COPY & PATCH JUST-IN-TIME COMPILATION FOR R

Bc. Matěj Kocourek

Master's thesis
Faculty of Information Technology
Czech Technical University in Prague
Department of Theoretical Computer Science
Study program: Informatics
Specialisation: System Programming
Supervisor: doc. Ing. Filip Křikava, Ph.D.
May 9, 2025

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of master's thesis

Title:	Copy & patch just-in-time compilation for R
Student:	Bc. Matěj Kocourek
Supervisor:	doc. Ing. Filip Křikava, Ph.D.
Study program:	Informatics
Branch / specialization:	System Programming
Department:	Department of Theoretical Computer Science
Validity:	until the end of summer semester 2025/2026

Instructions

R is a challenging target for native compilation due to its dynamic nature and interesting mix of language features, including first-class environments, lazy evaluation, and reflection without frontiers. The Rsh project took the endeavor to build an optimizing JIT for the R programming language. It succeeded in providing a compiler that compiles code whose peak performance beats GNU-R in several benchmarks. However, it is a slow compiler not suited for a baseline JIT. The "copy&patch" JIT, as proposed by Haoran Xu and Fredrik Kjolstad in their OOPSLA'21 paper, provides an interesting approach for building fast baseline JITs. The thesis aims to explore this idea and implement a prototype JIT for the R programming language.

Tasks:
- Get familiar with the R programming language and its byte code.
- Get familiar with the copy&patch approach.
- Implement a prototype of copy&patch on a subset of the R programming language.
- Evaluate the prototype.

Electronically approved by doc. Ing. Jan Janoušek, Ph.D. on 15 January 2025 in Prague.

Citation of this thesis: Kocourek Matěj. *Copy & patch just-in-time compilation for R*. Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2025.

First of all, I would like to thank my supervisor, doc. Ing. Filip Křikava, Ph.D. His fast responses, enthusiastic expertise and personal communication style were all very valuable to me, and were in large part responsible for the resulting work.
I would also like to thank my family for supporting me, be it in writing this thesis, my studies on CTU as a whole, or, in general and most importantly, in everything I do.
Similarly, I am thankful for my loving girlfriend, whose continuous support and sweet positivity never fails to give me energy and motivation, for finishing this thesis or otherwise.
Additionally, I am grateful for my friends, with some of whom we helped each other in our studies from the start of high school all the way to the end of master's course at FIT CTU. I would like to thank my best friend especially, who largely influenced my enthusiasm for my studies, not to mention his support and genuine friendship outside of school.
I am also very grateful to people who helped me with ideas or other suggestions for the implementation part of this thesis, mostly Matouš Kulhan and Michal Dufek.
Lastly, special thanks to all the people who helped me proofread this thesis, namely: Matouš Kulhan, Matěj Frnka, Leandra Sequeira and Jiří Kocourek.

Declaration

In Prague on May 9, 2025

Abstract

This thesis presents a prototype Just-In-Time (JIT) compiler for the R programming language based on the Copy-and-Patch approach. Due to R's highly dynamic nature, native compilation is a significant challenge. Building on the Ř project's existing bytecode instruction infrastructure, this work adapts all 104 of its instruction implementations for use in the Copy-and-Patch JIT. The resulting prototype compiles R bytecode to native code by copying and patching precompiled chunks of instructions at runtime, enabling extremely fast compilation with minimal overhead. Despite being conceived as a minimal proof-of-concept, the implementation supports full execution of complex R programs, as tested in all benchmark suites from the Ř project, including the widely used Shootout suite. Benchmarks show compilation times with the median value of 0.7 ms (maximum under 6 ms), which is several orders of magnitude faster than what Ř currently offers. Execution performance, although slower, follows the characteristics of Ř implementation, despite the baseline nature of the JIT. Several additional optimizations were implemented, one yielding up to 21% performance improvement in specific benchmarks. The project demonstrates that the Copy-and-Patch approach is a viable foundation for building fast, low-overhead JIT compilers for dynamic languages like R.

Keywords R language, Ř project, Rsh project, Just-In-Time compilation, JIT, Copy-and-Patch, baseline JIT, dynamic languages, bytecode interpretation, compiler optimizations, native code generation

Abstrakt

Tato práce představuje prototyp Just-In-Time (JIT) kompilátoru pro programovací jazyk R, založeného na principu Copy-and-Patch. Kvůli dynamické podstatě jazyka R představuje nativní kompilace značný problém. Tato práce staví na existující implementaci instrukcí z projektu Ř a adaptuje všech 104 podporovaných instrukcí pro použití s Copy-and-Patch JIT kompilátorem. Výsledný prototyp kompiluje R bytecode do nativního kódu kopírováním a záplatováním předkompilovaných kusů strojového kódu, což umožňuje velmi rychlou kompilaci s minimální režií. Přestože původním cílem byl pouze minimální prototyp, výsledná implementace plně podporuje běh komplexních R programů, jak bylo ověřeno na benchmarcích používaných v projektu Ř, včetně široce používané sady Shootout. Výsledky ukazují dobu kompilace s mediánem 0,7 ms (maximum 6 ms), což je o několik řádů rychlejší než stávající řešení projektu Ř. Výkon při běhu programu, ačkoliv nižší, odpovídá charakteristikám implementace Ř, a to i přes fakt, že jde o naprosto jiný způsob kompilace. Několik dalších optimalizací bylo rovněž integrováno, přičemž jedna konkrétní dosahuje až 21% zvýšení výkonu v některých sadách testů. Projekt demonstruje, že přístup Copy-and-Patch je plně použitelným řešením pro tvorbu rychlých, odlehčených JIT kompilátorů i pro dynamické jazyky, jako je R.

Klíčová slova jazyk R, projekt Ř, projekt Rsh, Just-In-Time kompilace, JIT, Copy-and-Patch, baseline JIT, dynamické jazyky, interpretace bytekódu, optimalizace kompilátoru, generování strojového kódu

Contents

Introduction **1**

1 Background **3**

1.1 The R Programming Language 3
1.2 Just-In-Time Compilation 5
1.2.1 The Ř Project 6
1.3 Copy-and-Patch Approach 7
1.4 R Bytecode 9
1.5 Memory Representation: SEXPs 10
1.5.1 Type CLOSXP 11
1.5.2 Type BCODESXP 12
1.5.3 Type EXTPTRSXP 15

2 Implementation **16**

2.1 Stencil Implementation 16
2.1.1 Stencil Design 17
2.1.2 Instruction Arguments 19
2.1.3 Usage of Ř 21
2.2 Additional Optimizations in Stencils 23
2.2.1 Eliminating Redundant Register Saving 24
2.2.2 Patching Values Instead of Addresses 26
2.2.3 Omitting Stack Overflow Check 28
2.2.4 Optimizing Pushing to Stack 28
2.2.5 Efficient Memory Access to Internal Relocations 31
2.2.6 Stencil Specialization 31
2.3 Stencil Compilation 33
2.4 Stencil Extraction 34
2.4.1 Parsing the Object File 36
2.4.2 Extra Optimizations 38
2.4.3 Exporting 40
2.5 Execution Engine 42
2.5.1 Bytecode Compilation 43
2.5.2 Preparation Run 44
2.5.2.1 Instruction Validation 44
2.5.2.2 Sizes and Lookup Tables 44
2.5.3 Memory Allocation 45

2.5.3.1 Private Memory 45
2.5.3.2 Shared Memory 46
2.5.4 Stencil Selection 47
2.5.5 Copying and Patching 48
2.5.6 Finalization 50
2.6 Runtime 50
2.6.1 Execution 52
2.6.2 Memory Management 53

3 Evaluation 55
3.1 Setup 55
3.2 Benchmark Datasets 56
3.3 Compilation Speed 56
3.4 Execution Performance 58
3.5 Optimization Evaluation 62
3.5.1 Omitting Stack Overflow Check 62
3.5.2 Memory Models 63

Conclusion 65

Attachment Contents 68

List of Figures

1.1 Example of a typical use of the R language [4]. 4
1.2 Copy-and-Patch framework overview [2]. 7
1.3 Copy-and-Patch process detailed overview [2]. 8
1.4 CLOSXP representation [16]. 11
1.5 Internal representation of a `BCODESXP` object [16]. 12
1.6 Bytecode representation in `BCODE_CODE` [16]. 13
1.7 Constant pool in `BCODE_CONSTS` [16]. 14
1.8 EXTPTRSXP representation [16]. 15

2.1 Memory layout. 45

3.1 Compilation speed for all benchmarks. 56
3.2 Execution performance speedup. 58
3.3 Performance comparison of benchmarks of the simple dataset. . 60
3.4 Performance comparison of benchmarks of the shootout dataset. 61
3.5 Performance comparison for turning off stack overflow checks. . 62
3.6 Performance comparison for different mcmodel options. 63
3.7 Execution performance speedup using PC-relative model. . . . 64

List of Tables

2.1 Compiler options used and their justifications in the project. . 35
2.2 Supported relocation types. 37
2.3 Relocation Types for Stencil Holes. 39

3.1 Stencil size in bytes for each R instruction. 57

List of Code listings

1.1 Example R bytecode. 10
1.2 Stack-based approach demonstration. 10
2.1 Stencil definition macro. 17
2.2 Stencil placeholder return function. 17
2.3 Ř stack macros. 18
2.4 Stencil for the POP instruction. 18
2.5 Stencil for the POP instruction compiled. 18
2.6 Constant argument extern symbols. 19
2.7 Instruction immediate access example. 19
2.8 Constant argument constpool symbol. 20
2.9 Constant argument control flow symbol. 20
2.10 Advanced constant access example. 20
2.11 Advanced constant access example compiled. 21
2.12 Ř function example. 21
2.13 ENDASSIGN2 stencil. 22
2.14 LDCONST stencil. 22
2.15 Constant argument constcell. 22
2.16 GETVAR stencil example. 23
2.17 Constant argument constcell. 23
2.18 STEPFOR stencil. 23
2.19 Comparison for optimized version of the ADD stencil. 25
2.20 Function attribute optimization and preprocessor check. 25
2.21 Init function. 26
2.22 Init function compiled. 26
2.23 Direct value patching of constants. 27
2.24 Comparison of relocating address vs value. 28
2.25 Optimized PUSHVAL. 29
2.26 Comparison of STACK_PUSH safety optimization. 29
2.27 Optimized PUSHVAL. 30
2.28 Comparison of STACK_PUSH optimization. 31
2.29 Ř extern attributes. 32
2.30 Optimized LDCONST stencils. 33
2.31 Example of specialized MATH1 stencils compiled. 33
2.32 Removing JMP at the end of stencil 40
2.33 Comparison of optimized version for the POP stencil. 40
2.34 Function for exporting stencils. 41
2.35 Example of exported stencil (compacted). 42
2.36 Header file including all stencils. 42
2.37 Package example use. 43
2.38 Decoding bytecode. 44
2.39 Allocating near memory. 46

2.40 Allocating shared memory. 47
2.41 LDCONST specilization. 48
2.42 Recursive compilation. 49
2.43 Patching of .rodata. 49
2.44 Patching function. 50
2.45 Creation of custom EXTPTRSXP. 51
2.46 Structure for the result of copy-patch. 51
2.47 Macros to check for RCP pointer. 51
2.48 Support for copy-patch in R evalution. 52
2.49 Calling of copy-patched code during runtime. 53
2.50 Freeing of internal structure. 54

List of abbreviations

JIT	Just-In-Time (compiler)
AST	Abstract Syntax Tree
SEXP	S Expression
Opcode	Operation code
IR	Intermediate Representation
ELF	Executable and Linkable Format
I/O	Input/Output
PC	Program Counter
BC	Byte Code
OS	Operating System

Introduction

The R programming language is widely used in data science, statistics, and research for its expressive syntax and rich ecosystem of libraries. Originally designed as an interpreted language for statistical computing, R has evolved into a powerful, dynamic language that supports high-level abstractions, including first-class environments, lazy evaluation, non-standard evaluation, and runtime allowing for unlimited reflection without frontiers. These features, while central to R's expressive power, also make it an unusually difficult target for both Ahead-of-Time and Just-In-Time (JIT) native compilation.

Native compilation of R code presents several core challenges. Its execution model includes ubiquitous use of environments (essentially first-class mutable scopes), which are often used for metaprogramming and state encapsulation. Lazy evaluation allows function arguments to be passed as unevaluated expressions and evaluated only when needed, possibly multiple times or not at all. Combined with R's extensive use of reflection and runtime introspection, such as modifying functions during execution or dynamically constructing and evaluating code, these language characteristics make it difficult to apply traditional compiler optimizations, particularly those relying on static analysis and predictability.

The Ř project [1] (pronounced Rsh) is an initiative that aims to bring high-performance native compilation to R. It introduces several optimizations into its alternative version of the GNU R runtime, specifically designed to address R's idiosyncrasies, and designs a new system for compiling R bytecode using LLVM. Ř has shown that it is possible to compile R code into optimized native code, achieving significant performance gains over the standard GNU R interpreter in several real-world benchmarks. However, the Ř compiler is slow, making it ill-suited for use as a *baseline* JIT – one that prioritizes low compilation latency and is suitable for fast, repeated invocations on small or short-lived programs.

This thesis explores an alternative approach to implementing a baseline JIT compiler for the R language by applying *Copy-and-Patch* technique, originally proposed by Haoran Xu and Fredrik Kjolstad in their OOPSLA 2021 paper [2].

Copy-and-Patch is a minimalistic and efficient method for JIT compilation that trades off some optimization potential for near-instantaneous compilation.

The goal of this work is to implement a working prototype of a Copy-and-Patch compiler that integrates with the R runtime, making use of the optimizations introduced in Ř, while significantly improving on the project's main disadvantage – compilation speed. If successful, it could be used as a baseline JIT for the Ř project.

Chapter 1

Background

This chapter introduces the contents required for the understanding of the followup work. It introduces the R language, its memory representation and its execution model. Special focus is on the R bytecode representation, as it is crucial for this project. It presents the idea of JIT compilation, specifically the Copy-and-Patch approach. Another section is dedicated to the Ř project, which serves as the implementation platform for this work.

1.1 The R Programming Language

R is a programming language released in 1993, designed primarily for statistical data analysis and visualization. As a high-level language, R allows users to perform complex statistical operations without requiring deep knowledge of computer science or low-level programming (an example of a typical use can be seen in Figure 1.1)

Although R is an open-source project and not limited to a single implementation, the GNU R [3] implementation is widely regarded as the de facto standard, therefore it will also be considered as such throughout this work.

R is dynamically typed, uses garbage collection, and is interpreted. While this high-level design makes R more user-friendly, it also results in reduced efficiency, often making R significantly slower than compiled languages.

R relies heavily on function calls for many constructs that are typically implemented as language primitives in other programming languages. Furthermore, functions in R are first-class objects. As a result, even core control structures, such as loops and conditionals, are implemented as ordinary functions and can be reassigned at runtime.

For example, the following is valid R code that overrides the `while` loop keyword with a function that exits the R environment, whenever used:

```
while <- function(...) q()
```

```
# Library
library(ggplot2)

# create a dataset
data <- data.frame(
name=c( rep("A",500), rep("B"
    ,500), rep("B",500), rep("C"
    ,20), rep('D', 100) ),
value=c( rnorm(500, 10, 5), rnorm
    (500, 13, 1), rnorm(500, 18,
    1), rnorm(20, 25, 4), rnorm
    (100, 12, 1) )
)

# Most basic violin chart
ggplot(data, aes(x=name, y=value,
     fill=name)) + geom_violin()
```

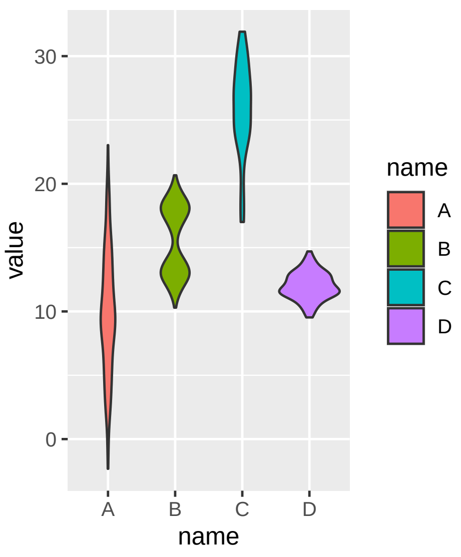


Figure 1.1 Example of a typical use of the R language [4].

This level of flexibility, while powerful, introduces significant challenges for optimization. Any compiler or runtime must conservatively assume that even fundamental operations may have been redefined, thereby restricting opportunities for specialization or inlining.

The primary limitation of the interpreter is instruction dispatch. Each instruction requires decoding the opcode, branching to the handler, and executing a fixed sequence of C operations, often including type checks, coercions, and memory allocations. This overhead accumulates rapidly in tight loops or recursive calls.

Optimizations like inlining primitives or unboxing common values have been explored and implemented in other interpreters [5, 6], but they are limited in R by the need to preserve reference semantics, handle non-standard evaluation, and manage environments as first-class values.

Optimizing R is notoriously difficult due to a variety of semantic and dynamic features. According to Flückiger et al. in their work on Ř [1], the list of the biggest challenges includes:

- **Out-of-order arguments:** R functions can be called with named arguments in any order (e.g., `add(y = 1, x = 2)`). The interpreter must reorder arguments at runtime to match formal parameters.
- **Missing arguments:** Functions can be called with missing parameters (e.g., `add(, 2)` or `add(y = 2)`). Defaults must be inserted if available; otherwise, errors are deferred until access.
- **Argument overflow:** Calls can supply more arguments than a function expects. The interpreter must detect and reject such calls at runtime.

- **Promises:** Function arguments are passed as promises (lazy thunks) that may contain side effects and are only evaluated on first access. Optimizations must preserve their deferred and state-dependent behavior.
- **Reflection:** R supports introspection, allowing access to local variables and environments from higher up the call stack. This necessitates maintaining first-class runtime environments.
- **Vector semantics:** R treats all values as vectors, including scalars (as length-one vectors). Implementations must assume boxing and vectorized behavior unless it can be proven otherwise.
- **Object attributes:** Any value can carry attributes, including class tags used for method dispatch. The runtime must inspect and respect these attributes when executing operations.

1.2 Just-In-Time Compilation

To address the limitations of interpreted languages, many language runtimes have adopted JIT compilation. A JIT compiler translates intermediate representations (IR) into machine code at runtime, often specializing based on runtime types or profiling information. This allows the system to eliminate type checks, fuse instructions, and reuse native registers.

Prominent examples include V8 for JavaScript [7], PyPy for Python [8], and GraalVM for polyglot languages [9]. These systems are complex, involving IR construction, speculative optimization, deoptimization support, and tiered compilation. While effective, they require deep integration with the runtime and a large engineering effort to maintain correctness.

Tiered compilation is an important feature of modern JIT compilers. This design involves multiple compilation strategies with increasing level of optimizations, but slower compilation speed. Code is first compiled quickly using a low-tier strategy to allow immediate execution, and then, if necessary, recompiled at runtime using more aggressive optimizations. This allows for fast startup times while still achieving high performance for frequently executed code. The low-tier strategy is called a *baseline JIT*, and it is the focus of this work.

One of the challenges that JIT compilers frequently encounter is inefficiency caused by memory addressing that does not fit within the constraints of relative addressing. In such cases, compilers must resort to absolute addressing. While relative addressing is generally faster and produces more compact code (since the target address can be embedded directly within a single instruction) absolute addressing often results in larger and slower code. It may require multiple instructions: one to load the absolute address into a register, and another to access the memory at that address. This contrasts with relative addressing, where the same operation can typically be performed in a single instruction.

For R, the constraints of the language and the interpreter make JIT compilation challenging to implement. The language's reliance on lazy evaluation, first-class environments, and copy-on-write semantics complicates optimization. Several JIT projects have attempted to work around this in the past (most notably FastR [10], pqR [11], and RLLVMCompile [12]) but they are no longer actively developed, never reaching mainstream use. Another similar project was CXXR [13], which made an effort to re-implement R in C++, but is now also abandoned.

1.2.1 The Ř Project

Ř is an optimizing just-in-time (JIT) compiler for the R language, developed by the PRL-PRG group over the past few years. It utilizes the LLVM compiler infrastructure as a backend and introduces several key enhancements over the standard GNU R interpreter. One of its central features is *contextual dispatch*, a mechanism in which functions are versioned and compiled under distinct assumptions. At runtime, the system selects the most appropriate version based on the dynamic context of the call site. This approach yields significant performance improvements, with benchmarks showing an average speedup of approximately 1.7× compared to GNU R [1].

More features of Ř include modified stack representation, allowing certain values to be kept in unboxed formats during execution, and a caching technique used for faster access to variables, called `BCells`. However, the compilation latency of Ř is not low enough to warrant its use as a baseline JIT. Therefore, it would benefit from integration into a tiered compilation system, where it serves as a higher-tier optimizer.

The Ř project is architecturally divided into two components: the server and the client. The server is implemented in Java and leverages LLVM to compile R code. In simplified terms, it analyzes R bytecode and emits C code consisting of function calls corresponding to each R instruction. The generated C code is subsequently compiled into a native object, which can then be executed within the R environment. Notably, the server component is not utilized in this project.

The functions invoked in this manner are part of the client component. These instruction functions are implemented using Ř's optimization techniques and consolidated within a C header file. This header is included from the source code generated by the server component, allowing this source code to inline it, and therefore use the implementations directly. In the context of this work, the client header file is repurposed to support the Copy-and-Patch approach, aiming to leverage the runtime speed advantages of Ř while combining it with a lightweight, fast-compiling JIT for the first tier of execution.

1.3 Copy-and-Patch Approach

As described earlier, one of the most important features of a baseline JIT is the speed of its execution. In this work, we will be using a novel approach called Copy-and-Patch, introduced by Haoran Xu and Fredrik Kjolstad in their article from 2021 [2]. This approach aims to provide the lowest compilation speed possible, while still allowing for the advantages of efficient code.

It is capable of translating both high-level languages and low-level bytecode programs into binary code by stitching together code from a library of precompiled binary implementation variants. These binary implementations are referred to as *stencils*, as they contain placeholders (called holes) that are later filled during the *patch* phase. The authors present a method for constructing a stencil library and describe the *Copy-and-Patch* algorithm itself used to generate optimized binary code from these templates.

This approach provides exceptionally fast compilation times, significantly outperforming traditional compilers, as the execution engine performs minimal work: parsing bytecode, copying the corresponding instructions into memory, and stitching them together. It also offers ease of maintenance: adopting Copy-and-Patch is primarily a one-time effort. Subsequent updates to the implementation require little to no additional work, since new features are automatically incorporated through the underlying stencils.

This approach has inspired several real-world implementations, most notably for Lua [14] and Python [15]. The Python implementation, now merged into the CPython codebase, is what inspired this work. Although R and Python differ significantly in many aspects, they share similar execution models, both of which could benefit from fast and easily maintainable JIT compiler.

Stencils are at the core of the Copy-and-Patch approach. Each stencil is created from a function compiled ahead-of-time, corresponding to a single bytecode instruction. These stencils are extracted from compiled ELF object files, along with relocation metadata (holes), using a custom linker-like wrapper. The resulting templates are stored as binary blobs in the runtime.

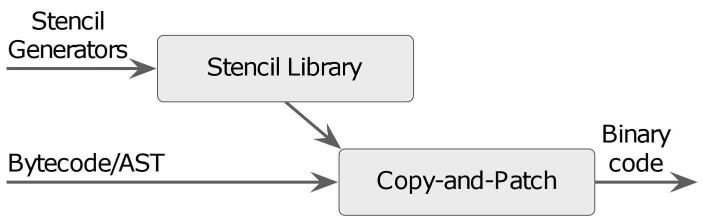


Figure 1.2 Copy-and-Patch framework overview [2].

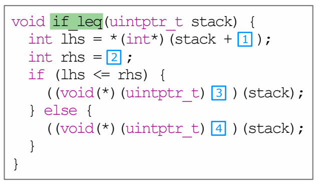
When the JIT compiler is invoked, a Copy-and-Patch engine instantiates these stencils into a contiguous code buffer. It applies relocations to inject immediate values (e.g. constants, offsets, jump targets) directly into the machine code at positions specified in the hole list. It then links the stencils together to form a complete function body, preserving control flow. The general overview of the Copy-and-Patch framework can be seen in Figure 1.2.

Stitching of code together allows elimination of *calls* that are usually present in the bytecode interpretation. The *calls* are optimized into *jumps* due to the tail-call optimization. These *jumps* right to the next instruction can then be omitted altogether, as this is effectively a NOP[1]. This is the main source of execution performance gain that the Copy-and-Patch approach can offer.

The phases of this approach could be split into three logical steps:

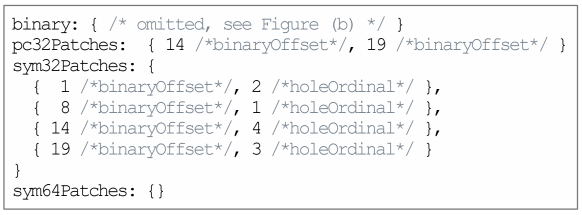
1. Compilation of stencils
2. Extraction of stencils
3. Copy-and-Patch compilation

The first two steps are run only once on the target machine (during installation). Only the last step is performed each time the JIT is invoked. Splitting this in such a way is what allows this approach to be so fast in compilation times, compared to regular compilers that effectively do all three steps each time they are invoked (compilation and linking of the executable).



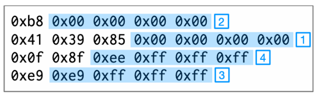
```
void if_leq(uintptr_t stack) {
  int lhs = *(int*)(stack + [1]);
  int rhs = [2];
  if (lhs <= rhs) {
    ((void(*)(uintptr_t) [3] )(stack);
  } else {
    ((void(*)(uintptr_t) [4] )(stack);
  }
}
```

(a) Stencil source in C



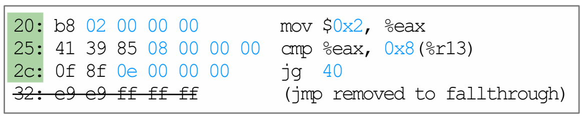
```
binary: { /* omitted, see Figure (b) */ }
pc32Patches:  { 14 /*binaryOffset*/, 19 /*binaryOffset*/ }
sym32Patches: {
  {  1 /*binaryOffset*/, 2 /*holeOrdinal*/ },
  {  8 /*binaryOffset*/, 1 /*holeOrdinal*/ },
  { 14 /*binaryOffset*/, 4 /*holeOrdinal*/ },
  { 19 /*binaryOffset*/, 3 /*holeOrdinal*/ }
}
sym64Patches: {}
```

(b) Generated stencil header



```
0xb8 0x00 0x00 0x00 0x00 [2]
0x41 0x39 0x85 0x00 0x00 0x00 0x00 [1]
0x0f 0x8f 0xee 0xff 0xff 0xff [4]
0xe9 0xe9 0xff 0xff 0xff [3]
```

(c) Compiled executable code



```
20: b8 02 00 00 00          mov $0x2, %eax
25: 41 39 85 08 00 00 00    cmp %eax, 0x8(%r13)
2c: 0f 8f 0e 00 00 00       jg  40
~~32: e9 e9 ff ff ff~~          (jmp removed to fallthrough)
```

(d) Result after Copy-and-Patch

Figure 1.3 Copy-and-Patch process detailed overview [2].

The steps of the Copy-and-Patch approach, as described in the article [2], are illustrated in Figure 1.3. In Figure 1.3a, a simple example of a stencil implemented in C is shown. The stencil contains four placeholders, referred to as "holes", marked in blue. The first hole represents an offset used to access a value, the second holds an immediate constant, and the third and fourth are function call sites that will later be patched with addresses pointing to appropriate locations in executable memory.

[1]No operation instruction - does no change to the execution environment

Figures 1.3b and 1.3c depict the stencil extraction process. Figure 1.3c shows the compiled machine code of the stencil, highlighting the positions of the holes. The extraction tool then processes this binary into a C header file, shown in Figure 1.3b, which is included in the Copy-and-Patch engine.

Finally, Figure 1.3d illustrates the completed result after executing the Copy-and-Patch process. The engine places the stencil into a concrete memory location and patches the holes with the appropriate values. The final `JMP` instruction is removed, as discussed earlier.

This project adopts a similar system, adapted to accommodate the R runtime and its unique characteristics.

1.4 R Bytecode

R is an interpreted language, but the interpreter comes in two forms. There's the AST interpreter which runs SEXPs as they are, and there's the byte code interpreter, which runs bytecode. Bytecode is generated from ASTs using the byte code compiler. The compiler is a Just-In-Time compiler that is turned on by default in the newest versions of R. If the JIT is running whenever function is executed twice, it gets byte code compiled before the second execution. [16]

R bytecode is an intermediate representation of R code designed to optimize execution by transforming high-level expressions into a compact, lower-level format that can be processed by the R virtual machine. This bytecode representation is generated by the compiler package and consists of a sequence of instructions that operate on R's internal data structures. Unlike machine code, which is executed directly by hardware, R bytecode is interpreted by the R bytecode engine, which is a part of the R runtime environment.

Each bytecode instruction corresponds to a fundamental operation in R, such as arithmetic calculations, function calls, or control flow management. Instructions are typically composed of an opcode and, in many cases, one or more arguments specifying constants or labels. Example of a real R bytecode can be seen in Code listing 1.1.

R's bytecode follows a stack-based execution model, where each instruction operates by manipulating an operand stack rather than using registers or direct memory access. Instructions push and pop values from this stack as required, performing computations before storing or returning results.

A simple example demonstrating this execution model is shown in Code listing 1.2. The instruction `LDCONST` loads a compile-time constant and pushes it onto the stack. As such, it has a stack effect of `0 POP` and `1 PUSH`, resulting in a net stack size change of `+1`. The instruction `ADD` pops two values from the stack, adds them together, and pushes the result back onto the stack. Thus, it has a stack effect of `2 POP` and `1 PUSH`, yielding a net stack change of `-1`. Finally, the `RETURN` instruction pops the top value from the stack and returns it from the R function. Its stack effect is `1 POP` and `0 PUSH`, again resulting in a net change of `-1`.

```
# if (x > 5) print('hello')
GETVAR x
LDCONST 5
GT
BRIFNOT @label1
GETFUN print
PUSHCONSTARG "hello"
CALL
RETURN
@label1
LDNULL
INVISIBLE
RETURN
```

Code listing 1.1 Example R bytecode.

```
# return (1 + 2)
LDCONST 1
LDCONST 2
ADD
RETURN
```

Code listing 1.2 Stack-based approach demonstration.

This execution model enables compact expression evaluation. The use of a stack simplifies instruction encoding, as operations do not need to reference specific memory locations or registers. Instead, operands are implicitly accessed based on their position in the stack.

While the bytecode interpretation reduces some of the parsing and evaluation costs of raw AST traversal, it does not fundamentally change the execution model. Each bytecode instruction still corresponds to a small, boxed operation, executed in isolation and mediated through the full interpreter stack.

1.5 Memory Representation: SEXPs

All values in R are heap-allocated and represented as `SEXP` (S-expression) objects. These are tagged unions with a fixed header and type-specific data. For example, integers are represented as `INTSXP` objects, which contain a header and a pointer to an array of 32-bit values. Symbols, closures, and environments have their own corresponding `SEXP` types, all sharing a uniform interface via macros like `TYPEOF(x)` and `INTEGER(x)`.

This model ensures uniformity and runtime type safety but imposes several costs. First, all values are boxed – primitive scalars cannot live on the stack or in registers for long. Second, every access involves an indirection and a tag check. Finally, the garbage collector must track all objects and roots, further increasing runtime overhead.

In numerical workloads, the cost of heap allocation and boxing is particularly visible. Even simple operations like `a + b` require heap-allocating the result as a new vector object, rather than reusing stack-allocated temporaries. The interpreter must also preserve R's copy-on-modify semantics, complicating in-place updates.

There are many SEXP types, but three are particularly important for this project. The explanations for the following subsections are taken from the book Everything You Always Wanted to Know About SEXPs But Were Afraid to Ask, by Konrad Siek [16].

1.5.1 Type CLOSXP

A closure is represented by a `CLOSXP`. The payload of a closure's SEXP is specified by a `closxp_struct`, which contains pointers to three other SEXPs: the closure's formals (argument definitions), its body, and its enclosing environment. This is visualized in Figure 1.4.

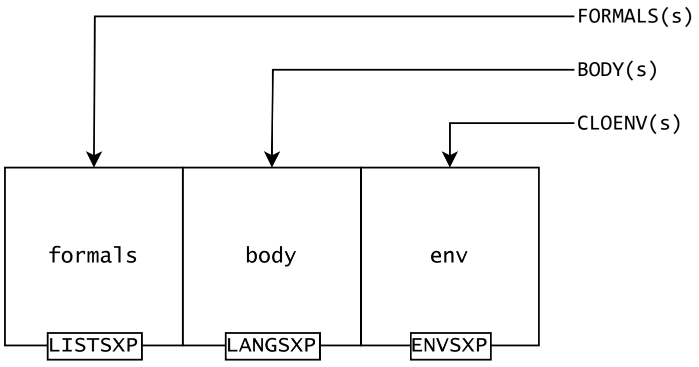


Figure 1.4 CLOSXP representation [16].

The formals are a list of formal arguments that the function accepts expressed as a pairlist (`LISTSXP`). If the function has no formal arguments, formals points to `R_NilValue`. If there are formal arguments, there is a list where the `tagval` point to symbols representing the names of arguments, and `carval` point to their values, if the arguments have default values. In arguments that do not have values, `carval` is set to `R_UnboundValue`.

The body is an AST representing the body of the function. The enclosing environment is the environment in which the function operates. These follow the structure we have laid out earlier.

1.5.2 Type BCODESXP

Internally, byte code compiled expressions in R are represented using a SEXP of type `BCODESXP`. There is no specialized payload structure for this type; instead, the general `listsxp_struct` is used, accessible through the `listsxp` field of the `u` union. Figure 1.5 illustrates the internal layout of a `BCODESXP` object.

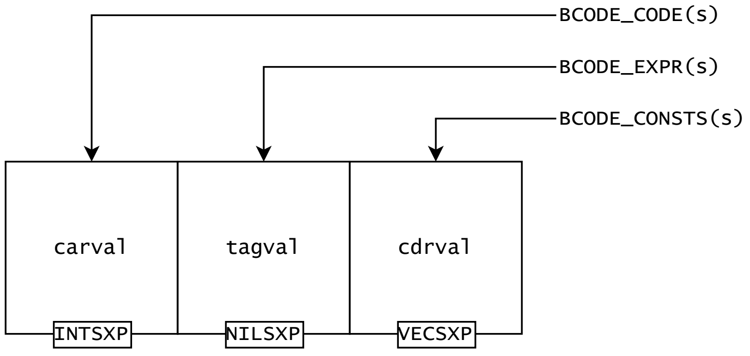


Figure 1.5 Internal representation of a `BCODESXP` object [16].

The `BCODE_CODE` field holds the actual byte code as a vector of encoded operations and arguments. The `BCODE_CONSTS` field refers to a vector of constants and associated metadata. The `BCODE_EXPR` field was likely intended to store the original abstract syntax tree, but is currently unused and omitted by the compiler.

As an example, consider compiling a simple literal expression:

```
library(compiler)
expr <- compile(42)
expr
<bytecode: 0x5555573ff110>
```

Disassembling this bytecode produces the instructions:

```
LDCONST 42
RETURN
```

The `BCODE_CODE` field contains these encoded instructions stored in an `INTSXP` vector. However, these are not plain integers. Each element represents a `BCODE` union, defined as:

```
typedef union { void *v; int i; } BCODE;
```

This allows an entry to store either a pointer to an operation or an integer argument. Due to platform-dependent alignment, each `BCODE` may occupy multiple slots in the underlying `INTSXP` vector. The exact number is calculated as:

```
int m = (sizeof(BCODE) + sizeof(int) - 1) / sizeof(int);
```

Typically, `m = 2`, meaning each instruction or argument spans two vector elements. Therefore, a minimal bytecode sequence (one instruction and one argument) may already require four entries. Figure 1.6 visualizes this layout.

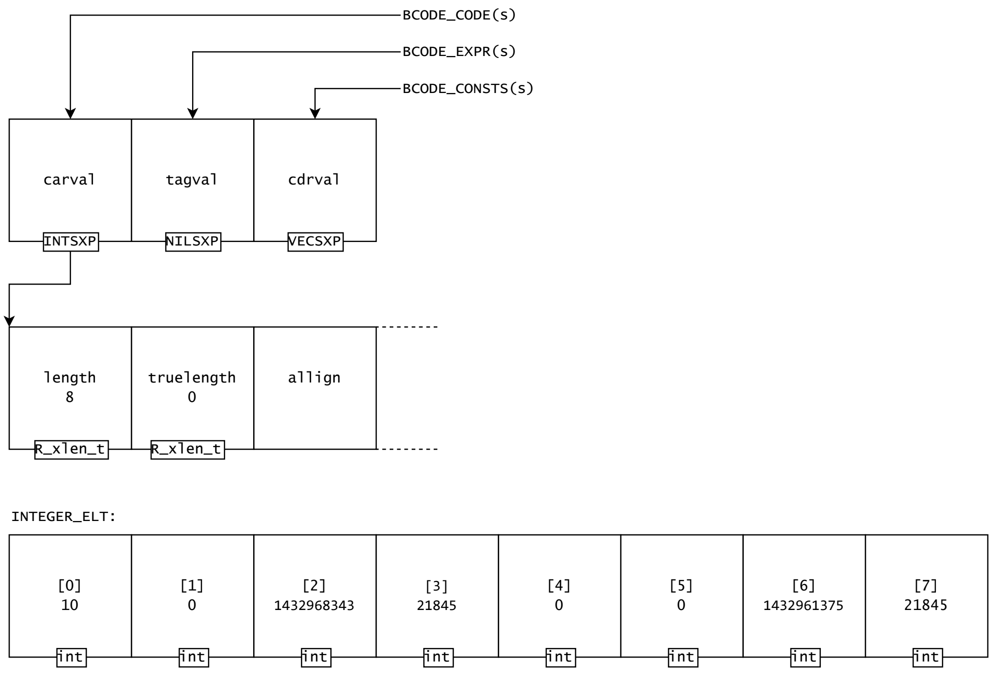


Figure 1.6 Bytecode representation in `BCODE_CODE` [16].

The first element of this vector always encodes the compiler version used during compilation. Subsequent entries store instructions and their arguments. To decode an instruction, the system uses the global `opinfo` array:

```
volatile static struct { void *addr; int argc; char *instname; }
    opinfo[OPCOUNT];
```

Each operation is matched by its address field (`addr`) to the corresponding `v` pointer from a `BCODE` entry. This matching is implemented in `findOp`:

```
static int findOp(void *addr) {
  int i;
  for (i = 0; i < OPCOUNT; i++)
    if (opinfo[i].addr == addr)
      return i;
  error(_("cannot find index for threaded code address"));
}
```

The argument count (`argc`) associated with each instruction is also read from this structure, allowing traversal of the bytecode vector.

Instruction arguments are not direct values, but are instead interpreted as indices into the `BCODE_CONSTS` vector. In the above example, the `LDCONST` instruction takes the argument `0`, referring to the first entry in `BCODE_CONSTS`, which stores the constant `42`. Figure 1.7 depicts this structure.

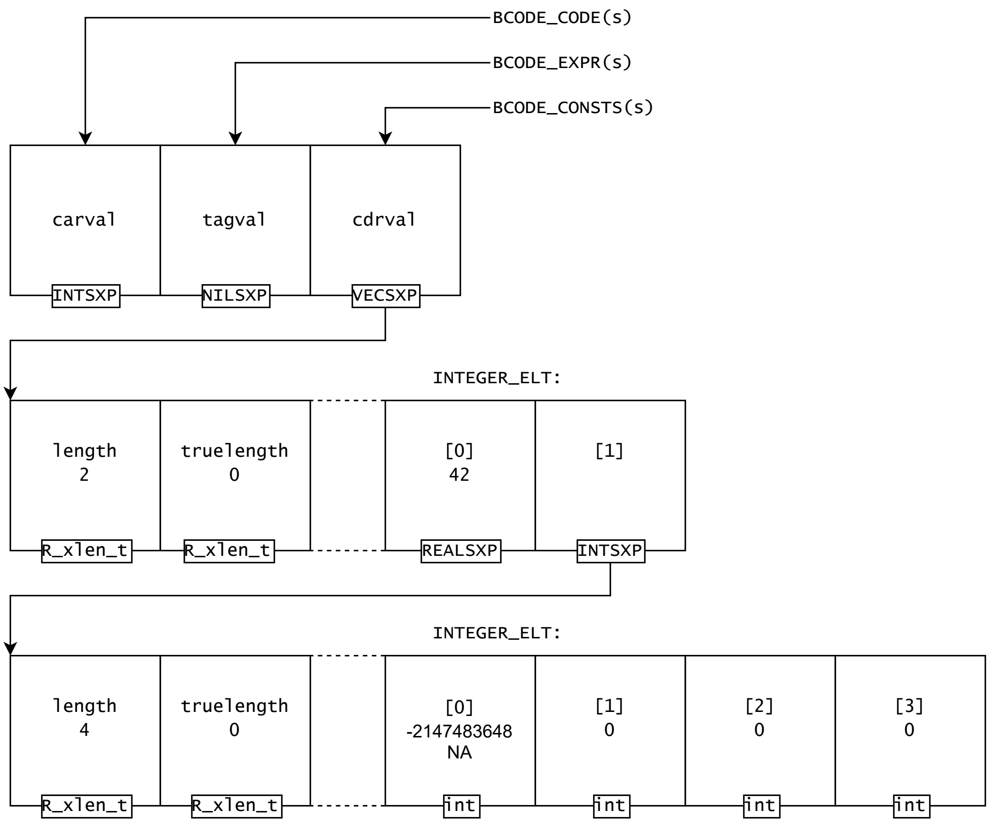


Figure 1.7 Constant pool in `BCODE_CONSTS` [16].

The first entry of `BCODE_CONSTS` is always the entire original expression. Thus, to reconstruct the uncompiled representation, one can retrieve element 0. Additionally, the vector contains an `expressionsIndex`, an integer vector that maps instructions back to their original source expressions. This index has the same length as the number of bytecode instructions (including the version) and aligns each to a corresponding constant expression. The first element is always `NA`, since the version is not derived from the source.

A more complex example, such as compiling a simple function, follows the same layout but with additional structure:

```
f <- cmpfun(function(x, y) x + y)
f
function(x, y) x + y
<bytecode: 0x55555b3b9130>
```

Here, the enclosing function is represented as a `CLOSXP`, but its body is no longer a `LANGSXP`; it becomes a `BCODESXP`. The `BCODE_CONSTS` vector now contains the entire uncompiled function body, variable references (`x` and `y`), and the + operation. Instructions like LDVAR and ADD reference these constants via their argument indices.

In summary, the BCODESXP representation enables R to encode executable expressions in a compact bytecode form, while preserving essential metadata for reconstruction, introspection, and debugging. It integrates instruction encoding, source mapping, and constant management within a unified structure that leverages existing SEXP mechanisms.

1.5.3 Type EXTPTRSXP

External pointers act as handles to C structures. They have a dedicated SEXP type: EXTPTRSXP. There is no dedicated payload type that would represent external pointers. Thus, listsxp_struct can be used to represent them, which is accessed via the listsxp field of the payload union. The internal layout can be seen in Figure 1.8.

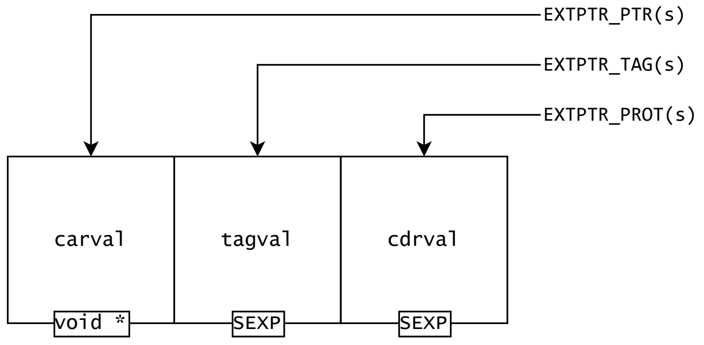


Figure 1.8 EXTPTRSXP representation [16].

The EXTPTR_PTR returns a pointer to a C structure, EXTPTR_TAG is used to provide identification to the pointer, and EXTPTR_PROT can provide a SEXP object that should be protected from GC, as long as the EXTPTR_PTR is alive.

Such pointer structures are not usually visible from standard R, but are crucial for some packages that require low-level memory addressing, such as this project.

Chapter 2

Implementation

This chapter describes both the static and dynamic components of the Copy-and-Patch system in detail. It begins with an overview of how stencils are compiled and extracted, followed by the structure of the runtime patching engine. It also discusses optimizations applied to reduce code size and improve performance, including strategies for register preservation, stack handling, relocation simplification, and instruction specialization. In each case, the discussion is accompanied by practical examples comparing optimized and unoptimized code paths.

Each stencil corresponds to a single bytecode instruction and is compiled as a standalone function. These functions are extracted from compiled object files using a custom toolchain and stored as header files, which are linked into the runtime engine. At runtime, when a function is JIT-compiled, a dedicated stencil Copy-and-Patch engine instantiates these templates into contiguous memory, patches immediate values and control flow targets, and links them together to form a complete function body.

The complete implementation of the extraction toolchain and the runtime engine consists of approximately 1,500 lines of hand-written C code. In addition, the stencil library comprises around 800 lines that adapt each instruction implementation from Ř into stencils. A minimal wrapper is also provided to expose the system as an R package, consisting of roughly 50 lines of combined R and C code. The full source code is included in the attachment.

2.1 Stencil Implementation

Stencils form the foundation of this project, making it essential to adopt a design that promotes both high performance and long-term maintainability. The central idea behind stencils is that they are compiled ahead of time using highly optimized code generation.

To support this objective, the C programming language was chosen as the implementation language. This decision not only enables low-level performance optimizations, but also aligns seamlessly with the existing Ř implementation of instruction handlers, which is likewise written in C.

2.1.1 Stencil Design

When designing the stencils, it is important to keep in mind that they serve as input to the subsequent extraction phase. This purpose influences the coding approach, making it distinct from conventional executable or library development.

The most straightforward method for implementing each R bytecode operation as a stencil is to define a C function named after the corresponding instruction. This naming convention simplifies both the implementation and the extraction process in the next phase.

Since R bytecode operates on a stack-based model, there is no need to pass temporary values between functions. This simplifies the stencil design and allows the use of a uniform function prototype across all stencils.

Based on this approach, a preprocessor macro can be introduced to standardize stencil definitions.

```c
#define RCP_OP(op) STENCIL_ATTRIBUTES SEXP _RCP_##op##_OP (void)
```

Code listing 2.1 Stencil definition macro.

This design also facilitates forwarding control to the next function call. All functions[1] conclude with a placeholder call that is later patched to point to the subsequent instruction to be executed. This placeholder may be omitted entirely if the current instruction is the last within the function (as further discussed in Section 2.4.2).

The placeholder call is implemented as an `extern` function and is invoked through an additional preprocessor macro created for this purpose.

```c
extern SEXP _RCP_EXEC_NEXT(void);
#define NEXT return _RCP_EXEC_NEXT()
```

Code listing 2.2 Stencil placeholder return function.

To manipulate the R stack, macros provided in Code listing 2.3, as defined in the Ř project, can be used.

[1]With the exception of the `RETURN` instruction, which returns the actual value and terminates the execution.

```
#define PUSH_VAL(n)                                         \
do {                                                        \
  int __n__ = (n);                                          \
  if (R_BCNodeStackTop + __n__ > R_BCNodeStackEnd) { \
    nodeStackOverflow();                                    \
  }                                                         \
  while (__n__-- > 0) {                                     \
    SET_SXP_VAL(R_BCNodeStackTop++, R_NilValue);            \
  }                                                         \
} while (0)

#define POP_VAL(n)                                          \
do {                                                        \
  R_BCNodeStackTop -= (n);                                  \
} while (0)

#define GET_VAL(i) (R_BCNodeStackTop - (i))
```

■ **Code listing 2.3** Ř stack macros.

A practical application of these features (shown in Code listings 2.1, 2.2, and 2.3) is demonstrated in Code listing 2.4, which contains a real implementation of a stencil for the `POP` instruction. Disassembly of this stencil can be seen in Code listing 2.5.

```
RCP_OP(POP) {
  POP_VAL(1);
  NEXT;
}
```

■ **Code listing 2.4** Stencil for the POP instruction.

```
subq  $0x10, R_BCNodeStackTop(%rip)
jmp   _RCP_EXEC_NEXT@PLT
```

■ **Code listing 2.5** Stencil for the POP instruction compiled.

It is worth noting that, as observed in the compiled code, control flow is transferred to the placeholder via a `JMP` rather than a `CALL`. This occurs due to the compiler applying tail call optimization – since there are no instructions following the call, the compiler replaces it with a more efficient `JMP`. As the Copy-and-Patch approach relies on this optimization for optimal performance, it is essential to verify that the optimization is indeed applied in the context of this project.

2.1.2 Instruction Arguments

Each R bytecode operation may include arguments that are known at compile time. These arguments must be accessible within the stencil code. While the intuitive approach might be to pass them as function parameters, doing so would violate the requirement for uniform function prototypes.

An alternative would be to pass a pointer to an array of arguments, allowing each function to retrieve only what it needs and increment the pointer accordingly. Although this would work, it introduces inefficiencies due to the additional indirection, pointer arithmetic, and consuming of a register solely for parameter passing.

A more efficient approach is to patch the compile-time arguments directly into the machine code during the patch phase. This method avoids the overhead and is also straightforward to use within the stencils. To support this, several read-only global variables are introduced, each corresponding to a possible argument position. These variables are marked with the `extern` keyword and use project reserved names.

When writing stencil code, these variables can be referenced to access the compile-time arguments. The compiler generates a relocation entry for each reference to these reserved names. During the extraction phase, these references are identified, and, rather than being relocated to internal symbols, are replaced with compile-time constants associated with the instruction (discussed more in Section 2.4.1).

To simplify the patching process, each immediate argument is represented by a separate `extern` variable with an ordinal suffix, rather than a single array. Since R instructions have at most four immediate arguments, this approach remains practical. This can be seen in Code listing 2.6.

```
extern const int _RCP_RAW_IMM0;
extern const int _RCP_RAW_IMM1;
extern const int _RCP_RAW_IMM2;
extern const int _RCP_RAW_IMM3;
#define GET_IMM(index) _RCP_RAW_IMM##index
```

Code listing 2.6 Constant argument extern symbols.

For instance, if a stencil prints its first immediate argument to the console, it could be implemented as shown in Code listing 2.7.

```
RCP_OP(EXAMPLE) {
  printf("%d\n", GET_IMM(0));
  NEXT;
}
```

Code listing 2.7 Instruction immediate access example.

Furthermore, additional runtime overhead can be avoided by resolving another layer of indirection during the patch phase. Although R bytecode arguments are integers, they are almost always used as either indices into a constant pool or labels for control flow.

In both scenarios, the indirection can be resolved ahead of time, and the evaluated result can be provided directly. This requires support from both the extraction tool and the patching process, but remains simple to use when developing stencils. To enable this, another set of reserved `extern` symbols, this time of the target type (e.g., `SEXP`), is used.

Likewise, for control flow instructions to efficiently perform jumps, additional `extern` placeholder functions are defined to represent jump targets. During extraction, the tool recognizes these reserved function names, enabling the patch algorithm to substitute them with the actual memory addresses corresponding to bytecode labels.

These implementations are shown in Code listings 2.8 and 2.9, respectively.

```
extern const SEXP _RCP_CONST_AT_IMM0;
extern const SEXP _RCP_CONST_AT_IMM1;
extern const SEXP _RCP_CONST_AT_IMM2;
extern const SEXP _RCP_CONST_AT_IMM3;
#define GETCONST_IMM(i) _RCP_CONST_AT_IMM##i
```

Code listing 2.8 Constant argument constpool symbol.

```
extern STENCIL_ATTRIBUTES SEXP _RCP_EXEC_IMM0(void);
extern STENCIL_ATTRIBUTES SEXP _RCP_EXEC_IMM1(void);
extern STENCIL_ATTRIBUTES SEXP _RCP_EXEC_IMM2(void);
extern STENCIL_ATTRIBUTES SEXP _RCP_EXEC_IMM3(void);
#define GOTO_IMM(i) return _RCP_EXEC_IMM##i()
```

Code listing 2.9 Constant argument control flow symbol.

The usage of both mechanisms is straightforward, as demonstrated in Code listing 2.10. In this example, the fictional instruction takes two immedi-

```
RCP_OP(EXAMPLE_ADVANCED) {
  Rf_PrintValue(GETCONST_IMM(0));
  GOTO_IMM(1);
}
```

Code listing 2.10 Advanced constant access example.

ate constant arguments: the first is an integer representing an index into the constant pool, identifying the constant to be retrieved; the second is an integer indicating the bytecode position or label to which control should jump after the instruction completes execution. Both arguments are resolved at compile time to eliminate the indirection that would otherwise be required within the stencil itself. The compiled result of this stencil is shown in Code listing 2.11.

```
lea   _RCP_CONST_AT_IMM0(%rip), %rdi
push  %rax
call  Rf_PrintValue@PLT
pop   %rdx
jmp   _RCP_EXEC_IMM1@PLT
```

Code listing 2.11 Advanced constant access example compiled.

As in earlier examples, the compiler applies tail call optimization. This confirms that even control flow–altering instructions conform to the design principles required for an efficient Copy-and-Patch implementation.

Several other symbols used to resolve indirections appear within the stencils, all adhering to the same principles illustrated in Code listing 2.8. Additionally, a relocation symbol for the R execution environment is introduced, referred to as `rho`.

2.1.3 Usage of Ř

The Ř implementation provides functions for each instruction, expecting all required values as arguments (e.g., input/output values, bytecode immediate constants, and others). An example function prototype is shown in Code listing 2.12.

```
void Rsh_EndAssign2(Value *rhs, Value lhs_cell, Value value,
                    SEXP symbol, SEXP rho);
```

Code listing 2.12 Ř function example.

This function operates on three stack values: the first used for both input and output, and the remaining two for input only. It also expects a `SEXP` symbol, which corresponds to a constant retrieved from the bytecode, and finally, the `rho` environment.

All of these arguments can be supplied from within the stencil environment. Thus, a stencil for this instruction can be easily implemented by including the Ř header and invoking the corresponding Ř function. Stack-based `Value` arguments are passed in order from bottom to top. If multiple constants are present, they are provided sequentially from the first to the last. An example of this can be seen in Code listing 2.13.

```
RCP_OP(ENDASSIGN2) {
  Rsh_EndAssign2(GET_VAL(3), *GET_VAL(2), *GET_VAL(1),
                 GETCONST_IMM(0), GET_RHO());
  POP_VAL(2);
  NEXT;
}
```

Code listing 2.13 ENDASSIGN2 stencil.

The compiler inlines the Ř function and generates the full stencil using the appropriate relocations. Once the Ř function finishes execution, the stencil removes the appropriate number of values from the stack, as specified in the instruction documentation.

Similarly, when a function needs to push new values onto the stack, the stencil performs the push before the Ř call and provides pointers to the newly added values. Example of this can be seen for in the LDCONST stencil seen in Code listing 2.14.

```
RCP_OP(LDCONST) {
  PUSH_VAL(1);
  Rsh_LdConst(GET_VAL(1), GETCONST_IMM(0));
  NEXT;
}
```

Code listing 2.14 LDCONST stencil.

As described earlier, Ř also incorporates a caching mechanism for variable access, using a technique called `BCells`. To support Ř functions that rely on this optimization, the stencil system must be capable of relocating BCell accesses for later patching. To see how this can be done in our stencil design, refer to Code listing 2.15. `BCells` are configured to "mirror" accesses to `SYMSXP`

```
EXTERN_ATTRIBUTES extern BCell _RCP_CONSTCELL_AT_IMM0;
EXTERN_ATTRIBUTES extern BCell _RCP_CONSTCELL_AT_IMM1;
EXTERN_ATTRIBUTES extern BCell _RCP_CONSTCELL_AT_IMM2;
EXTERN_ATTRIBUTES extern BCell _RCP_CONSTCELL_AT_IMM3;
#define GETCONSTCELL_IMM(i) &_RCP_CONSTCELL_AT_IMM##i
```

Code listing 2.15 Constant argument constcell.

constants representing variable names. Because of this, they are accessed using the same argument indices as constants retrieved from the constant pool.

The only case where BCells are accessed differently than described is in `for` loops. The `STEPFOR` instruction must access the BCell that mirrors the loop iteration variable. Since this variable's symbol is provided as an immediate

```
RCP_OP(GETVAR) {
  PUSH_VAL(1);
  Rsh_GetVar(GET_VAL(1), GETCONST_IMM(0),
             GETCONSTCELL_IMM(0), GET_RHO());
  NEXT;
}
```

■ **Code listing 2.16** GETVAR stencil example.

argument to the `STARTFOR` instruction, and `STEPFOR` contains a label pointing to it, we can relocate the BCell accordingly. For this specialized use case, another set of `extern` variables is introduced, as shown in Code listing 2.17.

```
EXTERN_ATTRIBUTES extern BCell _RCP_CONSTCELL_AT_LABEL_IMM0;
EXTERN_ATTRIBUTES extern BCell _RCP_CONSTCELL_AT_LABEL_IMM1;
EXTERN_ATTRIBUTES extern BCell _RCP_CONSTCELL_AT_LABEL_IMM2;
EXTERN_ATTRIBUTES extern BCell _RCP_CONSTCELL_AT_LABEL_IMM3;
#define GETCONSTCELL_LABEL_IMM(i) &_RCP_CONSTCELL_AT_LABEL_IMM##i
```

■ **Code listing 2.17** Constant argument constcell.

The `STEPFOR` stencil also serves as an example of one of the few cases where control flow is conditionally determined by the return value of the Ř function. In most other cases, the functions return `void`. The `STEPFOR` stencil implementation is shown in Code listing 2.18.

```
RCP_OP(STEPFOR) {
  if (Rsh_StepFor(GET_VAL(3), GET_VAL(2), GET_VAL(1),
       GETCONSTCELL_LABEL_IMM(0), GET_RHO()))
    GOTO_IMM(0);
  else
    NEXT;
}
```

■ **Code listing 2.18** STEPFOR stencil.

Following this methodology, all R bytecode instructions supported by Ř were implemented using Ř instruction functions and custom relocation symbols. The resulting collection of stencils is contained in a single file, prepared for compilation in the next stage. This file is included in the attachments.

2.2 Additional Optimizations in Stencils

Given that stencils are central to the design of this project, their efficiency directly impacts overall performance. This section outlines several techniques and strategies employed to optimize stencil implementations, ensuring they execute as efficiently as possible while maintaining correctness and compatibility.

2.2.1 Eliminating Redundant Register Saving

Each stencil is implemented as a standalone function body. Without any extra information, the compiler assumes standard calling conventions apply, meaning function calls must follow regular rules for register preservation. On Linux systems using the `x86_64` architecture, this follows the System V ABI [17], which requires roughly half of the general-purpose registers to be preserved by the callee.

However, this assumption introduces unnecessary overhead in our use case. Stencil code is not invoked through conventional function calls, but rather executed sequentially from copy-patched machine code. Consequently, preserving callee-saved registers, only to restore them before returning, is redundant. This not only increases the size of each stencil but also introduces avoidable performance penalties.

Starting from version 14, GCC supports the `no_callee_saved_registers` function attribute. When applied to a function, this attribute informs the compiler that it is not responsible for saving or restoring any callee-saved registers. The burden of preserving registers is instead shifted to the caller, allowing for more efficient function definitions when the compiler can determine that certain registers are not live across calls.

GCC further optimizes such function calls by analyzing register usage: it saves only those registers that are live and omits saving those that are not. This results in reduced code size and improved performance. In scenarios where a call to a function marked with this attribute occurs as the final instruction in the caller's body, the compiler can completely eliminate register-saving operations.

This usage pattern aligns precisely with our stencil design. Each stencil concludes by transferring control to the next stencil via a placeholder function. When this placeholder is defined with the `no_callee_saved_registers` attribute, the compiler omits all instructions related to register preservation.

The impact of this optimization is illustrated in Code listing 2.19. The original stencil, adhering to the standard calling convention, includes instructions to push callee-saved registers onto the stack at the beginning of the function. In contrast, the optimized version, employing the aforementioned attribute, omits these instructions entirely, resulting in a more compact and efficient code segment.

To ensure compatibility with multiple compilers, the stencil source code includes a preprocessor directive that verifies compiler support for the required attribute (see Code listing 2.20). If a supported compiler is not detected, a

```
push %r15
push %r14
push %r13
push %r12
push %rbp
push %rbx
sub  $0x18, %rsp
mov  R_BCNodeStackTop(%rip), %r12
mov  -0x20(%r12), %r13d
mov  -0x10(%r12), %eax
mov  -0x18(%r12), %rbp
mov  -0x8(%r12), %rdi
cmp  $0xe, %r13d
jne  80 <_RCP_ADD_OP+0x80>
...
```

(a) Start of ADD stencil (original)

```
sub  $0x18, %rsp
mov  R_BCNodeStackTop(%rip), %r12
mov  -0x20(%r12), %r13d
mov  -0x10(%r12), %eax
mov  -0x18(%r12), %rbp
mov  -0x8(%r12), %rdi
cmp  $0xe, %r13d
jne  80 <_RCP_ADD_OP+0x80>
...
```

(b) Start of ADD stencil (optimized)

Code listing 2.19 Comparison for optimized version of the ADD stencil.

warning is issued, but the compilation proceeds (without enabling this optimization).

```
#if __GNUC__ >= 14
#define STENCIL_ATTRIBUTES \
__attribute__((no_callee_saved_registers)) __attribute__ ((noinline))
#else
#warning "Compiler does not support no_callee_saved_registers \
  directive. Generated code will be slower."
#define STENCIL_ATTRIBUTES \
__attribute__ ((noinline))
#endif
```

Code listing 2.20 Function attribute optimization and preprocessor check.

While this attribute significantly reduces code size and improves runtime efficiency, its effective use during stencil execution requires the calling environment to assume responsibility for register preservation. Typically, the compiler handles this automatically when invoking a function marked with `no_callee_saved_registers`. However, doing so necessitates that the calling environment (specifically, the R runtime in our case) be compiled with GCC version 14 or later.

At the time of writing, GCC 14 has not yet achieved widespread adoption across distributions. Requiring this specific version solely to support one optimization would be impractical and introduce deployment constraints.

To circumvent this limitation while still benefiting from the optimization, we introduce a special-purpose stencil, referred to as the *init stencil* (see

Code listing 2.21 and 2.22). This stencil is inserted exactly once at the beginning of every patched program. Its role is to proactively back up all general-purpose registers prior to the start of execution and to restore them upon program termination.

By handling register preservation at the entry and exit points of the program, this approach ensures that execution remains fully transparent to the R environment, even in the absence of a compiler that supports the required attribute. Thus, the optimization remains effective without imposing additional requirements on the runtime environment.

```
extern STENCIL_ATTRIBUTES SEXP _RCP_EXEC_NEXT(void);
SEXP _RCP_INIT () {
  return _RCP_EXEC_NEXT();
}
```

Code listing 2.21 Init function.

```
push  %r15
push  %r14
push  %r13
push  %r12
push  %rbp
push  %rbx
push  %rax
call  _RCP_EXEC_NEXT@PLT
pop   %rdx
pop   %rbx
pop   %rbp
pop   %r12
pop   %r13
pop   %r14
pop   %r15
ret
```

Code listing 2.22 Init function compiled.

This optimization reduced the average stencil code size by approximately 2% without requiring changes to individual stencil implementations.

2.2.2 Patching Values Instead of Addresses

As outlined earlier, the Copy-and-Patch approach relies on patching values into stencils. However, current compilers do not allow this directly, or at least not intuitively.

It was previously established that immediate bytecode values would be patched using `extern` variables. The compiler emits relocations for these variables, and an external tool extracts this information to register the patch locations and contents.

But even if such a variable is declared `const`, the compiler still generates a relocation to its *address*. Since we already know the value to be patched, storing its address instead introduces an unnecessary level of indirection.

However, when the stencil takes the address of an `extern` variable (using the `&` operator), this address is relocated as a value.

We exploit this behavior by never accessing the external variables directly. Instead, we always take their address and cast it to the required type. Since the relocations are handled by our own tooling, we can safely patch in the actual value rather than the address, contrary to what the compiler would normally expect.

Instead of the symbols shown in earlier Code listings 2.6 and 2.8, stencils now use the pattern shown in Code listing 2.23.

```
extern const void* const _RCP_RAW_IMM0[];
extern const void* const _RCP_RAW_IMM1[];
extern const void* const _RCP_RAW_IMM2[];
extern const void* const _RCP_RAW_IMM3[];
#define GET_IMM(index) (unsigned)(int64_t)&_RCP_RAW_IMM##index

extern const void* const _RCP_CONST_AT_IMM0[];
extern const void* const _RCP_CONST_AT_IMM1[];
extern const void* const _RCP_CONST_AT_IMM2[];
extern const void* const _RCP_CONST_AT_IMM3[];
#define GETCONST_IMM(i) (const SEXP const)(&_RCP_CONST_AT_IMM##i)
```

Code listing 2.23 Direct value patching of constants.

These symbols are declared as `extern` arrays to force the compiler to emit absolute 64-bit relocations, even in non-large memory models. This is necessary for accessing arbitrary values such as integers or SEXP constants.

This technique assumes that patched values remain constant. If a value were to change at runtime, it would not be visible to the stencil. Therefore, this is safe only for truly immutable variables. In contrast, accesses to variables such as `BCell` or `rho` are modified by the R runtime, and therefore cannot make use of this optimization.

A comparison of compiled stencils with and without this optimization is provided in Code listing 2.24.

While the number of instructions remains the same, the optimized version uses a MOV variant that loads an immediate value directly into a register. In contrast, the unoptimized version performs a memory load from an address.

An additional performance gain may result from the compiler's assumption

```
mov  R_BCNodeStackTop(%rip), %rax
lea  0x10(%rax), %rdx
mov  %rdx, R_BCNodeStackTop(%rip)
xor  %edx, %edx
movl $1, R_Visible(%rip)
mov  %edx, (%rax)
mov  _RCP_CONST_AT_IMM0(%rip),
       %rdx
mov  %rdx, 0x8(%rax)
jmp   _RCP_EXEC_NEXT@PLT
```

(a) LDCONST (relocating address).

```
mov  R_BCNodeStackTop(%rip), %rax
lea  0x10(%rax), %rdx
mov  %rdx, R_BCNodeStackTop(%rip)
movabs _RCP_CONST_AT_IMM0(%rip),
    %rcx
xor  %edx, %edx
movl $1, R_Visible(%rip)
mov  %rcx, 0x8(%rax)
mov  %edx, (%rax)
jmp  _RCP_EXEC_NEXT@PLT
```

(b) LDCONST (relocating value).

Code listing 2.24 Comparison of relocating address vs value.

that a patched SEXP value, now appearing as an immediate, will not change. When only the address is known, the compiler must assume the underlying value may be updated externally, requiring reloading on each use. With this optimization, the compiler can safely assume immutability and skip redundant loads.

2.2.3 Omitting Stack Overflow Check

Each time a value is pushed to the stack, a safety check is performed to detect potential stack overflow. While this is the safest option, it introduces conditional branches into every stencil, which can degrade performance.

Because the stack macro is inlined and stencil bodies are replicated into memory via the Copy-and-Patch mechanism, these branches are executed only once per program path. It could be argued that this limits opportunities for branch prediction to learn and optimize such patterns, potentially resulting in frequent mispredictions.

To address this, we introduce a preprocessor macro that allows the safety check to be disabled entirely. The modified `PUSH_VAL` macro is shown in Code listing 2.25.

A comparison of compiled stencils with and without this optimizatio, using the `LDTRUE` instruction, is provided in Code listing 2.26.

2.2.4 Optimizing Pushing to Stack

The macro responsible for pushing values onto the stack, shown in Code listing 2.3, is not as optimized as it could be.

To avoid problems with the garbage collector, each value that should be pushed to stack needs to already have an assigned space there. There should not be a time where, in between R runtime calls, there is an unattended SEXP

```
#ifndef NO_STACK_OVERFLOW_CHECK
#define CHECK_OVERFLOW(__n__)                                         \
do {                                                                  \
  if (__builtin_expect(R_BCNodeStackTop + __n__ >                     \
  R_BCNodeStackEnd, 0)) {                                             \
    nodeStackOverflow();                                              \
  }                                                                   \
} while (0)
#else
#define CHECK_OVERFLOW(__n__)
#endif

#define PUSH_VAL(n)                                                   \
do {                                                                  \
  int __n__ = (n);                                                    \
  CHECK_OVERFLOW(__n__);                                              \
  while (__n__-- > 0) {                                               \
    SET_SXP_VAL(R_BCNodeStackTop++, R_NilValue);                      \
  }                                                                   \
} while (0)
```

Code listing 2.25 Optimized PUSHVAL.

```
mov  R_BCNodeStackTop(%rip), %rax
lea  0x10(%rax), %rdx
cmp  %rdx, R_BCNodeStackEnd(%rip)
jb   2d <_RCP_LDTRUE_OP+0x2d>
mov  %rdx,R_BCNodeStackTop(%rip)
movl $0x1, 0x8(%rax)
movl $0xa, (%rax)
jmp  _RCP_EXEC_NEXT@PLT
push %rax
call 33 <_RCP_LDTRUE_OP+0x33>
```

(a) LDTRUE with safe stack push.

```
mov  R_BCNodeStackTop(%rip), %rax
lea  0x10(%rax), %rdx
movl $0x1, 0x8(%rax)
mov  %rdx, R_BCNodeStackTop(%rip)
movl $0xa, (%rax)
jmp  _RCP_EXEC_NEXT@PLT
```

(b) LDTRUE with fast stack push.

Code listing 2.26 Comparison of STACK_PUSH safety optimization.

value anywhere in the stencil. Therefore, the values are changed directly on stack.

This requires stencils that grow the stack to first increase the stack size and insert a placeholder value for each slot. In the original implementation, `R_NilValue` was used as the placeholder. While logically correct, this choice introduces inefficiencies due to how it interacts with the Ř stack optimization.

As discussed earlier, Ř introduces an optimization that allows stack entries to hold primitive values (`int`, `double`, or `logical`) instead of relying exclusively on full SEXP objects. This is done to avoid the overhead of converting SEXP into primitive types every time there is an arithmetic operation.

This specialization is implemented as a tagged union, with the union values being the mentioned primitive types + SEXP (pointer) for cases where this optimization cannot be applied.

When a SEXP is placed on the stack, the union tag is set to zero and the pointer is stored in the SEXP field.

The compiler is intelligent, and when a code sets a value into a variable and later overwrites it, the first write is optimized out. There is potential for this optimization in the stencil code, as the empty values pushed on stack could be left out, if is it overwritten right after.

However, because the place to be overwritten is a union (and not all datatypes inside it have the same size), the compiler cannot/does not omit the first write out, when the following write could be of a smaller size.

This means that in some stencils, there is an unnecessary write to memory at the start of the stencil.

To address this, a different placeholder strategy is used: instead of initializing the slot with `R_NilValue`, we set the tag to indicate a primitive type and leave the value field unchanged. Since primitive-tagged stack entries are ignored during garbage collection, this is semantically safe. More importantly, the initialization now consists of a single memory write (just the tag) which can be optimized out if the field is overwritten shortly thereafter.

The revised `PUSH_VAL` macro is shown in Code listing 2.27.

```
 1  #define PUSH_VAL(n)                                         \
 2  do {                                                        \
 3    int __n__ = (n);                                          \
 4    if (R_BCNodeStackTop + __n__ > R_BCNodeStackEnd) {        \
 5      CHECK_OVERFLOW(__n__);                                  \
 6    }                                                         \
 7    while (__n__-- > 0) {                                     \
 8      (R_BCNodeStackTop++)->tag = INTSXP;                     \
 9    }                                                         \
10  } while (0)
```

Code listing 2.27 Optimized PUSHVAL.

Only Line 8 is changed: instead of using `SET_SXP_VAL`, we directly set the tag and advance the stack pointer.

A real-world comparison using the `LDTRUE` instruction is shown in Code listing 2.28. For simplification, the overflow check described in Section 2.2.3 is also disabled in this comparison.

The comparison was done with the real instruction LDTRUE. A stack push is performed, after which its value is set to logical one, using the Ř optimizing union.

```
mov  R_BCNodeStackTop(%rip), %rax
lea  0x10(%rax), %rdx
mov  %rdx,R_BCNodeStackTop(%rip)
mov  R_NilValue(%rip), %rdx
movl $0xa, (%rax)
mov  %rdx, 0x8(%rax)
movl $0x1, 0x8(%rax)
jmp  _RCP_EXEC_NEXT@PLT
```

(a) LDTRUE with old stack push.

```
mov  R_BCNodeStackTop(%rip), %rax
lea  0x10(%rax), %rdx
movl $0x1, 0x8(%rax)
mov  %rdx, R_BCNodeStackTop(%rip)
movl $0xa, (%rax)
jmp  _RCP_EXEC_NEXT@PLT
```

(b) LDTRUE with optimized stack push.

Code listing 2.28 Comparison of STACK_PUSH optimization.

The old method can be seen using the R_NilValue, even though it's completely unnecessary in this stencil.

2.2.5 Efficient Memory Access to Internal Relocations

When the large memory model is not in use, the compiler is capable of generating shorter and more efficient memory access instructions using relative addressing.

In the medium memory model, whether the compiler emits relative or absolute addressing for external symbols depends on the size of the referenced variable. For variables exceeding the size of a pointer, the compiler defaults to using absolute addressing, leading to longer and less efficient instructions.

The Ř runtime defines several `extern` SEXP arrays required for optimizations. Since the compiler cannot know their runtime size, it conservatively chooses absolute addressing. However, we know these symbols will always reside within reach for relative 32-bit access

To use the smaller memory access instructions for these `extern` symbols, we can annotate them with a compiler attribute that convinces the compiler that these symbols are near enough and can be relocated in the more efficient way.

This requires a minor change in how the arrays are declared in the Ř source. While this optimization could theoretically apply to the baseline Ř as well, we limit it to stencils by using a preprocessor macro that activates the attribute only when included from a stencil source file.

The macro is applied to all internal runtime arrays that are relocated from within the stencil code.

2.2.6 Stencil Specialization

Stencil specialization is an optimization approach already explored in the original article that inspired this work [2]. However, the scope for specialization

```
#ifdef RCP
#define EXTERN_ATTRIBUTES __attribute__((section(".data"), visibility
    ("hidden")))
#else
#define EXTERN_ATTRIBUTES
#endif
```

■ **Code listing 2.29** Ř extern attributes.

is more limited in R bytecode, as stencils typically operate on stack values, which are unknown at compile time.

Nevertheless, some bytecode instructions contain immediate operands, and this information *is* available during compilation. In certain cases, these immediate values enable useful optimizations.

The Ř project implements specialized versions of stencils for instructions where part of the runtime logic can be moved to compile time by leveraging the known immediates.

A primary example is the LDCONST instruction, which loads a constant from the constant pool based on an immediate index and pushes it onto the stack. Although this constant is always a SEXP, Ř's stack optimization allows stack entries to hold primitive types: `int`, `double`, or `logical` instead.

Since the type of each constant is known at compile time, the conversion from SEXP to a primitive type can be performed ahead of time. A specialized stencil is generated to directly load the primitive value, thereby reducing runtime work.

Each specialized version is implemented as a standalone stencil with a distinct name. This name allows the Copy-and-Patch stage to select the appropriate variant during code generation (described in Section 2.5.4). These variants are illustrated in Code listing 2.30.

A similar specialization strategy is applied to the `MATH1` instruction, which includes an immediate specifying which mathematical function to apply. Because the program will know during compile time which math operation it uses (based on its argument), this function can be specialized to each mathematical operation.

Code listing 2.31 shows two variants of the MATH1 stencil, each specialized to a specific mathematical operation. For clarity, only the differing basic block is shown. Each version contains a direct call to the corresponding math function, bypassing indirection and improving performance.

```
RCP_OP(LDCONST_INT) {
  PUSH_VAL(1);
  Rsh_LdConstInt(GET_VAL(1), GETCONST_IMM(0));
  NEXT;
}
RCP_OP(LDCONST_DBL) {
  PUSH_VAL(1);
  Rsh_LdConstDbl(GET_VAL(1), GETCONST_IMM(0));
  NEXT;
}
RCP_OP(LDCONST_LGL) {
  PUSH_VAL(1);
  Rsh_LdConstLgl(GET_VAL(1), GETCONST_IMM(0));
  NEXT;
}
RCP_OP(LDCONST_SEXP) {
  PUSH_VAL(1);
  Rsh_LdConst(GET_VAL(1), GETCONST_IMM(0));
  NEXT;
}
```

■ **Code listing 2.30** Optimized LDCONST stencils.

```
...
vmovsd   -0x8(%rbx), %xmm1
vmovapd  %xmm1, %xmm0
vmovsd   %xmm1, 0x8(%rsp)
call     floor
vmovsd   0x8(%rsp), %xmm1
vucomisd %xmm0, %xmm0
jp       5e <_RCP_MATH1_0_OP+0x5e>
...
```

(a) MATH1 stencil with argument 0 (floor).

```
...
vmovsd   -0x8(%rbx), %xmm1
vmovapd  %xmm1, %xmm0
vmovsd   %xmm1, 0x8(%rsp)
call     ceil
vmovsd   0x8(%rsp), %xmm1
vucomisd %xmm0, %xmm0
jp       5e <_RCP_MATH1_1_OP+0x5e>
...
```

(b) MATH1 stencil with argument 1 (ceil).

■ **Code listing 2.31** Example of specialized MATH1 stencils compiled.

As can be seen in the middle of the basic block, each stencil directly calls a different mathematical function.

2.3 Stencil Compilation

For our implementation, we selected the `x86_64` architecture running on a Linux operating system. This choice is motivated by the widespread adoption of this platform, as well as its proven compatibility with Ř.

The stencil source file is compiled into an object file, which serves as an intermediate representation from which stencil functions can be extracted. This object file must conform to the Executable and Linkable Format (ELF), the standard binary format used on Unix-like systems.

To compile the stencil code, we employed the GNU Compiler Collection (GCC). Our selection of GCC is primarily influenced by its support, beginning in version 14, for the `no_callee_saved_registers` function attribute. This attribute plays a role in one of our Copy-and-Patch optimizations (detailed in Section 2.2.1).

The stencil code is compiled with a specific set of compiler flags. Each option is selected for a particular purpose, aimed at ensuring compatibility with the function extraction process and the runtime system. The table 2.1 summarizes these options, their roles, and their relevance to our project.

It should be noted that the use of the medium memory model may not be suitable for all target systems. In environments where the operating system randomizes the memory layout of dynamically loaded libraries (a common security feature), the copy-patch compiler described in the next phase may be unable to express all necessary pointers using relative addressing. In such cases, the stencil code must be recompiled using the large memory model to ensure correctness.

Because the Copy-and-Patch approach relies on the ability to lay out machine code sequentially in memory, code alignment is not feasible. This limitation prevents the use of several optimizations typically enabled by default in GCC. Specifically, it is necessary to disable function and control flow alignment via the `-fno-align-functions`, `-fno-align-loops`, `-fno-align-jumps`, and `-fno-align-labels` flags.

Fortunately, the `-Os` optimization level implicitly disables these alignment optimizations while retaining all other optimizations available under `-O2`, making it an ideal fit for our use case [18].

Optimization option that focuses on size aggressively (`-Oz`) is not used, as it was tested that the average stencil size decreases by just 2%. Given that `-Oz` tends to disable certain performance-related optimizations in pursuit of size reduction, it was concluded that the trade-off was not justified.

All options combined produce the following prompt:

```
gcc stencils.c -o stencils.o -c -ffunction-sections -Os -march=native
     -fno-stack-protector -fcf-protection=none -fno-asynchronous-
    unwind-tables -fno-pic -mcmodel=medium -fno-merge-constants -fno-
    jump-tables
```

2.4 Stencil Extraction

After the object file is produced, a program is run to extract machine instructions and associate them with the corresponding R instructions, while also identifying the holes and their types based on relocation symbols.

Option	Description	Justification in Project
`-c`	Compile only; skip linking and produce an object file.	Required to generate an object file from which individual functions can be extracted using our extraction tool.
`-ffunction-sections`	Places each function in its own section in the object file.	Enables easier extraction of individual functions, as each resides in a separate section.
`-Os`	Optimize for size (with a balance for speed).	Important due to our Copy-and-Patch approach; keeping code size small minimizes memory overhead.
`-march=native`	Use extended instruction set of the CPU the compiler is running on	Uses the full potential of the CPU. Because stencils are to be compiled on each system during installation, it is safe to use.
`-fno-stack-protector` `-fcf-protection=none`	Disables stack canaries and control-flow protection.	These security features are unnecessary and would complicate analysis and patching; thus, they are disabled.
`-fno-asynchronous-unwind-tables`	Disables generation of unwind tables for exception handling or debugging.	Reduces object file size and avoids clutter, as we do not require exception handling.
`-fno-pic`	Disables generation of position-independent code.	Since the code will be patched to fixed memory addresses, this improves performance by avoiding indirection.
`-mcmodel=medium`	Uses the medium memory model, allowing larger address ranges where needed.	Necessary for accessing larger R runtime variables potentially located beyond lower 2 GB of memory.
`-fno-merge-constants`	Avoids merging identical constants and places them in a single section.	Simplifies analysis and extraction, as read-only data is centralized.
`-fno-jump-tables`	Disables generation of jump tables for switch statements.	Prevents the need to patch entire tables upon function copying, simplifying the patching process and minimizing code size.

Table 2.1 Compiler options used and their justifications in the project.

Once all the data is collected, it must be exported in a format suitable for use by the Copy-and-Patch compiler. Since reduced compilation time is one of the key advantages of this approach, as much processing as possible should be offloaded to the extraction tool. This means that the output format must be trivial to parse and integrate.

To meet this requirement, stencils are exported as C header files, which can be included in the compiler directly via the `#include` directive. This avoids any runtime parsing or file I/O overhead and effectively embeds the stencil data as part of the compiler binary.

2.4.1 Parsing the Object File

The extraction program is written in C and uses the Binary File Descriptor library [19] to parse the object file and extract the necessary information.

The program iterates over all sections in the object file. Each section carries metadata flags that describe its contents. Of particular interest are sections marked with `SEC_CODE`, indicating that the section contains executable instructions.

The only non-code section that is extracted is `.rodata`, which contains constants that the compiler was unable to inline, typically C strings and floating-point constants. This section is notable because it requires maximum system alignment when laid out in memory, which becomes relevant during stencil memory placement in the later phases of the project.

As the compiler was invoked with the `-ffunction-sections` flag, each function resides in its own section. This convention conveniently encodes the function name into the section name, simplifying the task of splitting code into separate stencils.

When the program encounters such a function section, it extracts its contents (raw executable bytes) and records them alongside the function's name. It then examines any associated relocation entries and records them as the holes within the stencil.

Each relocation has a type that specifies how it should be applied. The current implementation supports five relocation types (see Table 2.2), which covers all observed patterns using the selected compilation settings [2].

For each relocation, the program records:

- The offset in the instruction stream where the relocation occurs.
- The relocation addend – an integer value added during the patching phase.
- The relocation size
- Whether it is relative or absolute.

[2]When a large memory model is chosen during stencil compilation, only one of these is used (64-bit absolute).

Internal Name	Description	Relocation size
`R_X86_64_PLT32`	PC-Relative address of a function	32-bit
`R_X86_64_PC32`	PC-Relative address	32-bit
`R_X86_64_32`	Absolute address	32-bit
`R_X86_64_32S`	Absolute address	32-bit (signed)
`R_X86_64_64`	Absolute address	64-bit

Table 2.2 Supported relocation types.

This information is essential for reconstructing and correctly patching the executable code during the final stage of the pipeline.

Most importantly, every relocation entry corresponds to a reference to some symbol or data. The extractor must resolve what exactly is being referenced, so that this information can be exported alongside the stencil hole metadata.

In standard linking and loading, this mapping is done via symbol names. For example, consider the following code:

```c
#include <math.h>
extern double result;
void example(double a) {
  global_result = pow(a, 3);
}
```

When compiled, this function can produce the following assembly:

```
1 subq  $8, %rsp
2 movsd .LC0(%rip), %xmm1
3 call  pow
4 movsd %xmm0, result(%rip)
5 addq  $8, %rsp
6 ret
```

The example code contains three relocations: a constant literal (`3`) on Line 2, a function call to `pow` from the standard math library on Line 3, and an external global variable `result` on Line 4. The disassembler visualizes this, but in the actual object file, the executable section does not embed these symbol names directly – the immediate operands are zeroed.

Instead, this symbol information is stored in the object file's relocation table. The constant is emitted into the `.rodata` section, and its address is inserted via relocation. The other two (function and global variable references) are symbol relocations, where the linker performs a textual match on the symbol name and fills in the actual address at link time (or produces an error and halts, if unresolved).

When parsing the object file during stencil extraction, the relocation itself cannot be performed just yet – it can not be known where these symbols will live in memory at runtime.

Instead, the symbol name is recorded and exported as part of the stencil metadata in the generated header file. The Copy-and-Patch compiler can then delegate the task of resolving these symbols to the dynamic loader at runtime (see Section 2.4.3).

However, this approach is insufficient for the stencil design requirements. As discussed in Section 2.1.1, stencils often use external symbols to also allow for patching of their arguments or special values (for an example, see Code listing 2.14). These do not and can not work as regular patch symbols, as their address/value is supposed to change based on what their immediate bytecode value is. This compiler trick requires close coordination with the stencil extractor.

The extraction program determines these internal relocation types solely by the name of the symbol being reallocated, so it is important that the stencils use the exact symbols names that the extraction tool expects.

All supported internal relocation types are enumerated in Table 2.3, along with their meaning and how frequently they appear in real-world stencils. Internal names from the fourth row onward are also used as relocation symbols, when they are prefixed with an underscore (as seen in previous code listings).

First two hole types are reserved for standard relocations. The third type, `RCP_PRECOMPILED`, is used to refer to Ř's precompiled SEXP symbols (explained later). The remaining entries correspond to stencil-specific relocations.

Starting from the fifth row of the table, hole types correspond to relocations where the address patched depends dynamically on the bytecode's immediate value. The other types represent static addresses, consistent across all uses of the stencil.

To support multiple arguments per stencil, the argument's ordinal is encoded in the symbol name. If the symbol ends with `_IMM`, it must be followed by an integer (e.g., `RCP_RAW_IMM1`). The extractor parses this suffix, converts it to an integer, and stores it in the hole descriptor.

Ř applies an optimization to avoid recreating constant SEXP symbols and primitives repeatedly. For example, the ADD instruction requires access to the symbol "+" each time it executes. Instead of reconstructing it each time, Ř initializes a set of these constants once at startup and stores them in global arrays accessible to all instructions.

To integrate seamlessly with this mechanism, the extractor contains a list of all such arrays used by Ř and changes this relocation to only one array that will be given a type of `RCP_PRECOMPILED`. This allows stencil code to use these cached symbols efficiently without modifying Ř itself.

2.4.2 Extra Optimizations

One of the key assumptions in the Copy-and-Patch approach is that tail call optimization will be applied. In the stencils, control flow is passed to the next operation by calling a special placeholder function at the end of execution.

However, since stencils are not actually invoked as standalone functions, but rather copied into memory in a flat, sequential layout, this `JMP` becomes

Internal Name	What is Patched	Where It Is Used	Frequency
`RUNTIME_SYMBOL`	Address of any symbol available at runtime	Any access to external variables or functions	Very Often
`RODATA`	Address of the `.rodata` section	Access to values placed in `.rodata` (e.g., strings, floats)	Sometimes
`RCP_PRECOMPILED`	Address of an array of precompiled SEXP values	Access SEXP values computed at compile time	Rare
`RCP_RHO`	Address of variable specifying the current execution environment	Access to the current runtime environment	Often
`RCP_EXEC_NEXT`	Address of next instruction's code start	Stencil passes control flow to the next instruction	Often
`RCP_EXEC_IMM`	Address of the code for a given label	Stencil jumps to a specified label	Sometimes
`RCP_RAW_IMM`	Integer value of instruction immediate	Accessing the stencil's raw immediate argument	Rare
`RCP_CONST_AT_IMM`	SEXP constant pointed to by instruction immediate	Accessing the stencil's SEXP immediate argument	Very Often
`RCP_CONST_STR_AT_IMM`	Pointer to a string representing symbol name	Accessing the symbol name from a SEXP constant	Very Rare
`RCP_CONSTCELL_AT_IMM`	Address of a BCell for an instruction immediate	Ř cache optimization for variable retrieval	Sometimes
`RCP_CONSTCELL_AT_LABEL_IMM`	Address of a BCell from instruction at a given label	Ř cache optimization used in for loops	Very Rare

Table 2.3 Relocation Types for Stencil Holes.

redundant. It simply jumps to the instruction immediately following it, which would have been executed next anyway. In this context, the `JMP` serves no purpose – it's effectively a `NOP`.

This makes it safe and useful to remove the instruction entirely. Since it is always the final instruction in the function, there is no need to adjust or re-patch any surrounding code.

The extraction tool implements this optimization by scanning the final instruction of each stencil. If it detects an unconditional jump to the known identifier, it simply strips it from the output. The relevant code can be seen in Code listing 2.32, and a before/after comparison is shown in Code listing 2.33.

This optimization, however, is only applicable when using the medium memory model. In the large memory model, tail calls to the next stencil are often compiled into a sequence of two or more instructions (e.g., loading an address followed by an indirect jump), which may not be contiguous or easily

```
if (strcmp(descr, "EXEC_NEXT") == 0) {
  if (rel->address - rel->addend == stencil->body_size &&
       stencil->body[rel->address - 1] == 0xE9 /*JMP*/)
  {
    // This is the last instruction; safe to just delete
    stencil->body_size = rel->address - 1;
    return; // No relocation from this
  }
  else
    hole->kind = RELOC_RCP_EXEC_NEXT;
}
```

Code listing 2.32 Removing JMP at the end of stencil

```
subq $0x10,R_BCNodeStackTop(%rip)
jmp  _RCP_GOTO_NEXT@PLT
```

(a) POP stencil (original).

```
subq $0x10,R_BCNodeStackTop(%rip)
```

(b) POP stencil (optimized).

Code listing 2.33 Comparison of optimized version for the POP stencil.

detected. Handling this would require deeper instruction analysis and a more advanced removal strategy, which has not (yet) been implemented.

2.4.3 Exporting

During object file parsing, the program stores stencils in two categories. When it encounters a function, it checks the symbol name against a list of known R instructions. If a match is found, the stencil is registered in a lookup table using the instruction's opcode. If not, the stencil is added to a separate list of named stencils, intended for specialized or internal use.

Once all object files are parsed, the program begins exporting the discovered stencils.

A standard C header file is created for each stencil, where its executable code is exported as a byte array, initialized with a brace-enclosed list. Alongside the code, a list of holes is exported, describing every detail needed to patch the stencil: offsets, sizes, types, and patch values. When a relocation targets a runtime symbol, the header includes it as a reference to a C symbol using the `&` operator. This allows the dynamic loader of the Copy-and-Patch compiler to resolve the address automatically at runtime.

The header generation logic and resulting output are shown in Code listings 2.34 and 2.35, respectively.

All generated header files are then included from a single main header file, `stencils.h`, which builds the global instruction lookup table. Each implemented instruction is mapped to its corresponding stencil; unimplemented

```
void export_body(FILE *file, const StencilMutable *stencil,
                 const char *instruction_name)
{
  fprintf(file, "const Hole _%s_HOLES[] = {\n", instruction_name);
  for (size_t j = 0; j < stencil->holes_size; ++j)
  {
    const Hole *hole = &stencil->holes[j];

    fprintf(file,
    "{ .offset = 0x%lX, .addend = %ld, .size = %hu, .kind = %u, \
     .is_pc_relative = %u, .indirection_level = %u",
    hole->offset, hole->addend, hole->size, hole->kind,
    hole->is_pc_relative, hole->indirection_level);

    switch (stencil->holes[j].kind)
    {
    case RELOC_RUNTIME_SYMBOL:
    fprintf(file, ", .val.symbol = &%s",
            stencil->holes[j].val.symbol_name);
    break;
    case RELOC_RCP_EXEC_IMM:
    case RELOC_RCP_RAW_IMM:
    case RELOC_RCP_CONST_AT_IMM:
    case RELOC_RCP_CONST_STR_AT_IMM:
    case RELOC_RCP_CONSTCELL_AT_IMM:
    case RELOC_RCP_CONSTCELL_AT_LABEL_IMM:
    fprintf(file, ", .val.imm_pos = %zu",
            stencil->holes[j].val.imm_pos);
    break;
    }

    fprintf(file, " },\n");
  }

  fprintf(file, "};\n\n");
  fprintf(file, "const uint8_t _%s_BODY[] = {\n", instruction_name);
  print_byte_array(file, stencil->body, stencil->body_size);
  fprintf(file, "\n};\n\n");
}
```

Code listing 2.34 Function for exporting stencils.

ones are assigned an empty entry. Named stencils outside of the table remain accessible by their string identifiers. This main file also includes the `.rodata` byte array.

The structure and layout of `stencils.h` can be seen in Code listing 2.36.

```
const Hole __RCP_LDCONST_SEXP_OP_HOLES[] = {
{ .offset = 0x3, .addend = -4, .size = 4, .kind = 0, .is_pc_relative
    = 1, .indirection_level = 1, .val.symbol = &R_BCNodeStackTop },
...
{ .offset = 0x1D, .addend = 0, .size = 8, .kind = 7, .is_pc_relative
    = 0, .indirection_level = 0, .val.imm_pos = 0 },
...
{ .offset = 0x38, .addend = -4, .size = 4, .kind = 3, .is_pc_relative
     = 1, .indirection_level = 0 },
};
const uint8_t __RCP_LDCONST_SEXP_OP_BODY[] = {
  0x48, 0x8B, 0x05, ... , 0xE8, 0x00, 0x00, 0x00, 0x00,
};
};
```

Code listing 2.35 Example of exported stencil (compacted).

```
#include "RETURN_OP.h"
#include "GOTO_OP.h"
#include "BRIFNOT_OP.h"
#include "POP_OP.h"
#include "DUP_OP.h"
...
#include "SEQLEN_OP.h"

uint8_t rodata[] = { 0x66, 0x61, 0x63, ..., 0xFF, 0xDF, 0x41, };

const Stencil stencils[129] = {
  {0, NULL, 0, NULL}, // BCMISMATCH_OP
  {100, _RETURN_OP_BODY, 7, _RETURN_OP_HOLES}, // RETURN_OP
  {5, _GOTO_OP_BODY, 1, _GOTO_OP_HOLES}, // GOTO_OP
  {210, _BRIFNOT_OP_BODY, 13, _BRIFNOT_OP_HOLES}, // BRIFNOT_OP
  {8, _POP_OP_BODY, 1, _POP_OP_HOLES}, // POP_OP
  {47, _DUP_OP_BODY, 5, _DUP_OP_HOLES}, // DUP_OP
  ...
  {368, _SEQLEN_OP_BODY, 20, _SEQLEN_OP_HOLES}, // SEQLEN_OP
}
};
```

Code listing 2.36 Header file including all stencils.

2.5 Execution Engine

The project is distributed as an R package. Inside the package, the C language is used to implement all functionality. All stencils extracted in the previous step are included as header files, and are therefore compiled into the dynamic library loaded with the package.

Once installed, the package can be loaded into the R runtime, where it becomes capable of compiling closures using the Copy-and-Patch approach.

The main entry point is implemented to resemble `compiler::cmpfun`. Its first argument is the closure to compile; the second, optional, argument is a list of bytecode compiler options, which are forwarded internally when the compilation is triggered.

The function returns a new closure that can be executed from the R environment like any regular R function, but internally runs native, copy-patched `x86_64` machine code.

```
fun <- function() {
  print("Hello, world!")
}
res = rcp::rcp_cmpfun(fun, options = list(optimize = 3))
res()
# [1] "Hello, world!"
```

Code listing 2.37 Package example use.

Logically, it could be split into these phases:

1. Bytecode compilation
2. Preparation run
3. Memory allocation
4. Stencil selection
5. Copying and patching
6. Finalization

2.5.1 Bytecode Compilation

As the Copy-and-Patch approach relies on bytecode as its input format, the closure provided as an argument must be compiled using R's native bytecode compiler, if it has not been already compiled via JIT.

This is achieved by calling the `compiler::cmpfun` R function from within the package, which returns a new closure whose body is of type `BCODESXP`.

If a second argument is passed to the function in our package, it is forwarded to the bytecode compiler. This allows, among other things, the specification of optimization levels. This is particularly important, as usage of Ř currently requires compilation with optimization level 3.

Once bytecode compilation is complete, the result must be extracted into a format that the rest of the system can understand. This is done using `R_bcDecode`, which decodes the bcode object into a `VECSXP` of instructions.

Following the decoding, the program accesses the internal pointer of this object to retrieve a C-array of instructions, each followed by its arguments (immediate values), if it has any. Likewise, the constant pool pointer is retrieved directly from the `BCODESXP` object.

```
SEXP bcode_code = BCODE_CODE(bcode);
SEXP bcode_consts = BCODE_CONSTS(bcode);

SEXP code = PROTECT(R_bcDecode(bcode_code));

int* bytecode = INTEGER(code) + 1;
int bytecode_size = LENGTH(code) - 1;

SEXP* consts = DATAPTR(bcode_consts);
int consts_size = LENGTH(bcode_consts);

rcp_exec_ptrs res = copy_patch_internal(bytecode, bytecode_size,
    consts, consts_size);
UNPROTECT(1); // code
...
```

Code listing 2.38 Decoding bytecode.

2.5.2 Preparation Run

The program first iterates over the bytecode instructions in a preparation phase, performing several tasks required for subsequent stages.

2.5.2.1 Instruction Validation

If the program encounters invalid opcodes or instructions that are not supported by Ř, it raises an error and halts before any further processing is performed.

2.5.2.2 Sizes and Lookup Tables

To allocate the required memory space, the program must determine the total size in advance. For the code section, this is done by summing the sizes of all stencils based on their usage frequency in the program.

During this phase, a code lookup table is constructed. This table contains a prefix sum representing the start addresses of each instruction's stencil within the allocated memory region.

This structure is crucial for instructions that involve jumps (e.g., `GOTO`), as it allows the program to resolve the jump offsets during the patching step. Without this mechanism, jumping to a yet-unprocessed instruction would necessitate a second patching pass, introducing unnecessary complexity.

Additionally, a lookup table for `BCell` accesses is created. A naive implementation would assign each constant index in the constant pool a unique `BCell`, resulting in wasted memory since only a subset of constants is ever accessed using the `BCell` optimization.

Instead, the program identifies only the constants that are actually accessed via `BCell`, allocates just enough memory for them, and uses the lookup table to map each accessed constant to its corresponding `BCell` during patching.

2.5.3 Memory Allocation

After all required sizes are computed, memory can be allocated.

Memory placement is particularly important in this context. When using the smaller and faster memory model, the generated stencil code assumes that all external symbols reside within the first 2 GiB of memory, near the stencil code itself.

Because code is generated and patched at runtime (i.e., JIT-compiled), the external symbols may reside too far from the allocated stencil memory, causing address resolution issues.

To address this, two strategies are employed: placing the memory as close as possible to R runtime symbols, and allocating a contiguous block for both code and runtime variables, ensuring proximity and valid relative addressing. The memory layout proposed in the following sections can be seen in Figure 2.1.

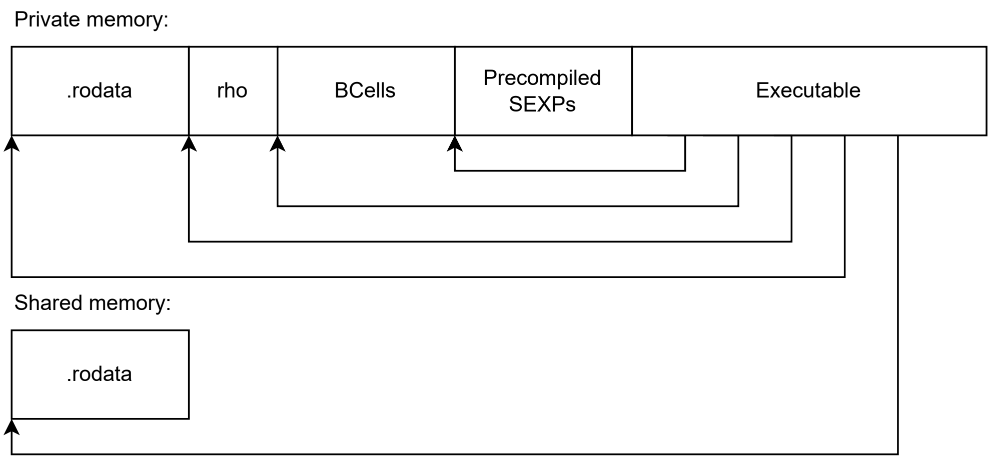


Figure 2.1 Memory layout.

2.5.3.1 Private Memory

To find an appropriately placed memory region, a helper function is used to locate the nearest available memory address to a given base address, satisfying the required size constraint.

This is achieved by parsing the `/proc/self/maps` file, which lists allocated and free memory regions under Linux. The function identifies the nearest free block large enough to accommodate the requested memory.

The function is called with an address close to any known R runtime function to ensure relative proximity within 2 GiB for `x86_64` relative addressing.

The allocated space is then divided into the following segments:

- Read only section of the stencil executable (`.rodata`)
- Current environment `SEXP` (set at runtime)
- `BCell` slots used by the compiled closure (reset at runtime)
- Precompiled `SEXP` symbols
- Executable stencil code for copy-patching

Once the total memory size is determined, including code and runtime variables, the program calls `mmap` with the requested address and access permissions (read/write initially).

```
void* mem_address = find_free_space_near(&Rf_ScalarInteger,
    total_size);

uint8_t* memory = mmap(mem_address, total_size, PROT_READ |
    PROT_WRITE, MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);
if (memory == MAP_FAILED)
  exit(1);
```

Code listing 2.39 Allocating near memory.

These segments are initialized with appropriate data where needed.

2.5.3.2 Shared Memory

The compiler may choose to use either PC-relative or absolute 32-bit addressing for external symbols. While the former is satisfied by placing memory close to the runtime, the latter cannot be guaranteed because the chosen address may not fall within the lower 2 GB of memory, rendering 32-bit absolute addresses invalid.

To resolve this, a secondary memory region is allocated explicitly within the lower 2 GB using the `MAP_32BIT` flag with `mmap`. This memory region contains a read-only copy of the necessary runtime data. The patching phase then selects between the near and lower-memory version depending on the type of address required (relative/absolute).

Because this memory segment does not need to be close to the JITed code and its contents are identical across all compiled closures, it is shared among all of them. A reference counting mechanism ensures that the memory is deallocated only after all closures using it are destroyed.

```c
uint8_t *mem_shared = NULL;
size_t *mem_shared_ref_count = NULL;
static void prepare_rodata()
{
  mem_shared = mmap(NULL, sizeof(rodata), PROT_READ | PROT_WRITE,
      MAP_PRIVATE | MAP_ANONYMOUS | MAP_32BIT, -1, 0);
  if (mem_shared == MAP_FAILED)
    exit(1);

  memcpy(mem_shared, rodata, sizeof(rodata));

  if (mprotect(mem_shared, sizeof(rodata), PROT_READ) != 0)
  {
    perror("mprotect failed");
    exit(1);
  }

  mem_shared_ref_count = malloc(sizeof(*mem_shared_ref_count));
  *mem_shared_ref_count = 1;
}
```

Code listing 2.40 Allocating shared memory.

2.5.4 Stencil Selection

In this phase, the core of the Copy-and-Patch approach begins, as the program iterates over the bytecode instructions. For each instruction, it selects the corresponding stencil to use.

For most instructions, only generic stencils are available, but a few have specialized versions (see Section 2.2.6).

The stencils are designed such that the Copy-and-Patch compiler can select the correct variant based on compile-time information.

A primary example is the LDCONST instruction, which is specialized based on the type of the constant it loads. Since the program now has access to the constant's SEXP type, it can convert the generic load to a specialized stencil that directly loads the primitive value.

This behavior is illustrated in Code listing 2.41.

A similar specialization exists for the MATH1 instruction. In this case, the stencil is selected based on the immediate argument, which specifies the mathematical function to execute.

```
SEXP constant = r_constpool[imms[0]];
if (constant->sxpinfo.scalar &&
    ATTRIB(constant) == R_NilValue)
{
  switch(TYPEOF(constant))
  {
    case REALSXP:
      return &_RCP_LDCONST_DBL_OP;
    case INTSXP:
      return &_RCP_LDCONST_INT_OP;
    case LGLSXP:
      return &_RCP_LDCONST_LGL_OP;
    default:
      break;
  }
}
return &_RCP_LDCONST_SEXP_OP;
```

Code listing 2.41 LDCONST specilization.

For instructions without specialized versions, a lookup table generated by the stencil extraction tool is used. This allows the program to determine the appropriate stencil in constant time based on the opcode of the instruction.

If the instruction is `MAKECLOSURE_OP`, the program recursively compiles the referenced closure from the constant pool and replaces it with its compiled, copy-patched version. This allows complete compilation of the original closure, including any nested functions or closures it contains.

2.5.5 Copying and Patching

After selecting the appropriate stencil, the program copies its body (raw `x86_64` instructions) into the allocated memory using `memcpy`, placing it immediately after the previous stencil. This creates a continuous block of executable code that can later be invoked as a single function.

Once copied, the stencil is patched in-place. The program iterates over the stencil's holes and fills them with the appropriate values or addresses.

The value to patch is determined by the patch type specified in the stencil metadata (see Table 2.2).

To increase the likelihood of success with the small (and faster) memory model, internal runtime symbols are allocated close to the executable code, as described in Section 2.5.3.

Furthermore, the shared read-only data section (`.rodata`) is duplicated into memory within the lower 2 GB. If a stencil requires an absolute address to `.rodata`, the patching algorithm uses this lower memory region to ensure that the 4-byte absolute address fits within the stencil's hole. This mechanism is

```
if(bytecode[i] == MAKECLOSURE_OP)
{
  SEXP fb = constpool[bytecode[i+1]];
  SEXP body = VECTOR_ELT(fb, 1);

  if(TYPEOF(body) == BCODESXP)
  {
    SEXP res = copy_patch_bc(body, stats);
    SET_VECTOR_ELT(fb, 1, res);
  }
  else if(IS_RCP_PTR(body))
  {
    DEBUG_PRINT("Using precompiled closure\n");
  }
  else
  {
    error("Invalid closure type: %d\n", TYPEOF(body));
  }
}
```

Code listing 2.42 Recursive compilation.

shown in Code listing2.43.

```
if(hole->is_pc_relative)
  ptr = (ptrdiff_t)ctx->ro_near;
else
  ptr = (ptrdiff_t)ctx->ro_low;
```

Code listing 2.43 Patching of .rodata.

Other patch types are handled in a straightforward manner: the `ptr` value is simply set to the value being patched.

After the value to patch is known, the program adds the addend to it, and if it is PC-relative, subtracts the current position of the memory to get offset instead of relative address.

After computing the final value to patch, the program verifies that it fits into the allocated hole size. If the stencils were compiled using the medium memory model, there is a possibility that, on some systems, a patched address will not fit. In such cases, an error is raised and compilation is aborted.

Finally, the computed value is written into the stencil hole, and the program proceeds to the next hole.

The overall patching process (excluding patch type selection) is illustrated in Code listing 2.44.

```c
static void patch(uint8_t* dst, const Hole* hole, int* imms, int
    nextop, const PatchContext* ctx)
{
  ptrdiff_t ptr;

  switch(hole->kind)
  {
    ...
  }

  ptr += hole->addend;
  if(hole->is_pc_relative)
    ptr -= (ptrdiff_t)&dst[hole->offset];

  if(!fits_in(ptr, hole->size))
    error("Offset does not fit into patch hole");

  memcpy(&dst[hole->offset], &ptr, hole->size);
}
```

Code listing 2.44 Patching function.

2.5.6 Finalization

The program resets runtime-initialized variables, such as `BCells` and the environment pointer, to default values.

If all preceding operations complete successfully, the program marks the allocated memory region as executable and returns a structure containing all the information required to execute the compiled closure. This structure is defined in R header files and is shown in Code listing 2.46.

An `EXTPTRSXP` object is then created to hold a pointer to this structure. It is tagged with an internal C-string, allowing it to be identified at runtime. A finalizer is registered for this object, ensuring that all allocated memory is released when the resulting `SEXP` is garbage collected. This is shown in Code listing 2.45, a continuation of Code listing 2.38.

The body of the original closure, compiled using `compiler::cmpfun`, is then replaced with this `EXTPTRSXP`, and the resulting closure is returned.

2.6 Runtime

Support for running copy-patched closures had to be integrated directly into R. First, a structure is defined to represent the compiled closure, enabling R internals to interact with the native code. This structure, shown in Code listing 2.46, includes a pointer to the executable code (`eval`), runtime-modifiable variables (`rho` and `bcells`), and memory metadata required for cleanup.

```
  ...
  rcp_exec_ptrs* res_ptr = malloc(sizeof(rcp_exec_ptrs));
  *res_ptr = res;

  SEXP tag = PROTECT(install(RCP_PTRTAG));
  SEXP ptr = R_MakeExternalPtr(res_ptr, tag, bcode_consts);
  UNPROTECT(1);// tag
  PROTECT(ptr);
  R_RegisterCFinalizerEx(ptr, &R_RcpFree, TRUE);
  UNPROTECT(1); // ptr
```

Code listing 2.45 Creation of custom EXTPTRSXP.

```
typedef struct rcp_exec_ptrs
{
  // Executable code
  SEXP (*eval)();

  // Local internal variables to set before execution (do not free!)
  SEXP * rho;
  SEXP * bcells;
  size_t bcells_size;

  // Memory management
  void* memory_private;
  size_t memory_private_size;
  void* memory_shared;
  size_t memory_shared_size;
  size_t* memory_shared_refcount;
} rcp_exec_ptrs;
```

Code listing 2.46 Structure for the result of copy-patch.

A tag is introduced to label external pointers produced by the compiler, and a macro is added to check whether a given `SEXP` is a custom pointer belonging to this project.

```
#define RCP_PTRTAG "rcp_exec_ptrs"

#define IS_RCP_PTR(fun)                                              \
  (TYPEOF(fun) == EXTPTRSXP &&                                       \
   strcmp(CHAR(PRINTNAME(EXTPTR_TAG(fun))), RCP_PTRTAG) == 0)
```

Code listing 2.47 Macros to check for RCP pointer.

2.6.1 Execution

In the implementation of R's `eval` function (responsible for evaluating R expressions) a check is added to detect whether the expression being evaluated points to a copy-patched closure.

```
switch (TYPEOF(e)) {
  case EXTPTRSXP:
  if(IS_RCP_PTR(e))
    tmp = rcpEval(e, rho);
  else
    tmp = e;
  break;
  case BCODESXP:
  tmp = bcEval(e, rho);
  ...
```

■ **Code listing 2.48** Support for copy-patch in R evalution.

If the check passes, a custom evaluation function is called (see Code listing 2.49).

At the start of this function, the external pointer is cast to the structure defined in Code listing 2.46. Because the memory for internal runtime variables (e.g., environment, `BCells`) is allocated alongside the code (see Section 2.5.3), they need to be initialized before execution.

This initialization happens in Lines 16–19, where the `BCell` cache is reset and the current environment is assigned.

However, because the copy-patched closure can be executed recursively, the program cannot just overwrite the internals, as this would break the function that initiated the recursion - after the call returns back to the caller, the environment and cache would be invalid.

Because of this, the program first needs to save the current values, which can be seen on Lines 6-13. BCell cache is backed up to stack, and current environment into a local variable. If the function is never executed recursively, this is unnecessary, but needs to be done to support recursion.

Next, global variables that could be left in a corrupted state are backed up in Lines 22–23. This precaution mirrors the original bytecode evaluation routine in R, which also preserves these values to prevent session-wide failures caused by incomplete closures.

With all preparations complete, the function calls the compiled native function, and control transfers into the executable memory region, as described in Section 2.5.5. Execution continues until the compiled closure completes and returns a result of type `SEXP`.

After execution, the function restores global variables (Line 29) and local runtime values (Lines 30–32), then returns the computed result.

```c
SEXP rcpEval(SEXP body, SEXP rho)
{
  rcp_exec_ptrs* ptrs = (rcp_exec_ptrs*)EXTPTR_PTR(body);

  /* save current bcells and rho - needed to support recursion */
  for (size_t i = 0; i < ptrs->bcells_size; ++i)
  {
    R_BCNodeStackTop->tag = 0;
    R_BCNodeStackTop->flags = 0;
    R_BCNodeStackTop->u.sxpval = ptrs->bcells[i];
    R_BCNodeStackTop++;
  }
  const SEXP rho_old = *(ptrs->rho);

  /* set up the new bcells and rho */
  for (size_t i = 0; i < ptrs->bcells_size; ++i)
    ptrs->bcells[i] = R_NilValue;

  *(ptrs->rho) = rho;

  /* save current globals */
  struct bcEval_globals globals;
  save_bcEval_globals(&globals);

  /* run the actual copy-patched code */
  SEXP res = ptrs->eval();

  /* restore everything to previous state */
  restore_bcEval_globals(&globals);
  *(ptrs->rho) = rho_old;
  for (size_t i = 0; i < ptrs->bcells_size; ++i)
    ptrs->bcells[i] = (--R_BCNodeStackTop)->u.sxpval;

  return res;
}
```

Code listing 2.49 Calling of copy-patched code during runtime.

2.6.2 Memory Management

When the `SEXP` closure containing the custom pointer is destroyed (for how this is triggered, see Code listing 2.45), a finalizer function is called.

This function, shown in Code listing 2.50, is responsible for releasing all memory allocated in Section 2.5.3, preventing memory leaks in the R environment.

It frees the private memory (internal variables, cache and code), and the pointer itself. If shared memory used across all compiled closures is no longer

```
void R_RcpFree(SEXP ptr)
{
  if(!IS_RCP_PTR(ptr))
  error("Attemted to free a non-rcp pointer");

  rcp_exec_ptrs* ptrs = (rcp_exec_ptrs*)EXTPTR_PTR(ptr);
  if(ptrs)
  {
    /* unmap private memory */
    munmap(ptrs->memory_private, ptrs->memory_private_size);

    /* unmap shared memory, if this is it's only use */
    if(ptrs->memory_shared_refcount != NULL &&
        --(*ptrs->memory_shared_refcount) == 0)
    {
      munmap(ptrs->memory_shared, ptrs->memory_shared_size);
      free(ptrs->memory_shared_refcount);
    }

    /* free the structure itself */
    free(ptrs);
    EXTPTR_PTR(ptr) = NULL;
  }
}
```

Code listing 2.50 Freeing of internal structure.

referenced (determined via reference counting) it is freed as well.

Chapter 3
Evaluation

This chapter presents an evaluation of the implemented system. First, a summary of the environment and used benchmarks is provided. Then, the performance of the project is assessed and compared to GNU R. Special attention is given to compilation speed and runtime performance. Additionally, we evaluate the effectiveness of selected optimizations introduced in Section 2.2.

3.1 Setup

All tests were conducted on a machine with an Intel Core i7-7700K CPU, 32 GB of RAM, and a fresh installation of Ubuntu 24.04. The R environment was compiled from source with default arguments, modified only to support the runtime system introduced in Section 2.6.

The benchmarks used for performance evaluation and the script for running them were imported from the Ř project. All 104 instructions implemented in Ř were successfully adapted for the Copy-and-Patch approach and verified for correctness. All 57 provided benchmarks were compiled and executed within the R environment without any issues, including real-world, non-trivial R code.

Performance was evaluated based on three key metrics:

- Compilation speed
- Execution speed
- Binary size of compiled code

Instruction counts per function were also recorded. All optimizations described in Section 2.2 were applied, except for stack overflow checks, which were evaluated separately. Unless stated otherwise, results are based on the absolute (large) memory model.

In this configuration, the total size of all compiled stencils is 75,950 bytes. The average stencil size is 611 bytes, with the average size of stencil for used instructions across benchmarks being 919 bytes. Individual stencil sizes for all supported instructions are listed in Table 3.1.

Each benchmark was executed 30 times, with 5 warm-up runs beforehand. The mean of these 30 measurements was used in execution performance comparisons.

3.2 Benchmark Datasets

The evaluation used the benchmark suites provided by the Ř project, covering a wide range of usage patterns:

- **Are We Fast Yet**: A cross-language suite of micro and macro benchmarks [20]
- **Real Thing**: Real-world R programs
- **Shootout**: Benchmarks from the popular cross-language competition [21]
- **Simple**: Custom microbenchmarks targeting individual R features

3.3 Compilation Speed

As the speed of compilation is one of the most important features of the Copy-and-Patch approach, it is important to evaluate if this is also the case for the introduced implementation. Measured compilation speed for all benchmark datasets can be seen in Figure 3.1.

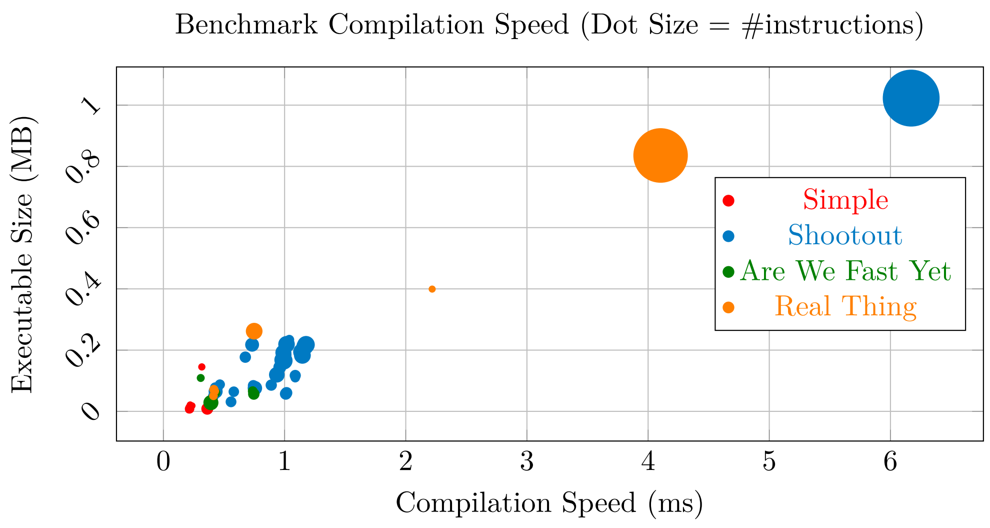


Figure 3.1 Compilation speed for all benchmarks.

Stencil	Size
RETURN	135
GOTO	12
BRIFNOT	275
POP	26
DUP	69
STARTFOR	1205
STEPFOR	1078
ENDFOR	51
INVISIBLE	26
LDCONST	95
LDNULL	78
LDTRUE	87
LDFALSE	85
GETVAR	1350
SETVAR	1000
GETFUN	174
GETBUILTIN	156
GETINTLBUILTIN	168
CHECKFUN	147
MAKEPROM	231
DOMISSING	161
SETTAG	114
DODOTS	431
PUSHARG	236
PUSHCONSTARG	120
PUSHNULLARG	113
PUSHTRUEARG	123
PUSHFALSEARG	123
CALL	729
CALLBUILTIN	729
CALLSPECIAL	422
MAKECLOSURE	180
UMINUS	482
UPLUS	458
ADD	847
SUB	837
MUL	848
DIV	831
EXPT	891
SQRT	536
EXP	536
EQ	887
NE	887
LT	887
LE	887
GE	887
GT	887
AND	498
OR	498
NOT	399
STARTASSIGN	731
ENDASSIGN	579
STARTSUBSET	744
DFLTSUBSET	525
STARTSUBASSIGN	673
DFLTSUBASSIGN	745
STARTSUBSET2	744
DFLTSUBSET2	525
STARTSUBASSIGN2	673
DFLTSUBASSIGN2	745
DOLLAR	493
ISNULL	54
ISLOGICAL	62
ISINTEGER	127
ISDOUBLE	62
ISCOMPLEX	62
ISCHARACTER	62
ISSYMBOL	60
ISOBJECT	60
ISNUMERIC	192
VECSUBSET	1207
MATSUBSET	1666
VECSUBASSIGN	2376
MATSUBASSIGN	1532
AND1ST	328
AND2ND	375
OR1ST	345
OR2ND	361
GETVAR_MISSOK	1317
SETVAR2	174
STARTASSIGN2	473
ENDASSIGN2	204
SETTER_CALL	1703
GETTER_CALL	990
SWAP	103
DUP2ND	69
STARTSUBSET_N	346
STARTSUBASSIGN_N	427
VECSUBSET2	1107
MATSUBSET2	1584
VECSUBASSIGN2	3117
MATSUBASSIGN2	1753
STARTSUBSET2_N	346
STARTSUBASSIGN2_N	427
SUBSET_N	1698
SUBSET2_N	1618
SUBASSIGN_N	1534
SUBASSIGN2_N	1751
LOG	429
LOGBASE	876
MATH1	527
COLON	796
SEQALONG	566
SEQLEN	488

Table 3.1 Stencil size in bytes for each R instruction.

As can be seen from the results, the introduced implementation is able to compile all individual benchmarks in under 7 ms, with the median of just 0.68 ms. Compared to the Ř implementation, this is a substantial improvement. For illustration, the introduced implementation approach is capable of compiling all 57 benchmarks combined in under 50 ms total, which is several orders of magnitude faster than compilation of just a single benchmark file in Ř [1].

Compilation time correlates closely with binary size, which is expected as memory bandwidth is the main bottleneck. Interestingly, number of R bytecode instructions are not a reliable predictor of executable size or compilation time.

For example, while the two benchmarks that produce the biggest code sizes (which are *pidigits* and *flexclust*) have large number of instructions, the benchmark *flexclust_no_s4* from the *RealThing* dataset does not - despite producing the third biggest executable size (0.4 MB). On the contrary, it does so with only one of the smallest number of instructions in the entire benchmark suite - 27. Compared to the benchmark with the second biggest executable size, which consists of 1,809 instructions, it can be seen that this metric by itself is not directly responsible for decreasing compilation speed.

3.4 Execution Performance

The execution performance speedup for all benchmarks is visualized in Figure 3.2.

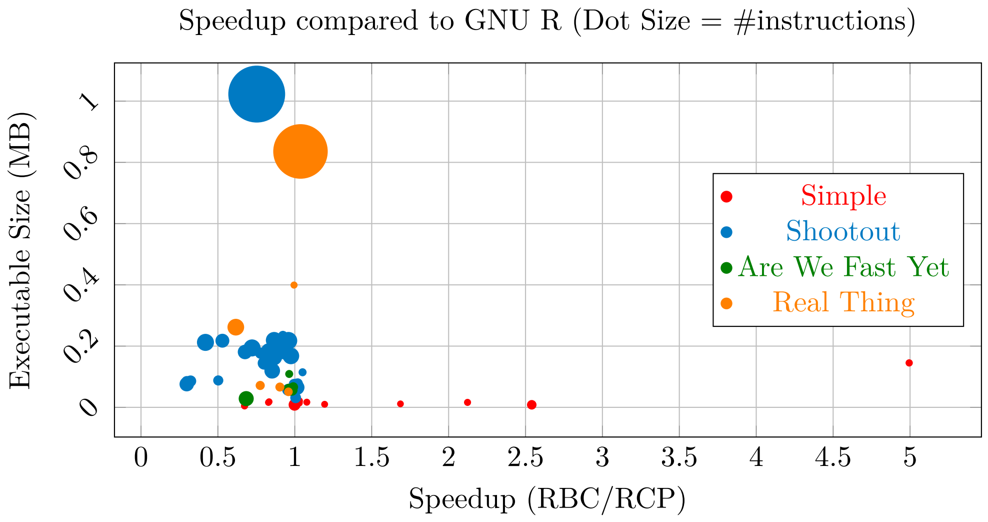


Figure 3.2 Execution performance speedup.

Across all benchmarks, the performance speedup is averaging to 1.00×. But as the benchmark results vary depending on the dataset, two are described separately and in more detail: *Simple* and *Shootout*.

The behavior difference between these two datasets is considerable. While the average speedup for the *Shootout* dataset is 0.82×, the *Simple* dataset performs much better, with an average speedup of 1.54×, and a maximum of 5.00×.

Detailed performance comparisons to GNU R for *Simple* and *Shootout* datasets are presented in Figures 3.3 and 3.4. Individual benchmarks are sorted by relative speedup.

It is interesting to observe that both the worst speedups (*binarytrees*, *binarytrees_naive* and *binarytrees_2* from *shootout*) and the best (*bytecodes* from *simple* finish their execution in less than third of a second in both cases, with *bytecodes* finishing in as low as 8 ms. While the relative difference of the mentioned benchmarks is the highest, the absolute difference in runtime is low enough that other execution environment overheads might be more noticeable, and it is up to further analysis if these results would scale in the same way for larger use cases.

It is very important to note however that the differences in execution speed, both positive and negative, are mostly due to the Ř implementation of instruction functions. The authors of Ř introduce many changes from the original implementation of GNU R that are beneficial in some use cases, but at the same time can also be responsible for slow downs in others. These changes are what causes some benchmarks to exhibit pathological behavior in both cases of the spectrum.

Additionally, it should be reiterated that this implementation uses as stencils the already established Ř implementations of instructions that are originally designed for a different purpose. It is beyond the scope of this work to optimize Ř implementation of instructions for the use with Copy-and-Patch (this would however be useful to do in future work). The main information gained from this section is that the implementation follows the performance trend of Ř with much improved speed of compilation, which was the goal of this work.

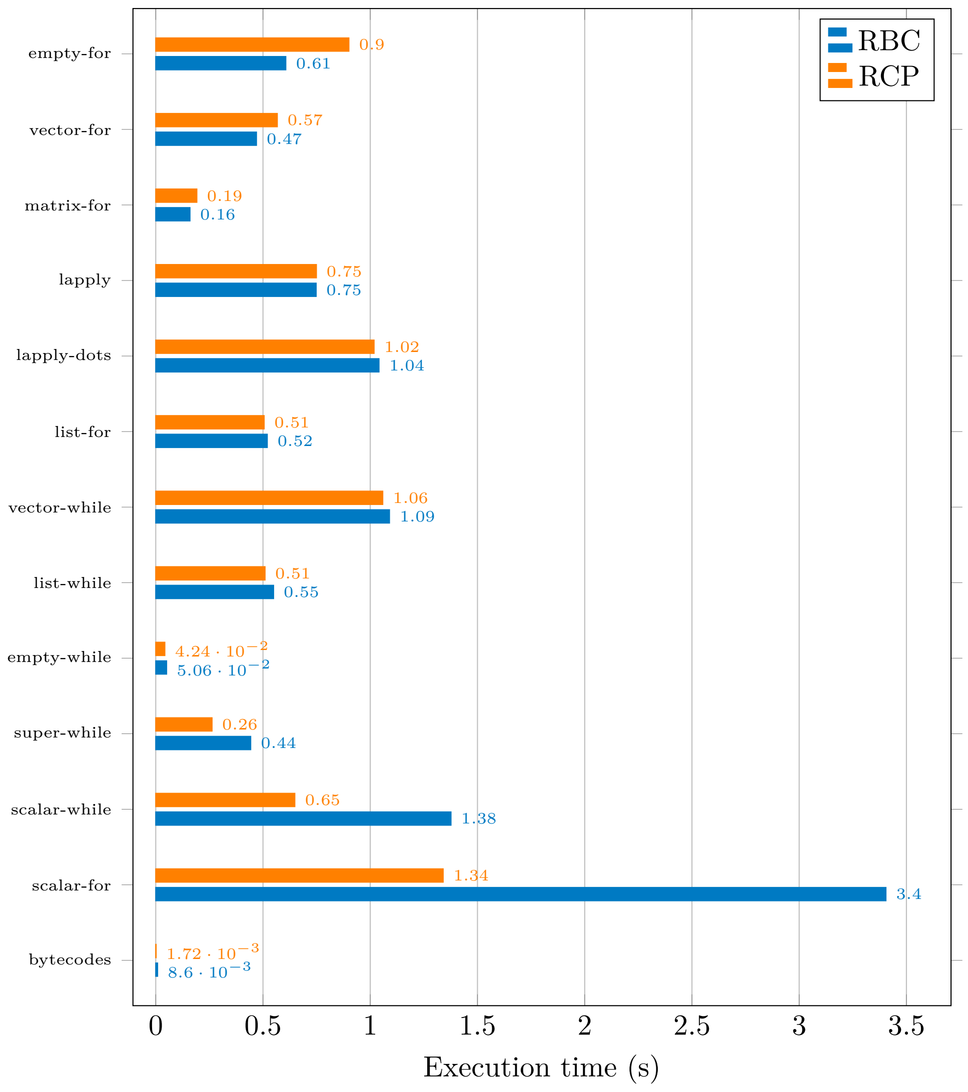


Figure 3.3 Performance comparison of benchmarks of the simple dataset.

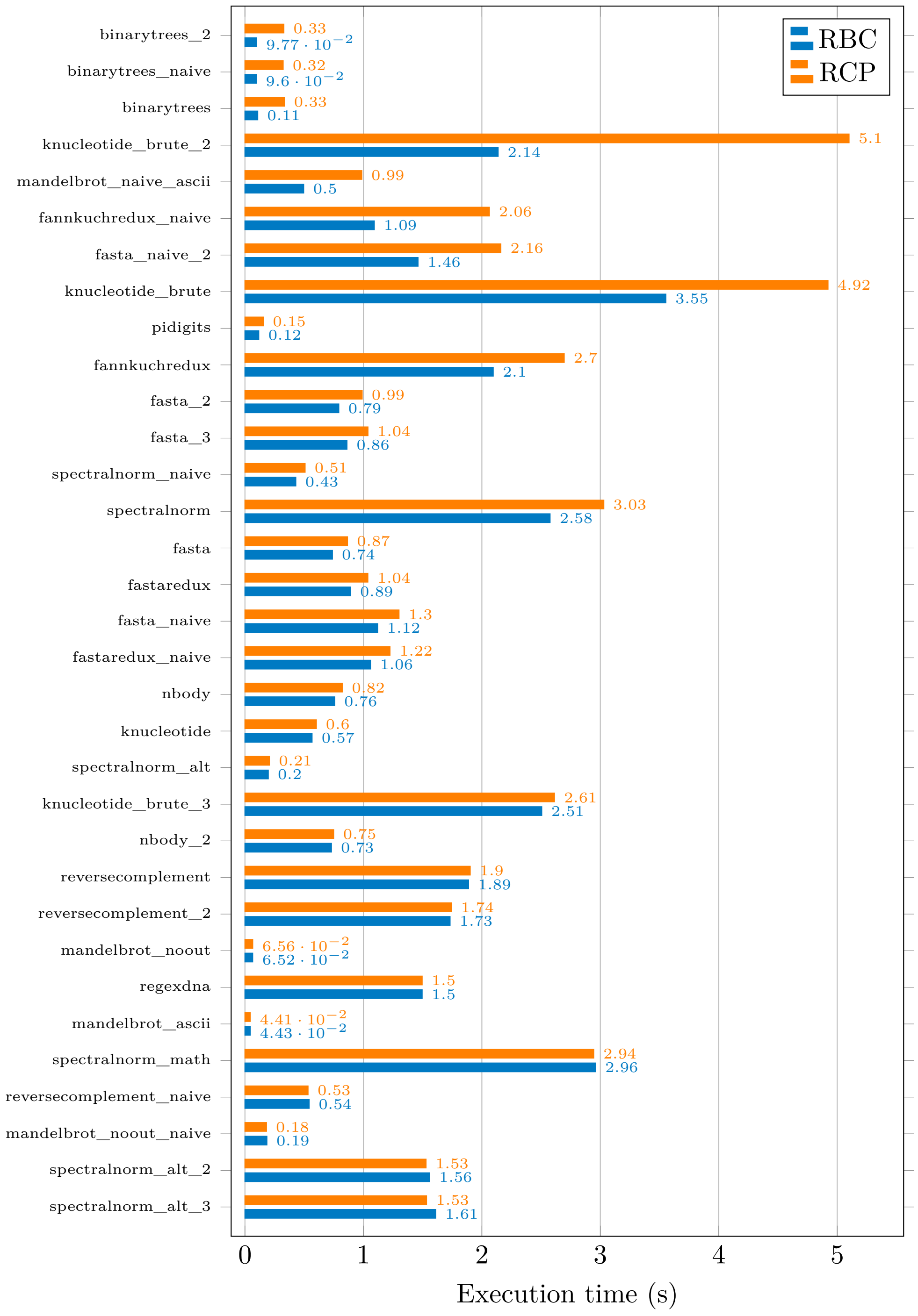


Figure 3.4 Performance comparison of benchmarks of the shootout dataset.

3.5 Optimization Evaluation

While the comparison of execution performance between the reference implementation and this project is mostly due to choices of Ř authors, there are also performance implications for how the Copy-and-Patch approach is implemented.

As described in Section 2.2, there were deliberate additional optimizations implemented. The two most likely to produce measurable differences were evaluated, and compared to a version without their usage.

3.5.1 Omitting Stack Overflow Check

This evaluation directly references the Section 2.2.3, where this optimization is described in detail. This optimization was evaluated on all benchmarks, and the speedup visualized in Figure 3.5.

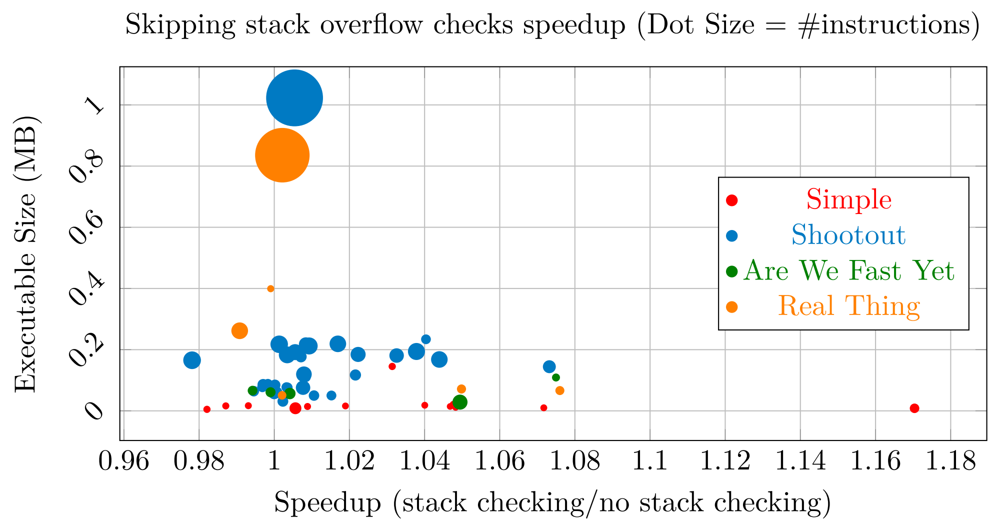


Figure 3.5 Performance comparison for turning off stack overflow checks.

As can be clearly seen from the graph, the optimization noticeably improved the execution performance of most benchmarks. The speed of execution was improved by 2% on average, specifically benefiting more the *simple* dataset where the difference was on average over 3%. The dataset *shootout* also gained measurable performance increase, although more modest (1%).

The largest improvement was observed in the *scalar-for* benchmark from the *Simple* dataset, likely due to tight loops with heavy stack manipulation.

Despite measurable gains, this optimization may not be worth the trade-off in safety. For most use cases, especially those involving JIT, preventing crashes on malformed programs is more important than a small performance boost.

3.5.2 Memory Models

Usage of different memory models (PC-relative and absolute) produces an important difference for an executable. As the absolute model must address every function and variable using full 64-bits, this increases the size of the executable considerably to the point it can slow down the execution.

However, the faster and more compact PC-relative model has its limitations, especially for a JIT. By default, it requires all accessed symbols to be near the executable, which might be difficult to achieve on some machines due to memory randomization techniques and other OS related specifics.

Despite considerable effort in this work to enable the usage of PC-relative memory model globally, in the environment where this work was tested, not all symbols required for all instructions were available in this way[1], so all performance evaluation so far were done using the absolute model.

This is unfortunate, as the relative memory model can offer significant performance improvements, as can be seen in Figure 3.6, where benchmarks which could not be compiled (7 out of the total of 53) were omitted.

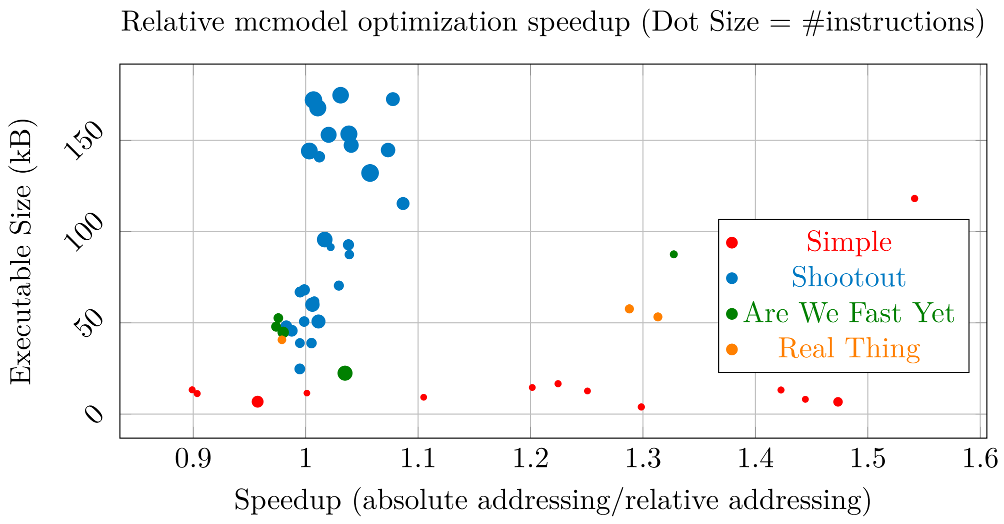


Figure 3.6 Performance comparison for different mcmodel options.

Usage of the PC-relative memory model improved the performance nearly for all benchmarks, with varying levels of success. On average, all benchmarks that were able to be compiled using this memory model were faster by more than 8%, while the *simple* dataset specifically received a performance boost of 21%.

As described, this technique also made the executable smaller, and that on average by 26% across all benchmarks. This in turn resulted in an additional

[1] Specifically the *cmath* library was loaded far from others, so any Ř instruction function using it will not work.

compilation speed increase by more than 3%, further confirming the theory that the execution engine is memory bandwidth-bound.

If the non-functioning benchmarks are filtered, a new comparison to GNU R can be made, visible in Figure 3.7.

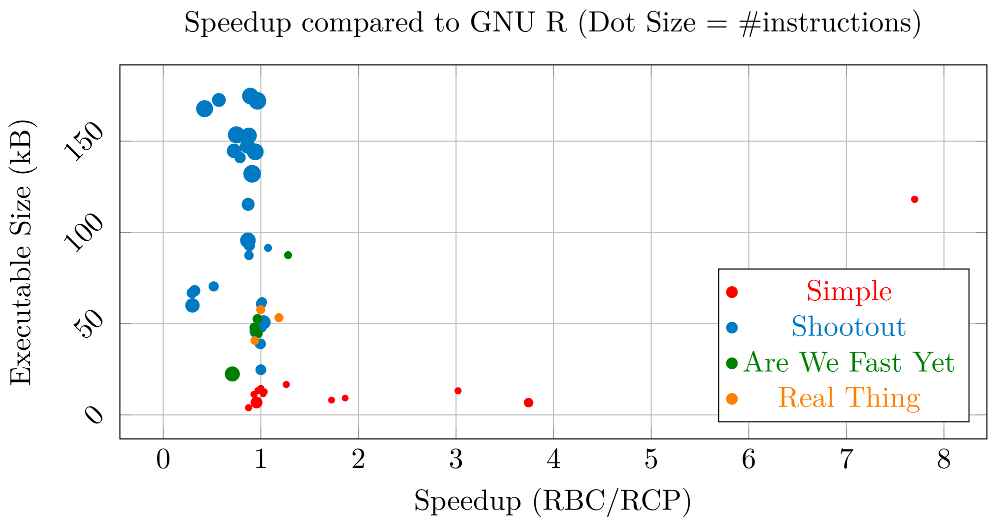


Figure 3.7 Execution performance speedup using PC-relative model.

As expected, this technique improved the performance of this implementation, which now shows close to 1.16× speedup over GNU R. It is important to note however, that the filtered out benchmarks have shown results in benefit of GNU R, so the new results might benefit from not including them more than from the optimization.

However, the *simple* dataset (where no benchmarks were left out), now improves on its already good speedup by gaining the result of 2.01× over the reference implementation.

With additional work, it might be possible to adapt all instructions to the PC-relative memory model, which shows promising results for the instructions which are already supported.

Conclusion

In this work, a functional prototype of the Copy-and-Patch approach for the R programming language has been successfully implemented. The Ř project was fully adapted to this approach, including all 104 R bytecode instructions it supports. Furthermore, what was originally meant to be a minimal proof-of-concept, has exceeded its initial scope. The prototype now runs complex, real-world R code and passes all provided benchmarks, including the widely recognized Shootout suite.

Several additional performance optimizations were developed and integrated. Notably, one optimization in particular (relative memory addressing model) showed a 21% performance boost in one of the benchmark suites.

The project was evaluated across multiple metrics, including compilation speed, execution time, and binary size. Compilation times were particularly impressive: all benchmarks compiled in under 6 ms, with a median of just 0.7 ms. Execution speed followed the performance characteristics of Ř, as expected, since this implementation reuses Ř's instruction definitions. While performance was generally slower than Ř, as is typical for a baseline JIT, the compilation speed was several orders of magnitude faster. Compared to GNU R, execution performance was roughly on par.

In further work, the Ř implementation of R instructions would benefit from detailed analysis and possible design changes for the use with Copy-and-Patch. Furthermore, some of the optimizations introduced could be extended or made compatible with more programs, namely implementation of the relative memory model, which showed promising results for programs on which it was able to be used at the moment. Additional improvements, like adjusting stencil inlining, may reduce binary size and improve compile times further.

Although the result is a fully functional and self-contained prototype R package, it is not yet ready for production. It has not undergone rigorous testing for bugs or edge cases in general usage. Lastly, a deeper analysis of benchmark behavior could provide more insight into specific performance characteristics and guide future improvements.

Bibliography

1. FLÜCKIGER, Olivier; CHARI, Guido; YEE, Ming-Ho; JEČMEN, Jan; HAIN, Jakob; VITEK, Jan. Contextual dispatch for function specialization. *Proc. ACM Program. Lang.* 2020, vol. 4, no. OOPSLA. Available from DOI: `10.1145/3428288`.
2. XU, Haoran; KJOLSTAD, Fredrik. Copy-and-patch compilation: a fast compilation algorithm for high-level languages and bytecode. *Proc. ACM Program. Lang.* 2021, vol. 5, no. OOPSLA. Available from DOI: `10.1145/3485513`.
3. R CORE TEAM. *R: The R Project for Statistical Computing* [`https://www.r-project.org/`]. 1993. Accessed: 2025-05-09.
4. WICKHAM, Hadley. *ggplot2: Elegant Graphics for Data Analysis*. Springer-Verlag New York, 2016. ISBN 978-3-319-24277-4. Available also from: `https://ggplot2.tidyverse.org`. Accessed: 2025-05-09.
5. PALL, Mike. *LuaJIT: A Just-In-Time Compiler for Lua* [`https://luajit.org/`]. 2005. Accessed: 2025-05-09.
6. MOZILLA. *SpiderMonkey JavaScript Engine*. 1996. Available also from: `https://spidermonkey.dev/`. Accessed: 2025-05-09.
7. GOOGLE. *V8 JavaScript Engine* [`https://v8.dev/`]. 2008. Accessed: 2025-05-09.
8. THE PYPY TEAM. *PyPy: A Fast, Compliant Alternative Implementation of Python* [`https://www.pypy.org/`]. 2007. Accessed: 2025-05-09.
9. GRAALVM TEAM. *GraalVM: Run Programs Faster Anywhere* [`https://www.graalvm.org/`]. 2019. Accessed: 2025-05-09.
10. GRAALVM TEAM. *FastR: An Implementation of R on GraalVM* [`https://www.graalvm.org/reference-manual/r/`]. 2019. Accessed: 2025-05-09.
11. NEAL, Radford M. *pqR: A Pretty Quick Version of R* [`http://www.pqr-project.org/`]. 2014. Accessed: 2025-05-09.

12. LANG, Duncan Temple. *RLLVMCompile: Compile R Code Using LLVM* [`https://github.com/duncantl/RLLVMCompile`]. 2013. Accessed: 2025-05-09.

13. RUNNALLS, Andrew. Aspects of CXXR Internals. *Computational Statistics*. 2010, vol. 25, no. 4, pp. 711–723. Available from DOI: `10.1007/s00180-010-0218-0`.

14. SILLYCROSS. *Building the fastest Lua interpreter.. automatically!* [`https://sillycross.github.io/2022/11/22/2022-11-22/`]. 2022. Accessed: 2025-05-09.

15. BUCHER, Brandt. *A copy-and-patch JIT compiler* [`https://github.com/python/cpython/pull/113465`]. 2023.

16. SIEK, Konrad. *Everything You Always Wanted to Know About SEXPs But Were Afraid to Ask*. Version 2. 2018. Available also from: `https://gitlab.com/kondziu/everything-you-always-wanted-to-know-about-SEXPs/-/raw/35b0354ed57a8f14a6f3d175742a0ae2cdd7a6c0/version2.html`.

17. *UNIX System V: User's Reference Manual*. AT&T, 1983.

18. GNU PROJECT. *Using the GNU Compiler Collection*. Free Software Foundation, 2025. Available also from: `https://gcc.gnu.org/onlinedocs/`. Accessed: 2025-05-09.

19. FREE SOFTWARE FOUNDATION. *BFD: The Binary File Descriptor library* [Online manual]. GNU Project, 1991. `https://ftp.gnu.org/old-gnu/Manuals/bfd-2.9.1/html_chapter/bfd_1.html`.

20. MARR, Stefan; DALOZE, Benoit; MÖSSENBÖCK, Hanspeter. Cross-language compiler benchmarking: are we fast yet? *SIGPLAN Not.* 2016, vol. 52, no. 2, pp. 120–131. ISSN 0362-1340. Available from DOI: `10.1145/3093334.2989232`.

21. *Measured : Which programming language is fastest? (Benchmarks Game)* [online]. [N.d.]. [visited on 2025-05-04]. Available from: `https://benchmarksgame-team.pages.debian.net/benchmarksgame/index.html`.

Attachment Contents

```
/
├── exec ...................... directory with compiled package and stencils
│   ├── stencils .......................... directory with compiled stencils
│   │   ├── headers ......................... directory of stencils as headers
│   │   ├── stencils.o ........................ compiled stencil object file
│   │   └── stencils.txt ..................... compiled stencil disassembly
│   └── rcp_1.0.0.0000.tar.gz ..................... compiled R package
├── measurements ............................... evaluation measurements
├── src
│   ├── rcp-code ............................................ source code
│   └── rcp-text ........................... source of the thesis in LaTeX
├── text ........................................................ thesis
│   └── thesis.pdf ....................................... thesis in PDF
└── license.txt .................................. license of source files
```