DEGREE PROJECT IN COMPUTER SCIENCE AND ENGINEERING,
SECOND CYCLE, 30 CREDITS
STOCKHOLM, *SWEDEN 2020*

Direct Heap Snapshotting in the Java HotSpot VM: a Prototype

LUDVIG JANIUK

KTH ROYAL INSTITUTE OF TECHNOLOGY
SCHOOL OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

Direct Heap Snapshotting in the Java HotSpot VM: a Prototype

Ludvig Janiuk

2020

Master's Thesis in Theoretical Computer Science
Supervisor: Philipp Haller
Examiner: Roberto Guanciale
Swedish title:
Direkt Heap-Snapshottande i Java HotSpot's VM: en Prototyp
School of Electrical Engineering and Computer Science

Abstract

The Java programming language is widely used across the world, powering a diverse range of technologies. However, the Java Virtual Machine suffers from long startup time and a large memory footprint. This becomes a problem when Java is used in short-lived programs such as microservices, in which the long initialization time might dominate the program runtime and even violate service level agreements. Checkpoint/Restore (C/R) is a technique which has reduced startup times for other applications, as well as reduced memory footprint. This thesis presents a prototype of a variant of C/R on the OpenJDK JVM, which saves a snapshot of the Java heap at some time during initialization. The primary goal was to see whether this was possible. The implementation successfully skips parts of initialization and the resulting program still seems to execute correctly under unit tests and test programs. It also reduces runtime by a minuscule amount under certain conditions. The portion of initialization being snapshotted would need to be further extended in order to result in larger time savings, which is a promising avenue for future work.

Sammanfattning

Programmeringsspråket Java används i hela världen, och driver en bred mängd olika teknologier. Javas Virtuella Maskin lider däremot av en lång uppstartstid och ett stort minnesavtryck. Detta blir ett problem när Java används för kortlivade program liksom microservices, i vilka den långa initialiseringstiden kan komma att dominera programmets körtid, och till och med bryta avtal om tjänstens tillgänglighet. Checkpoint/Restore (C/R) är en teknologi som har minskat uppstartstid samt minnesavtryck för andra applikationer. Detta arbete presenterar en prototyp där en variant av C/R applicerats på OpenJDK JVM, och sparar undan en kopia av Java-heapen vid en specifik tidspunkt under initialiseringen. Det främsta målet har varit att undersöka om detta är möjligt. Implementationen lyckas med framgång hoppa över delar av initialiseringen och det resulterande programmet verkar fortfarande exekvera korrekt under enhetstester och testprogram. Implementationen minskar också uppstartstid med en väldigt liten bråkdel under vissa omständigheter. För att spara mera tid skulle perioden som hoppas över med hjälp av snapshottet behöva vara större, vilket är en lovande riktning för framtida arbete.

Acknowledgements

The progress I've made in this thesis would not have been possible without the guidance and support of the Oracle JPG Group in Stockholm. I want to thank each and every one of the outstanding people there for their willingness to share knowledge, their patience, their passion, and their kindness.

In particular, I want to thank Tobias Wrigstad for guidance in strategy and writing, and Ioi Lam for his expertise and dedicated time which really boosted my progress. I'm also thankful to Claes Redestad, David Simms, Erik Österlund, Robbin Ehn, and all others who took time to explain JVM intricacies to me and answer all my questions. Finally, I want to thank Philipp Haller for being my adviser.

Contents

1 Introduction **1**
- 1.1 Problem Description . . . 1
- 1.2 Checkpoint/Restore . . . 3
- 1.3 The Vision of Heap Snapshotting . . . 3
- 1.4 Purpose . . . 3
- 1.5 Goals . . . 4
- 1.6 Contributions . . . 5
- 1.7 Ethical Considerations . . . 6
- 1.8 Plan of the Document . . . 6

2 Background and Related Work **7**
- 2.1 Java Primer . . . 7
- 2.2 Previous Work . . . 7
  - 2.2.1 GraalVM's "Run Once Initialize Fast" with Closed World Assumption . . . 8
  - 2.2.2 jaotc . . . 8
  - 2.2.3 jlink . . . 8
  - 2.2.4 Nailgun . . . 9
  - 2.2.5 Oracle's "Project Leyden" . . . 9
- 2.3 Checkpoint/Restore . . . 9
- 2.4 The JVM in depth . . . 10

3 Method **11**
- 3.1 Overview of Implementation . . . 11
- 3.2 Usage . . . 13
- 3.3 Evaluation: Overview of the Tests . . . 15
  - 3.3.1 No performance testing on real-world programs . . . 15
  - 3.3.2 System Properties of the Testing Environment . . . 15
  - 3.3.3 Testing Conditions . . . 15
- 3.4 DHS-vs-Stock . . . 16
- 3.5 Moments . . . 17
  - 3.5.1 Pretouch . . . 17
  - 3.5.2 Methodology Verification . . . 18
- 3.6 OpenJDK Unit Tests . . . 18

4 Approach **20**
4.1 Anatomy of the Snapshot 20
4.1.1 The Heap Snapshot 20
4.1.2 Class and Native Method Metadata 20
4.1.3 Snapshot Metadata 21
4.2 Heap Dumping: Saving the Snapshot 22
4.2.1 Saving the Heap to File 22
4.2.2 Saving Auxiliary Data Structures 23
4.3 Heap Restoring: Starting from the Snapshot 23
4.3.1 Reading the Snapshot Files 23
4.3.2 Synthetic Initialization 25
4.4 Common Concerns in Implementation 27
4.5 Simplifications, Trade-offs, and Limitations 27

5 Results **29**
5.1 DHS-vs-Stock 29
5.2 Moments 31
5.3 Correctness Tests 31
5.3.1 jtreg Test Results 31
5.3.2 Evaluation on Test Programs 31

6 Discussion **34**
6.1 Correctness Confidence 34
6.2 Sensitive Memory 34
6.3 DHS-vs-Stock 35
6.4 Moments 35
6.5 Reliability of Runtime Differences 36
6.6 Criticisms 37

7 Conclusions & Future Work **39**
7.1 Roadmap 39
7.2 Challenges 40
7.3 Project Leyden 40
7.4 Research Approaches for Future Work 41

A Build instructions **42**
A.1 Building 42

B The JVM in Depth: A Focus on Internals and Startup **43**
B.1 Memory Areas of the Java HotSpot VM 43
B.2 The Role of Classes 45
B.3 Oops and OopHandles 46
B.4 Class Loading Roadmap 46
B.5 Class Data Sharing 47

C Can Pointers Keep Their Meaning? 48
C.1 Native Function Pointers 48
C.2 Pointers Within the Heap 49
C.3 Pointers from Metaspace to Heap 49
C.3.1 Pointers to "Global Singleton Objects" 49
C.3.2 Pointers to "Identifiable Objects" 49
C.3.3 "Unidentifiable Objects" 50
C.4 Pointers from Heap to Metaspace 50
C.5 Other "Pointers" 50

Chapter 1

Introduction

1.1 Problem Description

The Java programming language is a technology used worldwide in countless applications, from embedded applications, to desktop programs, to servers. According to one estimate [Pot12], there were over 8 million Java developers in the world in 2012. Java virtual machines can take a relatively long time to start up compared to other languages, because of the expensive code verification, class loading, bytecode interpretation, profiling, and dynamic compilation they have to perform [Wim+19].

Microservices have become a very popular strategy and changed how server applications are written and deployed. Serverless architectures are an even more granular example. When considering long-lived programs such as monolithic web servers, startup time is of little concern and can largely be ignored. However, in short-lived programs, startup time begins to be a large part of the program lifetime and can even dominate it as seen in Table 1.1. Given the popularity of system architectures which rely on short-lived programs, this slow startup of Java does become a pain point. Microservices or Function-as-a-service functions written in Java could potentially be costing more in execution time fees than necessary; moreover, since cold VM startup can be an order of magnitude slower [Akk+18], this might lead to breaking service-level agreements. The first worker to spawn for a microservice will require a cold start of the language runtime [Wim+19].

This might force developers to choose languages other than Java for deployments which include the various kinds of short-lived programs. In such cases, developer time might be wasted re-implementing software packages and libraries for which there already exist open-source and/or time-tested solutions in Java, due to its 25-year old history.

Hello World version	**Runtime (ms)**
C++	0.89
Java	33.15

Table 1.1: Runtimes of two Hello World programs, written in C++ and Java, as measured once. The difference provides some illustrative inspiration for this work: should not the JVM be able to execute Hello World quickly?

1.2 Checkpoint/Restore

Checkpoint/Restore is a mature technology which has been successfully used to freeze and restore, and even migrate, whole groups of interconnected processes between machines in e.g. high-performance computing settings. A main example is CRIU, described later in the background.

1.3 The Vision of Heap Snapshotting

Perhaps ideas from Checkpoint/Restore could be used to mitigate Java's startup problem? After all, it is conceivable that JVM initialization is relatively deterministic: after being initialized, the JVM's runtime state might look very similar every time. So, it seems worth attempting to just start the JVM from such an "initialized state" directly, without actually running the initialization every time. That is exactly what this thesis attempts to prototype, and the same strategy has already been realized in the text editor Emacs (section 2.3).

Challenges There are of course multiple challenges to this approach: How do we know that a restored process is "safe" and stable? How do we go about implementing this - do we start from a snapshot just before the main function, and fix errors until it finally works, or do we start by snapshotting a very small early part, and try to move the snapshot ahead in a more iterative approach? When errors arise, how do we fix them? Are all state inconsistencies fixable? Will the fixes take up more time than is saved in the first place?

1.4 Purpose

The purpose of this work is to attempt to reduce JVM startup time. It has been said somewhere that frequent recompilations of the Linux kernel are responsible for the cutting down of a million trees. Perhaps a similar thing could be said about JVM initialization. If the initialization sequence of JVM's is mostly deterministic, then re-running it every time seems like a similar waste of computer time.

Saving computer power This is a good place to attack, as Java is incredibly popular, and used all around the world. Large parts of the world run Java, but in the modern world, the startup problem is exacerbated by e.g. microservices, as startup becomes not inconsequential, but a large part of total program runtime for short-running programs. Therefore, reducing this could have impact on the total amount of computation done in the world.

Having Java start faster Java startup is also an ergonomic factor for all the Java developers in the world. Faster startup means faster iteration, which means faster development.

Efficient Java usage in short-lived programs As Java is a language with 25 years of history, a rich set of libraries and software packages have been developed for it. Expertise in these is widespread. It would therefore be a shame if Java's startup time were a limiting factor in its adoption for today's diverse deployment needs, as opposed to the historic monolithic servers. This problem is another reason for this research.

Making existing deployments start faster Finally, Java is indeed used in existing microservice and serverless setups, and if JVM startup time were reduced, the running cost of all these existing deployments could potentially be reduced, without any more effort than a version update.

Memory sharing Used live memory is also a limiting factor for infrastructure providers. As a secondary purpose, it is worth considering whether synergy effects can lead to reduced usage of memory in server environments with several JVMs running in parallel, e.g. through copy-on-write.

1.5 Goals

The goal of this work is to investigate to what extent the ideas of Checkpoint/Restore can be applied to reduce JVM startup time, by developing a prototype focused on snapshotting the Java Heap. This prototype is in essence a source code patch to the JVM. A goal is also to facilitate future work on this problem.

Heap Snapshotting, not full Checkpoint/Restore While C/R is often used in multi-process environments (e.g. supercomputers), this work focuses on the startup of a single JVM process, on a single machine. It is not a goal to implement full C/R, i.e. the possibility to serialize program (or system-of-programs) state at arbitrary points during execution. We call the version of Heap Snapshotting presented in this thesis "Direct Heap Snapshotting" (DHS), and define it as: 1) Taking a snapshot of the Java Heap at a specific point during initialization; 2) On future runs, overwriting the Java heap in-place with the snapshot, and 3) Repairing the runtime state in any necessary way to enable starting execution from that point.

Implementation strategy As time is very limited, the goal is to produce a prototype testing the core idea of DHS, ignoring or deferring periphery issues as much as possible. The goal is not to create a production-ready patch that could easily be integrated into current workflows. Neither is it a goal to reach a point of actually saving time, meaning that not too much time is to be put on optimization. Several different approaches might be tested to find one that works well.

Definition of success A good Heap Snapshotting (HS) solution will be one that:

- Allows us to skip the execution of as many bytecodes as possible.
- Still achieves everything that those bytecodes achieved; heap state is equivalent to that after being "normally" initialized, and all if any side effects of initialization still happen.
- Does not impact future program execution in any negative way.
- Is able to perform restoration as quickly as possible, and crucially, the time to restore the heap must be a lot less than the time saved by not running the bytecodes.

Ideally a program running on the JVM should not be able to distinguish whether it has been initialized normally or merely heap-restored, but for looking at time passed. This is however a metric of success rather than a goal in itself.

We do not aim to achieve all parts of a "good" heap snapshot in this thesis, instead we leave a lot of it as future work.

Investigating implementation difficulty Another goal is to gauge the implementation difficulty of a "good" HS. It is after all possible that the JVM is so complex that trying to overwrite the whole heap with an earlier version and fix all the problems, is a futile attempt. So, we are interested in how much effort is required to produce a stable solution, which hopefully is also faster. At what point is trying to implement more Heap Snapshotting not worth the benefits, compared to other JVM startup optimizations?

1.6 Contributions

This thesis presents the following contributions:

- An implementation of Direct Heap Snapshotting in the JVM. The implementation takes a snapshot of the JVM heap and uses it to start without performing parts of the initialization. It overwrites the heap directly when restoring, and makes no restrictions on what java programs can be run with it (e.g. does not make the *closed-world assumption* as in [Wim+19]).
- Measurements of the implementation's performance, as compared to an unmodified JVM, focused on startup performance.
- Analysis of the implementation's stability and reliability through unit tests and executed programs.
- Discussion of the empirical results, and how they might be affected as future work progresses.

- The implementation should serve as a springboard for future work. It contains a lot of groundwork that is thought to enable more rapid development in the next stages.

1.7 Ethical Considerations

Higher time-efficiency and power-efficiency of Java has a lowering impact on cost, as well as on usage of resources. However, rebound effects might manifest in people deploying more services, thus negating the saved resources. One could consider whether improving developer ergonomics and efficiency is a net good for society. In a job market with high unemployment, people are looking to the software sector for jobs, and making developer work more efficient might reduce the demand of software developers, thus potentially compounding unemployment. But this would be an anti-innovation way of thinking - the solution to unemployment ought not be deliberate inefficiency. It is the opinion of the author that any ethical considerations of this research are negligible.

1.8 Plan of the Document

Chapter 2 introduces the basic knowledge that is required to serve as context for the rest of the work. Chapter 3 explains how to replicate the results of this thesis by first going over the build process of the source code patch that has been developed, then going over the broad strokes of how the code works, and finally detailing the setup of the tests performed in the evaluation process. Chapter 4 explains how the code works in more detail, also detailing design decisions and trade-offs. Chapter 5 summarizes the most important results both from the development work and from the evaluations. Finally, chapter 6 provides an analysis of the results and some interpretations, and chapter 7 gives conclusions and outlines the road ahead for future development of this research. The appendices contain some useful summaries of advanced but related JVM topics, as well as a broader speculation on the feasibility of larger heap snapshotting.

Chapter 2

Background and Related Work

2.1 Java Primer

Quoting Oracle's own description [Ora]:

> The Java™ Programming Language is a general-purpose, concurrent, strongly typed, class-based object-oriented language. It is normally compiled to the bytecode instruction set and binary format defined in the Java Virtual Machine Specification.

In the scope of this thesis, what's important are not details of the Java language itself, but instead how it is executed, i.e. the Java Virtual Machine. The JVM knows nothing about Java, but instead executes *bytecodes* contained in `.class` files. This is what allows Java to be platform-agnostic; as soon as a JVM has been implemented for a particular platform, *classfiles* can be executed on it. Usage of the Java *language* is not even necessary, any language that can be compiled to bytecodes can be hosted on the JVM [Lin+20a].

There are many *JVM vendors*: organizations or companies which develop and maintain their own implementations of the JVM. As long as a JVM implementation is conforming to the JVM specification, it should be able to execute any given classfiles. HotSpot [gro] is the reference JVM implementation provided by Oracle, but for example there exists also GraalVM [Gra] and RedHat OpenJDK [Red].

2.2 Previous Work

Before investigating the problem of improving Java startup, it is useful to consider what approaches have already been tested.

2.2.1 GraalVM's "Run Once Initialize Fast" with Closed World Assumption

The team behind GraalVM achieves two orders of magnitude faster Java startup compared to the HostSpot JVM, under certain restrictions which are argued to be suited for deployments such as microservices [Wim+19]. They use the ideas of Checkpoint/Restore in running initialization once, saving the heap status after initialization, and then being able to restore a program to start from that heap. While this is also a variant of snapshotting the heap, they load their snapshot into a dedicated "image heap" memory area, whereas Heap Snapshotting as described in this thesis happens in-place, overwriting the memory area of the Java Heap directly. They also utilize "a novel iterative application of points-to analysis" and ahead-of-time compilation. A notable limitation is that the GraalVM approach sacrifices the ability of the JVM runtime to load arbitrary classes with arbitrary class loaders, that is, they adopt the *closed-world assumption*. In contrast, the prototype of Heap Snapshotting presented in this thesis does not impose such a restriction: once the JVM is restored from the snapshot, it functions just as it if had been initialized normally. As compared to existing Checkpoint/Restore systems, they state:

> We believe that our approach is more suitable for microservices than checkpoint/restore systems, e.g., CRIU, that restore a Java VM such as the Java HotSpot VM: Restoring the Java HotSpot VM from a checkpoint does not reduce the memory footprint that is systemic due to the dynamic class loading and dynamic optimization approach, i.e., the memory that the Java HotSpot VM needs for class metadata, Java bytecode, and dynamically compiled code. In addition, it cannot rely on a points-to analysis to prune unnecessary parts of the application.

Their paper contains some tools that can be useful for research into heap restoration topics, such as a script for access tracing at runtime.

2.2.2 jaotc

The *Java Ahead-Of-Time Compiler* [Koz] is a tool introduced to allow classes to be compiled to native code ahead of program execution. This improves startup time as less time needs to be spent compiling and optimizing code. These gains are orthogonal with the goals of this thesis.

2.2.3 jlink

`jlink` is a Java tool that allows creating a custom JRE image for a specific application, optimizing away in advance modules that are not used. It also allows many other miscellaneous link-time optimizations [Ora17b][Red17].

2.2.4 Nailgun

Nailgun is a script that allows a JVM to be started once, ahead of time, and then when a program needs to be executed, that existing VM is adapted to execute the program, instead of starting a new one. It was originally meant to quickly execute command line programs on the JVM [Lam]. This clever idea is in line with the goals of this thesis as far as latency is concerned, since it allows one to start a program without waiting for JVM initialization. Sadly, the requirement of having a JVM constantly running is equivalent to having workers that are never killed. This is wasteful of memory resources on rarely-accessed services, which is the reason why cold starts are indeed tolerated in general. Nailgun is also not secure in its current implementation, because command information is transferred between processes with little to no protection. The project seems to now be maintained by Facebook [Fac].

2.2.5 Oracle's "Project Leyden"

Announced on April 27 2020 by Mark Reinhold, *Project Leyden* [Rei20] can be seen as a serious investment in alleviating the problem of slow Java startup. The project is currently in a very early stage, but the plan seems to be to add support for "static images" to Java - compiled executables which run just one Java program without the possibility of extension with custom class loaders. That is, this project aims to use the closed world assumption, just like GraalVM's solution.

2.3 Checkpoint/Restore

Checkpoint/Restore (C/R) is the idea of saving process state so that it can be reconstructed in the future [BW01]. It is used for load balancing and fault tolerance among machines, e.g. in high-performance computing or the CMS experiment of the Large Hadron Collider at CERN, but also for regular desktop computers, or container migration. Some technologies which implement C/R are DMTCP and CRIU [AAC07][Pic+16]. While these projects focus on checkpointing of whole processes or even groups of interdependent processes, the idea has also seen other uses. As one example, the build process of text editor Emacs involves running initialization lisp scripts. Instead of running these every time at startup, Emacs runs these as part of the build step, and then saves a snapshot of the program state which is loaded directly at startup in subsequent runs [Fre19].

A central challenge of any Checkpoint/Restore scheme is to save all necessary state, and handle all the necessary environment connections, so that a process can be continued at a later time. This is especially visible in DMTCP ([AAC07] page 1, introduction):

> DMTCP automatically accounts for fork, exec, ssh, mutexes/semaphores, TCP/IP sockets, UNIX domain sockets, pipes, ptys (pseudo-terminals),

terminal modes, ownership of controlling terminals, signal handlers, open file descriptors, shared open file descriptors, I/O (including the readline library), shared memory (via mmap), parent-child process relationships, pid virtualization, and other operating system artifacts.

Of course, all of these "operating system artifacts" are necessary for proper process functioning, and it is conceivable that if any of them is not treated, or restored improperly, then errors could manifest, perhaps in subtle ways.

2.4 The JVM in depth

Appendix B is an extension to this background which introduces, summarizes and defines many basic as well as advanced concepts intrinsic to JVM programming. If one is unfamiliar with the codebase and wants to follow along successive chapters on a details level, especially chapter 4, one is encouraged to read it. However, for the reader that is more interested in the big picture and research results, it is skipped from here because of its length.

Chapter 3

Method

In this chapter, I first give an overview over how the prototype developed performs Heap Snapshotting, then I give replication instructions by explaining the build process, usage, and finally evaluation strategies.

3.1 Overview of Implementation

The prototype that has been developed successfully snapshots the whole Java Heap at a certain point in initialization, and initializes from it on subsequent runs by using it to overwrite the Java Heap directly. The snapshot which is saved contains the heap and auxiliary data, and is saved to disk as three separate files. The role of each file as well as their detailed contents are described in Section 4.1. *Heap Dumping* is the process of writing the snapshot to disk, and involves concerns such as finding the right areas in memory, and traversing the class graph. It is described in detail in Section 4.2. *Heap restoration* is the process of loading and preparing the heap snapshot, and launching a program on it. This includes what we will sometimes refer to as "fixup procedures", and is described in Section 4.3.

The source code patch that has been developed consists of changes to 19 files in the OpenJDK HotSpot JVM source code, plus the addition of one file, totalling roughly 1500 lines of code added or changed. The largest changes have been in the following files:

```
src/hotspot/share/runtime/thread.cpp
src/hotspot/share/oops/klass.cpp
```

with some files only containing changes necessary to satisfy C++' rules on privacy. The code is written in such a way as to only perform extra functionality when enabled, so with default options, the modified JVM still behaves like the regular version. The basic structure of the code is captured by the pseudocode in Figure 3.1:

```
initialize_java_lang_classes() {
  //...

  if(/* Restoring the heap */) {
    restore_heap_dump();
  }else{
    // Do all initialization as normal

    initialize_class(vmSymbols::java_lang_String());
    initialize_class(vmSymbols::java_lang_System());
    //... Normal initialization which takes time

    if(/* Dumping the heap */) {
      save_heap_dump();
      exit(0);
    }
  }

  // Proceed with rest of initialization.
  // Not covered by snapshot yet.
}
```

Figure 3.1: The main structure of the code changes in the DHS patch.

```
# run heap dumping, do not print timestamps
jdk/build/linux-x64/images/jdk/bin/java
    -XX:+UnlockExperimentalVMOptions
    -XX:+UseEpsilonGC
    -Xmx1024M
    -Xms1024M
    -XX:EpsilonMaxTLABSize=8M
    -XX:MinTLABSize=8M
    -XX:HeapSnapshottingMode=4
    -version

# run minesweeper on restored heap, print timestamps
jdk/build/linux-x64/images/jdk/bin/java
    -XX:+UnlockExperimentalVMOptions
    -XX:+UseEpsilonGC
    -Xmx1024M
    -Xms1024M
    -XX:EpsilonMaxTLABSize=8M
    -XX:MinTLABSize=8M
    -XX:+JaniukTimeEvents
    -XX:HeapSnapshottingMode=3
    -jar minesweeper.jar
```

Figure 3.2: Examples of full run commands. Newlines added for readability.

3.2 Usage

Having built the modified JVM (refer to instructions in Appendix A), using DHS is a two-step processes controlled by the `HeapSnapshottingMode` option. First, the snapshot must be generated, and this is done by setting `HeapSnapshottingMode` to the code 4. Running this with the program you intend to run[1] will generate the snapshot and exit. Run with `HeapSnapshottingMode` set to the code 3 to start from the last generated snapshot.[2]

Both run modes also require a common set of command line options. Omitting any of them has a high chance of resulting in a crash. They are summarized in Figure 3.3 and full examples of run commands are given in Figure 3.2.

[1] Strictly speaking, any program will work, e.g. `-version`. Since the snapshot is very early in JVM initialization, snapshots should be program-agnostic.

[2] Codes 1 and 2 are reserved for expansion work. Code 0 is the default and results in a normal run, therefore, without this option the modified JVM behaves like a normal JVM.

`UnlockExperimentalVMOptions` Necessary to use e.g. Epsilon GC.

`UseEpsilonGC` Enable Epsilon GC.

`-xms1024m -xmx1024m` These set the heap size at 1 gigabyte, which is larger than normal. Used to facilitate running under Epsilon. I actually only needed a "minimum" heap size but without the other, the JVM outputs annoying warnings.

`EpsilonMaxTLABSize=8m, MinTLABSize=8m` Increase the size of TLABs to 8 megabyte so I can fit all of the used Heap into one TLAB during start, avoiding having to handle multiple TLABs when restoring. This might need to be increased further in the future, unless multiple TLAB support is implemented.

`-xShare:on` Forces CDS to be enabled. It's usually on by default, but CDS is relly necessary. There is also a check in the code patch that makes sure it's on.

`-xx:HeapSnapshottingMode=3` Essential. Controls the run mode. This makes it load the heap from snapshot during initialization.

`-xx:-JaniukTimeEvents` Suppress some timing debug output, See "timing tests".

`-xx:janiukprintstats=0` Suppress miscellaneous debugging output.

Figure 3.3: Explanations of common command line options needed for Heap Snapshotting.

3.3 Evaluation: Overview of the Tests

Evaluation has been performed in part focused on performance and in part on correctness and robustness. Correctness of the restored process was measured by running the parts of the JVM test suite that are relevant for the changed code. Being restored from a snapshot should not introduce any failing test cases. Apart from unit testing, confidence in correctness is also strengthened by running various real-world Java programs in the restored JVM. Any program which can run on an unmodified JVM should run without any errors on the modified version with heap restoration.

The DHS-vs-Stock test compares the total runtime of the DHS patch with an unmodified JVM by running a short-lived program under both in an interleaving fashion. In the "Moments" test, a breakdown of the impact of different operations during heap restoration is measured, by printing timestamps between the different operations. The goal is to find out which restoration operations are the most expensive. Something that has not been analyzed from a time perspective is time cost of dumping the heap. This is presumed to not be a relevant concern.

3.3.1 No performance testing on real-world programs

All the performance tests have been done only on `java --version`, and performance impact has not been measured in any way on real-world programs such as web servers, games, et.c. The reason for this is that the changes made only impact a very early part of JVM initialization, which happens long before even the first bytecode of a given program is executed. Therefore, the performance impact does not depend on the application being run. It is desirable to run with an application that is as short-lived as possible, since a longer execution time would only contribute noise to the measurements.

3.3.2 System Properties of the Testing Environment

The tests were done on an ASUS laptop computer running Ubuntu 18.04, Linux kernel version 4.15.0-101-generic. The processor is a Intel(R) Core(TM) i7-6700HQ CPU @ 2.60GHz with an L2 cache of 6144 KB. The stock version of Java compared against is Java 15.

3.3.3 Testing Conditions

When testing runtime at scales of 10's of milliseconds, it is difficult to avoid noise, and so efforts made to avoid it are important. Both the Moments test and the DHS-vs-Stock test were made under the following conditions. All other applications as well as background applications were turned off. Network was turned off to avoid spontaneous work. Bluetooth was turned off as well. Prior to starting tests, the system monitor was used to ensure that the processor was not busy performing any other work.

3.4 DHS-vs-Stock

This test is made with the purpose of investigating what impact DHS has on startup time, on the program `java -version` which just prints the version of the JVM and exits. The test is set up to compensate for variations in runtime, such as changes in system performance due to e.g. temperature and other variations. Two separate JVMs are compiled, one patched with the implementation of Direct Heap Snapshotting and one completely without. These are called "DHS" and "Stock". First, the Class Data Sharing (CDS) archives are initialized and DHS is run in heap dumping mode, so that a snapshot is established. Then both versions are run once each for the sake of warmup; these runs are not included in measurements. The Unix `perf stat` tool is then used to run the programs for 400 repetitions each, and to collect measurements including executed machine instructions and time elapsed. A bash script runs these two JVMs under `perf` 10 times in an interleaved fashion, that is: A, B, A, B, A, B, ... In the end, therefore, the sequence of executions is equivalent to running:

```
perf stat -r 400 [stock-jvm] [options]
perf stat -r 400 [dhs-jvm] [options] -XX:HeapSnapshottingMode=3
perf stat -r 400 [stock-jvm] [options]
perf stat -r 400 [dhs-jvm] [options] -XX:HeapSnapshottingMode=3
perf stat -r 400 [stock-jvm] [options]
perf stat -r 400 [dhs-jvm] [options] -XX:HeapSnapshottingMode=3
perf stat -r 400 [stock-jvm] [options]
perf stat -r 400 [dhs-jvm] [options] -XX:HeapSnapshottingMode=3
perf stat -r 400 [stock-jvm] [options]
perf stat -r 400 [dhs-jvm] [options] -XX:HeapSnapshottingMode=3
perf stat -r 400 [stock-jvm] [options]
perf stat -r 400 [dhs-jvm] [options] -XX:HeapSnapshottingMode=3
perf stat -r 400 [stock-jvm] [options]
perf stat -r 400 [dhs-jvm] [options] -XX:HeapSnapshottingMode=3
perf stat -r 400 [stock-jvm] [options]
perf stat -r 400 [dhs-jvm] [options] -XX:HeapSnapshottingMode=3
perf stat -r 400 [stock-jvm] [options]
perf stat -r 400 [dhs-jvm] [options] -XX:HeapSnapshottingMode=3
perf stat -r 400 [stock-jvm] [options]
perf stat -r 400 [dhs-jvm] [options] -XX:HeapSnapshottingMode=3
```

Where `[options]` is

```
-XX:+UnlockExperimentalVMOptions
-XX:+UseEpsilonGC
-Xmx1024M
-XX:EpsilonMaxTLABSize=8M
-Xms1024M
-XX:MinTLABSize=8M
-Xint
```

```
-XX:-UsePerfData
-version
```

`[stock-jvm]` is `jvm-stock/images/jdk/bin/java`, and
`[dhs-jvm]` is `jvm-dhs-version/images/jdk/bin/java`.

3.5 Moments

To measure what was taking the most part in restoration, the restoration procedure was segregated into reasonable and distinct periods at the level of the source code. At the start of and in-between each period, the function `print_time` was called, which prints a timestamp in nanoseconds to standard output together with an identifying mnemonic "tag" for this moment in time. This output is enabled with the `-XX:+JaniukTimeEvents` command line parameter. The DHS JVM version in restoration mode was run 400 times in succession under `perf stat`, interleaved with the same JVM build but with restoration turned off. The interleaving was done in the same way as in the DHS-vs-Stock test, 10 times. Finally, through programmatic analysis, differences between the outputted timestamps in each run were computed and averages collected. This gives an idea of how the total runtime of the restore operation is distributed between the individual parts of it.

3.5.1 Pretouch

One worry with this test, was that DHS contributes to a long runtime in other ways than simply how long it takes to run the fix-up procedures. One possibility imagined was that memory pages that are normally read into memory during normal startup, are left untouched until they would have to be paged in later in program initialization. This would make it hard to measure the total impact of DHS.

For this reason, Pretouching was implemented as a way to "collect" all runtime impacts during restoration time. In a "quick-and-dirty" implementation, pages are assumed to be over 2000 bytes, and a for loop iterates the heap, reads one value every 2000 bytes, and uses these to compute a checksum which is printed on standard output (only to avoid these reads being optimized away). This way all pages in the heap are ensured to be paged-in.

Why it was dropped However, this procedure was measured to take insignificant time and abandoned. We suspect this is due to the heap file being kept in memory by the OS anyways, due to the rapidly-iterated nature of the test. As it did not seem to change anything, Pretouching was not included in any of the tests that have been conducted. However, if future work on cold starts is conducted (where the OS file cache is made sure to be emptied, for example), then this technique might prove useful, so the code is left in the artifact.

```
make
    conf=x64-debug
    test=test/hotspot/jtreg/runtime
    jtreg="java_options=
        -xx:+unlockexperimentalvmoptions
        -xx:+useepsilongc
        -xmx1024m
        -xms1024m
        -xx:epsilonmaxtlabsize=8m
        -xx:mintlabsize=8m
        -xshare:on
        -xx:newcodeparameter=3
        -xx:-janiuktimeevents
        -xx:janiukprintstats=0"
    jtreg="test_mode=othervm"
    test
```

Figure 3.4: The command used to run OpenJDK tests relevant to DHS.

3.5.2 Methodology Verification

It is important to be clear on how precise the measurements of time differences actually are. To this end, some code was written to verify the methodology of computing differences. The code attempts to measure "nothing", "a small amount of work", and the same amount of work but repeated a few times. This should give an idea of the precision in the measurements, and whether the times scale linearly as expected [3]. "nothing" was measured to take around 2000 nanoseconds, and the scaling was confirmed. The figure of 2000 nanoseconds gave some perspective to other parts of running code, and contributed to the conclusion that Pretouch was essentially doing nothing.

3.6 OpenJDK Unit Tests

The OpenJDK distribution comes with a substantial amount of tests. For example, a test might be a Java program that is supposed to produce a certain output. All these tests are automated and configurable, and can be run with one command. They are run with the make system. The command that was used to run the tests is shown in Figure 3.4, and the individual options are explained in Figure 3.5. `jtreg` allows us to pass special options through its `java_options` command. `jtreg` is the Java unit test runner.

[3] See `https://github.com/LudwikJaniuk/direct-heap-snapshotting/blob/master/ludvig-diff-05-14.txt#L968-L987`

`conf` means which configuration to test out of the different build types. In this case, the debug one.

`test` specifies if to run only a subset of the tests. Since even that can take a lot of time, it's useful. As stated, the tests in runtime are the only ones relevant to the DHS patch (according to Oracle engineers).

`jtreg="java_options=..."` This passes the Java options necessary for running Heap Snapshotting to the JVM under testing. See Figure 3.3 for an explanation of these.

`jtreg="test_mode=othervm"` That means that the options will be passed to the VM running the tests, not the VM running `jtreg` framework

Figure 3.5: The command line options used in running the tests

Chapter 4

Approach

This chapter explains the implementation in more detail, expanding on implementation choices and trade-offs that were made, as well as explaining how the code does what it does. For more details on certain advanced Java topics such as e.g. *Metaspace*, consult appendix B or online documentation.

4.1 Anatomy of the Snapshot

It seems important to give an overview of the constituents of the snapshot that is saved during heap dumping, and restored in heap restoration. The snapshot consists of three files: the Heap Snapshot file itself, a file with metadata about snapshotted classes and native methods, and a file with metadata about the snapshot. Only some early Java classes are snapshotted at this stage, the snapshot does not contain information that depends on the program being run.

4.1.1 The Heap Snapshot

The heap snapshot is just a binary file that is an exact copy of the Java Heap as it was at snapshotting time. This is thanks to the heap being contiguous in this implementation. If we could not rely on the heap being continuous, this file would probably be more complicated, but it would nevertheless have to contain the information from the heap, to facilitate restoring it.

4.1.2 Class and Native Method Metadata

This file contains a table of class metadata objects, and a table of Native Method metadata objects.

Class Metadata Table

This is a table of every class that was loaded at snapshot time. Each entry contains:

- The `InstanceKlass/ArrayKlass` pointer of the Klass. This is a pointer into metadata that is presumed to be consistent between runs.
- A pointer to the class mirror inside the snapshotted heap
- Their initialization state, as is was at the point of snapshotting.

Native Method Table

The table of Native Method entries contains one entry for each native method in the classes that were loaded at snapshot time. Each entry contains:

- An InstanceKlass pointer to the class that owns this method. This is a pointer into Metaspace, and is assumed to be stable between runs.
- The Method pointer. This is a pointer into Metaspace, and is assumed to be stable between runs.
- A char array describing the memory area this native method was residing in.
- An offset into that memory area, denoting at which point within it the native method was. This and the above are used to find and restore the native method again.

4.1.3 Snapshot Metadata

The snapshot metadata file helps the loading code in loading the snapshot. It contains:

- The start location of the heap
- The length of the heap, as of snapshotting time
- Some oop pointers to global heap objects

Global Oop Pointers to Important Objects

These are pointers to the:

- System Thread Group
- Main Thread Group
- Thread Object

These need to be saved, because they are global important objects residing in the heap, and global pointers to them from outside the heap will have to point to the right place.

4.2 Heap Dumping: Saving the Snapshot

By Heap Dumping, we mean initializing a JVM or a whole Java program, and saving a copy of the Java Heap in persistent storage together with any auxiliary data that will be necessary for Heap Restoring. The data that is saved is called the Heap Snapshot. The point at which Heap Dumping occurs is called the Heap Snapshotting point. That point is supposed to be somewhere during program initialization, before the "actual" work of the program happens. The prototype that has been developed puts this point very early in the initialization process[1]. The computation that has happened before the Heap Snapshotting Point should in principle have been as deterministic as possible, so that any given execution would be able to proceed after it. After Heap Dumping, the program is customarily terminated.

Heap Dumping is similar to the "Checkpoint" part of Checkpoint/Restore, applied specifically to Java, and targeting the Java Heap instead of the whole program state as e.g. CRIU does.

This section is in large parts a commentary on the source code of the patch. For full understanding, it it useful to have to source code handy.

4.2.1 Saving the Heap to File

This process is simple in theory, but heap implementations can be much more complex than textbook examples.

A Straight Write

Epsilon GC is used because it implements the Heap as one contiguous chunk of memory. This "feature" is in no way necessary for Heap Snapshotting, but it reduces the time that had to be spent implementing the logic of dumping the heap. With this "straight write" being possible, we need only to find the start and length of the virtual memory area that is the heap, and write that to a file. However, such a naive saving procedure is probably very fragile. If the memory layout, architecture, endianness and so on of the target OS was different from the one that performed the dumping, then there would probably be lots of crashes. Still, this is just enough for laboratory condition testing.

Epsilon This means that we literally don't have a garbage collector, so long-running programs which allocate and deallocate even moderate amounts of memory won't survive for long under the current implementation. The only thing one can do is to increase the heap size available. This is not seen as a big issue.

[1] In order to understand the current specific temporal location of the snapshotting point, it would be most straightforward to look at the source code. We can say that it is after some native classes have been loaded, and after some static Java initialization methods have been run. It is before thread multiplicity has been introduced, and far before any classes of the specific program have been loaded, let alone any bytecodes of e.g. the main function having been executed.

4.2.2 Saving Auxiliary Data Structures

Apart from saving the heap itself, the heap dumping code needs to save additional data to be able to restore the snapshot later. These are the `JaniukMetadataAboutClasses` structure, called `classesmeta` as a global variable, and the `JaniukDumpData` structure, called `dump_data`. These are both filled in before being written to file in `save_heap_dump`. `dump_data` contains the heap start pointer, heap length, and three global heap objects `system_thread_group`, `main_thread_group`, and `thread_object`, which must be findable upon restore.

The data structure `classesmeta` is more complex. It contains a `JaniukTable` array, an array of `NativeMethodEntry`s, and a check value that has only been used to debug the file saving process, but could theoretically be used as e.g. a version value. Each `JaniukTable` contains information necessary to restore one class. To collect this information, `ClassLoaderDataGraph` is used to execute a closure on all loaded classes. This closure, `JaniukKlassClosure`, receives a Klass pointer, determines if the class should be saved, and writes its `InstanceKlass`/`ArrayKlass` pointer to an entry in `classesmeta.table`, as well as its mirror pointer and initialisation state. The same closure is used to iterate the methods of the Klass, and fill in the array of `NativeMethodEntry`s. The methods of interest are the ones that are native methods. The exact data about them and motivations are described elsewhere in section 4.3.2.

Whenever arrays are used in the snapshot, a relatively simple and low-level mechanism of fixed size arrays with sentinel values is used. This was the simplest to implement.

4.3 Heap Restoring: Starting from the Snapshot

We will now describe the practicalities of the Heap Restore procedure, that is, what happens when we start from a snapshot, instead of initializing normally.

This section is in large parts a commentary on the source code. For full understanding, it it useful to have to source code handy.

4.3.1 Reading the Snapshot Files

As described in section 4.1, the snapshot consists of three files, and all three need to be loaded before the restoring can take place. First, metadata about the snapshot is read. Next the heap snapshot file is memory-mapped over existing heap memory. This replaces any information already there. For this to work, some criteria must be met: the heap snapshot ought to be larger than the current heap, but not larger than the current TLAB. It is larger because it includes more initialisation, and in this way, nothing of the old heap is left. Support for several TLABs is not implemented at this point. A “straight read” with mmap is possible thanks to the heap being contiguous. In principle, `read` could be used

```
JaniukMetadataAboutClasses classesmeta;

class JaniukKlassClosure {
 // Called on each class by loaded_classes_do()
 void do_klass(Klass* k) {
  JaniukTable& next_entry = classesmeta.table[next_slot];
  InstanceKlass* ik = reinterpret_cast<InstanceKlass*>(k);
  next_entry.ik = ik;
  next_entry.mirror = ik->java_mirror();
  next_entry._init_state = ik->_init_state;
  // ...

  // Saves data on native methods
  ik->methods_do(save_method_if_native);
 }
};

void save_heap_dump() {
 // Iterate classes, save java mirors and possibly other class metadata
 JaniukKlassClosure collect_classes;
 ClassLoaderDataGraph::loaded_classes_do(&collect_classes);
 os::write(table_file, &classesmeta, sizeof(classesmeta));

 // Dump the heap
 char* heap_start = heap_start_location();
 unsigned int heap_len = heap_length();
 os::write(heap_file, heap_start, heap_len);

 // Write data about the heap dump
 dump_data.dump_time_heap_start = heap_start;
 dump_data.length_in_bytes = heap_len;
 dump_data.system_thread_group = Universe::system_thread_group();
 // ...
 os::write(dump_data_file, &dump_data, sizeof(dump_data));

 exit(0);
}
```

Figure 4.1: The main operations involved in Heap Dumping. This listing is severely edited for clarity, at the expense of correctness and faithfulness to the actual source code.

instead of `mmap`, but as we don't need to access the contents of the snapshot themselves at the point of restoring, `mmap` seems more appropriate. Note that the `fixed` flag for `mmap` is very much necessary. The heap must be mapped into an exact location in virtual memory, and the operating system needs to support this. For example, the Microsoft Windows function CreateFileMapping seems to lack this feature [Micb][Mica].

After the heap is mapped in, we also read the class metadata table, which will support the synthetic initialization process.

4.3.2 Synthetic Initialization

This is the process of fixing the state of the JVM process up so that initialization can be continued with the mapped-up heap in place. It can be thought of as *"waking up the transplanted brain"*. The main operations that need to be performed are initializing individual classes and fixing native method pointers, but there are other smaller steps as well.

Restoring Classes

When we restore the heap, we overwrite all class instances. Most class instances don't have any pointers to metadata or anything outside the heap, but unfortunately class mirrors are regular heap objects too, and that adds to the complexity. Each mirror has a pointer to the Klass instance it's mirroring, and of course those pointers might be "outdated" when overwriting. In the same way, each Klass instance has a pointer into the heap of its mirror. When we overwrite the heap, those pointers will be pointing to the wrong locations. The pointers from mirrors to Klass instances are not a problem as CDS makes them stable (we currently only restore shared classes, but this would be a problem to be solved in the future). The mirror pointers however must be restored. We call this "restoring mirrors."

Why do we need to restore Klass mirrors? One of the things that is skipped from the original code is `initialize_class` calls. Such a call creates the mirror of a Klass, among other things. The `InstanceKlass` instances on the other hand do exist already, before our snapshot part starts. When we restore we will put the mirrors back in memory. But the `InstanceKlass`es mirror pointers are null at this point. Therefore, we need to update them on where their (already existing) mirrors are in the mapped-in heap.

Restoring mirrors We iterate all the classes in the class table of the snapshot, and restore those that were fully initialized at the time of the dumping. Those make up the state of the snapshotted JVM, and so are expected to function properly. As such, the mirror fields of their InstanceKlass or ArrayKlass instances (both types are supported) must point to their actual mirrors in the Heap. We read the position of those mirrors in the class table too. However, we do not set the mirrors immediately during iteration.

Instead, we do something different. We check if the `class_loader_data` field is null, and if so, we call `load_shared_boot_class` and `define_instance_class` which is Java machinery, to perform a small but necessary part of the initialization of the class. This seems to pertain to initializing the state of the `InstanceKlass` or `ArrayKlass` in Metaspace, as well as registering the `Klass` with global data structures such as the `SystemDictionary`. The important part of `define_instance_class`, found through careful analysis of code and crashes, seems to be that it calls `add_to_hierarchy`; at the very least it seems to register the class with the `SystemDictionary`.

To get back to the mirrors, instead of setting them directly, this mechanism is hijacked, and the function `Klass::restore_unshareable_info` is modified to set the mirrors. The reason for this is that it might be called on more than just the current class, an all of these must have their mirrors set properly. So, we don't set the mirrors only on the classes for which `class_loader_data` is missing, but for all that are relevant for the initialization of these. The `is_restoring_heap_archive` switch is used to trigger that code change.
`restore_unshareable_info` must search for every class it needs to reset in the class table, so the variable `current_table_entry` is used so that at least we can skip the searches in the cases that there is no recursion. A cache hit, if you will.

For convenience, here is the call hierarchy for
`Klass::restore_unshareable_info`:
`Klass::restore_unshareable_info` is called by
`Instanceklass::restore_unshareable_info` is called by
`Systemdictionary::load_shared_class` is called by
`Systemdictionary::load_shared_boot_class`, called by the DHS patch in
`Threads::restore_classes`.

The `quick_init` function Finally, the `quick_init` function is called for each class. This function used to be quite large and try to replicate almost everything that was included in normal Java class initialization, but has been able to be cooked down to only two things. First, linking the class, because we have not yet figured out how to synthesize the linkage (this would be an excellend target for future work). Second, setting `init_state` to `fully_initialized` [Lin+20b] which is a marker that large parts of the existing code rely on.

Restoring Native Method Pointers

An important technique in restoring the current snapshot is *restoring native method pointers.* During JVM initialization, all native methods that are used are registered with the function `Method::register_native`. Then, the `Method` instance that represents that method in Java knows that it is actually a *native* method, and holds a pointer to the actual native library, which has been mapped in.

Due to address space randomization, these pointers will not be the same between different runs, so the pointers to the methods, which lay on the Heap,

are invalid and need to be changed. While one could re-run the specific code of the class which registers the method, this is not a general solution and needs to be manually implemented for every class. Instead, a general solution is implemented. During restoration, and after having parsed the virtual memory areas, all methods are traversed and the native methods identified. Then, their new addresses are computed using the native method table, and the parsed virtual memory areas are used to find a match. This does rely on the same libraries being loaded from the exact same paths. Also, it needs to compare string names of all areas. An improvement which might make this faster is to change to some kind of hash fingerprint routine. There is also the risk for name collisions.

4.4 Common Concerns in Implementation

Locating the Heap The implementation relies on the method `compressedoops::_heap_address_range.start()` to obtain the starting location of the heap. This has the side effect of adding a dependency on CompressedOops. This is only done because there is an easy interface here to find the start of the heap; in fact, the CompressedOops feature should not be necessary at all for Heap Snapshotting. If another way of finding the start of the Heap was implemented, this dependency would disappear.

Parsing VMAs In both Dumping and Restoration, we need to parse the file `proc/self/maps`, present on Unix systems, to figure out all the Virtual Memory Areas available to the process. The reason we are interested is because Native Methods reside in these, but the locations of these areas changes between runs due to address space randomization.

We parse this file in the `parse_proc_pid_maps` function. The algorithm is as simple as opening the file, iterating the lines using `fgets`, copying these into a buffer which we parse with `sscanf`, and saving the data we're interested in, in a `ParsedVMA` structure. This is the string name of the area, the location of the mapping, length, and offset within the file.

After this function has run (which it does as one of the first operations on both Dumping and Restoration), the `memory_areas_have_been_parsed` flag is set to true, to support assertions in parts of code that rely on the result of this function. The parsed memory areas are saved in the global `parsed_areas` array.

4.5 Simplifications, Trade-offs, and Limitations

As this is exploratory work and time was very limited, making as many simplifications as possible was deemed the wisest approach. The largest of these are presented here. They all have in common that they have narrowed the space of conditions under which this implementation of Direct Heap Snapshotting works, but in narrowing it, made the work actually implementable. None of

them should be difficult to solve in theory, but their implementation might of course be work-intensive.

A contiguous heap As described in section 4.2.1, Epsilon GC is used to provide a heap which is just a contiguous memory area. An extra large TLAB is used as described in Figure 3.3 so that the whole used part of the heap is inside one TLAB this early in initialisation. Thanks to this, there is no need to implement support for several TLABs in Heap Restoration.[2]

Unoptimized algorithms Only minimal efforts have been made to optimize the various algorithms introduced. These are mostly search algorithms. The VMA parsing algorithm is pretty straightforward, but might have benefited from finding a different approach to identification than string comparisons. The mirror restoring algorithm is in principle quadratic, albeit with a low constant (optimizations are made to try to find the right class at once "often"). Input sizes are small, and the time taken up by the algorithms is probably not responsible for the largest time wastes. Instead, moving data, reading and writing, is a more likely culprit.

Making friends, silencing asserts In several places, "good design" and encapsulation have been overridden or ignored. If something needed to be changed, the easy road has often been taken of simply adding that class as a friend where needed so private fields can accessed. Some asserts have also been removed. These asserts are well-meaning, but they don't predict the kind of changes this work introduces, so the easiest thing to do is to remove them.

None of this is truly necessary All of these compromises, hacks, and simplifications would obviously not be part of a final addition into the OpenJDK source code. But they have been made with the goal in mind of producing a prototype. Thanks to these shortcuts, the work was possible to complete in this short amount of time, and so they are something to be proud of. The author is confident that if any of this ever leads to real contributions to Java, the capable people who get the job will have no problem to solve these issues "properly". A future thesis student might have to fix some of them in the end, e.g. the TLAB size can probably not be scaled indefinitely, but the others might as well be kept for as long as this is exploratory research.

[2] As TLABs simply offer a "view" into the heap, having multiple wouldn't actually present any challenge for Heap Dumping.

Chapter 5

Results

The main result has been the prototype itself, published on GitHub at `https://github.com/LudwikJaniuk/direct-heap-snapshotting`, in addition to correctness test results assuring that it is relatively correct, performance measurements, and a set of approaches and methodologies that should facilitate future work. The final prototype snapshots the heap during a small part of the initialization of the JVM. Additionally, it already saves a bit of startup time under laboratory conditions.

5.1 DHS-vs-Stock

The *Direct Heap Snapshotting versus Stock* test is an interleaving test in which a restored version of the JVM with the DHS patch applied is measured repeatedly against a completely unmodified version of the JVM. The two things measured are number of executed instructions, and execution time, for a very short-lived program.

The stock version executes on average 820,316,08 machine instructions, whereas the DHS version executes 819,298,56 (that's 101752 instructions fewer on average, or a delta of -0.1240%). The time difference is also negative (DHS runs faster) in all 10 runs, but there is more variation. Stock takes on average 27.878 milliseconds to complete, compared with an average of 27.621 milliseconds for DHS. This is a time saving of 0.257 milliseconds on average, or -0.9217% change in total runtime. See Table 5.1 for full results.

	Machine instructions			Elapsed time (ms)		
Run	**DHS**	**Stock**	**Δ**	**DHS**	**Stock**	**Δ**
1:	819,308,46	820,324,16	−101,570	26.839	26.933	−0.093
2:	819,319,51	820,301,65	−98,214	27.239	27.499	−0.260
3:	819,264,52	820,330,92	−106,640	27.452	27.578	−0.126
4:	819,319,30	820,331,83	−101,253	27.649	27.818	−0.169
5:	819,278,20	820,319,09	−104,089	27.818	27.918	−0.100
6:	819,294,67	820,306,58	−101,191	28.037	28.071	−0.034
7:	819,306,98	820,333,98	−102,700	27.745	28.114	−0.369
8:	819,324,16	820,312,79	−98,863	27.741	28.129	−0.388
9:	819,272,90	820,305,05	−103,215	27.833	28.288	−0.455
10:	819,296,91	820,294,76	−99,785	27.881	28.469	−0.588
Avg:	819,298,56	820,316,08	−101,752	27.621	27.878	−0.257

Table 5.1: Complete time measurements from the DHS-vs-Stock test. Each row represents an average as measured by perf stat, from 400 runs of Stock, followed by 400 runs of DHS. Lowest differences highlighted in red.

5.2 Moments

In order for Snapshot Restoring to succeed, some "fixup" operations must be performed to repair the state. The runtime of these operations is a limiting factor in how much time is saved (or lost) in the end. Therefore it is interesting to analyze which of these takes the longest to run, as it would be the primary suspect in future optimization efforts. To this end, timestamps were printed between all the major distinct "time periods" during restoration, and then time deltas were computed and averaged into Figure 5.1.

History note This analysis already proved useful once. When run initially, it showed that the "Read Classes Metadata" period was responsible for over half of the restoration period. This prompted some investigation, and it was discovered that overcautious macro sizes[1] had led to a class metadata file size of over 2 megabytes, which was taking a long time to read into memory. But only a fraction of that file was used, the rest was just buffer space. These macros were changed to only as large values as necessary, and the time taken by the read operation in turn decreased to a small fraction of the restoration time.

Results As seen by Figure 5.1, the invocation of a Java static method is responsible for the largest contribution to runtime, followed by the time taken to restore all the classes, then by the time taken to parse VMA information from `/proc/self/maps`, and finally by the restoring of native functions.

Additionally, we have averages on the two total measures presented in Table 5.2: The synthetic restore operation was computed as taking on average 1.066 ms, while the normal (no-restore version) equivalent piece of code, when it is not skipped, took on average 1.191 ms. The difference between these two numbers is 0.12 ms, but one should look at the DHS-vs-Stock test before making too hasty assumptions about this being the total time saved of the runtime.

5.3 Correctness Tests

5.3.1 jtreg Test Results

The DHS patch passes all 709 unit tests pre-packaged with OpenJDK. These are the tests in the `/runtime` directory. According to Oracle engineers, these tests are the only ones in the test suite that would be relevant to the changes introduced by Direct Heap Snapshotting.

5.3.2 Evaluation on Test Programs

The tested programs included a distribution of Apache Tomcat 9 [Fou20], a SpringBoot [Spr20] server, and minesweeper game written in Java, found online. Anyone interested is encouraged to test on any Java programs of their choosing.

[1] Specifically, `J_NUM_NATIVE_METHODS` = 2000 and `J_MAX_STORED_PATH_LENGTH` = 1000

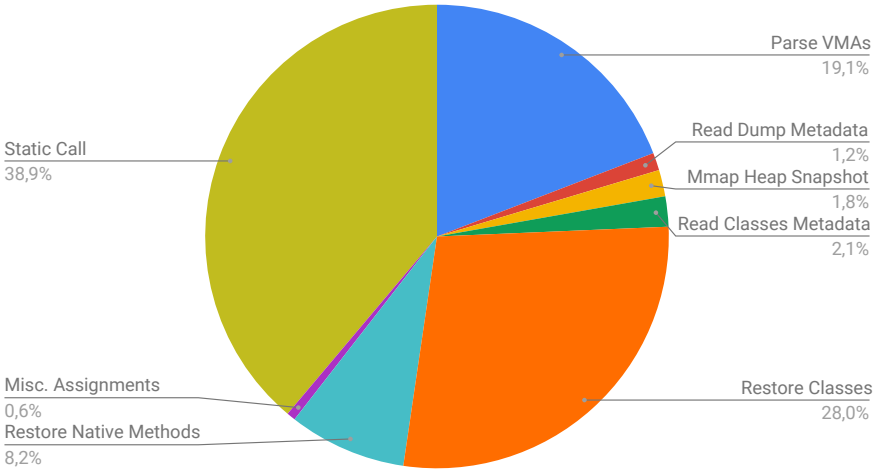

Figure 5.1: Breakdown of periods during restoration process. Average percentages of the restoration duration shown. Slices in chronological order clockwise starting at "Parse VMAs".

Period		**Time (ns)**
Parse VMAs		203,441
Read Dump Metadata		13,039
Mmap Heap Snapshot		19,623
Read Classes Metadata		22,388
Restore Classes		297,506
Restore Native Methods		87,261
Misc. Assignments		6507
Static Call	+	412,857
Synth	≈	1,066,010
Normal	-	1,191,985
Synth - Normal	=	−125,974

Table 5.2: Averages of time periods computed in the Moments test, including an average of the whole restoration period ("Synth") as well as of the whole snapshotted period when that is being run ("Normal"). Intended reading of the middle column: "All the periods sum up roughly to Synth, and lastly the difference between Normal and Synth is presented".

All tested programs were able to run without issues on the modified JVM, except of course they would eventually run out of memory, since Epsilon GC is used.

Chapter 6

Discussion

6.1 Correctness Confidence

Perhaps the biggest goal of this research is to try to get confidence that the heap-restoring approach works. We are doing a very unorthodox thing: overwriting the whole heap of a Java program during initialization. How could we ever be sure that his has been done "right", leading to a totally correct and consistent state? After all, changing just one bit of a program's state can completely change the rest of the execution. At the same time, we're not aiming at bit-exact equality, since some parts of the state depend on the environment and it is correct for them to be different between runs. In one sense, we can never be sure that this is correct. However, in another sense, an erroneous reset of the heap would probably manifest itself with very visible errors. We are doing this restore at a very early point in JVM initialization, so it is reasonable that disturbances in program state now would have time to compound and influence the rest of initialization and ultimately program execution. Therefore, we can be reasonably confident that this small snapshot is indeed restored correctly, since not only do test programs execute without problems, but the test suite also finds no failing tests. Nevertheless, it is possible that some state inconsistency lies dormant, but would cause bugs in very specific situations that have not been tested. However, this is the case with all software except perhaps formally proved programs...

6.2 Sensitive Memory

A research direction for the future might be using a type system approach to track which parts of data should be part of the snapshot. Some parts of the state should definitely not be kept, for example the time of program start, other environment-dependent values, sensitive data such as passwords or cryptographic keys. Conceptually, one could maybe annotate the sources of such data in the Java API, and then let the type system detect all other values com-

puted dependent on these. Something like a "taint-tracking system", and then we could save everything that wasn't "tainted". These are just some visions that were discussed in early meetings, but have not been investigated at all in the actual work in this thesis.

6.3 DHS-vs-Stock

In the comparison in startup time between DHS and a Stock JVM, the test results show consistently that this prototype of DHS improves startup performance, but the difference is minuscule. The relevant aspect however is that even when capturing such a small part of the startup in a snapshot, timing improvement is achieved under at least some conditions. If a larger portion of the initialization sequence were snapshotted successfully (without requiring much more expensive fixup procedures), large startup time savings would abound.

Criticism: file caching It should be noted that mapping the heap up probably has what one could call an unfair advantage in this testing setup. Since the program is re-run hundreds of times, it is very likely that the heap dump is cached by the OS in RAM memory, in effect not requiring a disk read. One might therefore argue that the time gain is invalid, since the use case we are considering is precisely a cold start scenario; if e.g. the given microservice is to be run hundreds of times in succession, current microservice frameworks can already handle that very well and allow the calls to happen without the need of restarting the JVM in the first place. Our response would be that on one hand, the time results here are again not the main result, but on the other hand, this OS-caching of the snapshot file could very well be implemented as a feature.

Unpursued path: daemon idea In the early stages of this thesis, the plan for a first prototype was actually to optimize CDS loading, by writing a daemon that would keep the CDS archive in memory, and then just mmap that into the initializing JVM at the right point in time. The idea was that in a microservice environment, such a daemon could be constantly on, keeping e.g. a CDS archive available for faster start, and since that could be the same, shared, CDS archive, it could be used as a resource between many starting JVMs and would not take much space. This prototype was prioritized away, but would still have been an interesting thing to implement. Perhaps future work could try it, seeing as it should be an easy first step to "get your feet wet". The daemon could, instead of keeping the CDS archive in memory, keep the heap snapshot instead. Or why not both? In retrospect, this idea is also very close to the Nailgun approach.

6.4 Moments

The moments test shows which optimization efforts might give the best return-on-investment. One should note that the mere act of printing timestamps likely

affects the total runtime. Therefore the total runtime resulting from this test should not be used for analysis in itself, instead one should look to the DHS-vs-Stock test for a comparison focused on total runtime, with timestamp printing turned off.

Advice on further optimization The static call is a call to the Java method `Finalizer.janiuk_funtion1`, which is the author's own added method that explicitly runs the static operations of `Finalizer`. They make sure thread state is set up correctly. It's possible that a different way could be found to achieve this without calling into Java, but this would require careful analysis of the side effects, as well as advice from Oracle engineers. If one wanted to optimize "Restore Classes", one would need to analyze deeper what actually takes time there, as this period recursively iterates all the classes loaded and performs some operations. It is not currently known whether one of the operations or the iteration itself is the main culprit. "Parse VMAs" might be the period with the greatest chance of being successfully optimized, as it's possible that this information is already parsed somewhere in the JVM codebase, or that the parsing algorithm can be made more efficient. This period is also a direct dependency for the "Restore Native Methods" operation, so if that one were somehow made unnecessary, Parsing VMAs could also be skipped. But this is unlikely.

6.5 Reliability of Runtime Differences

One might expect the difference between the "Normal" and "Synth" time periods in the Moments test to match approximately the difference in runtime measured in the DHS-vs-Stock test. After all, this is the time in initialization where changes are made. However, this is not the case. Synth runs for 0.126 milliseconds fewer than Normal, whereas the difference in runtime in DHS-vs-Stock is 0.251 milliseconds. It looks like we're saving even more time than what what we see through the Moments test. So where does the difference come from?

On one hand, there might be other sources of change in the total runtime. The JVM does many things lazily, such as resolving symbols, or JIT compilation. Some of these things might have happened already during Normal, thus not needing to be done later, but since Synth skips a lot of bytecode execution, they need to be done later in the program's life time. That could have been one explanation of unaccounted-for difference — if we were saving *less* time than indicated in Moment.

Curiously, the situation we have is the opposite. In the end, one must therefore also look at the large variation in runtime and conclude that comparisons cannot be made directly on the absolute value of the runtime difference. Perhaps with even more runs and stringent test conditions it could be measured (one could use a dedicated test server instead of a personal laptop), but it is not a goal of this thesis to measure these values with such precision. They would

be much different in a real setting anyway, due to all the laboratory condition changes.

6.6 Criticisms

Will this be integrated into Java? Chances are that Oracle would not take this approach. Oracle have high requirements on stability and robustness, so if they choose to implement Heap Snapshotting, they need a way to prove to themselves that it is safe. Despite the tests that have been done, it is totally conceivable that problems would arise under other, untested conditions. Instead, JVM developers might focus on refactoring environment-dependent initialisation such as native function registration to later in the startup process. This way, the first part can be more safely snapshotted.

Cold starts have not been tested Both the Moments and the DHS-vs-Stock tests have been conducted with a high degree of repetition, in an attempt to minimize variance in other factors affecting runtime. However, this means that the operating system has had a brilliant opportunity to cache all the disk accesses, instead probably serving the heap snapshot from memory. In effect, the test does perhaps have an unfair advantage as totally cold start scenarios might still have to serve a snapshot file from disk. It would definitely be valuable to repeat the DHS-vs-Stock test in a totally cold-start scenario, ensuring that all OS file caches are emptied between runs. However, it would also be possible to set up a real deployment with a snapshot kept always in memory, thereby avoiding slow disk reads.

Limited testing Another fair criticism of the results is that very limited testing has been carried out. Indeed, the net time gain might not be replicated on other machines or systems, and there might be programs that have not been tested which do crash when under Heap Snapshotting. In fact this is likely. However, what is important is that this much progress was achievable in a comparatively small amount of man-hours. This points to a real possibility for improvement in the JVM, and this point is not diminished if such counterexamples are found.

Microservices rarely restart This work focuses on JVM startup optimization and addresses serverless deployments as a use case. However, the overall goal of many microservice frameworks is to fulfill microservice requests continuously without the need of cold starts. If cold starts are minimized, startup optimization yields little return on investment.

While this observation is valid, the continuous running of a microservice server requires memory to be occupied, a tradeoff which might be prohibitively costly for services that are used sporadically. Additionally, in settings where one must guarantee that no state is kept between service invocations, complete teardown and restart between invocations might be necessary. One example is the

Secure Multi-execution framework of Devriese and Piessens which guarantees noninterference [DP10].

Can pointers keep their meaning? The reader is encouraged to visit chapter C for an extended discussion on the feasibility of larger Heap Snapshotting. The discussion goes into detail on potential problems that may arise with the many different kinds of references within the JVM, and whether those issues will in theory be solvable. While there are no certain answers, the discussion argues in favor of this being the case.

Chapter 7

Conclusions & Future Work

Direct Heap Snapshotting is a viable strategy for reducing startup time in the OpenJDK HotSpot JVM. While the HotSpot codebase is complex, it was possible for the author to implement a DHS patch for it in a few months.[1] Thus the complexity of implementation is high but not prohibitive. A lot more work would be required for a complete prototype, but even this small version saves some startup time already. More broadly, this work shows yet another time the potential in Checkpoint/Restore or similar schemes, and highlights the unexplored potential in improving startup time by applying these ideas to yet more technologies. The old mentality of not considering startup time an issue ought to be abandoned, as short-lived programs become more common. It is also an ergonomics issue, not only for programmers but also for all users of Java programs.

Future work If continued, this research could reduce JVM startup time, which in certain applications such as microservices could lead to big savings on total computation amount. Memory footprint savings are also easy to imagine. A starting point is clear: pushing the snapshot point forward is the first most obvious target for future work. The work on this was stopped only due to lack of time, and not any practical problem, so it is likely that there is much potential there.

7.1 Roadmap

Milestone: snapshot of JVM startup An important milestone will be when the whole JVM startup sequence can be snapshotted. This will be defined as the point when the first bytecode of the program gets executed (i.e. not a bytecode which is part of the usual initialization of the JVM). In a simple program, this is the main function, and in more complex programs this might

[1] Granted, with large amounts of support from the amazing Oracle engineers at the JPG Group in Stockholm

be e.g. the first static initializer of a class. Even this seems like an ambitious goal, as initialization becomes much more complex before it reaches here; for example, multithreading starts to play a bigger role.

Continuation: snapshot of program initialization Further on, an ambition can also be to snapshot further than the JVM itself; even more time gains can be had if e.g. library initializations are snapshotted as well. This might be implemented with a `SnapshotHeap()` API that lets the programmer declare up to where snapshotting would be safe, as afterwards the program depends on non-deterministic data. With such an approach, even program-internal (i.e. after libraries) parts could be snapshotted, as long as they are deterministic enough.

Detecting snapshot unsafety The API approach shifts responsibility on the programmer to know intricate details about JVM initialisation. This seems prone to error. Ideally, the heap snapshotting framework would detect if the snapshotted area of the code will be able to be restored safely. While desirable, it is not clear at all how to achieve this, but some ideas spring to mind. Perhaps a type system approach, tagging "safe" and "unsafe" data for snapshotting and then propagating those labels using static analysis could work?

7.2 Challenges

Implementation cost As the snapshot is pushed later and later in the initialization sequence, it is possible that each new step will be harder to restore than the next. Certainly, many important issues are not necessary to handle this early on, for example multithreading. It might be so that the number of things that need to be fixed turns out the be extremely large, and that they are of very varied character, not admitting of generic solutions. We cannot predict this.

Fixup cost Apart from the difficulty of implementation, the problems that arise from later snapshotting might turn out to require solutions which simply take too much time in restoration.

7.3 Project Leyden

It will be interesting to follow what Project Leyden leads to and what design decisions will be taken. The fact that Oracle has initiated a large project on this topic is an indicator of the seriousness of the underlying problem.

7.4 Research Approaches for Future Work

We hope that this paper will help in future work on Heap Snapshotting. Many best practices, helpful tips, troubleshooting strategies, and other useful resources were developed during this work, but these are not suited to be included in a thesis. Instead, the interested reader should look out for a series of blog posts that the author aims to publish together with the JPG Group.

Appendix A

Build instructions

A.1 Building

First, make sure you can build a stock JVM, instructions can be found in the OpenJDK documentation [Ope]. Then, apply the DHS patch on top of the commit indicated in the readme, specifically, commit `0905868db490` in mercurial. It is also recommended to update the hard-coded file paths for the snapshot (variables `heap_dump_path`, `table_path`, and `dump_data_path`) to paths which actually exist on your computer. After that, build normally. The working directory from which one builds is the `jdk` directory, the one that contains subdirectory `build`.

The build command can be e.g. `make conf=x64-debug jobs=7 jdk-image`. Of course, this requires that you have done `configure` first as per normal build procedure. Also, consult Figure A.1 to replace `x64-debug` with the appropriate build type suffix depending on the situation.

`slowdebug` (`linux-x64-slowdebug`[a]) Good for inspecting what happens in memory, preserves the most low-level details, but is sometimes prohibitively slow.

`debug` (`linux-x64-debug`) Is used to run tests.

`product` (`linux-x64`) For time testing. All the assertions are gone.

`release` I have been advised not to use this by Oracle engineers as it performs extra steps that are irrelevant unless one is actually shipping a JVM to customers.

[a] or `linux-x86_64-server-slowdebug`

Figure A.1: Different kinds of builds are appropriate for different goals.

Appendix B

The JVM in Depth: A Focus on Internals and Startup

This chapter has the goal of summarizing and giving context on the parts of HotSpot VM internals which are relevant to the understanding of Heap Snapshotting. Of course, official Oracle documentation or other educational materials probably offer a better presentation of these concepts, so the reader is encouraged to seek such materials out if interested.

B.1 Memory Areas of the Java HotSpot VM

There are many distinct "memory areas" in the HotSpot VM: the Heap, the Stack[1], the Constant Pool, Metaspace, the Code Cache, and so on. For this thesis, the most important parts are the Heap and Metaspace, as the Heap is what's being snapshotted, and Metaspace contains most of the runtime auxiliary supporting structures that are intricately linked to the Heap, and these links must be restored correctly. Figure B.1 gives an example of what might be in the Heap and the Metaspace in a hypothetical application. This imagined application has only one class, "Person", which is instantiated three times. The figure also exemplifies class mirrors and native class representations (i.e. *Klasses*), covered later in this chapter.

Heap The heap consists of Objects, and is generally divided into Regions[2] to facilitate garbage collection [Mic06]. It is not necessarily contiguous in memory,

[1] Not to be confused with the usual stack and heap of a process, which we will call the "C++ heap/stack" in this document if they need to be mentioned. So, all these JVM memory areas of course reside in the C++ heap.

[2] If implementing Heap Snapshotting on a discontinuous heap, one might use regions as useful "chunks" of heap memory.

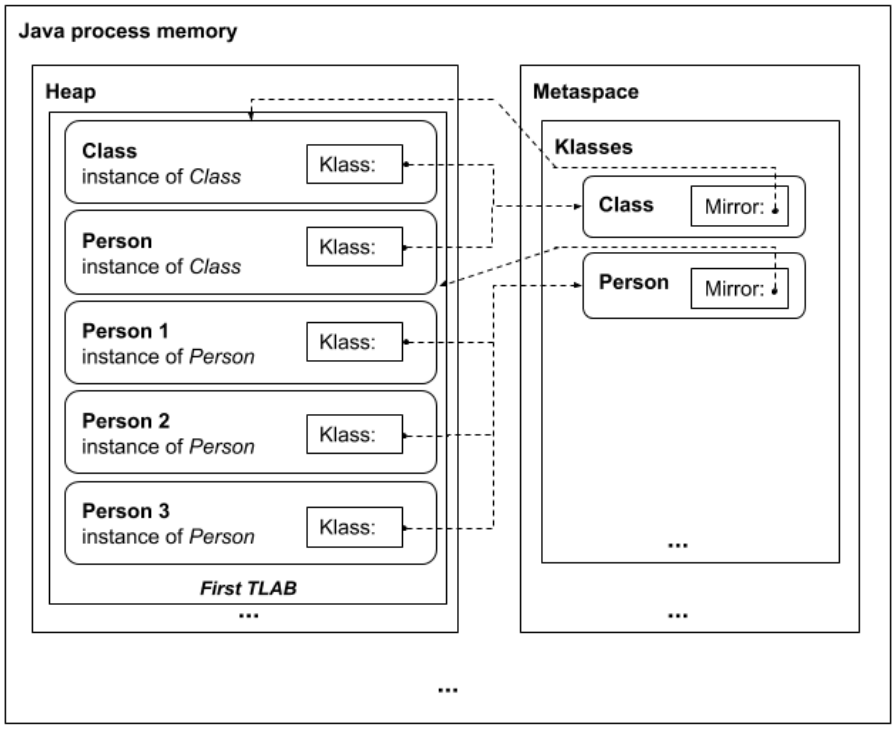

Figure B.1: An illustration of the memory areas of the OpenJDK JVM most relevant to this thesis, and the main Klasses, Class Mirrors, and instances in a hypothetical application. The many ellipses remind us that this is a very incomplete view of the VM's memory state.

but within this prototype, it is assumed to be. The Objects are all instances of some class, and for each class there is an instance of the class Class in the heap. This instance is referred to as the "Java mirror" of that class. Java mirrors are called so because they are the representation of the class in the Heap, or Java-land, as opposed to the memory structures that make up the class in Metaspace, or native memory (see fig. B.1). Those are instances of the C++ class `Klass`, or one of its in descendants. Much of the information describing the class (e.g. the number of fields) is duplicated between these two representations, thereby the term "mirror". The class `Class` has a mirror too, which is an instance of itself.

TLABs To speed up allocation, each thread has a Thread Local Allocation Buffer, which is a pre-reserved amount of Heap memory which that thread has exclusivity over. Therefore it can allocate more memory from there without locking. Only when the TLAB runs out, does the thread have to get another one.

Metaspace Metaspace is where most native auxiliary structures reside, which make up big parts of the functioning of the JVM, but which are not accessible from within Java. So, an InstanceKlass is not accessible from Java, but its mirror instance of `Class` is. Memory in the Metaspace is allocated and collected differently than the Heap, and does not fall under Garbage Collection as the Heap does.

Other areas There are many other memory areas in the JVM, each with their specialized use, but an understanding of the ones above should suffice to follow this document.

B.2 The Role of Classes

Classes are a very important part of runtime state. Methods, fields, and so on are all stored in some class. The heap consists of Java objects, which are simply instances of classes, and so if we are overwriting the heap, we must first make sure that all the supporting structures of the instantiated classes match up.

Native representations and mirrors Every loaded class in the HotSpot VM is represented as a "native representation" in Metaspace, and also as a "Java mirror" on the Heap. The native representation is an instance of the C++ class `Klass` or one of its subclasses, while the mirror is a regular Java object. Since all Java objects must be instances of some class, all class mirrors are instances of the Java class `Class`. Perhaps confusingly, this applies for the mirror of the class `Class` too - its mirror is an instance of itself.

ArrayKlass and InstanceKlass `Klass` has the polymorphic subclasses InstanceKlass and ArrayKlass, among others. So the existence of class A in a running JVM entails the existence of an instance of C++ datatype InstanceKlass corresponding to it, and if there are to be arrays of A, A[], then there must also be a corresponding instance of ArrayKlass. The "class" A[] is therefore treated as a separate class, and also has its own, separate, mirror.

B.3 Oops and OopHandles

"oop"s in OpenJDK are managed pointers to heap objects [Ora12]. An oop itself is a pointer into the Java heap. It might be compressed with the CompressedOops feature. Nevertheless, if one looks at e.g. InstanceKlass, it has a `_java_mirror` field which is an `OopHandle`, not an `oop`, and this represents quite an interesting mechanism. The OopHandle encapsulates the oop itself and stores it separately. Why is an extra indirection even necessary? The heap is, of course, under garbage collection. Objects might be deallocated or move around as part of their lifecycle, and direct pointers to them from metadata would be problematic as they would lose their relevance in such events. Therefore, when e.g. `InstanceKlass` wants to point to an object, it creates an `OopHandle` instead.[3] If we look at the Handle constructor which takes an oop, we see it runs `thread->handle_area()->allocate_handle` to allocate memory for the oop. This handle area is therefore a centralized location where the raw `oop`s actually reside, so when they need to be changed or nullified, one might simply look here.[4]

How the `OopHandle` indirection is useful In principle, when restoring the heap (and especially later, when it's not so early anymore), we will have to update where things are in the heap. We might definitely want to scan these HandleAreas then and just update oops in there. That might very well be the only reasonable way. In this thesis we have relied e.g. on persistence of InstanceKlasses, but something like this might be necessary further on.

B.4 Class Loading Roadmap

Chapter 5 of the The Java Virtual Machine Specification [Lin+20b] provides the authoritative resource on how class loading is supposed to take place. In short, after being compiled, a class is a classfile which is a binary format. These classfiles contain bytecodes, which are what the JVM needs to parse, verify, and then use to initialize the classes in memory. A representation of the class is created in memory, and the class is loaded. This might be done by the built-in *bootstrap classloader*, or by a custom classloader (within this thesis, only the bootstrap classloader has been used).

[3] This is well described in the source code: `src/hotspot/share/runtime/handles.hpp:37`.
[4] In truth, there are several such areas and it's all very complicated.

After the process of creation and loading, the class must be *linked*. This involves *verification* of the binary representation, *preparation* which entails initializing static fields[5], *resolution* of some symbolic references, fields, interfaces, methods, and so on, and other steps.

Finally, there is *initialization*, which means executing the class initialization method, typically named `<clinit>`. One can expect these to take some time as they involve executing bytecodes.

A central idea of heap snapshotting is precisely to want to avoid having to run most if not all of these steps, since their effects on the heap ought to be gotten from Heap Restoration.

B.5 Class Data Sharing

Abbreviated CDS, Class Data Sharing is a HotSpot feature which reduces start-up time and memory footprint by pre-loading a set of well-known classes into an internal representation during installation time, and dumping that representation to a file. During startup, the file is memory-mapped in, which is much faster than actually performing operations such as parsing and verification at start time [Ora93].

AppCDS is an extension to CDS which allows user-defined classes to also be archived in a similar manner. This process has to be done manually but can improve startup as well [Ora17a].

Stable pointers As CDS memory-maps its internal representation of classes into the same location in memory every time, it turns out to have the side effect in practice of making the memory addresses to these objects the same between runs, at least on the same machine and the same installation. This effect is exploited in this thesis.

[5] A big part of Heap Snapshotting is either "tricking" the state into thinking this has happened, or actually re-running static initialization manually.

Appendix C

Can Pointers Keep Their Meaning?

This chapter is perhaps a bit philosophical, and involves considerations on the overall feasibility of heap snapshotting. The main problem is that of pointers of all kinds keeping their meaning between runs. We work under an assumption of a more or less complete isomorphy existing between the entity graph[1] of the process at dumping time and at restoration time. The success of CRIU shows this to be true. In this chapter we explore some thoughts about the problem in general, not just in terms of where we are in this research. However, we use examples from the research for illustration, and we focus on the context of the JVM and the Heap specifically.

The information saved in the snapshot is only valuable insofar as references and pointers will keep their meaning when restored. For many kinds of references and pointers, this is not directly the case. What broad categories do we have, and what solutions exist? The issue is twofold: 1) Are references to each entity valid upon restoring, and 2) If not, can all the references to it be identified and updated?

C.1 Native Function Pointers

Let's start with an example. Method representations on the heap which refer to native methods, save the native pointer to their implementations in native libraries. Those pointers will be out of date upon Heap Restore due to address space randomization. In this case, the problem can be resolved by saving their *identities*, that is the names of their memory areas together with their address within them, and then computing their new locations from `/proc/self/pid`.

[1] We are avoiding the term "object graph" as that alludes to only the graph of Java Objects within the heap. Here we want to talk about these, in addition to all the entities that have any sort of reference to and from the heap. This includes entities in Metaspace, native artifacts in the operating system, and possibly many other examples.

Thus is problem 1 solved. We also know that the only reasonable references to these are in `Method` objects which represent native methods. Luckily, these keep their addresses intact between runs. Therefore, we can save a table of them and find them to update their native method pointer. Thus is problem 2 solved as well. This approach is described in 4.3.2.

Let us now consider some broad categories of references.

C.2 Pointers Within the Heap

These keep their relevance, with the caveat that one might imagine if the base of the heap changes, then these would all be destroyed too. However, this problem has not been observed, because the base of the heap doesn't change (at least within the conditions of this work). Also, because object pointers are Oops and Oops are managed pointers, the location of the indirection is what must actually not change, and again, this seems not to be the case. If the heap were to move its base, a fixup step of adding an offset to all the oops would be necessary (or this might be solved with a lazy solution, e.g. a write barrier).

C.3 Pointers from Metaspace to Heap

Generally, the Heap cannot be expected to look exactly the same at the same point in time between two runs. The same "set of objects" could be expected to be there, but the way they are organized can have depended greatly on order of operations, scheduling, memory allocation, etc. Therefore, in principle each pointer into the heap from outside needs to be found and updated upon heap restore.

C.3.1 Pointers to "Global Singleton Objects"

The easiest situation is if objects are singletons and have global identity. This would be for example global important objects such as the `ThreadGroup`. We have to save their location and restore them, and as long as they are few, this is very cheap to do. What makes this operation possible is that each such object has a clear "identity": we can talk about *the* `ThreadGroup`, and by the virtue of this identity, save and restore a pointer to it.

C.3.2 Pointers to "Identifiable Objects"

Most heap objects are by far not singletons, so if an object is referenced from outside, how should one find the corresponding object in the restored heap? Well, as long as the object has any other external way of being identified, this is doable. An example are class mirrors. While we couldn't find the mirror object at restore (short of doing a total search, perhaps), we can save a table at dump time stating "the mirror of `InstanceKlass` x is at address y". This external

identity of the mirror object (being the mirror **of** persistent entity x) is what allows us to find it and restore it.

C.3.3 "Unidentifiable Objects"

So what if an object in the heap is referenced by the Metaspace, but has no external identity that we can rely on? It certainly sounds like such objects would throw a monkey wrench into the machinery of Heap Snapshotting. However, it turns out that no such examples have been found so far, and indeed it is very hard to come up with even thought examples that would satisfy this category.

More formally, to find an Unidentifiable Object, there would have to be an entity (e) in the Metaspace, that references another entity (h) in the Heap. However, e is not always in the same location and has no unique identifier by which it can be found. One wonders if this is even possible. After all, if e has an important reference to h, but has no global identity, them it must be referenced by some other entity (say e'), because otherwise how would it ever be found and accessed? Now e' might be identityless too, but then the same argument applies, and it seems if the reference to h is to have some bearing on the program, there must be some chain of references that ends in some entity E that does have persistent identity. Then e could be found through the identity of E and following the same chain. It seems the natural conclusion is that Unidentifiable Objects are by definition garbage, and we wouldn't have to deal with them.

One might try to formulate a thought experiment from an array of indistinguishable objects, say a pool of resource objects residing on the heap. But even then, their locations within the containing array serve as a "good enough" identity, for if they are indistinguishable, then no one can say that *this* object was not first, *the other one* was. And if they *are* distinguishable, then from that distinguishability, identity ought to follow.

C.4 Pointers from Heap to Metaspace

All objects in the Heap are identifiable, by virtue of having a stable position within the snapshot, so for this kind of pointers problem 2 is solved by default. Problem 1 might still apply. Easy situations are e.g. InstanceKlass pointers which have persistence, but this might not be the case in the future, and then there might be more work required to find the actual targets of the pointers and solve problem 1.

C.5 Other "Pointers"

There is also the question of handling other types of references to/from the heap, including e.g. file handles, sockets, and references to volatile memory regions. The handling of these experiences the same problems even to a greater extent, and whether they should be attempted or not is a design question. Certainly a socket represents state setup outside of the program itself, so it seems like a

poor candidate for inclusion in Heap Snapshotting. On can look at CRIU for inspiration on how they have chosen to handle these issues, keeping in mind that their goals and purposes differ from this work.

Bibliography

[Ora93] Oracle. *Class Data Sharing*. https://docs.oracle.com/javase/8/docs/technotes/guides/vm/class-data-sharing.html. Accessed: 2020-05-30. 1993.

[BW01] M Bozyigit and M Wasiq. "User-level process checkpoint and restore for migration". eng. In: *ACM SIGOPS Operating Systems Review* 35.2 (2001), pp. 86–96. ISSN: 0163-5980.

[Mic06] Sun Microsystems. *Memory Management in the Java HotSpotTM Virtual Machine*. https://www.oracle.com/technetwork/java/javase/tech/memorymanagement-whitepaper-1-150020.pdf. Accessed: 2020-05-28. 2006.

[AAC07] Jason Ansel, Kapil Arya, and Gene Cooperman. "DMTCP: Transparent Checkpointing for Cluster Computations and the Desktop". In: (2007).

[DP10] D. Devriese and F. Piessens. "Noninterference through Secure Multi-execution". In: *2010 IEEE Symposium on Security and Privacy*. 2010, pp. 109–124.

[Ora12] John Rose (Oracle). *CompressedOops*. https://wiki.openjdk.java.net/display/HotSpot/CompressedOops. Accessed: 2020-05-30. 2012.

[Pot12] Priit Potter. *How many Java developers are there in the world?* https://plumbr.io/blog/java/how-many-java-developers-in-the-world. Accessed: 2020-06-03. 2012.

[Pic+16] Simon Pickartz et al. "Migrating LinuX Containers Using CRIU". In: *High Performance Computing - ISC High Performance 2016 International Workshops, ExaComm, E-MuCoCoS, HPC-IODC, IXPUG, IWOPH, P^3MA, VHPC, WOPSSS, Frankfurt, Germany, June 19-23, 2016, Revised Selected Papers*. Ed. by Michela Taufer, Bernd Mohr, and Julian M. Kunkel. Vol. 9945. Lecture Notes in Computer Science. 2016, pp. 674–684. DOI: 10.1007/978-3-319-46079-6_47. URL: https://doi.org/10.1007/978-3-319-46079-6%5C_47.

[Ora17a] Ioi Lam (Oracle). *JEP 310: Application Class-Data Sharing*. https://openjdk.java.net/jeps/310. Accessed: 2020-05-30. 2017.

[Ora17b] Oracle. *jlink*. https://docs.oracle.com/javase/9/tools/jlink.htm. Accessed: 2020-06-04. 2017.

[Red17] Claes Redestad. *Startup Challenges with Claes Redestad*. https://youtu.be/3r_tHGtpU7A?t=320. Accessed: 2020-06-04. 2017.

[Akk+18] Istemi Ekin Akkus et al. "SAND: Towards High-Performance Serverless Computing". In: *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. Boston, MA: USENIX Association, July 2018, pp. 923–935. ISBN: 978-1-939133-01-4. URL: https://www.usenix.org/conference/atc18/presentation/akkus.

[Fre19] Inc. Free Software Foundation. *GNU Emacs Lisp Reference Manual*. https://www.gnu.org/software/emacs/manual/html_node/elisp/Building-Emacs.html. Accessed: 2020-05-22. 2019.

[Wim+19] Christian Wimmer et al. "Initialize once, start fast: application initialization at build time". In: *PACMPL* 3.OOPSLA (2019), 184:1–184:29. DOI: 10.1145/3360610. URL: https://doi.org/10.1145/3360610.

[Fou20] Apache Foundation. *Apache Tomcat 9.0.35*. https://tomcat.apache.org/download-90.cgi. Accessed: 2020-06-03. 2020.

[Lin+20a] Tim Lindholm et al. *The Java® Virtual Machine Specification Java SE 14 Edition*. https://docs.oracle.com/javase/specs/jvms/se14/html/index.html. Accessed: 2020-05-22. 2020.

[Lin+20b] Tim Lindholm et al. *The Java® Virtual Machine Specification Java SE 14 Edition*. https://docs.oracle.com/javase/specs/jvms/se14/html/index.html. Accessed: 2020-05-22. 2020.

[Rei20] Mark Reinhold. *Call for Discussion: New Project: Leyden*. https://mail.openjdk.java.net/pipermail/discuss/2020-April/005429.html. Accessed: 2020-07-05. 2020.

[Spr20] Springboot. *Spring Initializr*. https://start.spring.io/. Accessed: 2020-06-03. 2020.

[Fac] Facebook. *Nailgun Background*. https://github.com/facebook/nailgun. Accessed: 2020-06-06.

[Gra] GraalVM. *GraalVM*. https://www.graalvm.org/. Accessed: 2020-06-06.

[gro] HotSpot group. *The HotSpot Group*. http://openjdk.java.net/groups/hotspot/. Accessed: 2020-06-06.

[Koz] Vladimir Kozlov. *JEP 295: Ahead-of-Time Compilation*. https://openjdk.java.net/jeps/295. Accessed: 2020-06-06.

[Lam] Marty Lamb. *Nailgun Background*. http://www.martiansoftware.com/nailgun/background.html. Accessed: 2020-06-06.

[Mica] Microsoft. *CreateFileMappingA*. https://docs.microsoft.com/en-us/windows/win32/api/winbase/nf-winbase-createfilemappinga.

[Micb] Microsoft. *Creating a File Mapping Object*. https://docs.microsoft.com/en-us/windows/win32/memory/creating-a-file-mapping-object.

[Ope] OpenJDK. *Building OpenJDK*. https://hg.openjdk.java.net/jdk-updates/jdk9u/raw-file/tip/common/doc/building.html. Accessed: 2020-06-04.

[Ora] Oracle. *Java^{TM} Programming Language*. https://docs.oracle.com/javase/8/docs/technotes/guides/language/index.html. Accessed: 2020-06-06.

[Red] RedHat. *Red Hat build of OpenJDK*. https://developers.redhat.com//products/openjdk/download. Accessed: 2020-06-06.

TRITA -EECS-EX-2020:797