Copy & Patch Just-in-Time Compilation for R

FACULTY OF INFORMATION TECHNOLOGY CTU IN PRAGUE

Author: Ing. Matěj Kocourek | **Supervisor**: doc. Ing. Filip Křikava, Ph.D.
Faculty of Information Technology, Czech Technical University in Prague

Motivation

The R programming language is widely used for data science and research thanks to its expressive syntax, rich ecosystem of libraries, and accessibility. But R's flexibility comes with a price: as an interpreted and highly dynamic language, it runs noticeably slower than many of its alternatives, especially when compared against compiled languages.

Ř [1] is a project that tackles this problem by translating R code into efficient native code, showing large performance gains over the standard interpreter. Its weakness, however, is compilation speed — too slow for use as a baseline just-in-time (JIT) compiler, which must work interactively, compiling code in milliseconds and often multiple times during program execution.

This work explores a different approach: using the Copy-and-Patch technique [2], a minimalist strategy that trades heavy optimizations for extremely fast compilation. The goal is a prototype JIT compiler that integrates with R, builds on top of Ř, and eliminates its biggest drawback — making high-performance R code practical in more interactive and short-lived workloads.

Copy-and-Patch

Baseline JITs must compile quickly. Copy-and-patch achieves this by reusing code instead of generating it from scratch. Programs are built from stencils — tiny precompiled templates of bytecode instructions with placeholders for runtime values like constants or jump targets.

Because most work is already done ahead of time, compilation is extremely fast. Maintenance is also easy: once the stencil library exists, updates come for free. This approach is already proven in Lua and Python. Our work applies it to R, which shares a similar execution model.

Practical example [2].
(a) A stencil in C has placeholders for offsets, constants, and calls.
(b) Compilation reveals placeholder positions in machine code.
(c) These are extracted into a header file.
(d) At runtime, the engine copies the stencil and patches the placeholders to produce executable code.

```c
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

```
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
```

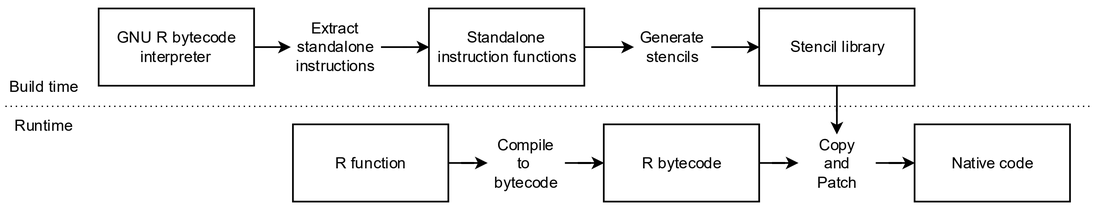
(b) Compiled executable code

```
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: {}
```

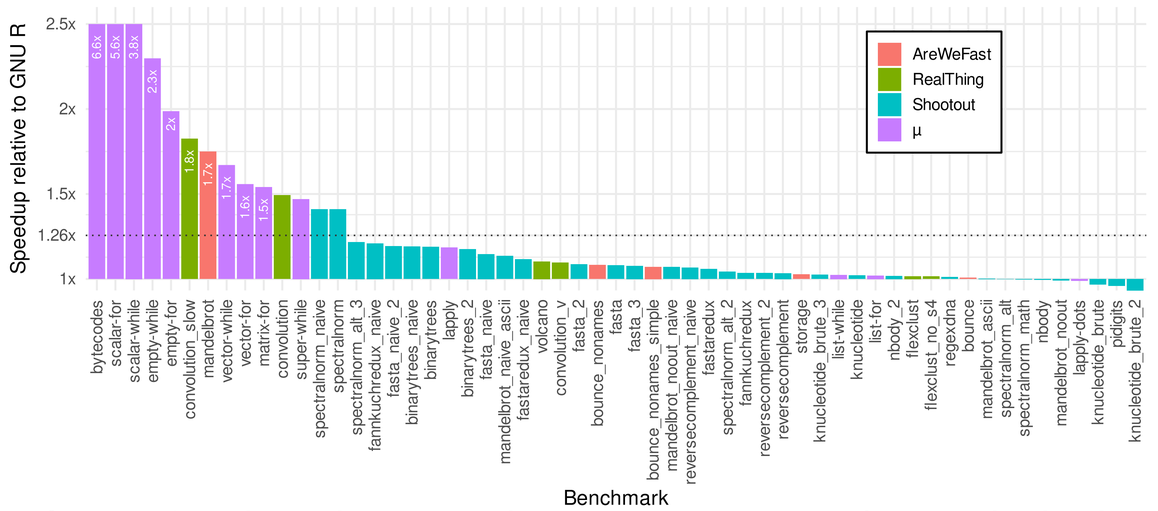
(c) Generated stencil header

```
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



The compilation pipeline. At build time, bytecode instructions are turned into stencils. At runtime, R's compiler produces bytecode, the matching stencils are selected, then copied and patched into a native function.



Speedup comparison. The speedup is measured across 57 programs spanning from micro-benchmarks to algorithmic problems and real-world applications. Every benchmark program was run 15 times, with the first 5 discarded as warmup runs. We compute the speedup by bootstrapping the ratio of mean values of benchmark runtimes between the compiled code and the GNU R bytecode interpreter.

Evaluation

The project was implemented in C, consisting of approximately **2,000** lines of code.

Out of the **57** benchmarks, **47** are faster (fastest speedup **6.6×**), **6** are on par, and only **4** are slower (slowest speedup **0.93×**) than the interpreter, with the average of **1.26×**. The largest speedups come from microbenchmarks (**1.91×**), but more complex benchmarks still benefit from the JIT, achieving **1.15×** speedup.

These results are supported by a range of implemented optimizations, both adapted from related works and novel.

With the average compilation time of just **0.25** milliseconds, the compiler is several orders of magnitude faster than the original Ř project and easily covers the requirement for a baseline JIT. Additionally, it can support collection of feedback information for heavier compilers, ready for integration in a teared execution model.

Conclusion

We successfully implemented a baseline JIT for R using copy-and-patch. Evaluation on **57** benchmarks shows a **1.26×** average speedup over GNU R and sub-millisecond compilation times, demonstrating that copy-and-patch is a fast, viable foundation for multi-tier compilation.

Publication

This project will be presented at the VMIL workshop (part of the SPLASH '25 conference in Singapore) [3], as well as to the R Development Core Team in Vienna.

References

[1] Flückinger *et al.* 2020 Contextual dispatch for function specialization. OOPSLA Proc. doi: 10.1145/3428288.
[2] Haoran Xu *et al.* 2021. Copy-and-patch compilation. OOPSLA Proc. doi: 10.1145/3485513
[3] Matěj Kocourek *et al.* 2025. Copy-and-Patch Just-in-Time Compiler for R. VMIL'25 Proc. doi: 10.1145/3759548.3763370 (submitted for publication)

Source code available at https://github.com/PRL-PRG/rcp. The results presented in this poster incorporate improvements implemented after submission of the original project. See results in the full work for historical purposes.