Reusing Contextually Specialized JIT Precompiled Units

A THESIS

Meetesh Kalpesh Mehta

In partial fulfillment of the requirements for the degree of

of

MASTER OF SCIENCE (BY RESEARCH)

in Computer Science and Engineering

Under the Supervision of

Dr. Manas Thakur

SCHOOL OF COMPUTING AND ELECTRICAL ENGINEERING (SCEE)

INDIAN INSTITUTE OF TECHNOLOGY (IIT) MANDI

June 2023

Reusing Contextually Specialized JIT Precompiled Units

A THESIS

Meetesh Kalpesh Mehta

In partial fulfillment of the requirements for the degree of

of

MASTER OF SCIENCE (BY RESEARCH)

in Computer Science and Engineering

Under the Supervision of

Dr. Manas Thakur

SCHOOL OF COMPUTING AND ELECTRICAL ENGINEERING (SCEE)

INDIAN INSTITUTE OF TECHNOLOGY (IIT) MANDI

June 2023

"Talent hits a target no one else can hit; Genius hits a target no one else can see."

–Arthur Schopenhauer

To CompL ...!

SCHOLAR DECLARATION

I hereby declare that the entire work embodied in this thesis entitled **Reusing Contextually Specialized JIT Precompiled Units** is the result of investigations carried out by me in the School of Computing and Electrical Engineering, Indian Institute of Technology Mandi, Mandi, Himachal Pradesh, India, under the supervision of **Dr. Manas Thakur,** Assistant Professor, School of Computing and Electrical Engineering, Indian Institute of Technology Mandi, Mandi, Himachal Pradesh, India.

I also declare that it has not been submitted elsewhere for the award of any degree or diploma. In keeping with general practice, due acknowledgments have been made wherever the work described is based on the findings of other investigators. Any omissions that might have occurred due to oversight or error in judgement are regretted.

Date: 08/07/2023
Place: IIT Mandi,
Himachal Pradesh, India

Mr. Meetesh Kalpesh Mehta
Enrollment No.: S20012
School of Computing and Electrical Engineering,
Indian Institute of Technology Mandi,
Mandi, Himachal Pradesh, India - 175075.

THESIS CERTIFICATE

It is certified that the entire work in this thesis entitled **"Reusing Contextually Specialized JIT Precompiled Units"** has been carried out by **Mr. Meetesh Kalpesh Mehta**, Enrollment No. S20012, under my supervision and guidance for the degree of Master of Science (by Research) in the School of Computing and Electrical Engineering (SCEE), Indian Institute of Technology Mandi, Mandi, Himachal Pradesh, India.

To the best of my knowledge and belief, present thesis completed by **Mr. Meetesh Kalpesh Mehta** fulfils the requirements of the M.S. (by Research) ordinance of the Indian Institute of Technology Mandi, Mandi, Himachal Pradesh, India. It contains original work of the candidate and no part of it has been submitted elsewhere for any degree or diploma.

Dr. Manas Thakur
Research Guide
Assistant Professor
School of Computing and Electrical Engineering,
Indian Institute of Technology Mandi,
Mandi, Himachal Pradesh, India - 175075.

Date:
Place: IIT Mandi,
Himachal Pradesh, India

Acknowledgments

I began my journey at IIT Mandi during the winter of 2020, amid the peak of the Covid-19 pandemic. After completing my B. Tech degree and passing the GATE examination, I was unsure about the direction I wanted to pursue. However, my interest in compilers, which was sparked by my final year project, drove me to seek out opportunities in this field.

Fortunately, I found an excellent mentor, Dr. Manas Thakur, who has been an invaluable guide and true friend throughout my journey. Working alongside him has allowed me to learn from the best in the world and develop my skills in this fascinating area of computer science. Collaborating with the entire Ř team, including Dr. Jan Vitek, Dr. Olivier Flückiger, Sebastián Krynski, Hugo Gualandi, and Jan Jecman, was a fulfilling and enjoyable experience. Working alongside such a rigorous research team has taught me invaluable lessons and allowed me to enhance my knowledge and skills.

I would like to extend my sincere thanks to Dr. Samar Agnihotri, Dr. Varunkumar Jayapaul, Dr. Sriram Kailasam, Dr. Pradeep Kumar, And Dr. Jinesh Machchhar, members of my annual progress committee for their time, and insights in reviewing my work and providing constructive comments and suggestions from the beginning.

I would like to extend my sincerest gratitude to Aditya, Arjun, and Prakash, my esteemed colleagues and fellow researchers, for their unwavering support and invaluable contributions throughout my research. The time spent with CompL (Compilers and Programming Languages), is definitely the highlight of this journey, from the long Arnehar walks to the exciting outings and the paper presentations, everything put together is the perfect balance for an enjoyable and rewarding research life.

Last but not least, I would like to express my gratitude to my parents, my brother (who is happily married now :)), and all the friends I have made throughout this journey, Dr. Saurabh, Amarjit, Sanket, and Govind.

Meetesh Kalpesh Mehta

Abstract

Just-in-time compilation has become an indispensable tool for speeding up programs written in dynamic languages. In these languages, offline static analysis efforts often prove futile due to reflection, possibility of side effects, and lack of types. JIT compilers can overcome these challenges by taking advantage of run-time feedback to perform context-sensitive speculative optimizations. However, these optimizations come at the cost of expensive compilation at run-time, which may lead to high warmup times and unexpected slowdowns due to deoptimization or late-stage compilations.

In this thesis, we present a novel approach to preserve context-sensitive JIT compiled units in an analyzable and reusable format. These compiled units can be reused across different runs of the same program or even different programs. Our strategy records multiple versions for each function, compiled under different contexts, and features a dispatching mechanism to pick the most appropriate of these versions at run-time. To improve the efficiency of this dispatching, we split our contexts into a two-level hierarchy. We also employ an offline pass that removes redundant code versions and identifies which parts of the contexts are most relevant for dispatching.

The implementation in this thesis extends Ř, a JIT for the R programming language. We evaluated it over a set of standard benchmarks and real world programs. Compilation times are significantly reduced, without sacrificing the peak performance of the JIT.

KEYWORDS: JIT, Just-in-time compilers, dynamic analysis, specialization, code reuse, serialization.

Contents

List of Figures

ABBREVIATIONS

AOT	Ahead of Time
AST	Abstract Syntax Tree
BC	Byte Code
CFG	Control Flow Graph
FV	Feedback Version
HAST	Hash of the AST
GC	Garbage Collector
JIT	Just In Time
L2	Two Level Dispatcher
LLVM	Low Level Virtual Machine
OBAP	Offline Bitcode Analysis and Processing
GNUR	Stands for GNU R, the standard implementation of the R programming language
SSA	Single Static Assignment
SEXP	S-Expression Format

Chapter 1

Introduction

The growing adoption of dynamic languages is in part due to the ability of just-in-time (JIT) compilers to perform powerful online optimizations. The general idea is that the cost of optimizations will be amortized by the benefit provided by the compiled code. This behavior of JIT-based runtimes may lead to huge warmup costs, which in some cases may never get amortized. One popular way to tackle the problem of huge warmup times is to use ahead-of-time (AOT) compilers [1, 2]; where code is compiled offline and the generated binaries are made available to the runtime for direct execution. Note that AOT compilation is beneficial (compared to plain interpretation) only when offline analysis may indeed lead to optimizations taking place. This is rarely true in dynamically typed languages (such as R), where even simple optimizations might not be possible (discussed in Chapter 2).

To conjure a strategy that obtains the benefits of JIT compilation without its warmup overheads, observe that JITs perform optimizations using a two-pronged approach: They can make assumptions about a function at its call site, and they can use its runtime profile as feedback to optimize its body incrementally. The more the JIT learns this way from a function's execution, the more can it optimize the function's future invocations. One could also visualize the state space of JIT-compiled binaries as being derived from multiple prior runs of a given program, each of which contributes to possibly different behaviors for each compiled function, based on two levels of contexts: call-site assumptions and feedback information. Given a call site to a function in future program runs, selecting a suitable binary from previously observed options while ensuring efficient dispatch poses a challenge.

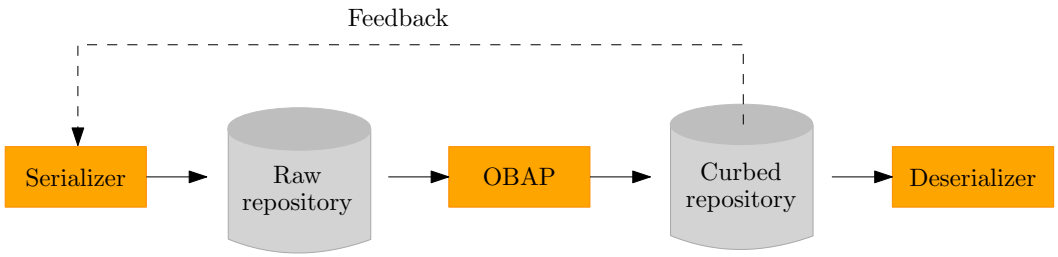


Fig. 1.1: Block diagram of our proposed approach.

Let us now consider this problem of linking to a suitable contextually specialized binary from multiple JIT-compiled versions of a function. If we are overly optimistic and pick a highly specialized binary, the dispatch may succeed for compatible contexts but fail and de-optimize for others. This may in turn cause extra compilations and/or degrade performance. If we are overly conservative and pick a binary that is generic enough to be dispatched across multiple contexts, we would end up sacrificing the peak performance that can be derived from specialization in the first place. The ideal solution should maintain all the contextually specialized binaries and very carefully pick the most relevant binary, based on an exact match of the two-level runtime context. The problem with such an approach would be to first deal with the challenge of code explosion while preserving the binaries and then incur an inefficient dispatch during runtime. In this thesis, we describe a three-staged approach that addresses all the challenges described above: It allows for an efficient dispatch from a set of functionally unique binaries[1], and successfully maintains peak performance with significant reductions in warmup time as well as late-stage compilations.

Figure 1.1 shows an overview of our approach. The idea involves three stages: (i) The *Serializer* compiles and serializes contextually compiled binaries while indexing them with call-site assumptions as well as feedback information. This stage also inserts indirections during serialization, such that absolute addresses can be patched in the future. (ii) The *Offline Bitcode Analysis and Processing (OBAP)* stage identifies functionally-unique binaries and removes the rest, while masking redundant parts of the context, to achieve a smaller repository of curbed binaries (the subset of functionally unique binaries tagged with the part of the context which makes them unique). (iii) The *Deserializer* is the runtime that dis-

[1]Given a set of compiled code versions (for a function), a functionally-unique set excludes compiled versions that are identical or similar (based on some metric) to some other version. Also see Section 2.6.

patches to the most relevant binary based on the runtime context while making use of the metadata generated in the OBAP phase to patch the binaries while linking. All the stages are designed keeping in mind the goal of replicating the performance of previously seen JIT compiled binaries while reducing the overheads as much as possible.

We have implemented our approach for the R programming language. R is a dynamically typed lazy language known to make the life of compiler designers notoriously difficult [3] due to possible side effects while forcing promises, reflective access to runtime stack, and so on, all of which deem it extremely unsuitable for AOT compilation. As our test bed, we use the research JIT compiler Ř [4], which takes R bytecode through various intermediate levels and converts the same to LLVM bitcode (a near-binary IR) for further optimization. Ř maintains multiple versions of a function and performs contextual dispatch based on assumptions about arguments at its call sites. The versions are created based on runtime feedback, such that each new compilation incorporates more assumptions to minimize subsequent deoptimizations. As may be expected, such contextual specialization and dispatch, though known to be highly effective in improving the peak performance of R functions, suffers from a high warmup time as well as overheads due to late-stage compilations. We thus process LLVM bitcodes through the pipeline described above and aim to achieve similar or better performance than Ř, while keeping the associated overheads to a minimum.

We evaluate our implementation over standard R benchmarks and real-world use cases, and compare it with baseline Ř in terms of peak performance, warmup time, and compilation overhead. In addition, we assess the individual components of our approach by showcasing their effects on binary reduction and context curbing. Throughout the text, we discuss alternate approaches and the design choices that we make to come up with an efficient strategy that is precise enough. Overall, we conclude that our scheme not only brings the benefits of JIT compilation at less cost but also makes it compelling for real-world scenarios by reducing the impact of classical problems like context explosion and redundant compilation.

Fig. 1.2: Per iteration performance of Ř over GNUR.

1.1 Motivation

In the world of dynamically typed languages such as R, where not much is known about a function before its execution, only runtime feedback provides suitable information that can adapt a function to the runtime. This approach is used by JIT compilers to make speculations that translate into improved real-world performance. Let us consider the following benchmark programs from the RBenchmarking [5] suite:

> **mandelbrot** defines functions to generate the Mandelbrot set and verify the results based on the number of iterations. It calculates the set by iterating over a grid and accumulates binary values to determine membership, returning the final result.
>
> **pidigits** defines functions for performing arithmetic operations on big integers and implements the pidigits algorithm. It calculates the digits of the mathematical constant pi using the Bailey-Borwein-Plouffe (BBP) formula.

A harness calls the benchmark programs with predefined inputs, and the time taken per call (or iteration) is plotted in Figure 1.2. Here mandelbrot is a simple program where one hot function is responsible for the majority of the runtime. On the other hand, pidigits is a more complex program where computation is divided into various functions. We compare the times taken by GNUR, Ř and $\check{R}_{no\_profiling}$ to answer the following:

- *Is profiling-based speculation really worth it?* Comparing both examples, it becomes evident that Ř outperforms GNUR by a significant margin. However, when profiling

is disabled, the performance of $\check{\text{R}}_{no\_profiling}$ is either worse or comparable to GNUR. This observation can be attributed to the highly dynamic nature of R [3], where limited optimization opportunities exist without speculation.

- *Is warmup really a problem?* In a simplistic scenario like mandelbrot, where a single hot function is repeatedly called and spans around 50 lines of code, the impact of warmup is unlikely to be significant. However, in real-world programs such as pidigits, warmup becomes a substantial concern, which raises questions about the benefits of having a JIT itself.

  In the case of pidigits, even after compilation, each iteration is only approximately 0.05 seconds faster than GNUR. Consequently, the time spent in the initial two iterations will only be amortized if the function is executed more than 1300 times. The decision to compile code never guarantees a definite payoff.

  For instance, if we only intended to run pidigits 100 times, opting not to use a JIT would result in significantly faster execution. This predicament is a common issue that affects all JITs, where the initial overhead and warmup period must be weighed against the subsequent performance gains.

It is clear that speculation-based JIT compilation in R provides improved peak performance, but as a result of this speculation, the resultant compiled code is heavily tied to that runtime. An attempt to reuse the older JIT binaries must account for this context-sensitive nature of the binaries and select the most relevant binary based on the runtime context.

We now use an R code snippet to illustrate how different runtime contexts can emerge for the same function; see Figure 1.3. Consider the call to `foo` at line 4 in program 1, where we call `foo` using the call-site context $C_1 = \langle int, int, missing \rangle$, indicating that the first two arguments passed were observed to be integers and the third argument was missing. This call-site context allows the JIT compiler to resolve some aspects of the caller that can lead to optimizations. For instance, under this calling context, it can be inferred that the arguments that are provided to the function are free from any side effects. Knowing the side-effect-free nature of variables `x`, `y` and `z` allows the JIT to safely remove line 2 (in `foo`), which would not be possible otherwise. This is because, in R, arguments are passed

```
1 foo <- function(x, y, z=TRUE) {
2     x;
3     if (z) res <- m1(x, y)
4     else res <- m2(g, y)
5     res <- bar(res, g)
6     res
7 }
```

```
1 # Program 1
2 m1 <- fun1; m2 <- fun2;
3 g <- matrix(...); bar <- fun3;
4 foo(10, 11)       # context C1
```

```
1 # Program 2
2 m1 <- fun1; m2 <- fun2;
3 g <- vector(...); bar <- fun3;
4 ...
5 foo(10, 11)       # context C2
```

```
1 # Program 3
2 m1 <- fun1; m2 <- fun2;
3 g <- matrix(...); bar <- fun3;
4 foo(10, 0)        # context C1
5 ...
6 foo(10, 11, FALSE)  # context C3
```

Fig. 1.3: Code snippet to demonstrate the problem of context explosion with JIT specialized binaries.

as unevaluated promises which are forced when needed. These promises can be laced with side effects and can also modify the runtime stack; in other words, the argument passed to the function can modify the function they are given to. The call-site context also allows the JIT to determine that the true branch will always be taken; because the default value of the argument `z` is `TRUE` and line 2 is free from side effects. Although we were able to apply several optimizations leveraging the call-site context, these optimizations represent only a fraction of the JIT's full potential. Additionally, the JIT leverages runtime feedback gathered from previous executions of the `foo` function to achieve further specialization through optimizations like inlining, scope resolution, constant folding, and more.

If we take a look back at what just happened, the JIT specialized each compilation of `foo` to the program's runtime in an effort to generate the most optimized binary. Now let us consider the calls to `foo` made from Program 1 (at line 4) and Program 2 (at line 5); we see that the call-site context is the same but in this case, the type of the variable `g` has changed (from `matrix` to `vector`). If we were to reuse the previously compiled binary for `foo` (from Program 1), we would end up deoptimizing making the code reuse effort futile. Thus, on one hand, it is possible to only serialize very generic JIT compilations (compilation done without any speculation), but that would mean that we end up losing significant performance (see $\check{\text{R}}_{no\_profiling}$ in Figure 1.2). In the case of Program 3, we see that the first call to `foo` is under a previously seen context $C_1 = \langle int, int, missing \rangle$, but the second call to `foo` is under a new context $C_3 = \langle int, int, bool \rangle$. We would like to be able to dispatch the first call to the binary that was compiled under C_1 and leave the new unseen context to the JIT.

As explained above, we see that small changes in the context can lead to differently optimized binaries getting generated. This context explosion can happen exponentially if we consider the compilations obtained from a large set of programs from different sources. If we were to reuse the previously generated code, we would need to account for all these compilations and match them accurately with the runtime. The problem of reuse can be solved by answering two basic questions.

(i) Is it possible to reduce the set of contexts by eliminating redundant contexts?

(ii) Is it possible to efficiently match the compilation context with the runtime context?

In this work, we answer both these questions and propose a set of general techniques that can be used to adapt the code reuse strategy for any JIT system.

Our approach. This work proposes to extend a just-in-time compiled system with *L2 dispatch strategy* and an *offline code repository*, with the aim of decreasing how many times the compiler is invoked. At a high level, our approach is as follows. Each time the system is asked to compile a function ($compile(f) \rightarrow V$), it does as requested, additionally, the pair $\langle C, V \rangle$, where context C encodes all of the compiler's assumptions, is stored in the repository. A single function may be associated with many such pairs. Subsequent calls to f query the repository for an applicable compiled code fragment. To bound the size of the repository, the OBAP process deduplicates $\langle C, V \rangle$ and $\langle C', V' \rangle$ if the context C' is entailed by C and if V' is similar to V.

In practice. The devil lies in the engineering details. Our implementation must answer a number of practical questions. *What is a context?* We extend previous work by [4] to include the runtime feedback information recorded by the interpreter. *How to dispatch on contexts?* We perform a two-level dispatch that uses both the information available at the call site, and an inline cache of previously observed feedback. *What to store in the repository?* We store a portable intermediate representation of the compiled functions along with the compile time contextual data. *How to curate the repository?* We remove redundant entries by using similarity metrics and then tag each entry with a representative context subset.

Expectations. What should we expect from the resulting system? We should be able to run all R programs as we have not restricted the semantics of the language. Given that some features of R generate code on the fly, it is to be expected that the compiler will be called from time to time. Furthermore, Ř has some limitations that prevent the compilation of a number of idioms, so interpretation overheads cannot be eliminated entirely. We expect to see a reduction in warmup times as we will not have to perform the most expensive steps of compilation (high-level optimizations in PIR and LLVM low-level optimizations). We expect to be able to reuse code across the same program and different programs using the same libraries. Peak performance should be unchanged, provided the dispatch costs are not prohibitive. We expect the code repository to be large but not unreasonable in size.

Results. Our empirical results suggest that the approach presented can reduce compile times by 3.38× while retaining peak performance. We also observed that by re-using previously compiled versions of functions, it is sometimes possible to reduce the impact of phase change behavior in programs. Finally, OBAP can reduce the size of code repositories by over 60%.

1.2 Objectives

The main aim of this thesis is to answer the following research questions.

RQ1. How effective is our scheme in comparison to a traditional JIT that uses runtime feedback to perform contextual specialization?

RQ2. How effectively can our offline processing phase get rid of redundant binaries and elide unnecessary specialization?

RQ3. What is the impact of the two-level dispatch help in preserving peak performance?

RQ4. Does our scheme handle pathological cases that arise in a complex JIT system involving multiple compilation contexts due to speculation and deoptimization?

1.3 Thesis Organization

The remainder of this thesis is structured as follows. In Chapter 2, we discuss the challenges of optimizing R, the idea of a JIT compilation, and the working of Ř. In this chapter, we also discuss the contextual-dispatch system of Ř and the redundancy of contexts in compiled code. Chapters 3,4, and 5 describe the three stages of our approach: the serializer, OBAP, and the two-level dispatch-based deserializer. In Section 6, we evaluate our approach against the baseline Ř compiler. Section 7 provides an overview of relevant related work, and Section 8 concludes the thesis by outlining future research directions.

1.4 Contributions

- We introduce a novel two-level dispatching strategy that allows for the reuse of contextually specialized JIT binaries in highly dynamic languages. The dispatcher matches the runtime context to the compilation context to pick the most relevant binary.
- We present an offline analysis that identifies redundant parts of contexts and identifies functionally unique binaries to prevent code bloat and context explosion.
- We employ a backtracking slot-selection algorithm that identifies a minimal set of slots that need to be compared during runtime for dispatching relevant binaries.
- We investigate various pathological scenarios that arise in a complex dynamic runtime and demonstrate improvements imparted by our approach to addressing the same.

Chapter 2

Background

The R programming language provides a wide range of dynamic features that allows its users to write highly expressive programs. R supports features like lazy evaluation, first-class closures, dynamic typing and reflection. Such features enable the implementation of some highly useful functionality, for example, (i) The ability to treat functions as first-class closures allow for the creation of higher-order functions (ii) dynamic typing allows the same function to return different values based on certain conditions (iii) lazy evaluation delays computation until values are actually needed. Despite all the benefits that these features may provide to the user, they often come at the cost of slow and complex runtimes. In the past, Ahead-Of-Time (AOT) compilers have been used to analyze and optimize programs offline, turning high-level program constructs into optimized machine code. This approach is, however, not very impactful for programs written in dynamic languages like R.

```c
// Language: C
// Parent scope
int x = 100
void f(int a) {
    if (...) b = a + 1;
    // Function scope
    print(x);
}
```

```r
# Language: R
# Parent scope
x <- 100
f <- function(a) {
    if (...) b <- a + 1;
    # Function scope
    print(x)
}
```

Fig. 2.1: Snippet showing late reification of environments in R.

To understand why AOT is ineffective in case of R, let us consider the code snippet in Figure 2.1. In case of C, when compiling the function `f`, we could be certain that the value of `x` (see `print(x)` at Line 7) is loaded from the parent scope, as no other declaration of `x` in the function scope shadows the global definition. This would allow the compiler to fix a global address for `x` and directly reference that in the compiled code of `f`. However, in case of R, this is not possible. Even though we do not see any other declaration of `x` in the function scope, it may still conditionally exist there. To demonstrate this, let us consider the following call to `f` in Figure 2.2.

```
1 badIdea <- function() {assign("x", 11, sys.frame(-1)); 1;}
2 f(badIdea())
3 # Output: 11
```

Fig. 2.2: An example demonstrating side effects caused due to forced arguments in R.

Here, *lazy evaluation* and *reification of environments at runtime* (environments are also first class in R) allow for the conditional presence of the binding to `x` in the function scope of `f`. When the call is made to `f(badIdea())`, the argument `badIdea()` is packed inside a promise (for lazy evaluation) and loaded onto the argument stack. Inside `f`, whenever the `if` condition is true, `b <- a + 1` at Line 5 is executed. This operation forces evaluation of `a`, leading to the execution of `badIdea()`. The call to `badIdea()` modifies the parent stack frame (which is the environment of `f`) and creates a new binding for `x` in the function scope of `f`. Thus, when execution reaches the `print(x)` statement (at Line 7), the new value of `x` is found and printed (i.e. 11). Such dynamic features, coupled with arbitrary side effects make even simple static analysis ineffective. To remedy this, most dynamic languages employ the use of JIT compilers (see Section 2.1).

Other R implementations. *GNUR* is the official implementation of the R programming language. R 2.13 introduced a bytecode compiler, documented by [6], improving performance and efficiency. The idea of a specializing interpreter for R was explored by Purdue FastR specializing interpreter [7], and Oracle FastR [8] extends upon this work with a full-fledged JIT backend written in GraalVM. An alternative implementation of R, focusing on parallel processing and optimized math libraries is provided by Microsoft [9].

```
1 # Version 1
2 f <- function(a) {
3     # Assume a has no side effects
4     if (...) b <- a + 1;
5     # Assume x is an integer
6     printInt(x)
7 }
```

```
1 # Version 2
2 f <- function(a) {
3     # Assume a is an integer
4     if (...) b <- a + 1;
5     # Assume x is an integer
6     printInt(x)
7 }
```

```
1 # Version 3
2 f <- function(a) {
3     # Assume a is a closure object
4     if (...) b <- a + 1;
5     # Assume x is an integer
6     printInt(x)
7 }
```

```
1 # Version 4
2 f <- function(a) {
3     # Assume a has no side effects
4     if (...) b <- a + 1;
5     # Assume x is any
6     print(x)
7 }
```

Fig. 2.3: Different JIT optimized versions of *f*.

2.1 Just-In-Time Compilation

The basic idea of a JIT compiler is to gather information about the program's runtime behavior, a process known as profiling, and utilize this information to make informed predictions. These predictions are then used to compile optimized code that caters to the most common use cases, referred to as speculation, during the program's execution. In situations where a speculation turns out to be incorrect, the JIT-compiled code cannot proceed with execution and reverts back to the base interpreter, a process called deoptimization. This fallback mechanism ensures that the program can continue running, albeit with a potentially slower performance. To illustrate the concept of JIT compilation, we will use the function `f` from the previous example (see Figure 2.1).

Let us consider four possible compilations of `f` shown in Figure 2.3. The first version of `f` assumes that `a` is free from side effects, meaning that the reaching stores to `x` belong to the parent scope [1]. This allows us to skip the environment lookup in the function scope of `f` and directly lookup in the parent scope. Also, the assumption that `x` is an integer allows us to call `printInt` (see Line 6), which is specialized to print integers. Similarly, in the second version, `a` is assumed to be an integer, which also implies that it is free from side effects. This additionally allows us to directly generate code for the addition of integers at

[1] If no side effects or explicit stores happen to a variable then we can determine if a binding exists in the current environment or not.

line 4 along with all the previous optimizations from the first version. In the third version of `f`, `a` is found to be a closure, which means that it may indeed have side effects. Thus the reaching stores to `x` cannot be determined. However, the call to `printInt` (see Line 6) can still be made as `x` is assumed to be an integer. The fourth version of `f`, makes only one speculation on `a` being free from side effects, meaning that we can improve the lookup of `x` as done in the previous versions but in this case, as type of `x` is assumed to be *any* (may be of any type), `print` (see Line 6) remains unoptimized. Depending on the runtime profiling, a differently optimized version of `f` is compiled, in other words, *"JIT tailors a function to better suit the needs of the runtime"*.

What happens when the speculation goes wrong? To prevent the execution of wrong code, the compiler (i) inserts guard conditions (if/else conditions) that check if the compile time speculations are valid at runtime (ii) trigger *deoptimization* when a guard fails. During deoptimization, the execution of compiled code stops and all the intermediate results are collected and control is transferred back to the interpreter [2]. The execution is then resumed from the point where the speculation has failed.

In a highly dynamic language like R, a JIT compiler must make meaningful optimizations to improve performance. Failure to optimize the code in a meaningful way would mean that we not only spend additional time in compilation but also have no way to amortize it in the future, meaning that we made the runtime even slower (see $\check{R}_{no\_profiling}$ in Figure 1.2 from the previous chapter). This is why JIT compilers for different languages employ different approaches and IRs to better optimize programs. In the next section, we discuss the working of the Ř JIT compiler and its various stages.

2.2 The Ř JIT Compiler

The Ř compiler is an open-source project available on GitHub. The compiler has been written in C++ and interacts with GNUR by being dynamically linked at runtime. Additionally, a few parts of the official GNUR implementation are modified to be able to accept EXTERNALSXP[3] objects. Figure 2.4 shows the different parts of Ř.

[2] This is often a slow process that can lead to slowdowns in the runtime if it occurs too frequently.

[3] EXTERNALSXP is a special tag given to S-Expression (SEXP) objects allocated/handled by Ř

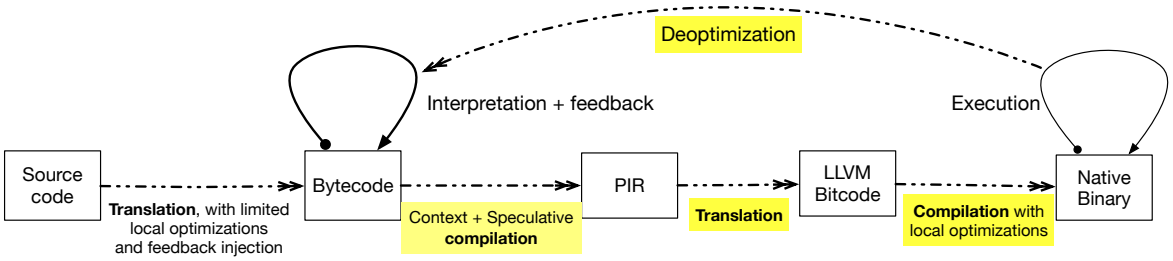


Fig. 2.4: Figure showing the pipeline of Ř.

1. **Source Code:** The GNUR parser reads user-input/script files and produces language objects stored in SEXP format. These SEXP objects are given to the Ř bytecode compiler which translates the language objects into bytecode.

2. **Bytecode:** The bytecode (called rir) is a sequence of fixed-size instructions and inline caches which can be interpreted by the Ř Virtual Machine (VM). These inline caches are used for recording runtime feedback (see Section 2.3).

3. **PIR:** When a particular piece of code is frequently executed in the runtime, it becomes a candidate for compilation. As a first step to the compilation process, the rir bytecode is translated into pir (which is a register-based IR with support for first-class environments; implementation details of pir are discussed in [3]). The pir optimizer takes the bytecode, the call-site context and the runtime feedback (see Section 2.4) to perform high-level optimizations like inlining and scope resolution.

4. **LLVM Bitcode:** Optimized pir code is translated into equivalent LLVM IR and low-level optimizations are performed. The final optimized code is then lowered into machine code and made available for execution.

5. **Native Binary:** During the execution of native code, if a guard condition fails, the execution transfers control back to the rir bytecode interpreter and records the reason for failure.

The highlighted parts of the pipeline represent the additional overheads that the Ř JIT compiler adds to the normal GNUR runtime. The work in this thesis primarily focuses on reducing the overheads of these parts of the existing implementation.

```
# R source
foo <- function(a, b, c=5) {
  res <- a * b
  res = app1(res, c)
  for (i in 1:10000) {
    if (a * i > 22) {
       res = res * 2;
    }
  }
  res
}
```

```
# rir bytecode
0:
  0   ldvar_cached_ a{1}
  9   [ double (s) ]
...
  28  ; *(a, b)
      mul_
  29  [ real (s) ]
...
  43  ldfun_ app1
  48  [ 1, <1>, valid,
      0x5623639c5740(closure) ]
...
  75  ; app1(res, c)
      call_ 2
  92  [ real (s) ]
...
  139 le_
  140 [ T ]
...
```

Fig. 2.5: Code snippet showing profiling in Ř.

2.3 Profiling in Ř

When R source code is translated into rir bytecode, additional inline caches known as feedback slots get added to the bytecode. Figure 2.5 shows the source code of function `foo` and its bytecode generated by Ř. In this example, the parts highlighted in red are the additional instructions that were inserted in order to collect runtime feedback. In Ř there are three different kinds of feedback collected at runtime. The slots at offset 9, 29, and 92 store the *type feedback* information; slot at offset 48 stores the *observed callees* information; slot at offset 140 stores the *branching* information. The following subsections discuss the storage layout and information collected by each of these slots.

2.3.1 Type feedback

To capture and store the resultant types of operations such as forcing promises (which can occur when loading arguments from the stack), evaluating expressions, or storing the results of function calls, 32-bit Type-Feedback slots are inserted into the bytecode. These slots serve the purpose of recording the types that arise from these operations.

```
1  struct ObservedValues {
2     enum StateBeforeLastForce {
3        unknown,
4        value,
5        evaluatedPromise,
6        promise,
7     };
8
9     static constexpr unsigned MaxTypes = 3;
10    uint8_t numTypes : 2;
11    uint8_t stateBeforeLastForce : 2;
12    uint8_t notScalar : 1;
13    uint8_t attribs : 1;
14    uint8_t object : 1;
15    uint8_t notFastVecelt : 1;
16
17    std::array<uint8_t, MaxTypes> seen;
18    ...
19    inline void record(SEXP e);
20    ...
21 };
```

Fig. 2.6: 32-bit struct in Ř for holding type-feedback information.

Figure 2.6 presents the implementation of the 32-bit struct that resides in each type feedback slot. When the bytecode interpreter encounters a feedback slot, the top element of the runtime stack is passed to `record(SEXP e)` (see Line 19). This function assumes the responsibility of setting the various fields of the slot and recording the type of the provided SEXP. The various fields of the type feedback slots are described as follows.

1. **numTypes:** A 2-bit value denoting the number of types seen.

2. **stateBeforeLastForce:** A 2-bit value (representing an enum of type StateBeforeLastForce) that is used to store the last state of promise before it was forced.

3. **notScalar:** A 1-bit value which is set to one if any recorded SEXP was a non-scalar (i.e. a vector of size greater than 1).

4. **attribs:** A 1-bit value which is set to one if any recorded SEXP had a non-empty attributes field.

5. **object:** A 1-bit value which is set to one if any recorded SEXP was an object.

```
1 struct ObservedTest {
2     enum { None, OnlyTrue, OnlyFalse, Both };
3     uint32_t seen : 2;
4     uint32_t unused : 30;
5     ...
6     inline void record(SEXP e);
7     ...
8 };
```

Fig. 2.7: 32-bit struct in Ř for holding test-feedback information.

6. **notFastVecelt:** A 1-bit value which is set to one if any recorded SEXP was a vector which is unsuitable for fast vector operations (if the observed SEXP is a vector with special attributes, then optimizing it directly may break the semantics so such optimizations get disabled when this field is set).

7. **seen:** It is an array that can hold upto three 8-bit values. Each element of this array represents a number that corresponds to the type of the object. For example, 0 corresponds to nil, 1 for symbols, 4 for closures, and so on (as defined by GNUR).

2.3.2 Test feedback

32-bit Test-Feedback slots are utilized for storing branching information within a program. These slots are strategically placed after branch condition checks inside frequently executed loops to enable the compiler to optimize the loops in favor of the dominant branch, thereby improving overall performance. It also aids in skipping the compilation of certain branches that have not been executed before. This optimization technique helps reduce unnecessary compilation overhead, resulting in more efficient code execution.

A boolean SEXP value (representing true or false, depending on which branch was taken) is provided to `record(SEXP e)` (see Line 6). When feedback value is stored in this slot, only the first 2-bits actually get used for storing information, while the remaining 30-bits are unused. The overall size of 32-bits ensures proper alignment of the bytecode and also allows for flexibility for additional profiling information in the future. The stored value in this slot may be one of the following.

```
1  struct ObservedCallees {
2     static constexpr unsigned CounterBits = 29;
3     static constexpr unsigned CounterOverflow = (1 << CounterBits) - 1;
4     static constexpr unsigned TargetBits = 2;
5     static constexpr unsigned MaxTargets = (1 << TargetBits) - 1;
6
7     uint32_t numTargets : TargetBits;
8     uint32_t taken : CounterBits;
9     uint32_t invalid : 1;
10
11    std::array<unsigned, MaxTargets> targets;
12    ...
13    void record(Code* caller, SEXP callee, bool invalidateWhenFull =
          false);
14    ...
15 };
```

Fig. 2.8: 32-bit struct in Ř for holding observed-callees information.

1. **None:** Previously never executed.

2. **OnlyTrue:** Only true branch is always taken.

3. **OnlyFlase:** Only false branch is always taken.

4. **Both:** Both branches have been taken.

2.3.3 Callee feedback

128-bit (32×4 bit) Observed-Callee slots store information about observed callees within a program. In R, functions are treated as first-class values, allowing their bindings to change dynamically during program execution. Consequently, every call site in R necessitates a lookup process to determine the latest binding for the corresponding function in the enclosing environment, followed by executing the call. However, to optimize performance for static call sites (call sites that invoke a fixed function), the compiler captures and retains the previously observed call targets at each call site.

Figure 2.8 provides an overview of the feedback slot layout employed to record this information. When a function lookup happens, the resolved function is given to `record()` (see Line 13). The various fields stored in this slot are explained below.

1. **numTargets:** A 2-bit value that represents the number of previously seen unique call targets. Ř stores up to three call targets at a given call site.

2. **taken:** A 29-bit counter whose value is incremented whenever the record function is called (if an overflow occurs, this is no longer incremented).

3. **invalid:** A 1-bit value that is set to true if more than three unique call targets are observed at a call site. If this bit is set, the compiler will completely skip performing all speculations based on this feedback slot information.

4. **targets:** An array of size three which holds an unsigned integer value. This unsigned integer value corresponds to the offset index (in the function extra pool, see Chapter 3) where the observed callees are actually stored.

Note that both type-feedback and test-feedback slots store information in a fixed-size (32-bit) container which may be reinterpreted as a 32-bit unsigned long. We use this reinterpretation for encoding the feedback slots in our approach. However, as callee-feedback contains indirect references, it is encoded by generating a unique identifier for each callee which is discussed in Section 3.3.

2.4 Contextual Dispatch

As discussed previously, JIT compilers use feedback information in order to compile specialized versions of functions. This idea is extended by Ř, instead of maintaining one specialized version based on the runtime feedback, Ř instead maintains multiple versions of functions at runtime. These versions are compiled under an assumption/context that must be true to be able to dispatch to that version. This work was done by [4] and implemented on the Ř JIT.

Let us consider the two versions of `f` shown in Figure 2.9. Here, the first version is only valid for execution if the argument `a` is guaranteed to be a *scalar double*. Similarly, the argument `a` in the second version must be a *vector double*. This contextual information allows us to further specialize a function without having to insert more guard conditions in the compiled code. Notice that when calling `f` with an argument of type *scalar double*, both

```
1 # Version 1: <double (s)>
2 f <- function(a) {
3     # a is guaranteed to be a
          scalar double
4     if (...) b <- a + 1;
5     # Assume x is an integer
6     printInt(x)
7 }
```

```
1 # Version 2 <double>
2 f <- function(a) {
3     # a is guaranteed to be a
          vector double
4     if (...) b <- a + 1;
5     # Assume x is an integer
6     printInt(x)
7 }
```

Fig. 2.9: Contextual Dispatch in ŘŘ.

the versions of `f` can be dispatched (because in R, a scalar is just a vector of size one). If we call `f` with a completely new type, let us say *character*, then both the versions cannot be dispatched, in which case we will be forced to execute the base interpreter version. These different compiled versions of `f` are compiled under a calling context (computed using the arguments provided at runtime) and get stored in an efficiently computable lattice at runtime, where the top of the lattice is the base interpreter context (largest context) and the bottom of the lattice is the most specialized context (smallest context). In this case, we can say that the first version of `f` is *smaller* than the second version (because all valid contexts for the first version are valid for the second but not vice versa[4]).

In a nutshell, contextual dispatch in ŘŘ allows for maintaining multiple versions of a function optimized under different call-site-based contexts. This disentangles the classes of behaviors of a function based on the context.

2.5 Runtime Behaviour of Contextual Dispatch

To lay the groundwork for the three-stage process described in this thesis, we first created an open-source visualization tool [10] called r-viz to gain a comprehensive understanding of the ŘŘ runtime. This tool served as a precursor to our development process and enabled us to gather valuable insights into the behavior of ŘŘ. The tool allows for the visualization of contextual traces that are generated by the ŘŘ runtime (with additional flags enabled). The tool uses a compressor (written in C++) to reduce the size of the generated traces which gen-

[4]The assumption *double (s)* (meaning a vector of doubles of size one) is smaller than *double* (meaning a vector of doubles). This is because the domain for *double (s)* only includes vectors of doubles of size one whereas the domain for *double* includes all the double vectors of any size. As the domain of the first assumption is a subset of the second assumption, we say that the first assumption is smaller.

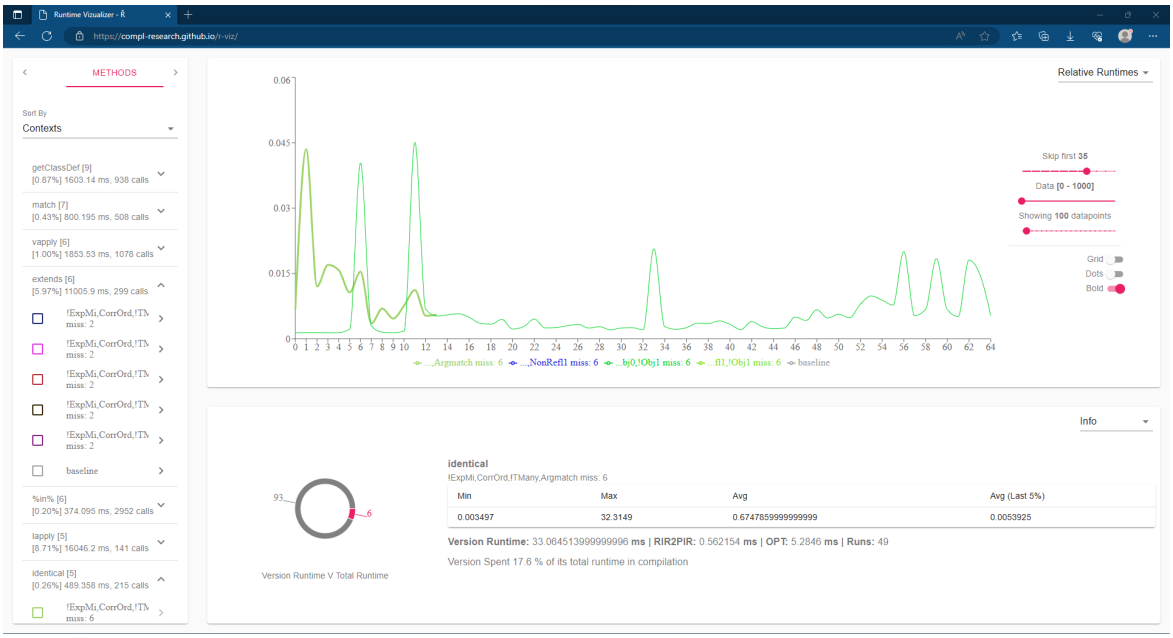


Fig. 2.10: Figure showing the layout of the r-viz visualizer.

erates a JSON format file that can be used by the visualizer (written in React). Figure 2.10 shows the layout of the visualizer. The left sidebar shows all the functions that were called at runtime, the dropdown at the top allows for sorting the functions by (i) runtime, (ii) the number of contextual specializations, and (iii) the number of calls. When a function is selected, the main top section shows the *relative runtime* behavior of the selected context with respect to the other contexts, and the main bottom section shows the statistics regarding the compilation such as times spent in the various stages of the compiler and how it compares to the actual runtime. Additionally, the bottom section can also be toggled to show the different functions that are called from within this function (the order is not always preserved as cases that invoke long calls are likely to generate inaccurate information).

In order to understand the function specializations in detail, the top right dropdown in the top section can be selected to show *contextual compilations* or *call box* apart from the default relative runtimes. Figure 2.11 shows the performance of the last n% iterations of each context, the lattice formed, and the basic details of compilation at runtime. Figure 2.12 shows the performance of the last n% iterations of each context, the lattice formed, and the basic details of compilation at runtime.

Additionally, r-viz is also capable of generating call graphs, both for contextually spe-

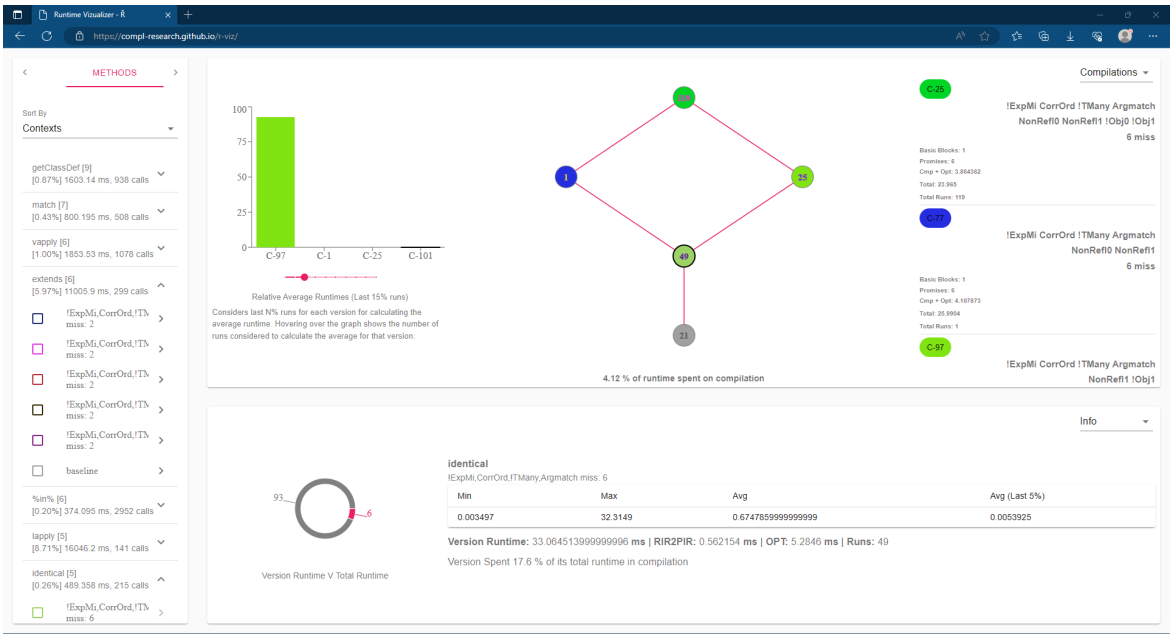


Fig. 2.11: Context lattice generated for the call-site specializations by r-viz.

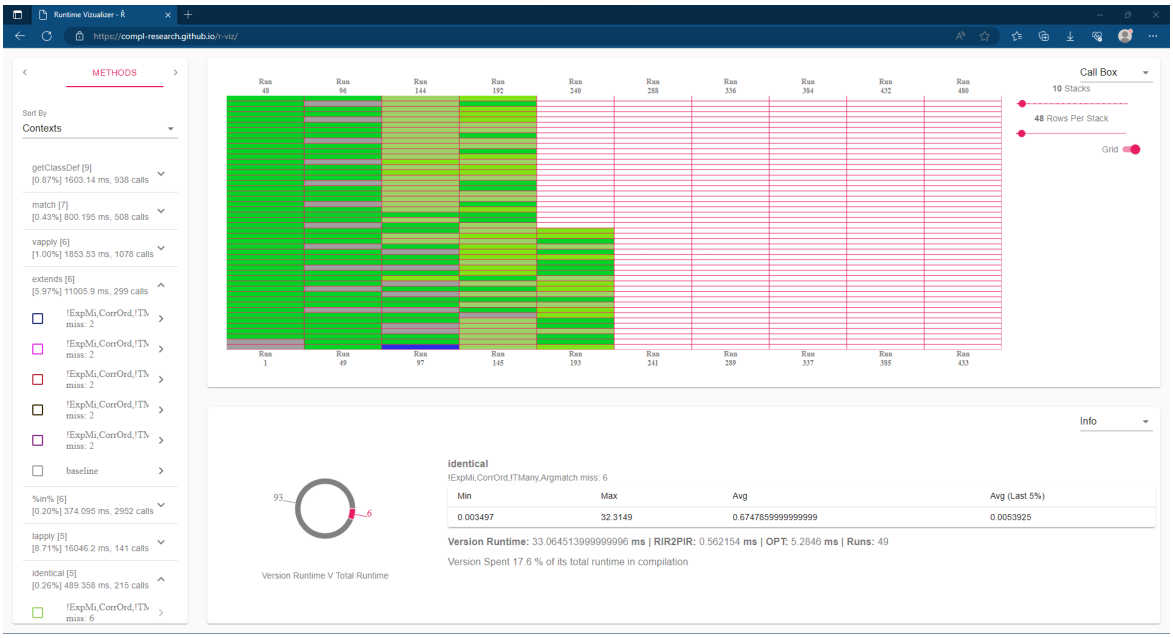


Fig. 2.12: Call box feature of the r-viz that shows the relative call order of different contexts at runtime.

```
# Program 1
f <- function(a) {
    res <- app1(a)
    res
}
app1 <- function(a) a+1 # def1
f(1)
app2 <- function(a) a+2 # def2
f(1)
f(1) # Compilation triggered
```

```
# Program 2
f <- function(a) {
    res <- app1(a)
    res
}
app1 <- function(a) a/2 #def3
f(1)
app2 <- function(a) a*2 #def4
f(1)
f(1) # Compilation triggered
```

Fig. 2.13: Contextual redundancy in Ř.

cialized calls as well as traditional calls. These features were extensively used to study regression programs such as *reg-test-s4* (which is a part of the Ř regression test suite) and the conclusions led to the approach discussed in this thesis. The main observations were (i) a lot of redundant specialization was happening in the runtime and many such compilations were never amortized. (ii) even when meaningful specializations were made, the time spent to make them was too large and unlikely to be recouped. These observations led the way to two important themes that we tackle in this thesis, namely, context explosion and code bloat. Our approach tackles both these shortcomings without affecting the existing JIT and also manages to improve it in some cases.

2.6 Redundancy in Compilation Contexts

In the context of Ř, a compiled version of a function is described by (i) source code and (ii) context (both call-site and runtime feedback). When comparing two compiled versions for the same function, if the compilation contexts are identical we can be sure that both the compiled binaries are the same. But in certain cases, even when this context is different, the resultant binary may still be the same. This can happen if the part of the context which is dissimilar does not trigger any new optimizations. For example, let us consider the code snippet in Figure 2.13. In both programs the same call-site arguments are provided, but the `app1` function differs. When `f` is compiled in the first program we would see that the feedback slot for `app1` contains `def1` and `def2`, while the second program would contain `def3` and `def4`. When `f` gets compiled, in both cases, this contextual feedback

```
# R source code
f <- function(a,b) {
  return (a+b);
}

# LLVM bitcode
define %struct.SEXPREC* @rsh1_0x55dd9cbc9ef0(i8* %code, %R_bcstack_t* %
    args, %struct.SEXPREC* %env, %struct.SEXPREC* %closure) {
  %1 = load %R_bcstack_t*, %R_bcstack_t** @ept_55dd9adbc850, align 8
  %2 = alloca %struct.SEXPREC*, i64 0, align 8
  %"PIR%3.0" = getelementptr %R_bcstack_t, %R_bcstack_t* %1, i64 0, i32
      2
  ...
  %"PIR%0.21" = load %struct.SEXPREC*, %struct.SEXPREC** %3, align 8
  ...
  %6 = bitcast %R_bcstack_t* %5 to i8*
  call void @llvm.memset.p0i8.i64(i8* align 8 %6, i8 0, i64 80, i1
     false)
  ...
```

Fig. 2.14: Figure depicting the LLVM bitcode generated from the provided R source code.

information is ignored by the compiler because `app1` is a polymorphic call site that is known to change frequently, hence no speculation is made in the compiled code. If we were to simply compare the context of these two compiled versions, they would be different, but in reality, the resultant optimized binary would be identical. The binary reduction phase (see Section 4.2) in OBAP handles this problem.

2.7 LLVM Bitcode

LLVM bitcode is an intermediate representation (IR) employed by the LLVM compiler infrastructure, serving as a low-level, platform-independent representation of a program. Notably, LLVM bitcode is represented in Static Single Assignment (SSA) form. This facilitates various optimizations during the compilation stage, such as constant propagation and dead code elimination. With its concise and structured nature, LLVM bitcode effectively captures the program's high-level semantics and low-level intricacies, enabling efficient optimization and code generation. In the case of Ř, the final optimized code (as depicted in Figure 2.4) is translated into LLVM bitcode to enable low-level optimizations before generating native executables.

The LLVM bitcode for `f` is depicted in Figure 2.14, with the compiled version represented by `rsh1_0x55dd9cbc9ef0` (see Line 8). The compiled functions expect the following input arguments:

1. `i8* %code`: Pointer to the code object of the called closure. It enables access to function-related entities like deoptimization metadata and promises.
2. `i8* %args`: Pointer to the argument stack; null if no arguments are provided.
3. `%struct.SEXPREC* %env`: Pointer to the enclosing environments SEXP object.
4. `%struct.SEXPREC* %closure`: Pointer to the callee closure object.

In Ř, the return type of all compiled functions is SEXP, specifically represented as `%struct.SEXPREC*`, which acts as the container for all R objects. The compiled LLVM code incorporates a range of instructions, including `load` (for retrieving values from addresses), `alloca` (for memory allocation), `bitcast` (for address conversion), `call` (for invoking functions), and other relevant instructions. Comprehensive documentation regarding LLVM bitcode is provided by LLVM and can be accessed online.

When converting high-level R code into LLVM, the compiled code often contains hard-coded memory references to runtime quantities. For instance, in Figure 2.14, the symbol `@ept_55dd9adbc850` (see Line 8) represents the memory address `0X55dd9adbc850`. In this specific example, the address corresponds to the runtime constant pool. In order to reuse the compiled code for multiple executions, it becomes necessary to patch these references when saving the compiled code. A comprehensive discussion on the patching process can be found in Chapter 3.

2.8 Summary

In this chapter, we explored the challenges and opportunities of optimizing dynamic languages, focusing on the R programming language. R provides a wide range of dynamic features that enable expressive programming but come at the cost of slow and complex runtimes. We first discuss the limitations of Ahead-Of-Time (AOT) compilers in optimizing

dynamic languages like R and how they are overcome using JIT compilation. JIT compilation involves gathering runtime information, making predictions, and compiling optimized code tailored to the program's behavior. We explain the concepts of speculation and deoptimization in JIT compilers, highlighting the importance of meaningful optimizations to improve performance. Next, we introduce the Ř JIT compiler, an open-source project written in C++, and discuss how runtime feedback is collected. We then discuss contextual dispatch in Ř, the problems with redundancy in compilation contexts and LLVM bitcode generation.

Chapter 3

Serializing JIT Binaries

The first stage of our system is the serializer which deals with recording compilation artifacts from various runs into a common repository. Any approach that involves the creation of such a repository must answer the following questions.

1. What level of specialization should be supported by the system?

   The Ř JIT compiler supports contextual compilation that can specialize a function based on both the runtime feedback as well as the call site arguments. If we disable these optimizations we can obtain a *generic binary* for each function which can be easily reused across runs, however, this comes at the cost of degraded peak performance. On the other hand, if we choose to serialize *specialized binaries* we can expect the same peak performance; however it's not very straightforward, optimizations such as inlining and type-speculation can insert *guard instructions* into the generated code. These guard instructions ensure that the speculations made during compile time hold at runtime; in case any check fails at runtime we deoptimize by falling back to the interpreter code. This whole process of deoptimization from compiled code to interpreter is a slow one and also invalidates the compiled code. Reusing must not only correctly patch these guard points but must also take care to select a relevant binary such that unnecessary deoptimizations are not introduced.

   In this thesis, we decided to serialize contextually-specialized binaries which in turn involves handling challenges emerging from inlining, speculation, and effective reuse.

2. Which compilation artifact will have the largest impact on the runtime?

   In a JIT compiler, the source code is often transformed into various intermediate representations (IR) to target different kinds of optimizations. The output of any intermediate stage (or the finally generated binary) in the compilation process is a valid candidate for a serialization artifact. However, each valid candidate has its pros and cons. Serialization of a very high-level artifact (like an early IR) might be easier to perform but might be less impactful overall when reused (as later stages of the compilation pipeline remain unaffected). On the other hand, serialization of a low-level artifact (such as object files) is very complicated and limits reusability (binaries generated cannot be shared across different users).

   In conclusion, the problem of identifying impactful compilation artifacts boils down to selecting an artifact that will both have a major impact on the compilation times and is high level enough to allow sharing across different users. In Section 3.2, we discuss our selection (i.e. the LLVM IR) and how this is handled by our system.

3. How would functions be correctly identified across different runtimes?

   A function in a given runtime is simply tagged by the address it exists at, however, in case we want to reuse these functions across different runs we need some way to tag functions that remain consistent. This is often done by generating a unique hash for each function. In Section 3.3, we present our approach to achieving the same.

4. How would the generated runtime binaries be patched?

   Serialized binaries are bound to have references to different runtime quantities. These references can be direct (like a pointer to the current stack pointer) or they can be indirect references (like a lookup into a specific index of a runtime object). Patching direct references can be done simply by replacing the addresses with external symbols which can be patched later. However, in the case of R, indirect references need more carefully handled. In Section 3.5 we look into the patching mechanism for the same.

5. What additional data would be needed to perform additional analysis and optimizations offline?

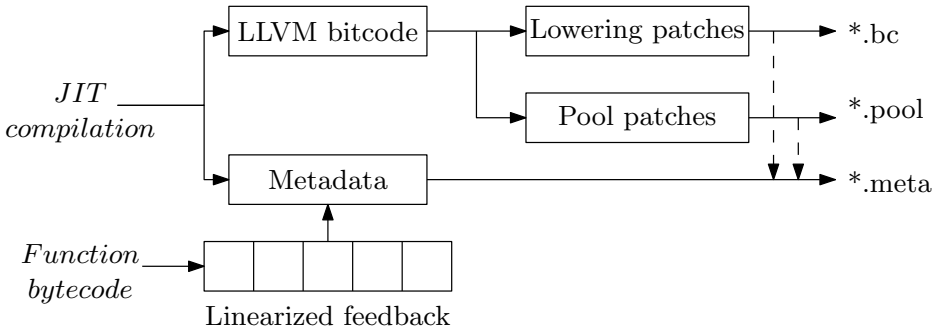


Fig. 3.1: Overview of the serializer stage.

For all the benefits that contextual specialization has to offer, there is also a huge possibility of redundant specialization (which happens when different sets of contextual information have no impact on the optimization process which can lead to the resultant binaries being functionally identical). When creating a code repository, we would like to maintain things in such a manner that it remains possible to find and remove redundant binaries offline.

3.1 Overview

Figure 3.1 shows the overview of the serializer stage. In Ř, when a function compilation is triggered, runtime feedback (which is stored in the form of inline caches in the bytecode) is used to perform speculative optimizations. Hence, along with the generation on **.bc* (LLVM bitcode) and **.pool* (serialized pool) we also serialize the runtime feedback as a linearized list (this is a vector obtained by a depth-first traversal of BC) in the **.meta* (metadata file).

3.2 The Code Repository

During the JIT compilation of functions in the serializer stage, we save the following:

- *Compiled function body (LLVM bitcode):* This is obtained when the optimized PIR code is lowered into LLVM IR. We patch generated LLVM IR (discussed in Section 3.4) and save the patched IR in LLVM bitcode format.

- *Constant pool:* When code is lowered to native, many runtime objects get shared across different binaries. For example, R integer objects, Symbols, and globals have no reason to be duplicated and exist as a single instance at runtime. This patching process is discussed in Sections 3.4.3 and 3.5.

- *Source pool:* It is similar in implementation to the constant pool but mainly responsible for storing LANGSXP objects (language objects from the GNUR). This is mainly done for the logical separation of objects at runtime. This patching process is discussed in Sections 3.4.6 and 3.5.

- *Profile data for function arguments (level 1 context):* When a function is compiled, assumptions are made by inferring properties of the call site. For example, if a function is called `f(1,2)`, the compiler creates a specialized version of `f` under context $c_x = \langle int, int \rangle$ which asserts the first two arguments as scalar integers. Similarly, when more kinds of arguments are seen, different versions are compiled.

- *Profile data for function body (level 2 context):* Runtime feedback such as previously seen types of variables and different call site targets are stored as inline caches into the rir bytecode. These inline caches are of fixed size and make up the level 2 context of a function.

- *Deoptimization information:* When an assumption fails in the compiled code, the runtime must fall back to a specific point in the code and reconstruct the interpreter state. This information is stored in the deoptimization metadata which is generated during function compilation.

- *Requirement map:* This is the set of code object dependencies created due to speculative optimizations such as inlining. A binary is valid for use only if all of its requirements are satisfied.

We chose to store the compiled function bodies in the form of LLVM IR (bitcodes), the last intermediate representation before Ř turns them into native executables. This offers some advantages. First, patching the IR to make it reusable across runs is easier than doing the same with machine code. Furthermore, the IR is more amenable to subsequent analysis

and processing. For example, our offline processing pass can re-compile it using a higher optimization level than was used by the base JIT compiler (For the sake of compilation times, Ř by default only compiles at a lower optimization level). Additionally, our binary reduction techniques (described in Section 4.2) make use of use-def chains and control flow information which are readily obtained from LLVM's single-static-assignment IR.

The profile data, or context, is collected by the base JIT as part of its normal JIT operations. We split it into two groups. The argument profile data (level 1 context) is the same context that Ř uses for its contextual dispatch feature [4]. It records the type of function arguments as well as whether the argument promise was used eagerly or lazily. The remainder of the profile data (level 2 context) refers to feedback slots inside the function body itself. The main components of the level-two context are the observed types and laziness of local variables. In addition to that, it also stores some other things such as the callees of each function call site, which powers optimizations such as inlining and replacing dynamic dispatch with static dispatch.

The deoptimization information describes how to fall back to the interpreter in case the speculative assumptions made by the JIT are violated at run-time. We must store a block of this information for each JIT guard in the function body. In Ř, this deoptimization data includes the location of the internal feedback slot to update, the bytecode offset from which to resume the execution, and the number of call-stack frames to collapse.

The requirement map is part of our serializer and deserializer system. It records all optimization assumptions that depended on other functions or objects. For example, if we inline a function, or if we speculatively replace a dynamic function call with a static function call to a known target. The requirement map helps the level-2 dispatcher pick the most appropriate code version, as will be explained in Section 4.4.

3.3 AST Hashing

Our serializer needs a way to identify and index functions across different program runs. To do this, we compute a unique identifier which we call the *hast* of the function. The hast is composed of two parts. The first part is the R namespace where the function was defined

```
1  void hash_ast(SEXP ast, size_t & hast) {
2    int len = Rf_length(ast);
3    int type = TYPEOF(ast);
4    if (type == SYMSXP) {
5      const char * pname = CHAR(PRINTNAME(ast));
6      hast = hast * 31;
7      charToInt(pname, hast);
8    } else if (type == STRSXP) {
9      const char * pname = CHAR(STRING_ELT(ast, 0));
10     hast = hast * 31;
11     charToInt(pname, hast);
12   } else if (type == LGLSXP) {
13     for (int i = 0; i < len; i++) {
14       int ival = LOGICAL(ast)[i];
15       hast += ival;
16     }
17   } else if (type == INTSXP) {
18     for (int i = 0; i < len; i++) {
19       int ival = INTEGER(ast)[i];
20       hast += ival;
21     }
22   } else if (type == REALSXP) {
23     for (int i = 0; i < len; i++) {
24       double dval = REAL(ast)[i];
25       hast += dval;
26     }
27   } else if (type == LISTSXP  type == LANGSXP) {
28     hast *= 31;
29     hash_ast(CAR(ast), ++hast);
30     hast *= 31;
31     hash_ast(CDR(ast), ++hast);
32   }
33 }
34 size_t charToInt(const char* p, size_t & hast) {
35   for (size_t i = 0; i < strlen(p); ++i) {
36     hast = ((hast << 5) + hast) + p[i];
37   }
38   return hast;
39 }
```

Fig. 3.2: Code snippet showing the AST hashing implementation.

(e.g. the base namespace, or the global namespace). The second part is a 64-bit hash of the function's AST; Figure 3.2 shows the implementation of the AST hashing function. The hash function is an implementation of djb2 [11] algorithm.

During the deserializer step, we will fetch compiled function bodies from the cache utilizing the hast. In the case of identical hashes, which can happen if two functions have the same body, the corresponding hast is blacklisted (that is, unavailable for serialization). We also blacklist a hast if it belongs to an anonymous namespace, which can happen when an inner function is compiled before its parent. Usually, such blacklists are not a limiting factor for serialization, as often, post compilation of the parent, the anonymous functions get attached to the same and get a valid unique hast.

3.4 Bitcode Patching

By default, the executable code we receive from the base JIT is not reusable across program runs because it contains pointers and other absolute references that change between runs. To remediate this problem, we patch the code to add some indirection where necessary. We call this the *lowering patches*. We rewrite the code as follows:

3.4.1 Pointers to global variables

These are the most basic patches that directly replace memory references with a unique symbol which can be looked up and patched later. These are memory references to global runtime variables such as the *R_BCNodeStackTop* (when calling a function in R, the arguments are loaded onto a stack which can be accessed by resolving their address relative to the Stack Pointer), *R_Visible* (a boolean that dictates whether a value returned by a top-level R expression should be printed or not), *R_GlobalContext* (a linked list like structure that points to the top of the runtime stack, this contains calling context information of functions), etc. The code snippet in Figure 3.3 shows the result of applying these patches.

```
-%1 = load %R_bcstack_t*, %R_bcstack_t** @ept_557fc718f850, align 8
+%1 = load %R_bcstack_t*, %R_bcstack_t** @spe_BCNodeStackTop, align 8
...
-store i32 1, i32* @ept_557fc718fd38, align 4
+store i32 1, i32* @spe_Visible, align 4
```

Fig. 3.3: Figure showing patches applied to global variables.

The prefix `spe` stands for a special runtime symbol and the postfix is the unique name of the referenced object.

3.4.2 Pointers to closure objects

The compiler can perform speculation on static calls, this means the insertion of a guard condition that needs to hold for the compiled code to be valid. During the execution of the function, this guard condition checks the current binding of the closure with the old value on which speculation was performed. During serialization, a map is maintained from the closure address to the hast of that closure. We use this mapping to replace the reference to a closure address with its equivalent hast value. The code snippet in Figure 3.4 shows the result of applying these patches.

```
-%196 = load i64, i64* getelementptr inbounds (%struct.SEXPREC, %
    struct.SEXPREC* @ept_557fc732d518, i32 0, i32 0, i32 0), align 4
+%1610 = load i64, i64* getelementptr inbounds (%struct.SEXPREC, %
    struct.SEXPREC* @"clos_NS:base:3206135521034569687_0", i32 0, i32 0,
    i32 0), align 4
```

Fig. 3.4: Figure showing patches applied to closure references.

The patched reference consists of (i) `clos` represents the type of object we are referencing to (in this case CLOSXP or closure object), (ii) `NS:base:3206135521034569687` is the unique identifier/hast of the code object and (iii) `0` is the bytecode offset of the code object, this is zero for outer functions and non-zero for inner functions.

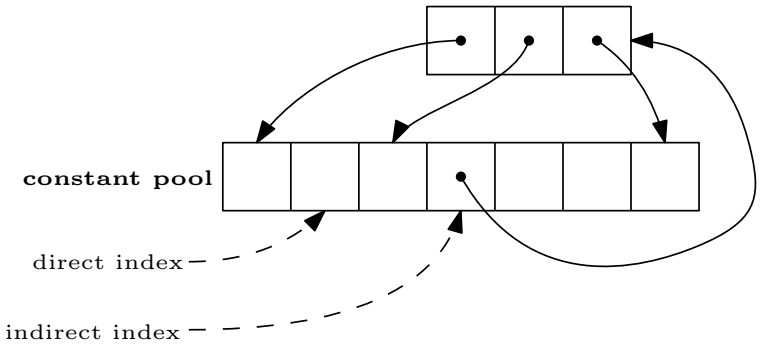


Fig. 3.5: Figure showing the different kinds of indirections in the constant pool.

3.4.3 Indirect references to objects stored in the constant pool

The compiled code often contains references to other objects such as strings, vectors, raw data, or other code objects. The constant pool is a dynamically growing array that stores these objects such that they are protected from the **G**arbage **C**ollector.

Figure 3.5 shows the two kinds of objects that can be referenced from the compiled code. The first kind is direct references to the constant pool where we access the object by the offset index at which it is stored. The second kind contains one more level of indirection which is created when we compile calls that use the named call convention (For example, when we compile a call to `foo(a, b, c)` using the named call convention `foo(c=100, a="Hello", b="Hello")`. We must first load the objects, `100`, `"Hello"` and `"Hello"` onto the stack. Additionally, to make sure the arguments are bound correctly, we need to store extra information that says the first argument on the stack must be bound to `"c"`, the second argument to `"a"`, and the third argument to `"b"`. Many calls in the code might be using such symbols, so we simply reuse the old entries from the constant pool if they already exist. We do this by creating an extra level of indirection that points back into the constant pool itself).

The code snippet in Figure 3.6 shows the patches applied to these references. These references are encoded as follows: (i) `poolp` indicates that this is a constant pool reference, (ii) `RT` is the name of the pool being referenced, in this case `RT` means the runtime constant pool (when we perform deserialization we load the serialized pool into memory separately) and (iii) `2` is the offset index where the object lives inside the constant pool.

```
; For direct references
-%176 = getelementptr %R_bcstack_t, %R_bcstack_t* %171, i64 -1, i32 2
-%177 = load %struct.SEXPREC*, %struct.SEXPREC** %176, align 8
+%180 = load i32, i32* @poolp_RT_2, align 4
+%185 = getelementptr %struct.SEXPREC*, %struct.SEXPREC** %184, i32 %180
+%186 = load %struct.SEXPREC*, %struct.SEXPREC** %185, align 8
; For named calls
-%PIRe0.26 = call %struct.SEXPREC* @createStubEnvironment(%struct.SEXPREC
    * @ept_557fc7365380, i32 2, i32* @ept_442fc718f850, i32 1)
+%PIRe0.26 = call %struct.SEXPREC* @createStubEnvironment(%struct.SEXPREC
    * @env_base, i32 2, i32* @poolp_RT_104, i32 1)
```

Fig. 3.6: Figure showing patches applied to constant pool references.

In section 3.5, we discuss how the references to the `RT` (runtime constant pool) are translated to reference a serialized pool.

3.4.4 Patching deoptimization reason

When a deoptimization occurs, we update the old value that was used for speculation to a more generalized value to prevent deoptimization loops. For example, if a feedback slot contained `[int]` and we compiled a function based on the assumption that the value is of type `int` and later during execution, we find that the actual type was `float`, before deoptimizing we store the updated value `[int, float]` into the feedback slot so that future compilations do not make the same assumption. This requires us to remember the address where the feedback slot exists at runtime. Code snippet in Figure 3.7 shows the memory address `94007299470432` being replaced with an equivalent hast reference `@"code_NS:base:3206135521034569687_0"`.

```
-@"PIR%7.2" = private constant %DeoptReason <{ i32 5, i32 0, i8* inttoptr
    (i64 94007299470432 to i8*) }>
+@"code_NS:base:3206135521034569687_0" = available_externally externally_
    initialized global i8
+@"PIR%7.2" = private constant %DeoptReason <{ i32 5, i32 0, i8*
    @"code_NS:base:3206135521034569687_0" }>
```

Fig. 3.7: Figure showing patches applied to deoptimization reason object.

3.4.5 Patching deoptimization metadata

The deoptimization metadata contains information about the stack frames that need to be collapsed and the locations in the code that the runtime needs to deoptimize to resume execution. Figure 3.8 implementation of this object; we simply new fields to allow referencing which is relative to the hast.

```
1  struct FrameInfo {
2    Opcode* pc;
3    +uintptr_t offset;
4    Code* code;
5    +char hast[1000];
6    +int index;
7    size_t stackSize;
8    bool inPromise;
9  };
10 struct DeoptMetadata {
11   FrameInfo frames[];
12   ... // method defs
13 };
```

Fig. 3.8: Code snippet showing the additional information stored in the Deoptimization Metadata.

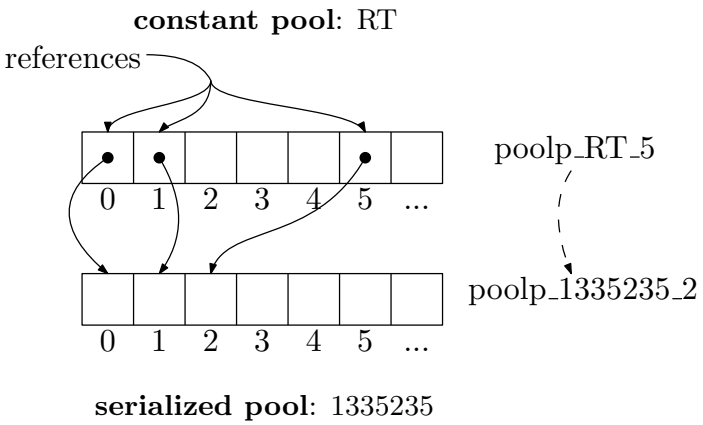


Fig. 3.10: Figure showing the process of patching the serialized pool references.

3.4.6 Patching references to the source pool

The code that is produced may reference objects in the source pool, which maintains objects that may have a mutable set of attributes. Direct serialization of such objects can result in unexpected GC behavior at runtime due to attributes not being set correctly post-deserialization. To remedy this we replace these references to the source pool with equivalent hast references so that references can be synced at runtime during deserialization.

```
-%"PIR%6.24" = call %struct.SEXPREC* @call(i64 -1, i8* %code, i32 10, %
    struct.SEXPREC* inttoptr (i64 94007322315120 to %struct.SEXPREC*), %
    struct.SEXPREC* %72, i64 1, i64 16399)
+%69 = load i32, i32* @"srcIdx_GE:6812338248726832941_2", align 4
+%"PIR%6.24" = call %struct.SEXPREC* @call(i64 -1, i8* %code, i32 %69, %
    struct.SEXPREC* inttoptr (i64 94007322315120 to %struct.SEXPREC*), %
    struct.SEXPREC* %72, i64 1, i64 16399)
```

Fig. 3.9: Figure showing patches applied to source pool references.

3.5 Pool Patching

For R language object references that belong to the source pool we perform patching as discussed previously (see Section 3.4.6), where we sync the references during deserialization. We perform the same for the object references that manage to slip into the constant

pool. After ensuring all objects that cannot be directly serialized are handled, we proceed with the serialization of objects that still can be directly serialized. Figure 3.10 shows the process by which the `RT` pool references are made relative to the serialized pool.

3.6 Summary

The chapter is focused on the challenges of serializing contextually-specialized binaries, which refers to the process of converting program code and data into a serialized format that can be easily transmitted, stored, and reconstructed later. The chapter analyzes the design considerations required for developing a serializer capable of preserving specialized binaries. Specialized binaries are binaries that are customized to specific runtimes and are highly optimized for performance. Preserving specialized binaries during serialization is essential because the loss of specialization can significantly impact the performance of the deserialized code. The chapter describes the different components of the serializer, including the AST hash, lowering patches, pool patches, and bitcode serialization.

The methodology for generating the unique identifier known as the hash of the AST is outlined in detail. The hash of the AST is used to identify the program code and data during serialization and deserialization. The chapter describes how the hash is generated and the implementation specifics of the hash of the AST are also discussed, including the algorithm used to generate the hash (see Section 3.3).

The chapter also discusses the types of patches used during bitcode serialization. Lowering patches are used to transform the program code and data into a lower-level representation that can be easily serialized. Pool patches are used to create a serialized pool that contains all the necessary information for deserializing the program code and data. The methodology for creating the serialized pool and generating references relative to it is explored in detail.

In summary, the chapter explores the challenges involved in serializing contextually specialized binaries and provides an exhaustive analysis of the design considerations required for developing a serializer capable of preserving such binaries. The chapter also provides a comprehensive overview of the system and its components, along with the methodology for generating the AST hash, using lowering patches and creating the serialized pool.

Chapter 4

Offline Bitcode Analysis and Processing

In the second stage of our system, we process all the bitcodes and metadata collected in the serializer stage. The serialized bitcodes cannot be used directly because for a given function there can exist multiple binaries that are compiled under different contexts (including both the first and second-level context). The performance during the deserialized run depends on picking the most relevant binary, at each call site. Picking a very generic binary decreases the chances of deoptimization at the cost of performance, whereas picking a very specialized binary may lead to a higher number of deoptimization events. Further, at a call site to a given function, ideally, we should dispatch to the most relevant binary for that function by closely matching the run-time context with the compilation context of the binaries.

The above ideal scenario poses two major challenges. First, for each function, there

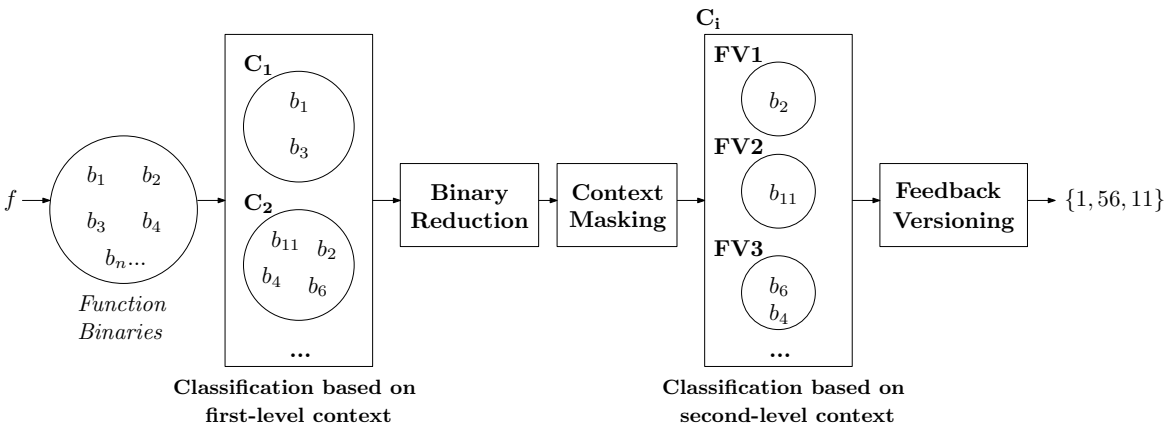


Fig. 4.1: Overview of the OBAP stage.

may be many serialized binaries obtained under different compilation contexts. Second, each context is a very large object, comprising not only the assumptions about arguments at call sites but also the run-time feedback information used to optimize various parts of the function. When we collect numerous binaries from different programs, the number of such large contexts can rise exponentially. For example, it is not uncommon to observe hundreds of contextual binaries for Ř functions, with each of them consisting of thousands of feedback slots. In this section, we present novel strategies to reduce the number of binaries as well as the size of the stored contexts, which ultimately leads to a very efficient dispatching scheme; we name this reduction phase OBAP, for *Offline Bitcode Analysis and Processing*.

Figure 4.1 shows the crucial stages of our OBAP phase. The first stage classifies serialized binaries on the basis of the first-level context (which is a set of predicates denoting assumptions about the arguments [4]). After this classification, we employ an offline analysis pass to remove redundant binaries and perform context curbing; this leaves us with a set of functionally unique binaries under each first-level context (see Section 4.2). We next reduce this set further using function-feedback information as the second-level context, essentially grouping the binaries with the same feedback together (see Section 4.3). We name each group thus obtained as a *feedback version*.

However, owing to the possibly enormous number of feedback slots, dispatching based on comparing the second-level context is usually not a practical solution. We solve this problem by presenting a *slot-selection algorithm* that selects a small fixed-size set of relevant feedback slots that can allow us to quickly compare the second-level context at run-time. Finally, observe that after slot selection and feedback versioning, we may still be left with a number of binaries under each feedback version. In order to dispatch to the most relevant binary without having to perform arbitrarily many comparisons, we next present an *ordering strategy* that arranges the binaries in a greedily optimal order (see Section 4.4).

4.1 Running Example

Let us consider the R code snippet for `foo` and its compiled rir bytecode shown in Figure 4.2 (the various feedback slots generated by Ř are highlighted in red; also see Sec-

```
1  # R source
2  foo <- function(a, b, c=5) {
3    res <- 1
4    if (x) {
5      res = app1(res, b)
6    }
7    res = res + 1
8
9    return(app2(res, c))
10 }

 # rir bytecode
 BB0:
   0   push_ 1
   5   visible_
   6   stvar_cached_ res{0}
   15  ldvar_cached_ x{1}
   24  [ <?> ]
 ...
```

```
BB1:
  40  ldfun_ app1
  45  [ 0, <0>, valid ]
...
  72  call_ 2
  89  [ <?> ]
...
BB2:
  103 ldvar_cached_ res{0}
  112 [ <?> ]
...
  123 add_
  124 [ <?> ]
...
  138 ldfun_ app2
  143 [ 0, <0>, valid ]
...
  170 call_ 2
  187 [ <?> ]
...
```

Fig. 4.2: Code snippet showing the creation of feedback slots in rir bytecode.

tion 2.3 for more details). We see that bytecode for `foo` contains seven different feedback slots which are initially empty. We will next look at how different runtime parameters affect these feedback slots and their impact on the final compiled binary. As depicted Figure 4.3, we call `foo` from four different executions as follows:

Program 1: `x` is bound to `1`, `app1` is bound to `bitwXor` and `app2` is bound to `bitwShiftL`. As the execution proceeds, the binding to `app1` eventually changes to `fun1`, following which the next call to `foo` (at Line 8) triggers a compilation.

Program 2: `x` is bound to `FALSE` and `app1`, `app2` are bound to `bitwShiftL`. The call to `foo` at line 6 triggers a compilation.

Program 3: `x` is bound to `FALSE`, `app1` is bound to `bitwShiftL` and `app2` is bound to `bitwXor`. The call to `foo` at line 6 triggers a compilation.

Program 4: `x` is bound to `1`, `app1` is bound to `fun2` and `app2` is bound to `bitwShiftL`. As the execution of proceeds, the binding to `app1` eventually changes to `fun3`, following which the next call to `foo` (at Line 8) triggers a compilation.

```
# Program 1
x = 1
app1 = bitwXor
app2 = bitwShiftL
foo(1, 10)
...
app1 = fun1
foo(1, 10) # compile
```

```
# Program 2
x = FALSE
app1 = bitwShiftL
app2 = bitwShiftL
...
foo(1, 1000) # compile
```

```
# Program 3
x = FALSE
app1 = bitwShiftL
app2 = bitwXor
...
foo(1, 1) # compile
```

```
# Program 4
x = 1
app1 = fun2
app2 = bitwShiftL
foo(1, 10)
...
app1 = fun3
foo(1, 10) # compile
```

Fig. 4.3: Code snippet showing different runtime contexts under which `foo` is called.

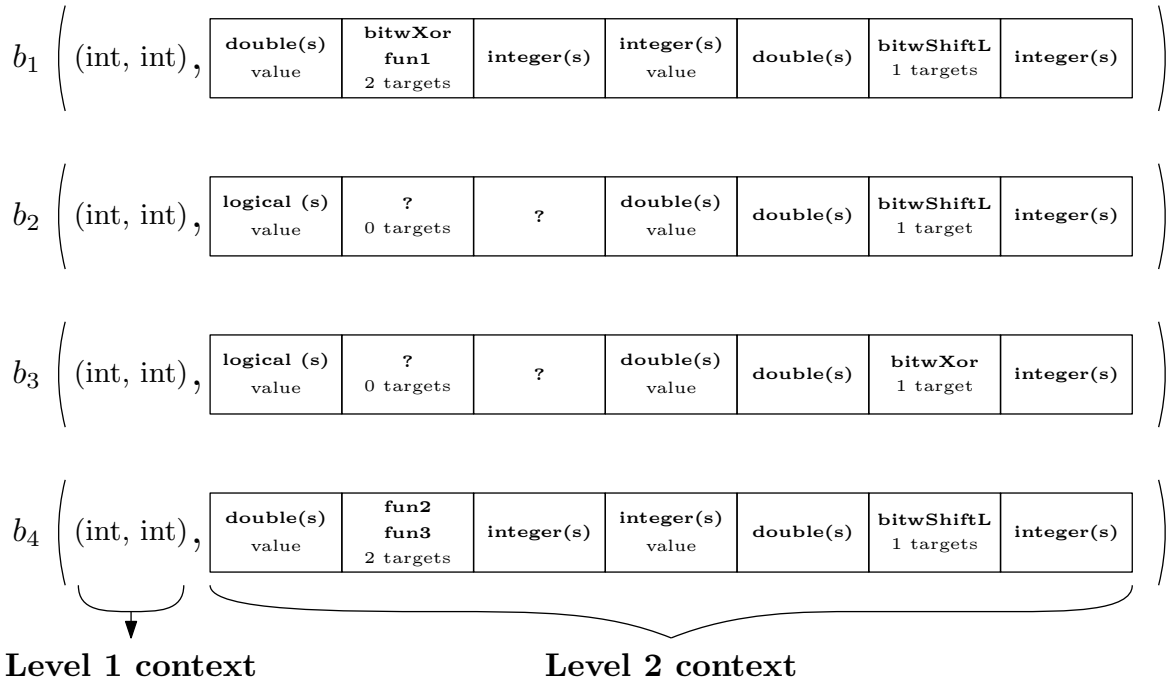


Fig. 4.4: Contextually specialized binaries obtained after performing serialization.

For all these executions, the serializer takes action to preserve both the compiled code and the associated compilation context, this is depicted in Figure 4.4. During serialization, the second-level context is obtained by traversing over the feedback slots in the function bytecode and saving them into a vector[1]. The OBAP phase first classifies binaries based on the first level context, in this example all the four binaries b_1, b_2, b_3 and b_4 belong to the same call-site context $\langle int, int \rangle$, but they differ in their second-level context.

Feedback Version. Every compilation in Ř has a corresponding second-level context which is also known as its feedback version. A second-level context is a list of feedback values (see Figure 4.4). The second-level context and the feedback version for a compilation are initially the same, but the feedback version may change after binary reduction and context curbing. For our running example, this process is shown in Figure 4.9.

In Section 4.2, we show how redundancy is eliminated and the second-level context is curbed. After this stage, Section 4.3 discusses how the slot selection algorithm is used to obtain a small subset of the second-level context.

4.2 Binary Reduction

As part of our offline analysis, we first classify the set of binaries for a given function into different groups based on the first-level context. Each resultant group contains a set of binaries that can be dispatched if the first-level context at run-time is satisfied. After this, the *binary reduction* phase identifies and removes functionally equivalent binaries from each group. To do so, we compare the binaries using a combination of three program analyses: (i) function weights (ii) breath-first call order; (iii) argument-effect analysis.

The algorithms presented for the given analysis were implemented for LLVM IR where compiled function bodies are comprised of basic blocks, containing instructions that are structured in a Single Static Assignment (SSA) format. The term *RshFunction* refers to an instruction that calls a Ř builtin. Ř defines these built-in functions as a dynamically linked library, enabling interaction between the compiled binary and the Ř runtime.

[1]For a given function, the number of feedback slots always remains constant. This is because the translation from ASTs to rir bytecode is deterministic. This is usually the case in many non-optimizing/lightly-optimizing compilers.

```
1 Function WEIGHTS (body)
2     BB ← BBITERATOR (body);
3     res ← 0;
4     while curr ← ∗BB do
5         for insn ← curr[0] to curr[n] do
6             if insn.type == RshFunction then
7                 res ← res + rshFunctionWeight[insn.type];
8         curr ← curr.next;
9     return res;
```

Fig. 4.5: Algorithm for function weights analysis.

(i) Function weights analysis. This analysis assigns different weights to the basic operations in a function body and sums them up. The weights of these operations are decided based on their complexity. Operations such as forcing promises and function calls are given higher weights as they may consist of side effects leading to failed speculation and de-optimization. On the other hand, simple operations such as array lookups and arithmetic operations are given lower weights. If two binaries differ in the function weights then they are likely to be optimized differently.

Figure 4.5 provides detailed information about the implementation of this analysis. The analysis takes the function *body* as input. Line 5 iterates over all the instructions inside each basic block (`BBIterator` function is used to get the starting basic-block address). If an instruction calls a Ř specific builtin, the predefined weight for that builtin is obtained from the *rshFunctionWeight* map (see Line 7). The resulting weights are accumulated in the variable *res*, and the final result is returned.

To establish similarity under this analysis, the final result (*res*, which is an unsigned integer) must be equal.

(ii) Level-wise call-order analysis. The change in the order in which different operations are performed indicates that the control flow of the program might have changed. Hence, in this analysis, our objective is to summarize the order in which functions are called at different levels in the control flow graph. Our call-order analysis traverses the control-flow graph (CFG) of an LLVM-bitcode function in a breadth-first order and lists the set of functions called at each level. We ignore the back edges in the CFG as small changes in the control

```
1  Function LEVELWISECALLORDER (body)
2      BB ← BBITERATOR (body) ;
3      res ← new Vector<Set< RshFunction >>;
4      workingLevel ← 0;
5      curr ← new Set< RshFunction >;
6      callback ← Function (bb, level)
7          if workingLevel ≠ level then
8              workingLevel ← level;
9              res.push(curr);
10             curr ← new Set< RshFunction >;
11         for insn ← bb[0] to bb[n] do
12             if insn.type == RshFunction then
13                 curr.push(insn)
14     BBBREATHFIRSTITERATOR (BB, callback) ;
15     return res;
```

Fig. 4.6: Algorithm for breath first call order analysis.

flow can be missed if the nodes are merged.

Figure 4.6 provides a detailed implementation of this analysis. The basic blocks of *body* are iterated in a breadth-first order using the `BBBreathFirstIterator` function (see Line 14). This function takes two inputs: the starting basic block *BB* (which is the first basic block where control enters when the function is called; `BBIterator` function is used) and a *callback* function. The *callback* function is invoked every time a basic block is visited. It is provided with *bb* (the current basic block) and the *level* (or distance) of the basic block from the starting basic block (see Line 6). Within the algorithm, two local variables, *workingLevel* and *curr*, are used to keep track of the last processed level and the set of Ř operations seen at that level, respectively. When the *callback* function is called, the algorithm first checks if the current level is the same as the last processed level. If they are different, the results of the previous level are stored in the final *res* vector (see Line 9), and *workingLevel* and *curr* are reset (see Line 8 and Line 10). If the last processed level is the same as the current level in the *callback*, the algorithm iterates over the instructions of the basic block and adds all *RshFunction* instructions to the *curr* set (see Line 13). Finally, the final result is stored in the *res* vector, which contains the set of *RshFunction* calls at each level.

To establish similarity under this analysis, the resultant vectors must be of the same size

```
1  Function ARGEFFECTANALYSIS (body)
2      BB ← BBITERATOR (body) ;
3      args ← GETARGUMENTSLOADEDFROMSTACK (body) ;
4      argImpact ← new Map<Instruction, Set<Instruction>>;
5      for arg ← args[0] to args[n] do
6          impact ← GETIMPACT (arg) ;
7          argImpact.put(arg, impact);
8      res ← new Map<Instruction, Vector<Set< RshFunction >>>;
9      for e ← argImpact[0] to argImpact[n] do
10         arg ← e.first;
11         impact ← e.second;
12         r ← new Vector<Set< RshFunction >>;
13         workingLevel ← 0;
14         curr ← new Set< RshFunction >;
15         callback ← Function (bb, level)
16             if workingLevel ≠ level then
17                 workingLevel ← level;
18                 r.push(curr);
19                 curr ← new Set< RshFunction >;
20             for insn ← bb[0] to bb[n] do
21                 if insn.type == RshFunction ∧ impact.includes(insn) then
22                     curr.push(insn)
23         BBBREATHFIRSTITERATOR (BB, callback) ;
24         res[arg] ← r;
25     return res;
```

Fig. 4.7: Algorithm for argument effect analysis.

and the set of operations performed at each level must be equivalent.

(iii) Argument-effect analysis. This analysis is used to measure the impact of the call-site predicates that were assumed for the different arguments, as part of the first-level context for a given function. For instance, consider two first-level contexts $C1$ and $C2$. Suppose $C1$ records the predicate "argument X is an integer" while $C2$ records the more precise predicate "argument X is a non-lazy integer". If in the binaries compiled under $C1$ and $C2$, the operations related to argument X are identical, then we can infer that the additional precision in $C2$ was not beneficial. Consequently, we can deprecate $C2$ in favour of $C1$. We determine this similarity by traversing the use-def chains for each argument in a breadth-first order and summarizing the set of operations performed at each level.

Figure 4.7 provides a detailed implementation of this analysis. Initially, the arguments

loaded from the stack are determined using the `GetArgumentsLoadedFromStack` function (see Line 3), which returns a list of instructions directly loading values from the argument stack. Next, we iterate over this list (see Line 5) and recursively find all the instructions impacted by each *arg*. The `GetImpact` function (see Line 6) is utilized for this purpose, leveraging use-def chains to identify all impacted instructions starting from *arg*. The results are stored in the *argImpact*[] map (see Line 4), which maps each argument to the set of affected instructions. For each argument, a level-wise call-order analysis is performed (as discussed in the previous analysis), while ensuring that only the instructions belonging to the set of impacted instructions for that argument are included (see Line 21). The obtained level-wise result for each argument is then stored in the *res* map (see Line 24), which maps the argument (loaded from the stack) to its level-wise call-order summary with respect to the instructions it affects.

To establish similarity under this analysis, the resultant map must contain the same keys (i.e. the same set of arguments must be loaded from the stack) and the level-wise call-order summaries must be identical.

For the binary reduction phase, we use the results of the above three analyses as follows: Given a function, we first calculate the analysis results for all its binaries and then perform a pairwise $\binom{n}{2}$ comparison for the same. When comparing a pair of binaries, we use a strict criterion that requires that they compare equally under all three analyses mentioned above. This strict deprecation criterion ensures that functionally unique binaries remain intact while mostly identical binaries get deprecated.

Context curbing. The similar binaries obtained above, however, cannot be directly deprecated. For example, in Figure 4.8 we see that b_1 and b_4 are identified to be similar (in cases where there are multiple potential targets for a call, the compiler generates generic code that is valid for all targets, rather than optimizing specifically for a single call site). In such cases, as part of OBAP, we pick one of the binaries and deprecate the rest. Before deprecation, however, the redundancy in the second-level context is identified and removed by context curbing. We identify the common part of the second-level context and "mask" the part of the context that differs. We use this mask to dispatch the representative binary for all three contexts while ensuring that we do not end up recompiling for a previously

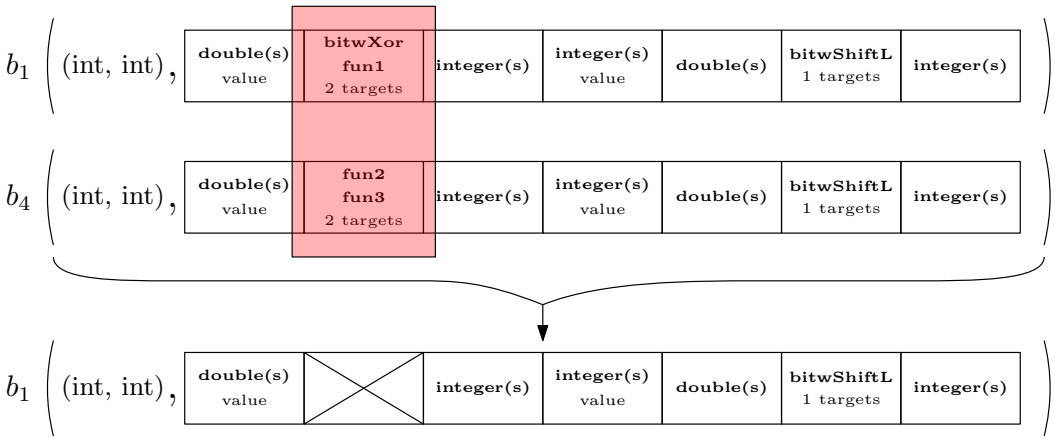


Fig. 4.8: Second-level context curbing.

deprecated context. Overall, though our strategies to identify similar binaries use simple program analyses as described above, we see in Chapter 6 that the reduction achieved due to the same is quite impactful in establishing the viability of our serialization approach for large programs.

4.3 Feedback Versioning

JIT compilers rely heavily upon the run-time feedback information to perform optimizations. This run-time feedback is collected at different crucial points during program execution, such as the types observed for loaded variables, call-site targets, and branching information, and recorded in feedback slots. Ideally, in order to obtain the most relevant binary for a function at a given call site, we should compare the run-time context with not only the call-site predicates but also all the feedback slots for that function. In the *feedback versioning* stage, we start with a set of functionally unique binaries for each first-level context and aim to classify them using a second-level *feedback context* such that they can be dispatched efficiently. We facilitate this by classifying the binaries into groups compiled under the same feedback information and calling each unique group a *feedback version*. However, in Ř, we have observed that the number of feedback slots for typical programs may vary from a few tens to even a few thousand, which poses a serious challenge to the dispatch overhead that may be incurred using such a scheme. In order to address this problem, we next present a simple backtracking-based algorithm that can find the relevant feedback slots

for comparing the second-level context for each binary. Our goal is to select a small, limited subset of feedback slots that can be used to efficiently distinguish and dispatch the relevant feedback version at run-time.

Slot Selection. We model the problem of finding a small subset of relevant feedback slots as a graph reduction problem. Each node in this graph represents a unique second-level context/function feedback vector, as described in Section 4.1, and is referred to as a *Feedback Version* (FV). Each labeled edge represents a feedback slot index, we draw an edge between two nodes if the values contained in the FV at that index differ. Figure 4.9 depicts an edge (index 5) connecting FV2 and FV3, attributed to their inclusion of `bitwShiftL` and `bitwXor` values, respectively. We refer to this set of differing indices among two FVs as the *diffset*. Our goal is to use a set of operations to reduce the graph in a way such that all the nodes become disconnected. The slot selection algorithm described in Figure 4.10 is used to reduce this graph. The inputs given to the algorithm are as follows.

- *wl*: It is initialized as a set of $\binom{n}{2}$ pairings of all the FV's in the graph.
- *sol*: It initialized as an empty set that holds the currently selected edges.
- *BUDGET*: It is the maximum size of the desired solution (or the maximum number of slots that can be compared at runtime).
- *FinalSol*: It is initialized as an empty that holds the final solution.

Figure 4.9 shows an end-to-end example of all the operations performed in OBAP. We continue with the program discussed in the running example (see Section 4.1). We start with four binaries (b_1, b_2, b_3, b_4) compiled under four unique second-level contexts (each unique value contained in the second-level context is shaded with a different color). The first stage of OBAP performs binary reduction which identifies b_1 and b_4 to be similar. Following this, the differing slot at index 1 is masked and a new curbed context representing both b_1 and b_4 is created. This reduced set is provided to the slot selection stage. Let us now solve the obtained graph using the algorithm described in Figure 4.10. The first two lines check if the solution exceeds the allocated budget and if so, returns *false*. Line 4 checks the graph for *trivial solutions* that might exist in the graph. In the given graph we see that edge 5 is the only

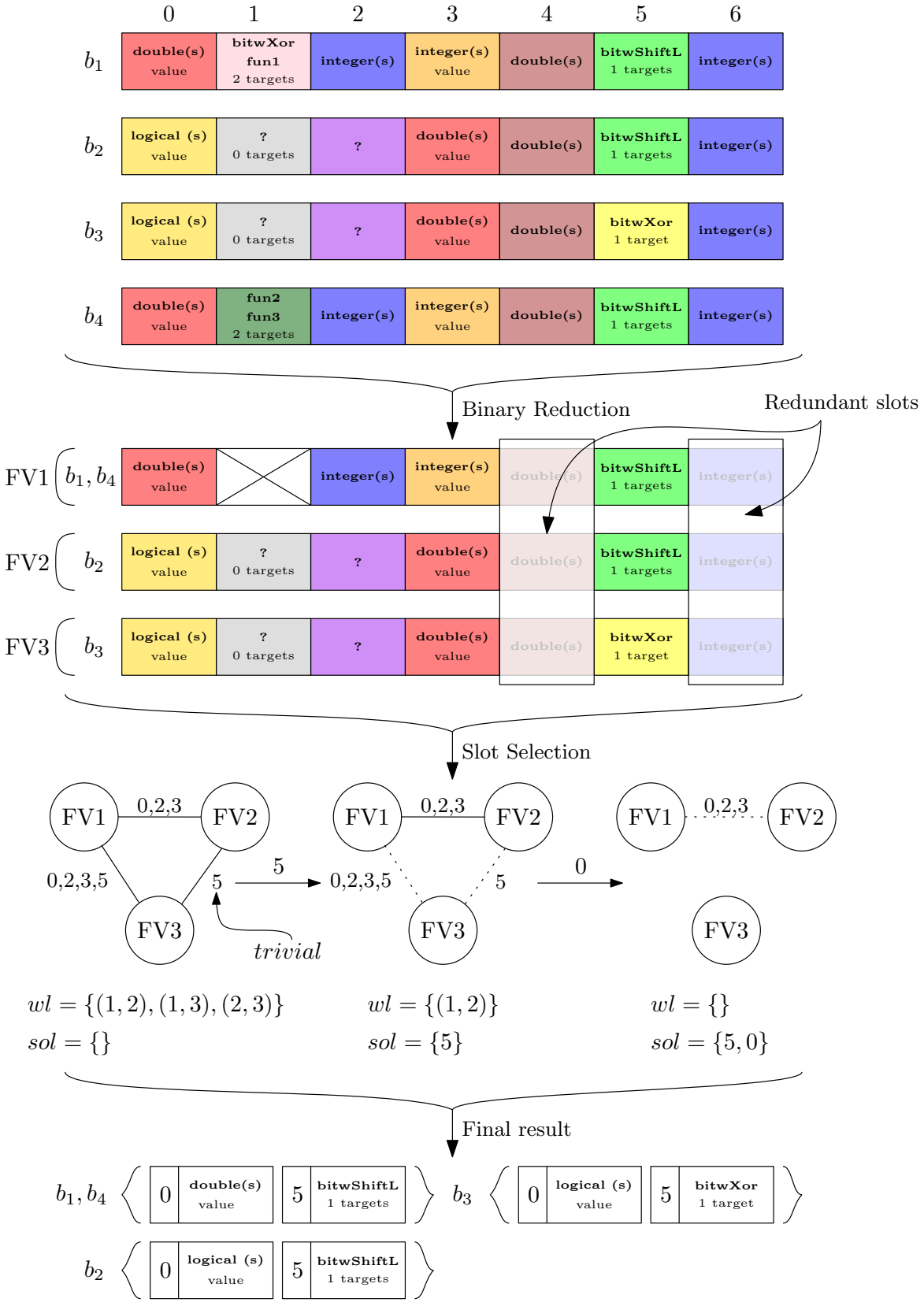


Fig. 4.9: Example to demonstrate slot selection.

edge that exists between FV2 to FV3 (in order to be able to distinguish between FV2 and FV3 we will always need to include 5, hence this is a trivial solution). This step adds all such trivial solutions to the *triv.sol* and then for all other occurrences of this edge in the graph, it removes all the connecting edges between the nodes that contain this edge. The addition of 5 to the solution set leads to the removal of all the edges between (FV2, FV3) and (FV1, FV3) (as edge 5 is also contained in the set of edges connecting FV1 and FV3). When no edges are connecting two nodes directly, we remove its entry from the worklist; hence the entries `(1,3)` and `(2,3)` are removed. We then store this reduced worklist into *triv.wl*. Line 5-7 checks if the updated worklist is empty and if so, the global variable *FinalSol* is updated with the obtained solution. Line 7 calls the function `getDiffUnion` which returns a union of all the remaining edges in the updated worklist; this is stored in the variable *rDiff*. In the given example, this operation returns {`0,2,3`}, which are the edges that still exist between the reduced worklist entries i.e. `(1,2)`. Line 9 calls the function `sortByReduction`, which sorts the edges by the number of reductions they perform on the worklist and stores it into the *sDiff*; in this case we get {`0,2,3`}. We sort the values in descending order based on the number of reductions they perform. Line 10-11 iterate over all the combinations of sDiff as follows, if *sDiff* contains {$v1, v2, .., vn$} `COMBINATIONS(sDiff)` results in $\{\{v_1\}, \{v_2\}, .., \{v_1, v_2\}, \{v_1, v_3\}, ..., \{v_1, v_3, v_n\}\}$. This ensures that the solutions with the highest number of reductions are selected first. Line 12 creates a new set *tsol* that contains the union of *triv.sol* (currently selected solutions) and *c* (current set of solutions yet to be checked). Finally, in Lines 13-15 we recursively try new solutions to reduce the graph. When the algorithm completes, we are left with the solution $\{5, 0\}$. Thus values at these indices get are tagged to the binaries and made available in the final repository.

4.4 Level-2 Collisions

After we classify the binaries based on their second-level context, in a few cases, multiple binaries may be left under the same feedback version. We attribute this to two main possibilities: (i) The binary reduction analysis was unable to identify similarity in otherwise similar binaries. (ii) The binaries differ inside the body of inlined functions, which are not

```
1  Function SOLVE (wl, sol)
2      if sol > BUDGET then
3          return false;
4      triv ← SOLVETRIVIAL (wl, sol) ;
5      if triv.wl == 0 then
6          FinalSol ← triv.wl;
7          return triv.sol <= BUDGET
8      rDiff ← GETDIFFUNION (triv.wl) ;
9      sDiff ← SORTBYREDUCTION (rDiff, triv.wl) ;
10     combi ← COMBINATIONS (sDiff) ;
11     for c ← combi[0] to combi[n] do
12         tsol ← triv.sol ∪ c;
13         if tsol <= BUDGET then
14             if SOLVE (triv.wl, tsol) then
15                 return true;
16     return false;
```

Fig. 4.10: Slot selection algorithm.

covered by the first or second-level contexts.

To have a complete specification of the L2 dispatch mechanism, we must specify which binary should be chosen in these cases. In the first case, which happens when the binary reduction fails, the choice does not matter because the binaries have equivalent behaviour. The difficult case is the second one when the definition of two-level context was not powerful enough to distinguish between different binaries. To give a concrete example of how this can happen, suppose that we have a function f that inlines g. The second level context can tell the difference between binaries for f that invoke g and ones that don't because f has feedback slots that record the callees of all function calls in its body. However, in those cases where g is inlined then the feedback slots of f don't tell anything about the parts of the compiled binary that come from the inlined body of g. This limitation happens because we record the feedback during the bytecode interpretation phase, which is before any inlining has taken place. If we end up with multiple versions of f that are completely identical except that they inlined different versions of g, then we will not be able to tell the difference based on the feedback slots for f. Perhaps it might have been possible to design a more complex level-2 context that took into account the feedback slots of the inlined function. However, we have not pursued this avenue because in our experiments we found that L2 collisions

were fairly rare, happening in less than 2% of L2 contexts. Instead, we make an educated guess and hope for the best.

When we make this choice there are two ways to get it wrong, as we have discussed in the Introduction. If we pick a binary that is too specific then it may hit a JIT guard and deoptimize to a more general version. Conversely, if we pick a binary that is too general then we might have worse steady-state performance than a more specific binary. In this work, we have opted to err toward the more specific binaries, to preserve steady-state performance.

First, we filter out all the binaries that don't satisfy their requirement map. If the current code versions in the running environment are not compatible with the requirements listed in the requirement map then that binary was not eligible to be picked in the first place. Then, among the binaries that satisfy all their requirements, we choose the one that has the largest requirement map. The idea behind this is that each entry in the requirement map refers to a speculative optimization made by the JIT compiler. Therefore, we expect that binaries with a larger requirement map will be more optimized and more specific than binaries with a smaller requirement map.

4.5 Summary

In this chapter, we delve into the complexities involved in analyzing and processing serialized code repositories. After the serialization process, the resulting repository is not immediately usable and requires several stages of processing through the OBAP. The primary objective of the OBAP is to transform the raw repository into a refined, trimmed-down version, free from redundant functions and equipped with essential metadata required for the dispatcher. The OBAP operates at a high level by first classifying function binaries based on their first-level context i.e. the call-site context. This is followed by a process of binary reduction and context masking. Finally, the remaining functions are classified based on their second-level context, and feedback versioning is applied. To aid in understanding the concepts covered in this chapter, we have included a real-world example. During the binary reduction stage, three program analysis techniques are employed on the LLVM bitcode to identify and classify functions based on their similarity. Once similar functions have been

identified, the context masking technique is applied to refine the analysis further.

Context masking helps to isolate the relevant context for each function by removing any irrelevant or redundant data from the context. This ensures that irrelevant parts of contexts are not selected for dispatch in the future. After the context masking stage, the remaining functionally unique binaries are further classified based on their second level of context. These classifications are referred to as Feedback Versions (FVs). To allow for efficient dispatching, we employ a slot selection algorithm to identify a relevant subset of context that can effectively distinguish between the various FVs. This selection process is crucial for achieving optimal performance during the dispatching phase.

Overall, this comprehensive approach to repository optimization ensures that the resulting repository is both lean and effective, making it suitable for use in the runtime.

Chapter 5

Deserializer and L2 Dispatcher

In this chapter, we discuss the Deserializer, the third and final stage of our system. It features an efficient two-level (L2) dispatcher that, at a given call site, selects the most relevant binary based on the run-time feedback for the function arguments as well as the internal feedback slots. Maintaining a code cache that stores code versions requires an efficient mechanism to support it. To develop this mechanism, we must consider the following.

1. Loading everything at once vs loading things incrementally.

   Loading everything at once (i.e. the entire serialized code repository) has its advantages (i) Access to disk happens only once, which means during deserialization the additional step of loading the serialized pool, the serialized binary does not take any extra time. (ii) Much simpler in implementation. However, this also comes with dis-

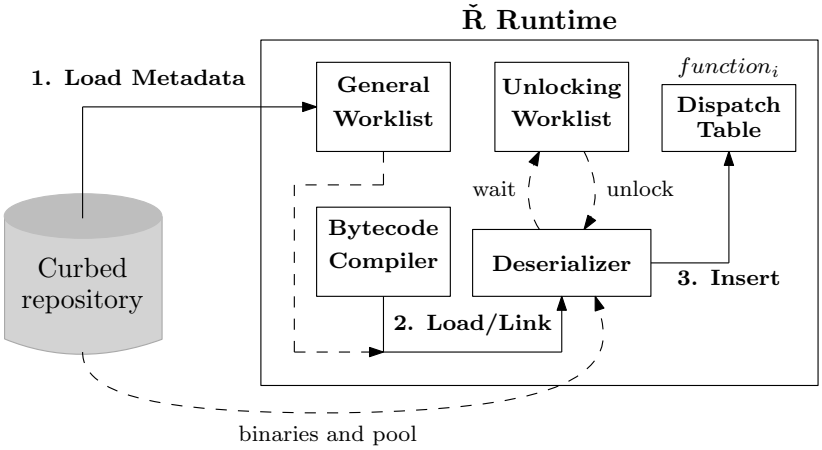


Fig. 5.1: Overview of the Deserializer stage.

advantages that make it unsuitable for our use case (i) The repository can be huge and loading everything at once will consume considerable memory (might not even be feasible in some cases). (ii) Loading unnecessary functions that we don't use simply wastes valuable resources.

2. What to do in case of a new second-level context at runtime?

   When we encounter a new second-level context (i.e. a context that has not been serialized and added to the repository in the past), we can either dispatch a saved version that matches the runtime context partly and hope it dispatches successfully or we can let the JIT handle the new context with a new compilation. Although the first approach may seem appealing, it could result in disabling a binary that could be useful in the future if dispatch fails. Therefore, it may be wise to consider alternative approaches that would not permanently disable the binary. A function can go through multiple phases at runtime, and the main aim of our dispatcher is to select the most precise binary for the phases that have already been seen.

3. When to resolve the references in the deserialized code?

   When we deserialize code, we can eagerly patch all the references inside it so that whenever the binary gets loaded into the runtime it is ready to be dispatched, or we can do so lazily as well i.e. we could resolve the references upon the first call to the function. The first approach is simpler but likely to suffer from unexpected slowdowns (although not as noticeable as JIT compilation, still significant). Therefore, we opt for the second approach as it not only saves us time during deserialization but also allows the deserialized code to be available dynamically as needed. Since not all parts of the deserialized code might be immediately needed, compiling only what is needed helps reduce unexpected slowdowns in the runtime.

Figure 5.1 shows the high-level operations that get performed during runtime in this stage. First, the runtime loads all the metadata from the code repository into the runtime, which is stored into a *general worklist*. When the runtime encounters a new function, it checks the general worklist and tries to load/link the associated binaries. If the dependencies

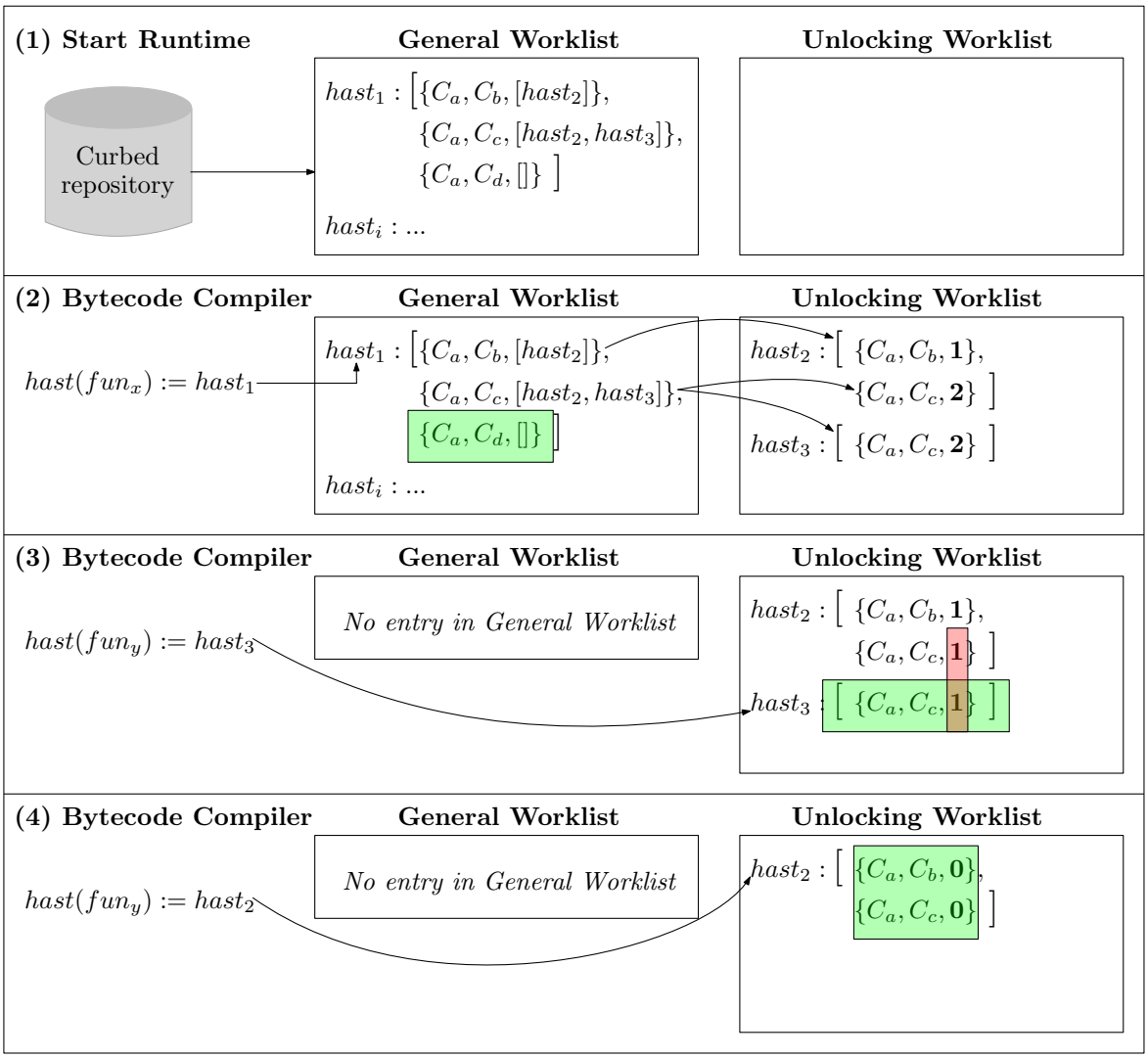


Fig. 5.2: Overview of the Deserializer stage.

are already satisfied or empty, we load/link them immediately and add them to the function dispatch table, otherwise, we add them to the unlocking worklist. The next Section 5.1 discusses the working of this process in detail.

5.1 Deserializer

In order to understand the process by which the binaries are loaded into the runtime, let us consider the example shown in Figure 5.2. As the first step, the deserializer phase starts by loading all the metadata about the serialized binaries that were prepared during the OBAP phase. This metadata contains information about the different function versions

and their corresponding dependencies. We then add this metadata to a *general worklist*, which is a mapping from a function hast (a unique identifier generated for each function, see Section 3.3) to the binary metadata, as shown in Figure 5.2 (1). When a function is compiled to bytecode, we compute its hast. We then use that hast to look into the general worklist and check if any serialized binaries are available, as shown in Figure 5.2 (2). If serialized binaries exist, the linker tries to load them. Loading all the serialized binaries, however, is not always possible at this point in time, as some binaries may have dependencies that have not yet been satisfied. To accommodate this, an *unlocking worklist* is created, which maintains an inverse mapping from the dependencies to the list of binaries that are waiting for the dependency. All the binaries in this worklist maintain a counter that is decremented whenever a dependency is satisfied. When a new bytecode is compiled (see Figure 5.2 (3)), after processing the general worklist, the unlocking worklist is checked. If this new compilation resolves the dependencies for some binary it is loaded into the runtime. At this point, the serialized pool is synced with the runtime, and the bytecode is loaded into the appropriate dispatch table. However, we delay the final patching of the binary until the first call to the function is made, ensuring that we do not end up spending time patching binaries that will not be used immediately.

5.2 Patching

When a function is loaded into the runtime by the deserializer, it remains unpatched until it is called for the first time. This process is commonly referred to as lazy patching, and Figure 5.3 illustrates how the runtime handles this. During the deserialization process, the function initially contains an address pointing to the module that holds the LLVM bitcodes. However, the function itself is not immediately executable. Upon the first invocation of the function, all the necessary indirections are patched, and LLVM generates an executable code. This executable replaces the original stub address, allowing subsequent calls to the function to directly access the compiled executable. In summary, the deserializer loads a function in an unpatched state, and when the function is called for the first time, the necessary patching and executable generation occur, resulting in subsequent calls directly

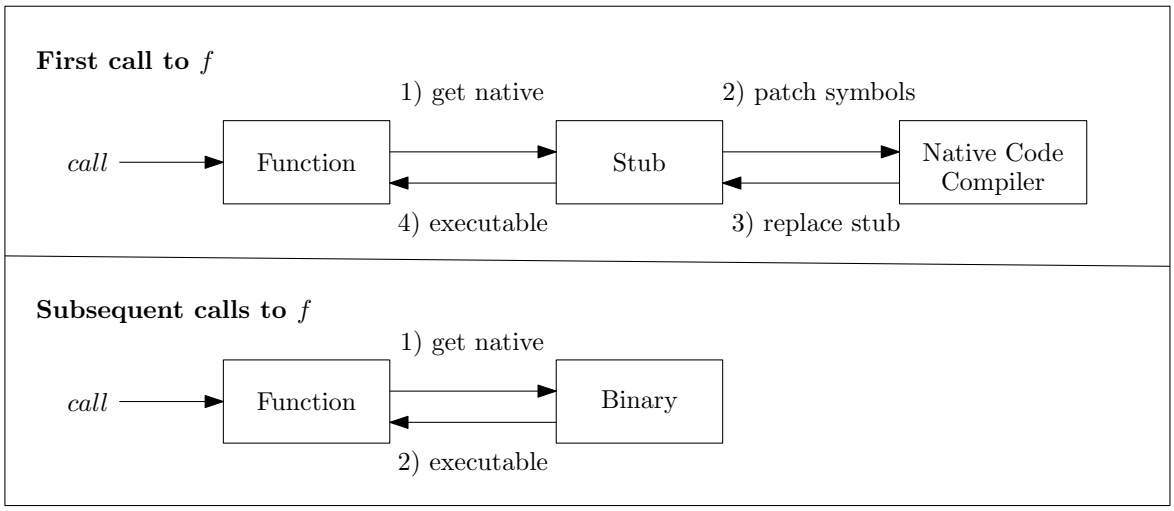


Fig. 5.3: Overview of the patching stage.

invoking the compiled executable.

In this section, we address the resolution of the indirections introduced in Chapter 3 into runtime addresses. The patching process is responsible for identifying the type of indirection based on the prefix (see Section 3.4). We outline the different types of indirections as follows:

1. **Direct References:** These indirections are replaced with the current runtime address (see Section 3.4.1).

2. **Hast References:** The reference is substituted with the corresponding runtime closure. The runtime maintains a hash map that maps from the hast value to the closure address.

3. **Pool References:** References that were relative to the serialized pool are transformed to reflect the deserialized pool. When the deserializer loads the serialized pool into the runtime, it is placed at a specific offset in the runtime pool. As a result, all references to the serialized pool need to incorporate this offset to obtain the final index.

5.3 L2 Dispatcher

In this section, we detail the second-level (L2) dispatcher; see Figure 5.4. When a deserialized binary is linked, a new L2 dispatcher is created for the first-level context in which the

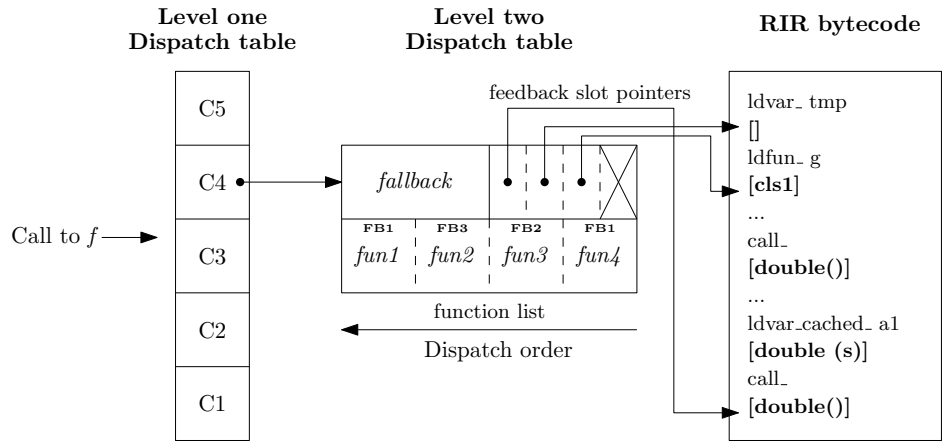


Fig. 5.4: Overview of the L2 Dispatcher.

binary was compiled. This L2 dispatch table is then inserted into a slot in the first-level dispatcher based on the first-level context of the binary. We maintain a fixed number of pointers to different feedback slots of the function, which are used to compare the second-level context of binaries in the L2 dispatcher. Our implementation of the L2 dispatcher consists of three main parts: (i) function list; (ii) feedback slot pointers; and (iii) fallback function. The function list is a dynamically growing array that stores the function binaries. This list is initially empty and gets populated as the dependencies of the serialized binaries are satisfied. The feedback slot pointer is a fixed-size list that contains pointers to the relevant feedback slots of the function. The fallback function is a placeholder that is initially empty but can store new JIT compilations in case a new runtime context is observed for the function. This allows the JIT to fall back to existing operations until a suitable run-time context that can be used for dispatching emerges.

We now describe the various operations that are performed on the L2 dispatch table:

(i) Create. A new L2 dispatch table is inserted into the existing first-level context dispatcher. When inserting the L2 dispatch table there are two possible cases: (a) The slot at which the L2 dispatcher is to be inserted is an empty slot; in this case, we create a new L2 dispatcher and set the fallback to a dummy function; (b) The slot at which the L2 dispatcher is to be inserted already contains a JIT-compiled function; in this case, we create a new L2 dispatcher and set the fallback to the existing function. The newly created L2 dispatch table is then inserted into the corresponding first-level context slot. Also, during the creation of a new L2 dispatcher, the relevant feedback slot pointers are also updated to point to the

relevant feedback slots in the function.

(ii) Insert. The insert operation inserts newly linked deserialized binaries into the function list whereas the JIT compiled functions are inserted as fallback function slot which is stored separately from the function list.

(iii) Dispatch. The dispatch operation iterates over the function list in the reverse order and dispatches to the first matched binary, i.e., the function feedback matches the run-time feedback and the function is not disabled (a function is disabled if a previous execution of the function resulted in a deoptimization).

(iv) Deoptimization policy. In case a binary dispatched by the L2 dispatcher deoptimizes we disable that binary and remove it from the function list.

5.4 Summary

This chapter provides an in-depth exploration of the deserializer phase of the implementation. The deserializer is a critical component of the system, responsible for converting serialized data into a format that can be processed by the system. To facilitate a thorough understanding of the deserializer phase, we begin by presenting a block diagram that illustrates its runtime operation and high-level operations. We then delve into the internal workings of the deserializer, where we discuss the general worklist, unlocking worklist, loading, and linking processes.

The general worklist is a critical component of the deserialization process, responsible for holding the mapping from function hash to the corresponding metadata. When the rir bytecode compiler encounters a function, it checks the general worklist to determine if there are any binaries available for deserialization. If the dependencies required for deserialization are satisfied, the function and its associated serialized pool are sent to the load/link process. The load/link process then takes care of loading the binaries into the runtime and inserting the deserialized binary into the function dispatch table. If the dependencies are not yet satisfied, the function and its associated serialized pool are put into the unlocking worklist. The unlocking worklist contains an inverse mapping from the dependency to the binary that needs to be deserialized. When all dependencies are satisfied, the unlocking process sends

the corresponding binary to the load/link process for deserialization. Once deserialized, the binary is added to the function dispatch table, making it available for use by the system.

The L2 dispatcher is a crucial component of the deserialization process, responsible for managing and dispatching the deserialized binaries to their respective execution contexts. It consists of three main components: the function list, pointer to feedback slots, and fallback slot for handling new contexts. This chapter provides a detailed exploration of the various operations involved in managing the L2 dispatcher, including creation, insertion, dispatch, and deoptimization policy. By carefully managing these operations, the L2 dispatcher ensures that the system can efficiently and effectively execute the deserialized binaries. Additionally, we examine the fallback slot and its importance in handling new contexts and preventing overgeneralization.

Overall, this chapter highlights the complexity and importance of the deserializer phase within the larger system architecture. It demonstrates the careful planning and execution required to ensure efficient and reliable deserialization and provides a foundation for future development and refinement of the system. Through our exploration of the deserializer phase, we hope to contribute to the broader discourse on system design and optimization and inspire further research in this area.

Chapter 6

Evaluation

Our primary goal is to preserve the benefits of just-in-time compilation while reducing the associated costs that come along with the same. JIT compilers often suffer from slow warmup times and unexpected slowdowns due to late-stage compilations. We evaluate this by comparing the performance profiles of programs with and without our scheme, focusing particularly on the time spent towards JIT compilation. On the other hand, the major challenge associated with a scheme that tries to reuse previous compilations is to obtain the additional advantages offered by the profiling feedback gained during runtime. We evaluate this goal by computing the impact with and without our two-level dispatch scheme. Further, as facilitating such a two-level contextual dispatch might lead to the problem of code bloat and context explosion, we show that our offline analysis passes can effectively get rid of the binaries that are functionally equivalent. Throughout the discussion, we also highlight several subtle aspects of our scheme with interesting case studies. Keeping in mind our stated goals, we next validate the research questions discussed in Section 1.2.

6.1 Experimental Setup

For our experiments, we use the benchmarks that come with Ř [5] and three real-world programs. **RMarkdown** is a document processing application that imports many R libraries for plotting and data analysis [12]. It is not computationally intensive but it does stress the compiler by loading substantial amounts of code. **Raytracer** is a more computational

benchmark with significant phase chages [13]. Lastly, we experiment with an R library from the CRAN repository, **Recommenderlab** [14], for which we run the library's examples included in it. Additionally, we use a unit-test generator called genthat [15] that allows us to demonstrate our handling of context explosion.

Experiments are run on a dedicated benchmark machine which features an i7-12700 CPU with speedstep and SMT disabled, stepping 2, microcode 0x1f, 16 GB of RAM, and Ubuntu 20.04 on 5.14.0-1051-OEM Linux Kernel. The experiments are built on Ubuntu 20.04.1 based containers and executed on Docker runtime 20.10.12. To reduce background noise, the setup uses the core shielding feature from cpuset to run the Docker containers in reserved CPU cores. We also store intermediate results in a 2 GB in-memory filesystem, in order to reduce I/O fluctuations.

6.2 Compilation Time and Peak Performance

In this section, we measure the per iteration speed-up of our implementation, $\check{\mathrm{R}}_{bc}$, named after the LLVM bitcodes in its code repository. We compare it against the default $\check{\mathrm{R}}$, which uses traditional contextual dispatch. In this experiment, a harness executed each benchmark program 15 times with predetermined inputs, enabling measurement of warmup and steady-state performance.

Figure 6.1 compares the per iteration times taken by $\check{\mathrm{R}}_{bc}$ and $\check{\mathrm{R}}$ in each of the 15 iterations. For brevity, we have chosen a subset of programs that represent all kinds of interesting cases from the RBenchmarking suite. In general, $\check{\mathrm{R}}_{bc}$ is much faster in the first iterations than baseline $\check{\mathrm{R}}$. We attribute this efficiency to (i) the reduced number of compilations; (ii) the reduction in the number of bitcode binaries to be loaded (due to OBAP); (iii) additional LLVM passes that we run offline to optimize the serialized binaries; and (iv) an efficient two-level dispatch implementation. We now discuss the kinds of warmup improvements observed for different classes of programs:

1. *Significantly faster warmup*: In programs such as binarytrees, flexclust_no_s4, knucleotide, pidigits, and spectralnorm, where computation is not limited within one hot loop, the deserializer is able to incrementally link binaries as they become available,

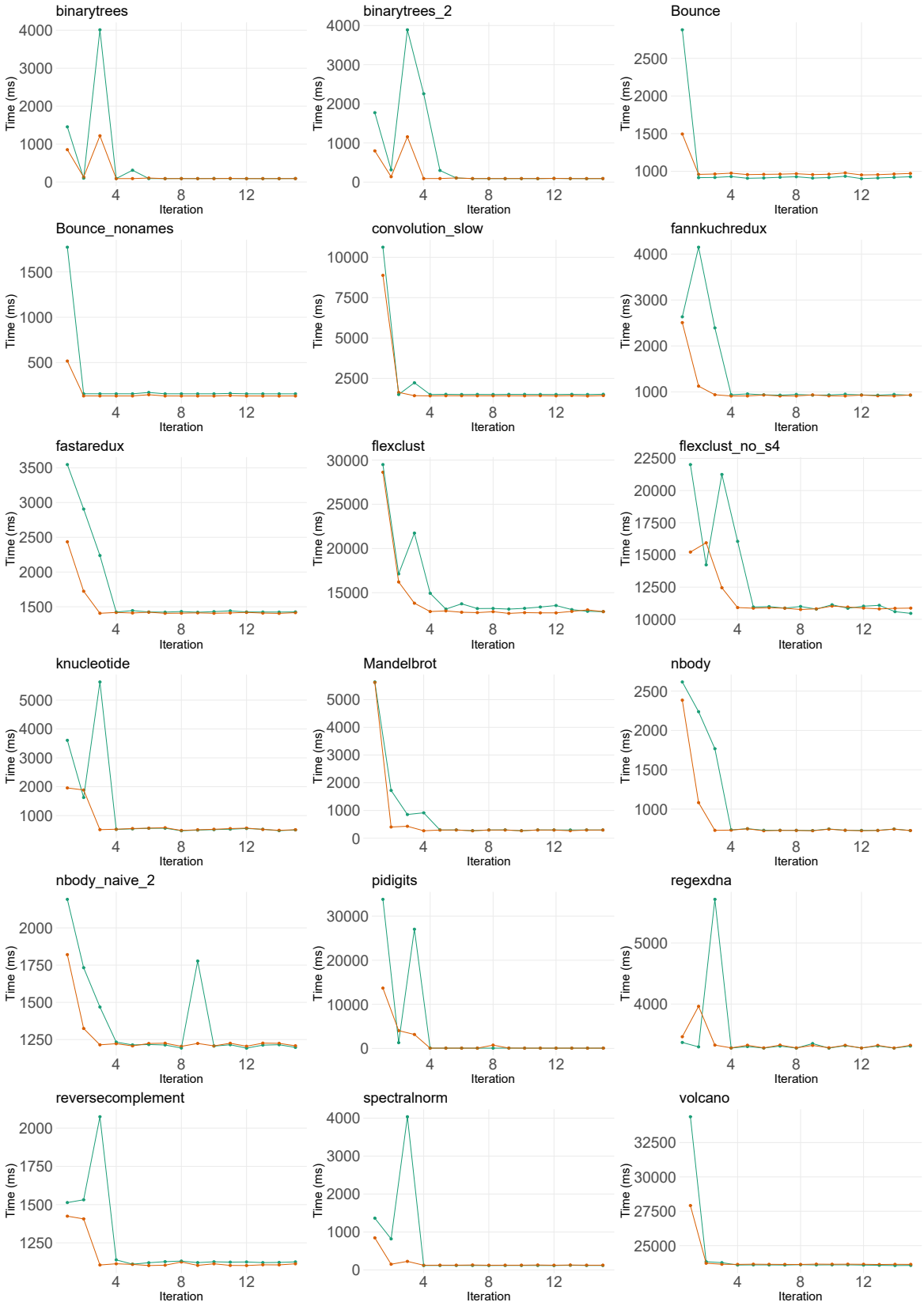


Fig. 6.1: Figure comparing $\check{\mathrm{R}}_{bc}$ (red line) with $\check{\mathrm{R}}$ (green line) for the first 15 iterations of benchmark programs. A representative subset of 18 programs from the RBenchmarking suite are shown.

thus leading to improved runtimes starting from the first iteration itself.

2. *One iteration warmup*: In programs such as fannkuchredux and Mandelbrot, peak performance is dependent on a few critical functions that only get linked after the first iteration, due to which the serialized binaries for those functions can only be executed starting the next iteration. Consequently, though $\check{\text{R}}_{bc}$ starts similar to $\check{\text{R}}$ (that is, in the interpreter), it quickly reaches a steady state as soon as the linking dependencies are satisfied.

3. *No recompilation pauses*: In nbody_naive_2, $\check{\text{R}}$ triggers additional compilation to occur at iteration 9. This is based on a non-deterministic JIT heuristic that performs these compilations based on the execution time of those functions. On the other hand, as the corresponding binary is already available as well as ready for linking, $\check{\text{R}}_{bc}$ is able to avoid the slowdown and immediately able to dispatch to the same.

4. *Almost zero compilation*: In numerous programs, the number of compilations have been reduced to zero; while the remaining compilations have become a fraction of their previous values (see Figure 6.2).

5. *Peak performance is preserved*: All the benchmarks show that the peak of $\check{\text{R}}$ is preserved $\check{\text{R}}_{bc}$.

To better understand the warmup behavior, we also measured the JIT compilation overhead in isolation, which is shown in Figure 6.2. For $\check{\text{R}}$ this was the total time compiling functions, while for $\check{\text{R}}_{bc}$ this was the time spent compiling plus time spent linking and loading bitcode from the cache. We see that $\check{\text{R}}_{bc}$ is consistently better than $\check{\text{R}}$ for all the benchmarks, with an average improvement of 3.38×. The primary reason for a less reduction in compile time in flexclust and rmarkdown (discussed in Section 6.2.1) is because of (i) anonymous R functions cannot be cached unless their enclosing scope is resolved first (this is due to a limitation of $\check{\text{R}}$) (ii) new compilation contexts that were not compiled before.

Benchmark	JIT Overhead (ms)		Number of Compilations		OBAP Results		
	Ř	$\check{\mathbf{R}}_{bc}$	Ř	$\check{\mathbf{R}}_{bc}$	Time Spent (ms)	Binary Reduction (%)	Slot Selection (%)
binarytrees	5462	1880	18	2	250	0	3.1
binarytrees_2	7396	1763	19	0	280	0	3.9
bounce	1936	511	7	0	190	0	0
bounce_nonames	1598	367	7	0	180	0	0
convolution_slow	1836	596	8	0	200	0	0
fannkuchredux	2116	430	3	0	170	0	0
fastaredux	3155	907	8	0	200	0	0
flexclust	100885	56518	404	175	2080	41.8	0.5
flexclust_no_s4	34675	14035	116	8	870	36.3	0.7
knucleotide	9076	2750	49	1	400	50.0	0
mandelbrot	1395	259	8	0	180	0	0
mandelbrot_ascii	1560	583	5	0	180	0	0
nbody	3982	1764	13	2	260	0	50.0
nbody_naive_2	2205	643	6	0	190	0	0
pidigits	60827	21279	102	15	1270	28.6	1.7
regexdna	2495	718	7	0	200	0	0
reversecomplement	1358	453	5	0	170	0	0
rmarkdown	496691	286584	1501	489	3090	48	0.8
spectralnorm	5014	409	7	0	180	0	0
volcano	7786	2232	20	0	270	0	0

Fig. 6.2: Table showing the reduction in compilation and OBAP statistics; JIT overhead is a sum of compilation time, deserializer load/link time and LLVM bitcode to machine code generation time (in case of normal Ř the deserializer times are zero).

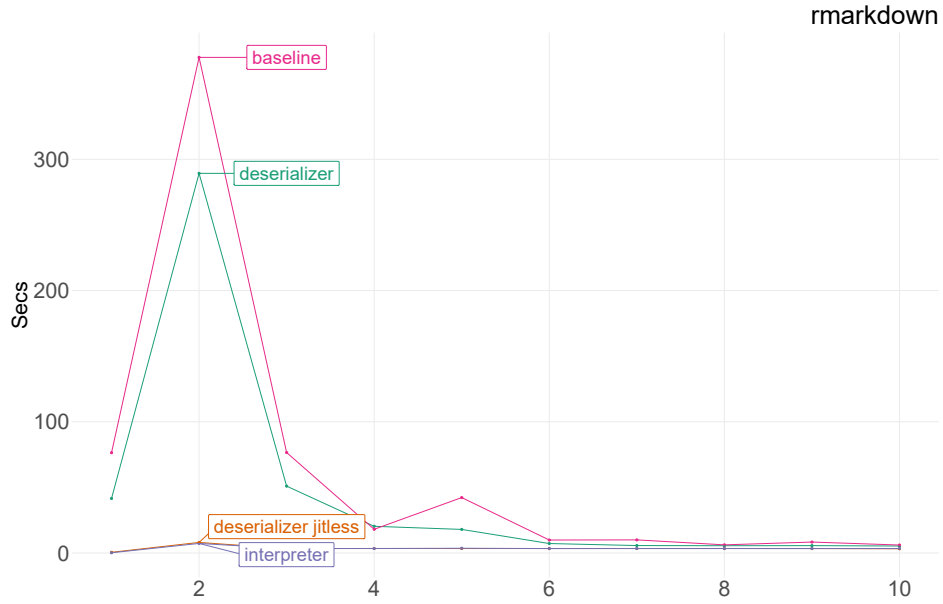


Fig. 6.3: Real-world performance

6.2.1 Real-world performance

The previous experiment addressed the performance of benchmark programs, this experiment looks at a real-world application, RMarkdown. We can provide additional data points for **RQ(1, 2, 3)**.

The repository is built with one run of RMarkdown in serializer mode. This yields 1,384 binaries which are reduced down to 1,091 versions (a 48% reduction, see Figure 6.2).

Figure 6.3 shows the wall-clock time for iterations of RMarkdown under several configurations: interpreter (ŘÌ with interpreter only), baseline (ŘÌ with compiler), deserializer ($\check{\text{R}}_{bc}$ with compiler), and deserializer jitless ($\check{\text{R}}_{bc}$ jitless).

RMarkdown is an I/O intensive application with little computation in R. The interpreter is quite fast here. This is a common scenario where the only hope for improvement is to alleviate the warmup costs that will not be amortized later on. We see the ŘÌ incurs a massive 380 seconds pause and takes eight iterations to reach steady state. $\check{\text{R}}_{bc}$ has better warmup times and reach steady state faster. $\check{\text{R}}_{bc}$ in jitless mode is almost matching the interpreter.

6.2.2 End-to-end performance

Our previous experiments were best cases scenarios in which the code being run is identical to the code observed when serializing. This experiment explores the impact on the performance of version skew. Imagine that we want to test a compiler exhaustively. In the R ecosystem, one could run the compiler on the extensive test suite that comes with each package. Unfortunately, this is also a worst-case scenario for a just-in-time compiler. With Ř we have observed that a single package can take tens of minutes to run through its test suite. Even worse, each new release of the package requires running everything from scratch.

With $\check{\text{R}}_{bc}$ one would like to pay the cost of building a repository once, and then, when there is a new release of the package, only compile the functions that were changed. This experiment is set up to evaluate this use case.

We checked out Recommenderlab from its Git repository at six different points over two years. Thus obtaining different, but related releases. We built the repository over the first check out, and then used the same repository to test all the releases. We expect that, since the code bases diverge slowly, the benefits of our approach may decrease. Figure 6.4 shows the wall-clock time for one run of the test suite (red for Ř and green for $\check{\text{R}}_{bc}$). The results are encouraging. While $\check{\text{R}}_{bc}$ still performs compilation on every run, it reduces the overhead over Ř by almost a third. Over time, as the code of each release diverges, $\check{\text{R}}_{bc}$ spends slightly more time compiling.

6.2.3 Performance under context explosion

In order to measure the impact of OBAP in cases where the code repository can grow exponentially, we use a unit test generation tool called genthat [15] which uses various vignettes and code examples to generate a large variety of tests for a given library. We use genthat to generate 21475 unit tests for 8 popular R libraries spanning over 900000 lines of code. Along with this we also include a rmarkdown based program that makes use of other popular data science libraries. We first run all these tests to obtain our initial code repository, following which we again run the same tests in a feedback loop to capture even more contexts. This results in a code repository of 11137 different binaries for a total of

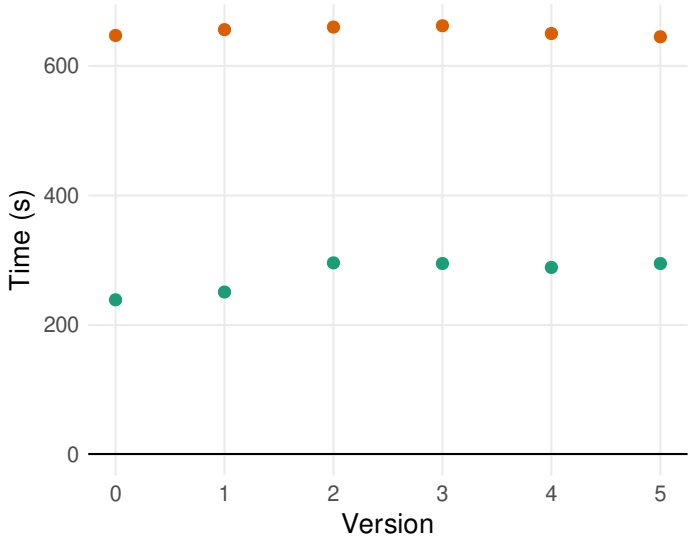


Fig. 6.4: Version skew (Ř in red, Ř_{bc} with compiler in green)

909 parent functions. We then run our OBAP pass over the obtained code repository, which reduces the size of the repository by 80% to finally include only 2794 unique functional binaries in a total time of 61 seconds. We also find that 91% of the binaries were contextually compiled versions of existing functions in the repository. The number of L2 collisions is only 2.2% of the total number of binaries processed, meaning that the second-level context is adequately able to handle a large number of binaries. We perform the following runs of rmarkdown program (i) **deserializer** using the repository created from a single serializer run of the same program (ii) **genthat** using the repository generated by the genthat unit tests, (iii) **baseline** the default Ř implementation; and plot the overall runtime for the same (see Figure 6.5). As can be seen, the genthat repository consistently outperforms the case where there was just one feedback run.

In a nutshell, we see that our approach is quite suitable for large programs and efficiently handles context explosion without losing performance. In Section 6.3, we study this phenomenon further by running serialization runs in a feedback loop, in order to observe how a steady state in terms of the number of compilations is reached over repeated executions of the same program.

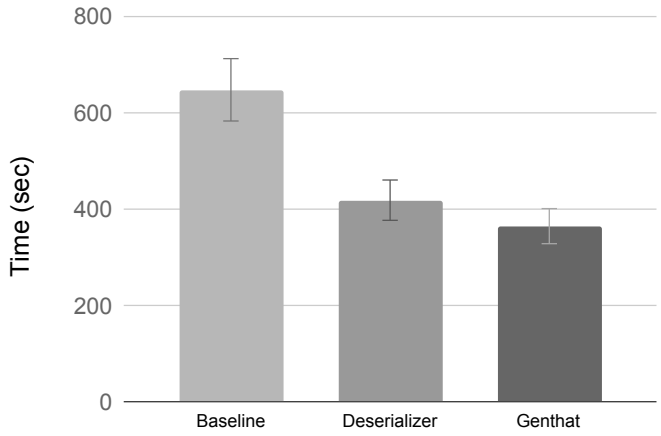


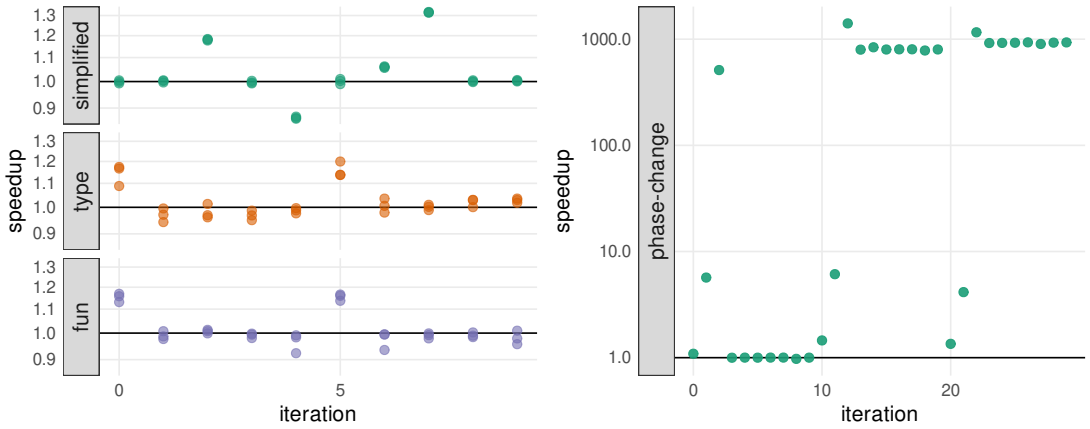
Fig. 6.5: Speedups when using a large code repository.

6.2.4 Phase change behaviour

Apart from warmup performance, our scheme is potentially beneficial in programs with phase change behavior. A phase change happens when a program, after continuing to call a function in a given context for some time, shifts to a new kind of runtime context. In a traditional JIT compiler such as Ř, phase change triggers a deoptimization event that leads to recompilation in a more generalized context, one that reflects the union of the new type feedback with the previous ones. Such recompilations may cause an unexpected slowdown in the runtime as well as degrade the steady-state performance, due to overgeneralization. We now discuss how $\check{\text{R}}_{bc}$ handles each of these possibilities.

Figure 6.6a compares $\check{\text{R}}_{bc}$ and Ř when running three different versions of an R ray tracer [13]. It shows the speedup that $\check{\text{R}}_{bc}$ exhibits over Ř. At iteration 5 we introduce a phase change. In the *simplified* and in the *type* variants, we change the type of the height map used by the algorithm. In the *fun* variant we change the interpolation technique. As can be seen in the first iteration after the phase change, Ř must recompile the new program while $\check{\text{R}}_{bc}$ can get away with loading it from the cache, which is much faster. This can be seen as an increase in a speedup of $\check{\text{R}}_{bc}$ over Ř in the iteration right after a phase change. Afterward, both versions of the compiler return to a steady state and have the same performance.

To show how $\check{\text{R}}_{bc}$ handles overgeneralization during recompilation events, consider the code snippet in Figure 6.7. We call a function `foo`, which in turn must look up the defi-



(a) (b)

Fig. 6.6: (a) Ray-tracings with recompilation-induced phase change at iteration 5. (b) Performance degradation caused by over generalization after phase change.

```
1  foo <- function(a,b) {
2    res=0
3    for (i in 1:100000) {
4      res = res
5        + app1(a,1)
6        + app2(1,b) }
7    res
8  }
9  # C1
10 app1 = function(a,b) {
11   bitwShiftL(a,b) }
12 app2 = app1
13 foo(60,2) # 10 times
14 # C2
15 app2 = function(a,b) {
16   a = as.integer(a)
17   bitwShiftL(a,b) }
18 foo(60,2) # 10 times
19 # C3
20 app1 = function(a,b) {
21   a = as.integer(a)
22   bitwShiftR(a+5,b) }
23 app2 = function(a,b) {
24   b = as.integer(b)
25   bitwShiftR(a+6,b) }
26 foo(60,2) # 10 times
```

Fig. 6.7: Code snippet to demonstrate slowdowns due to overgeneralization of contexts.

nitions of `app1` and `app2` from the environment. During the first 10 invocations of `foo`, functions `app1` and `app2` are bound under context C1, as shown at lines 9-12. In the 11th iteration we redefine `app2`, replacing it with a polymorphic version of the same function. Finally, in the 21st iteration `app1` becomes polymorphic as well. In Ř these phase changes lead to degraded performance, as shown in Figure 6.6b. On the other hand, L2 dispatch in $\check{R}_{bc}$ is able to contextually separate the different call-site targets and select a more specialized binary for the new contexts at iterations 12 and 22, subsequently leading to highly performant runs that are not marred by overgeneralization.

6.3 Iterative Serialization

Notwithstanding the improvements in JIT compilation time shown above, it is possible that some functions in a program get called in newer contexts in subsequent runs of the program. In order to demonstrate this aspect, Figure 6.2 (Column 3) shows the number of compilations under Ř and $\check{R}_{bc}$ for all the programs under consideration. As can be seen, there are non-zero compilations for a few programs even in $\check{R}_{bc}$. Our investigation into the causes of these compilations led to multiple interesting consequences. Firstly, recall from Section 3.3 that anonymous functions cannot be linked until their enclosing functions have been compiled, which leads to compilations in the initial runs of a program. More interestingly, we have designed $\check{R}_{bc}$ in a way that it is possible to create a "serialized-bitcode repository" iteratively; that is, keep running the serializer in a feedback loop while executing deserialized bitcodes obtained previously. This allows us to "train" our system in a way that it keeps learning new compilations, while OBAP keeps reducing redundancies and the L2 dispatcher ensures picking up the most relevant binaries in the next run. Figure 6.8 shows how the number of new compilations reduces after multiple serializer iterations, for three RBenchmarking programs from Figure 6.2 that had leftover compilations even in the deserializer run. The more we train our serializer, the number of new compilations reduces gradually and finally reaches a near-steady state. Similarly, we can see that the number of new contexts serialized in each run reduces steadily. We also observed zero compilations and serializations at steady state for most programs that had non-zero compilations in Figure 6.2.

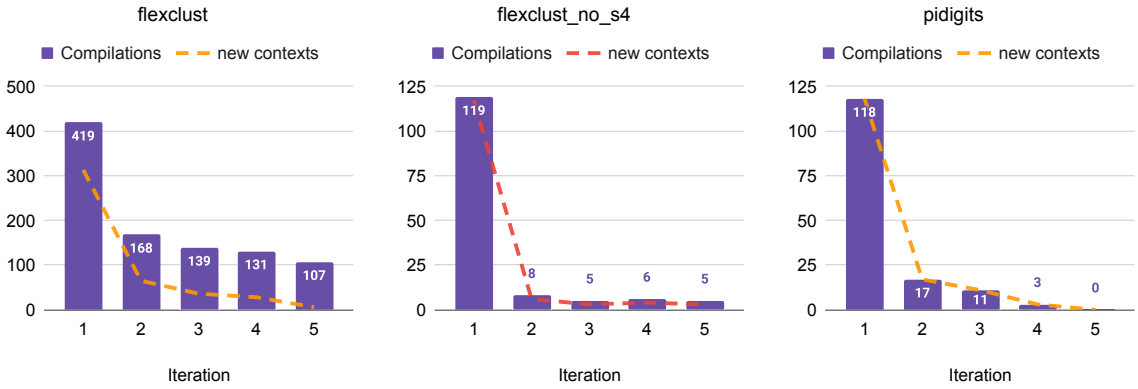


Fig. 6.8: Figure showing the emergence of new contexts under iterative feedback.

For the flexclust program, where the number of compilations did not reduce to zero even after a large number of iterations, we traced down the reasons in the base JIT compiler. Ř is currently used exclusively for research purposes and, as such, has unresolved corner cases. Flexclust hit two of those. The first one is that compiled functions expect their arguments in a fixed order. If the call site ordering does not match its definition, e.g. for named calls, the caller is responsible for reordering the arguments before the invocation. When the callee is statically known, the caller's code includes the instructions to do so. Otherwise, it falls back to the target's baseline (interpreter) version and its compilation is skipped, potentially forever. Now, a callee running a loop in the interpreter can decide to perform on-stack-replacement and jump out to the newly compiled version (discarded after exiting). This compilation might also trigger further compilation of new callees (e.g. for inlining). Finally, if the initial callee is itself invoked within an outer loop, each iteration is bound to execute its baseline version, triggering the recompilation chain just described. The second issue is that Ř does not provide good support for the R S4 object system and will sometimes fail to compile functions that perform S4 dispatch, even to a baseline version. Because of that, its inner functions get instantiated and compiled on every invocation and the system never stabilizes. This problem resonates with the limitation for serializing anonymous functions described above.

We expect that the end goal of having zero compilations for all programs would also allow us to further improve the Ř compiler itself. Thus, we have found that the approach proposed in this thesis, apart from its intended goals of obtaining performant reusable R

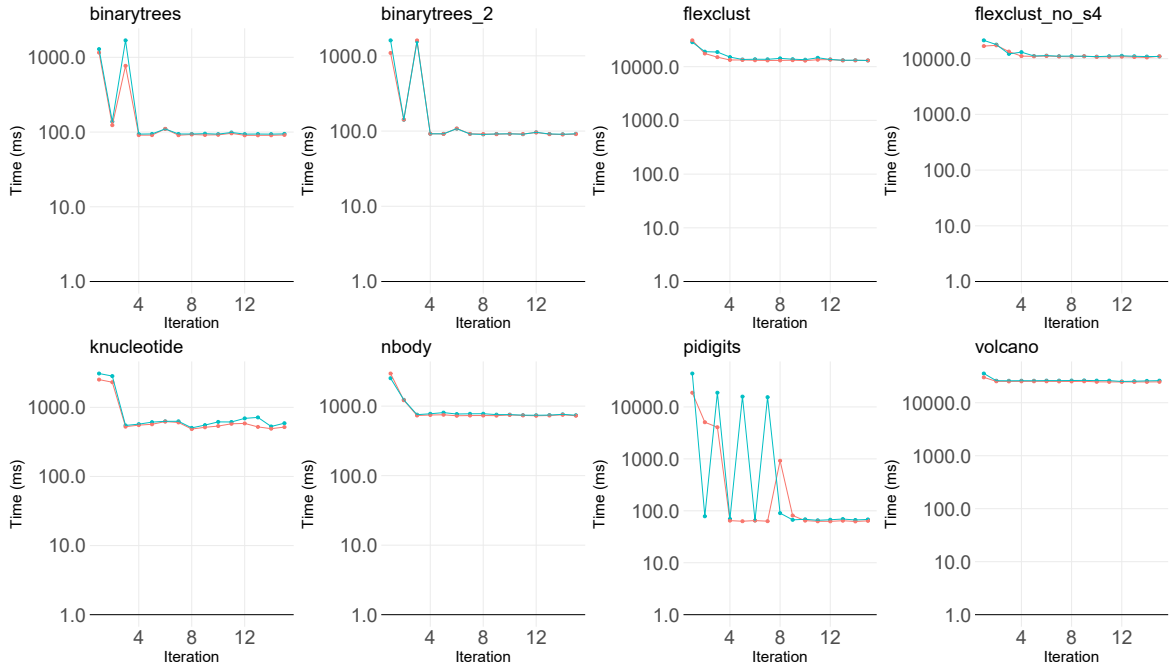


Fig. 6.9: Relative performance of the last-seen strategy (blue line) vis-à-vis $\check{\text{R}}_{bc}$ (red line).

binaries with minimal JIT cost, also helps us discover interesting corner cases and debug the same in a dynamic JIT compiler.

Overall, based on the observations in the last three sections, we see that our proposed scheme leads to significant improvements in the time spent in JIT compilation (including a reduction in warmups as well as phase-change times) while preserving (and in some cases, improving) the performance of the underlying programs. In the next section, we elaborate on the impact of the various components of our approach, viz. the two-level dispatch mechanism as well as the OBAP pass.

6.4 Impact of OBAP and Two-Level Dispatch

In order to target fast compilation, typical JIT compilers maintain only one binary per compiled function [16, 17]. Ř, on the other hand, maintains multiple binaries each with different assumptions about arguments at call-sites. Going one step further, in this thesis we have proposed an approach that tries to use the learnings from different runs of a program in optimizing the same for later runs; we accomplish this using a first-of-its-kind two-level dispatch based on both call-site assumptions (first-level context) and recorded runtime feed-

back (second-level context). In order to address the potential overheads of such a scheme, our OBAP pass reduces code bloat and determines a greedily optimal number of slots that need to be compared in the second-level dispatch. In this section, we show that this OBAP supported two-level dispatch mechanism is capable of as well as necessary for preserving performance and minimizing recompilations.

The last three columns in Figure 6.2 show the characteristics of our OBAP pass. We see that for the larger benchmark programs (those that incur a significant amount of JIT compilation, such as flexclust, pidigits, and rmarkdown), the OBAP pass is able to identify and eliminate functionally redundant binaries. Similarly, the slot selection algorithm is successful in identifying a very small number of slots that are representative of the second-level context. For programs with less or negligible reduction (such as bounce and fastaredux), we found that there was limited scope due to a very small number of compilations and there were no redundant binaries or slots in the first place. We also observe that the time taken to perform OBAP is in the order of a few hundred milliseconds, which, considering its offline nature, is very reasonable. In a nutshell, we note that our OBAP phase is quite effective in supporting a two-level dispatch scheme for preserving JIT benefits in large programs. We next evaluate what would happen if instead of OBAP and two-level dispatch, we were to use a naive serialization strategy that only dispatches based on the first-level context.

In case we were to dispatch based on only the first-level context, we need to identify a representative binary for each context. This is an interesting problem: If we were conservative and picked a very generic binary, it might get dispatched in most contexts but end up losing performance. Whereas if we were ambitious and chose to pick a very specialized binary, we might achieve good performance in compatible contexts but introduce a large number of compilations in others. We tried to experiment with this fact by choosing the last seen second-level binary for a given first-level context (that is, effectively disabling L2), as in a single run of the serializer, this binary would have been obtained with the most amount of feedback for a given first-level context. Figure 6.9 shows the per iteration performance of this last-seen approach, relative to $\check{\mathrm{R}}_{bc}$, for the programs for which $\check{\mathrm{R}}_{bc}$ reduces binaries or slots as seen in Figure 6.2. Notably, though selecting a good single binary gives steady-state performance similar to $\check{\mathrm{R}}_{bc}$ in many cases (due to high specialization), it fails to suit

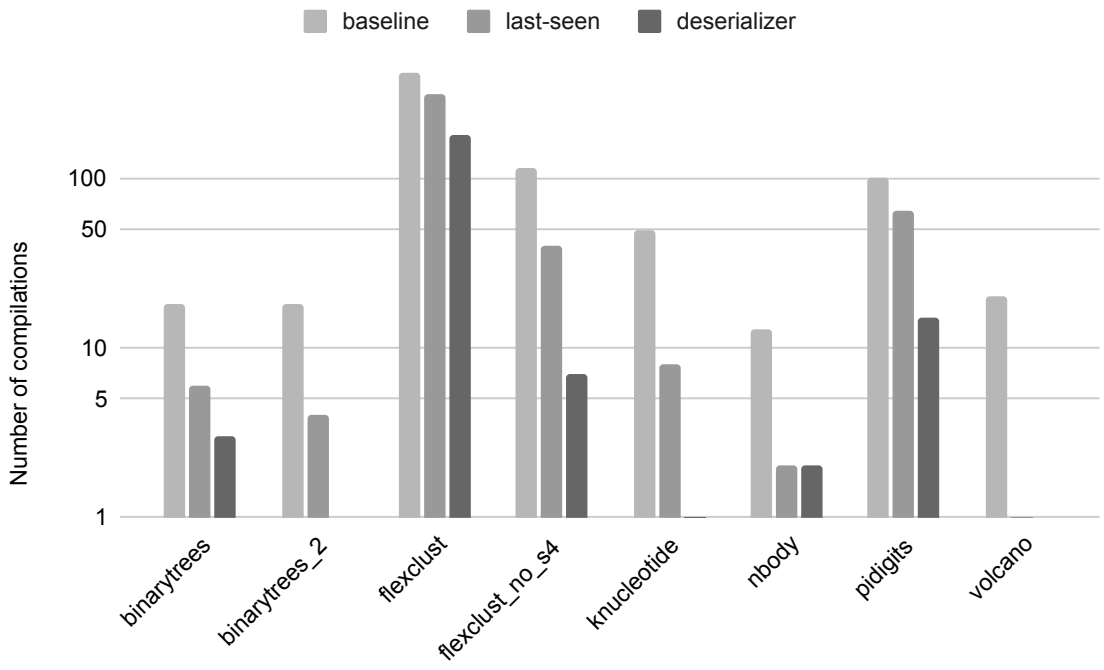


Fig. 6.10: Number of compilations when using only last-seen strategy.

all kinds of contexts and leads to recompilation spikes as visible for pidigits, where the runtime context changes constantly. In order to further validate our hypothesis, we counted the number of compilations under this approach vis-à-vis $\check{\text{R}}_{bc}$ for the programs of interest; see Figure 6.10. It is clear that the alternative approach leads to a large number of compilations for the programs in which it performs worse (e.g. pidigits). Combining this observation with the fact that a two-level dispatch helps avoid the bad effects of overgeneralization (as seen in Section 6.2), we conclude that L2 dispatch as performed by $\check{\text{R}}_{bc}$ is an apt way of replicating the performance of a traditional JIT system while scaling it to different runtime contexts.

6.5 Discussion

The techniques discussed in this thesis are designed to address a scenario where the repository can be built from various users of R, running different systems and hardware, with the aim of collecting as much contextual compilation data as possible. Our approach assumes that a function has a finite number of ways it is actually used, and over time, we can gather all these cases in our repository. In this section, we further explore the applicability

of our techniques in different scenarios, along with their respective pros and cons.

Performance improvements when using precompiled binaries over IR? Our current implementation utilizes platform-independent LLVM bitcodes to ensure they can be collected from different users running different hardware, yet remain available for future use. However, if we target a specific platform, we can easily compile these bitcodes offline into object files using the LLVM llc static compiler. During this compilation process, all the references are replaced with loads from the global object table and the function relocation table. The entire deserializer process can then remain intact, accepting these object files as they are. Our experiments have shown that using object files instead of bitcodes results in a warmup improvement of 9× (over 3.38× when using bitcodes). The ability to compile bitcodes offline can be viewed as a simple user choice that can further enhance the overall user experience.

What happens when some contexts are missing? Due to the non-deterministic nature of Ř, it is not always possible to eliminate all compilations using our techniques. This limitation arises from both the inherent constraints of Ř itself and our specific implementation. However, as demonstrated in our end-to-end performance evaluation (refer to Section 6.2.2), even in scenarios where not all contexts are available, a significant portion of the repository still gets utilized whenever possible. In cases where some contexts are missing, the JIT compiler simply recompiles the corresponding functions. However, if the JIT is disabled, we might experience a performance degradation in terms of peak performance when certain contexts are absent.

6.6 Summary

This chapter presents an evaluation of our three-stage process on standard benchmarks and real-world programs. We begin by describing our methodology and experimental setup. Then, we compare our system's performance against the existing Ř implementation in terms of compilation times and peak performance behaviors. To demonstrate the performance improvement of our approach, we showcase representative benchmarks selected from the RBenchmarking suite. Our findings reveal that warmup times are significantly reduced, while peak performance is maintained. We achieve an improvement of 3.38× in compilation

times, and we also identify interesting behaviors such as avoidance of late-stage compilations due to the binary already existing in the deserializer stage. Overall, our approach saves significant time in the warmup phase and provides an all-around speedup.

Next, we evaluate our approach when running a real-world program, end-to-end performance when code changes over time and during context explosion. We see that our approach is able to handle large real-world programs and provide improvements as seen previously. The end-to-end performance results show that large parts of the repository can be reused even when the code base is changing over time. In measure performance of OBAP under context explosion we use genthat to generate unit tests for popular R libraries. The OBAP phase efficiently manages the large repository, and our results show an improvement in per-iteration times for the Rmarkdown program.

We also investigate the phase change behavior for two programs: (i) a known ray-tracing implementation and (ii) a synthetic program where multiple phase changes are induced. Our system is easily able to handle these cases.

To understand the workings of serialization and deserialization together, we perform iteration serialization where we update the repository after a program's run. In all of these runs, both the serializer and deserializer are on. This helped us discover existing bugs in the compiler, and we found that for most programs, the number of new contexts under this scheme steadily reduces.

Finally, we compare our strategy to the naive last-seen strategy. We find that the L2 dispatch performs much better in programs that are highly dynamic and create a large number of contexts at runtime.

Chapter 7

Related Work

Caching compilation artifacts or profile information to speed up JIT warmup is not a new idea and there is a large number of publications in this space. In this section, we discuss some of this previous work and compare it to our contextual dispatching technique.

One of the first JIT compilers to cache its compilation results was the Quicksilver compiler [18, 19]. They developed ways to patch the cached binaries, in order to update the external references within. They also described how to invalidate the cache if the method in question has been redefined, or if any of the methods it inlines were redefined. These patching and cache-invalidation concerns are unavoidable and must be answered by any JIT compiler that wants to do this sort of caching. One of the main differences between Quicksilver and our work is that we compile multiple contextual versions of each method.

ShareJIT [20] is a mechanism in the Android Runtime, which caches JIT-produced executables. They faced the problem of choosing between compiling multiple specialized versions for each method or a general one-size-fits-all version. While in our work we have focused on keeping as many versions around as possible, ShareJIT often chose in the opposite direction. Mobile devices have limited storage capacity and reducing the size of the code cache was paramount. They even chose to disable some compiler optimizations, such as method inlining, because it worked against the sharing of compilation artifacts.

Some publications have focused on caching profile data instead of compilation artifacts. Although this does not save compilation time, it is simpler and requires less disk space. It also sidesteps questions such as how to deal with variations in the operating system environ-

ment, or security concerns if the cache is shared among multiple users. One of the earliest works to cache profile data was that of Arnold, Welc, and Raja for the J9 Java VM [21]. They used the profile data to allow the JIT to immediately compile methods it expects will be hot, without having to wait to collect the profile data again. Similar to our work, they also had to decide what to do when different runs produced different profile data. Their answer was to combine the profile data into a single profile for each program. However, that resulted in suboptimal peak performance when programs were trained on bimodal inputs. Our approach does not suffer in terms of peak performance like this, because we do not merge the profile data collected across runs.

Another work that cached profile data was Ottoni and Liu's Jump Start system for HHVM [22]. Their target language was Hack, a dialect of PHP, which like R is a highly dynamic language. Therefore, similarly to us, they had to record a good deal of run-time-type feedback, something that the Java VMs do not have to do as much. However, unlike us, they only store a single profile for each function. Their use case assumes that the VM powers a fleet of web servers that are all running the exact same workload.

Finally, we note two papers that discuss caching IR versus caching executables. In their first attempt at a code-caching JIT for Javascript, Zhuykov et al. serialized the executables [23]. They were disappointed to not see any speedups from the caching output of the baseline level of the JIT. They found that patching the code from the cache took just as long as compiling it from scratch because the baseline JIT has fewer optimizations and runs quite fast already. For their second attempt, they switched gears and decided to serialize the compiler IR instead of the final binary [24]. This choice was largely because patching IR is easier than patching native binaries. In our work, we also cache the IR, although our motivation stems from unlocking the analysis to identify redundant binaries. There are situations where we are able to identify two equivalent programs by looking at the IR, which would be harder to do by looking only at the final executable. Another difference between Zhuikov's work and ours is, again, that we record multiple versions for each function and pick the best one at run-time; they only remember the last version encountered.

Chapter 8

Conclusion and Future Work

Just-in-time (JIT) compilers optimize programs by learning from their live history during execution. They can observe the arguments at call sites, gather information about the types flowing into various expressions, remember call targets, and specialize a function based on speculations made with the feedback collected thus far. These compilers are capable of recovering from wrong assumptions, by deoptimizing and recompiling to accord a more generic behaviour. All this makes a JIT-compiled program performant, but at the cost of significant time spent in making these decisions and recompiling functions to suit the current context; which is decided by call-site assumptions and feedback information. This cost is often witnessed in the form of high warmups that degrade user experience, and sudden slowdowns during deoptimization and recompilation.

There have been attempts to reduce the cost brought in by JIT compilation by performing ahead-of-time compilation. However, it rarely helps for dynamic and lazy languages where the amount of information known statically is incapable of allowing meaningful optimizations. In this thesis, we offer a novel view of JIT optimizations as ones that can be derived based on the insights gained not only during the current run but from a series of runs, and not only for one program but also from uses of shared libraries in different programs. We thus propose a scheme that records information about various JIT-compiled functions across different call-site and feedback contexts, and efficiently reuses these contextually specialized JIT binaries in future executions.

An important challenge with a scheme that generates a plethora of contextual binaries

for various functions is to restrict code explosion by identifying binaries that perform similar amount of computation, as well as parts of the contexts across these binaries that express redundant specialization. We do all this before allowing our binaries to be reused – offline – and provide a novel two-level dispatch mechanism that selects the most relevant binary based on the runtime context during execution.

In the offline analysis, we also identify a small set of relevant feedback slots for indexing the leftover binaries, which our dispatcher later makes use of for a fast contextual lookup. Our evaluation over an optimizing R compiler shows that we reduce warmup times significantly, handle phase changes during deoptimization efficiently, and overall enable a performance similar to a fully blown JIT system at a much smaller associated cost.

While the prototype implementation explored here is for the R language, our approach may extend to other dynamic languages. The main issue that should be investigated is whether the cost of speculative contextual dispatch is acceptable in language with lighter-weight functions. In R, function calls are extremely expensive, and each function tends to do a lot of work. So the dispatching overhead can be amortized easily. It would be interesting to reproduce our experiments in the context of a language such as JavaScript or Python.

In future, it remains to be seen how our approach integrates with a tiered VM architecture comprising both an interpreter and a compiler. In particular, we would like to develop a tiered strategy that uses an interpreter for fast execution of the cold regions of a program, and switches to a historically learnt efficient binary for the portions deemed suitable to be tweaked for performance. We also look forward to the community extending and applying our proposed techniques to develop JIT strategies for even more runtimes, and for even more exciting programming languages.

References

[1] G. Krylov, G. W. Dueck, K. B. Kent, D. Maier, and I. D'Souza, "Ahead-of-Time Compilation in OMR: Overview and First Steps," in *Proceedings of the 29th Annual International Conference on Computer Science and Software Engineering*, ser. CASCON '19. USA: IBM Corp., 2019, p. 299–304.

[2] M. Thom, G. W. Dueck, K. Kent, and D. Maier, "A Survey of Ahead-of-Time Technologies in Dynamic Language Environments," in *Proceedings of the 28th Annual International Conference on Computer Science and Software Engineering*, ser. CASCON '18. USA: IBM Corp., 2018, p. 275–281.

[3] O. Flückiger, G. Chari, J. Ječmen, M.-H. Yee, J. Hain, and J. Vitek, "R Melts Brains: An IR for First-Class Environments and Lazy Effectful Arguments," in *Proceedings of the 15th ACM SIGPLAN International Symposium on Dynamic Languages*, ser. DLS 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 55–66. [Online]. Available: https://doi.org/10.1145/3359619.3359744

[4] O. Flückiger, G. Chari, M.-H. Yee, J. Ječmen, J. Hain, and J. Vitek, "Contextual Dispatch for Function Specialization," *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, nov 2020. [Online]. Available: https://doi.org/10.1145/3428288

[5] RBenchmarking, "RBenchmarking," https://github.com/reactorlabs/RBenchmarking, 2018, [Online; accessed 6-September-2023].

[6] L. Tierney, "A Byte Code Compiler for R," http://www.stat.uiowa.edu/ luke/R/compiler/compiler.pdf, 2019.

[7] L. Stadler, A. Welc, C. Humer, and M. Jordan, "Optimizing R Language Execution via Aggressive Speculation," *SIGPLAN Not.*, vol. 52, no. 2, p. 84–95, nov 2016. [Online]. Available: https://doi.org/10.1145/3093334.2989236

[8] T. Kalibera, P. Maj, F. Morandat, and J. Vitek, "A Fast Abstract Syntax Tree Interpreter for R," *SIGPLAN Not.*, vol. 49, no. 7, p. 89–102, mar 2014. [Online]. Available: https://doi.org/10.1145/2674025.2576205

[9] Microsoft and R. C. Team, *Microsoft R Open*, Microsoft, Redmond, Washington, 2017. [Online]. Available: https://mran.microsoft.com/

[10] M. K. Mehta, "RSH WIZ. (2021)," https://github.com/CompL-Research/rsh-wiz.git, 2021, [Online; accessed 19-February-2023].

[11] D. J. Bernstein, "djb2 hash function," http://www.cse.yorku.ca/~oz/hash.html, 1991, [Online; accessed 6-September-2022].

[12] M. Love, R. Irizarry, V. Carey *et al.*, "Genomicsclass/labs: RMD source files for the Harvardx series ph525x. (2013)," https://github.com/genomicsclass/labs, 2013, [Online; accessed 6-September-2023].

[13] T. Morgan, "Throwing Shade Ray Tracer," https://www.tylermw.com/throwing-shade/, 2008, [Archived at https://web.archive.org/web/20210514032050/https://www.tylermw.com/throwing-shade.

[14] M. Hahsler, "Recommenderlab: An R framework for developing and testing recommendation algorithms," May 2022.

[15] F. Křikava and J. Vitek, "Tests from traces: automated unit test extraction for R," in *27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2018)*, ser. Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis. Amsterdam, Netherlands: ACM Press, Jul. 2018, pp. 232–241. [Online]. Available: https://hal.archives-ouvertes.fr/hal-02131523

[16] M. Paleczny, C. Vick, and C. Click, "The Java Hotspot Server Compiler," in *Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium - Volume 1*, ser. JVM'01. USA: USENIX Association, 2001, p. 1.

[17] E. OpenJ9, "The Eclipse OpenJ9 Virtual Machine," https://www.eclipse.org/openj9/, 2020.

[18] M. Serrano, R. Bordawekar, S. Midkiff, and M. Gupta, "Quicksilver: A Quasi-Static Compiler for Java," in *Proceedings of the 15th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA

'00. New York, NY, USA: Association for Computing Machinery, 2000, p. 66–82. [Online]. Available: https://doi.org/10.1145/353171.353176

[19] P. G. Joisha, S. P. Midkiff, M. J. Serrano, and M. Gupta, "A Framework for Efficient Reuse of Binary Code in Java," in *Proceedings of the 15th International Conference on Supercomputing*, ser. ICS '01. New York, NY, USA: Association for Computing Machinery, 2001, p. 440–453. [Online]. Available: https://doi.org/10.1145/377792.377902

[20] X. Xu, K. Cooper, J. Brock, Y. Zhang, and H. Ye, "ShareJIT: JIT Code Cache Sharing across Processes and Its Practical Implementation," *Proc. ACM Program. Lang.*, vol. 2, no. OOPSLA, oct 2018. [Online]. Available: https://doi.org/10.1145/3276494

[21] M. Arnold, A. Welc, and V. T. Rajan, "Improving Virtual Machine Performance Using a Cross-Run Profile Repository," in *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA '05. New York, NY, USA: Association for Computing Machinery, 2005, p. 297–311. [Online]. Available: https://doi.org/10.1145/1094811.1094835

[22] G. Ottoni and B. Liu, "HHVM Jump-Start: Boosting Both Warmup and Steady-State Performance at Scale," in *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2021, pp. 340–350.

[23] R. Zhuykov, V. Vardanyan, D. Melnik, R. Buchatskiy, and E. Sharygin, "Augmenting JavaScript JIT with ahead-of-time compilation," in *2015 Computer Science and Information Technologies (CSIT)*, 2015, pp. 116–120.

[24] R. Zhuykov and E. Sharygin, "Ahead-of-time compilation of JavaScript programs," *Programming and Computer Software*, vol. 43, no. 1, pp. 51–59, jan 2017.

Publication Based on this Thesis

Refereed Publications:

1. M. K. Mehta, S. Krynski, H. Gualandi, M. Thakur, J. Vitek. 2023. “Reusing Just-in-Time Compiled Code”. Conditionally accepted in Proc. ACM Program. Lang., OOPSLA 2023, Cascais, Portugal, Oct 22-27, 2023.