RESEARCH-ARTICLE

Copy-and-Patch Just-in-Time Compiler for R

MATĚJ KOCOUREK, Charles University, Prague, Czech Republic

FILIP KŘIKAVA, Czech Technical University in Prague, Prague, Czech Republic

JAN VITEK, Northeastern University, Boston, MA, United States

Open Access Support provided by:

Northeastern University

Charles University

Czech Technical University in Prague

Published: 09 October 2025

Citation in BibTeX format

VMIL '25: 17th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages
October 12 - 18, 2025
Singapore, Singapore

Conference Sponsors:
SIGPLAN

Copy-and-Patch Just-in-Time Compiler for R

Matěj Kocourek
Charles University
Prague, Czech Republic
matej.kocourek@matfyz.cuni.cz

Filip Křikava
Czech Technical University
Prague, Czech Republic
filip.krikava@fit.cvut.cz

Jan Vitek
Northeastern University
Boston, USA
j.vitek@northeastern.edu

Abstract

Copy-and-patch is a technique for building baseline just-in-time compilers from existing interpreters. It has been successfully applied to languages such as Lua and Python. This paper reports on our experience using this technique to implement a compiler for the R programming language. We describe how this new compiler integrates with the GNU R virtual machine, present the key optimizations we implemented, and evaluate the feasibility of this approach for R. Copy-and-patch also allows extensions such as integration of the feedback recording required by multi-tier compilation. Our evaluation on 57 programs demonstrates very fast compilation times (980 bytecode instructions per millisecond), reasonable performance gains (1.15x–1.91x speedup over GNU R), and manageable implementation complexity.

CCS Concepts: • **Software and its engineering → Just-in-time compilers**; *Runtime environments*; General programming languages.

Keywords: Baseline JIT compilers, Copy-and-Patch, R programming language

ACM Reference Format:
Matěj Kocourek, Filip Křikava, and Jan Vitek. 2025. Copy-and-Patch Just-in-Time Compiler for R. In *Proceedings of the 17th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages (VMIL '25), October 12–18, 2025, Singapore, Singapore.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3759548.3763370

1 Introduction

Dynamic programming languages use just-in-time (JIT) compilation to recover runtime performance lost due to the lack of information available at compile time. JIT compilers are organized in tiers to amortize compilation costs; before any significant optimization can occur, the compiler needs to learn about the program. This is the job of the baseline compiler: to record runtime behavior and pass this feedback to upper tiers.

VMIL '25, Singapore, Singapore

ACM ISBN 979-8-4007-2164-9/25/10
https://doi.org/10.1145/3759548.3763370

Building a baseline JIT is a challenging task. The compiler needs to be fast enough not to interfere with program execution, but also produce code that is good enough to amortize the compilation overhead. In this paper, we explore the feasibility of copy-and-patch compilation [21] for the R programming language.

Copy-and-patch offers a pragmatic approach to compilation by directly reusing an existing interpreter implementation. It works by partially applying the implementation of interpreter operations—whether AST node operations or bytecode instructions—to generate a library of native code chunks. Each chunk corresponds to a specific operation and contains placeholders for runtime values such as constants, addresses, or jump targets. During compilation, the compiler selects appropriate chunks for each operation in the program, copies them into a contiguous memory region, and patches the code with concrete values. This achieves fast compile times since the compiler avoids complex analysis or optimization phases—it simply stitches pre-compiled code fragments together. The resulting code quality is harder to predict since the compiler does not perform any optimizations.

The technique has been used for Lua in the LuaJIT Remake [5] and Python in the CPython implementation [16]. The research question we address is whether this approach is practical for the R programming language. Our long-term goal is an optimizing JIT built outside of the GNU R virtual machine. The reason is that GNU R is developed and maintained by volunteers. Embedding a JIT in the virtual machine's codebase would significantly increase the maintenance burden. Our first step is to build a baseline JIT that can be maintained and updated independently. Specifically, we aim to build a baseline JIT with support for feedback recording that is both fast and produces code good enough to be used in practice.

Our copy-and-patch JIT is able to compile 980 bytecode instructions per millisecond. This enables feedback recording by simply recompiling warmed-up functions and injecting recording instructions at appropriate places. Regarding performance, on a suite of 57 benchmarks, the compiler achieves a 1.26x speedup over GNU R. While this result is greatly influenced by small benchmarks (speedup of 1.91x), it manages to improve the performance of complex benchmarks as well (speedup of 1.15x).

The prototype is implemented as an R package (called `rcp`), but it requires a very lightly modified R virtual machine that exposes a few internal built-ins. It consists of approximately

1,200 lines of C code for the copy-and-patch runtime, 600 lines for the stencil extractor, and 900 lines dedicated to adapting R instructions from headers into stencils. It is available as open source and can be found at: https://github.com/PRL-PRG/rcp.

The prototype was developed for x86_64 Linux but can be adapted to other architectures in the future. Since the first half of the pipeline relies on a compiler with built-in cross-platform support, only the patching process and architecture-specific optimizations would require modification.

2 Background

In this section, we review R and the GNU R implementation including the bytecode interpreter and the optimizations it performs. We then give a brief overview of the copy-and-patch technique.

2.1 The R Programming Language

The R programming language is a popular language for statistical computing and data science. It is dynamically typed with vector-based types and operations, copy-on-write semantics for shared data, and call-by-need evaluation. It features multiple dispatch, first-class closures, and extensive reflection allowing full read-write access to the call stack [11]. This combination of features has made the language a challenging target for optimizing compilers [3].

R is an interpreted language; its implementation, GNU R, contains both an AST interpreter and a bytecode interpreter. By default, user functions are initially AST interpreted, then compiled into bytecode on their second invocation. Package functions are bytecode compiled upon installation. The R virtual machine, including the bytecode interpreter, is implemented in C, while the bytecode compiler is written in R as part of the standard library [19]. The rest of the library is written in a mixture of R and C.

The bytecode is a stack-based language with 129 operations. The interpreter performs several optimizations:

- scalar value unboxing and scalar arithmetic and relational operation optimization,
- optimization of certain vectorized operations on vectors of doubles, integers and booleans,
- optimization of assignments and subsets of unshared vectors,
- caching the results of variable lookups by storing pointers into the cons cells of the linked list representing a call frame,
- inlining bytecode compiled closure calls, and
- inlining promise evaluation.

This results in fairly complex instructions. For example, the `GetVar` operation checks cached symbols, handles unboxed values, evaluates promises, performs environment lookups, and initializes binding caches as needed, spanning over 200 lines of C code. The bytecode interpreter comprises over 5,000 lines of code with heavy use of C macros. It is implemented as a single monolithic function using either a traditional switch statement or direct threaded code if supported by the C compiler [14].

2.2 Copy-and-Patch JIT

Copy-and-patch is a technique for building a baseline JIT compiler from an existing interpreter. It works for both AST and bytecode interpreters. In this paper, we focus on bytecode. The technique is based on the concept of a binary *stencil*. A stencil is a partial implementation of a bytecode instruction with *holes*— placeholders for runtime values. The technique consists of two components: a stencil library and the copy-and-patch algorithm. The stencil library contains implementations of the target language's bytecode instructions, often including multiple variants of the same instruction, each with different optimizations. The algorithm operates in two phases. First, the *copy* phase walks the bytecode and selects the most appropriate stencil for each instruction, stitching the stencils together to form a complete executable. Next, the *patch* phase fills the holes in the copied stencils with concrete values such as jump targets, constants, or addresses of external symbols.

This approach achieves very fast compilation times since the compiler merely copies and patches existing code. However, it cannot perform optimizations beyond peephole optimizations.

3 Implementation

Figure 1 shows the compilation pipeline of our copy-and-patch JIT compiler with both build time and runtime components. At build time, we extract the implementation of individual bytecode instructions from the monolithic C function into standalone functions, which are then turned into stencils that form the stencil library. At runtime, we use the existing R bytecode compiler to compile functions from AST to bytecode. This bytecode is then traversed and corresponding stencils are selected. Finally, the selected stencils are copied and patched into a native function.

3.1 Extracting Standalone Instructions

One advantage of the copy-and-patch approach is that it can reuse an existing bytecode interpreter. However, the interpreter's implementation prevents direct reuse. For stencils, we need bytecode instruction code as standalone functions, essentially converting each switch case into a C function with explicit dependencies.

Refactoring the bytecode interpreter is non-trivial because the code is tightly coupled and relies on many global variables and macros. Furthermore, some instructions have complex interactions with the interpreter state, making it difficult to isolate them or are bytecode dependent, *e.g.*, `Call` which inlines calls to bytecode compiled closures.

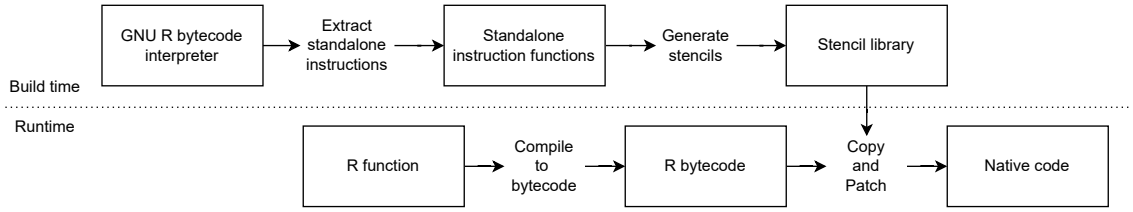

Figure 1. The compilation pipeline.

Instead of modifying the GNU R virtual machine, we decided to implement the compiler as an R package. We have extracted each bytecode instruction as a separate C function, exposing any private API when necessary. This often required some refactoring of the original code to eliminate dependencies on global state and adjusting the optimizations the interpreter does. For example, inlining calls to JIT compiled closures instead of byte compiled closures.

The result is a standalone header file which contains 2,500 of C code representing the bytecode instructions. We had to modify 18 files from the R virtual machine with 937 insertions and 838 deletions. However, the vast majority of these changes were mechanical, changing the visibility of C functions and structures which involved moving code between header files.

3.2 Stencils

Listing 1 shows the implementation of the `BRIFNOT` R bytecode instruction stencil on which we demonstrate how the stencils work. Each stencil is a standalone C function that returns a `SEXP` (the opaque pointer representing runtime values in the R virtual machine) and takes no arguments (line 1). Stencils use the standalone header file from Section 3.1 that contains the extracted bytecode implementation. This header is included in the stencil source file, and its functions are called directly from stencils, `Rsh_BrIfNot` (line 2) in this case. Function calls are always inlined, embedding the complete implementation within each stencil.

```
1  SEXP _RCP_BRIFNOT_OP (void) {
2    if (Rsh_BrIfNot(
3      *(R_BCNodeStackTop - 1),
4      (SEXP)&_RCP_CONST_AT_IMM0,
5      ...
6    )) {
7      return _RCP_EXEC_IMM1();
8    } else {
9      return _RCP_EXEC_NEXT();
10   }
11 }
```

Listing 1. Example stencil implementation.

The function's exit points (lines 7 and 9) return calls to symbolic placeholders representing either the branch target or the next instruction. Holes generated from these external symbols are filled during the patch phase with actual instruction addresses. Since the callee shares the same prototype, the compiler emits a tail-call, optimizing the control flow into a single `jmp` instruction. This is a key requirement for efficient copy-and-patch implementation [21]. When this instruction is the last in the function and refers to the next instruction (line 9), the peephole optimizer can eliminate the jump entirely, creating a fall-through.

R bytecode instructions often have immediate arguments, such as constants or jump labels. Rather than passing them at runtime, which would consume registers and introduce indirection, we patch these values directly into the code using external symbols as placeholders. The most common patched arguments are a SEXP representing an object from the constant pool (line 4) and a jump label (line 7). The extractor parses the placeholder target name to determine which original bytecode argument to patch (0 and 1 in this example).

Since R bytecode is stack-based, instructions operate on stack values. Stack pointer adjustments are performed at function entry and exit when required. The stack pointer is a global variable patched with the correct address during compilation. Stack values are accessed using pointer arithmetic (line 3).

In the final system, all stencil implementations are collected into a single compilation unit, ready for the extraction and patching phases that form the core of the copy-and-patch strategy.

3.3 Stencil Specialization at Compile Time

Stencil specialization is a key optimization technique previously introduced in [21]. Its applicability in R bytecode is limited due to the dynamic nature of stack values, which are not known at compile time. However, certain bytecode instructions carry immediate operands that are known during compilation and enable targeted optimizations.

We generate specialized stencil versions for instructions where parts of the logic can be resolved ahead of time. A notable example is the `LDCONST` instruction, which loads a constant from a pool using an immediate index. While all constants are represented as `SEXP` at runtime, their types (`int`, `double`, or `logical`) are known at compile time.

This type information allows us to precompute conversions from `SEXP` to primitive values. Specialized stencils can load these primitives directly, reducing runtime overhead. Each variant is implemented as a distinct stencil. The copy-and-patch mechanism selects the optimal version during code generation.

3.4 Stencil Specialization at Runtime

R's dynamic type system causes redundant type dispatch at runtime. We mitigate this overhead using self-modifying code for runtime stencil specialization when type information becomes available.

This approach extends the standard copy-and-patch model. Instead of copying a single stencil into executable memory and patching during compilation, we pre-patch all possible stencil variants in auxiliary memory. We reserve sufficient space in the executable segment for the largest variant. At runtime, when type information is known, we copy the appropriate pre-patched stencil into place without additional patching.

The `for` loop in R demonstrates this technique. R implements loops using three instructions: `STARTFOR`, `STEPFOR`, and `ENDFOR`. `STARTFOR` initializes the loop and executes once. `STEPFOR` executes on each iteration, checking the condition and advancing the counter. Since R supports iteration over more than 10 different types, `STEPFOR` contains a type switch for all cases. The iteration type remains constant during loop execution, making this switch unnecessary overhead. We generate specialized `STEPFOR` stencils for each type and install the correct variant during `STARTFOR` execution.

3.5 Avoiding Callee-Saved Register Overhead

Each stencil is implemented as a standalone function that adheres to standard calling conventions. On x86_64 Linux systems, this requires compliance with the System V ABI, which mandates preserving several callee-saved registers. However, this introduces unnecessary overhead since stencil code is not invoked via regular function calls but executed sequentially from copy-patched machine code.

We leverage the `no_callee_saved_registers` attribute introduced in GCC 14. This attribute informs the compiler that it need not preserve callee-saved registers. Unnecessary `push`/`pop` instructions are eliminated, resulting in smaller and faster stencils. When a call to such a function occurs as the final instruction, the compiler removes all register preservation logic entirely.

This usage aligns with stencil execution, where control transfers to the next stencil via a known placeholder. Applying the attribute to that placeholder ensures compact and efficient stencil code. The compiler can utilize six additional registers without saving them in the function prologue. The CPython copy-and-patch implementation uses a similar optimization with LLVM [16].

3.6 Patching Values Instead of Addresses

In the copy-and-patch model, immediate bytecode values are patched into stencils via `extern` variables. Compilers typically emit relocations for their *addresses*, introducing unnecessary indirection when the actual value is known.

We eliminate this overhead by exploiting compiler behavior: when an `extern` variable address is explicitly taken using the `&` operator, the relocation encodes the address value. Casting this address to the desired type allows our tooling to patch the final value directly, bypassing memory indirection.

This approach declares symbols as `extern` arrays, ensuring 64-bit absolute relocations even in the medium memory model that can be used for compilation.

The resulting change in the machine code is that the `mov` instruction that loads from a 32-bit relative address changes into a `mov` instruction with a 64-bit immediate value, avoiding the memory lookup seen in the unoptimized case.

In addition to removing a memory access, the compiler can now treat constants as immutable, enabling further optimizations such as load elimination.

The base optimization has already been used in Lua JIT [5] and CPython [16], but we enhance it further by introducing the novel idea of declaring symbols as arrays to force the use of absolute relocations.

3.7 Relative Memory Model

Due to the way memory addressing works on x86_64, it can address symbols using three general ways (simplified). First, and the fastest way to address, is using PC-relative instructions, where the accessed memory has to be (in bytes) within reach of a 32-bit signed integer. This provides the fastest and smallest instructions, some of which can, due to the nature of CISC, perform more complex operations encoded in one instruction. The second way is to address using an absolute 64-bit integer. This makes for larger instructions (8 vs. 4 bytes for the operand), which most often also cannot accomplish the same complex task in one instruction, like the PC-relative ones. For example, it cannot perform a call to a function without first loading its address into a register. The third way to address is using an indirection—a table is created in memory that stores absolute 64-bit addresses to symbols, and it is placed near the executable code, so it can be accessed in PC-relative manner.

In the patching phase of copy-and-patch, the program must process relocations whose addresses cannot generally be known ahead of time during the compilation of stencils. This makes it impossible to use the PC-relative memory model by default, as some of the symbols may reside outside the relatively addressable area. However, it is likely that some symbols will still fall into the relatively addressable range, and could therefore take benefit from the faster relative model.

To solve this in the most efficient way, the copy-and-patch determines for each relocation (hole) whether its address is near enough to be used with a PC-relative instruction, and in such a case burns it into the stencil. If the address is not within reach, it uses the third described option, an indirection table, to store the absolute address of the symbol.

To be able to replace instructions in such a manner without complex readjusting of the machine code, the stencils need to be compiled with a memory addressing mode that generates instructions that can, in their stead, fit the PC-relative instructions if the optimizing replacement should take place.

A call in the indirect memory model is encoded using 6 bytes, while in the PC-relative model it takes just 5. The indirect memory model is therefore chosen in the stencil compiler. The one additional byte is replaced with a NOP instruction if needed.

The usage of the relative memory model is a key optimization in copy-and-patch and as such has been used in related projects such as CPython [16]. Our project, however, extends this functionality by choosing the location in memory where the program will be placed, optimizing it in such a way as to ensure the compiled program can make use of as many PC-relative instructions as possible. For our use case, it was determined that almost all relocations point to R runtime functions and global variables, so the copy-and-patch tries to find free space in memory near where the R runtime was loaded. It accomplishes this by parsing the `/proc/self/maps` file, finding a free space in memory close to R runtime symbols. It then uses this information to request memory at this address using the `mmap` system call, which can accept a hint where the memory should be allocated. It is important to note here that the prototype targets just one platform and architecture, supporting this for multiple would involve additional engineering costs.

3.8 Feedback Recording

As we do not want to change the bytecode, support for feedback recording must be implemented in the baseline JIT. The feedback should provide sufficient information for future upper-tier optimizing compilers to perform speculations and specializations.

We record three kinds of feedback: the types of values for every load, store, and call, whether branches were taken, and call site target closures. Each closure has a feedback vector, an array of feedback slots. Each feedback kind is organized in a lattice, and the size of each feedback entry is upper bounded (*e.g.*, we record up to three callee targets).

Feedback recording adds significant overhead (between 15% and 30% for the benchmarks used in Sec. 4); therefore, we do not want to start recording too early. Furthermore, recording early may lead to capturing unrepresentative behavior, which can hinder the optimization process due to feedback pollution [7]. Therefore, we implement lazy feedback allocation, starting to record only after a function has been executed several times. Since the copy-and-patch JIT compiler is fast, we can afford to simply recompile the entire function with feedback recording enabled. This is accomplished by adding new recording stencils after instructions that produce feedback.

4 Evaluation

In this section we evaluate the performance of our copy-and-patch JIT. Concretely, we focus on compilation speed and the resulting code performance.

4.1 Methodology

Our benchmarking suite consists of 57 programs spanning from micro-benchmarks to algorithmic problems and real-world applications from well-established benchmark suites. Some programs are variants that implement different solutions to the same problem (*e.g.* recursion, loops, vectorized operations). The benchmarks are organized into four suites, based on their origin:

- μ: Small code fragments known within the R community for their poor performance. Like any micro-benchmarks, they are too limited for drawing broad conclusions, however their simplicity facilitates detailed performance analysis.
- *AreWeFast*: Three benchmarks translated to R (by Kalibera *et al.* [6]) from the established AreWeFast suite [10], including Bounce (a bouncing balls physics simulation), Mandelbrot (computing the Mandelbrot set), and Storage (a tree creation program).
- *Shootout*: Programs from the well-known Computer Language Benchmarks Game [4]. This suite features multiple implementations of classic algorithms that explore different programming styles. Since most original programs used explicit loops, the suite includes more idiomatic R versions that leverage vectorized operations.
- *RealWorld*: Three representative applications. Flexclust implements a clustering algorithm from the flexclust package [9]. Convolution performs matrix updates through nested loops, representing code typically rewritten in C for performance. Volcano conducts ray-casting simulation across terrain using the built-in volcano elevation dataset, based on code by Morgan-Wall [12].

Table 1 summarizes the benchmark suite: the execution time in seconds, the bytecode size as the number of instructions and the source lines of code excluding blank lines and comments[1]. The numbers are reported as mean ± stddev (min .. max).

To run the experiments, we used a dedicated four-core i7-7700K CPU 4.2 GHz, with 32 GB of RAM and Ubuntu Noble with 6.11.0-24 kernel. For the baseline performance

[1]As reported by the `cloc` tool.

Table 1. Summary of the benchmark suite.

Benchmark suite	Count	Mean execution time [s]	Number of instructions	Lines of code
AreWeFast	5	0.08 ± 0.1 (0.007 .. 0.3)	125.6 ± 46.4 (60 .. 190)	44.4 ± 13.9 (20 .. 55)
RealThing	6	7.8 ± 8.7 (0.2 .. 23)	547.5 ± 677.6 (84 .. 1,809)	122.5 ± 168.8 (18 .. 448)
Shootout	33	1.2 ± 1.2 (0.04 .. 5.1)	293.6 ± 306.5 (62 .. 1,897)	62.2 ± 56.4 (10 .. 333)
μ	13	0.4 ± 0.3 (0.001 .. 1)	36.5 ± 51.7 (10 .. 207)	10.7 ± 13.4 (4 .. 55)
Total	57	1.5min	14,077	3,148

experiment, we used GNU R version 4.3.2 compiled with GCC 14.2.0. The same compiler was used for generating the stencils library[2]. Every benchmark program was run in a fresh R process 15 times, with the first 5 discarded as warm-up runs. Since the compiler does not use any runtime profile information, we pre-compile each function before running the benchmarks, effectively using it as an ahead-of-time compiler. Feedback recording was not present.

Reporting benchmarks results is challenging due to non-determinism in processors combined with the possible side-effects of the operating system. To counter that we have disabled CPU frequency scaling, turbo boost and address space layout randomization. Furthermore, we have run each benchmark with a fixed CPU affinity.

The raw data used in our evaluation section are available at https://github.com/PRL-PRG/rcp/actions/runs/17174343478.

4.2 Compilation Speed and Size

One of the premises of the copy-and-patch approach is that it achieves very fast compilation times. Once the stencils are extracted, the compilation is essentially a `memcpy` plus patching and minor peephole optimizations. Figure 2 shows the compilation speed of the compiler on the benchmark suites. It is linear in the number of bytecode instructions, with a mean of 980 bytecode instructions per millisecond. The compilation time is mostly dominated by the size of the bytecode. However, the size of the stencils and the number of holes varies:

Stencil size [Bytes]	602.5 ± 525.7 (6 .. 2,899)
Stencil holes	29.1 ± 22 (1 .. 104)

Larger stencils tend to have more holes. The biggest stencils are related to assignments and subsetting operations having over 2 kB with over 80 holes. These are large because they optimize common cases where the target is a primitive vector that may be unshared, the index is a scalar integer, and in assignments, the right-hand side is a scalar of the same type as the vector. But even `GetVar` has 1,525 bytes and 75 holes contributing to the large binary sizes shown in Figure 3. This stencil size effect is visible in outliers like `pidigits` and `flexclust`, which differ by fewer than 100 instructions (4%) yet produce binaries that are more than 20% larger.

[2]with switches `-Os -fno-math-errno -fno-trapping-math -mcmodel=medium -fno-pic -fno-plt -fno-jump-tables -fno-stack-protector -fcf-protection=none -ffunction-sections -fno-asynchronous-unwind-tables`

Table 2. Summary of benchmark speedups

Category	Count	Distribution
Faster	47	1.32 ± 1.09 (1.01 .. 6.58)
Same	6	1 ± 0.0059 (0.99 .. 1.01)
Slower	4	0.96 ± 0.023 (0.93 .. 0.99)

4.3 Performance

Figure 4 shows the speedup of the compiled code over the GNU R bytecode interpreter. We compute the speedup by bootstrapping the ratio of mean values of benchmark runtimes between the compiled code and the GNU R bytecode interpreter. As performance measurements are inherently noisy, bootstrapping provides confidence intervals for the speedup estimates by resampling the observed data. Out of the 57 benchmarks, 47 benchmarks are faster, 6 benchmarks are on par, and only 4 benchmarks are slower than the interpreter with the worst slowdown of 0.93 (*cf.* Table 2).

The average speedup on the entire benchmark suite over the interpreter is 1.26x. The largest improvements come from the μ suite (1.91x). However, the JIT also managed to speed up some of the larger and more complex programs: RealThing by 1.23x, AreWeFast by 1.16x, and Shootout by 1.08x. This is a good result, considering that the compiler does only a few simple code optimizations (value patching and stencil specialization). Given the nature of the bytecode, the dispatch in the interpreter is not high, and neither is instruction decoding (two common things targeted by baseline JITs). The fact that the stencils are large is making the generated code less cache friendly. It could perform better in the case of thinner bytecode instructions as is the case in Xu *et al.* [21].

The benchmarks where the JIT performs significantly better than GNU R are the ones in which there is a large number of loop iterations, and the loop code is dominated by scalar arithmetics over unboxed values. This is the best scenario in

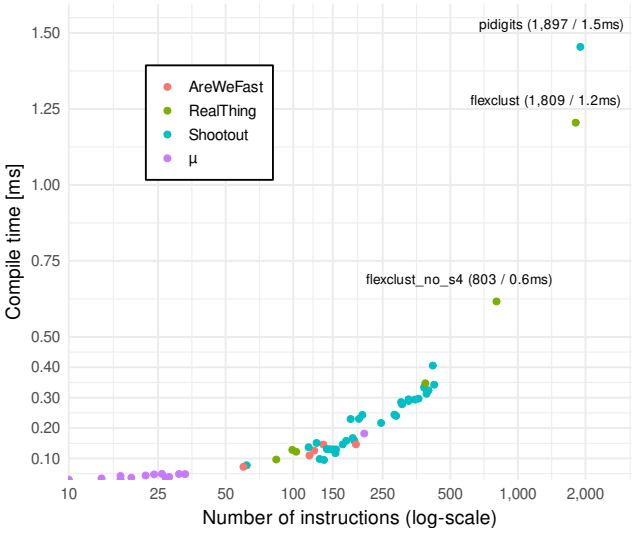

Figure 2. Compilation speed.

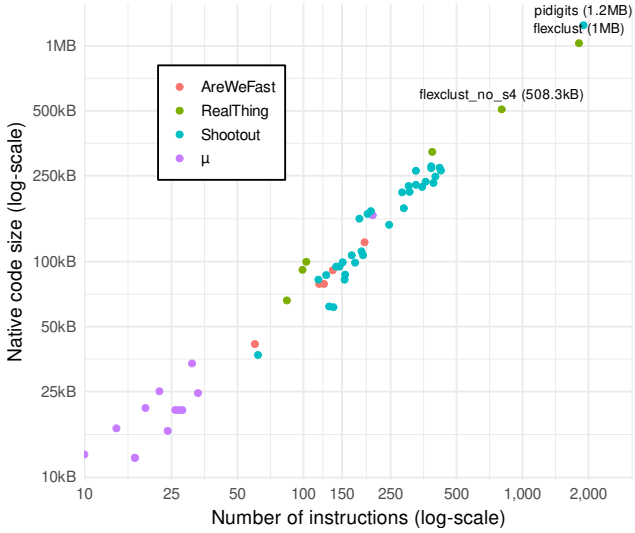

Figure 3. Compilation size of the generated native code.

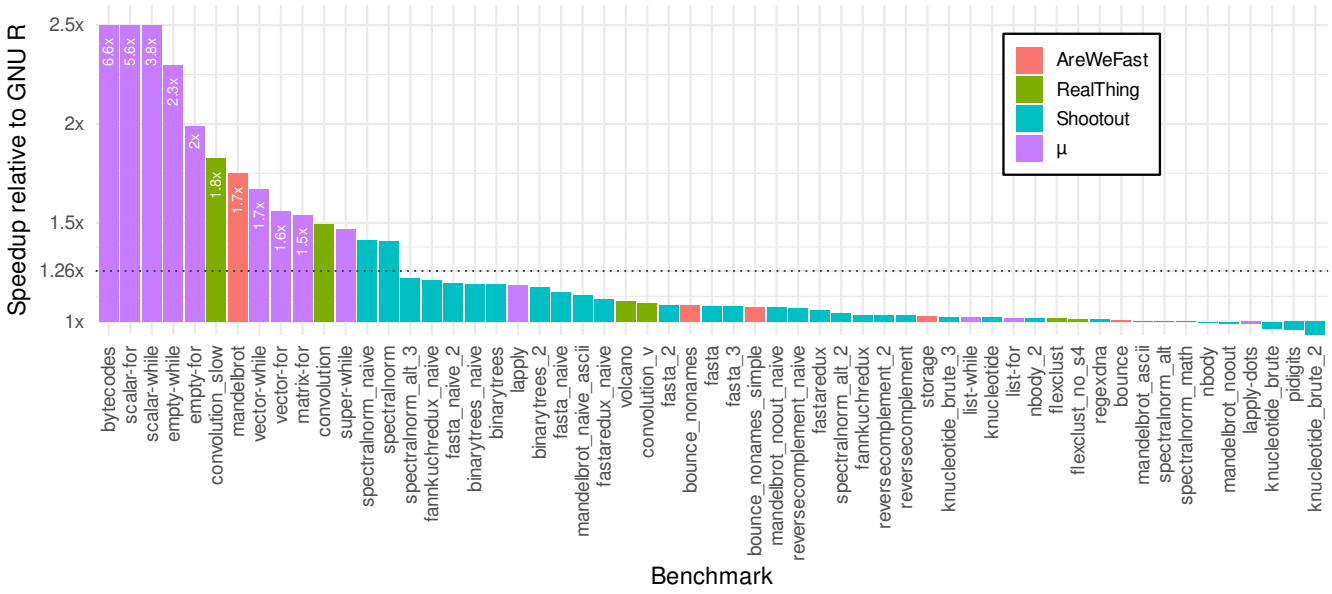

Figure 4. The speedup over GNU R. The dotted line indicates the geometric mean.

which we can observe the effect of dispatch and instruction decoding overhead removed.

Lastly, it is important to note that, since we had to adapt the implementation of bytecode instructions for our use case, we do not share the exact same code as the GNU R interpreter, and it is therefore impossible to do a clean performance comparison of just the copy-and-patch technique itself (like it could be done in related projects). This likely means that there exist performance differences that may not come from the copy-and-patch approach, but are instead due to the differences in the implementation of the instructions. Some of the performance benefits shown in this section could therefore have been related to an issue in the interpreter which might miss an unboxing opportunity or similar optimizations.

4.4 Performance of Optimizations

From the optimization techniques that we have implemented, the major contributor to the performance improvements is the use of the relative memory model. This model allows for more efficient memory addressing by using 32-bit relative offsets instead of full 64-bit absolute addresses. This

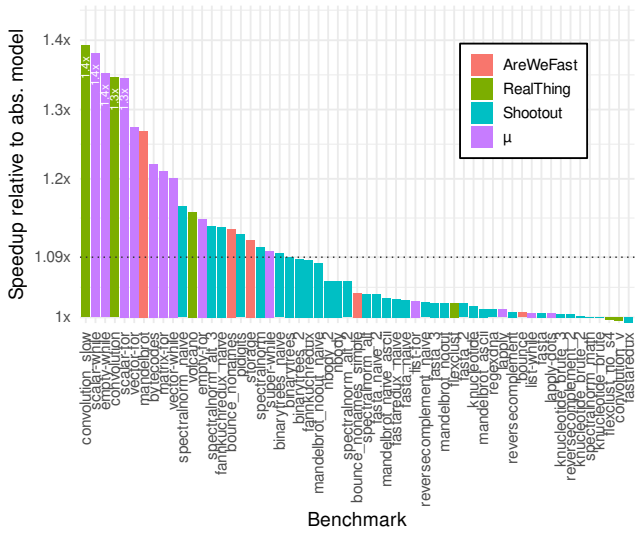

Figure 5. Comparison of relative to absolute memory models. The dotted line indicates the geometric mean.

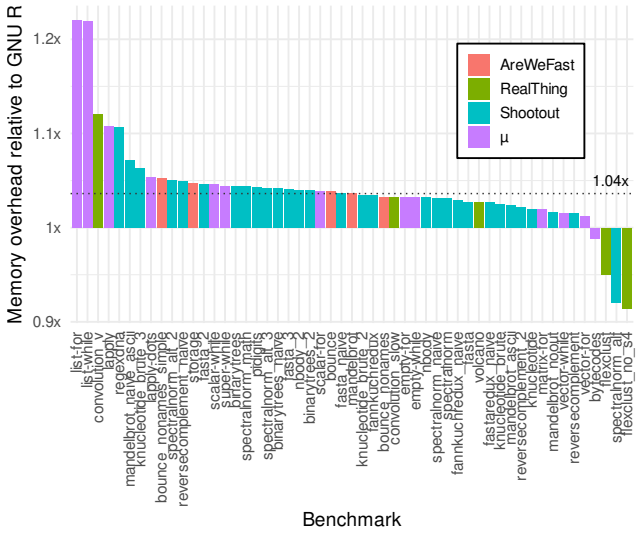

Figure 6. Memory overhead compared to GNU R. The dotted line indicates the geometric mean. Only benchmarks with differences greater than 1% are shown.

reduces instruction encodings and results in better instruction cache utilization, and faster address calculations. The stencils, which are large, benefit from this approach as they can use shorter, more efficient addressing modes for accessing runtime data structures and function parameters.

Figure 5 shows the improvement over the absolute 64-bit model for each benchmark. In our testing environment, 95 out of 116 total relocations (82%) were replaced by the faster PC-relative addressing. The geometric mean of the comparison speedup is 1.09x, with the results showing consistent improvements of varying magnitude over the absolute memory model across all benchmark suites. Out of the 57 benchmarks, 44 benchmarks are faster and 13 are on par, with none being slower. It is likely that the differences in speedup across benchmarks are due to varying number of calls to internal R runtime functions or accesses to internal R global variables, which cause the most overhead in absolute memory model.

Compared to this, the optimization of self-modifying `for` loops was not as widely successful, providing measurable performance benefits mostly just for benchmarks specifically designed to test the performance of `for` loops. This consisted of 3 benchmarks[3] from the μ suite (in the range of 1.06x–1.11x speedup), and just one benchmark from the RealThing suite[4] (1.04x speedup). Other benchmarks were on par or showed results too noisy for sound comparison.

4.5 Memory Overhead

We measure memory overhead by recording the maximum resident set size (RSS) of the R process for each benchmark execution using the `time`[5] command.

The JIT shows higher memory consumption in 49 benchmarks, lower consumption in 4 benchmarks, and comparable consumption within 1% in 4 benchmarks. The geometric mean RSS normalized to GNU R across all benchmarks is 1.04 (0.9 .. 1.2). Figure 6 shows memory overhead for benchmarks where the difference exceeded 1%. The JIT allocates memory primarily for compiled stencils and functions. Lower memory consumption in some benchmarks results from changes to the allocation profile, which affects garbage collection behavior. This is also the side-effect of implementation differences between bytecode instructions discussed in Section 3.1.

5 Related Work

The copy-and-patch approach has been used for different languages, *e.g.*, WebAssembly [21] or various database query languages [8, 21]. However, the most relevant use for our work has been its deployment to Python and Lua.

Copy-and-Patch in CPython. Our work has been inspired by the experimental copy-and-patch JIT [2] introduced in CPython 3.13 [16]. However, there are two major differences.

First, the Python version is implemented directly in the CPython virtual machine, leveraging the recently added DSL for bytecode specification [17]. The stencils are therefore

[3] `scalar-for`, `empty-for` and `vector-for`
[4] `convolution_slow`

[5] The `time(1)` command from the `time` Ubuntu package, *cf.* https://packages.ubuntu.com/noble/time

directly generated from the same definition as the bytecode interpreter, which significantly reduces the maintenance burden.

Second, in Python the compiler does not compile entire functions; instead, it works on linear traces. Specifically, it compiles optimized micro-op traces that originate from the specializing adaptive interpreter [18]. This interpreter collects profiling information and detects hot code paths. Furthermore, it implements quickening [1], *i.e.*, in-place bytecode rewriting with their type-specialized versions. Hot code traces are then compiled into micro-ops, fine-grained operations that are further optimized (*e.g.*, removing redundant checks). Only then is the copy-and-patch JIT compiler invoked to compile these traces into machine code. The size of the trace is upper-bounded to constrain memory usage. Therefore, in Python, the compiler is not merely a baseline JIT.

The specializing adaptive interpreter, together with various other improvements that have landed since CPython 3.11, makes it harder for the JIT to deliver any significant speedup so far. With modern C compilers that perform well with threaded interpreter style, the dispatch overhead is rather small. The result is that the technique is less effective in this context. According to benchmarks published by Wouters [20], the geometric mean speedup across benchmarks ranges from 0.95x to 1x.

Copy-and-Patch in Lua. The LuaJIT Remake project [5] by Haoran Xu represents the primary implementation of the copy-and-patch approach. The project aims to reimplement the multi-tier LuaJIT virtual machine [13] from scratch.

The current implementation uses the Deegen meta compilation framework [22]. Deegen generates an efficient bytecode interpreter and baseline JIT from DSL and C++ definitions of bytecode instructions. The interpreter incorporates state-of-the-art optimizations from JavaScriptCore [15] and outperforms the original LuaJIT on most benchmarks. The copy-and-patch JIT compiler includes polymorphic inline caches, JIT hot-cold code splitting, and on-stack replacement.

Our implementation shares several techniques with LuaJIT Remake: patching immediate values as addresses (*cf.* Section 3.6), tail-call optimization with fall-through jump elimination, and self-modifying code. While additional techniques could be adopted, R's semantic complexity over Lua makes them more challenging to implement. Furthermore, the Deegen-generated compiler takes two shortcuts that we could not take: it does not consider garbage collection and ignores C bindings.

6 Discussion

This paper addresses whether the copy-and-patch approach is feasible for the R programming language: whether we can build a baseline JIT compiler that: (1) can be implemented outside the GNU R virtual machine with reasonable effort, (2) supports lazy feedback recording, and (3) delivers performance improvements over the GNU R bytecode interpreter.

Effort. Extracting bytecode instructions from the monolithic interpreter required non-trivial effort. However, we believe this approach is simpler than designing new bytecode with integrated feedback recording support. An alternative template JIT approach (essentially copy-and-patch without patching) would be less effective given R's bytecode characteristics. The dispatch and instruction decoding costs are relatively low, making straightforward template compilation insufficient. The low-level optimizations enabled by patching do improve code quality.

Lazy Feedback Recording. The fast compilation speed enables recompilation, which greatly simplifies lazy feedback recording implementation. Moreover, this approach also enables other dynamic program analyses, such as code coverage or profiling.

Performance. The generated code shows some performance gains, with largest improvements primarily concentrated in micro-benchmarks with scalar arithmetic operations. Larger, more complex programs still benefit from the JIT, but with a smaller margin. This indicates that effectiveness of this technique is limited by R's larger bytecode instructions and the absence of high-level optimizations. Several optimizations could improve performance further: profile-guided optimization of stencil compilation, selective inlining in stencil code (avoiding inlining for slow paths to reduce binary size), and extended self-modifying code (runtime instruction specialization). However, these optimizations would increase compiler complexity, reducing the technique's main benefit. Furthermore, their performance impact would likely be limited. In R, significant performance improvements require high-level optimizations: inlining, type speculation, and operation elision. Removing explicit callee environments, skipping promise creation, and avoiding copies provide the largest performance gains. These optimizations require an upper-tier optimizing compiler. However, such optimizations need program behavior information, which our baseline compiler provides.

Limitations. The copy-and-patch approach has limitations. Supporting additional architectures is non-trivial. While the C compiler covers the stencils code, new architectures require custom stencil generators and patch mechanisms. We support R debugging and profiling by dropping native code and executing in the AST interpreter. However, debugging or profiling native code directly would require significant additional effort. Similar issues arise with other compilation approaches.

7 Conclusion

This paper explores the feasibility of copy-and-patch compilation for the R programming language by implementing a baseline JIT compiler that operates outside the GNU R virtual machine. We extracted standalone bytecode instruction implementations from the monolithic interpreter, created a stencil library, and developed a compiler that achieves 980 instructions per millisecond compilation speed. The implementation builds on several optimizations introduced in [16], [5] and [21], adapting these for the use with R, and extending some of them with novel ideas applicable for our use case. The compiler also supports lazy feedback recording through function recompilation.

The evaluation on 57 benchmarks demonstrates promising performance results, achieving 1.26x geometric mean speedup over GNU R. The largest speedups come from micro-benchmarks, but more complex benchmarks still benefit from the JIT, achieving 1.15x speedup.

The copy-and-patch approach proves viable for R as a foundation for multi-tier compilation. Other than the performance gain, it offers fast compilation and effective feedback recording capabilities.

While additional compilation tricks such as performance-guided optimizations of the stencils could yield further performance gains, we believe that major improvements require deeper analysis and high-level optimizations. Pursuing such optimizations would compromise the intentional implementation's simplicity provided by this approach. However, this baseline JIT serves its purpose well, providing essential feedback to guide the next-generation compiler.

For the future work, we plan to test it on larger R programs such as computational notebooks and allow full JIT compilation of R packages.

Acknowledgments

We thank the anonymous reviewers for their insightful comments and suggestions to improve this paper. We also give special thanks to Brandt Bucher for his consultation and valuable feedback. This work was supported by the Czech Science Foundation grant 23-07580X and by the Czech Ministry of Education, Youth, and Sports under the ERC-CZ program (grant agreement LL2325), as well as by NSF grants CCF-1910850, CNS-1925644, and CCF-2139612.

References

[1] Stefan Brunthaler. 2010. Efficient interpretation using quickening. (2010). doi:10.1145/1899661.1869633

[2] Brandt Bucher and Savannah Ostrowski. 2024. PEP 744 – JIT Compilation. https://peps.python.org/pep-0744/.

[3] Olivier Flückiger, Guido Chair, Ming-Ho Yee, Jan Ječmen, Jakob Hain, and Jan Vitek. 2020. Contextual Dispatch for Function Specialization. *Proc. ACM Program. Lang.* 4, OOPSLA (2020). doi:10.1145/3428288

[4] Isaac Gouy. [n. d.]. Computer Language Benchmarks Game. https://benchmarksgame-team.pages.debian.net/benchmarksgame/.

[5] Haoran Xu. [n. d.]. LuaJIT Remake. https://github.com/luajit-remake/luajit-remake.

[6] Tomáš Kalibera, Petr Máj, Floréal Morandat, and Jan Vitek. 2014. A Fast Abstract Syntax Tree Interpreter for R. In *Conference on Virtual Execution Environments (VEE)*. doi:10.1145/2576195.2576205

[7] Sebastián Krynski, Michal Štěpánek, Filip Říha, Filip Křikava, and Jan Vitek. 2024. Reducing Feedback Pollution. In *Workshop on Virtual Machines and Intermediate Languages (VMIL)*. doi:10.1145/3689490.3690404

[8] Wilco v. Leeuwen, Thomas Mulder, Bram van de Wall, George Fletcher, and Nikolay Yakovets. 2022. AvantGraph query processing engine. *Proc. VLDB Endow.* 15, 12 (2022). doi:10.14778/3554821.3554878

[9] Friedrich Leisch. 2006. A Toolbox for K-Centroids Cluster Analysis. *Computational Statistics and Data Analysis* 51, 2 (2006).

[10] Stefan Marr, Benoit Daloze, and Hanspeter Mössenböck. 2016. Cross-language compiler benchmarking: are we fast yet?. In *Symposium on Dynamic Languages (DLS)*. doi:10.1145/2989225.2989232

[11] Floréal Morandat, Brandon Hill, Leo Osvald, and Jan Vitek. 2012. Evaluating the Design of the R Language: Objects and Functions for Data Analysis. In *European Conference on Object-Oriented Programming (ECOOP)*. doi:10.1007/978-3-642-31057-7_6

[12] Tyler Morgan-Wall. 2018. Throwing Shade: Raytracing and Rayshader. https://www.tylermw.com/posts/rayverse/throwing-shade.html.

[13] Mike Pall. [n. d.]. A Just-In-Time Compiler for Lua. LuaJIT. https://luajit.org/.

[14] Ian Piumarta and Fabio Riccardi. 1998. Optimizing direct threaded code by selective inlining. (1998). doi:10.1145/277652.277743

[15] Filip Pizlo. 2020. Speculation in JavaScriptCore. https://webkit.org/blog/10308/speculation-in-javascriptcore/.

[16] Python Software Foundation. 2023. GH-113464: A copy-and-patch JIT compiler. https://github.com/python/cpython/pull/113465.

[17] Python Software Foundation. 2025. Python bytecode definitions in C-like DSL. https://github.com/python/cpython/blob/main/Python/bytecodes.c.

[18] Mark Shannon. 2021. PEP 659 – Specializing Adaptive Interpreter. https://peps.python.org/pep-0659/.

[19] Luke Tierney. 2023. A Byte Code Compiler for R. http://homepage.stat.uiowa.edu/~luke/R/compiler/compiler.pdf.

[20] T. Wouters. 2025. Python Benchmarking Public. https://github.com/Yhg1s/python-benchmarking-public.

[21] Haoran Xu and Fredrik Kjolstad. 2021. Copy-and-patch compilation: a fast compilation algorithm for high-level languages and bytecode. 5, OOPSLA (2021). doi:10.1145/3485513

[22] Haoran Xu and Fredrik Kjolstad. 2024. Deegen: A JIT-Capable VM Generator for Dynamic Languages. *CoRR* (2024). doi:10.48550/ARXIV.2411.11469

Received 2025-07-14; accepted 2025-08-11