs the kernel to not reserve swap space for the VMAs. This combined with memory overcommitting enables the reserved shadow VMAs to exceed the physical memory, but will cause a segmentation fault if more data is written to these areas than there is memory physically available.

CRIU fails to create a checkpoint of AddressSanitizer because it does not take into account that VMAs may not be backed by physical memory and therefore can be excessively large. CRIU attempts to allocate a buffer for caching the pagemap[38] entries of the application's VMAs. The size of this buffer would be 30GB (as seen in figure 8) that exceeds the available memory in our host.

Pagemap is a data structure exposed by the Linux kernel that presents metadata for every memory page of a process. Most importantly, for pages backed by physical memory the pagemap contains the number of the corresponding physical page frame. This is illustrated in figure 9. For every page of memory there is a 64 bit long entry in the pagemap.

We studied the source code of CRIU to determine whether the pagemap cache (PMC) functionality can be disabled to enable processing large VMAs without allocating prohibitive amounts of memory for the PMC. We found that there exists a configuration variable `CRIU_PMC_OFF` used for a Linux kernel bug workaround, but this setting unfortunately does not affect memory allocation of the PMC. Memory is still allocated for the PMC even when this option is set.

We concluded that because of how the pagemap processing is currently implemented in CRIU a nontrivial amount of changes is required to make tasks with very large VMAs checkpointable. We also found out that this is an already known issue[39] reported to CRIU developers, but at the time of writing the issue is still outstanding.

To have more evidence of the root cause of the failure we also verified that the size of the PMC buffer CRIU attempts to allocate corresponds with the expected memory size of holding the pagemap entries of AddressSanitizer's largest VMA. We performed this by tracking the size of the memory allocations of AddressSanitizer with the `strace` utility. We then calculated how much memory the pagemap entries the largest allocated VMA would require and compared this with the allocation we saw from CRIU.

With `strace` we tracked memory allocations with the `MAP_NORESERVE` option set and selected the largest (15 392 894 357 504 bytes). A VMA of this size would require

[37] As documented in `mmap(2)` of Linux man-pages, version 4.04

[38] Pagemap, from the userspace perspective. `https://www.kernel.org/doc/Documentation/vm/pagemap.txt`

[39] CRIU Issue #392: CRIU can't dump huge mappings. `https://github.com/checkpoint-restore/criu/issues/392`

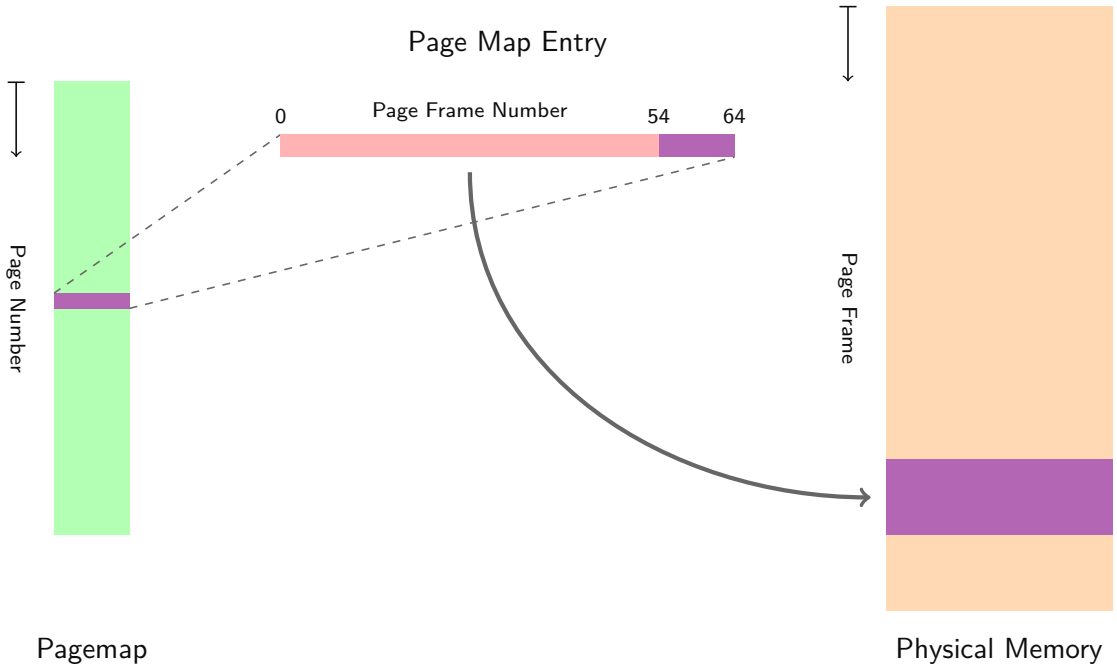


Figure 9: The pagemap provides metadata of pages and maps them to physical page frames.

3 758 030 849 page frames on our host with 4096 byte page size. Since based on the Linux kernel documentation the pagemap entries consist of 64 bits for each page frame this means that the pagemap entries of this VMA require 30 064 246 792 bytes of memory. This corresponds with the size of CRIU's failed memory allocation we observed.

5.3 Valgrind

Checkpointing of Valgrind was successful but required a small modification of the CRIU tool to enable the checkpoint to be successfully restored. When a checkpoint is restored CRIU currently expects all processes to have a vDSO segment mapped. However, Valgrind unmaps this segment from its address space which results in its checkpoints to not be restorable. We also noted that Valgrind uses named pipes and shared memory files during its operation requiring these files to be available when the checkpoint is restored.

We tested the tool as described in section 4.4. Creating a checkpoint of the tool was without trouble. However, attempting to restore the processes resulted in failure and an error message from CRIU. The error is shown in figure 10. The error states that CRIU could not find a vDSO segment in the memory image of a process to be restored.

The vDSO (virtual dynamic shared object) is a shared library mapped to every processes' address space by the Linux kernel. The purpose of this library is to avoid system call context switches when calling kernel functions that are read-only and can be implemented entirely in user space. Functions in the vDSO segment read the

```
pie: 7: Error (criu/pie/parasite-vdso.c:252):                    ↩
vdso: Can't find vDSO area in image
pie: 7: Error (criu/pie/restorer.c:1689): Restorer fail 7
```

Figure 10: CRIU unable to restore Valgrind.

data they require from another read-only segment called *vvar* the kernel exposes [8].

By studying the source code of Valgrind we verified that Valgrind unmaps the vDSO segment. The start address of the vDSO segment in the processes' address space is identified by the `AT_SYSINFO_EHDR` entry in the auxiliary data vector the kernel passes to every program it executes[40]. In Valgrind the code that processes this entry simply unmaps the segment. From the version history of the source code of Valgrind we discovered the reason to unmap the vDSO is because its position is randomized and therefore it can cause issues with analyzing programs that require control over the position of their memory mappings[41].

We also studied the source code of CRIU to determine the root cause of the checkpoint restoration failure. We identified that CRIU expects to find a vDSO segment in a checkpoint image and terminates otherwise. This check is part of the vDSO proxification CRIU performs when it restores a checkpoint. The vDSO proxification is performed for all restored checkpoints and currently there is no option to bypass it.

vDSO proxification is a feature of CRIU that enables checkpoint images to be restored under a different Linux kernel version that potentially has a different internal layout in the vDSO exposed to programs[42]. Based on the documentation of CRIU if CRIU detects the vDSO segment of the restored process is different from the vDSO of the running kernel CRIU modifies the restored vDSO to be a trampoline that invokes functions at their locations in the new kernel's vDSO.

Because creating the checkpoint of Valgrind was without trouble we investigated if the vDSO proxification feature can be disabled altogether, without excessive effort, to enable us to evaluate the overall feasibility of checkpointing and restoring Valgrind despite this shortcoming of CRIU. By making a small modification to the checkpoint restoration procedure we successfully disabled the vDSO proxification which enabled Valgrind to be successfully restored.

Since Valgrind unmaps the vDSO it shouldn't be an issue that no proxification is performed when the checkpoint is restored. However, since the kernel maps the vDSO for all new processes it can be a problem for Valgrind, that has explicitly unmapped the vDSO, to have it mapped again in the new process restored from the checkpoint. Therefore for reliable operation a fix would be needed to CRIU.

We also noted during the testing of Valgrind that it creates several named pipes and shared memory files under the `/tmp` directory. For successful restoration of the

[40] As documented in `getauxval(3)` of Linux man-pages, version 4.04

[41] Valgrind Commit. `https://sourceware.org/git/?p=valgrind.git;a=commit;h=3a004915a2cbdcdebafc1612427576bf3321eef5`

[42] CRIU, Vdso. `https://criu.org/Vdso`

processes these files must be available in the same location.

5.4 Android Emulator

Checkpointing of the Android Emulator was unsuccessful because of its use of `userfaultfd(2)`. At this time CRIU does not support checkpointing of this kernel feature that is used by QEMU to enable fast live migration of guest software. Enabling the Android Emulator to be checkpointed would require either CRIU to support checkpointing userfaultfd or disabling post-copy migration support in QEMU.

Userfaultfd is a recently introduced feature of the Linux kernel that enables applications to process their page faults in user space. The main users of this feature are currently QEMU and CRIU that use it for enabling migrated applications to start execution before all of their memory pages have been transferred to the destination host. Userfaultfd enables these tools to be notified when a migrated application accesses a page that has not yet been transferred[43] [13].

Applications use userfaultfd by calling the `userfaultfd(2)` system call[44] to create a userfaultfd file descriptor. The application reads the file descriptor to receive an event when a page fault occurs for a memory range the application has registered. Memory ranges are registered using `ioctl(2)` on the userfaultfd file descriptor.

In QEMU userfaultfd is used for post-copy memory copying in the live migration of guests[45]. QEMU gets notified of page faults in its guest system enabling it to prioritize the retrieval of a page before its access takes place. The use of userfaultfd is enabled for all Linux builds of QEMU (and therefore the Android Emulator) and there is no runtime option for disabling this feature.

We observed the failure to checkpoint the Android Emulator in the error message of CRIU shown in figure 11. We can see that CRIU is unable to process the file descriptor of userfaultfd type which implies that checkpointing applications utilizing userfaultfd is not supported at this time.

```
(00.058807) Error (criu/files-ext.c:96): ↩
Can't dump file 26 of that type [600] (anon anon_inode:[userfaultfd])
```

Figure 11: CRIU cannot checkpoint userfaultfd file descriptor.

To enable CRIU to checkpoint the Android Emulator it needs to be able to save the state of the emulator's userfaultfd object. This situation is complex because we are effectively using an application that performs process migration on an application that also performs process migration. Because CRIU itself is already using userfaultfd to enable post-copy migration of processes there is additional complexity to enable the nested use of userfaultfd.

[43] CRIU, Userfaultfd. `https://criu.org/Userfaultfd`

[44] As documented in `userfaultfd(2)` of Linux man-pages, version 4.16

[45] QEMU, Features/PostCopyLiveMigration. `https://wiki.qemu.org/Features/PostCopyLiveMigration`

Supporting the nested use of userfaultfd is not achievable at this time because it requires additional kernel support. Corbet has stated that this remains an open issue [12]. For CRIU we did not find information about future plans of implementing checkpointing of userfaultfd. A possible alternative solution would be to disable migration support in QEMU since this functionality is provided by CRIU in our use case. Because there is no runtime option for this in QEMU this would require code changes.

5.5 Performance

For AFL and Valgrind that were checkpointable we did performance measurements as presented in section 4.6. We found the time to create a checkpoint to have a linear correlation with the checkpoint image size. The largest files in the checkpoint images were the contents of the memory pages of the processes. We can conclude that the performance of creating a checkpoint is mostly dependent on the size of the verification tool's allocated memory that agrees with previous research [29, 33]. Overall, we observed good performance without notable issues.

In our measurement AFL showed seemingly logarithmic growth for the checkpoint creation time and the size of the image throughout the measurement period. We observed the creation time stay below 200 ms and the image size reach approximately 100 MB. Since the measured times were in 100-millisecond range they showed a high variance due to random effects of the system that are visible at this scale. We also found the checkpoint creation time to seem to correlate linearly with the image size.

Graphs of the measurements of AFL are presented in figures 12 and 13. In figure 12 we have omitted the first measurement of the checkpoint creation time. This particular value was a significant outlier at 621ms. Although a quite high value, we believe this outlier was likely caused by a combination of the operating system's I/O caching, disk latency and scheduling. On the first execution the checkpointing utility needs to be loaded from disk which causes latency. On subsequent executions much smaller values were observed.

Checkpoint creation time of AFL increased from 100 ms to 160 ms approximately, staying below 200 ms. The growth of the values seemed to be logarithmic. High variance of 40 ms – 50 ms was observed in the values.

We attribute the variance in the checkpoint creation time to noise caused by the measurement technique. Our time measurements are subject to the effects of the operating system's I/O caching, scheduling and interference from other activity in the system. Because CRIU's execution times in this test are in the scale of 100 ms – 200 ms these random variations are visible.

The size of the checkpoint image of AFL we observed to increase from approximately 25 MB to 100 MB. Similar to the time measurement the growth in these values seemed logarithmic. Since the the largest contributor to the size of the checkpoint images is the memory size of the processes the logarithmic growth is likely the result of how AFL internally uses memory.

For Valgrind our measurements showed linear growth for both values. Checkpoint creation time increased up to approximately 1440 ms and image size up to approx-

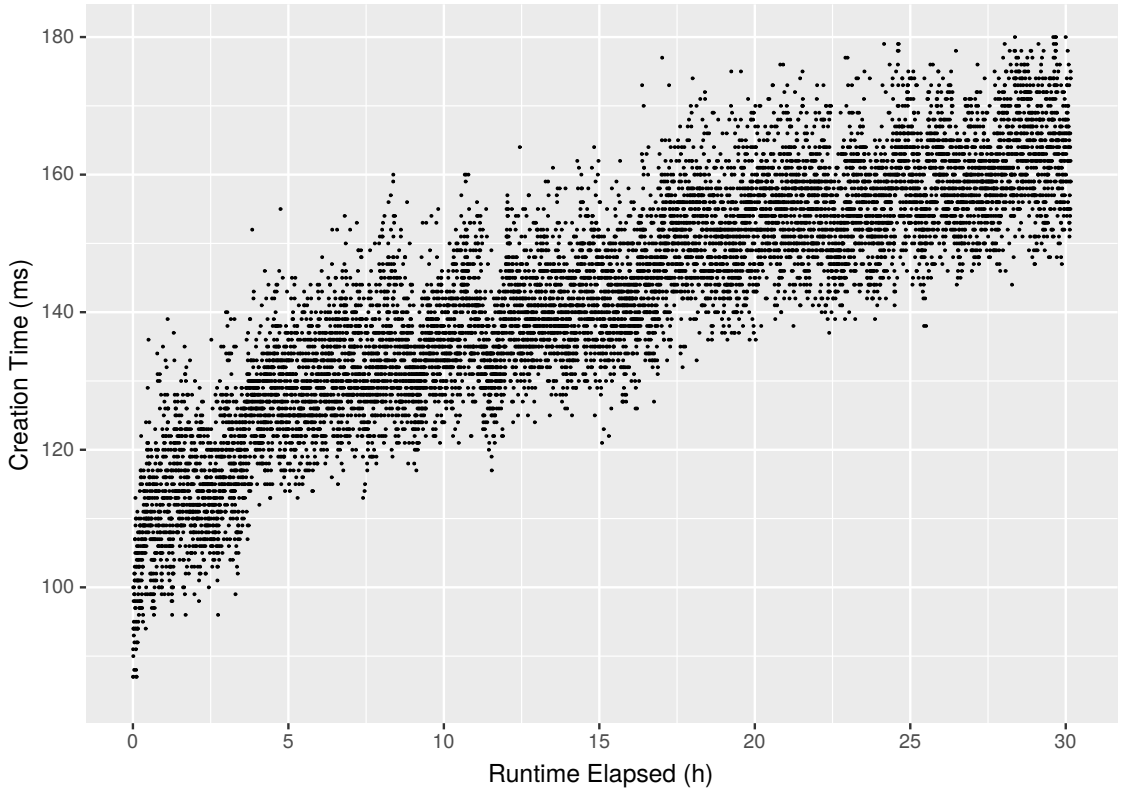


Figure 12: Checkpoint creation time, AFL

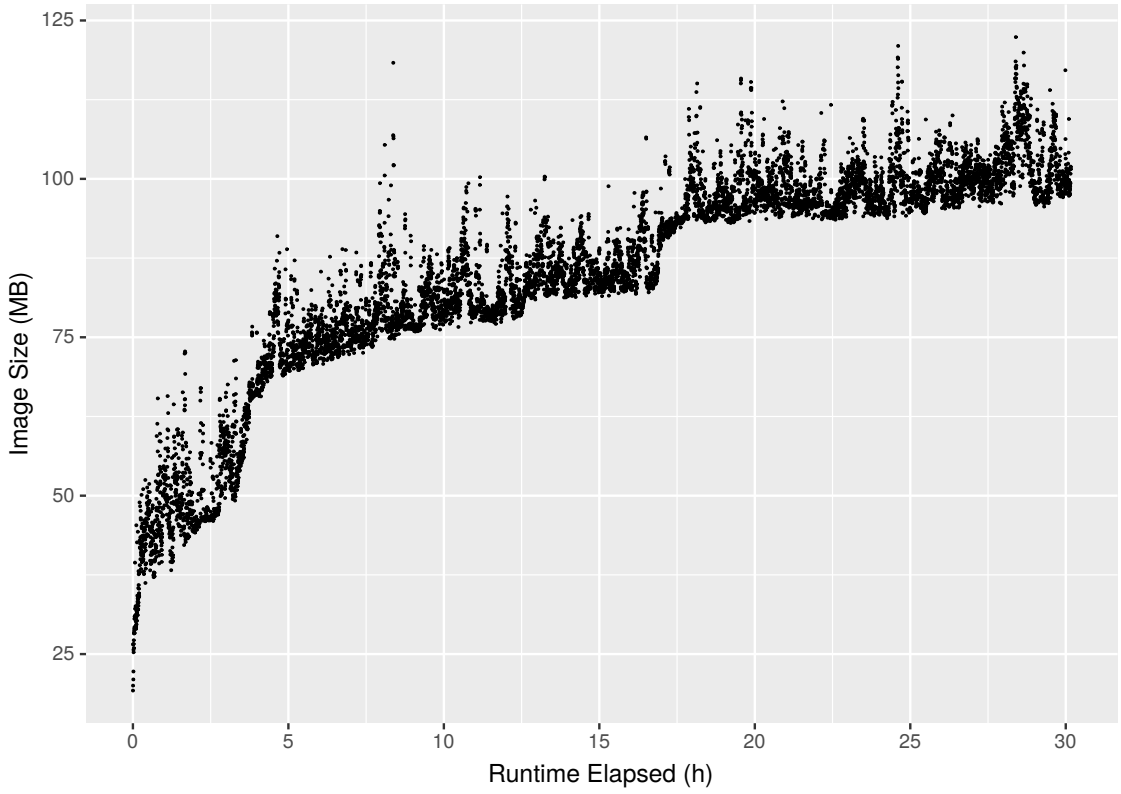


Figure 13: Checkpoint image size, AFL

imately 1413 MB. Both settled to a constant level during the last 6 hours of the measurement.

The measurements of Valgrind are presented as graphs in figures 14 and 15. In figure 14 we can see a similar outlier than we saw in the results of AFL. Since its value of 1286 ms fits within the range of the other values it was left visible.

Creating a checkpoint of Valgrind took between 750 ms to 1440 ms. The times seemed to grow linearly until the end of the measurement. For the last 6 hours the values remained at a constant level only showing random variance. Variance in the values was small. The checkpoint images of Valgrind grew in size with a similar linear progression. The sizes increased from 1150 MB to 1420 MB. There was little if any random variance in the sizes of the images.

5.6 Analysis

Our results show that verification tools are a difficult target for CRIU. Most of the tools revealed limitations in CRIU and only two out of four could be checkpointed with reasonable effort. Preemption of arbitrary verification tools by checkpointing is therefore not easily achieved at this time. We also noted that many of the steps require high access privileges which is a challenge for integrating preemption to CD-platforms. In addition, files accessed by the verification tools need to be transported with the checkpoint which requires additional tools to be practical. Lastly, some tools need to be prepared for checkpointing by executing them in a namespace.

From our results we can deduce that to checkpoint verification tools CRIU has to not only support checkpointing of many different kinds of system resources but it also needs to take into account the many options that change their semantics. There are also ways to use system resources that make their checkpointing complicated. With AddressSanitizer we saw that options of the `mmap` API make it possible to allocate more memory than physically available, which CRIU does not take into account. With Valgrind we found that it unmaps its vDSO memory segment, usually present in all applications, leading to an unexpected situation for CRIU. Checkpointing of userfaultfd used by Android Emulator was not supported at all.

We found that most of the verification tools we tested are using resources that are complex to checkpoint. Processes have a large surface area consisting of many kinds of system resources and the state of each of them needs to be processed in a different way. Our results confirm our expectation that verification tools are even more troublesome than ordinary applications because they utilize uncommon and sophisticated kernel features. Our tests of CRIU show that the feasibility of checkpointing verification tools is at this time largely dependent on the tools in question. Preemption of arbitrary verification tools is prevented by the number of resources that are not well supported by CRIU.

When we performed our tests we noted that many of the necessary commands require high privileges. Creating and restoring checkpoints itself require high privileges. However, privileges are also required for creating namespaces that isolate the processes' resources. This was needed for starting the AFL tool since it required an IPC-namespace. High access privileges are a problem for integrating preemption

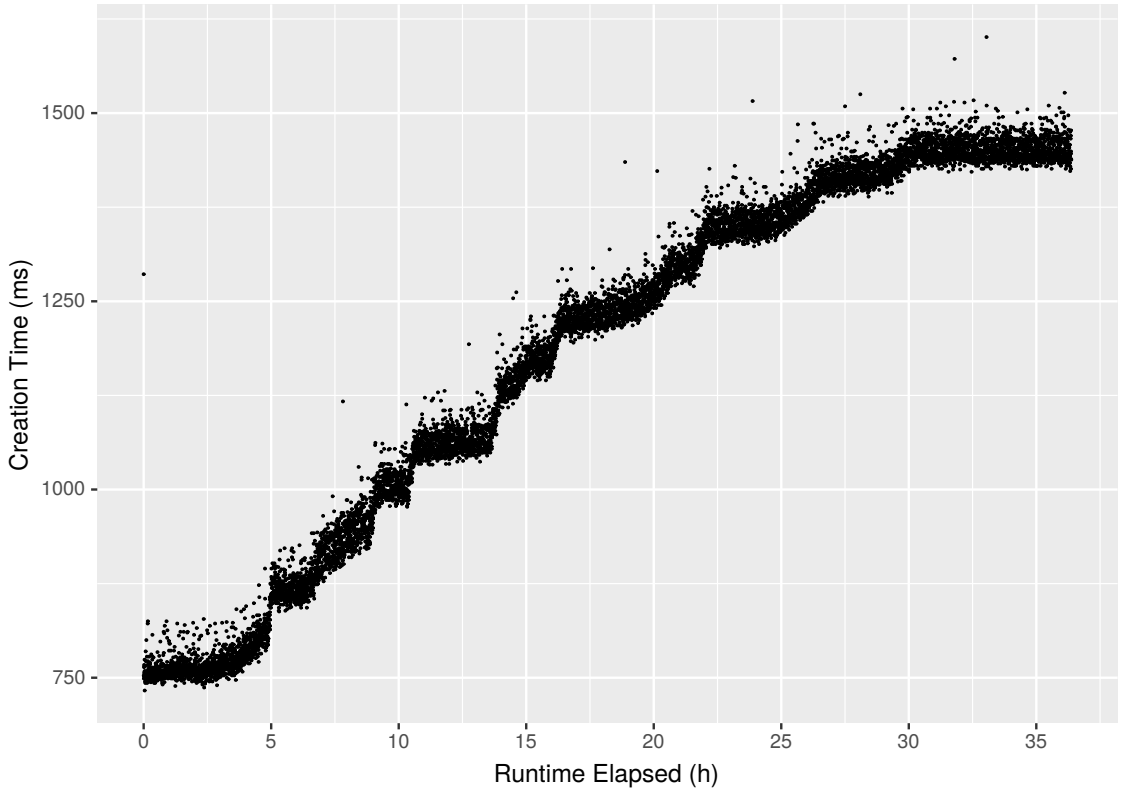


Figure 14: Checkpoint creation time, Valgrind

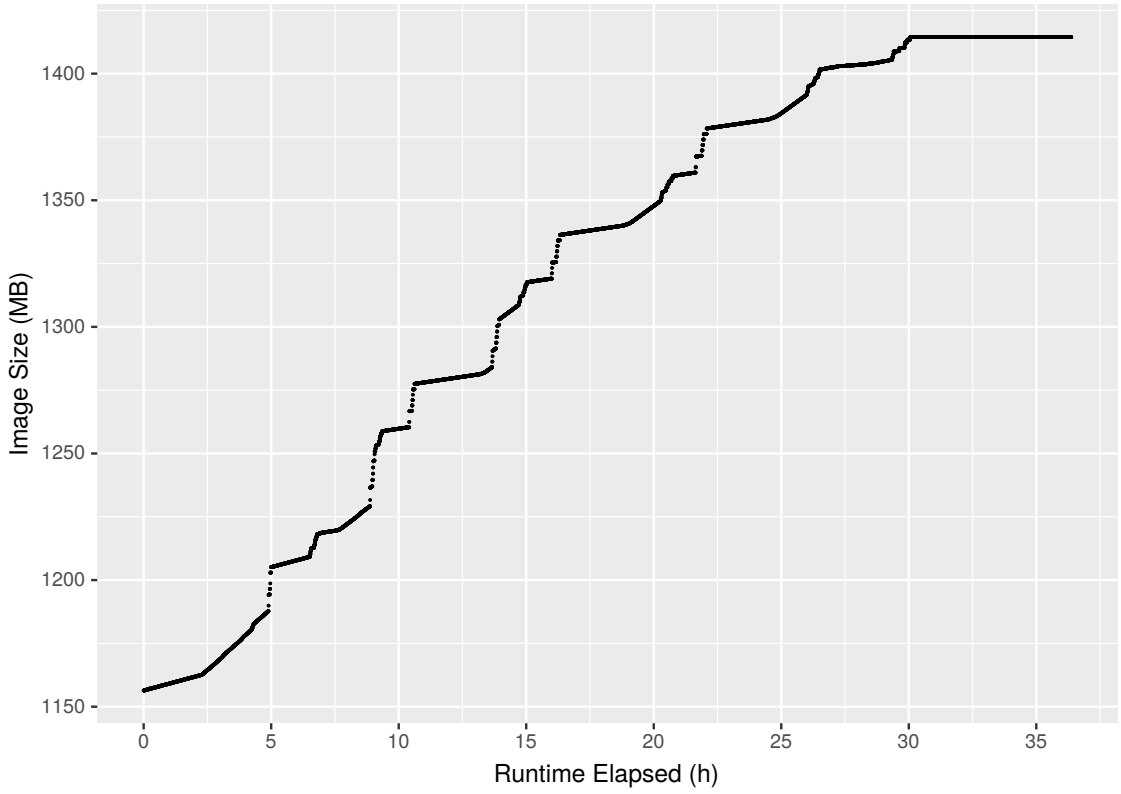
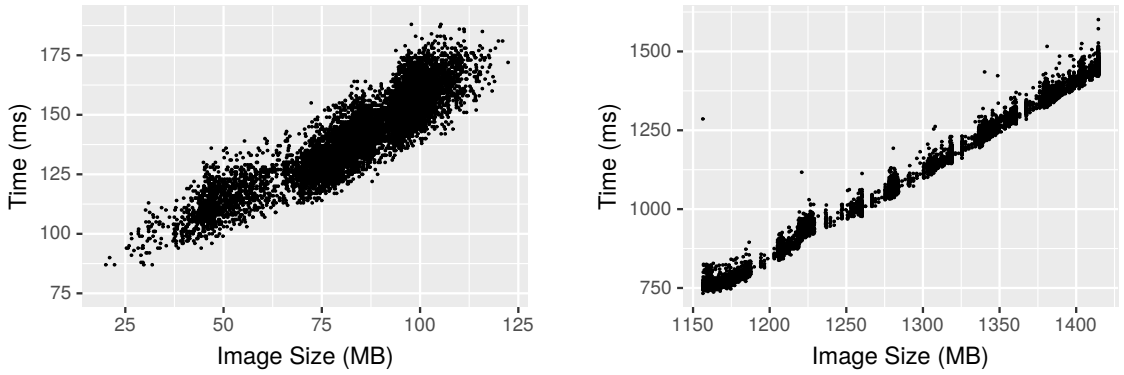


Figure 15: Checkpoint image size, Valgrind



(a) AFL (b) Valgrind

Figure 16: Correlation of checkpoint image size and checkpoint creation time.

with CRIU into an existing CD-platform.

Files accessed by the tools pose another problem for checkpointing. To be able to resume the execution of the verification tool the files it accesses need to be available in the same location they were in before the processes were checkpointed. This not only concerns data files of the verification tool but also temporary files, named pipes, and files used as shared memory that are stored in the file system. The issue with these files is that the tools place them in various places. For example, AFL keeps its files in the working directory whereas Valgrind places them in `/tmp`.

We also noted that there exist resources, such as the System V shared memory used by AFL, that need to be isolated in order to enable the tool to be checkpointed. This requires that the verification tool is executed in a namespace.

Our performance measurements did not reveal any particular issues with checkpointing AFL and Valgrind. We saw the checkpoint creation times correlate linearly with the checkpoint image size. This implies the memory size of the verification tools has the greatest effect on the checkpointing performance. The checkpointing times of the tools were in the range of 200 ms and 2 s respectively which is usable in CD-systems.

The relationship of the checkpoint image size and creation time is illustrated in figure 16. We can see a strong linear correlation between these variables. Creating a checkpoint involves saving the memory contents of the processes into the checkpoint image which is written to disk. Therefore the size of the processes' memory areas has a significant effect on the performance of checkpointing. This is consistent with previous results of Duell, Laadan and Li [14, 27, 29]. For instance, for another checkpointing system Laadan found that 80% of the checkpointing time was taken by the saving and restoring of the processes' memory contents [27].

The scale of the checkpoint creation times means preemption by checkpointing is feasible in an interactive CD-system. Comparing to the 10 minute maximum acceptable build waiting time found by Laukkanen the preemption times of 200 ms to 2 s are not significant [28].

In our results we saw outlying measurements that suggested the effects of the

operating system's I/O caching and scheduling. In our results we ignored these outliers since they do not describe the performance of the checkpoint creation itself. However, this kind of outliers do raise the question of whether in the real-world use of CRIU for preemption the interactivity would be negatively affected by the time it takes to load CRIU itself. If we expect preemption to be an uncommon operation the tool will not be found in the operating system's cache. Investigating this effect would require specifically measuring the checkpointing performance with an empty cache.

The size of checkpoint images not only affects the performance of creating the checkpoint. When a task is preempted in a cluster we may want to resume it in another host. This means that the images need to be transferred over network and therefore their size affects performance. For this reason it can be worthwhile to compress the images before transferring them. Since most of the data in the images is taken by the processes' memory contents the compressibility of the images largely depends on this data. For example, we could compress the checkpoint images of AFL and Valgrind with factors of 90% and 97% using LZMA-compression.

We have now presented our findings from the tests and measurements we performed. We discussed how some of the verification tools were not successfully checkpointed due to shortcomings in CRIU. We also showed that the performance of checkpointing depends on the verification tool's memory size. The interpretation and implications of these results we will discuss in the next section.

6 Discussion

Our results show that system-level checkpointing of verification tools is possible, but there are still considerable limitations in the capabilities of the CRIU checkpointing utility tested. This hinders its adoption to wider use. The performance of checkpointing verification tools with CRIU was found suitable for interactive use in clusters. Previous research was found to be in favor of our choice of system-level checkpointing and supported our use-case of preemption for improving scheduling interactivity and resource utilization. We found the previous research on checkpointing to consider mostly its use-cases in improving fault-tolerance and availability. Our research contributes to widening the use-cases to achieving interactivity and higher resource utilization.

6.1 Preemption of Verification Tools with CRIU

Based on our results it is possible to implement preemption of verification tools using CRIU. However, we saw in our evaluation that two out of four tested verification tools could not be checkpointed due to limitations in CRIU's capability to checkpoint certain resources. The number of verification tools we could test was necessarily limited so making strong claims of the overall checkpointability of verification tools is not possible. However, our result indicates that at this time it's likely that many verification tools will not be preemptible with this method.

To make checkpointing as a preemption mechanism acceptable for wider adoption, outside using it with a few selected verification tools, effort needs to be put into making the checkpointing utility support all available system resources. This means also less commonly utilized resources and their different configurations (eg. memory overcommitting) need to be taken into account by the utility. Comprehensive capabilities are especially important if checkpointing is to be used in a CD-platform offered as a service, since users will be upset if their jobs are terminated due to a failed preemption.

The capabilities of checkpointing have been discussed in previous research, but most of the research has focused on its use with computational and service applications and exploring the limits of the technology has not been the main focus of the research. These applications have been discussed by eg. Huang and Nadgowda [24, 33]. Litzkow has stated that supporting all resources has not been an attainable goal for the Condor system and effort has been on supporting only the most important kinds [30]. However, Condor is a library-level solution so its capabilities are necessarily more limited than those of system-level solutions.

Other research shares the same focus and sentiment. For the BLCR system Duell states that supporting all kinds of program behaviour is not feasible for the project since its focus is on scientific applications [14]. Roman points out the difficulty of implementing system-level checkpointing and discusses checkpointing of a number of system resources [38]. For the Linux-CR system Laadan mentions the possibility of comprehensive resource support, but does not delve further into the issue [27].

We found little research on CRIU's usability with verification tools. A single

report by Klode discusses checkpointing the Android Emulator [26]. Klode had successfully checkpointed the emulator and our findings are contrary. However, since the userfaultfd support has been added to the Android Emulator only recently the previous results do not hold anymore for the new versions of the emulator explaining the different outcomes.

Another significant observation we made is that successful checkpointing requires the verification tools to be isolated from other processes in the system. This is because the tools cannot be checkpointed if they use resources that are global in the system or shared with processes outside the checkpoint. For the tools we tested these resources included System V IPC, sockets, terminals and also importantly the file system. These resources can be isolated using kernel namespaces.

Using namespaces enables isolating processes from each other but requires the namespaces of the right kind to be created when the verification tool is launched. In practice, implementing this isolation of the processes is very similar to what container runtimes such as Docker[46] and LXC[47] provide. Therefore it is probably most convenient to execute the verification tools in containers and use the container runtime's commands to execute CRIU.

The use of containers will not change the checkpointing procedure much. The container runtimes will internally use CRIU to checkpoint the processes in the container. The container runtime creates namespaces to isolate the resources of the processes which we did manually. The file system bundling (or layering) provided by the container runtime will make it easier to contain the files accessed by the tools. Checkpointing of containers with CRIU has been evaluated recently by Huang, Nadgowda and Qiu [24, 33, 36]. At this time, experimental support for container checkpointing is supported by both Docker[48] and LXC[49].

Generic preemption of verification tasks by checkpointing does not seem feasible at this time but an intermediate solution could enable preemption in CD-platforms. Checkpointing can be used for tasks that support it and tasks that don't can be terminated. This would enable some of the benefits of preemption, such as shorter response times and flexibility to execute work when the system is idle. For tasks that support checkpointing the preemption would be work saving and efficient.

To integrate preemption using CRIU into a CD-platform would require changing the scheduling model of the platform to support job preemption. The underlying cluster scheduler also needs to support preemption and the cluster management system needs to be able to migrate and resume the preempted tasks. Since CRIU checkpoint creation and restoration require high privileges the cluster management system needs these privileges to be able to preempt tasks.

In this thesis we tested CRIU on a single host only. However, in an actual computation cluster the preempted processes would likely be migrated to another

[46]Docker. `https://www.docker.com/`

[47]Linux Containers. `https://linuxcontainers.org/`

[48]Docker checkpoint. `https://docs.docker.com/engine/reference/commandline/checkpoint/`

[49]lxc-checkpoint(1). `https://linuxcontainers.org/lxc/manpages/man1/lxc-checkpoint.1.html`

host to be resumed. This means that there can be differences between the host where the checkpoint was created and where it is restored. For a practical implementation in a cluster CRIU needs to be able to account for these differences reliably.

A limitation in our tests regarding the reliability of the checkpoint restoration is that we did not test restoring the tasks in different systems. If the checkpoints are dependent on the underlying platform they may not be restorable in heterogeneous hosts. In computation clusters the hosts are likely to be similar, but differences may still exist due to eg. different hardware versions and resource availability. In previous research CRIU has been tested for migration purposes but we did not find results of its support of heterogeneous platforms.

6.2 Performance of CRIU

The performance of CRIU we observed seems to indicate it is suitable for integration to interactive CD-platforms. Creating a checkpoint of the AFL task was in the range of 75 ms to 175 ms and 750 ms to 1440 ms for the Valgrind task. These are short compared to the optimal verification task execution time of 2 minutes proposed by Laukkanen [28]. Therefore based on these figures the execution time overhead of preemption should not constitute a significant latency for the verification task execution.

In interpreting the results of the performance measurement we need to take into account what we measured is only the time it takes to create the checkpoint image. In a real cluster the actual delay caused by task preemption would include also the time spent transferring the checkpoint image and the time to restore the checkpoint. The memory size of the verification tool and the size of the tool's data files have the greatest effect on the transfer time. We assume the restoration time to be similar to the checkpoint creation time.

In integration to a CD-platform the transferring of the checkpoint image and the tool's data files is likely going to cause the greatest latency since this is performed over a network. The size of the verification tool's memory can be a problem since some tools use it in significant amounts. On the other hand, the size of the tool's data files are probably not often a problem since test tools and the tested executables are usually small to enable fast execution. However, in some cases, such as with the Android Emulator and the system image it requires, there can be a large amount of data to be carried with the checkpoint.

By making a few assumptions and using the maximum latency figures we saw we can try to estimate the performance of real-world use of preemption by checkpointing. Assuming the checkpoint resumption time is equal to the creation time we can estimate the creation and resumption to take approximately 3 seconds for a 1400 MB image. The size of the data files of the verification tools we assume to be negligible. What is left is the transferring time of the checkpoint image that we, in a 1 Gbit/s local-area network, approximate to be 11–22 seconds depending on whether the image is transferred directly to the destination host or first to intermediate storage. This estimate also does not account for speed of the disks.

With the assumptions we made we can estimate the total delay of preempting a

process with 1400 MB of memory to be in the range of 20–50 seconds. The transferring of the checkpoint image is the largest contributor to this estimate. Compared to Laukkanen's optimal 2 minute response time this is significant [28]. What can be concluded from this is that the checkpoint image transfer times have the greatest effect on the delay the user experiences due to task preemption.

A task-switching latency of nearly a minute would be harmful for the user-experience of a CD-system. However, many verification tools probably have much smaller memory usage than what we used for this estimate. Also, the estimate is for the total time the task preemption would require, but in reality a new task can begin execution right after the previous task has been evicted. By using the process migration feature of CRIU, transferring the image directly to the destination host and compressing the data in transit the task switching time could be in many cases reduced further.

To compare our results to those of others we can look at the results of Li, Nadgowda and Huang that seem to be measured in environments not too different from ours. Li has measured preemption with CRIU and Nadgowda and Huang process migration with CRIU likewise.

Li has measured the checkpointing of a process with 3 GB of memory to take approximately 50 seconds. Li's measurement was performed to a local SSD-disk and did not include transferring the image [29]. Li's results seem to be significantly different to ours, but this is likely explained by a different measurement technique.

If we calculate the data rate of saving the checkpoint image we get 120 MB/s for Li's result and 1000 MB/s for ours. In our measurement CRIU benefits of the operating system's I/O-caching and therefore the checkpoint image will be written into the cache in memory. As the checkpoint image is deleted between measurements this data may not be written to disk at all. Li does not mention the exact way their result is obtained, but since their results explicitly mention the type of storage medium used the measurement most likely includes writing of the data to the storage. This agrees with the 120 MB/s data rate, which is realistic for an SSD-disk of the time.

Nadgowda has measured process migration with CRIU. Nadgowda's results for a process with 250 MB memory footprint showed an approximately 0.7–0.8 second response time for checkpoint restoration [33]. If we make similar assumptions than with our earlier estimate, include the image transfer time and extrapolate Nadgowda's results for a 1400 MB image we can get a total time of approximately 20 seconds. This is in the same scale with our results.

In research by Huang the average checkpoint creation time was measured to be approximately 2200 ms. However, the measurement was made with Docker containers and the storage medium used was NFS network file system. The size of the checkpoint image in Huang's checkpoint creation time measurement is not evident [24]. The times observed by Huang are likely affected by the use of Docker and the overhead of network attached storage, so our results are not directly comparable. However, the scale of the results is similar to ours.

As Mårtensson describes, time consumption of the software delivery process is one of the most important factors in how developers use continuous integration [31] that

means it is very important that the CD system provides feedback without additional delay. However, the results of Laukkanen that we use as a benchmark are based on preliminary literature review of scarce material and other than that work there is little research on the interactive usage of CD systems.

To conclude, our measurements indicate that the performance of CRIU is suitable for interactive use in CD-platforms but the real-world responsiveness of preemption depends greatly on the size of the verification tool's memory and data files. Earlier we found that CRIU is usable with some verification tools but it's not yet capable of preempting arbitrary tools. In the next section we discuss the feasibility of preemption by checkpointing in general.

6.3 Preemption by Checkpointing

Preemption by checkpointing was recommended by Arora for enabling long running tasks to be executed in shared supercomputing clusters. The issue checkpointing solves, as Arora describes, is that in these clusters execution time is limited to time slots that are not long enough to execute tasks to completion. Arora suggests checkpointing can be used for saving the state of the tasks in order to resume them when another time slot is available [5].

Arora's work supports our view of using preemption by checkpointing as a tool to make it possible to execute long running verification tasks in an interactive CD-system. The task execution model Arora describes is not too dissimilar from the batch-execution model in current CD-systems. With the help of preemption by checkpointing long verification tasks can be executed in parts without reserving the system's resources for the entire duration of the task. This enables us to limit the run time of tasks to maintain the interactivity of the CD-system. Long running tasks can be preempted when their allotted execution time is up to allow the tasks of other users to execute which enables fair- and interactive service.

Other work we found supporting the use of checkpointing as a preemption mechanism was by Roman. One of the ways we can implement task switching is to suspend- and resume running tasks. However, this leaves the suspended tasks in memory that means that the number of parallel active tasks would be limited by the amount of memory in the system. Roman recommends preemption by checkpointing as a way to free up memory for runnable processes [38]. Preemption by checkpointing frees up all resources of the preempted tasks so that the runnable processes have the full resources of the system available.

Our research was initiated because of the problem that in a shared CD-system we cannot execute long-running verification tasks without either reserving the system for the entire duration or having to terminate the task to free up the system for other work. The disadvantage of terminating the task is the inefficiency and waste of resources because of losing all progress every time.

Because checkpointing saves the progress of the preempted tasks we assumed it will increase the efficiency of task execution compared to if the tasks were simply terminated. Efficient preemption would enable more flexible scheduling techniques to be used, including priority preemption and rescheduling. We discovered several

articles that provide evidence supporting this view. Hargrove, Vavilapalli and Li have stated multiple reasons why preemption by checkpointing can make task execution in clusters more efficient and reduce waiting time.

In a paper describing the BLCR checkpointing system Hargrove points out that preemption by checkpointing can be used for rescheduling tasks to different nodes if a more efficient allocation of tasks to nodes can be reached [21]. In this thesis we did not research the migration of tasks, but this kind of task rescheduling would still be a desirable feature because of the increased efficiency it achieves. Hargrove also states that preemption enables shorter task queuing times by allowing long running tasks to be executed during times the cluster is otherwise not used. This is similar to the argument of interactivity we made based on the article by Arora [5, 21].

In an article describing the YARN[50] cluster resource management framework Vavilapalli has described a case for checkpointing similar to the rescheduling presented by Hargrove. Vavilapalli states that checkpointing is one way task preemption can be performed in the case the cluster scheduler needs to change the resource allocation of active tasks. Vavilapalli states that by using checkpointing it is possible for the cluster scheduler to preempt tasks to maintain scheduling guarantees such as a task queue's maximum share of cluster capacity under contention. Because checkpointing saves the state of the preempted tasks this rescheduling is efficient and gives the possibility to overcommit resources [45].

The work by Li further motivates the demand for work-preserving preemption by checkpointing in shared clusters. Li has presented that the use of work-preserving preemption with the YARN framework can lead to 30% shorter task execution response time and reduce resource waste by 67% [29]. This kind of figures imply that if we would utilize shared cluster platforms, such as YARN, as the platform for large-scale CD-systems there would be significant efficiency gain achievable if the tasks support checkpointing. As we believe that CD-systems are evolving towards execution on shared cluster platforms it becomes more relevant to have a preemption mechanism for verification tools.

In our evaluation we chose to focus on system-level checkpointing. Since our use case is checkpointing of verification tools that don't natively support checkpointing it is important that checkpointing doesn't require modification of the target applications. Several articles we found, including those from Li, Hargrove, Roman and Silva, stress the transparency of SLC and that it should be the easiest checkpointing technique to take into use for existing applications [21, 29, 38, 43]. This supports our choice to focus on SLC as the checkpointing method.

The literature we studied indicates that checkpointing is a feasible and even recommended way to implement preemption. It can be used for enabling interactive scheduling of tasks of different execution times in a shared system. It enables the active tasks to exceed the amount of memory on the host by freeing the resources of preempted tasks. Because it is work-saving, checkpointing can greatly increase the efficiency of scheduling in a cluster by enabling task rescheduling and meeting scheduling guarantees. In addition, we learned that the SLC method is the easiest

[50]What Is Apache Hadoop? `https://hadoop.apache.org/`

to take into use for the verification tasks.

6.4 Scheduling Practices in CD-platforms

Work scheduling in CD-platforms seems to be an issue that has not been extensively studied. In our exploration of CD-platforms we noted that all of the publicly available ones rely on some variation of simple batch scheduling. The scheduling models we saw were more or less similar to the traditional cluster scheduling model that is known to be inefficient and not interactive. Only the proprietary state-of-the-art platforms of Google and Microsoft showed innovation in this field by combining continuous delivery with data-parallel computation and execution in a shared cluster. Preemption was not supported by any of the platforms we studied.

We believe the main reason for the deficiencies in task scheduling in CD-platforms is that most of the recent platforms have been designed primarily for the easy modeling of continuous delivery with pipelines. The interactivity and efficiency of task scheduling have not received the same level of attention. We suspect these systems have been used mostly in small projects and therefore their limitations in scalability, largely caused by their inflexible scheduling models, have not been noticed. However, with growing adoption of continuous delivery in large projects we believe these limitations need to be addressed.

The fact that modern CD-platforms commonly support executing jobs using an external cluster management system means there is an opportunity to improve the job scheduling efficiency and interactivity. However, in the platforms we explored these benefits have not been realized yet. Most of the platforms use the Kubernetes cluster management system which unfortunately supports preemption only by termination. Kubernetes seems to place more emphasis on service jobs for which termination is a suitable preemption method because service processes are treated as disposable.

Based on our literature review and what we found about the current scheduling practices in CD-platforms we believe there is much room for improvement. One key for enabling more advanced scheduling techniques, such as rescheduling and priority preemption is the ability to save the state of running tasks. This reinforced our view that preemption by checkpointing would be a valuable tool.

6.5 Future Research

This thesis provides only a cursory analysis of the feasibility of checkpointing in CD-systems and only for a limited set of verification tools. We uncovered several areas where more research could be conducted. These include improving the compatibility and reliability of CRIU, implementing better scheduling practices in CD-platforms and improving the integration of checkpointing in CD-platforms and container management systems.

To improve CRIU's compatibility with verification tools its support for different kinds of resources and their semantics needs to be improved. We found resources that were unsupported and ways to utilize them that make them not checkpointable. Better compatibility will require effort not only in developing CRIU but it may be

necessary to introduce changes to the Linux kernel to enable certain resources, such as userfaultfd, to be checkpointable. Making CRIU support all these resources is not an easy task because of the large number of them and their configurations. Because of the complex semantics of some resources, dedicated research may be needed for each of them making wide application compatibility an arduous task.

Improving the compatibility of CRIU also requires testing of a wider range of applications. CRIU has been successfully tested with many ordinary applications but these applications are probably heterogeneous in the kind of system resources they use. Verification tools utilize system resources in uncommon ways and seem to be very efficient in revealing gaps and edge cases in CRIU's resource support. It would be therefore useful to test CRIU with more verification tools. Edge cases discovered from these real applications can then be integrated to the comprehensive test suite of CRIU[51].

For ensuring reliable operation CRIU checkpointing and restoration needs to be tested in many environments and configurations. In this thesis we tested CRIU in a single host only and to evaluate its reliability it should be tested in a real cluster environment. We noted that in existing research the effects of heterogeneous hosts had not been tested extensively and as stated in the literature SLC-checkpoints may not be portable between systems. Therefore, to evaluate and improve the reliability of CRIU it should be tested in heterogeneous hosts. Differences in kernel version, CRIU version, hardware (eg. CPU model) and resource availability are properties that in a real cluster could differ between the host where the checkpoint is created and where it is restored. Therefore research on the effects these differences have on the reliability of CRIU would is relevant.

One of the issues we noted during our research is that the scheduling practices in continuous delivery systems are not very efficient or flexible. Remedying this requires CD-platforms to be thought of as interactive task execution systems. This could mean, for example, that running tasks are grouped by their owner to enable fair resource allocation among the users of the system. The underlying cluster management systems need to support this by offering flexible scheduling of tasks, preemption and resource quotas. Research is needed in how to implement these in practice and how these could be used to improve the user experience of the CD-platform.

We have discussed the implications of our results and how they relate to previous research in the field. We also presented opportunities for future research. In the next section we conclude this thesis.

[51] ZDTM test suite. `https://criu.org/ZDTM_test_suite`

7 Conclusion

To improve the flexibility and efficiency of scheduling in continuous delivery (CD) platforms we set out to investigate whether system-level checkpointing (SLC) can be used to enable preemption of verification tools. Our initial motivation was to enable more kinds of verification tools to be integrated to CD-systems, to get more out of available resources and to improve developers' productivity. We found the current open-source CD-platforms to be lacking in their practices in task scheduling leading to inflexibility, poor efficiency and lack of interactivity.

We approached the problem by looking for a method for preempting verification tools. By studying literature we found process checkpointing to be a feasible technique for this purpose. Our aim was then to select a checkpointing technique that is capable of capturing the wide range of resources verification tools utilize and that is suitable for integration to an interactive CD-platform. We also had practical criteria, such that a tool for the checkpointing is available for our Linux platform.

Based on literature we evaluated the suitability of the different checkpointing methods. The outcome was that SLC is the most suitable for this purpose because it does not require modification of the verification tools – a very complex task for some of the commonly used tools. This means SLC can be used with proprietary verification tools as well. In addition, a SLC-utility Checkpoint/Restore in Userspace (CRIU) was available for our platform. We therefore selected CRIU for detailed evaluation.

We tested CRIU for checkpointing American Fuzzy Lop (AFL), Valgrind, AddressSanitizer and Android Emulator. Out of these AFL and Valgrind were successfully checkpointed. The others were not checkpointable due to limitations in the resource support of CRIU. The large surface area of processes means supporting all resources and their edge-cases is a difficult task and more work is required to make CRIU to reliably support most verification tools.

For AFL and Valgrind we measured the performance of checkpointing them with CRIU. We measured the time to create a checkpoint and the size of the checkpoint image at regular intervals covering an approximately 30 hour period. The outcome was that CRIU performed without any noticeable performance issues. Consistent with previous research we noted the checkpoint creation time and the image size are linearly dependent and primarily determined by the memory size of the checkpointed applications.

We can conclude from our results that checkpointing is a feasible way to implement preemption of verification tools. SLC-checkpointing is in theory able to capture all resources utilized by processes but in practice there are still limitations in CRIU affecting verification tools especially. This means arbitrary verification tools are not yet reliably checkpointable but with coordination between the CD-platform and its users selected tools could be enabled for preemption.

The performance of task preemption with CRIU depends largely on the size of the verification tool's memory and its data files. A tool that has a large amount of data files or that uses lots of memory will be slow to checkpoint and transfer to storage. To make the preemption overhead less visible to users it is possible to utilize

the kernel's I/O-caching to enable a new task start execution before the previous one has been saved to storage. The conclusion is that preemption can be used to build an interactive CD-system but large verification tools can cause noticeable delays.

We believe that support for preemption at the cluster level will be one of the next steps in the development of more sophisticated cluster computing platforms. Similarly to how it enables interactivity of processes in a single host preemption can at the cluster level make the cluster computing system an interactive distributed computer. Unfortunately it seems to us that the continuous delivery platforms are lagging behind in their practices of cluster computing and scheduling.

Why the practices in CD are lacking may be because it is only a supporting activity in software development. It likely does not enjoy the same amount of resources and interest in businesses as development that is directly responsible for creating marketable value. This leads to less research in the area. We believe CD needs to be seen as an essential core practice in all software development. Continuous delivery itself needs to shed its image as just batch task execution and become an application of data-parallel cluster computing. The CD-platforms of the future need to reflect this change.

References

[1] Pavel Begunkov (Silence). "Checkpoint and Restore of File Locks in Userspace". In: *Proceedings of the 13th Central & Eastern European Software Engineering Conference in Russia*. CEE-SECR '17. St. Petersburg, Russia: ACM, 2017, 13:1–13:4. DOI: 10.1145/3166094.3166107.

[2] Ishfaq Ahmad. "Editorial: Resource Management of Parallel and Distributed Systems with Static Scheduling: Challenges, Solutions and New Problems". In: *Concurrency: Practice and Experience* 7.5 (Aug. 1995), pp. 339–347. DOI: 10.1002/cpe.4330070502.

[3] Valentina Armenise. "Continuous Delivery with Jenkins: Jenkins Solutions to Implement Continuous Delivery". In: *Proceedings of the Third International Workshop on Release Engineering*. RELENG '15. Florence, Italy: IEEE Press, 2015, pp. 24–27. DOI: 10.1109/RELENG.2015.19.

[4] Ritu Arora. "Raising the Level of Abstraction of Application-level Checkpointing". In: *Companion to the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications*. OOPSLA Companion '08. Nashville, TN, USA: ACM, 2008, pp. 919–920. DOI: 10.1145/1449814.1449908.

[5] Ritu Arora and Trung Nguyen Ba. "ITALC: Interactive Tool for Application-Level Checkpointing". In: *Proceedings of the Fourth International Workshop on HPC User Support Tools*. HUST'17. Denver, CO, USA: ACM, 2017, 2:1–2:11. DOI: 10.1145/3152493.3152558.

[6] Fabrice Bellard. "QEMU, a Fast and Portable Dynamic Translator". In: *Proceedings of the Annual Conference on USENIX Annual Technical Conference*. ATEC '05. Anaheim, CA: USENIX Association, 2005, pp. 41–41. URL: http://dl.acm.org/citation.cfm?id=1247360.1247401.

[7] Eric Boutin et al. "Apollo: Scalable and Coordinated Scheduling for Cloud-scale Computing". In: *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*. OSDI'14. Broomfield, CO: USENIX Association, 2014, pp. 285–300. ISBN: 978-1-931971-16-4. URL: http://dl.acm.org/citation.cfm?id=2685048.2685071.

[8] Daniel Pierre Bovet. *Implementing virtual system calls*. LWN.net, https://lwn.net/Articles/615809/. Oct. 2014. (Visited on 05/15/2018).

[9] Greg Bronevetsky et al. "Application-level Checkpointing for Shared Memory Programs". In: *SIGPLAN Not.* 39.11 (Oct. 2004), pp. 235–247. DOI: 10.1145/1037187.1024421.

[10] Brendan Burns et al. "Borg, Omega, and Kubernetes". In: *Queue* 14.1 (Jan. 2016), 10:70–10:93. DOI: 10.1145/2898442.2898444.

[11] Gerry Gerard Claps, Richard Berntsson Svensson, and Aybüke Aurum. "On the journey to continuous deployment: Technical and social challenges along the way". In: *Information and Software Technology* 57 (Jan. 2015), pp. 21–31. DOI: 10.1016/j.infsof.2014.07.009.

[12] Jonathan Corbet. *The next steps for userfaultfd()*. LWN.net, https://lwn.net/Articles/718198/. Mar. 2017. (Visited on 07/18/2018).

[13] Jonathan Corbet. *User-space page fault handling*. LWN.net, https://lwn.net/Articles/636226/. Mar. 2015. (Visited on 07/17/2018).

[14] Jason Duell. *The Design and Implementation of Berkeley Lab's Linux Checkpoint/Restart*. Tech. rep. Lawrence Berkeley National Laboratory, Apr. 2005. DOI: 10.2172/891617.

[15] Hamed Esfahani et al. "CloudBuild: Microsoft's Distributed and Caching Build Service". In: *Proceedings of the 38th International Conference on Software Engineering Companion*. ICSE '16. Austin, Texas: ACM, 2016, pp. 11–20. DOI: 10.1145/2889160.2889222.

[16] Dror G. Feitelson and Morris A. Jettee. "Improved utilization and responsiveness with gang scheduling". In: *Job Scheduling Strategies for Parallel Processing*. Ed. by Dror G. Feitelson and Larry Rudolph. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pp. 238–261. DOI: 10.1007/3-540-63574-2_24.

[17] Dror G. Feitelson et al. "Theory and practice in parallel job scheduling". In: *Job Scheduling Strategies for Parallel Processing*. Ed. by Dror G. Feitelson and Larry Rudolph. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pp. 1–34. DOI: 10.1007/3-540-63574-2_14.

[18] Martin Fowler. *Continuous Integration*. martinfowler.com, https://www.martinfowler.com/articles/continuousIntegration.html. May 2016. (Visited on 05/17/2018).

[19] Ionel Gog et al. "Firmament: Fast, Centralized Cluster Scheduling at Scale". In: *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*. OSDI'16. Savannah, GA, USA: USENIX Association, 2016, pp. 99–115. ISBN: 978-1-931971-33-1. URL: http://dl.acm.org/citation.cfm?id=3026877.3026886.

[20] Peter Gutmann. "Fuzzing Code with AFL". In: *;login:* 41.2 (2016). URL: https://www.usenix.org/publications/login/summer2016/gutmann.

[21] Paul H. Hargrove and Jason C. Duell. "Berkeley lab checkpoint/restart (BLCR) for Linux clusters". In: *Journal of Physics. Conference Series* 46 (Sept. 2006). DOI: 10.1088/1742-6596/46/1/067.

[22] Michael Hilton et al. "Usage, Costs, and Benefits of Continuous Integration in Open-source Projects". In: *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ASE 2016. Singapore, Singapore: ACM, 2016, pp. 426–437. DOI: 10.1145/2970276.2970358.

[23] Benjamin Hindman et al. “Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center”. In: *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*. NSDI’11. Boston, MA: USENIX Association, 2011, pp. 295–308. URL: http://dl.acm.org/citation.cfm?id=1972457.1972488.

[24] C. Huang and C. Lee. “Enhancing the Availability of Docker Swarm Using Checkpoint-and-Restore”. In: *2017 14th International Symposium on Pervasive Systems, Algorithms and Networks 2017 11th International Conference on Frontier of Computer Science and Technology 2017 Third International Symposium of Creative Computing (ISPAN-FCST-ISCC)*. June 2017, pp. 357–362. DOI: 10.1109/ISPAN-FCST-ISCC.2017.69.

[25] Kohsuke Kawaguchi. *Hudson*. Java One Conference, https://web.archive.org/web/20140701020639/https://www.java.net//blog/kohsuke/archive/20070514/Hudson%20J1.pdf. 2007. (Visited on 05/26/2018).

[26] Julian Andres Klode. *Checkpointing Android Emulators*. Tech. rep. Oct. 2015. URL: http://www.mathematik.uni-marburg.de/~klode/qemu/report.pdf (visited on 10/12/2018).

[27] Oren Laadan and Serge E Hallyn. “Linux-CR: Transparent Application Checkpoint-Restart in Linux”. In: *Linux Symposium*. Vol. 159. 2010. URL: http://www1.cs.columbia.edu/~orenl/papers/ols2010-linuxcr.pdf.

[28] Eero Laukkanen and Mika V. Mäntylä. “Build Waiting Time in Continuous Integration: An Initial Interdisciplinary Literature Review”. In: *Proceedings of the Second International Workshop on Rapid Continuous Software Engineering*. RCoSE ’15. Florence, Italy: IEEE Press, 2015, pp. 1–4. DOI: 10.1109/RCoSE.2015.8.

[29] Jack Li et al. “Improving Preemptive Scheduling with Application-Transparent Checkpointing in Shared Clusters”. In: *Proceedings of the 16th Annual Middleware Conference*. Middleware ’15. Vancouver, BC, Canada: ACM, 2015, pp. 222–234. DOI: 10.1145/2814576.2814807.

[30] Michael Litzkow et al. *Checkpoint and Migration of UNIX Processes in the Condor Distributed Processing System*. Tech. rep. University of Wisconsin-Madison Department of Computer Sciences, 1997. URL: http://digital.library.wisc.edu/1793/60116.

[31] Torvald Mårtensson, Pär Hammarström, and Jan Bosch. “Continuous Integration is Not About Build Systems”. In: *2017 43rd Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. Aug. 2017, pp. 1–9. DOI: 10.1109/SEAA.2017.30.

[32] Atif Memon et al. “Taming Google-scale Continuous Testing”. In: *Proceedings of the 39th International Conference on Software Engineering: Software Engineering in Practice Track*. ICSE-SEIP ’17. Buenos Aires, Argentina: IEEE Press, 2017, pp. 233–242. DOI: 10.1109/ICSE-SEIP.2017.16.

[33] Shripad Nadgowda et al. “Voyager: Complete Container State Migration”. In: *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*. June 2017, pp. 2137–2142. DOI: 10.1109/ICDCS.2017.91.

[34] Nicholas Nethercote. *Dynamic binary analysis and instrumentation*. Tech. rep. UCAM-CL-TR-606. University of Cambridge, Computer Laboratory, Nov. 2004. URL: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-606.pdf.

[35] Nicholas Nethercote and Julian Seward. “Valgrind: A Framework for Heavy-weight Dynamic Binary Instrumentation”. In: *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’07. San Diego, California, USA: ACM, 2007, pp. 89–100. DOI: 10.1145/1250734.1250746.

[36] Yuqing Qiu et al. “LXC Container Migration in Cloudlets under Multipath TCP”. In: *2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC)*. Vol. 2. July 2017, pp. 31–36. DOI: 10.1109/COMPSAC.2017.163.

[37] R. Owen Rogers. “Scaling Continuous Integration”. In: *Extreme Programming and Agile Processes in Software Engineering*. Ed. by Jutta Eckstein and Hubert Baumeister. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 68–76. DOI: 10.1007/978-3-540-24853-8_8.

[38] Eric Roman. *A Survey of Checkpoint/Restart Implementations*. Tech. rep. Lawrence Berkeley National Laboratory, Tech, 2002. URL: https://crd.lbl.gov/assets/pubs_presos/CDS/FTG/Papers/2002/checkpointSurvey-020724b.pdf.

[39] Wolfram Schulte. “Changing Microsoft’s Build: Revolution or Evolution”. In: *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ASE 2016. Singapore, Singapore: ACM, 2016, pp. 2–2. DOI: 10.1145/2970276.2985779.

[40] Malte Schwarzkopf. “Operating System Support for Warehouse-Scale Computing”. University of Cambridge, Oct. 2015. DOI: 10.17863/CAM.26443.

[41] Malte Schwarzkopf et al. “Omega: Flexible, Scalable Schedulers for Large Compute Clusters”. In: *Proceedings of the 8th ACM European Conference on Computer Systems*. EuroSys ’13. Prague, Czech Republic: ACM, 2013, pp. 351–364. DOI: 10.1145/2465351.2465386.

[42] Konstantin Serebryany et al. “AddressSanitizer: A Fast Address Sanity Checker”. In: *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*. USENIX ATC’12. Boston, MA: USENIX Association, 2012, pp. 28–28. URL: http://dl.acm.org/citation.cfm?id=2342821.2342849.

[43] Luís Moura Silva and João Gabriel Silva. “System-Level versus User-Defined Checkpointing”. In: *Proceedings Seventeenth IEEE Symposium on Reliable Distributed Systems (Cat. No.98CB36281)*. Oct. 1998, pp. 68–74. DOI: 10.1109/RELDIS.1998.740476.

[44] Jaspal Subhlok et al. “Exploiting Task and Data Parallelism on a Multicomputer”. In: *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPOPP ’93. San Diego, California, USA: ACM, 1993, pp. 13–22. DOI: 10.1145/155332.155334.

[45] Vinod Kumar Vavilapalli et al. “Apache Hadoop YARN: Yet Another Resource Negotiator”. In: *Proceedings of the 4th Annual Symposium on Cloud Computing*. SOCC ’13. Santa Clara, California: ACM, 2013, 5:1–5:16. DOI: 10.1145/2523616.2523633.

[46] Abhishek Verma et al. “Large-scale Cluster Management at Google with Borg”. In: *Proceedings of the Tenth European Conference on Computer Systems*. EuroSys ’15. Bordeaux, France: ACM, 2015, 18:1–18:17. DOI: 10.1145/2741948.2741964.

[47] Hua Zhong and Jason Nieh. *CRAK: Linux Checkpoint/Restart As a Kernel Module*. CUCS-014-01. Tech. rep. Department of Computer Science, Columbia University, 2001. URL: http://systems.cs.columbia.edu/archive/pub/2001/11/crak-linux-checkpoint-restart-as-a-kernel-module/.

A Test Environment Specification

CPU	Intel Core i7-4860EQ, 1.80GHz
Memory	DDR3, 8GB, 1.6GHz
Disk	60GB INTEL SSDSC2CW06
Operating System	Debian Buster
Kernel	Linux 4.17.0-3-amd64

B Software Versions Used in Evaluation

CRIU	3.8.1
AddressSanitizer (GCC)	Debian 7.3.0-15
Valgrind	3.13.0-2
Stress-ng	0.09.23
Android Emulator	27.3.8.0 (build_id 4848055)
Android System Image	Android 24 Default, ARM EABI v7a, Version 7