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 21, 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 On-Stack Replacement **3**
2.1 Overview . . . 3
2.1.1 Definition . . . 3
2.1.2 Why is OSR interesting? . . . 4
2.2 On-Stack Replacement Mechanisms . . . 5
2.2.1 The OSR Points . . . 5
Guarded & Unguarded . . . 5
OSR Entries & OSR Exits . . . 5
2.2.2 The Transition Mechanism . . . 6
High-Level requirements for OSR transitions: transformation challenges . . . 6
The ideal transition: technical challenge . . . 7
Transition as a Function call . . . 8
2.2.3 Caching vs. Generating on the Fly . . . 10
One version . . . 10
Multiple Versions . . . 10
2.3 The Deoptimization case . . . 10
2.3.1 Why the Deoptimization case is more interesting? . . . 10
2.3.2 Where do we exit? . . . 11
The Interpreter . . . 12
The base version instrumented . . . 12
A less optimized version . . . 12
2.4 Constraints on the OSR Mechanism . . . 13
2.4.1 space vs. time . . . 13
2.4.2 Limits on supported optimizations . . . 13

3 Related Work **15**
3.1 The origins: SELF Debugging . . . 15
3.2 OSR & VMs . . . 18
3.2.1 HotSpot . . . 18
3.2.2 Jikes RVM . . . 19
3.2.3 WebKit VM . . . 20
3.3 OSR & LLVM . . . 21

3.3.1 What is LLVM? 21
3.3.2 Why do we want OSR in LLVM? 22
3.3.3 OSR as an LLVM library: the MCJIT OSR implementation 23
3.4 A classification summary 27

4 OSR Kit **29**
4.1 Overview 29
4.1.1 An LLVM Library 29
4.1.2 LLVM IR 29
4.2 Resolved & Unresolved OSR 29
4.2.1 The Unresolved OSR 29
Similarities with MCJIT 29
4.2.2 The Resolved OSR 29
4.3 The API 29
4.3.1 OSR Points 29
4.3.2 OSR Conditions 29
4.4 Using OSR Kit to deoptimize 29
4.4.1 The unease of deoptimizing 29
4.4.2 SOMETHING 29
4.4.3 SOMETHING ELSE 29
4.5 Extending OSR Kit 29
4.5.1 Handling versions 29
4.5.2 Unguarded OSR points 29
4.5.3 Fusing versions? 29

5 Case Study **31**
5.1 The RJIT project 31
5.2 OSR in RJIT: requirements 31
5.3 Implementation 31
5.3.1 The Function Call in RJIT 31
5.3.2 Challenges 31
5.4 OSR Conditions 31

A Appendix Title Here **33**

Bibliography **35**

List of Figures

2.1 OSR exit options 11

3.1 physical vs. source-level stacks 16
3.2 Recovering the source-level state 17
3.3 The WebKit FTL 20
3.4 SSA example 22
3.5 OSR classification 23
3.6 Retrofitting an existing JIT with OSR support 24
3.7 A CFG of a loop with no OSR point 25
3.8 The CFG of the loop in Figure 3.7, after inserting an OSR point 25
3.9 The transformed CFG of the loop in Figure 3.8 after the OSR Pass 26
3.10 A CFG of a running function before inserting the blocks for state recovery 26
3.11 The CFG of the loop represented in Figure 3.10 after inserting the state recovery blocks, form CITE. 27

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 function 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 function 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, the VM suspends the function's 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 and transfer of execution both take 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.

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 run time. A regular, and correct compiler would not allow to inline a function that might be modified during the execution. The OSR mechanism, however, 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.

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. 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, 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 the program's correctness in the presence of aggressive speculative optimizations. 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 program control flow, or associated with special framework dependent data. In the literature, such boundaries were given various names, e.g., *interrupt points*(CITE), *OSR entries*, and *OSR exits*(CITE). This section presents the terminology for such points that will be used in the reminder of this report.

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 in Section 3.1, 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.

Guarded & Unguarded

CONSISTENCY: PUT DEFINITIONS.

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.

OSR Entries & OSR Exits

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

2.2.2 The Transition Mechanism

The OSR mechanism can be narrowed down to its core functionality: allowing a transition at run time from the middle of a running function to a corresponding instruction in the middle of another version of this function. This transition mechanism is therefore the cornerstone of any OSR implementation. This section strives to unify all the existing implementations under a single, general, high-level list of requirements to perform an OSR transition.

In the reminder of this section, we use the terminology introduced by definitions 2.2.4 and 2.2.5. These definitions hold, regardless of the type of OSR transition, i.e., OSR exit or OSR entry, being performed.

Definition 2.2.4 *A function from which we perform an OSR transition is called a **from function** or **source version**.*

Definition 2.2.5 *A function in which we resume the execution after an OSR transition is called a **continuation function** or **destination version**.*

High-Level requirements for OSR transitions: transformation challenges

An OSR transition relies on a mapping between two program points and their respective states. Such a mapping is possible only if all the information available at the OSR point in the from function is sufficient to reconstruct the state expected at the landing site in the continuation function.

Reconstructing the state at the landing site means that all the expected live variables must be assigned the correct values to safely resume the execution in the continuation function. Propagating live values is, itself, a challenge, and requires to keep a mapping between the live values in the from function and the live values in the continuation function. Compiler optimizations might eliminate variables, fuse them, create new ones, and hence prevent a one-to-one mapping between the two sets of live values. As a result, the OSR transition implementation must provide a function f that takes as input the set of live variables and their assigned values S_s from the source version s and generates a new set of live variables mapped to their values S_d for the destination version d.

$$f : S \rightarrow S$$

$$f(S_s) = S_d$$

An OSR transition is transparent to the user, i.e., it should not be observed by the user running the program. The OSR transition can be viewed as a mechanism that deviates the program from its current execution path in order to either improve the performances (the optimization process) or preserve correctness (the deoptimization process). It is part of the compilation/execution framework and, aside form a performance variation, should not impact the execution of the program, e.g., a correct program should not crash because of an OSR transition.

An OSR transition can be either *unidirectional* or *bidirectional*. In the case of a bidirectional mapping between the source and the destination version, the OSR framework must also provide a function f' that is the inverse of f.

$$f' : S \rightarrow S$$

$$f'(S_d) = S_s$$

This corresponds to the deoptimization case and presents a greater challenge than the optimization case. Consider the unoptimized code in the Snippet **??**, and its optimized version in Snippet ??.

LISTING 2.1: Unoptimized case

```
a <- 5
b <- 6
c <- a+b
```

LISTING 2.2: Optimized case

```
c <- 11
```

The function f' must provide a way to re-generate a and b from c. To this end, it must remember the steps that led to the optimized version of the code, and must be able to reverse them. This implies a lot of metadata attached to each version of the optimized code generated by the compiler and might quickly increase the space complexity of the OSR support. EXAMPLE VARMAP FROM JIKES!!!

The ideal transition: technical challenge

At a machine code level, an ideal OSR transition can be summarised into 5 steps:

1. Save the content of the registers.
2. Save the local live values, i.e., the on-stack values in the current stack frame.
3. Modify the current stack frame to introduce the expected local values, add expected stack frames or remove the stack frames that are not expected.
4. put the correct content inside the registers.
5. modify the program counter and resume execution.

In order to perform such a modification of the runtime stack and registers, the OSR framework must have complete control over the machine executing the program. This requirement is easier to satisfy if the execution is performed inside a virtual machine that enables to access and modify the execution stack and the registers. Section 3.3 expands on the relation between OSR implementations and Virtual Machines. Furthermore, as explained in 2.2.2, reconstructing a state from a set of live values is challenging. Performing this operation at such a low level is hard. The compiled version of the code loses the high-level semantics specific to the language, translates some high-level operations into several sequential instructions and performs some low level ressource management (e.g., register allocation) that requires the OSR mechanism to perfectly understand how the compiled code is generated in order to enable the OSR transition at machine code level.

While tempting for its fine-grained control over the transition, the implementation of the OSR mechanism at a low level seems to unveil great challenges. An alternative solution is to encapsulate the entire process at a higher level, closer to the original program semantics.

Transition as a Function call

An OSR transition can be modelled as a function call. As explained previously, an OSR transition needs to propagate values and jump to a continuation point outside of the current execution path. A function call allows to "modify" the program counter. The OSR transition therefore correspond to a function call from the from function, to a continuation function responsible for executing the new version of the code.

In order to propagate the current execution state, there are two main solutions:

1. Dumping the state in a shared memory location(e.g., as in MCJIT OSR (CITE) presented in REF), or
2. Passing all the needed values as function call arguments to the continuation function(e.g., OSR Kit, described in REF).

The first option implies that the from function must dump its state into a buffer before calling the continuation function. The continuation function starts its execution by loading the values it needs from the buffer. This solution has the drawback of increasing both the from function and the continuation function lengths, i.e., the OSR propagation of the state has a greater impact on the performance of the execution than a simple function call cost.

LISTING 2.3: Figures/ffromBuff.c

```
int f_from(a,b) {
//Some code.
//...

  if(OSR_CONDITION)
    goto OSR;
CONT:
//...

OSR:
//Saves the state
//into a shared buffer.
  save_state();
  return f_continuation(a,b);
}

int f_continuation(a,b) {
//Loads the state
//from the shared buffer.
  load_state();
CONT:
//Equivalent for CONT.
//...
}
```

For the second option, most of the work needed to propagate the state is performed during the compilation. Values are passed as arguments to the call to the continuation function. If there is not a one-to-one mapping between the from variables and the continuation ones, some compensation code is executed, either before the call or at the beginning of the continuation function, in order to assign the correct values to the continuation local variables. The cost of executing the compensation code is also present in the first option, i.e., both solutions have to perform the same set of transformations in order to generate the expected local values. On the other hand, the second option does not require to access shared memory to store and load propagated values, and is therefore expected to have a smaller cost at execution.

Listing 2.4: Figures/funCall.c

```
int f_from(a,b) {

  //Some code.
  //...
  //[Live values: c,d,e,f]
  if(OSR_Condition)
    goto OSR;
CONT:
//Some code.
//...
OSR:
  f_continuation(a, b, c, d, e, f);
}

int f_continuation(a,b, c, d, e, f) {
CONT:
  //Some code.
}
```

Listing 2.5: Figures/MultipleEntry.c

```
int fbase (a,b) {
PROLOG:
  if (OSR_LABEL == 0001)
    goto OSR1;
  else if (OSR_LABEL == 0002)
    goto OSR2;
  else
    goto ENTRY;
ENTRY:
  //The regular entry code
  //...
CONT1:
  //Some point in the code
  //that corresponds to osr1.
  //...
CONT2:
  //Some point in the code
  //that corresponds to osr2.
  //...
OSR1:
  load_state1();
  goto CONT1;
OSR2:
  load_state2();
  goto CONT2;
}
```

MODIFY TO MATCH PICTURE AND REFORMULATE (IT'S MESSY)

The OSR transition requires to enter the continuation function at a special entry point, that corresponds to the the one in the from function. In order to decrease the space complexity, one might want to enable multiple entry points per function. Consider the case of deoptimization in FIGURE, where function f inlined two of its function calls, one to g and one to z. This function exits to the base version of f before the corresponding call. Allowing multiple entry points in f_{base} enables to have only one continuation function for both exits. This solution, coupled with the shared buffer to propagate the execution state, can be implemented as a simple function f_{base} instrumented as in FIGURE. This instrumented version of f_{base} can serve regular function calls, and also be used as a continuation function for both OSR exits. In order to do so, the f_{base} starts by checking a flag to know if it is performing an OSR or serving a regular function call. According to this flag, it jumps to the correct spot inside the function. This instrumentation has an impact on performance as it requires an extra computation at the beginning of the function. On the other hand, the second technique that we described to propagate the state modifies the function's signature. If the set of values to propagate is different at the two points, we need different functions, and hence, should allow only a single entry point for each continuation function. We can expect this solution to have, in general, a higher space complexity but a better performance during execution (i.e., smaller execution time).

2.2.3 Caching vs. Generating on the Fly

One version

Multiple Versions

2.3 The Deoptimization case

This section...

2.3.1 Why the Deoptimization case is more interesting?

On-stack replacement main interest lies in dynamic deoptimization of code. As explained in 2.1.2, OSR can be used to deoptimize code in order to preserve correctness. A compiler, equipped with an efficient implementation of OSR, is allowed to perform adaptive optimizations, based on some assumptions that might fail at some point, during the execution. Put in another way, OSR enables aggressive speculative optimizations of code by providing a mechanism that preserves correctness. OSR strives to enable common optimizations performed by static programming language compilers in dynamic language compilers. For example, consider the function inlining optimization, which replaces a function call site with the body of the function called. Inlining in static programming languages is a common optimization(CITE OR GIVE EXAMPLES?). It enables to eliminate the overhead of a function call and return, and enlarges the scope considered for further optimizations. While very common in static languages, inlining proves to be more difficult to implement in compilers for dynamic languages, if functions can be redefined at runtime. OSR allows to perform speculative inlining of function calls. The assumption taken by the compiler is that the function call corresponds to a specific function body. A guarded OSR point checks that the inlined function was not redefined before executing the inlined body. If the assumption fails, i.e., the function was redefined, the OSR mechanism allows to exit to a version of the code where the call was not inlined.

The OSR mechanism is only a tool, which efficiency depends on the quality of the assumptions made. This is even more visible in the deoptimization case. The optimization case remains useful for regular optimizations, i.e., an OSR entry can be allowed by some profiling data gathered during the execution, and based on some fact that will hold for the rest of the execution. On the other hand, an agressive assumption-based optimization exposes a trade-off between the performance improvement that it allows on one side, and the cost it has in the case of an assumption failure on the other. As a result, the OSR deoptimization process allows a greater flexibility but should be coupled with an efficient profiler in order to identify cases where and when the program's execution might benefit from aggressive speculative optimizations.

2.3.2 Where do we exit?

The OSR deoptimization case is interesting in the latitude it leaves with regard to where the optimized code should exit. In the OSR entry case, the continuation function is a more optimized version of the code. In the OSR exit case, there is still a choice to make: should the continuation function be a fully unoptimized version of the code? Or should we strive to obtain the most optimized version of the code? This section describes several targets for OSR exits, and tries to present the advantages, disadvantages, and requirements for each option.

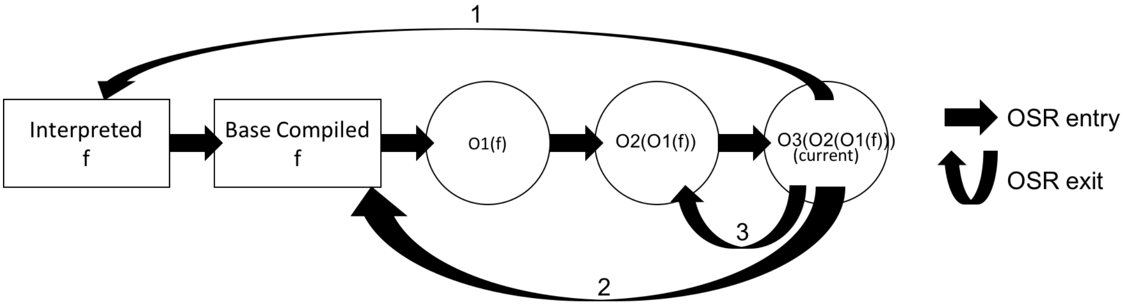

FIGURE 2.1: OSR exit options.

The Interpreter

A first option is to OSR exit back to the interpreter. Some virtual machines, such as Graal EXAMPLES AND REF, CONTINUE SENTENCE.

Going back to the interpreter enables to have complete control over the execution environment. The execution framework can therefore easily generate the correct state to resume the execution. CONTINUE. MAKES SENSE SINCE FAILURE OF ASSUMPTION, RESUME FROM BEGINNING TO REGENERATE PROFILING DATA.

BAD PERFORMANCE.

The base version instrumented

Another option is to OSR exit to a base, compiled version of the function. Going back to a base version of the function corresponds to the OSR exit 2 in Figure 2.1. This is the approach taken by the WebKit framework (see Section 3.2.3). It represents a good intermediary between performance and complexity. If the OSR framework does not have control over the execution environment, it requires careful instrumentation of the compiled code in order to properly propagate the execution state. If the continuation function is generated on the fly, taking this OSR transition implies suffering the cost of compiling a proper continuation function. On the other hand, the compiled version of the continuation function is expected to run faster than its interpreted equivalent.

On a theoretical point of view, going back to the base version avoids cascading OSR exits. Since an exit was taken, it implies that one of the compiler's assumptions failed. As it is hard to keep track of interferences and relations between different optimization steps, a compiler profiler might not have the means to detect how one assumption's failure impacts other assumptions that were taken. It might be the case that undoing one optimization would yield an incorrect version of the code due to interferences between optimizations (in the worst case), or trigger other OSR exits. An OSR exit also gives some feedback to the compiler's profiler that might improve the decisions that it takes for later speculative optimizations. Restarting the optimization process from scratch might therefore yield a more performant version of the code, compared to the version obtained by undoing the optimization that failed.

A less optimized version

CLEAN UP AND REFORMULATE (MESSY)

A third option is to exit to an optimized version of the function that does not rely on the assumption that failed. It corresponds to the OSR exit 3 in Figure 2.1. The function to which we exit is a fully optimized one, supposed to yield better performances than both the interpreted one and the base version. This behaviour can be observed in MCJit OSR implementation CITE (see Section 3.3.3) and OSR Kit CITE(see Chapter 4). This option strives to get the best possible performance, even in the presence of an assumption's failure. As explained previously, there are two options to get the continuation function: either generate on the fly a new fully optimized version that does not rely on the assumption that failed, or "undo" the optimization that failed. Undoing an optimization is hard, especially when it is intertwined with other transformations performed on the code. An alternative solution is to keep the version of the code that was used to produce the optimized one, and use it as the continuation function for the OSR exit. If the optimization that failed was the last transformation performed on the code, we exit to a version that already has undergone all the available transformations and cannot be further optimized. If that is not the case, we exit to a version of the code that is missing several optimizations. For example, consider Figure 2.1. If the assumption used for transformation $O3$ fails, we can easily exit to the version on which we applied the transformation, i.e., $O2(O1(f))$. This corresponds to the OSR exit 3 in Figure 2.1. However, if the assumptions used for $O2$ fails, there is no previous version of f that corresponds to $O3(O1(f))$ and generating one on the fly, from $O3(O2(O1(f)))$, is hard, since it requires to undo $O2$. One possible solution, in that case, is to generate the continuation function on the fly. Starting from f, we apply $O1$ and then $O3$, yielding $O3(O1(f))$. This is a viable solution if and only if it is possible to find a mapping between $O3(O2(O1(f)))$ and $O3(O1(f))$. If such a mapping does not exist, the OSR implementation must ensure that no transformation messes with the OSR instrumentation. In that case, we can always go back to the version on which we applied the transformation that failed, e.g., $O1(f)$ in our example. OSR Kit, presented in Chapter 4, can be used to implement such a solution.

2.4 Constraints on the OSR Mechanism

2.4.1 space vs. time

SUMMARIZE ALL THE TIME VS SPACE CONSTRAINTS. MAYBE A LISTING?

2.4.2 Limits on supported optimizations

TAIL CALL ELIMINATION.

Chapter 3

Related Work

3.1 The origins: SELF Debugging

SELF is a pure object-oriented programming language. It 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). As a result, the language's implementation depends on a set of aggressive optimizations to achieve good performances[2]. 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 "Debugging optimized code with dynamic deoptimization"[2], 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).

Hölzle, Chambers, and Ungar[2] 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 site, only a single activation frame is visible, while the source level code expects to see two of them. Figure (FIG), taken from [2], 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.

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. Hölzle, Chambers, and Ungar[2] highlight 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

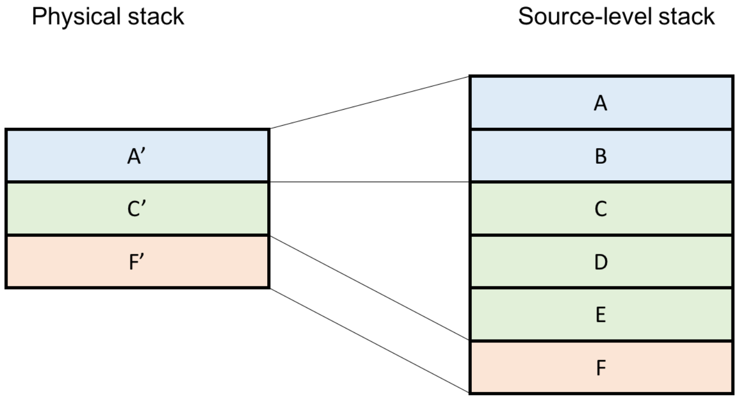

FIGURE 3.1: Displaying the stack, figure from [2].

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 3.2 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.

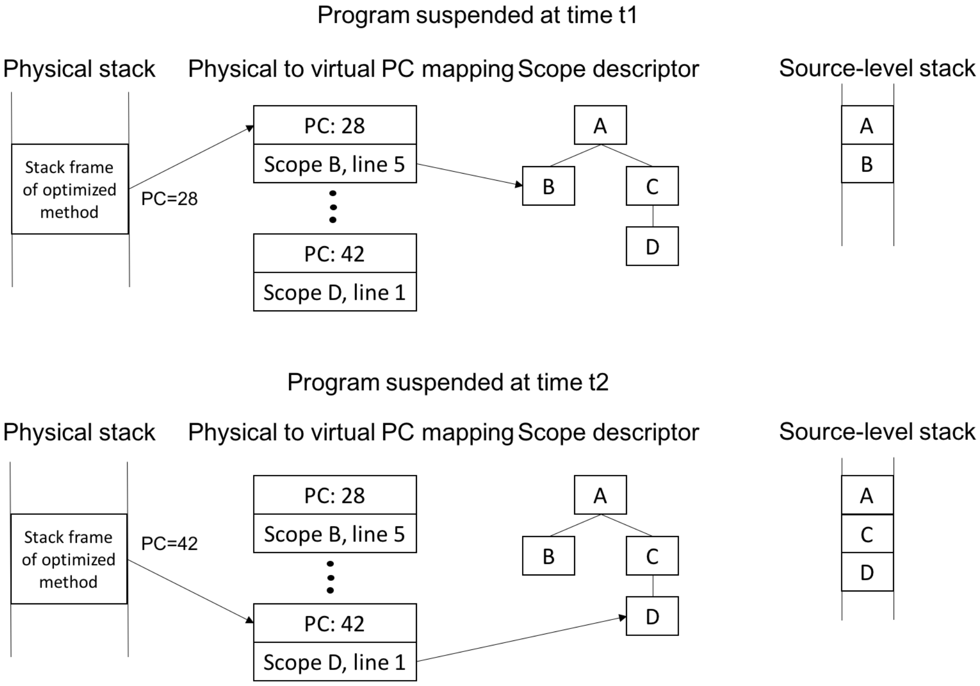
Program suspended at time t1
Physical stack
Physical to virtual PC mapping
Scope descriptor
Source-level stack
Stack frame of optimized method
PC=28
PC: 28
Scope B, line 5
PC: 42
Scope D, line 1
A
B
C
D
A
B
Program suspended at time t2
Physical stack
Physical to virtual PC mapping
Scope descriptor
Source-level stack
Stack frame of optimized method
PC=42
PC: 28
Scope B, line 5
PC: 42
Scope D, line 1
A
B
C
D
A
C
D

FIGURE 3.2: Recovering the source-level state (from CITE).

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 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.

3.2 OSR & VMs

Virtual machines are privileged environments in which On-Stack replacement can be used to its full power. As seen in 2.1.2, 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, maintains important run time data, and state informationabout 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 various virtual machines, then we briefly introduce LLVM, a VM presenting an interesting framework in which we believe On-Stack replacement mechanism should fit.

3.2.1 HotSpot

The Java HotSpot Performance Engine(CITE web) is a Java virtual machine developed and maintained by Oracle. The Java HotSpot VM provides features such as a class loader, a bytecode interpreter, Client and Server virtual machines, several garbage collectors, just-in-time compilation and adaptive optimizations.

The Java HotSpot Engine supports on-stack replacement in order to provide efficient deoptimization[4]. When a class loading invalidates an optimization decision, such as a call inlining, the methods relying on this decision need to be deoptimized. Threads that are executing in a method that needs to be deoptimized are stopped as soon as they reach a safepoint. The JVM state is recorded as an input of safepoints and procedure calls. This implies, as a side effect, that the entire JVM state is marked as "live" at a safepoint, hence extending the live range of some values. The Java HotSpot Engine then extracts the native frame corresponding to this execution trace, and converts it into a byte code interpreter frame. The execution of the method then continues in the interpreter. UNCOMMON TRAPS

3.2.2 Jikes RVM

The Jikes Research Virtual Machine (RVM)(REF) is an open source, self hosted, i.e., it is entirely implemented in Java, virtual machine for Java programs. It provides advanced state-of-the-art features such as dynamic compilation, adaptive optimizations, garbage collection, thread scheduling, and synchronization (CITE).

Jikes RVM is an extensible framework in which different On-Stack replacement techniques have been implemented. Fink and Qian[1] implemented OSR support in JikesRVM. Their implementation relies on JVM scope descriptor, associated to method activation frames. A scope descriptor contains the thread running the activation, the bytecode index that corresponds to the program counter, values of all local and stack locations, and a reference to the activation's stack frame. The OSR transition is divided into three steps:

1. Extract the compiler-independent state from a suspended thread.
2. Generate the new code for the suspended activation.
3. Transfer the execution in the suspended thread to a new compiled code.

Fink and Qian[1] generates the target function by compiling a specialized version of the method for each activation that is replaced, as well as a new version for future invocations. In other words, instead of allowing multiple entry points per function [3, 4], this JikesRVM OSR implementation generates a specialized version of the target function that has only one entry point, corresponding to the instruction from which the OSR transition was triggered. Each such method contains a special *prologue*, responsible for saving values into locals and loading values on the stack. OSR transitions can be taken at special points, called *OSR points*, introduced by the optimizing compiler and that correspond to points where the running activation may be interrupted. The implementation makes a distinction between unconditional and conditional OSR points. An OSR point is implemented as call that takes all live variables as arguments. This constraints some optimizations such as dead code elimination, load elimination, store elimination and code motion by extending the liveness scope of some variables. An OSR point transfers control to an exit block, i.e., it can be viewed as a non-return call.

Soman and Krintz[5] proposed a general-purpose OSR mechanism, for JikesRVM, that presents less restrictions for compiler optimizations than the previous approach, while decoupling the OSR implementation from the optimization process. They extend the previous OSR implementation[1] to enable OSR transition at, and accross points at which the execution can be suspended. In order to do so, they rely on a special data structure called a *variable map*(VARMAP). A VARMAP is associated with each method, and consists in a list of thread-switching points and their live variables. When the compiler performs optimizations, the VARMAP is updated accordingly. Once the compilation completes, the VARMAP is compressed into a compressed map that contains an entry for each OSR point present in the method. EXPAND ON HOW OSR IS PERFORMED. Soman and Krintz[5] also propose an alternative lazy triggering of on-stack replacement. Lazy triggering consists in taking an OSR transition due to events in the environment, i.e., events triggered by the runtime. Whenever the runtime deems an assumption invalide, it invokes a helper function called *OSR helper* that either patches the code of the executing methods to call the OSR process, or it modifies return addresses of the callees of the method to be replaced to point to the OSR helper. When a callee returns, the OSR helper creates a new stack frame with the state extracted from the specialized method's stack and saves all of the specialized method registers into its stack frame. The return address of the OSR helper points to the current instruction in the specialized code, which is then used during OSR to identify the location to resume the execution in the new version of the method. Lazy triggering improves the code efficiency by avoiding the extra cost of guards evaluations.

3.2.3 WebKit VM

WebKit is an open source web browser engine used to improve JavaScript performances. It exhibits a Four-Tier VM architecture. The WebKit run-time compilation flow is described by Figure 3.3.

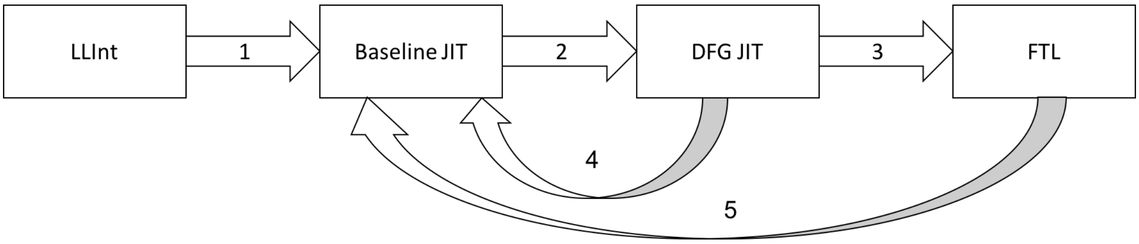

FIGURE 3.3: The WebKit Four-tier optimization flow.

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 3.3), and the ones that involve LLVM (i.e., 3 & 5 in Figure 3.3).

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 enables 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.

3.3 OSR & LLVM

3.3.1 What is LLVM?

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 3.4 provides an example where we have to choose between two possible values for a variable after the merging of two control flow branches.

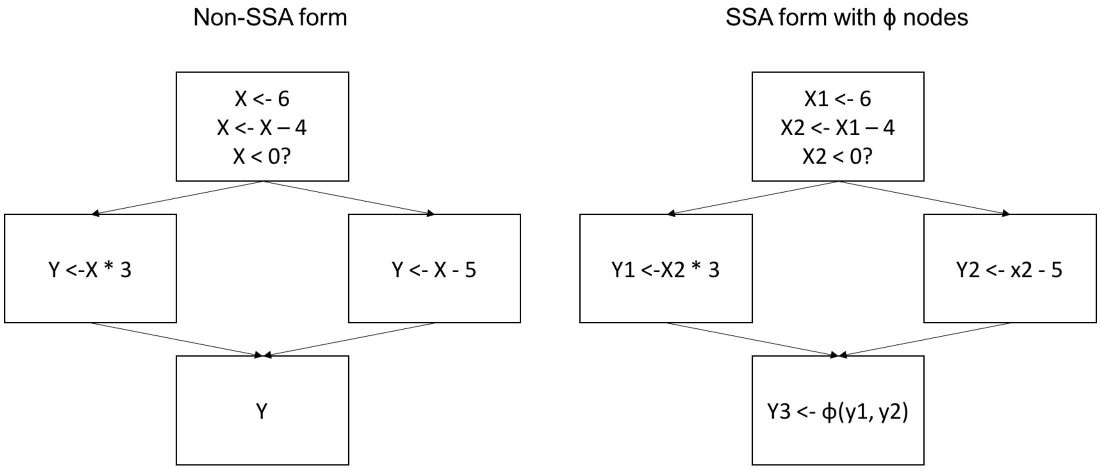

FIGURE 3.4: Example of ϕ-node in SSA form.

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, 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).

3.3.2 Why do we want OSR in LLVM?

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 2.1.2, 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.

3.3.3 OSR as an LLVM library: the MCJIT OSR implementation

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. There 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 deoptimization by going back to the previous version of the function. Figure 3.5 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.

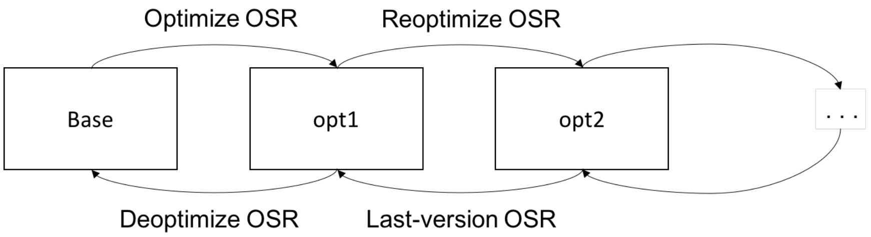

FIGURE 3.5: OSR classification from CITE

The MCJit library that provides OSR functionalities fit into the regular JIT infrastructure provided by LLVM as described in Figure 3.6 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

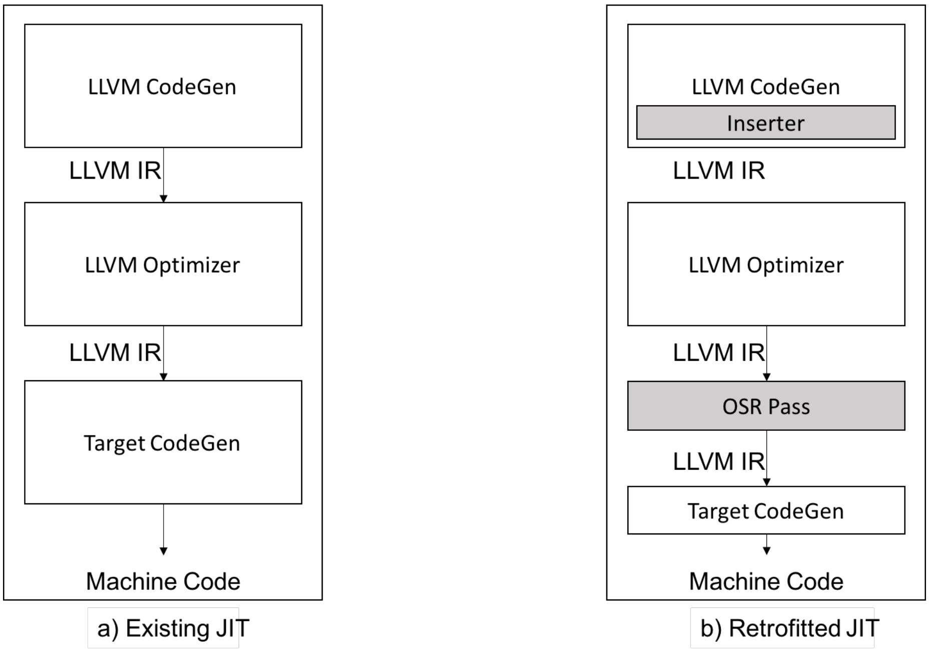

FIGURE 3.6: Retrofitting an existing JIT with OSR support from CITE

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 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 3.7, 3.8, and 3.9 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 3.7 control flow graph (CFG) is the original CFG. Figure 3.8 is the resulting CFG after the Inserter is executed. Figure 3.9'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 3.10 represents the CFG of f before the OSR instrumentation. Figure 3.11 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

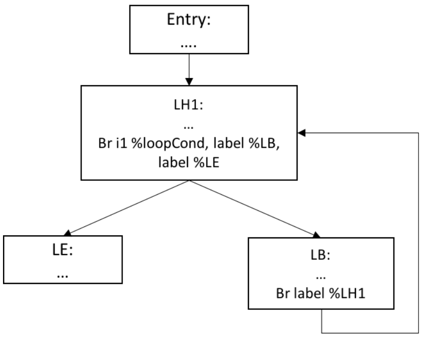
Entry:
....
LH1:
...
Br i1 %loopCond, label %LB,
label %LE
LE:
...
LB:
...
Br label %LH1

FIGURE 3.7: A CFG of a loop with no OSR point from CITE.

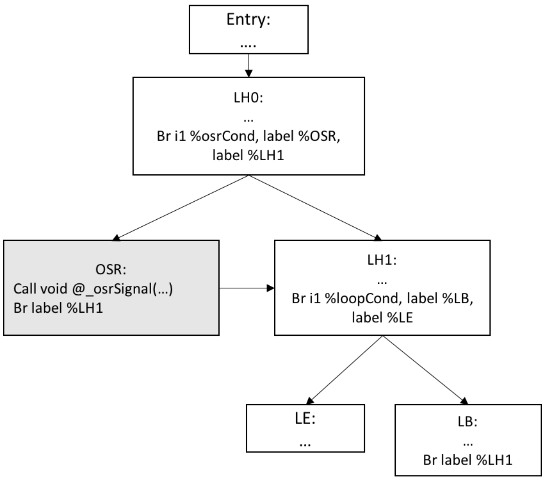
Entry:
....
LH0:
...
Br i1 %osrCond, label %OSR,
label %LH1
OSR:
Call void @_osrSignal(...)
Br label %LH1
LH1:
...
Br i1 %loopCond, label %LB,
label %LE
LE:
...
LB:
...
Br label %LH1

FIGURE 3.8: The CFG of the loop in Figure 3.7, after inserting an OSR point from CITE.

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 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.

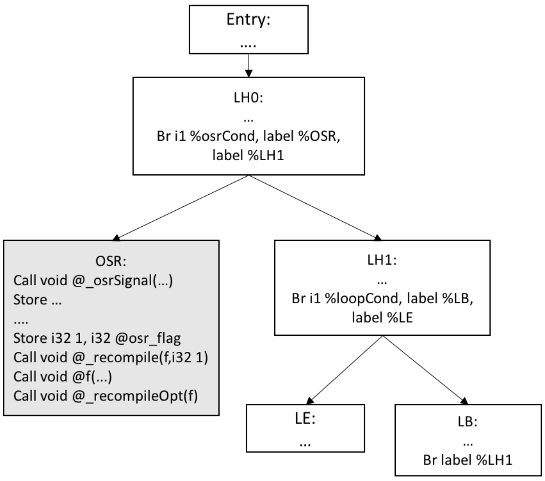
Entry:
....
LH0:
...
Br i1 %osrCond, label %OSR, label %LH1
OSR:
Call void @_osrSignal(...)
Store ...
....
Store i32 1, i32 @osr_flag
Call void @_recompile(f,i32 1)
Call void @(...)
Call void @_recompileOpt(f)
LH1:
...
Br i1 %loopCond, label %LB, label %LE
LE:
...
LB:
...
Br label %LH1

FIGURE 3.9: The transformed CFG of the loop in Figure 3.8 after the OSR Pass from CITE.

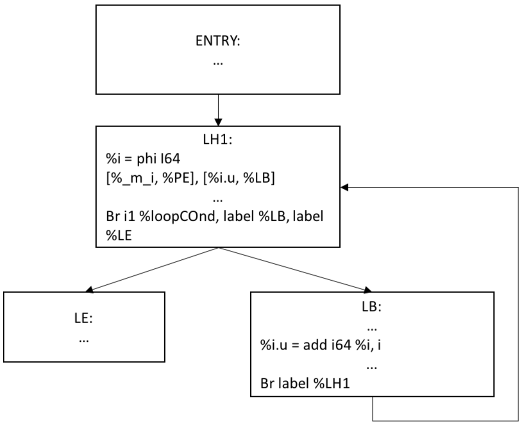
ENTRY:
...
LH1:
%i = phi I64
[%_m_i, %PE], [%i.u, %LB]
...
Br i1 %loopCOnd, label %LB, label %LE
LE:
...
LB:
...
%i.u = add i64 %i, i
...
Br label %LH1

FIGURE 3.10: A CFG of a running function before inserting the blocks for state recovery, from CITE.

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 2.1.2, 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.

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

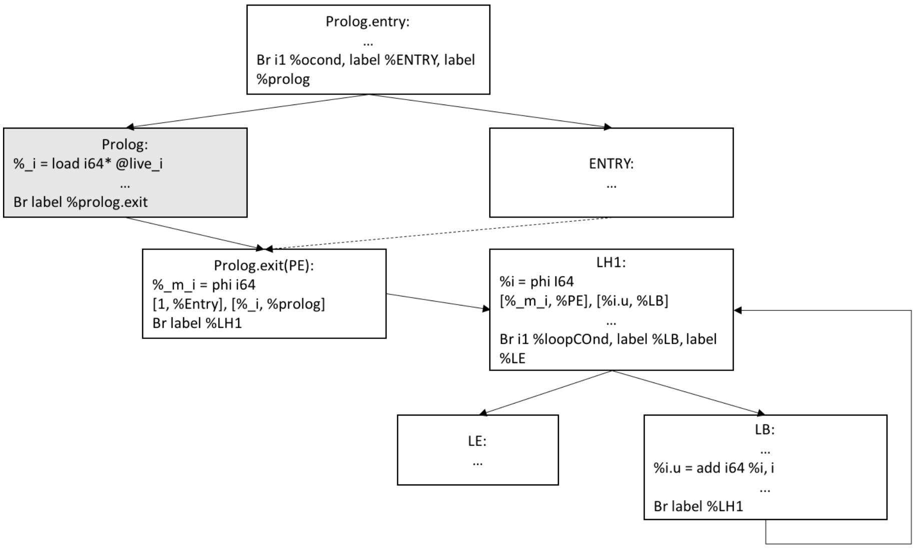
Prolog.entry:
…
Br i1 %ocond, label %ENTRY, label %prolog
Prolog:
%_i = load i64* @live_i
…
Br label %prolog.exit
ENTRY:
…
Prolog.exit(PE):
%_m_i = phi i64
[1, %Entry], [%_i, %prolog]
Br label %LH1
LH1:
%i = phi I64
[%_m_i, %PE], [%i.u, %LB]
…
Br i1 %loopCOnd, label %LB, label %LE
LE:
…
LB:
…
%i.u = add i64 %i, i
…
Br label %LH1

FIGURE 3.11: The CFG of the loop represented in Figure 3.10 after inserting the state recovery blocks, form CITE.

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.

3.4 A classification summary

Chapter 4

OSR Kit

4.1 Overview

4.1.1 An LLVM Library

4.1.2 LLVM IR

4.2 Resolved & Unresolved OSR

4.2.1 The Unresolved OSR

Similarities with MCJIT

4.2.2 The Resolved OSR

4.3 The API

4.3.1 OSR Points

4.3.2 OSR Conditions

4.4 Using OSR Kit to deoptimize

4.4.1 The unease of deoptimizing

4.4.2 SOMETHING

4.4.3 SOMETHING ELSE

4.5 Extending OSR Kit

4.5.1 Handling versions

4.5.2 Unguarded OSR points

4.5.3 Fusing versions?

Chapter 5

Case Study

5.1 The RJIT project

5.2 OSR in RJIT: requirements

5.3 Implementation

5.3.1 The Function Call in RJIT

5.3.2 Challenges

5.4 OSR Conditions

Appendix A

Appendix Title Here

Write your Appendix content here.

Bibliography

[1] Stephen J Fink and Feng Qian. "Design, implementation and evaluation of adaptive recompilation with on-stack replacement". In: *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*. IEEE Computer Society. 2003, pp. 241–252.

[2] Urs Hölzle, Craig Chambers, and David Ungar. "Debugging optimized code with dynamic deoptimization". In: *ACM Sigplan Notices*. Vol. 27. 7. ACM. 1992, pp. 32–43.

[3] Nurudeen A Lameed and Laurie J Hendren. "A modular approach to on-stack replacement in LLVM". In: *ACM SIGPLAN Notices*. Vol. 48. 7. ACM. 2013, pp. 143–154.

[4] Michael Paleczny, Christopher Vick, and Cliff Click. "The java hotspot TM server compiler". In: *Proceedings of the 2001 Symposium on Java TM Virtual Machine Research and Technology Symposium-Volume 1*. USENIX Association. 2001, pp. 1–1.

[5] Sunil Soman and Chandra Krintz. "Efficient and General On-Stack Replacement for Aggressive Program Specialization." In: *Software Engineering Research and Practice*. 2006, pp. 925–932.