

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 6, 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

| | |
|---|------------|
| Declaration of Authorship | iii |
| Abstract | vii |
| Acknowledgements | ix |
| 1 Introduction | 1 |
| 1.1 Proposed Solution | 1 |
| 1.2 Paper Overview | 1 |
| 2 Related Work | 3 |
| 2.1 On Stack Replacement, General Principle | 3 |
| 2.1.1 Definition & Overview | 3 |
| 2.1.2 The origins: SELF debugging | 4 |
| 2.1.3 Why is OSR interesting? | 7 |
| 2.2 On Stack Replacement & Virtual Machines | 8 |
| 2.2.1 In Java | 8 |
| 2.2.2 LLVM | 8 |
| LLVM, formerly called Low Level Virtual Machine | 8 |
| Why On-Stack replacement in LLVM is interesting | 9 |
| Examples of OSR implementation in LLVM | 9 |
| 2.3 A High-Level Description of Existing Solutions | 15 |
| 2.3.1 The OSR points | 15 |
| 2.3.2 The Transition Mechanism | 16 |
| 2.3.3 Constraints and Limitations | 16 |
| 2.3.4 Generating on the Fly VS Caching | 16 |
| 2.3.5 Discussion | 16 |
| 3 Theoretical Model | 17 |
| 3.1 The pontential deoptimization targets | 17 |
| 3.1.1 The Interpreter | 17 |
| 3.1.2 The Base function | 17 |
| 3.1.3 The Less Optimized function | 17 |
| 3.2 Versioning | 17 |
| 3.2.1 A base function & an instrumented base function | 17 |
| 3.2.2 Transitions between different versions | 17 |
| 3.2.3 The space vs. time trade-off | 17 |
| 3.3 Our Model for Deoptimization | 17 |
| 4 Implementation | 19 |
| A Appendix Title Here | 21 |

List of Figures

| | | |
|------|--|----|
| 2.1 | physical vs. source-level stacks | 4 |
| 2.2 | Recovering the source-level state | 6 |
| 2.3 | SSA example | 9 |
| 2.4 | The WebKit FTL | 10 |
| 2.5 | OSR classification | 12 |
| 2.6 | Retrofitting an existing JIT with OSR support | 12 |
| 2.7 | A CFG of a loop with no OSR point | 13 |
| 2.8 | The CFG of the loop in Figure 2.7, after inserting an OSR point | 13 |
| 2.9 | The transformed CFG of the loop in Figure 2.8 after the OSR Pass | 14 |
| 2.10 | A CFG of a running function before inserting the blocks for state recovery | 14 |
| 2.11 | The CFG of the loop represented in Figure 2.10 after inserting the state recovery blocks, from CITE. | 15 |

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

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 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 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 optimizations 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 optimizations. 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 optimizations, 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-optimize 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 optimizations performed by the SELF compiler. Displaying the stack according to a source-level view is impeded by optimizations 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.

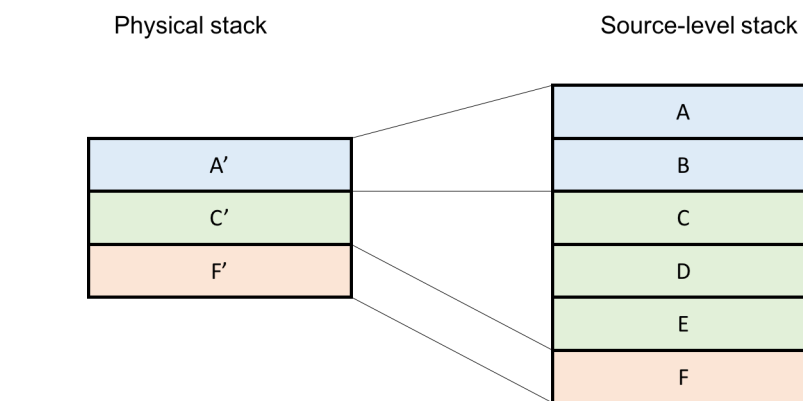


FIGURE 2.1: Displaying the stack CITE.

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 optimizations re-order, merge, intersperse and

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

Compiler optimizations 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 unoptimized 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). Figure 2.2 is taken from CITE and displays a method suspended at two different points. At time t1, the stack trace from the debugger displays frame B, hiding the fact that B was inlined inside of A. At time t2, D is called by C which is called by A, hence, the debugger displays 3 virtual stack frames instead of only one physical frame.

The de-optimization 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 unoptimized 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 optimized to the unoptimized activations.

Holzle(CITE) also describes *lazy deoptimization*, a technique to deoptimize a stack frame that is not at the current top of the execution stack. Lazy deoptimization defers

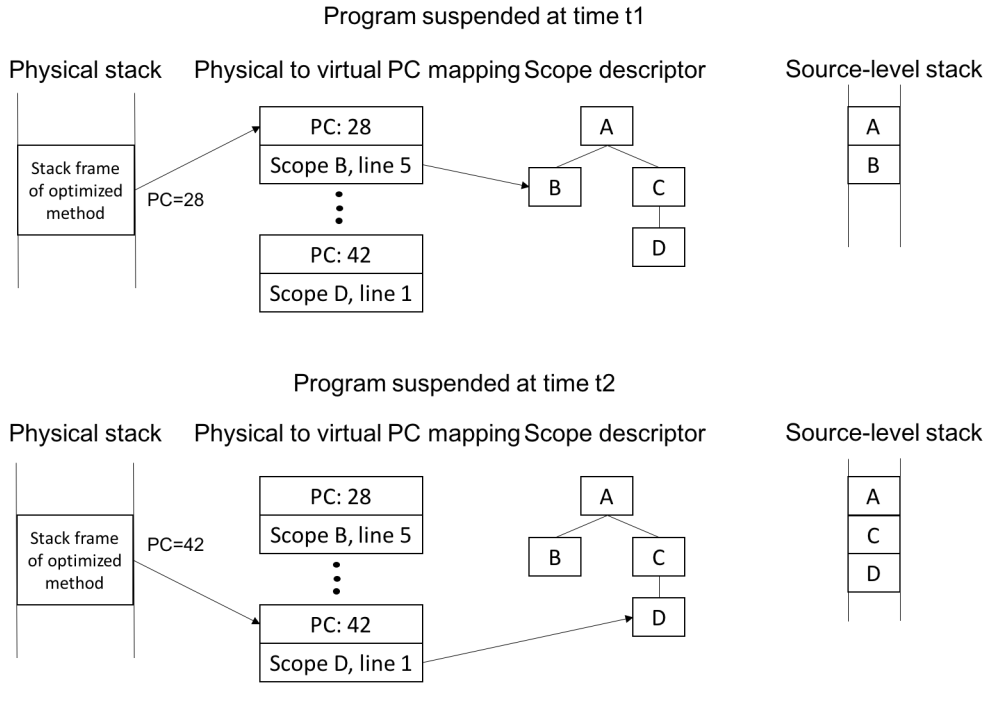


FIGURE 2.2: Recovering the source-level state (from CITE).

the deoptimization transformation until control is about to return into the frame, hence enabling deoptimization for any frame located on the stack.

Deoptimization at any instruction boundary is hard. It requires to be able to recover the state at every single point of the program. Holzle (CITE) relies on a weaker easier restrictions by enabling deoptimization only at certain points called *interrupt points*. At an interrupt point, the program state is guaranteed to be consistent. The paper (CITE) defines two kinds of interrupt points: method prologues, and backward branches (i.e., end of loop bodies). Holzle(CITE) therefore estimates the length of the longest code sequence that cannot be interrupted to be a few dozen of instruction, i.e., the average length of a code sequence that does not contain neither a call nor a loop end. Interrupt points are also inserted at all possible run time errors to allow better debugging of synchronous events such as arithmetic overflow. The generated debugging information are needed only at interrupt points, which reduces the space used to support basic debugger operations (as opposed to allowing interrupts at any instruction boundary).

Providing a debugger for SELF limits the set of optimizations that the compiler can support, and decreases the performances of the program when the execution is resumed. Tail recursion elimination saves stack space by replacing a function call with a goto instruction, while fixing the content of registers. SELF debugger is unable to reconstruct the stack frames eliminated by this optimization and hence, it is not implemented in the SELF compiler. More generally, tail call elimination is one important limitation for the SELF debugger.

The debugger slows down the execution when the user decides to resume. The execution should proceed at full speed, but some stack frames might have been unoptimized, hence implying that a few frames might run slowly right after resuming

execution.

2.1.3 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 & Virtual Machines

Virtual machines are privileged environments in which On-Stack replacement can be used to its full power. As seen in 2.1.3, OSR is as useful as the compiler's profiler is efficient. A virtual machine (VM) has control over the resources allocation, enables to control the code that is generated by the compiler, and maintains important run time data, state information, and other useful informations about the program being executed.

This section presents several examples of VMs that support On-Stack replacement. The section is divided into two parts: we first presents several solutions that provide OSR for the Java programming language, then we briefly introduce LLVM virtual machine, a virtual machine presenting an interesting framework in which we believe On-Stack replacement mechanism should fit.

2.2.1 In Java

2.2.2 LLVM

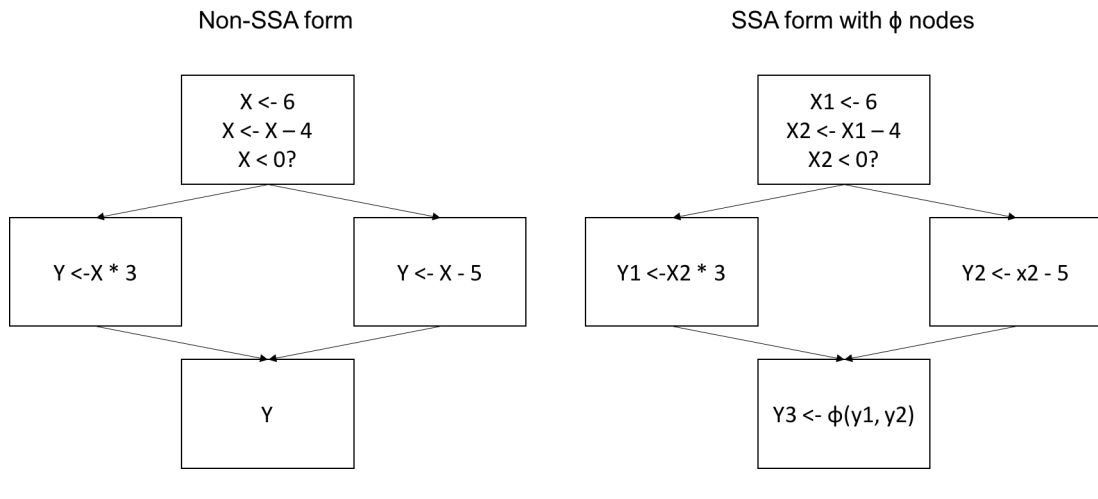
LLVM, formerly called Low Level Virtual Machine

LLVM is a compiler infrastructure providing a set of reusable libraries. LLVM provides the middle layers of a compiler system and a large set of optimizations for compile-time, link-time, run-time, and idle-time for arbitrary programming languages. These optimizations are performed on an intermediate representation (IR) of the code and yield an optimized IR. The LLVM framework also provides tools to convert and link code into machine dependent assembly code for a specific target platform. LLVM supports several instruction sets including ARM, MIPS, AMD TeraScale, and x86/x86-64(CITE?).

The LLVM intermediary representation is a language-independent set of instructions that also provides a type system. The LLVM IR is in static single assignment form (SSA), which requires every variable every variable to be defined before it is used and assigned exactly once. SSA enables or improves several compiler optimizations among which constant propagation, value range propagation, sparse conditional constant propagation, dead code elimination, global value numbering, partial redundancy elimination, strength reduction and register allocation. The SSA requirement for variables to be assigned only once requires a special mechanism, called a ϕ -node, one a value depends on which control flow branch was executed before reaching the current variable definition. Figure 2.3 provides an example where we have to choose between two possible values for a variable after the merging of two control flow branches.

The LLVM IR type system provides basic types (e.g., integers, floats), and five derived types: pointers, arrays, vectors, structures, and functions. Any type construct can then be represented as a combination of these types.

The LLVM framework is a versatile tool that enables to implement many programming languages paradigms. LLVM compilers exist for several mainstream/popular languages such as Java, Python, Objective-C, and Ruby have an LLVM compiler. Other languages, like Haskell, Scala, Swift, Rust, Ada, and Fortran also have an LLVM compiler implementation. LLVM basic types enable to support object-oriented languages,

FIGURE 2.3: Example of ϕ -node in SSA form.

such as Java and Python, dynamically typed languages like R or statically typed like Scala. LLVM also enables to model functional languages such as Haskell, as well as imperative ones. Furthermore, it supports reflection and, thanks to dynamic linking, modular languages (e.g., Haskell). The tools provided enable static compilation as well as dynamic compilation techniques such as Just-In-Time compilation (JIT).

Why On-Stack replacement in LLVM is interesting

On-Stack replacement high-level mechanism is language-independent. Therefore, implementing OSR as a clean modular addition to LLVM would enable developers to leverage this feature in many programming languages, without requiring them to write a new compiler from scratch. Furthermore, as explained in 2.1.3, OSR is a useful tool for dynamic and adaptative optimizations. LLVM already provides implementations for many compiler optimizations (CITE) and tools to allow dynamic recompilation of code. Developers can therefore focus on language specific challenges, such as efficient profilers and new speculative systems, rather than on the optimizations and OSR implementations.

Implementing OSR for LLVM not only serves several languages, but also allows to provide a solution for several target platforms. As explained previously, LLVM supports several instructions sets corresponding to different architectures. By implementing OSR in LLVM, we get portability among these platforms for free.

Examples of OSR implementation in LLVM

OSR has been implemented in LLVM in several projects, such as the WebKit web browser engine and the MCJit project. In WebKit, OSR and LLVM are part of the fourth-tier architecture compiler for JavaScript. The Fourth-Tier LLVM (FTL) is an LLVM-based Just-In-Time compiler. The WebKit run-time compilation flow is described by Figure 2.4.

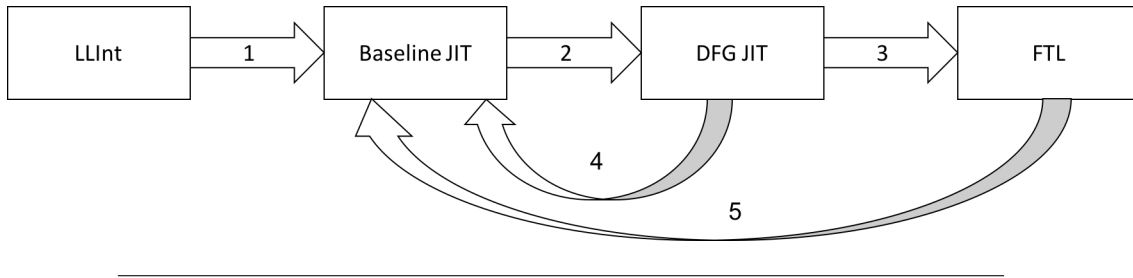


FIGURE 2.4: The WebKit Four-tier optimization flow.

Forward arrows represent *OSR entries*, i.e., a transformation that yields a more optimized version of the code at run-time. Backward arrows correspond to *OSR exits*, i.e., a transformation that yields a less optimized version of the code at run-time. The low level interpreter (LLInt) is used for low latency start up. The baseline JIT generates WebKit bytecode with no optimization enabled. The transition from the first tier to the second one happens when a statement is executed more than a hundred times or a function is called more than six times. The data flow graph (DFG) JIT is triggered when a statement executes more than a thousand times or a function is called more than sixty-six times. The FTL tier relies on LLVM machine code optimizations to generate a fast version of portions of the code. In order to hide the costs of the translation to LLVM IR and its compilation time, the FTL is triggered only for long running portions of the code that are currently executing. There are two kinds of transitions in WebKit: the ones contained entirely inside the WebKit framework (i.e., transitions 1, 2 & 4 in Figure 2.4), and the ones that involve LLVM (i.e., 3 & 5 in Figure 2.4).

Transitions to and from LLVM are hard. There is no control over the stack layout or the optimized code produced by LLVM. In the case of transition 3, a different LLVM version is generated for each entry point that the framework desires to have inside this function. In WebKit, such entry points are located at loop headers. This choice makes sense with regard to the condition to enter the FTL, i.e., transition 3 is taken for long running portions of code that could be improved thanks to LLVM low level optimizations. WebKit has to generate a different version for each entry points for two main reasons: LLVM allows only single entry points to functions (going around this limitation would require to modify LLVM IR and implementation), and instrumenting a function with several entry points would impact on the quality and performance of the generated native code by extending the code's length and restricting code motion.

Performing transition 3 requires to get the current state of execution and identify the entry point corresponding to the current instruction being executed. The DFG dumps its state into a scratch buffer. An LLVM function with the correct entry point is then generated, and instrumented such that its first block loads the content of the scratch buffer and correctly reconstructs the state. The mapping between the DFG IR nodes and the LLVM IR values is straight forward since both IR's are in SSA. A special data structure, called a Stackmap, enables to keep the mapping between LLVM values and registers/spill-slots.

Transition 5 is harder as it requires to extract the execution state from LLVM. WebKit has two different mechanisms to enable OSR exits: the exit thunk and the invalidation points. In the first case, WebKit introduces exit branches at OSR exit points. The

branch is guarded by an OSR exit condition and is a no-return tail call to a special function that takes all the live non-constant and not accounted for bytecode values. The second mechanism enables to remove the guard. Since we assume that the portion of code that is instrumented is executed a lot of times, the cost of testing the condition can have a great impact on the overall execution time. This mechanism relies on a special LLVM intrinsics, namely patchpoints and stackmap shadow bytes. A patchpoint enables to reserve some extra space in the code, filled with nop sleds. When the WebKit framework detects that an exit should be taken, it overwrites the nop sleds with the correct function call to perform the OSR exit. This breaks the optimized version of the code which cannot be re-used later on and must be collected. The stackmap shadow bytes improves on this technique by allowing to directly overwrite the code, without having any nop sled generated before hand.

WebKit is a project that heavily, and successfully relies on OSR to improve performances. The web browser engine is used in Apple Web browser Safari and enables a net improvement of performances why proving to be reliable(CITE?). Although successful, it does not provide a general and reusable framework for OSR in LLVM that other projects could reuse.

The MCJIT OSR support(CITE) is an attempt at providing an OSR library compatible with the standard LLVM implementation. Lameed & Hendren claim to have come up with a clean modular, and re-usable technique completely defined at the LLVM IR level and compatible with the standard LLVM distribution. Their implementation answers to five challenges, listed in the paper CITE and that we reproduce here:

1. Identifying correct interrupt points and using the current LLVM IR to represent them.
2. Using the LLVM tools to correctly transform the IR while preserving a correct control flow graph.
3. Making a new version of a function available at the same address as the old one.
4. Providing a clean API for the OSR library, that is compatible with LLVM's inlining capabilities.
5. Integrating OSR without modifying the LLVM installation.

The paper (CITE) claims to support optimization and re-optimization, as well as de-optimization by going back to the previous version of the function. Figure 2.5 shows that this feature only allows single-steps to be taken, i.e., the OSR library implemented in CITE does not seem to allow to skip intermediary versions.

The MCJit library that provides OSR functionalities fit into the regular JIT infrastructure provided by LLVM as described in Figure 2.6 taken from CITE. The left figure is a normal JIT in LLVM. The LLVM CodeGen is the front-end of the compiler infrastructure that generates the LLVM IR. The LLVM Optimizer contains a collection of transformations and optimizations that run on LLVM IR. The Target CodeGen outputs the machine code corresponding to the LLVM IR input. The MCJit OSR API instruments the LLVM CodeGen to insert OSR points where the OSR transition can be triggered. A point is a call to the `genOSRSignal` function, which takes as arguments a

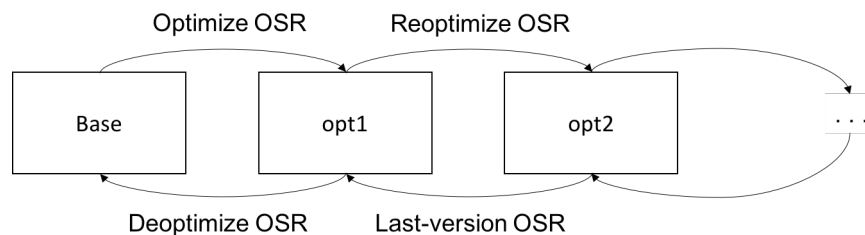


FIGURE 2.5: OSR classification from CITE

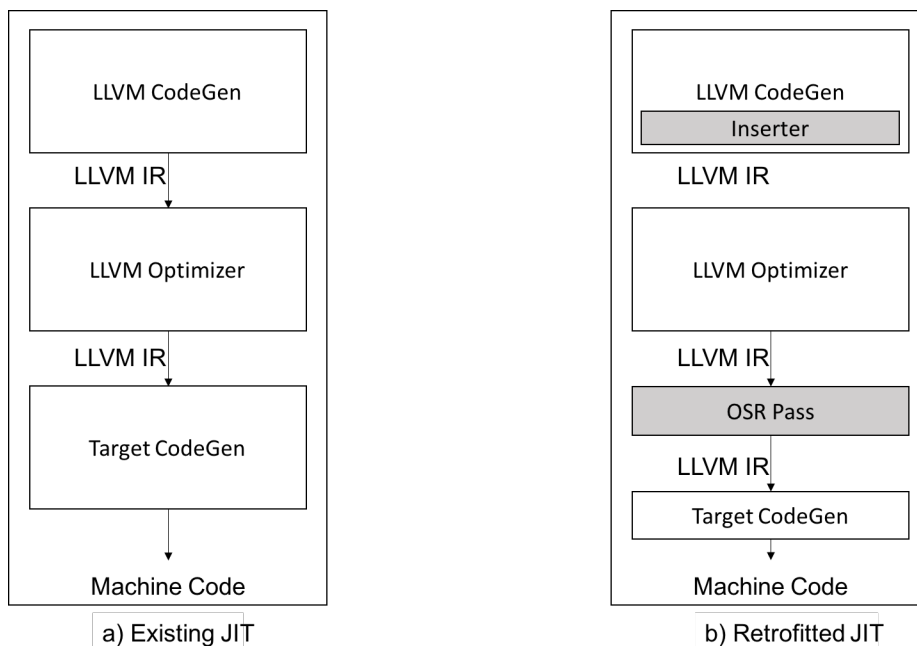


FIGURE 2.6: Retrofitting an existing JIT with OSR support from CITE

pointer to a code transformer responsible for generating the new version of the function. The code transformer takes as arguments a pointer to the function that needs to be transformed, and a special OSR label to identify which OSR points triggered the call, if the function contains several ones. The OSR pass is responsible for instrumenting OSR points with the correct OSR machinery. The instrumentation saves the live values and creates a descriptor that contains four elements:

1. A pointer to the current version of the function.
2. A pointer to the control version of the function, i.e., a copy of the old version of the function.
3. A variable mapping between the original version and the control version.
4. The set of live variables at the OSR points.

Figures 2.7, 2.8, and 2.9 give an example of OSR instrumentation at a loop header. LH1 is the loop header. LB is the body of the loop and LE the loop exit. Figure 2.7 control flow graph (CFG) is the original CFG. Figure 2.8 is the resulting CFG after the

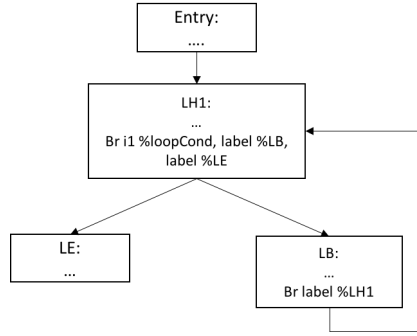


FIGURE 2.7: A CFG of a loop with no OSR point from CITE.

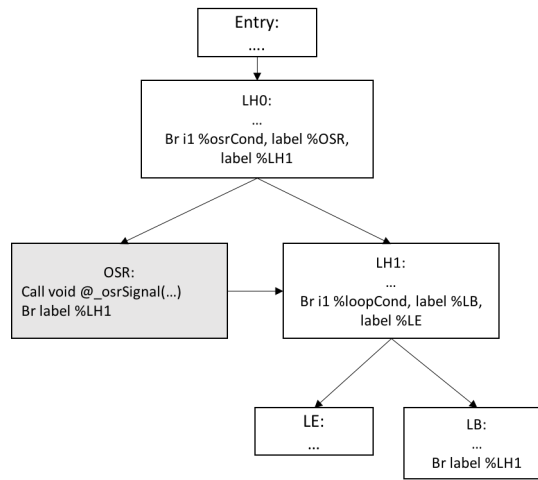


FIGURE 2.8: The CFG of the loop in Figure 2.7, after inserting an OSR point from CITE.

Insertion is executed. Figure 2.9's CFG corresponds to the result of the OSR pass. The *recompile* call in the OSR block recompiles f using the correct code transformer. Then f calls itself, executing the new version of the function. This works since the new version lives at the same address as the previous one and is instrumented to jump to the correct instruction, i.e., the one corresponding to the current point at which OSR was triggered. Figure 2.10 represents the CFG of f before the OSR instrumentation. Figure 2.11 shows the instrumentation of f that enables to jump to the correct instruction in the middle of the function. A prolog entry block is inserted at the function header. This block checks the *OSR flag* to know if an OSR transition is being performed. If that is the case, it branches to the prolog block that restores the state before resuming the execution at the correct instruction.

The OSR implementation proposed in CITE presents interesting features. The implementation is done entirely at the LLVM IR, hence making it language-independent. Furthermore, since the transformation function is provided by the user, any kind of language specific transformation can be used during the OSR. As a result, MCJit OSR support is an interesting modular OSR library.

On the other hand, this library does not allow to have several versions of the same function, live at the same time. This restriction can impede the overall performance

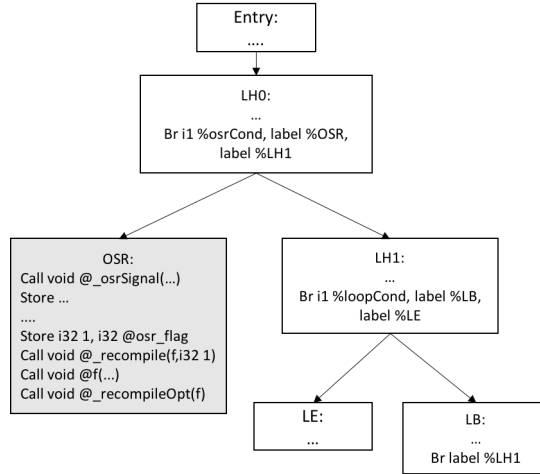


FIGURE 2.9: The transformed CFG of the loop in Figure 2.8 after the OSR Pass from CITE.

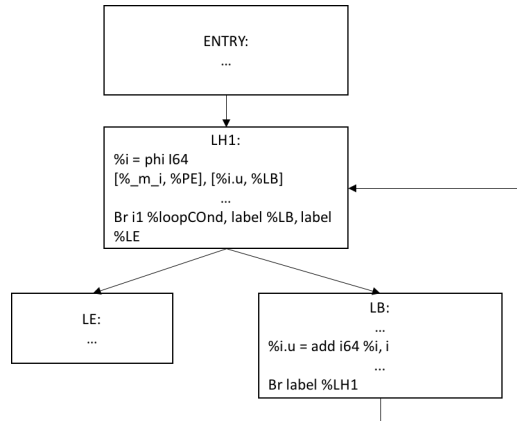


FIGURE 2.10: A CFG of a running function before inserting the blocks for state recovery, from CITE.

of the program by extending the scope in which an assumption on which we base the transformation must hold. For example, a portion of the code, call it *A*, can trigger an OSR, while portion *B* is such that the assumption on which the optimization is based does not hold. The choice that the user has is to either optimize for *A*, and deoptimize for *B*, or to prevent *A* from optimizing by enlarging the scope in which the assumption is supposed to hold. None of these solutions is good if *A* and *B* are executed many times, one after the other. We will either lose a lot of execution time performing OSR, or keep executing *A* without optimization.

In the paper, the deoptimization process is not described. It is implied that it relies on the same set of tools provided by the framework as the optimization process, but no example is provided. As explained in 2.1.3, deoptimization is the most interesting feature of OSR, as it is required to preserve correctness. Being able to identify where a function should exit is a hard task. The MCJit library does not provide tools to ease this process and leaves the responsibility to the developer to instrument his functions correctly in order to exit to the correct landing pad.

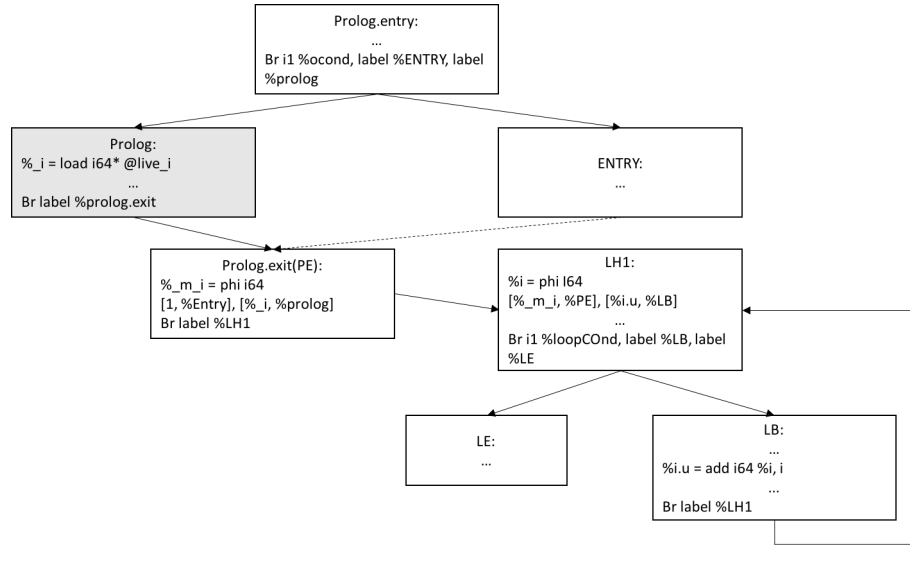


FIGURE 2.11: The CFG of the loop represented in Figure 2.10 after inserting the state recovery blocks, form CITE.

To the best of our knowledge, MCJit OSR library is not currently used in production or any important project. As mentioned earlier, WebKit is efficiently integrated in Apple Safari’s web browser, which provides useful feedback on its performances. The lack of usage of MCJit OSR library prevents us from collecting performance results and assess its efficiency. Furthermore, the experimental evaluation in CITE relies on an example that seems artificial for our use of OSR. The case study presents a dynamic inliner that decides to inline a function call if the function is less than 20 basic blocks long, or if it is less than 50 basic blocks long and has an interpreter environment associated with its body. This requirement for inlining is one that can be checked statically and, hence, seems a little artificial.

2.3 A High-Level Description of Existing Solutions

2.3.1 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 2.3.1.

Definition 2.3.1 *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 2.3.2 *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 2.3.3 *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

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

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

3.1 The potential 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.