ECOLE POLYTECHNIQUE FEDERALE DE LAUSANNE

MASTER THESIS

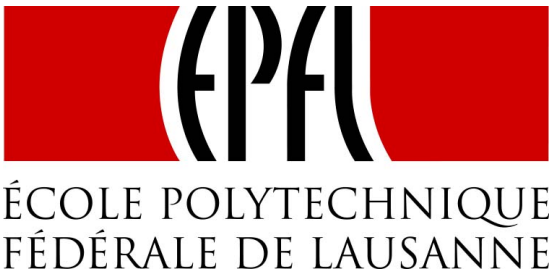
Efficient Runtime Deoptimization for R

Author:
Adrien GHOSN

Supervisors:
Prof. Jan VITEK,
Prof. Viktor KUNCAK

Submitted to the School of Computer Science in partial fulfilment of the requirements for the degree of Master in Computer Science at the

Ecole Polytechnique Federale de Lausanne



March 2, 2016

Declaration of Authorship

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

- 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.

Signed:

Date:

Efficient Runtime Deoptimization for R

by Adrien GHOSN

Submitted to the School of Computer Science in partial fulfilment of the requirements
for the degree of Master in Computer Science
at the

ECOLE POLYTECHNIQUE FEDERALE DE LAUSANNE

March 11th 2016

Abstract

R is a dynamic programming language that combines functional and object-oriented features. R has become a popular language for statistical computing, with more than 2 millions users worldwide[28]. However, R suffers from poor performances[23]. As a dynamic and reflective language, R provides very few guarantees about a program at runtime. It is thus hardly amenable to many common compiler optimizations. As a further matter, R performance bottlenecks seem to be inherent to its semantics[23].

This thesis presents RJIT OSR, an on-stack replacement based prototype for runtime deoptimization in R. On-stack replacement consists in dynamically transferring execution between different programs, while propagating the execution state. RJIT OSR relies on on-stack replacement to preserve an R program's correctness while enabling aggressive, speculative and unsound compiler optimizations. RJIT OSR allows the compiler to target R performance bottlenecks, by locally breaking R semantics.

Keywords: R, LLVM, on-stack replacement, just-in-time compilation, runtime deoptimization, speculative optimizations.

Acknowledgements

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

Contents

List of Figures

List of Tables

List of Abbreviations

OSR	**O**n-**S**tack **R**eplacement
JIT	**J**ust **I**n **T**ime
AOT	**A**head **O**f **T**ime
VM	**V**irtual **M**achine
LLVM	**L**ow **L**evel **V**irtual **Machine**
AST	**A**bstract **S**yntax **T**ree
SEXP	**S**ymbolic **Exp**ression

Chapter 1

Introduction

Dynamic languages, such as JavaScript, Ruby, Python or R, often suffer from poorer performances than statically typed languages, like C, Java or C#. This disparity has multiple causes, among which the execution at runtime of many common programming language constructs, e.g., type tests and method dispatch, that static languages are able to evaluate during compilation. In order to improve dynamic languages performance, language designers have started to investigate ways to shift the compilation later in a program's execution, by relying on just-in-time compilers. Since in dynamic languages, little is known about the program ahead of time, a just-in-time compiler relies on runtime profiling data to perform adaptative optimizations and generate efficient compiled code at runtime. The set of optimizations enabled in a JIT compiler are often more restricted than in a static compiler, where the entire program is available during the compilation. For example, in a static compiler, function g in Figure 1.1 can be type-checked, H and f are known, and their calls can be inlined and optimized. In a dynamic language, H and f might not be yet defined or can be modified, their types are unknown, and no optimization can be performed. As a result, new techniques are needed to lift restrictions in JIT compilers, and enable more agressive, speculative optimizations.

```
1  # Base version of g
2  g <- function(x) {
3    H(f(x))
4  }
5
6  # Unsugared version of g
7  g <- function(x) {
8    t <- f(x)
9    H(t)
10 }
```

FIGURE 1.1: Example.

On-stack replacement (OSR) is a concept that consists in replacing a program that is executing, by another program, while preserving the execution state. Being able to switch between different programs, at runtime, while preserving the progress made so far by the execution, enables to implement aggressive speculative optimizations in JIT compilers. The lack of information about the program's content is compensated by allowing the compiler to take any assumption about the program, and perform optimizations based on this assumption. Whenever the assumption fails at runtime, the invalidated compiled program is replaced by a correct version, which resumes the execution.

```
1  # First optimized version of g
2  g1 <- function(x) {
3    t <- f(x)
4    t * (p ^ y)
5  }
6
7  # Second optimized version of g
8  g2 <- function(x) {
9    t <- f(x)
10   t * (3 ^ 4)
11 }
```

FIGURE 1.2: Optimized versions.

Using OSR, a JIT compiler can assume that H will not be redefined, and inline it during the JIT compilation of g. It can further resolve p and y, type-check them, and optimize the computation of $t*(3^4)$. Whenever the assumption fails, e.g., f triggers a redefinition of H, OSR allows to stop the execution of the program between lines 9 and 10, extract the state, and replace the function with the unsugared version. The execution resumes at line 9 in Figure 1.1. On-stack replacement implementations are state-of-the-art features in advanced virtual machines and JIT compilers.

The R programming language suffers from very poor performances[23]. It further exhibits a lazy evaluation of dynamically typed elements which prevents many common compiler optimizations from being performed. Even worse, performance bottlenecks in R are intimately linked to R semantics. A JIT compiler for R, equipped with an efficient OSR implementation, could lift such restrictions, by generating more efficient unsound code, while preserving the correctness of the program. The focus of this thesis is to provide an OSR implementation in RJIT, a JIT compiler for R, that enables to perform speculative optimizations while preserving the correctness of the program's execution. The next section gives an overview of our solution and its implementation.

1.1 Proposed Solution

The first contribution of this thesis is an overview and synthesis of existing on-stack replacement implementations. This thesis describes in details the main approaches taken to implement OSR, and their advantages and drawbacks.

The second and main contribution of this thesis is the implementation of RJIT OSR, an efficient OSR mechanism in the RJIT compiler for R, that enables agressive speculative optimizations while preserving the correctness of the program. The implementation focuses on the on-stack replacement deoptimization mechanism in RJIT, and strives to provide code instrumentation with as little overhead as possible. RJIT OSR prototypes a mechanism that could, in the future, enable to remove performance bottlenecks that are specific to R semantics.

The third and final contribution of this thesis is the implementation of an OSR-based speculative inliner in RJIT. Function call inlining presents an interesting challenge in R, that requires to consider most of R and RJIT specificities. The OSR inliner is used to evaluate our OSR implementation and provides an example of an interesting use of the OSR concept in an R compiler.

1.2 Paper Overview

The rest of this thesis is organized as follows: Chapter 2 provides an overview of the on-stack replacement concept, defines OSR related vocabulary, and gives a high-level description of OSR mechanisms. Chapter 3 presents related work, i.e., implementations of OSR mechanisms in different virtual machines. It also provides a classification summary that regroups the differences between each implementation. Chapter 4 describes RJIT OSR, the implementation of efficient OSR deoptimization in the RJIT compiler for R. Chapter 5 presents experimental results obtained with a speculative inliner, based on RJIT OSR. Finally, Chapter 6 concludes and presents ideas for future work.

Chapter 2

On-Stack Replacement

This chapter presents the on-stack replacement concepts. The first section provides an overview of on-stack replacement principles. A second section presents on-stack replacement mechanisms, and defines OSR related vocabulary. The third section focuses on the use of on-stack replacement for runtime deoptimization. Finally, the fourth section summarizes constraints on OSR implementations.

2.1 Overview

2.1.1 Definition

On-Stack replacement (OSR) is a concept that consists in replacing a program that is currently executing with another, potentially very different program, while preserving the execution state. OSR implementations allow to interrupt running code during the execution of a function, transform it, e.g., re-optimize it, and resume the execution in the new function at the correct instruction point and state. The action of transferring the execution using the OSR mechanism is called a *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 transitions between different levels of code optimizations. We can therefore reduce it to two main purposes: (1) transforming an executing function into a more optimized version of itself, and (2) undoing transformations that were previously performed. While similar, these two types of transformation have very different goals.

In several virtual machines[26, 20, 16, 9, 29, 8, 6, 27], some of which will be presented in Chapter 3, on-stack replacement has been used to improve the performance of long running functions. When the virtual machine (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 it. However, both techniques rely on the same assumption, namely, that runtime profiling data enables 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 runtime. 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 allow potentially erroneous transformations while preserving correctness.

In summary, 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. The next section presents the advantages that OSR uncovers in both cases.

2.1.2 Why is OSR interesting?

This section highlights the benefits of using on-stack replacement, with respect to both otpimization and 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 than just-in-time (JIT) compilation. In fact, the OSR mechanism, in theory, allows to perform a recompilation at any instruction boundary. As an adaptative compilation mechanism, it enables to delay the compilation of a code artefact and therefore enables to gather more information about the current execution profile. These information can then be used to produce higher quality compiled code, with better execution performances.

For dynamic languages, code specialization is an efficient technique to improve performances[10]. Code specialization consists in tuning the code to better fit a particular use case and set of types, 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. On-stack replacement enables to speculatively specialize code, while preserving the program's correctness.

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 function's execution, 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 of program's execution.

OSR mechanisms enable 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

On-stack replacement 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*[16], *OSR points* [9, 16, 19, 20], *OSR entries*, and *OSR exits*[19, 20]. 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[16] 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. As a result, different OSR implementations yield different restrictions on the location of OSR points.

Guarded & Unguarded

OSR points can be divided into two categories: *guarded* and *unguarded*.

Definition 2.2.2 *A guarded OSR point is an OSR point which execution is subjected to an explicit boolean test, called the OSR condition. In other words, the framework has enough information locally to test whether or not an OSR transition should be taken.*

Definition 2.2.3 *An unguarded OSR point does not require to check a boolean condition before being executed.*

The distinction between guarded and unguarded OSR points is present in the literature (e.g., Jikes RVM OSR implementation[9, 29] and the WebKit OSR implementation[19]). Both solutions present different advantages and fit different use cases.

While simple to model at a high-level, a guarded OSR point presents the disadvantage of increasing the code's length. A boolean condition needs to be tested every time the execution comes across an OSR point. As explained previously, one of OSR's most interesting use case is long running loop's optimization. Executing many times the branching test can therefore greatly impact the program's performance.

Unguarded OSR points enable to avoid this overhead by letting the framework modify the running function on external/environmental events. In WebKit[19], for example, an unguarded OSR point corresponds to a portion of code that can be overwritten at run time, without recompiling the running function, in order to introduce an unconditional execution of an OSR point. This can be viewed as a lazy triggering of an OSR transition, i.e., the instrumentation to enable the transition can be added just before being executed.

OSR Entries & OSR Exits

There is a further distinction between OSR points that are responsible for optimizing the code, and the ones that are used to deoptimize. For that, we will use the same terminology as in WebKit[19] and McOSR[20].

Definition 2.2.4 *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 2.2.5 *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.*

At the implementation level, OSR entries and exits are very similar, if not identical. They both are OSR points and will trigger an OSR transition. Conceptually, however, an OSR exit is part of the mechanism that enables to protect the correctness of the program. Mistakingly not taking an OSR entry is a missed opportunity for better performances. Missing a required OSR exit leads to a possibly incorrect behaviour of the program.

2.2.2 The Transition Mechanism

The OSR mechanism can be narrowed down to its core functionality: allowing a transition at runtime 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 2.2.6 and 2.2.7, and taken from the OSR Kit[6] project. These definitions hold, regardless of the type of OSR transition, i.e., OSR exit or OSR entry, being performed.

Definition 2.2.6 *A function from which we perform an OSR transition is called a* ***from function*** *or* ***source version****.*

Definition 2.2.7 *A function in which we resume the execution after an OSR transition is called a* ***continuation function*** *or* ***destination version****.*

High-Level requirements and challenges for OSR transitions

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 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.

$$f : S \to S$$

$$f(S_s) = S_d$$

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 from 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.

$$f' : S \to S$$

$$f'(S_d) = S_s$$

If the source is a less optimized version of the destination, f' is used during the deoptimization case. The deoptimization case presents a greater challenge than the optimization one. Consider the unoptimized code in Figure 2.1a, and its optimized version in Figure 2.1b.

```
1 a <- 5
2 b <- 6
3 c <- a+b
```

(A) Unoptimized case

```
1
2
3 c <- 11
```

(B) Optimized case

FIGURE 2.1: Variable elimination example.

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. The VARMAP structure in the Jikes RVM OSR implementation[29] and the scope descriptors in SELF debugger[16] are examples of such metadata attached to the code that enable to recover the values of a and b from c.

The OSR transition at a low-level

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 needed.
4. put the correct content inside the registers.
5. modify the program counter and resume the execution.

There exists several examples of OSR implementations that manipulate the program state at machine code level to allow the continuation function to execute in the current stack frame[4, 16, 30]. In order to perform such a modification of the runtime stack and registers, the OSR framework must have extensive 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's semantics.

The transition as a function call

An OSR transition can be modelled as a function call. As explained previously, a 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 and jump to another portion of code. The OSR transition can therefore be viewed as a tail function call, from the source version, to the destination version containing the newly generated 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 the McOSR implementation[20] and WebKit[19]), or
2. Passing all the needed values as function call arguments to the continuation function(e.g., as in OSR Kit[6] and Jikes RVM OSR [9]).

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 propagation of the state has a greater impact on the performance of the execution than a simple function call cost. Figure 2.2 provides a simple example of an OSR mechanism, implemented as a function call, that relies on a buffer to propagate the state in the continuation function. In the from function Figure 2.2a, lines 10 to 13 enable to save the current state in a shared buffer. In the continuation function, i.e., Figure 2.2b, lines 4 to 7 load the state from the buffer.

```
1  f_from <- function(a, b, c) {
2    delta <- b^2 - 4 * a * c
3    if(delta != 0)  //OSR_CONDITION
4      goto OSR
5  CONT:
6    x1 <- (-b)/(2*a)
7    return [x1]
8  OSR:
9    //live values [a,b,c,delta]
10   liveValues <- getLiveValues()
11   for (v in liveValues) {
12     BUFFER.push(v)
13   }
14   return f_cont(a,b,c)
15 }
```

(A) From function.

```
1  f_cont <- function(a,b,c) {
2  //loads the state from
3  //the shared buffer.
4    a <- BUFFER[0]
5    b <- BUFFER[1]
6    c <- BUFFER[2]
7    delta <- BUFFER[3]
8
9  CONT:
10 //Replacement for the CONT
11   if(delta > 0)
12     return pair(a,b,delta)
13   else
14     return i(a, b, c, delta)
15 }
```

(B) Continuation function.

FIGURE 2.2: Using a buffer to pass live values in an OSR transition.

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. Figure 2.3 provides the same OSR transition as in Figure 2.2, but passes live values as arguments to the continuation function.

```
1  f_from <- function(a, b, c) {
2    delta <- b^2 - 4 * a * c
3    if(delta != 0)  //OSR_CONDITION
4      goto OSR
5  CONT:
6    x1 <- (-b)/(2*a)
7    return [x1]
8  OSR:
9    //live values [a,b,c,delta]
10   return f_cont(a,b,c,delta)
11 }
```

(A) From function.

```
1  f_cont <- function(a,b,c,delta) {
2
3  CONT:
4  //Replacement for the CONT
5    if(delta > 0)
6      return pair(a,b,delta)
7    else
8      return i(a, b, c, delta)
9  }
10 //
11 //
```

(B) Continuation function.

FIGURE 2.3: Live values as arguments in an OSR transition.

```
1  fbase <- function (a,b) {
2  PROLOG:
3    if (OSR_LABEL == 0001)
4      goto OSR1
5    else if (OSR_LABEL == 0002)
6      goto OSR2
7    else
8      goto ENTRY
9  ENTRY:
10   //The regular entry code
11   //...
12 CONT1:
13   //Some point in the code
14   //that corresponds to osr1.
15   //...
16 CONT2:
17   //Some point in the code
18   //that corresponds to osr2.
19   //...
20 OSR1:
21   load_state1()
22   goto CONT1
23 OSR2:
24   load_state2()
25   goto CONT2
26 }
```

FIGURE 2.4: Instrumented function with multiple entry points.

The OSR transition requires to enter the continuation function at a special entry point, that corresponds to the OSR point in the from function. In order to decrease the space complexity, one might want to enable multiple entry points per function.

Consider Figure 2.4, where f_{base} provides three entry points: (1) the regular entry point ENTRY, (2) a continuation CONT1 and (3) a second continuation CONT2. The function f_{base} can therefore be used as a continuation function for any from function that has an OSR transition from a point that corresponds to either CONT1 or CONT2, as well as for regular function calls. In order to allow multiple entry points, the f_{base} is instrumented, i.e., a prologue block is executed at its beginning. The prologue is responsible for identifying the f_{base}'s caller. In Figure 2.4, this is done by checking the value of a general flag, called *OSR_LABEL*. According to this value, it then jumps to the correct instruction inside the function, and, if an OSR transition is being performed, loads the state from the shared buffer, as described before.

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. Therefore, we can expect the second solution to have, in general, a higher space complexity (i.e., many versions of the function live in memory at the same time) but a better performance during execution (i.e., smaller execution time).

2.2.3 Versionning

On-stack replacement requires, at some point, to have at least two versions of a function: the from function and the continuation function. Furthermore, the continuation function can either be generated on the fly, i.e., the OSR transition includes a transformation step to generate the correct continuation function, or a previously compiled and cached version can be used.

One version

Allowing only one live compiled version of the function enables to simplify the high-level model of the program's execution. Some OSR implementations, such as the McOSR[20], replace the from function with the continuation one in-place, i.e., the continuation function is compiled and registered at the from function's address. This technique enables to propagate the optimized version through out the code without requiring any modification of the callers. One can view it as an eager propagation of optimizations. Therefore, one can assume that the overall performance of the execution should be improved by the global use of this optimized version. On the other hand, this technique requires to ensure that any OSR entry taken is based on a condition that holds at any possible call site of the function. It might therefore prevent the function's specialization in local sub-scopes. Moreover, depending on the OSR implementation, it might require to trigger an OSR transition for frames in the middle of the execution stack.

Multiple Versions

Fine-grained specialization of functions enables to improve performances locally. In fact, allowing multiple versions of the same function to be live at the same time enables to optimize a function according to local conditions, regardless of other call sites

states. While attractive for the extra flexibility it provides, this solution presents several disadvantages. First, it might increase space complexity. In fact, allowing such fine specialization might lead to having one version per call site. If not carefully handled, this can lead to a dramatic increase of memory usage. Second, this technique is a lazy propagation of optimizations. Assume different calls to the same function. In one call execution, call it C_1, an OSR condition allows to take a specific OSR entry. Other calls, that will take the same OSR entry, do not benefit from the OSR condition tested by C_1, i.e., they will have to wait until they reach the same OSR entry, and verify the same condition, before taking the OSR transition.

Caching vs. Generating on the fly

Depending on the reusability of a function's version, and its likelihood to be needed later in the execution, a time vs. space complexity trade-off appears. Generating a continuation function on the fly is time consuming. If the same OSR entry is taken multiple times, and leads to the same version of a function, caching this version might improve run time performances. On the other hand, if OSR entries are rarely taken, i.e., if OSR transitions are rare, generating the continuation function on the fly enables to save memory space. The cost of the compilation is amortised by the improvements allowed by the optimization and the live range of the function. In other words, generating on the fly is viable if the time performance gain of the optimized version over the entire execution is larger than the time required to generate the continuation function. Furthermore, generating the continuation function on the fly enables to take into account the latest profiling data gathered about the program's execution and use it during the compilation.

2.3 The Deoptimization case

This section presents an overview of the deoptimization case with on-stack replacement. The deoptimization case presents several interesting design decisions and trade-offs.

2.3.1 Why is deoptimization 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 callee. Inlining in static programming languages is a common optimization. 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 run time. 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.

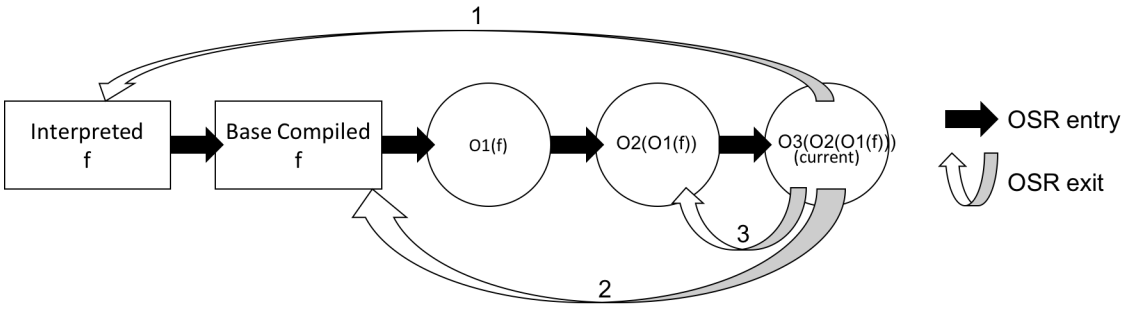


FIGURE 2.5: OSR exit options.

The Interpreter

A first option is to OSR exit back to the interpreter. This approach has been taken by some virtual machines, such as Graal[8] or the Java HotSpot VM[26] (see Section 3.2.1 for more details). This corresponds to the OSR exit 1 in Figure 2.5.

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. Furthermore, this approach gets rid of any assumption taken by the execution framework and goes back to a safe environment. Another way to put it would be to say that, since an assumption failed, the framework takes the conservative decision to go back to an assumption free environment to resume the execution.

Going back to the interpreter is a conservative approach that might impede on the performance of the program. The interpreted version of the function is assumed to be slow. As a result, the function's execution will drastically slow down after exiting to the interpreter. The question is then the following: can we provide the same correctness guarantees while exiting to a more efficient version of the function?

The compiled base version

Another option is to OSR exit to a base, compiled version of the function. It corresponds to the OSR exit 2 in Figure 2.5. This is the approach taken by the WebKit framework[19](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

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 2.5. The function to which we exit is a fully optimized one, supposed to yield better performances than both the interpreted and the base versions. This behavior can be observed in McOSR[20] (see Section 3.3.3) and OSR Kit[6](see Section 3.3.4). This option strives to get the best possible performances, 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 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 underwent 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 2.5. 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 2.5. 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 Section 3.3.4, can be used to implement such a solution.

2.4 Constraints on the OSR Mechanism

2.4.1 The OSR trade-offs

On-stack replacement is a powerful technique, that requires to be tailored to its use cases in order to exhibit good results. As seen in this chapter, OSR can be implemented in several different ways, which in turn might lead to very sparse performance results. For example, should the framework cache the compiled versions or should it generate the continuation functions on the fly? This dilemma cannot be answered in the general case. It depends on many conditions, among which the code reusability, i.e., what is the likelihood that the same continuation will be needed later during the program's execution? There is also the trade-off between time and space complexities: caching increases the space usage, generating on the fly increases the time spent in the compiler.

The variety of performances that can be observed does not only depend on the design choices, or the target language, but also on the program's specificities. A program with a certain profile might benefit from a specific OSR implementation, while another might be more amenable to another OSR design. It is hard to know, before hand, which OSR implementation would yield the best results on a specific program, without having enough profiling information about the program's behavior. Furthermore, the program's behaviour might depend on some runtime condition, and one OSR implementation is not guaranteed to be the most efficient in every run.

2.4.2 Limits on optimizations

The OSR instrumentation of the code limits the optimizations that the compiler can perform. First, the OSR points must be preserved by the compiler, during the transformations. This has also the disadvantage of impeding on the compiler's performance and more specifically on the quality of the code that it produces. For example, in the Java HotSpot[26], the OSR instrumentation might extend the live range of variables,

therefore preventing the compiler from removing them. Another example is the SELF Debugger[16], which deoptimization process does not support tail call elimination and therefore does not implement it in the compiler. Tail call elimination is a compiler optimization that enables, in the case of a tail call, to avoid allocating a new stack frame for the callee. It is commonly used to optimize tail call recursions. The SELF debugger has no way of knowing how many stack frames were eliminated and is therefore unable to reconstruct them properly. However, this limitation does not extend to every deoptimization implementation. A carefully instrumented execution framework could keep track of state information and execution flow in order to enable tail call elimination OSR support.

OSR, and more specifically OSR exits, restricts code motion. In the case of deoptimization, the guarded OSR point protects some portion of code that must not be executed if the OSR exit is to be taken. This implies that the compiler must not move any instruction that depends on this OSR exit condition above the OSR point. The OSR instrumentation therefore limits code motion, which in turn might decrease the number of opportunities for compiler optimizations.

Chapter 3

Related Work

This chapter presents the on-stack replacement related work and focuses on OSR implementations in virtual machines. The first section describes the OSR origins and SELF debugger runtime deoptimization mechanisms. The second section reports different on-stack replacement implementations in virtual machines. The third section describes OSR implementations developed in the low-level virtual machine (LLVM). Finally, the fourth section contains a recapitulatory table of different OSR implementations.

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[4, 17]. As a result, the language's implementation depends on a set of aggressive optimizations to achieve good performances[3, 16]. SELF provides an interactive environment, based on interpreter semantics at compiled-code speed performances.

Providing source level 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"[16], the authors came up with a new mechanism that enables to dynamically deoptimize code at specific interrupt points in order to provide source code level debugging while preserving expected behaviors.

Hölzle, Chambers, and Ungar[16] present the main challenges encountered to provide debugging behaviors, 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 3.1, taken from [16], 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', and hence disappears 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[16] highlight the impact of code motion and instruction

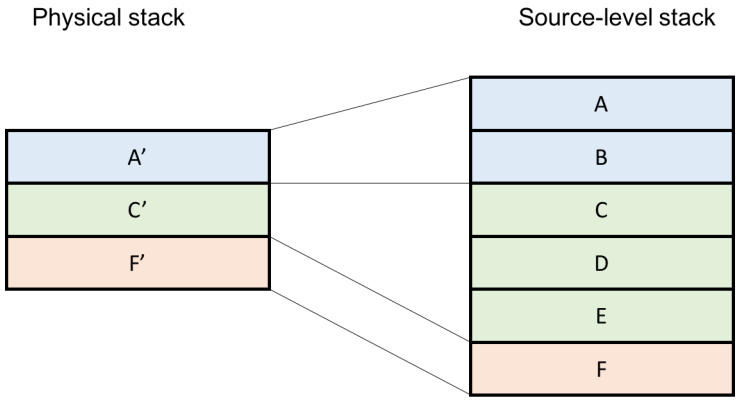


FIGURE 3.1: Displaying the stack, figure from [16].

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 source-level single-stepping for the debugger.

Compiler optimizations prevent dynamic changes from being performed in the debugger. Hölzle, Chambers, and Ungar[16] identify two separate issues: changing variable values, and modifying procedures (i.e., functions). To illustrate the first case, the paper[16] 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, Hölzle, Chambers, and Ungar[16] provide an example where a function has been inlined by the compiler, but redefined by the user in the debugger.

The paper[16] 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 values 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[16] 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.

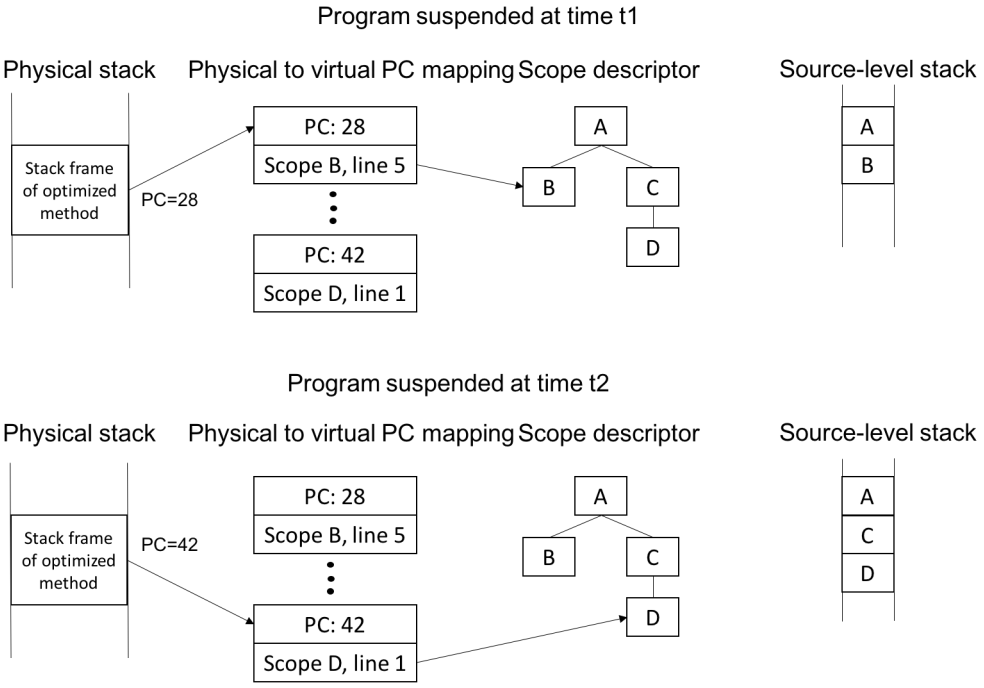


FIGURE 3.2: Recovering the source-level state (from CITE).

The deoptimization process follows 5 steps described in [16] and summed up here:

1. Save the physical stack frame and remove it from the runtime stack.
2. Determine the virtual activations in the physical one, the local variables and the virtual PC.
3. Generate completely unoptimized compiled methods and 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.

Hölzle, Chambers, and Ungar[16] also describe *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. The SELF debugger relies on a weaker, relaxed restriction by enabling deoptimization only at certain points called *interrupt points*. At an interrupt point, the program state is guaranteed to be consistent. The paper[16] defines two kinds of interrupt points: method prologues, and backward branches (i.e., end of loop bodies). It estimates the length of the longest code sequence

that cannot be interrupted, i.e., the length of a code sequence that does not contain neither a call nor a loop end, to be a few dozens of instructions. Interrupt points are also inserted at all possible runtime 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 section 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 information about the program being executed. Furthermore, modern virtual machines already provide adaptative strategies to recompile hot functions[2, 26, 18, 31], and dispatch calls to the newly compiled versions. OSR enables this transition to happen when the function is executing, rather than during a future call.

This section presents several examples of VMs that support on-stack replacement. The section is divided into two parts: we first present 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 Java HotSpot

The Java HotSpot Performance Engine[24] 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 Server Compiler[26] supports on-stack replacement of interpreter frames by compiled-code frames, and deoptimization from compiled-code back to the interpreter. It relies on adaptative optimizations, that focus on performance critical

methods[26, 18]. The Java HotSpot Server compiler identifies such methods by using method-entry and backward-branch counters, along with other heuristics related to the method's caller. Thanks to these counters, the runtime system is able to identify frequently executed or long running methods and compile them to improve their runtime performance. It can further decide to compile one or several of the methods callers. This is called an *upward traversal* and enables to speed the path that leads to a hot method's invocation. On-stack replacement is used when the backward-branch counter exceeds the *OnStackReplacementThreshold* value. The continuation function is compiled with an *additional* entry point at the target of the backward-branch. A continuation function is therefore able to serve OSR transitions as well as future method invocations. The runtime system transfers the execution from the interpreter to this newly compiled code.

In HotSpot, deoptimization might be triggered by two different events: the compilation of a reference to an uninitialised class, and a class loading that invalidates some previous compilation optimization. Class initialization and class loading are performed by the interpreter. Whenever a reference to an uninitialized class is compiled, HotSpot generates an uncommon trap, i.e., a trampoline back to interpreted mode. The optimized code is marked as being unusable, and any thread entering the method is interpreted until the recompilation of the method is achieved properly. The second form of deoptimization happens when a class loading invalidates some optimization performed while generating compiled code (e.g., inlining of methods). A thread that is executing in the invalidated method is "rolled forward"[26] to a safepoint. When the safepoint is reached, their native frame is extracted and converted into an interpreter frame. The thread then continues its execution in the interpreter. For consistency, the class load that invalidated the compiled method becomes visible to the thread only when it reaches its safepoint.

The deoptimization process requires the Java HotSpot to be able to generate an interpreter JVM state at different points of the program. In order to do so, the Java HotSpot Server records the "exact JVM state"[26] at each safepoint and procedure call. This implies that the entire JVM state is considered live at the safepoint, which in turn, might extend the live range of some values. During code emission, the compiler emits a table that maps the JVM state to the resting place of the generated optimized code's JVM state information.

3.2.2 Jikes RVM

The Jikes Research Virtual Machine (RVM)[25] is an open source, self hosted, i.e., it is entirely implemented in Java, virtual machine for Java programs. One specificity of the Jikes RVM is that it exhibits a *compile-only* approach, i.e., the system compiles all bytecode to native code before execution. The RVM provides two compilers: a *baseline* compiler, which generates poor quality code quickly, and an *optimizing* compiler. The optimizing compiler provides a full suite of optimizations, categorized into three different levels. The RVM provides advanced state-of-the-art features such as dynamic compilation, adaptive optimizations, garbage collection, thread scheduling and synchronization[25].

Jikes RVM is an extensible framework in which on-stack replacement techniques have been implemented in two steps, i.e., a first implementation[9], and then an extend version of OSR[29]. Fink and Qian[9] implemented OSR support in Jikes RVM. 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 locals 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[9] generate 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 [20, 26], 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 a 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[29] proposed a general-purpose OSR mechanism, for the Jikes RVM, 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[9] to enable OSR transition at, and across 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 automatically updated accordingly. Once the compilation completes, the VARMAP is encoded into a compressed map that contains an entry for each OSR point present in the method. The VARMAP is decoupled from the compiled code, i.e., it does not influence the liveness of local variables. Soman and Krintz[29] 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 invalid, 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 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[19] is an open source web browser engine used to improve JavaScript performances. It exhibits a Four-Tier VM architecture. The WebKit runtime compilation flow is described by Figure 3.3.

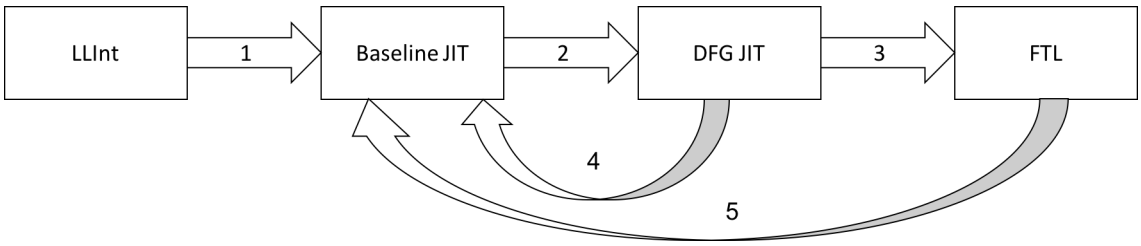


FIGURE 3.3: The WebKit Four-tier optimization flow.

Forward arrows represent *OSR entries*, i.e., a transformation that yields a more optimized version of the code at runtime. Backward arrows correspond to *OSR exits*, i.e., a transformation that yields a less optimized version of the code at runtime. 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 Fourth-Tier LLVM(FTL)[27] relies on LLVM machine code optimizations to generate a fast version of portions of the code. In order to hide the cost 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 of the function for each entry point for two main reasons: 1) LLVM allows only single entry points for functions, and 2) 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 as inputs. 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 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 while proving to be reliable. Although successful, it does not provide a general and reusable framework for OSR in LLVM that other projects could benefit from.

3.3 OSR & LLVM

3.3.1 What is LLVM?

LLVM[14, 21], formerly called Low-Level Virtual Machine, is a compiler infrastructure that provides 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[14].

The LLVM intermediate representation is a language-independent set of instructions that also provides a type system. The LLVM IR is in static single assignment (SSA) form, which requires every variable to be defined before being 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[1]. The SSA requirement for variables to be assigned only once requires a special mechanism, called a ϕ-node, when 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 merging of two control flow branches.

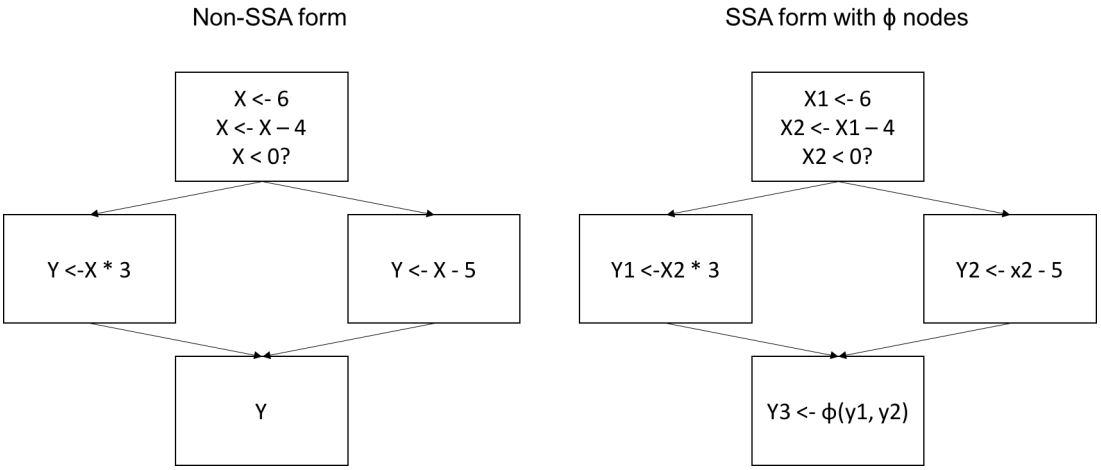


FIGURE 3.4: Example of ϕ-node in SSA form.

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-based compilers exist for several mainstream/popular languages such as Java, Python, Objective-C, and Ruby. 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[14] 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 instruction sets corresponding to different architectures. By implementing OSR in LLVM, we get portability among these platforms for free.

3.3.3 McOSR for LLVM

The McOSR OSR support[20] 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 reusable technique completely defined at the LLVM IR level and compatible with the standard LLVM distribution. The implementation has been tested in the McVM/McOSR[5, 22], an open source VM and JIT for MATLAB, built on LLVM. Their implementation answers five challenges, listed in the paper[20], and reproduced here:

1. Identifying correct interrupt points and using the current LLVM IR to represent them.
2. Using the LLVM tools to 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.

Lameed and Hendren[20] claim 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[20] does not seem to allow to skip intermediary versions.

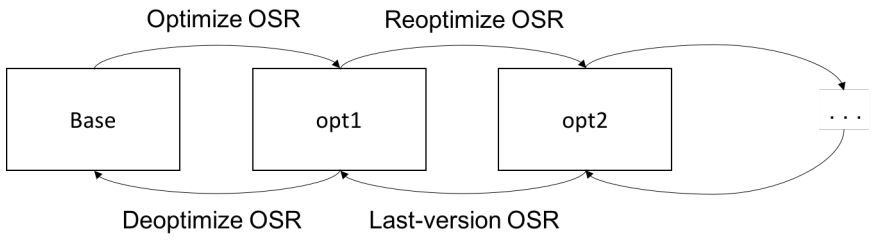


FIGURE 3.5: OSR classification from[20].

The McOSR library on-stack replacement implementation fits into the regular JIT infrastructure provided by LLVM as described in Figure 3.6 taken from[20]. The left

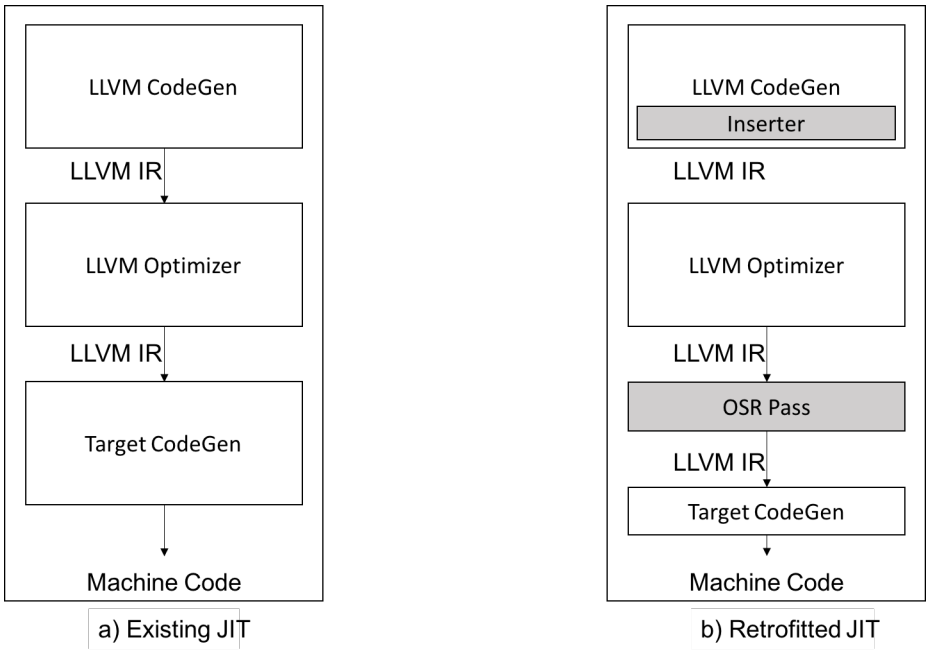


FIGURE 3.6: Retrofitting an existing JIT with OSR support from CITE

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 McOSR 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 point 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

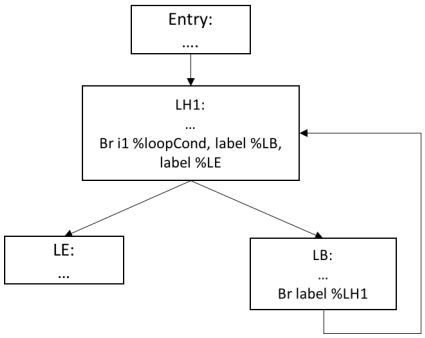


FIGURE 3.7: A CFG of a loop with no OSR point from[20].

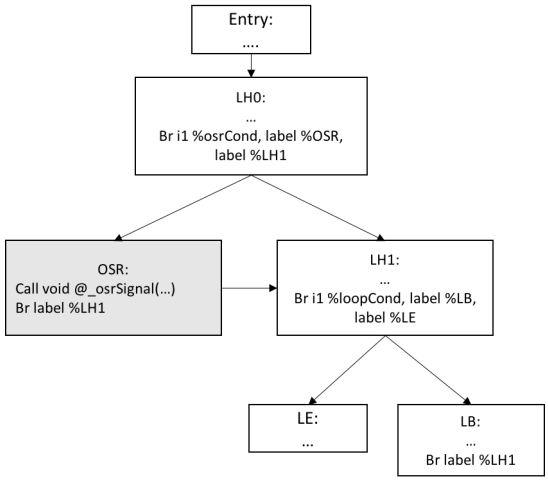


FIGURE 3.8: The CFG of the loop in Figure 3.7, after inserting an OSR point from[20].

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 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[20] 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, McOSR support is a modular, and general purpose 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

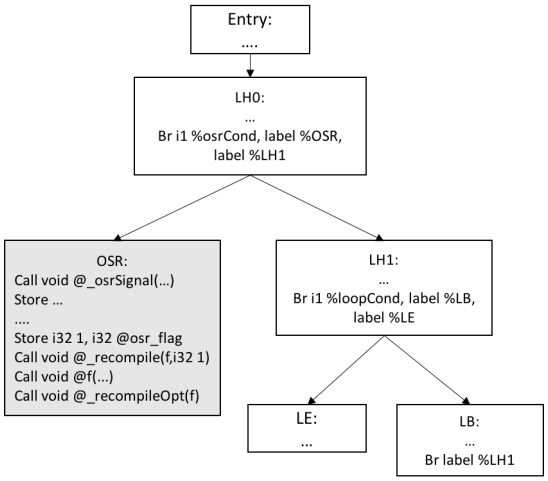


FIGURE 3.9: The transformed CFG of the loop in Figure 3.8 after the OSR Pass from[20].

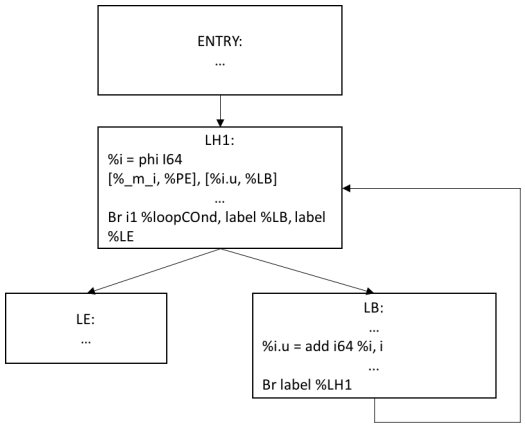


FIGURE 3.10: A CFG of a running function before inserting the blocks for state recovery, from[20].

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 2.1.2, 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 McOSR 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, McOSR library is not currently used in production or any important project. As mentioned earlier, WebKit is efficiently integrated in Apple

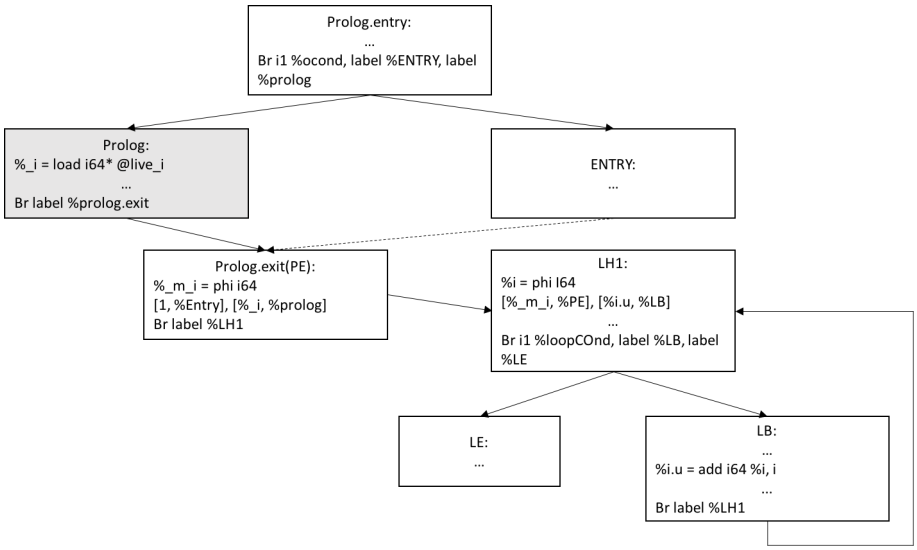


FIGURE 3.11: The CFG of the loop represented in Figure 3.10 after inserting the state recovery blocks, form[20].

Safari's web browser, which provides useful feedback on its performances. The lack of usage of McOSR library prevents us from collecting performance results and assessing its efficiency. Furthermore, the experimental evaluation in[20] 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.

3.3.4 OSR Kit for LLVM

OSR Kit[6] is a general purpose, target independent, implementation of on-stack replacement for LLVM. As such, it can be used by any LLVM based compiler. The main goals of the OSR Kit project are:

1. To allow chained OSR transitions, i.e., a continuation function can be instrumented to allow OSR transitions.
2. To support OSR entries and exits via the same instrumentation.
3. To allow transitions at arbitrary function locations.
4. To allow continuation functions to be either generated at runtime, or already known at compilation time (i.e., generated on the fly or cached).
5. To encapsulate and hide the OSR implementation details from the front-end.
6. To encode OSR transitions entirely at the LLVM IR level.
7. To limit the intrusiveness of the OSR instrumentation.
8. To allow the LLVM's compilation pipeline to generate the most efficient native code for an instrumented function.

The OSR Kit library's main contribution is the implementation of an efficient and flexible OSR transition mechanism. This transition can be used to implement OSR entries and OSR exits alike. OSR points are allowed to be inserted at any instruction boundary. The OSR library is designed to give the user as much freedom as possible.

OSR Kit provides mechanisms for generating the continuation function on the fly or reusing a cached version. Where other implementations, such as McOSR[20], made a clear choice between the two techniques, the OSR Kit library decided to enable both of them, hence allowing the user to experiment and select the design that better suits specific use cases.

OSR Kit strives for non-instrusive instrumentation of the code in order to have as little impact as possible on the LLVM transformation passes performances. One of the main advantages of using the LLVM framework is that LLVM transformation passes and optimizations can be automatically added to the compilation pipeline in order to improve the quality of the code generated. A heavy instrumentation of the LLVM IR might impede on the performances observed for such passes.

Open vs. Resolved OSR

The *open OSR* scenario corresponds to the case where the continuation function is generated on the fly, when the OSR transition is fired. Deferring the compilation of the continuation function allows profiled-guided compilation strategies to gather as much information as possible about the current state of the execution, and therefore generate the best possible code.

The open OSR scenario is implemented as a call to a stub function, call it f_{stub}. The f_{stub} function is responsible for generating the continuation function, and then transferring the execution to it. The f_{stub} function receives all the values live at the OSR point as input, and propagates them to the continuation function. The continuation function is generated by a special *gen* function, that takes as inputs the base function f, i.e., the from function, and the OSR point that triggered the transition.

Figure 3.13 corresponds to the signature of the OSR Kit function that enables to insert open OSR points. The *destFunGenerator* is the f_{stub} function and is of type *DestFunGenerator*, described in Figure 3.12.

The open OSR is similar, in theory, to the McOSR implementation[20]. Both of them rely on a *gen* function to generate the continuation function when the OSR transition is fired. However, D'Elia and Demetrescu[6] made the choice to perform these operations, i.e., generating the continuation function and transferring the execution, inside a separated function, rather than instrumenting the from function directly. This choice was made in order to minimize the code injected inside the from function. In fact, the instrumentation, even when inserted in a separated branch, might interfere with compiler optimizations. Examples of such interferences are the increase of register pressure, the alteration of code layout and instruction cache behavior.

The *resolved OSR* scenario corresponds to the case where the continuation function is known prior to fire the OSR transition, and the from function has been instrumented

```
1  /* @brief Signature of a code generator for open OSR points
2  /*
3  /* A code generator for open OSR transitions takes four parameters: the
4  /* Function @F1 in which the OSR point was inserted, the instruction
5  /* @OSRSrc for which an OSR decision was taken, the address @extra
6  /* of the additional data structure created by the front-end for the
7  /* OSR point, and the run-time address @profDataAddr of the
8  /* (optional) profiled Value.
9   */
10 typedef void* (*DestFunGenerator)(llvm::Function* F1,
11                                    llvm::Instruction* OSRSrc,
12                                    void* extra, void* profDataAddr);
```

FIGURE 3.12: The DestFunGenerator type.

to transfer the execution to this specific compiled version. In the resolved OSR scenario, the transition is expected to be faster than in the open scenario, since there is no compilation to perform. This technique also enables to reuse functions that were compiled previously, e.g., either because several from functions have the same continuation function, or because an OSR point is fired several times during the execution of the program.

The resolved OSR is implemented as a call to the continuation function, which takes as inputs all the live variables at the OSR point. The continuation function is instrumented to jump at the correct location inside the code to resume the execution. Figure 3.14 illustrates the resolved OSR scenario.

Figure 3.15 corresponds to the signature of the OSR Kit function that enables to insert resolved OSR.

Resolved OSR points and conditions

In the OSR Kit library[6], there is no distinction at the implementation level between OSR exits and entries. The general mechanism used is the OSR point. An OSR point corresponds to a labelled LLVM basic-block that contains the call to the continuation function and the return instruction to propagate the result of the call. OSR points are protected by an OSR condition. The OSR Kit library allows an OSR condition to be any vector of LLVM instructions. The last instruction is automatically used by the framework as condition for an LLVM branch instruction. If the condition succeeds, the OSR transition is fired. Otherwise, the execution continues in the function.

Figures 3.16 and 3.17 provide an example of OSR instrumentation. In Figure 3.16, we have the R declaration of two functions, f and g, and the simplified LLVM IR generated for g in RJIT. Figure 3.17 presents the instrumented version of g, in which a resolved OSR point has been inserted. Lines 5 and 6 correspond to the OSR condition. In this example, the OSR is fired if line 1 does not yield the same function for f as the one that was used by the compiler when it generated the instrumented function. If the OSR is fired, the execution jumps to the *OSR_fire* basic block and calls the continuation function, namely *OSRCont*. The OSR point *OSR_fire* calls the continuation and

```
1  /* @brief Insert an open OSR point in a function.
2  /*
3  /* @param Context LLVM Context to use for OSRKit.
4  /* @param F Source function where to insert the OSR point.
5  /* @param OSRSrc Instruction in @F to take an OSR decision for.
6  /* @param extraInfo Address of the auxiliary data structure to pass to
7  /* the code generator.
8  /* @param cond Condition to evaluate in order to take an OSR decision.
9  /* @param profDataVal Value to profile at the OSR point; use @c nullptr
10 /* when no Value needs to be profiled.
11 /* @param destFunGenerator Code generator to invoke when the OSR is
12 /* fired.
13 /* @param valuesToTransfer Custom set of values to transfer at the OSR.
14 /* point; use @c nullptr to transfer all the live values.
15 /* @param LA LivenessAnalysis results computed for @F.
16 /* @param config Additional settings for the OSR point insertion.
17 /* @return A pair in which the first element is the function with the
18 /* inserted OSR point and the second is the OSR stub function.
19  */
20 static OSRPair insertOpenOSR(llvm::LLVMContext& Context, llvm::Function& F,
21                              llvm::Instruction& OSRSrc, void* extraInfo,
22                              OSRCond& cond, llvm::Value* profDataVal,
23                              DestFunGenerator destFunGenerator,
24                              std::vector<llvm::Value*>* valuesToTransfer,
25                              LivenessAnalysis* LA, OSRPointConfig& config);
```

FIGURE 3.13: The insertOpenOSR prototype.

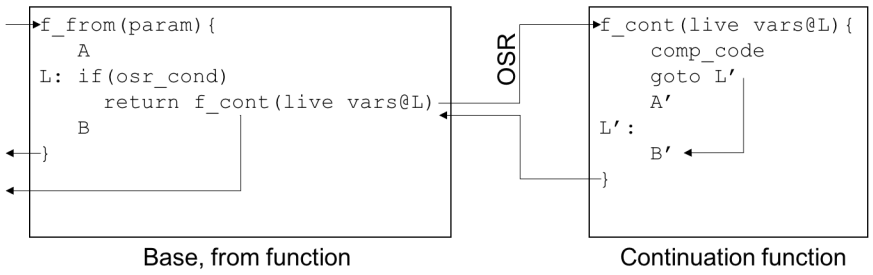


FIGURE 3.14: Resolved OSR scenario, from[6].

propagates its return value line 15. The *OSR_split* block corresponds to the regular continuation of the function f when the OSR is not fired.

The continuation function & StateMap

The OSR Kit library heavily relies on a special object, called a *StateMap*, to keep a mapping between the from function and the continuation function. The StateMap enables to register unidirectional and bidirectional mappings between LLVM instructions and function arguments. The default constructor for a StateMap relies on the LLVM ValueToValueMapTy[15](VMap) to automatically extract the mappings. A VMap can be filled with such mappings when the LLVM cloning functions are used, i.e., the OSR Kit encourages you to generate the continuation function by cloning either the base or the from function such that a VMap is already available when the framework needs

```
/* @brief Insert a resolved OSR point in a function.
/*
/* @param Context LLVM Context to use for OSRKit.
/* @param F1 Source function where to insert the OSR point.
/* @param OSRSrc Instruction in @F1 to take an OSR decision for.
/* @param F2 Target function for the OSR transition.
/* @param LPad Landing pad in @F2 where to resume the execution at.
/* @param cond Condition to evaluate in order to take an OSR decision.
/* @param M StateMap object between @F1 and @F2.
/* @param config Additional settings for the OSR point insertion.
/* @return A pair in which the first element is the function with the
/* inserted OSR point and the second is the OSR continuation function.
 */
static OSRPair insertResolvedOSR(
                    llvm::LLVMContext &Context, llvm::Function &F1,
                    llvm::Instruction &OSRSrc, llvm::Function &F2,
                    llvm::Instruction &LPad, OSRCond &cond,
                    StateMap &M, OSRPointConfig &config);
```

FIGURE 3.15: The insertResolvedOSR prototype.

```
> f <- function(a) a
> g <- function() f(8)

define  @g(%consts, %rho, i32 %useCache) #0 gc "rjit" {
start:
  %f = call  @getFunction(%rho, %consts, i32 1)
  %0 = call  @userLiteral(%consts, i32 2)
  %1 = call  @constant(%consts, i32 0)
  %2 = call  @icStub_1(%0, %1, %f, %rho, (, , i32)* @g, i64 13)
  ret  %2
}
```

FIGURE 3.16: Simplified original LLVM IR.

to introduce the instrumentation. The StateMap also provides an API to register and unregister mappings by directly providing the two instructions.

Both deoptimization and optimization rely on the same kind of continuation functions, and on the same instrumentation tools. A continuation function, in OSR Kit, has a special function signature that takes as inputs all the variables that were live at the instruction at which the OSR transition was fired. Its return type is the same as the from and the base (unmodified and non-instrumented) functions. The OSR Kit instrumentation inserts a special basic block, called *OSR ENTRY*, at the beginning of the continuation function (see Figure 3.18). The OSR ENTRY is responsible for executing the *compensation code* and jumping to the correct block inside the continuation function. The compensation code is specified by the user as a vector of instructions associated with a source value, when the OSR is inserted in the code. The continuation instruction, i.e., the instruction to which the OSR ENTRY jumps to, is assured by the OSR Kit library to be the first instruction in the destination block. Everything between the OSR ENTRY and the continuation block becomes dead code. The dead instructions are automatically eliminated by the LLVM transformation passes. In any case, their only impact is on the space used up by the program, their presence does not increase the

```
define @gFrom(%consts, %rho, i32 %useCache) #0 gc "rjit" {
start:
  %f = call @getFunction(%rho, %consts, i32 1)
  %0 = call @userLiteral(%consts, i32 2)
  %1 = icmp ne %f, inttoptr (i64 12515896 to )
  br i1 %1, label %OSR_fire, label %OSR_split

OSR_split:                                        ; preds = %start
  %2 = call  @constant(%consts, i32 0)
  %3 = call  @icStub_1(%0, %2, %f, %rho, (, , i32)* @gFrom, i64 13)
  ret  %3

OSR_fire:                                         ; preds = %start
  %OSRRet = call  @OSRCont(%rho, %consts)
  ret  %OSRRet
}
```

FIGURE 3.17: Simplified instrumented LLVM IR.

execution time of the continuation function. It might, however, impact on later optimizations performed on the code.

```
define @OSRCont(%consts_osr, %rho_osr, %f_osr, %anon_osr) gc "rjit" {
OSR_entry:
  call void @fixClosure(i64 1)
  br label %OSRCont_split

start:                                            ; No predecessors!
  %f = call @getFunction( %rho_osr, %consts_osr, i32 1)
  %0 = call @userLiteral( %consts_osr, i32 2)
  br label %OSRCont_split

OSRCont_split:                                    ; preds = %OSR_entry, %
    start
  %anon_fixSSA = phi [%anon_osr, %OSR_entry], [%0, %start]
  %f_fixSSA = phi [%f_osr, %OSR_entry], [%f, %start]
  %1 = call @constant(%consts_osr, i32 0)
  %2 = call @icStub_1(%anon_fixSSA, %1, %f_fixSSA, %rho_osr, ( , , i32)
      @rfunction.1.2, i64 0)
  ret %2
}
```

FIGURE 3.18: Simplified continuation function LLVM IR.

D'Elia and Demetrescu[6] claim that passing live values as arguments to the continuation function is more efficient than loading them from a buffer. According to [9], generating a dedicated function to resume the execution, as opposed to instrumenting a general function to serve as continuation as well as a regular function, is expected to lead to better results. In other words, it is supposed to be better to keep the base function to serve regular calls, and trigger OSR transitions whenever we need them, during the execution of the function. This claim is debatable in the case of deoptimization. Chapter4 expands on the question.

3.4 A classification summary

This section presents a summary of the various OSR implementations presented in this Chapter. The summary is presented in Table 3.1 and focuses on the state propagation technique, the types of OSR points supported, the implementation of the transition mechanism, and the target of OSR exits.

	SELF	HotSpot	Graal	Jikes RVM	WebKit	McOSR	OSR Kit
State propagation	Scope descriptors	JVM state & native frames	FrameState nodes	JVM scope descriptor, VARMAP & call arguments	Buffer & Stackmaps	Mapping between live values	Call arguments
Transition mechanism	Stack frames & PC manipulation	Trampoline to interpreter	Trampoline to interpreter	Function call	Trampoline to baseline	Function call	Function call
Modification of function signature	No	No	No	Yes	No	No	Yes
Multiple entry points Opt/Deopt	Yes/Yes	No/Yes	No/Yes	No/No	No/Yes	No/Yes	No/No
OSR points locations	Mostly method prologues & backward branches	Mostly backward branches & safepoints	Between side effecting nodes	Mostly method prologues & loop headers	Mostly method prologues & loop headers	Mostly method prologues & loop headers	Anywhere the state can be reconstructed
Guarded/ Unguarded	Both	Both	Both	Both	Both	Guarded	Guarded
Exit target	Unoptimized compiled version	Interpreter	Interpreter	Less optimized compiled version	Baseline JIT	Less optimized compiled version	Less optimized compiled version

TABLE 3.1: Summary of different OSR implementations features.

Chapter 4

RJIT OSR

CORRECT COMPILATION INTO TRANsFORMATION

4.1 Overview

4.1.1 Justification & Goals

R is a programming language and software environment for statistical computing and graphics, developed by the R Foundation for Statistical Computing[32]. R is a dynamic, lazy, functional, object oriented programming language where the basic data type is the vector. The dynamic features provided by the language include reflection over the environment, the ability to generate source code for unevaluated expressions and to treat text as code (e.g., functions *parse* and *eval*). Even though R is considered functional, i.e., functions are first-class values, and arguments are deep copied and lazily evaluated, the language does not optimize recursions and, instead, encourages vectorized operations.

Although very flexible and widely used for statistical data analysis, the R programming language's implementation suffers from poor execution performances, compared to equivalent programs executed in C or Python[23]. According to Morandat et al.[23], on the Shootout baseline benchmarks suite[35], R is on average 501 times slower than C and 43 times slower than Python. According to [23], R programs usually consume large amounts of memory, and spend an important part of the execution (e.g., on average 30% for the Shootout benchmarks) doing memory management. These slow downs have various causes. For example, in R, all user data must be heap allocated and garbage collected, when languages like C allow stack allocated data. Another R behavior that has a large impact on memory consumption is boxing. All numeric data has to be boxed in R, leading to high overheads in programs containing an important amount of single numbers. For example, in the Bioconductor corpus of programs[33], 36% of vectors allocated contained only a single number[23]. It is important to note that other R specific behaviors, such as looking up variables, matching parameters with arguments, and copy-on-write mechanisms used for call-by-value semantics have non negligible impacts on the performance of R programs[23].

The RJIT project[34] strives to improve R programs performances by providing an LLVM based JIT compiler for R. The RJIT compiler plugs into a modified version of the GNUR interpreter and enables to JIT compile R programs into native code. The RJIT compiler is still pretty young, only a few months old. As a result, RJIT is currently trying to identify code behaviors that impact the execution performances, and plans to

optimize code compilation to decrease their costs. Due to R semantics, such compilation optimizations might be hard to implement. More specifically, as explained above, R performance limitations are intrinsically linked to R semantics. On-stack replacement therefore appeared as an interesting solution to enable aggressive and speculative optimizations, able to generate compiled code that does not respect R semantics, while preserving the correctness of the program's execution. The goal of this master thesis is thus to prototype a flexible and extensible OSR mechanism for speculative optimizations.

This master thesis prototypes OSR deoptimization mechanisms in RJIT. The RJIT OSR implementation relies on the OSR Kit[6] library transition mechanism. The OSR Kit library code is available on Github[7], and can be easily integrated in any LLVM project by simply copy-pasting the OSR Kit files inside of it. Instead of reimplementing the same transition mechanisms for RJIT OSR, the choice was made to modify and extend this library. This enables to focus on the deoptimization case which, according to Section 2.3.1, is the most challenging and interesting use case of on-stack replacement. The OSR deoptimization further allows speculative optimizations, which enable the RJIT compiler to port static compilation optimizations to R, while preserving the correctness of the program being executed.

4.1.2 OSR Kit limitations

While very flexible, the OSR Kit[6] library presents several disadvantages and exhibits costly behaviors that do not perfectly fit the deoptimization process. This section lists the OSR Kit library's limitations with regard to OSR exits implementation.

The main advantage of the open OSR is to leverage profiled-guided compilation strategies to generate efficient code. However, generating a transformation that removes optimizations is hard and does not play well with the OSR Kit instrumentation. Such a transformation needs to take into account other optimizations that might have been performed on the code after the optimization was applied, i.e., deoptimizing requires to keep track of which transformations performed on the code depend on this optimization, and undo them. For the OSR point, the framework relies on a StateMap that matches values in the from and the continuation functions. That implies that, in order to generate the continuation function on the fly for the deoptimization case, one is required to 1) be able to reverse an optimization, that might have interfered with other optimizations, and 2) to generate a correct StateMap to give as input to the OSR Kit machinery. Both of these requirements are hard to satisfy. Finally, the deoptimization case is used to preserve correctness in the program and must be conservative. Performance is a soft issue, and hence leveraging profiled-guided compilation strategies is not a main objective.

Once an OSR exit is taken, it is more likely to be fired in subsequent calls. For example, in the case of call site inlining, the OSR exit is fired when the inlined function is redefined. In subsequent calls, the OSR exit will also be triggered. As a result, if the OSR exit is an open OSR point, the framework will have to perform expensive operations to generate the correct continuation function every time the OSR exit is triggered. In the light of these observations, we can conclude that the open OSR design does not

fit well the deoptimization case.

The resolved OSR corresponds to our requirements for deoptimization. The optimized version is obtained by applying a transformation on a base function. Since OSR points cannot be tampered with, the mapping between the OSR exit and the base function is guaranteed to be preserved, regardless of the subsequent (valid) optimizations performed on the optimized function. Another way to see this is that the OSR exit is a call to the continuation function, which takes as arguments all the values needed at the continuation function to reconstruct the computation state and continue the execution. As long as the OSR exit call is preserved, and its arguments valid, the continuation function should be able to be executed properly. For the overall execution to be correct, additional constraints, such as preventing instructions with side-effects from crossing OSR points, are however still required. In the light of these observations, it makes sense to instrument the base function to generate a continuation function for the OSR exit.

The OSR Kit way of generating the continuation function is expensive. In the deoptimization case, inserting a resolved OSR exit requires to 1) clone the base function in order to obtain two copies: one is kept as is, the other will be optimized, 2) generate the continuation function. Since the continuation function might have a different signature than the base function, the framework has to generate a clone of the body of the base function that will be instrumented with the OSR ENTRY block. Generating two clones of the base function, in order to insert a resolved OSR exit, might be very expensive for big functions.

Finally, as it was the case for the open OSR, the resolved OSR does not take into account the fact that an OSR exit might be triggered at every call, once it has been fired a first time. Although less expensive than the open OSR, a resolved OSR exit that fires at every execution adds the costs of evaluating the OSR condition, and a function call and return, compared to the base unoptimized function. The OSR Kit framework does not provide the mechanisms required to avoid this extra cost.

4.1.3 The RJIT compiler

General

CITE GNUR
The RJIT project is an LLVM based JIT Compiler for the R programming language. It relies on GNUR, an interpreter for R. GNUR provides and AST interpreter, a bytecode compiler and a bytecode interpreter for R. The RJIT compiler takes as input a symbolic expression (SEXP), i.e., a special C structure generated by GNUR while parsing R programs. A SEXP represents R constructs and contains the AST representation of an R element. A SEXP has a type, and can be seen as a lisp-like cons cells, composed of two pointers, CAR and CDR. A SEXP has many other elements, that we do not detail here. Depending on its type, each SEXP element corresponds to a specific attribute (see examples for closures and functions below). The GNUR interpreter has been instrumented to call the JIT compiler before evaluating a non-compiled SEXP. The RJIT compiler generates native code, using the LLVM framework, that corresponds to the R function's AST it received as argument. The native code is then wrapped into the appropriated SEXP constructs, and returned to the GNUR interpreter. Afterwards, the

GNUR interpreter executes the native code.

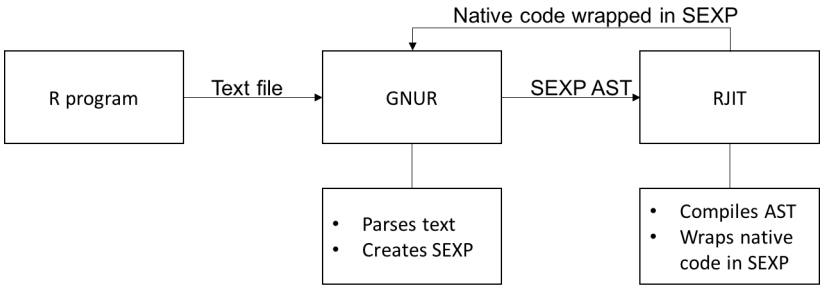


FIGURE 4.1: RJIT compilation flow.

The RJIT compilation

RJIT walks the SEXP ast and generates the corresponding LLVM IR. Since R is a dynamic language, most R constructs need to be simulated by LLVM calls to special functions, called R intrinsics. Intrinsic functions provide basic functionalities, such as resolving a closure from a symbol and the current environment, getting a value associated with a symbol, setting a local variable, creating a promise, or performing generic arithmetic operations. Figure REF provides R code performing some of these standard operations, and the equivalent, simplified, LLVM IR generated for this code, with the corresponding intrinsic calls.

```
1  f <- function(a) {
2   a <- 2
3   w <- a+2
4   g(w*4)
5  }
6
7  define  @f(%consts, %rho, i32 %useCache) #0 gc "rjit" {
8  start:
9    %0 = call  @userLiteral( %consts, i32 1)
10   call void @genericSetVar( %0,  %rho,  %consts, i32 2)
11   %a = call  @genericGetVar( %rho,  %consts, i32 2)
12   %1 = call  @userLiteral( %consts, i32 3)
13   %2 = call  @genericAdd( %a,  %1,  %rho,  %consts, i32 4)
14   call void @genericSetVar( %2,  %rho,  %consts, i32 5)
15   %g = call  @getFunction( %rho,  %consts, i32 6)
16   %3 = call  @userLiteral( %consts, i32 7)
17   %4 = call  @constant( %consts, i32 8)
18   %5 = call  @icStub_1( %3, %4, %g, %rho, (@f, i64 0)
19   ret  %5
20 }
```

R resolves callee of function calls at runtime, when the call is executed. In order to simulate this behavior, the RJIT compiler needs to rely on a special instrumentation to compile calls to functions, especially the ones to functions that are not yet defined. To that end, the JIT compiler replaces function calls with inlined cached stubs (i.e., ic stubs). An ic stub takes as parameters the function call's arguments, a pointer to the callee extracted from the environment, and a pointer to the caller. The ic stub, when executed, triggers the compilation of a wrapper function, call it the ic callee, that caches

the resolved callee. This wrapper function tests that the callee pointer that it gets as argument points to the same function as the one used during its own compilation. If that is the case, it performs the call to the callee, and propagates the value returned after the call. If the callee pointer references another function, it performs an ic stub call. Once the ic callee function is compiled, the ic stub replaces itself in the caller with a call to the ic callee. In order to do so, the ic stub relies on LLVM intrinsics called Stackmaps[13] and Patchpoints[12], and on the caller's pointer it received as argument.

The compilation flow can be reduced to the following steps:

1. The function's AST is extracted from the SEXP.
2. The JIT compiler traverses the AST and produces the corresponding LLVM IR.
   - Calls to non-intrinsic functions are compiled as ic stubs.
   - A special instrumentation is set at every call instruction as an LLVM attribute[11].
3. When the *jitAll* function is called on the compiler instance, every call instruction in every function in the module is visited.
   - Safepoints are inserted at every call and used by the garbage collector during the execution.
   - Patchpoints[12] are inserted at ic stubs.
4. The native code corresponding to each function is generated and inserted into the corresponding SEXPs.

The output of the compiler at step 3 is a blotted piece of LLVM IR from which R semantics are hard to extract. Figure 4.2 defines a very simple R function. Figure 4.3 provides the non-instrumented RJIT LLVM IR. Figure 4.4 provides the RJIT LLVM IR instrumented with safepoints.

```
1 f <- function(a) a+1
```

FIGURE 4.2: R simple function.

```
1 define %struct.SEXPREC addrspace(1)* @rfunction(%struct.SEXPREC addrspace(1)*
      %consts, %struct.SEXPREC addrspace(1)* %rho, i32 %useCache) #0 gc "rjit"
      {
2 start:
3   %a = call %struct.SEXPREC addrspace(1)* @genericGetVar(%struct.SEXPREC
      addrspace(1)* %rho, %struct.SEXPREC addrspace(1)* %consts, i32 1) #1, !
      r_ir_type !0
4   %0 = call %struct.SEXPREC addrspace(1)* @userLiteral(%struct.SEXPREC
      addrspace(1)* %consts, i32 2) #1, !r_ir_type !1
5   %1 = call %struct.SEXPREC addrspace(1)* @genericAdd(%struct.SEXPREC
      addrspace(1)* %a, %struct.SEXPREC addrspace(1)* %0, %struct.SEXPREC
      addrspace(1)* %rho, %struct.SEXPREC addrspace(1)* %consts, i32 0) #1, !
      r_ir_type !2
6   ret %struct.SEXPREC addrspace(1)* %1, !r_ir_type !3
7 }
```

FIGURE 4.3: Non-instrumented RJIT LLVM IR.

```
1  define %struct.SEXPREC addrspace(1)* @rfunction(%struct.SEXPREC addrspace(1)* %consts, %struct.SEXPREC addrspace(1)* %rho,
       i32 %useCache) #0 gc "rjit" {
2  start:
3    %safepoint_token = tail call i32 (i64, i32, %struct.SEXPREC addrspace(1)* (%struct.SEXPREC addrspace(1)*, %struct.SEXPREC
         addrspace(1)*, i32)*, i32, i32, ...) @llvm.experimental.gc.statepoint.p0f_p1struct.SEXPRECp1struct.SEXPRECp1struct.
         SEXPRECi32f(i64 4, i32 0, %struct.SEXPREC addrspace(1)* (%struct.SEXPREC addrspace(1)*, %struct.SEXPREC addrspace(1)*,
         i32)* @genericGetVar, i32 3, i32 0, %struct.SEXPREC addrspace(1)* %rho, %struct.SEXPREC addrspace(1)* %consts, i32 1,
         i32 0, i32 2, %struct.SEXPREC addrspace(1)* %consts, %struct.SEXPREC addrspace(1)* %rho)
4    %a.1 = call %struct.SEXPREC addrspace(1)* @llvm.experimental.gc.result.p1struct.SEXPREC(i32 %safepoint_token)
5    %safepoint_token.2 = tail call i32 (i64, i32, %struct.SEXPREC addrspace(1)* (%struct.SEXPREC addrspace(1)*, i32)*, i32, i32
         , ...) @llvm.experimental.gc.statepoint.p0f_p1struct.SEXPRECp1struct.SEXPRECi32f(i64 5, i32 0, %struct.SEXPREC
         addrspace(1)* (%struct.SEXPREC addrspace(1)*, i32)* @userLiteral, i32 2, i32 0, %struct.SEXPREC addrspace(1)* %consts,
         i32 2, i32 0, i32 3, %struct.SEXPREC addrspace(1)* %a.1, %struct.SEXPREC addrspace(1)* %consts, %struct.SEXPREC
         addrspace(1)* %rho)
6    %0 = call %struct.SEXPREC addrspace(1)* @llvm.experimental.gc.result.p1struct.SEXPREC(i32 %safepoint_token.2)
7    %safepoint_token.3 = tail call i32 (i64, i32, %struct.SEXPREC addrspace(1)* (%struct.SEXPREC addrspace(1)*, %struct.SEXPREC
          addrspace(1)*, %struct.SEXPREC addrspace(1)*, %struct.SEXPREC addrspace(1)*, i32)*, i32, i32, ...) @llvm.experimental.
         gc.statepoint.p0f_p1struct.SEXPRECp1struct.SEXPRECp1struct.SEXPRECp1struct.SEXPRECp1struct.SEXPRECi32f(i64 6, i32 0, %
         struct.SEXPREC addrspace(1)* (%struct.SEXPREC addrspace(1)*, %struct.SEXPREC addrspace(1)*, %struct.SEXPREC addrspace
         (1)*, %struct.SEXPREC addrspace(1)*, i32)* @genericAdd, i32 5, i32 0, %struct.SEXPREC addrspace(1)* %a.1, %struct.
         SEXPREC addrspace(1)* %0, %struct.SEXPREC addrspace(1)* %rho, %struct.SEXPREC addrspace(1)* %consts, i32 0, i32 0, i32
         4, %struct.SEXPREC addrspace(1)* %0, %struct.SEXPREC addrspace(1)* %a.1, %struct.SEXPREC addrspace(1)* %consts, %struct
         .SEXPREC addrspace(1)* %rho)
8    %1 = call %struct.SEXPREC addrspace(1)* @llvm.experimental.gc.result.p1struct.SEXPREC(i32 %safepoint_token.3)
9    ret %struct.SEXPREC addrspace(1)* %1, !r_ir_type !0
10 }
```

FIGURE 4.4: Instrumented RJIT LLVM IR.

The function's SEXP

In GNUR and RJIT, a function is represented by a special SEXP. This SEXP has a NATIVESXP type when the function is compiled by the RJIT compiler, and a LANGSXP, i.e., a language construct, when it is not. In the case of a compiled function, the CDR element contains a pointer to the native code of the function. In the case of a non-compiled function, the CDR contains the AST of the function. The CAR element, in both cases, contains the constant pool associated to the function, i.e., the list of symbols defined inside the function. For a compiled function, the TAG element of the SEXP contains the LLVM IR of the function.

When the GNUR interpreter evaluates a function SEXP, it checks its SEXP type. If the function is compiled, it performs the call to the native code. If not, it calls the RJIT compiler, and executes the resulting native code.

In RJIT, the LLVM IR for a function always has a signature of type T:

$$T : (\text{SEXP}, \text{SEXP}, i32) \rightarrow \text{SEXP}.$$

where the first SEXP element is a pointer to the constant pool associated with the function, and the second SEXP a pointer to the environment. As shown in Figure REF, the constant pool (CP) is used to access variable symbols. For example, line 11 corresponds to a call to the *genericGetVar* intrinsic, that enables to retrieve a local variable's value from the environment. The symbol corresponding to the variable is extracted from the CP, at index 2, and used along side the environment (i.e., *rho*), to access the corresponding value. In general, the environment is used to access variable values as well as function definitions (e.g., line 15).

The closure's SEXP

In R, a closure contains a function and the environment associated to it. A closure SEXP has type CLOSXP. The BODY of a SEXP closure is a function SEXP. The TAG of a SEXP closure is an environment SEXP, of type ENVSXP. A closure also contains the formals, i.e., the arguments symbols of the function. The getFunction intrinsic, that enables to resolve a function, returns a closure. The environment is made of cons cells, and is essential to the execution of a function. Local variables and function argument values are set inside the environment.

4.2 OSR Handler

This section presents the OSR Handler, a special singleton implemented in RJIT that strives to enable efficient OSR deoptimizations. The OSR Handler has two main goals: 1) to mitigate the limitations of the OSR Kit[6] library exposed in 4.1.2, 2) to adapt the OSR Kit library to the RJIT framework.

This section proceeds as follow: first, it exposes the additional challenges that are inheritant to the use of the OSR Kit library in RJIT. Then, it presents the OSR Handler

implementation, and the solutions adopted for each problem encountered while enabling OSR deoptimization in RJIT.

4.2.1 Additional challenges in RJIT

The RJIT LLVM IR uncovers additional challenges in the use of OSR Kit[6] for the deoptimization case. RJIT LLVM IR specificities, such as the instrumentation for the garbage collector and the ic stub calls, require specially care when used along side the OSR library.

In order to perform transformations based on R semantics, a fresh non-instrumented (i.e., without the Safepoint and Patchpoint machinery) version of the LLVM IR is needed. The non-instrumented IR corresponds to the compiler's output, before calling the *jitAll* function. For functions that were never compiled, invoking the compiler and extracting the non-instrumented IR seems like a reasonable solution. However, for functions that were fully compiled previously, i.e., functions that have an instrumented LLVM IR, re-invoking the compiler in order to obtain a non-instrumented IR is not a satisfactory solution. We therefore have to find a way to avoid re-compiling functions unnecessarily.

Simply cloning the LLVM IR is not enough in RJIT. In order to fully compile and execute a clone, its LLVM attributes need to be set at each of its call instructions to allow the Safepoint instrumentation. This is mandatory and, if not done properly, leads to failures during the execution. Another problem with clones arises from the targets of function calls. In LLVM, a callee has to be an LLVM function declared in the same module. Therefore, compiling a clone of a function compiled in a different module requires to ensure that every callee is defined in the current compilation module. One last thing to take care of are the ic stubs calls. An ic stub takes as argument a pointer to its caller. When a function containing ic stubs is cloned, the caller pointer arguments point to the original function. This needs to be fixed if the clone is to be fully compiled and executed. Section 4.1.2 states that one drawback in the OSR Kit implementation comes from the number of clones that have to be generated to insert a resolved OSR point. In RJIT, the cost of fixing the LLVM IR, as explained in this paragraph, has to be added to the cost of cloning.

OSR Kit continuation functions are not compatible with ic stubs. The ic stubs expect to receive pointers to their callers as argument. In RJIT, every function is supposed to have the following type signature:

$$T : (\text{SEXP}, \text{SEXP}, i32) \rightarrow \text{SEXP}.$$

According to Section 3.3.4, the OSR Kit[6] library modifies the continuation function's signature in order to pass all the live values during an OSR transition. As a result, the continuation function might not be of type T. Enabling any type in the ic stub is not a viable solution, i.e., it requires too much work and goes against the type policies enforced by the RJIT framework. We therefore have to come up with an alternate solution.

The next sections present the solutions implemented in RJIT to mitigate the OSR Kit limitations in the deoptimization case, while answering the above challenges particular

to the RJIT framework.

4.2.2 Reducing the number of compilation

TALK ABOUT GARBAGE COLLECTION!!!

Transformations that act on R semantics have to be performed on a non-instrumented LLVM IR. As explained before, obtaining such IR might be harder than expected. There are three cases to distinguish:

1. The function was never compiled before.
2. The function was compiled, but not instrumented.
3. The function was compiled and instrumented.

For case 1, the only option is to compile the function, and extract the generated IR before the final instrumentation. Case 2 is the best scenario. The LLVM IR can simply be extracted from the SEXP function, cloned, and used for the transformations. This case happens when the function was compiled during the current compilation unit, i.e., the function is part of the current module and was generated using the current instance of the compiler, on which the *jitAll* function was not called yet. Case 3 is the worst scenario. The non-instrumented LLVM IR no longer exists. The naive solution is to recompile the function from scratch, which seems very expensive.

The OSR Handler enables to record non-instrumented IRs that it encounters. Non-instrumented IRs are registered in the *base version* map, a map from a SEXP closure to a NATIVESXP function SEXP. This singleton provides a special function, called *getFreshIR*, that takes as parameter a SEXP closure and a compiler instance, and returns a function SEXP (see signature Figure REF). The OSR Handler extracts the function from the closure, and checks its type. Depending on the type of the function and the content of the base version map, it selects the less costly solution to get the non-instrumented IR.

```
1  /**
2   * @brief      Returns a non-instrumented version
3   *         of the function contained in the closure.
4   *
5   *
6   * @param[in]  closure  SEXP of type CLOSXP.
7   * @param      c        Current compiler instance pointer.
8   *
9   * @return     A function SEXP containing uninstrumented LLVM IR.
10  */
11 static SEXP getFreshIR(SEXP closure, rjit::Compiler* c);
```

FIGURE 4.5: The getFreshIR prototype.

The OSR Handler *getFreshIR* function compiles and registers non-NATIVESXP functions (case 1). The OSR Handler invokes the compiler on the function, and creates two

clones of the resulting non-instrumented IR. Both clones are not part of the current compilation module, i.e., their IRs are not instrumented when the jitAll function is called. One clone is wrapped inside a copy of the function SEXP and stored in the base version map. The other clone is also wrapped inside a copy of the function SEXP and returned to the getFreshIR caller.

Calling *getFreshIR* on a function that was not previously compiled can be viewed as an ahead of time (AOT) compilation strategy. RJIT is a JIT compiler, i.e., functions are compiled just before being executed. Using the *getFreshIR* can force the function's compilation to happen at anytime. As a small optimization, the OSR Handler sets the result of the compilation inside the corresponding closure. By doing so, we ensure that a subsequent call to this function will already benefit from the compiled version, and will not trigger a useless call to the JIT.

The *getFreshIR* strives to avoid recompiling NATIVESXP. When the closure passed as argument to the *getFreshIR* call has an entry in the base version map, a clone of the stored function SEXP is returned. Cloning the function SEXP requires to clone the LLVM IR, and insert it in a new function SEXP with the same constant pool. When the closure passed as argument does not have an entry in the base version map, the OSR Handler must determine if the IR is instrumented or not. RJIT creates one module per compiler instance, and performs the *jitAll* call just before returning to the interpreter. As a result, if an LLVM IR's module is the same as the current compiler's module, the LLVM IR is not instrumented. In this case, the getFreshIR creates an entry for this closure in the base versions and returns the proper clone to the user, as explained before. For an LLVM IR with a different module, that does not have an entry in the base version map, the OSR Handler has no order choice than to recompile the function. In this case, it also creates a proper entry in the base version map and returns a copy to the user. This situation can be avoided by making sure that for every function compiled, a proper entry is registered in the OSR Handler's base version map.

```
1 typedef std::pair<SEXP, Function*> BaseEntry;
2 static std::map<SEXP, BaseEntry> baseVerions;
```

FIGURE 4.6: The base version map.

The non-instrumented IR obtained through the *getFreshIR* corresponds to the result of a compilation of the original AST of the function, i.e., no transformation has been applied on this IR. In other words, all the transformations performed while compiling the function the first time are not reflected on the stored IR. This is a choice that we made in order to keep different optimization passes independent. In some cases, that solution implies that we have to re-do the same transformation on the same function several times. For example, consider a function A that calls B several times. An inliner would inline every call sites in A. The inliner could further decide to inline all the calls inside B's body. In that case, if the inliner relies on the OSR Handler to get B's IR, it will have to run on each separate clone to inline the call sites they contain, therefore performing the same transformations multiple times. On the other hand, if the inliner's transformations depend on the location of the call to B, it is guaranteed to obtain fresh and independent IRs through the OSR Handler.

```
1  /**
2   * @brief      Adds a function SEXP to the current compilation module.
3   *
4   * @param[in]  f     The SEXP of type NATIVESXP to add to the module.
5   * @param      m     The LLVM module.
6   */
7  static void addSexpToModule(SEXP f, Module *m);
8
9  /**
10  * @brief      Corrects the LLVM IR of a function so that it compiles in
11  * the current compilation module.
12  *
13  * @param[in]  func  A function SEXP of type NATIVESXP.
14  * @param      c     The current instance of the compiler.
15  *
16  * @return     The corrected SEXP.
17  */
18 static SEXP resetSafepoints(SEXP func, rjit::Compiler *c);
19
20 /**
21  * @brief      Schedules a function SEXP for relocation, i.e., enables
22  * to load the native code inside the SEXP CDR
23  * once the compilation is over.
24  *
25  * @param[in]  formals  Formals of the enclosing closure of the function.
26  * @param[in]  fun      Function SEXP of type NATIVESXP.
27  * @param      m        The current module of compilation.
28  */
29 static void addRelocations(SEXP formals, SEXP fun, rjit::JITModule *m);
```

FIGURE 4.7: Utility functions.

A cloned function that needs to be fully compiled has to be fixed. As explained in Section 4.2.1, the IR of a cloned function has incorrect attributes instrumentation, incorrect arguments to its ic stub calls, and might have missing targets in its call instructions, i.e., the callee has not been declared in the module. A cloned function also needs to be added to the compilation module in order to be fully compiled. The OSR Handler provides utilities functions to help the user fix the LLVM IR of cloned functions in RJIT. The *resetSafepoints* function takes care of adding the proper safepoints and patchpoints to the IR, as well as fixing the arguments of ic stubs. It further verifies that every callee in call instructions has been declared in the module. If that is not the case, it adds the proper function definition to the module. The OSR Handler provides another function, *addSexpToModule*, that enables to add a function to the current compilation module. Finally, the compilation engine requires to register an explicit mapping between an LLVM IR and a SEXP to properly link the generated native code in the function SEXP. The *addRelocation* function enables to register a mapping for a clone SEXP and the cloned function it contains. These utilities functions are divided so that each of them provides a single, fully contained functionality. It enables the user to have a fine-grained control over the LLVM IR clones, and only perform the desired operations.

4.2.3 Simplifying the OSR exit insertion

Section 4.1.2 explains our choice to use resolved OSR from the OSR Kit[6] library. The resolved OSR API is provided in Section 3.3.4.

RJIT imposes additional constraints on the OSR exits. The continuation instruction needs to be the exact match of the from instruction. In other words, considering the *insertResolvedOSR*, the *lPad* argument is the instruction in the continuation function that corresponds to the *src* argument in the from function. The OSR Kit library provides some flexibility in the choice of these two instructions. This flexibility is not required in RJIT, and is removed to ensure the correctness of the implementation.

The OSR Handler provides a special cloning function, called *getToInstrument*, that clones an LLVM function, adds it to the same module as the original function, fixes its IR, and automatically registers a StateMap between the original and the cloned function in its *statemaps*, a map from a pair of LLVM function pointers to a StateMap (see Figure REF).

```
1 typedef std::pair<Function* , Function*> FunctionPair;
2 static std::map<FunctionPair, StateMap> statemaps;
```

FIGURE 4.8: The statemaps.

The cloned function can be called to generate what is called, in RJIT, the *toInstrument* function. The *toInstrument* function is the argument passed as the continuation function to the *insertResolvedOSR* call. As explained in Section 4.1.2, the OSR Kit library will clone the *toInstrument* function to generate the continuation. It relies on a StateMap that is passed as argument to the function call. Thanks to the OSR Handler statemaps, this argument can be omitted and automatically retrieved.

The *toInstrument* function is added to the module, and will therefore be fully compiled upon the *jitAll* call. As explained in Section 4.2.1, the continuation function's ic stub calls cannot use the continuation function's pointer as argument for the caller. The *toInstrument* function's address is used instead, which in turns requires the toInstrument function to be part of the module and go through the entire compilation. The *toInstrument* function has another important role detailed in Section 4.2.4.

The OSR Handler provides a function, called *insertOSRExit*, that exhibits the simplified interface described in this section to insert OSR exits. The function's prototype is provided in Figure REF. Section 4.2.4 describes in details the role of the last argument.

4.2.4 Improving the exits

The continuation function mechanism does not fit the OSR exits well. As explained in Section 4.1.2, once an OSR exit is fired, it is likely to be fired in subsequent calls. Triggering an OSR exit has a cost that might not be negligible. The OSR Kit[6] library does not provide any mechanism to improve that special case. Furthermore, Section 4.2.1

```
1  typedef std::pair<Function *, Function *> OSRPair;
2  typedef std::vector<Instruction *> OSRCond;
3  typedef std::vector<Instruction *> InstVector;
4
5  /**
6   * @brief      Insert and OSR exit at the src instruction.
7   *
8   * @param      optimized     The from function.
9   * @param      toInstrument  The model for the continuation function.
10  * @param      src           The instruction at which we insert the exit.
11  * @param[in]  condition     The condition to fire the exit.
12  * @param[in]  compensation  Global compensation code to execute at OSR Entry
        .
13  *
14  * @return     The optimized function and the continuation created.
15  */
16 static OSRPair insertOSRExit(Function *optimized, Function *toInstrument,
17                              Instruction *src, OSRCond condition,
18                              InstVector compensation);
```

FIGURE 4.9: The insertOSRExit prototype.

explains that continuation functions do not have valid signatures for ic stubs.

The OSR Handler, via the *insertOSRExit* method, enables to add a special compensation code at the beginning of the OSR ENTRY block in the continuation function, to answer these limitations. The compensation code can be any valid vector of LLVM instructions. This can be used to avoid triggering the OSR exit repeatedly once it has been fired a first time. For example, the compensation vector can contain a special call to a C++ function, that takes as argument a unique identifier for the continuation function. The C++ function then uses this identifier to retrieve the corresponding toInstrument function. The toInstrument function is a valid, fully compiled version of the original function, that does not contain the invalidated optimization. The C++ function then replaces the incorrect optimized function by the toInstrument function in the closure. As a result, all subsequent call to the closure will execute a correct version of the function. This enables to avoid the cost of triggering an exit upon every call to the function.

The compensation code in the OSR Handler does not rely on the OSR Kit compensation code mechanism. The OSR Kit compensation code is used to fix the execution state, is associated to values transferred from one version to another, and is not meant to access the execution environment. Since it pertains to a different goal, we decided to implement our own solution. The instructions that enable to replace the function's version in the closure are left to the user. This provides some flexibility on how to decide whether or not to replace the current function.

Replacing the invalidated optimized version by the *toInstrument* function presents several advantages. First, since the *toInstrument* was used to generate the continuation function, we know that it is correct and does not contain the invalidated transformations. Second, when the continuation function's ic stubs are executed, the RJIT framework generates inlined cached version of the target function, and is supposed to replace the ic stubs by a call to this inlined cached functions. For that purpose, it uses

the ic stub argument that points to its caller. This argument, in the continuation function, points to the *toInstrument* function. As a result, when the continuation function's ic stub are executed, the ic stubs in the toInstrument function are replaced by inlined cached functions, which are faster than the ic stubs. This ensures that taking an OSR exit will have as little impact on the overall execution time as possible. Furthermore, this mechanism ensures that executing a function twice, while experiencing an OSR exit in the first execution, will still have good performances compared to executing the non-optimized version the same number of times.

4.2.5 Walkthrough a simple example

In this Section, we provide a full example of a possible use of the OSR Handler's mechanisms. It describes which functions are called, and how every element is used. This example enables to show how the OSR Handler abstracts away all the different challenges that we described before, and enables to focus on the transformations performed on the input function.

Assume a transformation process *TransP*, that speculatively inlines function calls inside the input function. *TransP* receives as input the closure to optimize, and returns a closure containing the optimized version of the code. Every function call inlined has a special OSR exit instrumentation to handle run time redefinition of the inlined functions.

TransP gets the LLVM IR corresponding to the input function by calling the OSR Handler *getFreshIR* function. Call this copy the *working copy*. It then lists all the function calls performed inside the IR. For each function call, sequentially, it gets a *toInstrument* copy of the working copy by calling the OSR Handler clone function. Each toInstrument is wrapped into its own SEXP, added to the module, has its IR corrected by the *reseSafepoints* function, and is added to the relocations for the final native code to SEXP mapping. It resolves the callee, and gets its LLVM IR via the OSR Handler *getFreshIR*. It then inlines the call in its working copy and calls the OSR Handler *insertOSRExit* function. It then calls the *resetSafepoints* on the working copy to fix the IR, and returns its enclosing closure. The working copy contains all the OSR exits needed for correctness, and all the continuation functions, and their corresponding *toInstrument* versions are generated and part of the current module.

4.3 Other prototypes & Future work

This Section describes solutions, design ideas, that were partially implemented but not fully explored during this master thesis. RJIT being a young project, it lacks some supporting tools, e.g., a profiler or a code analyser, that could uncover new possibilities for the OSR implementation. In the future, the development of the RJIT framework should enable to provide new incentives to implement more complex OSR-based speculative optimizations, or extend the OSR mechanisms.

4.3.1 Transitive StateMaps

For the moment, in RJIT, the continuation function for an OSR exit is an instrumented version of the function that was used to generate the optimized version. This is a requirement that results from the use, in OSR Kit[6], of StateMaps to insert the OSR instrumentation. The from and the contination functions need to have a StateMap that records one-to-one mappings between their instructions. In some cases, however, one might want to provide a different continuation function. OSR Kit does not forbid this use of the library, but does not provide tools to make it easier.

Consider the following chain of speculative optimizations performed on a base function A:

$$A \xrightarrow{1} B \xrightarrow{2} C \xrightarrow{3} D \xrightarrow{4} E \xrightarrow{5} F \rightarrow ...$$

Each letter corresponds to a new version of the function. Each version is obtain by performing a speculative optimization of the previous version. If an assumption fails, the continuation function will be an instrumented version of the base version on which the speculative optimization was performed. For example, if the current version is F, and the OSR exit that corresponds to arrow 3 is taken, the function will exit to a continuation function that corresponds to version C. One might want, instead, to OSR exit to A, or to generate on the fly a new continuation function that underwent all the transformations, except number 3. This might not always be possible, and requires to understand all the interactions between the different optimizations. However, if all the transformation are independent, i.e., if none of them impacts on the others, this should be feasible. We should be able to find a corresponding continuation instruction in any other version.

As a first step towards a more complex mapping between the from and the continuation function, we implemented a transitive StateMap constructor. Suppose two fully generated StateMaps, S_1 and S_2, such that S_1 is a mapping between A and B, and S_2 a mapping between A and C. Our transitive constructor enables to generate a new StateMap, S_3, that maps B and C.

The transitive constructor can be extended to create StateMaps between different functions, generated by applying an arbitrary number of transformations on the same base function. Consider the above chain of transformations. The transitive StateMap constructor can be used to generate a transitive map between every pair of versions in the chain. The resulting StateMap might not be complete, but it ensures that every instruction that is present in the base version A and both versions that we want to map, e.g., C and F, will be present in the resulting StateMap.

If we further extend the framework to automatically update the StateMap while modifying an LLVM IR, we might even be able to uncover new possible mappings between different versions, and lift the restriction on the transitive StateMaps, i.e., we could generate mappings between instructions, even if they do not appear in A and in both versions C and F. This solution would be similar to the VARMAP in the Jikes RVM OSR implementation[29]. According to discussion we had with the OSR Kit designers, D'Elia and Demetrescu[6] are working on a similar idea on the OSR Kit build compensation branch[7]. The idea is to provide basic LLVM transformation functions, e.g, addInstruction, removeInstruction etc., that automatically update the StateMap.

4.3.2 Unguarded OSR points & lazy deoptimization

The OSR Kit[6] library only provides guarded OSR points. As explained in Section 2.2.1, unguarded OSR points correspond to portion of the code that can be overwritten, at runtime, based on external events detected by the runtime framework. In certain cases, unguarded OSR points enable to decrease the overhead introduced by the OSR mechanisms. For example, consider the unoptimized function in Figure 4.10a. In f, the variable x is deemed constant. The compiler would therefore like to speculatively replace the return instruction by the value of x, hence avoiding the cost of an environment lookup (Figure 4.10b). A guarded OSR exit does not allow any performance gain in that case. In fact, the guard, i.e. the OSR condition, must check that x is equal to 3, and hence cancels any benefit obtained from the constant propagation optimization. With an unguarded OSR point, however, the execution environment (here GNUR), is instrumented to detect reassignments to x. Assuming there are more reads then writes, and that the value for x is rarely modified, we can expect the unguarded point to have better performances than a guarded one.

```
1 x <- 3
2
3 f <- function() {
4   g()
5   return x
6 }
```

(A) Unoptimized function.

```
1 x <- 3
2
3 f <- function() {
4   g()
5   return 3
6 }
```

(B) Optimized Function.

FIGURE 4.10: Use case for unguarded OSR points.

In RJIT OSR, we were able to construct a prototype for unguarded OSR Exits and lazy deoptimization. The base assumption is that any function call might invalidate a speculative optimization. Each call instruction is therefore instrumented with a patchpoint**??** that enables to insert, at runtime, a call to the continuation function, whenever a speculative optimization is invalidated. We were able to re-use parts of the OSR Kit library to generate the continuation function and the OSR Exit call itself. Consider the example in Figure REF. The function g, re-assigns the x to a new value. GNUR is instrumented to detect re-assignments to symbols used during a speculative optimization. It therefore detects that x was re-assigned, and patches the reserved bytes following the call to g with a call to the OSR exit. When g returns to the invalidated f, the call to the continuation function is executed, hence triggering a lazy deoptimization of f.

This prototype was not inserted in the final version of the OSR Handler. Unguarded OSR points require to instrument the runtime environment, i.e., GNUR, to collect feedback on the current execution and detect assumptions failures. The types of checks performed inserted in GNUR depend on the optimizations performed. As explained previously, for the moment, RJIT does not perform any optimization. We therefore decided to avoid instrumenting GNUR before knowing which kinds of events need to be monitored.

4.3.3 On-the-fly compilation

Generating the OSR exit continuation on the fly is hard in general. Section 4.1.2 details the challenges that this technique presents. One option to simulate this behavior in RJIT is to save the LLVM IR of the base function in the OSR Handler, and remove it from the compilation unit. In other words, we generate and save the IR, but we do not complete its compilation. An open OSR is inserted in the optimized version. The f_{stub} function is then responsible for identifying the correct continuation instruction inside the saved base IR, and completing its compilation using profiled guided techniques.

On-the-fly compilation of continuation functions requires to guarantee that a transitive StateMap can be generated between the from function, and the newly generated function. RJIT is not mature enough to make this solution worth exploring. First, there is no profiler, and hence, no profiled-guided transformation. Second, the kind of transformations that RJIT would like to perform are not yet known. It is too early in the implementation process to identify which optimizations might improve the execution of the function. As a result, we did not put efforts in the implementation of this solution.

4.3.4 A cleaner, more integrated way of getting a fresh IR

ALSO SOLVES THE GC PROBLEM

The OSR Handler is a prototype, a proof of concept, to experiment OSR transitions in RJIT. As such, the choice was made, at the beginning of the implementation, to encapsulate everything related to the OSR deoptimization mechanism, e.g., the OSR Handler, the OSR Inliner (see Chapter 5), in their own classes. In other words, everything added to the RJIT project was kept as independent to the main RJIT elements as possible. This had the double benefit of 1) enforcing clear and stable interfaces that integrate well in the compilation flow, and 2) enabled to merge the master branch updates into the OSR branch easily. Since the RJIT project is under active development, the second point was essential. On the other hand, the OSR mechanisms could be greatly simplified, and made cleaner, if properly integrated inside the RJIT framework.

The OSR Handler, and more specifically the base versions map, could be, in the future, replaced by a careful instrumentation of the function's SEXP. For the moment, the TAG attribute of a compiled function's SEXP contains the LLVM IR used to generate the native function. If the function went through the entire compilation flow, the LLVM IR is instrumented. In RJIT, this TAG LLVM IR is mostly used for debugging purposes. In the future, a non-instrumented snapshot of the function's module could be stored instead. When a transformation needs to be performed on an already compiled function, the snapshot can be loaded inside the current compilation module, and the non-instrumented LLVM IR of the function extracted from it. This has the double benefit of easily providing an LLVM IR to work on, while removing any need to fix call instruction targets in the IR (see Section 4.2.1 for more details on this matter).

For the moment, many parts of the implementation rely on the fact that a function's SEXP TAG contains the LLVM IR of the native code. As a result, this would not be a trivial change and needs the approval of the entire team. However, after talking to the

main developers, the decision was made to make this change happen in the future.

Chapter 5

Case Study: An R Inliner

This Chapter presents experimental results obtained for the RJIT OSR. The first section describes an implementation of a speculative inliner tool, implemented inside the RJIT compiler, that relies on OSR exits to preserve the program's correctness. The second section presents benchmarking results obtained while evaluating the OSR Inliner, and RJIT OSR main features, i.e., the compensation code to replace invalidated functions, and the *getFreshIR* function.

5.1 A speculative inliner for RJIT

5.1.1 Justification

R is a dynamic language. Dynamic programming languages have the particularity to perform, at runtime, common programming behaviors executed during the compilation in static programming languages. Extending the program, adding code, extending objects and definitions, type setting or modifying the type system are such behaviors that, in dynamic programming languages, can take place at runtime. These particularities make certain common static compilation optimizations hard to implement in a dynamic language. One example of such optimization is inlining (i.e., inline expansion), that replaces a function call site with the callee's body.

Chapter 2 presents the on-stack replacement(OSR) mechanism. OSR techniques enable to implement *speculative* optimizations, i.e., to transform a function based on the assumption that the result of the transformation will be correct and improve the program's performance at runtime. If the assumption fails, the OSR mechanisms preserve correctness in the program. Thanks to this versatile tool, static programming languages optimizations can be performed speculatively in a compiler for a dynamic programming language.

In R, and more specifically in RJIT, inlining functions is hard. An R function is wrapped in a closure, and might be redefined at any time during the program's execution. As a result, the only viable way of allowing function inlining in RJIT is to rely on OSR mechanisms.

Inlining is an interesting optimization, and its impact on the code performance is hard to predict. It holds an important trade-off between time and space, i.e., how much improvement in terms of speed can be obtained, at the cost of some extra space. Furthermore, excessive inlining might increase register pressure and deplete the instruction cache, therefore decreasing the speed of the program's execution. Finally, inlining

can be viewed as a first step toward other optimizations. Copying a function's body at its call site enables to enlarge the scope considered by the compiler, and might therefore uncover new optimization opportunities.

Apart from being a viable OSR-based speculative optimization in RJIT, inlining allows to put some pressure on the OSR Handler's mechanisms. First, as explained in Chapter 4, the OSR mechanism requires to obtain fresh clones of functions bodies via the OSR Handler's functions. Second, in order to inline a function call, a copy of the callee's body needs to be obtained. This body, in RJIT OSR, can be obtained through the OSR Handler API. Function inlining might further force AOT compilation of the inlined functions, hence enabling us to test every aspect of the functionalities provided by the OSR Handler.

RJIT does not gather profiling information that enable to identify good candidates for inlining. This is, however, far from being a limitation. On the contrary, it is a perfect case study for OSR mechanisms in R. Although inlining function calls in R produces unsound compiled code, the OSR mechanism enables to ensure that the program is correct. As explained in Section 4.1.1, R performance bottlenecks are consequences of the language semantics. In the future, on-stack replacement is destined to enable to break R semantics, and to produce unsound, highly optimized code. Profiling will give insights on whether or not the program might benefit from such optimizations, but is not expected to be able to ensure that any of these transformations is sound. Not relying on a profiler and blindly inlining function calls, whenever it is possible, shows how much flexibility this OSR implementation truly brings to RJIT.

5.1.2 Challenges

An OSR-based speculative inliner for RJIT needs to be implemented at the LLVM IR level. This requires to identify function calls inside the LLVM IR, extract the callee's SEXP from the environment, and perform the appropriate transformations on the IR. While relatively simple conceptually, implementing such a mechanism presents several challenges in RJIT.

An R function call corresponds to several instructions in the RJIT LLVM IR. First, a call to the *getFunction* intrinsic is inserted. The call takes as parameter the AST of the call, stored in the constant pool associated to the function SEXP. This intrinsic enables to resolve, at runtime, the callee. To do so it explores the current environment. If no match is found, it accesses the enclosing environments until either reaching a definition or triggering an error letting the user know that the callee could not be found. The compiler then emits instructions to load arguments to the function call. Finally, an ic stub is generated. The ic stub takes as arguments the result of the call to the *getFunction* intrinsic, the arguments provided to the call, a pointer to the caller, the current constant pool, and the environment. When executed once, the ic stub replaces itself with a call to an inlined cached wrapper function dedicated to the callee. Figure 5.1 provides a simple R function that performs a function call, and the equivalent LLVM IR generated by RJIT. Line 5 resolves the callee closure from the environment, line 6 loads the argument to the call, and line 8 performs the call to the ic stub.

In the OSR inliner, function calls can be identified by looking for ic stubs. The OSR inliner has to run on non-instrumented LLVM IR. The ic stubs have not yet been replaced by inlined cached wrappers. Therefore, even though a function call is compiled into several instructions, ic stubs and their arguments are enough to obtain all the information corresponding to a function call.

```
1  > f <- function(a) g(a)
2
3  define SEXP @f(SEXP %consts, SEXP %rho, i32 %useCache) #0 gc "rjit" {
4  start:
5    %g = call SEXP @getFunction(SEXP %rho, SEXP %consts, i32 1)
6    %0 = call SEXP @userLiteral(SEXP %consts, i32 2)
7    %1 = call SEXP @constant(SEXP %consts, i32 0)
8    %2 = call SEXP @icStub_1(SEXP %0, SEXP %1, SEXP %g, SEXP %rho, SEXP (SEXP,
         SEXP, i32)* @f, i64 0)
9    ret SEXP %2
10 }
```

FIGURE 5.1: Example of RJIT LLVM IR for a function call.

The OSR inliner needs to obtain a clone of the callee's LLVM IR, in order to inline a function call. If the callee was never compiled before, the *getFreshIR* function will trigger an AOT compilation of the function. Since the OSR inliner heavily relies on fresh IR, one challenge is to try to minimize the amount of cloning required in order to perform the transformations.

Inlining in R requires to create a dedicated environment for the inlined function. Implementing a sound name mangler in R is hard. It requires to check that fresh names do not collide with parent or child environments. Furthermore, replacing arguments with their actual values is not semantically correct, due to the way promises are evaluated. For these reasons, creating a dedicated environment appears to be the only sound solution.

The OSR Kit mechanism requires to associate a *toInstrument* clone to every function transformed. The continuation function has to be a clone of the base function used to generate the optimized one. One challenge is to carefully implement the inlining in order to limit the number of *toInstrument* functions generated.

Finally, all the challenges related to the cloned and continuation functions mentioned in Section 4.2.1 need to be considered. Fixing the LLVM IR requires to go through every instruction in the cloned functions. As a result, in order to save compilation time, it is important to perform such operations only when they are truly needed.

5.1.3 Implementation

MORE MODES OF EXECUTION !!!!!

This section describes the implementation of the OSR inliner. The OSR inliner relies on a special C++ class, called *FunctionCall*, to extract function calls in the LLVM IR and easily access its elements. The OSR inliner also provides different modes of execution,

that enable more or less aggressive speculative inlining.

The FunctionCall class provides a static function, called *getFunctionCalls*, that takes an LLVM IR as input, and extracts the function calls it contains. For each ic stub call, the function creates an instance of the FunctionCall class. A FunctionCall object gives quick access to each element of the function call, i.e., the *getFunction* call instruction, the arguments to the call, and the additional elements of the ic stub call. The *getFunctionCalls* returns a list of FunctionCall instances. It is important to note that, for efficiency and better integration, the *getFunctionCalls*, or any similar function that needs to go through the LLVM IR instructions and match a specific pattern, will be able to rely on the pass & match mechanism being developed in RJIT. RJIT provides a special matcher mechanism, combined to the LLVM passes implementation, that enables to extract special patterns from the LLVM IR. One of its main goals is to reduce the number of iterations on the entire LLVM IR, by merging different passes. Unfortunately, this feature was not yet ready at the time at which the RJIT OSR project was implemented.

The OSR inliner implements is a very basic, aggressive, speculative inlining algorithm. RJIT does not have a profiler or a static analyzer yet. As a result, the OSR inliner cannot rely on any extra information to decide if a call should be inlined or not. The OSR inliner proposes different modes of execution, that enable more or less aggressive inlining techniques. The default mode of execution inlines all the calls inside the current function, as long as a body can be obtained for the callee. The OSR inliner does not inline recursive calls. A more agressive mode, called *all inline*, enables to recursively apply the OSR inliner on the body of the functions being inlined. Function calls that contain ellipsis or missing arguments are not inlined by the OSR Inliner.

The OSR inliner implements a very naive inlining algorithm. All functions calls are extracted from the outer function using the *getFunctionCalls* function. The OSR Inliner then traverses the list of function calls, and groups them by call targets. The callee's closure can be obtained by looking up the symbol of the function call in the outer function's environment. In the *all inline* mode, the OSR inliner generates a fully inlined LLVM IR for each different callee, i.e., it inlines all the calls inside each compiled callee. In the default mode, the OSR inliner simply calls the getFreshIR method to obtain the callee's LLVM IR. The OSR inliner creates the toInstrument clone of the outer function. The toInstrument is used to create the continuation function for each OSR exit inserted in the outer function. It is wrapped in a valid function SEXP, added to the module, and has a valid IR. Then, for each different callee, the OSR inliner proceeds as follows:

1. In the body of the callee, update accesses to the constant pool by changing the index to $\text{LENGTH}(\text{CP}_{\text{outer}} + \text{index})$.
2. Append the callee's constant pool to the outer function's one.
3. For each function call to the callee:
   (a) Obtain an unused clone of the callee.
   (b) Insert instructions to create a new environment that contains the correct arguments to values mappings, before the ic stub.
   (c) Replace all accesses to the environment, i.e., rho argument, in the callee's body by the newly created environment.

(d) Move the callee's body inside the outer function, after the instruction of step 3b.

(e) Replace all the callee's return instructions by forwarding the return value to a ϕ-node inserted after the callee's body.

(f) Remove the ic stub.

(g) Replace the ic stub in the transitive mapping between the toInstrument and the outer function by a mapping between the toInstrument corresponding ic stub and the ϕ-node created in step 3e.

(h) Call the insertOSRExit with the outer function as the from function, the toInstrument as the continuation function, instructions that compare the result of the *getFunction* call with the hard coded address of the callee's closure as the OSR condition, and the *constant* call as the transition point, i.e., the instruction that was directly above the ic stub in the original outer function.

4. Call the *resetSafePoints* function to correct the optimized function's IR.

5. Set the optimized function SEXP inside the closure.

6. Add the optimized function SEXP to the relocations.

7. Return the closure.

It is important to note that the OSR inliner forces the compilation of every callee. As a result, any function contained in the outer function, and inlined, will also be compiled. When in the *all inline* mode, they are also fully inlined. Any subsequent call to a callee will therefore benefit from either a fully compiled, or a fully inlined and compiled version of the function.

The continuation function for an OSR exit corresponds to the fully unoptimized base function. The toInstrument function is used to generate the continuation function and does not contain any inlined call. This enables to limit the number of toInstrument functions generated during the transformation of the outer function and avoid triggering OSR exits in cascade. This also illustrates that, even though it does not correspond to the behavior encouraged by the OSR Kit library, a few careful updates (step 3g) to the transitive map enable to preserve a mapping between a base function, and a working copy being modified. On the other hand, ideally, the continuation function should inline all calls that did not fail. This solution can be implemented with on-the-fly compilation of the continuation function, but is not provided in this prototype.

The OSR Inliner provides a compensation code generator mechanism. The compensation code is a simple call to a C++ function, that takes as argument an integer identifier. The identifier corresponds to a function SEXP containing the toInstrument version. The C++ function uses the identifier to retrieve the toInstrument function SEXP, and replaces the invalidated function in the closure with this SEXP.

CODE EXAMPLE OF INLINING, TOINSTRUMENT AND CONT.

5.2 Tests

5.2.1 GNUR RJIT vs. Inlining on Shootout benchmarks

INLINING = ENABLE OTHER OPT

This section presents an evaluation of the OSR Inliner, based on the Shootout benchmarks[35]. As explained in Section **??**, the OSR Inliner is a proof of concept, showing that the RJIT OSR enables to break R semantics at a function's level while preserving the overall correctness of the program. In this section, we compare the all inline mode, i.e., the agressive OSR Inliner, against the regular mode, i.e., RJIT without any inlining. We first report some general numbers and observations gather in the all inline mode. We then compare the execution time obtained in both modes. Finally, we describe some experiments performed on the OSR Exits.

We instrumented the RJIT compiler to gather interesting information about the OSR Inliner's execution on the Shootout benchmarks. A first observation is that none of the benchmarks triggered an OSR Exit. This is a surprising result, since the OSR Inliner relied on no profiler, and blindly inlined function calls whenever it could. We therefore hope that, in most R programs, the redefinition of functions might be rare enough to consider very agressive speculative inlining. We note, however, that this might be a specificity of the benchmark suite, and recommend to inspect a broader class of R programs before making any assertion. OSR Exits are studied in more details later in this section, and in Section REF.

The Shootout benchmarks present very heterogenous results in terms of number of calls inlined. On all versions of the *revercomplement*, *spectralnorm* and *mandelbrot* benchmarks, only one call was inlined, namely, the benchmark's most important function was inlined in the function serving as entry point for the program. These benchmarks all contain one function, implementing the benchmark's algorithm. All calls correspond to built-in functions, which are either library functions implemented in C, or functions with a lot of arguments and default values. Since the OSR Inliner does not support default arguments, these calls could not be considered for inlining. On the other hand, other benchmarks present more interesting results. For example, the *pidigit.r* benchmark's execution led to 36 calls inlined (out of 237 calls, counting calls to built-ins functions), which corresponds to a total of 9747 lines inlined. In this benchmark, the OSR Handler enable to provide fresh LLVM IR, without going through the AST to LLVM IR translation, in 10 separate compilation modules. It is also important to note that the JIT compiler is involved less times, in the all inline mode, than on the regular mode. In fact, OSR Inliner performs AOT compilation of callees, and set the result inside the corresponding closure.

The use of the OSR Handler base version map, for the Shootout benchmarks, yielded a disappointing result. All the benchmarks, except *pidigit.r*, did not re-use LLVM IR accros different compilation modules. This, however, has several explanations. First, the OSR Handler base version map is intended to avoid re-translating ASTs to LLVM IR, for already compiled functions. A fresh IR is needed when a function is re-compiled, or, in the case of the OSR Inliner, when a function call is inlined. RJIT, for the moment, does not collect any profiling information on the program's execution and does

not perform any optimization. Thus, no re-compilation is triggered for unmodified, already compiled, functions. Another explanation is that the OSR Inliner mainly inlines user-defined functions, i.e., the functions defined inside the benchmark (as opposed to built-ins functions). When the OSR Inliner runs on the benchmark's entry point, it recursively inlines calls, hence reaching all the user-defined functions and inlining them whenever it is possible. Subsequent JIT compilations correspond to calls that the OSR Inliner could not inline previously, mainly library functions, that do not call any of the function's cached in the base version map. As a result, in order to evaluate the efficiency of the base version map, another form of experiment is required. Section 5.2.3 focuses on the evaluation of the getFreshIR function and the OSR Handler's mechanisms.

In a regular execution, the OSR Inliner's performance is close to RJIT execution time. On average, over the entire Shootout benchmarks, in the all inline mode, the execution time is equal to 1.033 times the regular RJIT execution time. The slowest execution time registered was for the *"spectralnorm-alt.r"*, that equals on average 1.21 times the equivalent RJIT execution time. We suspect that the overhead introduced by registering fresh IRs, and the OSR Inliner's transformations performed on the IR, are the main reasons for this small slow down. In fact, benchmarks that executed faster in the all inline mode than in the regular are the ones cited above, where only one call was inlined. Moreover, benchmarks that present numerous calls inlined, e.g., *pidigit.r*, have ratios closer to 1.10. As a result, we suppose that the overhead of blindly inlining function calls is superior to the possible gain in speed. We perform a second experiment where the entire compilation overhead is removed.

Our second experiment measures the average execution time per benchmark, while removing the compilation overhead. For both modes, we execute the benchmark a first time, so that all functions are compiled. We then measure 100 complete executions in the all inline mode, and compare the results to the regular RJIT execution. Once again, the difference between the two modes of execution is negligible, i.e., the all inline mode's execution is, in average, equal to 1.004 times the regular RJIT execution. We notice a small decrease in the average ratio between the two modes of execution. As before, we look for benchmarks that performed better in the all inline mode than in the regular RJIT one. Once again, the benchmarks that are made of one long function seem to be faster in the all inline mode. But, we also notice that other benchmarks, such as *binarytrees-naive.r*, the *fasta*'s, and *nbody*'s, seems to perform better in the all inline mode. Unfortunately, this is not enough to affirm that the slow down observed in the previous experiment is entirely due to the overhead introduced by the OSR Inliner's transformations. As a result, we conclude that OSR Inliner's transformations cannot account for the entire slow down. Other factors, related to inlining (e.g., register pressure), are also responsible for the slow down.MEASURING CACHE MISSES, SEE MORE OF THEM IN INLINE, BUT NOT ENOUGH TO BUILD A MODEL.

The similarities in terms of execution time, between the two modes, might be explained by the way the OSR Inliner inlines calls. The OSR Inliner preserves the function call's logic by creating a dedicated environment for the inlined function. Creating the environment is one of the dominating costs of a function call. Implementing a more aggressive inliner, that removes the need to create a dedicated environment for the inlined function, would potentially lead to a greater difference, in terms of execution time, compared to the regular RJIT.

Forcing OSR Exits to be triggered, in the Shootout benchmarks, does not have a significant impact on the performances. We performed several experiments, where OSR exits were triggered arbitrarily. Since the OSR Inliner introduces a compensation code to replace an invalidated version by a correct one, any OSR Exit can be fired only once. Furthermore, since all OSR Exits exit to a fully non-optimized version of the function, only one OSR Exit per function is fired. As a result, we observed no significant change in the average ratio of execution times in both modes. We were, however, able to slow down the all inline mode by modifying the OSR Inliner such that cascading OSR Exits can happen.

benchmark	#clones	#toInstrument	#calls	#inlined calls	#inlined lines
binarytrees-2.r	20	2	118	2	166
binarytrees-naive.r	20	2	115	4	332
binarytrees.r	18	2	95	2	190
fannkuchredux-naive.r	7	1	5	1	26
fannkuchredux.r	9	1	7	1	26
fasta-2.r	22	2	33	4	556
fasta-3.r	22	2	40	4	728
fasta-naive.r	21	3	34	5	743
fasta-naive2.r	21	3	35	5	819
fasta.r	28	2	74	4	428
fastaredux-naive.r	21	3	41	5	1055
fastaredux.r	22	2	40	4	864
knucleotide-brute.r	21	3	40	3	488
knucleotide-brute2.r	21	3	41	3	527
knucleotide-brute3.r	21	3	40	3	517
knucleotide.r	29	3	51	3	490
mandelbrot-ascii.r	9	1	17	1	138
mandelbrot-naive-ascii.r	7	1	13	1	152
mandelbrot-naive.r	15	1	26	1	162
mandelbrot-noout-naive.r	9	1	17	1	124
mandelbrot-noout.r	9	1	16	1	133
mandelbrot.r	17	1	30	1	148
nbody-2.r	27	2	41	3	264
nbody-3.r	29	2	42	3	264
nbody-naive.r	21	2	49	3	560
nbody-naive2.r	23	2	37	3	370
nbody.r	25	2	39	3	274
pidigits.r	86	14	237	36	9747
regexdna.r	13	1	19	1	95
reversecomplement-2.r	15	1	17	1	64
reversecomplement-naive.r	15	1	18	1	74
reversecomplement.r	15	1	17	1	58
spectralnorm-alt.r	15	1	90	1	84
spectralnorm-alt2.r	15	1	90	1	88
spectralnorm-alt3.r	13	1	91	1	118
spectralnorm-alt4.r	13	1	88	1	79
spectralnorm-math.r	33	1	202	1	35
spectralnorm-naive.r	11	1	86	1	76
spectralnorm.r	11	1	85	1	72

TABLE 5.1: Statistics on OSR Inliner's execution on the Shootout benchmarks.

5.2.2 OSR Exit vs. Replacing the closure

The OSR Handler enables to add compensation code at the beginning of continuation functions, and replace the invalidated version with a correct one. This mechanism allows to avoid triggering the OSR exit every time the function is executed.

In simple examples, the OSR Exti cost can be neglected. For example, if the OSR condition corresponds to a single, not expensive, comparison, its cost is very small. On the other hand, if the condition needs to perform an expensive operation, such as a lookup in the environment, or if the OSR Exit is located inside a hot loop, executing an invalidated code might increase the program's execution time.

We construct a proof-of-concept micro-benchmark to illustrate the gain of performance allowed by the compensation code. There are three elements that influence the execution time: 1) The OSR condition, 2) the function call to the continuation function, 3) the execution of ic stubs inside the continuation function. The call to the continuation function is considered as constant. The two other elements, however, can be made as expensive as we want, very easily. It is also important to note that for OSR Exits contained in loops, without compensation code, these costs appear at each iteration.

Figure 5.2 presents the code for our micro-benchmark. We insert an OSR exit before each function call, with an OSR condition that retrieves the callee's closure from the environment and always triggers the exit. We execute the benchmark with and without compensation code to replace the invalidated function, and measure the entire execution time. All measures are performed once all compilations are done, and g has been executed once. We observe a 40% slow down in the code without compensation compared to the one with compensation. The micro-benchmark is a fairly simple and common piece of code. We can therefore be confident that the compensation code has the potential to improve performances in regular R programs.

```
1  f <- function(a) a+1
2
3  f_prime <- function(a) {
4    for (i in 1:1000) {
5      f(i*a)
6      f(i*a)
7    }
8  }
9
10 g <- function() f_prime(100)
11
12
13 #Force compilaton
14 g()
15
16 #Start chrono
17 g()
18 #End chrono
```

FIGURE 5.2: The micro-benchmark.

5.2.3 OSR Handler getFreshIR vs. Parsing the AST

The OSR Handler, via the *getFreshIR* and its utilities functions, enables to obtain a fresh, non-instrumented, and correct LLVM IR for a function. This is meant to avoid the translation step from ASTs to LLVM IR every time a fresh non-instrumented IR is needed, e.g., for optimizations. Obtaining a fresh IR via the OSR Handler, for an already compiled function, is supposed to be faster than walking the entire function's AST and re-generating the corresponding LLVM IR instructions. The OSR Handler, however, performs some expensive operations, with non-negligible costs, such as registering a copy of the function in the base version map during the function's first compilation, cloning the base version entry to obtain a fresh working copy, and fixing the LLVM IR of clones. As a result, we evaluate the OSR Handler's mechanisms performance, compared to a regular translation from ASTs to LLVM IR. To do so, we rely on the Shootout benchmarks[35], instrumented such that each function in the benchmarks is used.

Registering an entry in the base version map increases the cost of a function's first compilation. A first experiment therefore consists in comparing the cost of this first compilation via the OSR Handler, and via the regular RJIT mechanism. Unsurprisingly, the OSR Handler exhibits poorer performances. The reason is that it has to generate the LLVM IR corresponding to the function's AST, as well as cloning the obtained IR and registering the clone in the base version map, hence introducing an overhead of VALUE compared to the RJIT compilation. We evaluate this overhead by breaking down a function's compilation into three steps: 1) the translation (i.e., from GNUR AST to LLVM IR), 2) registration (i.e, adding an entry in the base version map), 3) the jitAll (i.e., inserting safepoints and generating the native code). We measure these values over a 1000 compilations of every function implemented in the Shootout Benchmarks. In average, the cost of registering a function's entry in the base version map is equal to 0.21 times the cost of the translation, with a standard deviation of 0.07. If we further compare this result to the overall cost of a complete compilation, i.e., the cost of the translation and jitAll, we obtain a ratio of 0.004, with a standard deviation of 0.002. From these numbers, we deduce that the cost of registering the entry is negligible compared to native code generation's cost. As a result, we conclude that the overhead introduced by the OSR Handler to populate the base version map is insignificant compared to the overall execution, and acceptable compared to the translation's cost.

The OSR Handler is supposed to improve the RJIT's performances in the presence of optimizations re-compilations. The expected execution flow is that functions a jitted, and executed. When a function becomes hot, i.e., when it has been executed several times, it becomes a candidate for re-compilation and optimizations. This corresponds to the real use case of the OSR Handler's mechanisms. As a result, we need to evaluate the cost of obtaining a fresh and correct IR from the OSR Handler, when the base version map already contains an entry for the function, and compare it to the cost of generating the IR from the AST. We therefore perform a second experiment, where the OSR Handler has a fully generated base version map. In this experiment, we further ensure that every call instruction inside the clones obtained via the *getFreshIR* function need to be fixed. This corresponds to the worst case scenario for the execution of the *resetSafepoints* function, and thus provides an upper bound on the performance evaluation.

Our second experiment yields encouraging results. A first observation is that the

time ratio between obtaining the IR via the OSR Handler and via RJIT translation is constant, regardless of the number of LLVM IR instructions generated, with a median value at **0.33**. In other words, it seems that obtaining a fresh and correct IR, for an arbitrarily sized function, via the OSR Handler, is in average at least **three** times faster than generating it from the AST. These results can further be improved if less instructions need to be fixed. As explained in Section 4.3.4, a future solution would be to save an entire module per function in order to avoid having to fix LLVM IR instructions. With this solution, the cost of obtaining a fresh non instrumented IR would include the cost of loading an LLVM module into the current compilation module. Since it is not part of our implementation, we do not provide any estimation for this cost.

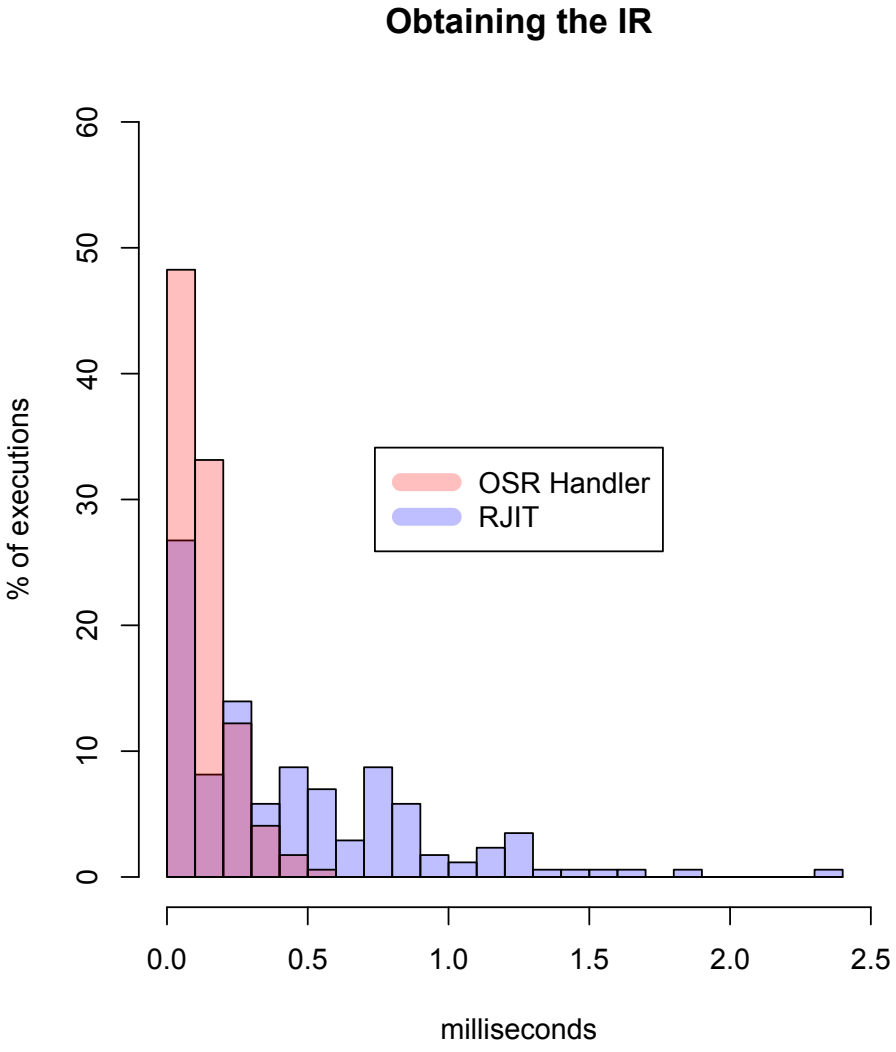


FIGURE 5.3: OSR Handler vs. RJIT: histogram of the time required to obtain a fresh and correct IR for all the functions in the Shootout benchmark.

Figure 5.3 presents an histogram corresponding to our second experiment. The histogram represents the distribution of times observed to obtain a fresh, non-instrumented,

and correct IR, via RJIT and via the OSR Handler, on the same set of functions. For the OSR Handler, all executions took less than 0.6 milliseconds, which represents one quarter of RJIT's time span. In RJIT, 30% of executions are still above 0.6 milliseconds. For the OSR Handler, 81.4% of the executions are under 0.2 milliseconds. For RJIT, only 34.9% took under 0.2 milliseconds.

In the light of these results, we conclude that avoiding to generate fresh IRs from ASTs for already compiled function yields better results, i.e., three times faster, regardless of the number of instructions contained in the function, and even when all call instructions are invalidated. We can further affirm that the overhead introduced to CONTINUE THIS

The OSR handler introduces a memory overhead that can be quantified. For each live and compiled function, the OSR Handler contains a function SEXP that points to the same constant pool, and contains a clone of the non-instrumented LLVM IR of the function. Since, for the moment, a garbage collection induces a flush of the OSR Handler's base version map, i.e., all entries are deleted, we can say that only live functions, compiled since the last garbage collection, have an entry in the OSR Handler. Flushing the entire base version map is too drastic. A more subtle implementation would flush only entries corresponding to collected closures. This requires, however, a more complex interaction with the garbage collector, that we did not have time to implement. Finally, the solution described in section 4.3.4 will exhibit similar costs, but will also be easier to integrate with the garbage collector.

Chapter 6

Conclusion

We presented RJIT OSR, a prototype for runtime deoptimization in RJIT, based on the OSR Kit library[6]. We adapted the library to better fit the deoptimization case, and answered challenges particular to the RJIT project. Re-using LLVM IR accros different different compilation modules proved to be three times more efficient than re-doing the AST to LLVM IR translation. Replacing an invalidated function by a correct closure, as part of the exit mechanism, mitigates the overhead introduced by the OSR instrumentation in the presence of assumption failures.

This report presents an overview of different OSR implementations, and common OSR mechanisms. In RJIT OSR, we implemented several prototypes to experiment with different techniques present in the literature. We believe that RJIT could greatly benefit from unguarded OSR points and lazy deoptimizations, once a good interaction with GNUR runtime environment is devised. We also think that providing transformation primitives that automatically update the values mapping between a base function, and an optimized one, is of great interest.

The implementation of a speculative inliner, relying on on-stack-replacement deoptimizations for correctness, enabled to validate our RJIT OSR prototype. Function call inlining is not semantically correct in R. However, thanks to RJIT OSR, we were able to aggressively inline calls throughout the Shootout benchmarks[35]. One of RJIT's goal is to aggressively and speculatively optimize away costly behaviors inherent to R semantics. On the other hand, our experience with inlining showed that implementing R-level transformations at the LLVM IR is challenging. Devising an intermediary representation, closer to R semantics, between GNUR ASTs, and the LLVM IR, that facilitates optimizations, and encoding OSR Exits at that level might be an interesting idea. LLVM would then only be part of the final compilation steps.

We presented interesting future work. RJIT will soon be able to gather profiling informations about the program at runtime. Once these information will become available, RJIT will need to enable runtime re-compilation and optimization of code. As a result, a better integration of OSR Handler's mechanisms, such as the fresh non-instrumented LLVM IR for each compiled function, might be needed. We mentioned attaching non-instrumented IR directly to SEXPs. Other on-stack replacement mechanisms, such as unguarded points and open OSR points, will also become more attractive.

Appendix A

Appendix Title Here

Write your Appendix content here.

Bibliography

[1] Andrew W. Appel. "SSA is functional programming". In: *SIGPLAN notices* 33.4 (1998), pp. 17–20.

[2] Matthew Arnold et al. "Adaptive optimization in the Jalapeno JVM". In: *ACM SIGPLAN Notices*. Vol. 35. 10. ACM. 2000, pp. 47–65.

[3] Craig Chambers. "The design and implementation of the self compiler, an optimizing compiler for object-oriented programming languages". PhD thesis. Stanford University, 1992.

[4] Craig Chambers and David Ungar. "Making pure object-oriented languages practical". In: *ACM SIGPLAN Notices*. Vol. 26. 11. ACM. 1991, pp. 1–15.

[5] Maxime Chevalier-Boisvert, Laurie Hendren, and Clark Verbrugge. "Optimizing MATLAB through just-in-time specialization". In: *Compiler Construction*. Springer. 2010, pp. 46–65.

[6] Daniele Cono D'Elia and Camil Demetrescu. "Flexible On-Stack Replacement in LLVM". In: *Proceedings of the 2016 International Symposium on Code Generation and Optimization (CGO 2016)*. 2016.

[7] Daniele Cono D'Elia and Camil Demetrescu. *OSRKit OSRKit*. 2016. URL: `https://github.com/dcdelia/tinyvm` (visited on 02/01/2016).

[8] Gilles Duboscq, Thomas Würthinger, and Hanspeter Mössenböck. "Speculation without regret: reducing deoptimization meta-data in the Graal compiler". In: *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java platform: Virtual machines, Languages, and Tools*. ACM. 2014, pp. 187–193.

[9] 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.

[10] Andreas Gal et al. "Trace-based just-in-time type specialization for dynamic languages". In: *ACM Sigplan Notices* 44.6 (2009), pp. 465–478.

[11] LLVM Developer Group. *LLVM Attribute LLVM::Attribute*. 2003. URL: `http://llvm.org/docs/doxygen/html/classllvm_1_1Attribute.html` (visited on 02/01/2016).

[12] LLVM Developer Group. *LLVM Patchpoints LLVM Patchpoints.* 2003. URL: `http://llvm.org/docs/StackMaps.html#llvm-experimental-patchpoint-intrinsic` (visited on 02/01/2016).

[13] LLVM Developer Group. *LLVM StackMap LLVM StackMap.* 2003. URL: `http://llvm.org/docs/StackMaps.html#llvm-experimental-stackmap-intrinsic` (visited on 02/01/2016).

[14] LLVM Developer Group. *LLVM web site LLVM web site*. 2003. URL: `http://llvm.org/` (visited on 02/01/2016).

[15] LLVM Developer Group. *ValueMapper.h LLVM ValueMapper.h*. 2003. URL: http://llvm.org/docs/doxygen/html/ValueMapper_8h.html (visited on 02/01/2016).

[16] 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.

[17] Urs Hölzle, Craig Chambers, and David Ungar. "Optimizing dynamically-typed object-oriented languages with polymorphic inline caches". In: *ECOOP'91 European Conference on Object-Oriented Programming*. Springer. 1991, pp. 21–38.

[18] Urs Hölzle and David Ungar. "A third-generation SELF implementation: reconciling responsiveness with performance". In: *ACM SIGPLAN Notices*. Vol. 29. 10. ACM. 1994, pp. 229–243.

[19] Apple Inc. *WebKit The WebKit Open Source Web Browser Engine*. 1998. URL: https://webkit.org/ (visited on 02/01/2015).

[20] 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.

[21] Chris Lattner and Vikram Adve. "LLVM: A compilation framework for lifelong program analysis & transformation". In: *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*. IEEE. 2004, pp. 75–86.

[22] McLAB. *The McVM Virtual Machine and its JIT Compiler The McVM Virtual Machine and its JIT Compiler*. 2012. URL: http://www.sable.mcgill.ca/mclab/mcvm_mcjit.html (visited on 02/01/2016).

[23] Floréal Morandat et al. "Evaluating the design of the R language". In: *ECOOP 2012–Object-oriented programming*. Springer, 2012, pp. 104–131.

[24] Oracle. *The Java HotSpot Performance Engine The Java HotSpot Performance Engine.* URL: http://www.oracle.com/technetwork/java/whitepaper-135217.html (visited on 02/01/2016).

[25] Jikes RVM Project Organization. *Jikes RVM Jikes RVM*. 2013. URL: www.jikesrvm.org (visited on 02/01/2016).

[26] 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.

[27] Filip Pizlo. *WebKit FTL WebKit FTL*. 2014. URL: https://webkit.org/blog/3362/introducing-the-webkit-ftl-jit/ (visited on 02/01/2016).

[28] David M. Smith. *Revolution Analytics: a 5-minutes history Revolution Analytics: a 5-minutes history.* URL: http://www.slideshare.net/RevolutionAnalytics/revolution-analytics-a-5minute-history (visited on 02/01/2016).

[29] 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.

[30] Toshio Suganuma, Toshiaki Yasue, and Toshio Nakatani. "A region-based compilation technique for dynamic compilers". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 28.1 (2006), pp. 134–174.

[31] Toshio Suganuma et al. "A dynamic optimization framework for a Java just-in-time compiler". In: *ACM SIGPLAN Notices*. Vol. 36. 11. ACM. 2001, pp. 180–195.

[32] R Core Team. *The R project The R project*. 1993. URL: `www.r-project.org` (visited on 02/01/2016).

[33] The Bioconductor Team. *Bioconductor The Bioconductor open source software for bioinformatics.* URL: `http://www.bioconductor.org/about/` (visited on 02/01/2016).

[34] The RJIT compiler team. *The RJIT compiler The RJIT compiler.* URL: `https://github.com/reactorlabs/rjit` (visited on 02/01/2016).

[35] Various. *The Computer Language Benchmarks GameThe Computer Language Benchmarks Game, a corpus of simple programs implemented in various languages and used to get a performance baseline.* URL: `http://benchmarksgame.alioth.debian.org/` (visited on 02/01/2016).