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

December 24, 2015

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

Chapter 2

Related Work

2.1 On Stack Replacement, General Principle

2.1.1 Definition & Overview

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 optimisations. We can therefore reduce it to two main purposes: transforming an executing piece of code into a more optimised 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 optimisation, 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 optimisation opportunities. In this case, OSR is used to improve performance.

On-Stack replacement allows a compiler to perform speculative transformations. Some optimisations 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 optimisation. 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 The origins: SELF debugging

The SELF programming language is a pure object-oriented programming language. SELF relies on a pure message-based model of computation that, while enabling high expressiveness and rapid prototyping, impedes the languages performances(CITE from self paper). Therefore, the language's implementation depends on a set of aggressive optimisations to achieve good performances(CITE). SELF provides an interactive environment, based on interpreter semantics at compiled-code speed performances.

Providing source level code interactive debugging is hard in the presence of optimisations. Single stepping or obtaining values for certain source level variables might not be possible. For a language such as SELF, that heavily relies on aggressive optimisations, implementing a source code level debugger requires new techniques.

In (CITE Holzle), the authors came up with a new mechanism that enables to dynamically de-optimise code at specific interrupt points in order to provide source code level debugging while preserving expected behaviour (CITE from holzle).

In (CITE), Hölzle et al. present the main challenges encountered to provide debugging behaviours, due to the optimisations performed by the SELF compiler. Displaying the stack according to a source-level view is impeded by optimisations such as inlining, register allocation and copy propagation. For example, when a function is inlined at a call spot, only a single activation frame is visible, while the source level code expects to see two of them. Figure (FIG), taken from (CITE), provides another example of activations discordances between physical and source-level stacks. In this figure, the source-level stack contains activations that were inlined by the compiler. For example, the activation B is inlined into A', hence disappearing from the physical stack.

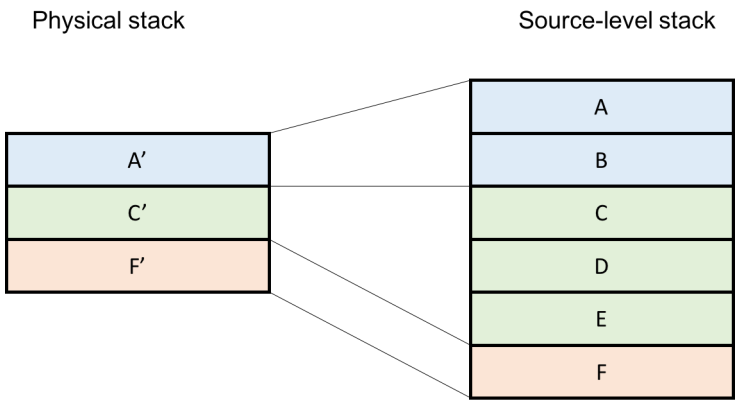

FIGURE 2.1: Displaying the stack CITE

Single-stepping is another important feature for a debugger. It requires to identify and execute the next machine instruction that corresponds to the source operation. Holzle(cite) highlights the impact of code motion and instruction scheduling on the machine instruction layout. Such optimisations re-order, merge, intersperse and

sometimes delete source-level operations, therefore preventing a straight forward implementation of single-stepping for the debugger.

Compiler optimisations prevent dynamic changes from being performed in the debugger. Holzle(CITE) identifies two separate issues: changing variable values, and modifying procedures (i.e., functions). To illustrate the first case, Holzle CITE relies on an example where a variable is assigned the sum of two other variables. The compiler identifies the two variables as being constants and replaces the addition by a direct constant assignment. A debugger that allows to change variable values at run time would then yield a non correct behaviour if the user modifies one of the two variables. This problem does not arise in the case of unoptimised code since the addition is still present. For procedures, Holzle CITE describes an example where a function has been inlined by the compiler, but redefined by the user in the debugger.

Holzle(CITE) distinguishes two possible states for compiled code: *optimized*, which can be suspended at widely-spaced interrupt points, from which we can reconstruct source-level state, and *unoptimized*, that can be suspended at any source-level operation and is not subjected to any of the above debugging restrictions.

In order to deoptimize code on demand, SELF debugger needs to recover the unoptimized state that corresponds to the current optimized one. To do so, it relies on a special data structure, called a *scope descriptor*. The scope descriptors are generated during compilation for each source-level scope. This data structure holds the scope place in the virtual call tree of the physical stack frame and records locations and values of its argument and local variables. It further holds locations or value of its subexpressions. Along with the scope descriptor, the compiler generates a mapping between virtual (i.e, scope descriptor and source position within the scope) and physical program counters (PC). REF is taken from CITE and displays a method suspended at two different points.

The de-optimisation process follows 5 steps described in CITE and summed up here:

1. Save the physical stack frame and remove it from the run time stack.
2. Determine the virtual activations in the physical one, the local variables and the virtual PC.
3. Generate completely unoptimised compiled methods an physical activations for each virtual one.
4. Find the unique new physical PC for each virtual activation and initialise (e.g., return addresses and frame pointers) the physical activations created in the previous step.
5. Propagate the values for all elements from the optimised to the unoptimised activations.

2.1.3 Why is OSR interesting?

2.2 On Stack Replacement & Virtual Machines

2.2.1 In Java

2.2.2 LLVM

2.3 A Description of Existing Implementations

2.3.1 The OSR points

2.3.2 The Transition Mechanism

2.3.3 Constraints and Limitations

2.3.4 Generating on the Fly VS Caching

2.3.5 Discussion

Chapter 3

Theoretical Model

3.1 The OSR points

3.2 The Transition Mechanism

3.3 Constaints

Chapter 4

Implementation

Appendix A

Appendix Title Here

Write your Appendix content here.