ECOLE POLYTECHNIQUE FEDERALE DE LAUSANNE

MASTER THESIS

Efficient Deoptimization

Author:
Adrien GHOSN

Supervisor:
Prof. Jan VITEK, Prof. Viktor KUNCAK

A thesis submitted in fulfilment of the requirements for the degree of Master in Computer Science

in the

LARA
Computer Science

January 19, 2016

Declaration of Authorship

I, Adrien GHOSN, declare that this thesis titled, "Efficient Deoptimization" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed:

Date:

"Thanks to my solid academic training, today I can write hundreds of words on virtually any topic without possessing a shred of information, which is how I got a good job in journalism."

Dave Barry

ECOLE POLYTECHNIQUE FEDERALE DE LAUSANNE

Abstract

Faculty Name
Computer Science

Master in Computer Science

Efficient Deoptimization

by Adrien GHOSN

The Thesis Abstract is written here (and usually kept to just this page). The page is kept centered vertically so can expand into the blank space above the title too...

Acknowledgements

The acknowledgements and the people to thank go here, don't forget to include your project advisor…

Contents

List of Figures

List of Tables

List of Abbreviations

List of Symbols

ω	angular frequency	rad

For/Dedicated to/To my…

Chapter 1

Introduction

Put the introduction here.

1.1 Proposed Solution

Description of what I've done.

1.2 Paper Overview

Explain each chapter.

Chapter 2

On-Stack Replacement

2.1 Overview

2.1.1 Definition

On-Stack replacement (OSR) is a set of techniques that consist in dynamically transferring the execution, at run time, between different pieces of code. The action of transferring the execution to another code artefact is called an OSR transition.

On-Stack replacement can be viewed, at a high level, as a mechanism that allows to transform the currently executing code, into another version of itself. This transformation mechanism has been used to allow the bi-directional transition between different levels of code optimizations. We can therefore reduce it to two main purposes: transforming an executing piece of code into a more optimized version of itself, and undoing transformations that were previously performed. While similar, these two types of transformation have very different goals.

In several virtual machines (CITE PAPERS), some of which will be presented in (REFERENCE), On-Stack replacement has been used to improve the performance of long running functions. When the VM identifies a piece of code as being "hot", i.e., it hogs the execution, it suspends its execution, recompiles it to a higher level of optimization, and transfers the execution to the newly generated version of the function. This differs from a simple Just-In-Time (JIT) compiler, since the recompilation takes place during the execution of the function, rather than just before its execution. However, both techniques rely on run time profiling data to uncover new optimization opportunities. In this case, OSR is used to improve performance.

On-Stack replacement allows a compiler to perform speculative transformations. Some optimizations rely on assumptions that are not bound to hold during the entire execution of a program. A simple example is function inlining in an environment where functions can be redefined at any time. A regular and correct compiler would not allow to inline a function that might be modified during the execution. The OSR mechanism, on the other hand, enables to perform such an optimization. Whenever the assumption fails, i.e., the function is redefined, the OSR mechanism will enable to transfer the execution to a corresponding piece of code where the inlining has not been performed. In this case, OSR is used to preserve correctness.

On-Stack replacement is a powerful technique, that can be used to either improve performance, or enable speculative transformations of the code while preserving correctness. In the next subsection, we present the historical origins of On-Stack replacement and detail its most interesting features.

2.1.2 Why is OSR interesting?

This section highlights the benefits that On-Stack replacement enables. We divide them into two separate cases: OSR with regards to optimization, and OSR for deoptimization.

On-Stack replacement increases the power of dynamic compilation. OSR enables to differ compilation further in the future than dynamic compilation techniques such as Just-In time (JIT) compilation. A function can be recompile while it is executing. This enables more aggressive adaptative compilation, i.e., by delaying the point in time when the recompilation is performed, OSR enables to gather more information about the current execution profile. These information can then be used to produce higher quality compiled code, displaying better performances.

For dynamic languages, code specialization is the most efficient technique to improve performances (IS THAT TRUE? FIND AND CITE). Code specialization consists in tuning the code to better fit a particular use of the code, hence yielding better performances. Specialization can be viewed as a mechanism relying on the versioning of some piece of code. One of the main challenges is to identify which version better fits the current execution need. This requires to gather enough profiling information, some of which might not be available until some portion of the code is executed multiple times.

OSR, coupled with an efficient compiler to generate and keep track of specialized functions, enables to uncover new opportunities to fine tune a portion of code. While techniques like JIT compilation can generate specialized code at a function level, i.e., before the execution of a function, OSR enables to make such tuning while a function is running. For example, in the case of a long running loop inside a function, JIT techniques would need to wait until the function is called anew to improve its run time performance by recompiling it. OSR, on the other hand, gives the compiler the means to make such improvements earlier, hence yielding a better overall performance for the executing program.

OSR is a tool that enables the compiler to recompile and optimize at almost any time during the execution of a program. A clever compiler can perform iterative recompilation of the code in order to improve the quality of the generated compiled artefact. OSR enables these iteration steps to be closer to each other and potentially converge to a better solution faster than other dynamic compilation techniques.

On-Stack replacement's most interesting feature is deoptimization. While optimization enables to increase performance, deoptimization's goal is to preserve correctness of the program that executes. OSR allows speculative optimizations which, in turn, weakens the requirements for compiled code correctness. In other words, the compiler can generate aggressively optimized code. Virtually any assumption can be used to generate compiled code and, if the assumption fails, OSR enables to revert back to a safe version during the execution.

2.2 On-Stack Replacement Mechanisms

2.2.1 The OSR Points

The OSR mechanism is enabled at specific instruction boundaries in the user's code. Depending on the OSR implementation, these points can be sparse, restricted to special points in the control flow, or associated with special framework dependent data. This section presents different implementations of such points in the available literature. In the examples of OSR implementations given so far (CITE SECTIONS), they have been called *interrupt points*(CITE), *OSR entries*, and *OSR exits*(CITE). In the reminder of this paper, we will refer to such points as *OSR points*, as defined in Definition 2.2.1.

Definition 2.2.1 *An OSR point is a portion of code that belongs to the OSR instrumentation. It is located at an instruction boundary at which the program can be suspended. OSR points can correspond to several instructions inserted by the OSR framework and enable to switch the execution from one version of the code to another.*

An OSR point has to be located at a point of the code where the state is *safe*, i.e., points where the state is guaranteed to be consistent. For example, as explained before, the SELF debugger considers points at method prologues and at the end of loop bodies. An OSR point must be such that the state of the computation can be extracted and transferred to another version of the code from which we can resume the execution.

An OSR point can be guarded or unguarded. The WebKit OSR framework (CITE) and the Jikes RVM OSR implementation(CITE) distinguish OSR points that are guarded, i.e., there is a boolean condition wrapping the execution of the point, from unguarded ones. A guarded OSR point is such that the framework has enough information locally to test whether or not an OSR transition should be taken. While simple to model at a high-level, a guarded OSR points presents the disadvantage of increasing the length of the OSR instrumentation, hence impacting on the overall performance of the program. As a result, implementations likes WebKit and Jikes RVM OSR provide a second mechanism that we call unguarded OSR points. In WebKit, an unguarded OSR point is a portion of the code that can be overwritten on some external condition, and hence instrumented to trigger an unconditional transition to another version whenever it is re-written. It also enables to lazily define the transition, i.e., the instrumentation that enables to jump out of the current version can be added just before being executed. TALK ABOUT IN JIKES

There is a further distinction between OSR points that are responsible for optimising the code, and the ones that are used to unoptimize. For that, we will use the same terminology as in WebKit and MCJit(CITE).

Definition 2.2.2 *An OSR entry is an OSR point that enables to replace the current version of the code with one in which we expect to have better performances. They are used in the optimization process.*

Definition 2.2.3 *An OSR exit is an OSR point that enables to exit the current version of the code when it is invalidated. OSR exits are responsible for preserving the correctness of the program. They are used in the deoptimization process.*

CONTINUE

Guarded & Unguarded

Entries & Exits

2.2.2 The Transition Mechanism

High-Level Steps

The Ideal transition

Transition as a Function call

Single entry point. Multiple entry points.

2.2.3 Caching vs. Generating on the Fly

2.2.4 One version

2.2.5 Multiple Versions

2.2.6 Discussion

time vs. space, reusability etc.

2.3 The Deoptimization case

2.3.1 Why the Deoptimization case is more interesting?

2.3.2 Where do we exit?

The Interpreter

The base version instrumented

A less optimized version

2.4 Constraints on Mechanism

2.4.1 space vs. time

2.4.2 Limits on supported optimizations

Chapter 3

Theoretical Model

Introduce Section by saying we focus on deoptimization for our model and GOAL.

3.1 The pontential deoptimization targets

3.1.1 The Interpreter

3.1.2 The Base function

3.1.3 The Less Optimized function

3.2 Versioning

3.2.1 A base function & an instrumented base function

3.2.2 Transitions between different versions

3.2.3 The space vs. time trade-off

3.3 Our Model for Deoptimization

Chapter 4

Implementation

Appendix A

Appendix Title Here

Write your Appendix content here.