Leveraging Copy-and-Patch JIT Compilation for Low-Overhead Dynamic Program Analysis

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

Pierre Donat-Bouillud
Czech Technical University
Prague, Czech Republic
pierre.donat.bouillud@fit.cvut.cz

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

Abstract

Copy-and-patch compilation is an approach for building baseline just-in-time compilers from AST or bytecode interpreters. It features very fast compilation times while generating good quality code. It has been used as a baseline tier for a number of languages including Lua, Python, and R. In this paper, we explore copy-and-patch as a foundation for dynamic program analysis, and implement four analyses on top of an existing copy-and-patch JIT for R: code instrumentation, code coverage, performance profiling, and native debugging.

Two properties make copy-and-patch particularly well-suited for this purpose. First, since the code templates (stencils) are compiled by a standard C compiler, DWARF debug information is available as a by-product. This lets us reuse native tooling for profiling and debugging directly, with R and native code appearing in a single unified view. Second, analyses that operate at the bytecode level reduce to injecting stencils into the bytecode stream at JIT compile time, incurring minimal overhead. Our prototype evaluation yields instrumentation with no measurable overhead; code coverage that is faster than the existing solution while covering 17% more lines; and profiling that remains, on average, faster than GNU R.

CCS Concepts: • **Software and its engineering** → **Dynamic analysis**; **Just-in-time compilers**; **Software testing and debugging**; **Software performance**; *Runtime environments.*

Keywords: copy-and-patch, just-in-time compilation, dynamic analysis, runtime instrumentation, type tracing, code coverage, profiling, debugging

MPLR '26, Brussels, Belgium

ACM ISBN 979-8-4007-2433-6/2026/06
https://doi.org/10.1145/3828170.3828176

ACM Reference Format:
Matěj Kocourek, Filip Křikava, Pierre Donat-Bouillud, and Jan Vitek. 2026. Leveraging Copy-and-Patch JIT Compilation for Low-Overhead Dynamic Program Analysis. In *Proceedings of the 23rd ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes (MPLR '26), June 30, 2026, Brussels, Belgium.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3828170.3828176

1 Introduction

Dynamic programming languages are widely popular due to their support for rapid development and ease of use. However, beyond simple scripts, developers often need to understand various aspects of a program's behavior, such as its performance characteristics or which parts of the code are exercised by its tests. Language designers may also want to study dynamic behavior—for instance, which patterns programmers use—to improve language ergonomics and performance [40]. Each of these dynamic program analysis tasks must be implemented in the language runtime without degrading performance.

A further concern arises from the mixing of dynamic languages with machine code. Many popular extensions in Python [3, 35] or Ruby [6] wrap native-compiled libraries. The R standard library includes over 1,400 built-ins in C or Fortran, and R programmers routinely turn to writing native-compilable code for performance-critical work [32]. Presenting dynamic analysis results for both the dynamic language and its accompanying native code in a single unified view is therefore highly desirable.

One way to unify the dynamic language and its native code is to compile the dynamic language to native code as well, which is what just-in-time (JIT) compilers do. In this paper, we focus on the copy-and-patch compilation approach, a technique for building baseline JIT compilers from bytecode interpreters, used for Lua (LuaJIT Remake) [18], Python (CPython) [33], and R (RCP) [23].

Concretely, we present an extension to RCP that adds support for four common dynamic analysis tasks: dynamic code instrumentation with type tracing and code coverage,

performance profiling, and debugging. While the technique should be applicable to any language with a copy-and-patch JIT, we focus on R for two reasons. First, R already supports all of these analyses either directly in the virtual machine or via extensions, but they are all based on AST interpretation and thus incur some overhead. Second, native code plays a significant role in R, so being able to cross the language boundary is particularly useful.

The approach has two main advantages. First, because copy-and-patch stencils are compiled by a standard C compiler, DWARF debug information is produced as a by-product, with no additional instrumentation required. This enables native profiling with `perf` and source-level debugging with `gdb`, transparently crossing the R/native boundary. Second, analyses that operate at the bytecode level—instrumentation and code coverage—reduce to injecting additional stencils at copy-and-patch compile time, keeping the implementation compact and overhead low: instrumentation adds no measurable cost over uninstrumented RCP, and coverage is roughly 2× faster than the existing solution.

Availability. The work described in this paper is an extension to an open-source virtual machine and is publicly available at https://github.com/PRL-PRG/rcp.

2 Background

R is a dynamic programming language widely used for statistical computing and data analysis. The de facto standard implementation, GNU R, includes both an AST interpreter and a bytecode compiler [36] with an accompanying interpreter.

Copy-and-patch compilation [41] works by partially evaluating each interpreter case into a pre-compiled native code template, called a *stencil*, with holes left for constants such as operand addresses and jump targets. At compile time, the JIT simply selects the stencil corresponding to each bytecode instruction, copies it into an executable buffer, and fills in the holes—no register allocation, instruction selection, or optimization passes are needed. The result is near-instantaneous compilation that still produces reasonable native code, since each stencil inherits the optimizations applied by the ahead-of-time C compiler used to build the stencil library.

The R copy-and-patch JIT compiler [23] (RCP) is based on the GNU R bytecode interpreter. It is implemented as an R package (supporting Linux x86-64), but currently requires a custom build of the GNU R virtual machine due to its reliance on a number of private VM APIs. It consists of two components:

- A stencil library containing pre-compiled native code for each bytecode instruction, extracted from the GNU R bytecode interpreter.
- A compiler that uses the stencil library to copy and patch the executable code.

The whole pipeline can be seen in Figure 1. The R source is first compiled to bytecode by the GNU R `compiler` package. Each bytecode instruction has a corresponding stencil: a small native code template produced ahead of time by compiling a C implementation of that instruction with an optimizing compiler. Symbols that depend on run-time values—operand addresses, jump targets, constants—are emitted as relocations in the stencil object file rather than as concrete addresses. The stencil extractor records both the machine code bytes and the associated relocation entries for each stencil. At JIT compile time, the compiler walks the bytecode sequence, copies each matching stencil into a contiguous executable buffer, and resolves the relocations by patching the copied bytes with the actual run-time addresses. For example, the `LDNULL_OP` stencil has a relocation for the address of R's `NULL` constant; at patch time this is replaced with the actual pointer. Adjacent stencils are stitched together by removing inter-stencil tail jumps, since the next stencil is copied immediately after the current one. The result is a single native function that executes the entire bytecode sequence without interpreter dispatch overhead.

3 Extending RCP with Instrumentation

Our extension to the copy-and-patch compiler can add custom instrumentation compiled from C into the bytecode of a function at copy-and-patch time. The instrumentation can inspect anything that an opcode can see, for instance, the local environment. As proof-of-concept demonstrations, we apply it to two use cases: function type tracing and code coverage.

3.1 Plugin Stencil Framework

The compiler supports defining *plugin stencils*, possibly during execution. These are seen by the copy-and-patch compiler as regular stencils, and work exactly the same as opcode stencils do (copy in place, patch holes). This has several advantages, mainly allowing the plugin stencils to inspect (and even modify) the runtime at the same level as a regular bytecode opcode stencil from the stencil library, making it very powerful. They have full access to the bytecode stack, execution context, and all of the R runtime functions and global variables.

The plugin stencils are inserted either (a) just before specific opcode positions in the compiled function or (b) before all occurrences of a particular opcode in the function. There is no limit to the number of plugin stencils that can be defined. If multiple plugin stencils are defined for a single position, they are queued in the order they were added. During copy-and-patch, the compiler simply emits plugin stencils at a given position before emitting regular opcodes. This makes it very easy to both use and implement, and imposes practically no additional compilation overhead.

Listing 1 shows the definition of a plugin stencil and the use of the API to insert it at position 0 in a function.

```
1 RCP_STENCIL_FUNCTION(_RCP_ENTRY_MESSAGE)
2 {
3   Rprintf("%s\n", GETCUSTOM());
4   return _RCP_EXEC_NEXT(...)
5 }
6 ...
7 const char* data = "Hello!";
8 add_plugin_stencil_pos(plugins, 0, &
      ↪ _RCP_ENTRY_MESSAGE, data);
```

Listing 1. Plugin stencil and registering it at position 0.

A plugin stencil can access a custom argument that can be used to carry information. This is implemented as a regular stencil hole, and can be accessed through the `GETCUSTOM()` macro from within the stencil. In Listing 1, the stencil will get a pointer to `data`, which contains a string that will be printed at its execution.

3.2 Type Tracing

Our aim is to trace the types of function arguments and return values at run time for all top-level functions in a package. The existing solution [38] implements this in the R language itself. We hope to achieve better performance by using the introduced dynamic instrumentation framework, which uses compiled C code instead.

R is a lazy-evaluated language, where function arguments are *promises*: they are not evaluated until their value is needed. Consequently, we can be sure about the type of an argument only after it has been evaluated. In addition, in R, arguments can be passed by name, not only by position, so we need to keep track of the names of the arguments in addition to their types.

Step 1: compilation. Function parameters are represented by a linked list with their names as R symbols. We record the symbol of the first parameter, which will be used later to detect arguments. The traces are stored in an R environment (in this context, a hashmap) with the package and function names as keys and resizable arrays as values.

Then, the crux of the instrumentation compilation is to insert a plugin stencil before each of the exit points (*exit hook*) of the functions, which correspond to the `RETURN` and `RETURNJMP` opcodes, which are the latest points at which an argument can be forced, and to bake the resizable array `trace` in it, as shown in Listing 2.

Step 2: tracing. Stencils have access to the function environment at run time. The function environment contains the local variables first, then the arguments, starting with the symbol of the first argument we have recorded at compilation time. The exit hook stencil iterates over the arguments, checking if they have already been evaluated and recording their name and their type in the result array we get with `GETCUSTOM()`. If an argument has still not been evaluated, we indicate its type is a promise. This only records the wrong

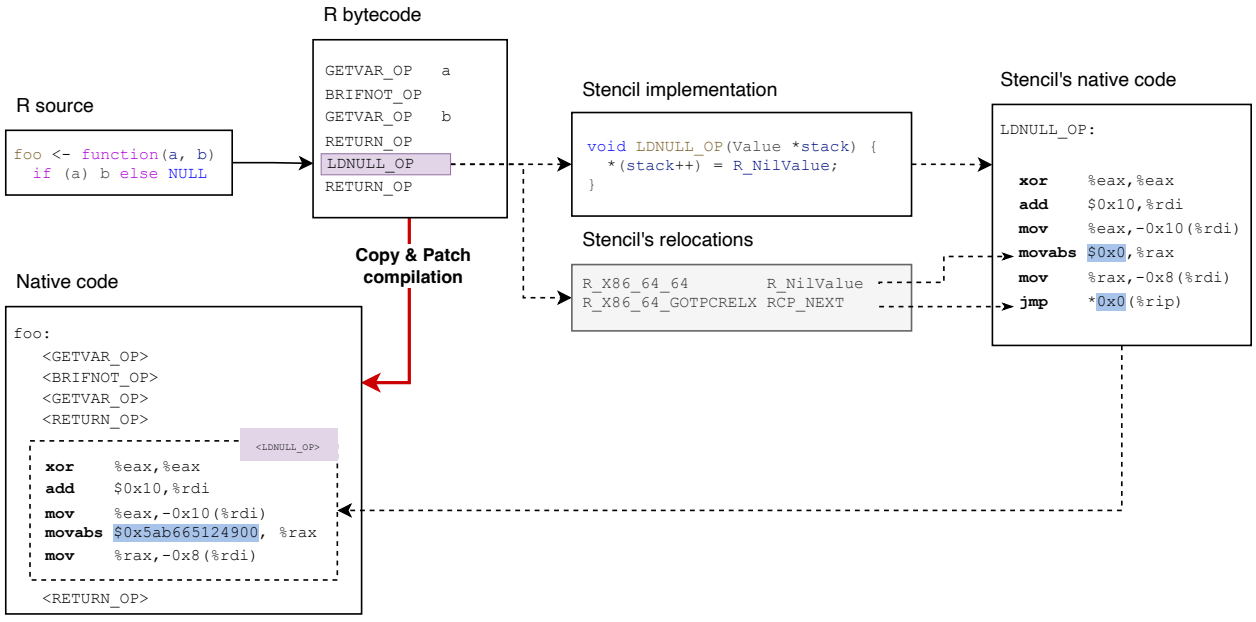

Figure 1. Overview of the R copy-and-patch compilation, from R source code to the native code.

```
1 add_plugin_stencil_instr(plugins, bytecode,
      ↪ bytecode_size, RETURN_BCOP, &
      ↪ _RCP_EXIT_HOOK, trace);
2 add_plugin_stencil_instr(plugins, bytecode,
      ↪ bytecode_size, RETURNJMP_BCOP, &
      ↪ _RCP_EXIT_HOOK, trace);
```

Listing 2. Registration of the plugin stencils as exit hooks for type tracing. `trace` is the opaque pointer carrying the collected information for the traced function.

type in the unlikely case when an argument is modified and assigned a value of another type in the body of the traced function.

The return value is the value at the top of the stack before the return opcode, so we can directly record its type.

Step 3: back to the R world. The native C pointers are converted back to R named lists of traced calls with the names of arguments and their types. Environments are more difficult to explore and inspect for R users so we also provide a helper function that returns the traced names and arguments as a data frame. Both functions have wrappers on the R side.

3.3 Code Coverage

Code coverage measures which parts of a program are exercised during execution. The standard tool for R is `covr` [19], which works by modifying R's parsed expressions to insert trace callbacks at each expression. This approach operates entirely within the AST interpreter and therefore incurs overhead proportional to the number of expressions executed.

Our approach instead builds on source references preserved through R bytecode compilation. We first explain how we refine these source references to obtain sufficiently precise source-level coverage, and then describe how we use them to place lightweight coverage instrumentation in the generated native code.

3.3.1 Source Reference Modification.

R bytecode (*BC*) compiler can attach source references (*srcref*s) to parsed expressions when source retention is enabled. A source reference is essentially a pair of begin/end locations in the original source (file or text). For functions, the parser records one *srcref* for the function itself and then one *srcref* for each top-level expression in a code block, *i.e.* each argument of the sequencing function "`{`".

The bytecode compiler preserves this information in the constant pool. It records which instruction belongs to which *srcref*. This works well for straight-line code, but can be too coarse for compound expressions. For example, the expression

```
if (f()) g() else h()
```

carries only one *srcref*, causing the guard and both branches to mark a single source location.

To recover the missing granularity, we run a preliminary AST rewriting pass before bytecode compilation. Since R 3.0.0, the parser exposes raw parse data containing a detailed table of tokens and higher-level syntactic constructs. We use this representation to identify subexpressions that need a separate source location and insert synthetic sequencing blocks with *srcref*s assigned manually. Concretely, we rewrite the above example

```
if ( { f() } ) { g() } else { h() }
```

This yields distinct srcrefs for the guard and for each branch, which then propagate through bytecode compilation and make the resulting coverage map precise enough at the source-expression level. This rewrite is completely transparent to the user and does not change the semantics of the program nor the run-time performance.

Availability. We created the described tool as an open-source R package available at https://github.com/PRL-PRG/imputesrcref.

3.3.2 Code Coverage Instrumentation.

As we aim to do code coverage and not bytecode coverage, it is unnecessary to cover each bytecode instruction separately. Instead, we aim to record information for each source reference only once.

For this purpose, we create our own structure, which serves as an "inverted" source reference array – for each unique source reference, we record the bytecode instruction which uses it first. Given this new structure, we are able to know where exactly we need to emit the coverage recording snippet. This is illustrated in Figure 2.

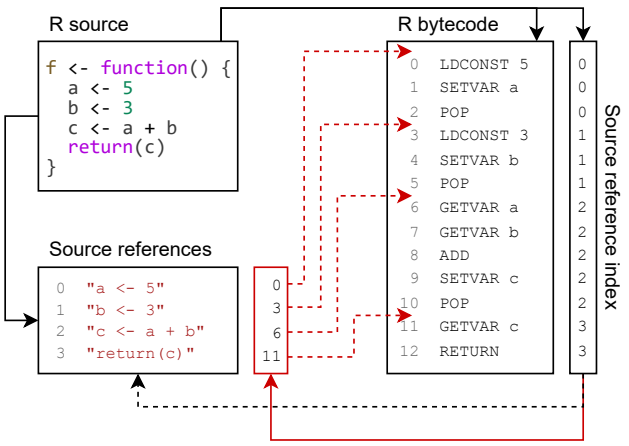

Figure 2. Source reference system in GNU R, and how we build on top of it (in red). Red arrows in bytecode denote where we emit native recording.

With the copy-and-patch JIT, we implement code coverage using the plugin stencil mechanism described in Section 3. For each source expression in the compiled function, the compiler inserts a lightweight stencil that increments a counter in the coverage array. This stencil (Listing 3) contains a single hole (abstracted with the `GETCUSTOM()` macro), which will be patched with the direct address of the integer corresponding to the source reference it belongs to.

```
1 RCP_STENCIL_FUNCTION(_RCP_CUSTOM_COVERAGE) {
2   int *coverage_counter = (int *)GETCUSTOM();
3   *coverage_counter += 1;
4   return _RCP_EXEC_NEXT(...);
5 }
```

Listing 3. Coverage instrumentation stencil.

This stencil is compiled into native code shown in Listing 4. The jump instruction can be removed [23], which makes the whole code consist only of loading an address into a register and incrementing it. With such a compact way to record, the per-expression overhead is minimal compared to the callback-based approach of `covr`.

```
1   movabs _RCP_CUSTOM_DATA(%rip), %rax
2   incl   (%rax)
3   jmp    _RCP_EXEC_NEXT
```

Listing 4. Coverage instrumentation stencil assembly.

To be able to easily visualize the output coverage, we chose to save the coverage information into the same format that `covr` uses. This way, we can have the best of both worlds – efficient recording and the versatile infrastructure of `covr`.

Additionally, we created a fork of the `covr` package, changing the backend of its coverage collection infrastructure to use our own copy-and-patch coverage tooling instead. This makes for efficient coverage recording with no change to the interface for the end user.

An alternative to our approach would be to instead use the `–coverage` C compiler option with `gcov` [12] to inject the native stencils with instrumentation. This would provide more detailed coverage information, which we would then map back to the original source code.

The advantage of this approach is that we can then use `gcov` to get the coverage information for both R and C at the same time. However, it would require significant effort to map the results, and would have a higher performance overhead. If we only care about R coverage, it does not provide any benefit.

4 Profiling and Debugging

In this section we describe how we can leverage the copy-and-patch JIT compiler to add support for performance profiling, and consequently for bytecode-level debugging. R has its own built-in profiler, `Rprof`, which is based on sampling the R stack during AST interpretation (it is implemented inside the AST interpreter). As such, it does not consider native code. Moving the R bytecode execution into native code, we can now use native code profiling tools to get a complete profile of R programs including the code in C/C++ or Fortran. This in turn enables us to use native debuggers to debug R code at the bytecode level.

4.1 The `perf` Profiler

For this purpose, we use `perf` [29], a widely used code profiler on Linux. It provides an abstraction over hardware performance counters to profile dynamic control flow and identify execution hotspots. A common approach is to sample the target program at a given rate to record which functions are executing. To contextualize these samples, `perf` must walk the call stack to map instruction addresses to function names, utilizing one of three unwinding methods: LBR, FP, or DWARF.

- *LBR (Last Branch Record)* uses hardware registers on Intel CPUs (Haswell and newer) to store a ring buffer of recent branch decisions, reconstructing a limited-depth call graph (typically 16 to 32 entries).
- *FP (Frame Pointers)* relies on base pointers saved in the generated code to walk the stack. While it is fast and incurs low runtime overhead, it is highly sensitive to compiler optimizations. Since GCC 4.6 (released in May 2011) [13], the default behavior for optimized builds is to omit frame pointers.[1] Most Linux distributions ship optimized binaries. Even if we had compiled R with frame pointers, calls to shared system libraries would sever the unwinding chain. Furthermore, FP cannot reconstruct function inlining, meaning inlined functions will be missing from the profile.
- *DWARF* uses compiler-emitted debugging information to reconstruct the stack. `perf` achieves this by snapshotting current CPU registers and a segment of the program's stack (typically 8 kB) at run time for post-processing. While DWARF unwinding is computationally slower and produces significantly larger data files than FP, it can largely preserve inlined function hierarchies and correctly unwind through libraries lacking frame pointers.

Because of the visibility limitations imposed by standard compiler optimizations on FP, we decided to support DWARF unwinding. This also allows us to support other system profilers, such as `samply` [34], a user-space profiler that uses real-time DWARF unwinding to achieve high accuracy with a smaller footprint. Furthermore, we had to extend the copy-and-patch compiler to emit exception handling frames

[1] The `-fomit-frame-pointer` flag frees the dedicated frame pointer register (e.g., `rbp` on x86-64) for general-purpose use and eliminates the instruction overhead of establishing stack frames, at the cost of breaking traditional, register-based stack unwinding.

(`.eh_frame`) in the generated code anyway to support unwinding of C++ exceptions. This is to support the case when an R library's C++ code calls back to R, which calls another C++ function that throws an exception. Without `.eh_frame`, the runtime would not be able to unwind past the jitted code, resulting in a crash instead of a caught exception. R package authors are advised to avoid C++ exceptions [21] but we have run into issues with this.

4.2 Supporting DWARF in RCP

Generating DWARF information is usually no small endeavor as the format is rather complex [7, 8]. At its core, DWARF is a bytecode language with a self-describing schema that encodes the precise mapping between machine instructions and source-level constructs such as functions, variables, and types. The difficulty lies in the fact that this bytecode must be emitted correctly for every instruction in the generated code, requiring intimate knowledge of the calling convention, register allocation, and stack layout at each program point. However, since copy-and-patch stencils are compiled by a regular C compiler, we can let it do most of the work by simply compiling the stencils with `-g`. Each stencil is compiled as a normal C function, so the compiler emits standard `.eh_frame` FDEs (Frame Description Entries) in the stencil object files. At build time, the stencil extractor parses the `.eh_frame` section, locates each stencil's FDE and extracts the raw CFI (Canonical Frame Information) instruction bytes, DWARF bytecode that describes how to compute the CFA (Canonical Frame Address) and locate saved registers at each instruction. CFA is a reference address for a stack frame, typically the value of the stack pointer register immediately before a call instruction. These are stored as byte arrays alongside the stencil machine code in the stencil library. This increase in build time is negligible and is a one-time cost. The size increase is small: the 117 stencils take up 68 kB of machine code, an average of 586.8 bytes each. The extracted DWARF CFI data takes up 1.6 kB total, an average of 13.9 bytes per stencil.

At run time, when a function is JIT-compiled, a single `.eh_frame` blob is assembled in memory containing: a CIE (Common Information Entry) setting the default unwinding rules for the JIT code, such as the return address register and the stack pointer register; and one FDE that concatenates the individual stencils' CFI programs into a single unwinding description. The CFI cannot be used verbatim because the stencils are copied into a single contiguous code block, so their relative addresses change. Instead, we need to patch them with the stencil's actual address in the assembled code — effectively replaying each stencil's unwind rules at its relocated position within the single FDE that covers the entire compiled function.

Finally, we need to register the assembled `.eh_frame` with the C++ runtime via the libgcc function `__register_frame()`, which makes it visible to the C++ unwinder. This enables throw/catch to unwind through JIT frames — critical because R's error handling internally uses longjmp-like mechanisms and the JIT must interoperate with C++ exception unwinding for stack cleanup. When a closure is collected by R's garbage collector, we unregister the unwinding information with `__deregister_frame()`.

4.3 Native Profiling

There are two ways to add support for profiling JIT code in Linux with `perf`: the older perfmap [28] and the newer jitdump format [9]. The perfmap is a simple textual format that only provides symbol-level mapping (address to function name). The jitdump is a binary format that additionally supports source-line attribution and unwinding info by embedding the raw `.eh_frame` bytes so that `perf` can unwind call stacks through JIT frames without relying on frame pointers.

We implement the `jitdump` interface in the copy-and-patch runtime. Whenever a function is compiled, the runtime emits a code-load record containing its native address range, symbol name, and the associated DWARF unwinding information. This makes the generated code visible to `perf` as if it were a regular shared object, allowing samples inside JIT-compiled R code to be symbolized and unwound together with the surrounding C frames. In the reports, `perf` materializes these code blobs as synthetic objects such as `jit-81.so`.

Listing 5 illustrates the result on the Mandelbrot benchmark. The report interleaves JIT-compiled R code, shown as symbols from synthetic `jit-*.so` objects, with functions from the GNU R runtime and built-ins. The *Self* column gives the time spent directly in a symbol, while *Children* includes its callees. This unified view is the main benefit of the approach: it becomes possible to see, in a single profile, how much time is spent in compiled R code versus runtime services such as allocation, garbage collection, argument matching, or native built-ins.

Concretely, in this profile, we can see that the majority of time is spent in the `execute` function of the JIT-compiled code, which is expected since it is the main execution loop. We can also see that a significant portion of time is spent in the `R_gc_internal` function, which is responsible for garbage collection.

4.4 Native Debugging

Debugging JIT code is difficult because the code is generated at run time instead of being loaded from object files where debuggers can look for debug information. The GNU project debugger, `gdb`, offers a JIT interface [14] that allows a JIT compiler to register in-memory object files containing DWARF debugging information, enabling source-level debugging of JIT code in `gdb`.

DWARF debug information. Since we already generate DWARF information for stack unwinding and performance profiling, we can also leverage it for debugging. Next to

```
  Children      Self  Object     Symbol
    84.56%    75.53%  jit-81.so  execute
     7.83%     7.57%  R          R_gc_internal
     7.74%     0.00%  R          RunGenCollect (inlined)
     7.68%     1.44%  jit-19.so  base::bitwShiftL
     5.70%     0.68%  rcp.so     Rsh_Call
     4.60%     0.00%  jit-30.so  base::gc
     4.60%     0.00%  R          do_gc
     4.60%     0.00%  R          R_gc (inlined)
     3.94%     2.06%  R          Rf_allocVector3
     3.91%     0.26%  R          make_applyClosure_env
     3.15%     2.90%  R          Rf_findVarInFrame3
     2.82%     0.43%  R          do_bitwise
     2.64%     0.17%  R          R_findVar
     2.47%     0.09%  R          Rf_matchArgs_RC
     2.47%     0.00%  R          R_findVarInFrame (inlined)
     2.39%     1.53%  R          Rf_matchArgs_NR
     2.14%     0.00%  R          bitwiseShiftL (inlined)
     1.80%     0.34%  R          forcePromise
     1.79%     1.37%  R          CONS_NR
     1.62%     0.85%  R          Rf_coerceVector
     1.28%     1.03%  R          rcpEval
     1.20%     0.09%  jit-35.so  base::bitwXor
     1.20%     0.00%  R          coerceToInteger (inlined)
     1.20%     0.09%  jit-97.so  execute_prom_1
     1.18%     0.09%  R          Rf_NewEnvironment
     1.11%     0.26%  R          Rf_eval
```

Listing 5. Example of perf report when running the Mandelbrot benchmark with 16 kB stack sampling at 99 Hz (abridged).

`.eh_frame`, we need to generate the additional DWARF sections required for debugging:

- `.debug_abbrev`, abbreviation table defining the DIE schemas[2] for compile units, subprograms, parameters, and types.
- `.debug_info`, describing the JIT function as a subprogram with two parameters: a pointer to the value stack and a pointer to the locals, plus the code address range. We pin the location of the parameters to `rdi` and `rsi` registers. While this is technically true only at the beginning of each stencil, copy-and-patch ensures that each stencil finishes with the value stack and locals pointers still in the same registers (it does so by emitting a *fake* call that is removed at the patch phase).
- `.debug_line`, a DWARF line number program mapping each stencil's address to a sequential bytecode instruction index. For this to work, we write a pseudo-assembly source file with one line per bytecode instruction, so `gdb`'s list command can display a human-readable representation of the compiled bytecode.

This is packed into an in-memory object file (ELF on Linux) and registered with `gdb` at run time using the JIT interface. With this, we can use `gdb` to set breakpoints and inspect the state of the R programs at the bytecode level.

Usage. R users typically debug at the language level using GNU R's built-in AST debugger, which operates on R expressions and call frames. The `gdb` JIT interface targets a different audience: package authors or developers working at the boundary between R and native code. In a standard GNU R runtime, `gdb` backtraces through interpreted code are opaque — the AST evaluator appears as deep recursive eval chains, while the bytecode interpreter appears as a flat loop of the interpreter call where all instructions execute within the same C frame, making it impossible to determine which R function or operation is actually in progress. With JIT-compiled code, each R function gets its own native symbol and each bytecode instruction maps to a distinct address range with source-line attribution, so `gdb` backtraces show meaningful function names and instruction-level positions interleaved with the C runtime frames. This makes the `gdb` JIT interface particularly valuable for diagnosing issues at the R-native boundary — understanding how R function calls translate into native execution or inspecting stack state when native code is invoked from JIT-compiled R.

An example is shown in Figure 3. The R program calls several functions before reaching the native routine `c_call`, which contains a bug that triggers a segmentation fault. When debugging this execution with the R AST interpreter, `gdb` would show 31 frames, most of them recursive calls to `Rf_eval`. Relating those eval frames back to the original R functions is tedious, especially in the presence of higher-order functions and lazy evaluation. The situation is worse for bytecode-compiled functions, because the bytecode interpreter executes an entire R function within a single C frame. Even invoking the built-in `R_GetTraceback` from `gdb` via the `call` command produces the R stack trace in Figure 3b, which omits the call to `h`—the function where the fault actually arises. By contrast, debugging the same code with `gdb` and the copy-and-patch JIT (Figure 3c) yields a backtrace in which the calls to `f`, `g`, and `h` appear directly, interleaved with the C runtime frames. The backtrace also shows that the fault occurs in `h` while forcing the promise `f_prom_1` created in `f`. This capability has already proved useful when diagnosing issues with RCP.

5 Evaluation

In this section, we evaluate our RCP extension for dynamic program analysis. We focus on two questions: how much implementation effort the extensions require, and whether operating at the native-code level indeed provides low overhead compared with existing source-level approaches.

All measurements were collected on a dedicated four-core Intel Core i7-7700K CPU running at 4.2 GHz, with 32 GB of RAM and Ubuntu Noble with 6.11.0-24 kernel. For the baseline performance experiment, we used GNU R version 4.3.2 compiled with GCC 14.2.0.

5.1 Type Tracing

We compare the copy-and-patch-based instrumentation with one using only R code, with the `injectr` package [25] using

[2]Debugging Information Entry is the fundamental unit of DWARF — each DIE is a tagged node in a tree that describes one program entity (a compilation unit, function, variable, type, etc.). The `.debug_info` section is essentially a tree of DIEs, and `.debug_abbrev` defines their schemas, *i.e.* which tag and attributes each DIE code carries.

```r
h <- function(b) .Call("c_call", b)
g <- function(a) a + 1
f <- function(x, y) g(x(y))
f(h, 42)
```

(a) R code with bug in `c_call`.

```
x(y)
g(x(y))
f(h, 42)
```

(b) R stack trace via `R_GetTraceback`

```
Program received signal SIGSEGV, Segmentation fault.
c_call at call.c:13
13 *value = lhs;
(gdb) bt
#0 c_call at call.c:13
#3 h at h.S:3
#7 f_prom_1 at f_prom_1.S:3
#12 forcePromise at eval.c:955
#13 g at g.S:1
#17 f at f.S:3
#24 Rf_eval at eval.c:1292
#28 Rf_mainloop at main.c:1226
#29 main at Rmain.c:29
```

(c) `gdb` backtrace for RCP-compiled R code (repeating frames abridged).

Figure 3. Debugging a cross-language bug with `gdb` and the copy-and-patch JIT. The R code (top left) triggers a segfault in native code. The R-level stack trace (bottom left) omits the call to `h` where the error originates. The `gdb` backtrace (right) using copy-and-patch-compiled native frames clearly shows `h`, `f_prom_1`, `g`, and `f` interleaved with C runtime frames.

a corpus of 8 popular R packages from CRAN, a curated repository of R packages (see Appendix A). For a package to be included in CRAN, it has to have some executable code, such as tests, examples and vignettes. Examples are executable code in the documentation of a function, and vignettes are longer-form documentation showing how to solve a particular problem. We extract this code from the package with `runr` [26] and run it to gather type information from the type tracing.

Packages usually have a small number of vignettes, but a large number of examples and tests. To speed up the experiments, for each package in the corpus, we concatenate all the code from examples, and all the code from tests, into 2 files. Each of the extracted files is wrapped within an R `local` function to isolate it from the other extracted snippets in the concatenated file, and surrounded by timing code to measure the wall-clock time of executing the code and to save the traced types. This makes it possible to load the traced package and compile it for RCP variants only once, which gave speedups of up to 10–20×. It also gives a fairer comparison with GNU R, which has a bytecode compiler that is more likely to trigger compilation in a long concatenated file than in many quick-running files.

***R-based instrumentation with* injectr.** It is implemented as an R package called `rtypes` that depends on `injectr`. `injectr` rewrites the AST of a function to insert entry hooks and exit hooks. For exit hooks, it inserts an `on.exit` callback, which is an R mechanism to register a function to be called when the function exits, no matter what. We still use C code to check if an argument (a *promise* in R) has been evaluated or not.

This means that, unlike the copy-and-patch instrumentation, the `injectr` instrumentation will be able to collect the types of arguments even if the function does not reach an exit point (`return`) such as with S3 generics,[3] or when an error is thrown. In practice, only 11.5 % of the traced functions have at least one signature mismatch between RCP and `injectr`.

Overhead. In addition to the R-based instrumentation, we also compare the RCP instrumentation with vanilla GNU R (*baseline*) and vanilla RCP, both without any instrumentation.

For a fair comparison, we deactivate tracing of S3 generics for both versions. We also do not include timing data of files for which the set of traced functions differs in the collected traces between copy-and-patch instrumentation and `injectr` instrumentation. This concerns 116 files out of 832 files in total.

We compare both the total wall-clock time and the per-package totals across the four R variants, for 10 runs, and report ratios of medians with bootstrapped 95% intervals. Figure 4 shows that R-based instrumentation adds a lot of overhead compared to vanilla R, with a 1.53x [1.53, 1.54] slowdown. The overhead of the stencil-based instrumentation for RCP is much smaller, with a 1.00x [1.00, 1.00] slowdown compared to RCP without instrumentation. RCP instrumentation is 1.52x [1.52, 1.53] faster than the R-based one.

Note also that RCP without instrumentation is about as fast as vanilla R (1.00x [0.99, 1.00]). This is consistent with results from [23], which were measured on a benchmark with hot paths written in R, whereas our corpus consists of real-world packages whose hot paths often include a lot of native code.

Implementation effort. The lines of code for the copy-and-patch instrumentation, and for the `injectr` instrumentation give an idea. We distinguish between the frameworks

[3] S3 is one of the main R OOP systems. It is based on generic functions and method dispatch using a class tag attached to values.

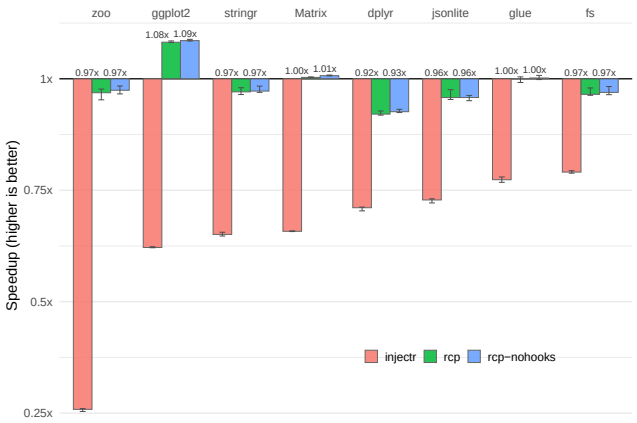

Figure 4. Execution times of the copy-and-patch instrumentation with tracing (`RCP`), without tracing (`RCP-nohooks`) and the R-based instrumentation (`injectr`) relative to vanilla R baseline, for all 10 runs.

and the type tracing use case itself. For RCP, the plugin stencil framework for copy-and-patch is 48 lines of C, and the actual exit hooks and specific compilation logic for type tracing is 351 lines of C and 31 of R. For the R-based approach, the `injectr` framework is 320 lines of R, and 50 of C, and the type tracing with package `rtypes` is 238 lines of R, and 24 of C. The type tracing itself is the same order of magnitude for both approaches, but the instrumentation framework based on copy-and-patch is smaller than `injectr`, as it reuses the copy-and-patch stencil infrastructure.

5.2 Code Coverage

We compare the RCP-based coverage with the standard `covr` package using a corpus of 8 popular R packages from CRAN (see Appendix A). For context, we also include runs without coverage recording. We compare four configurations: (1) vanilla GNU R, which runs the test suite without coverage collection and leaves the tested package uncompiled; (2) `covr`, which uses the standard `covr` callback-based instrumentation; (3) RCP + coverage, which uses our coverage mechanism from Section 3.3 with the tested package compiled by RCP and coverage recording enabled; and (4) RCP, which runs the test suite without coverage collection with the tested package compiled by RCP. Comparing vanilla R with `covr` isolates the overhead of source-level coverage recording, while comparing RCP with RCP + coverage isolates the overhead of our native coverage recording.

To keep the comparison uniform, all measurements use the `covr` interface. We use its built-in timing support, which reports the combined cost of setting up coverage instrumentation and running the test suite. In our approach, this time also includes JIT compilation for RCP and enhancing of the source references.

Implementation effort. Implementing code coverage on top of the instrumentation framework required 131 lines of C code in the JIT compiler and 6 lines in the stencil. The implementation emits coverage data in the same format as `covr`, allowing us to reuse its existing reporting and visualization infrastructure.

Correctness. On average, RCP-based coverage reports 17% more covered lines than `covr`. The two methods instrument at different granularities, which explains the discrepancy. `covr` rewrites the source AST and wraps every sub-expression—including constant literals, bare symbols, and individual **`if`**/**`else`** and **`switch`** branches—in its own counter, whereas RCP derives coverage from the compiled bytecode's source-reference table and records a hit only where an executed instruction carries a source reference. The methods therefore agree on statement-level coverage but diverge on sub-expression leaves. RCP credits reached control-flow positions that `covr` attributes only to a taken branch (for example, evaluated conditions and forced promises), which accounts for the bulk of its additional covered lines. Conversely, because its granularity is bounded by the bytecode source-reference table, RCP cannot mark the small set of lines whose only executed expression is a constant, a bare symbol, a branch result, or a short-circuited operand—lines `covr` counts.

Overhead. Each test suite was run 10 times. We report relative performance as ratios of medians, aggregated across packages by their geometric mean, with bootstrapped 95% confidence intervals.

Figure 5 summarizes the results per package. As the raw run time is not directly comparable across methods (each records a different number of covered lines), we interpret performance as the number of lines covered per second. The figure reports this throughput for `covr` and for RCP-based coverage across the benchmark packages.

Because RCP covers more lines in comparable wall-clock time, its throughput exceeds that of `covr`: RCP-based coverage covers 1.2 [1.01, 1.46] as many lines per second as `covr`. The instrumentation overhead of RCP remains modest. Relative to vanilla R, RCP-based coverage has a slowdown of 1.09 [1.05, 3.01], while RCP without coverage has a slowdown of 1.08 [1.02, 2.82], so the coverage recording itself adds little on top of the JIT.

These numbers may appear weaker than those shown in Figure 6, but the experiments are not directly comparable: the time reported here is whole-test-suite wall-clock and includes RCP's JIT compilation and source reference enhancing, whereas the profiling benchmarks report runtime only.

5.3 Profiling

Finally, we look at the implementation and performance of the native profiling extension to RCP. We give an overview of the implementation effort, the performance of the profiler

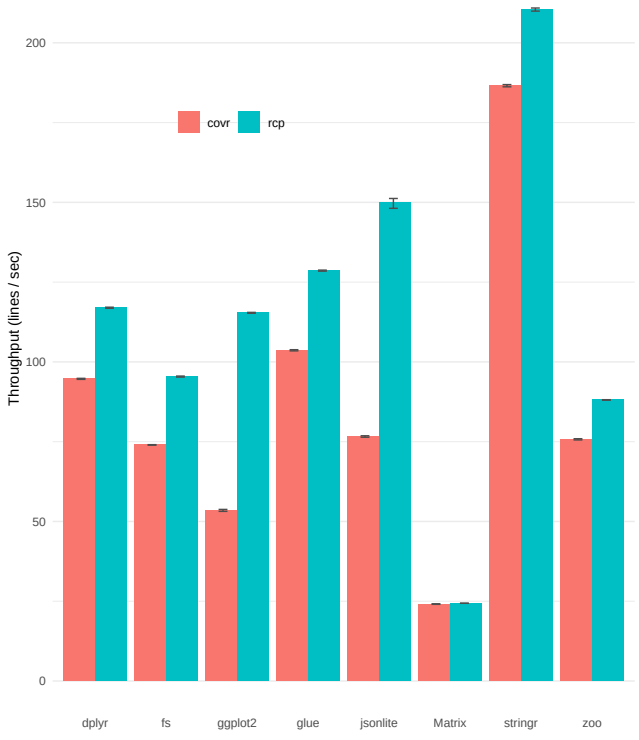

Figure 5. Coverage throughput (source lines covered per second) for covr and RCP with code coverage, per benchmark package (higher is better).

and discuss some of the results that could be obtained with it.

Implementation effort. The implementation of native profiling and debugging support required ~1,864 lines of code. The main portion is in a custom DWARF library (649 lines) that provides the core infrastructure (LEB128 encoding, CIE parsing, CFI instruction decoding and a CFI state machine for computing stack frame layouts). We decided to implement it from scratch as using `llvm-dwarfdump` and the DWARF specification seemed simpler than using an existing library such as `libdwarf`. Assembling the `.eh_frame` section is then done in 115 lines. The rest of the code was split between the build-time DWARF extractor from compiled stencils (299 lines) and the runtime support for GDB (566 lines) and perf (235 lines).

Overhead. Next we look at the overhead of the copy-and-patch with native profiling support. For this we used benchmarks from the *Shootout* suite and *AreWeFastYet* suite [31] translated to R (by Kalibera *et al.* [22]). Unlike the package workloads used in Sections 5.1 and 5.2, profiling overhead is best evaluated on longer-running, deterministic benchmarks, where the cost of periodic sampling is measurable and not dominated by package loading, I/O, or short-running tests.

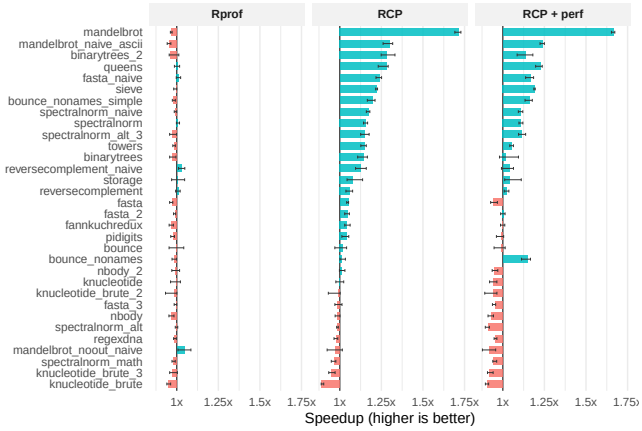

Figure 6. Execution time of Rprof, RCP and RCP with `perf` profiling relative to GNU R baseline.

Together, the suites consists of 32 benchmarks with 1,907 lines of R code and runs for a total of 51.5 seconds in GNU R.

We compare four configurations: GNU R, RCP without profiling, RCP with `perf` profiling (16 kB stack snapshots at 999 Hz), and GNU R with `Rprof`. The last configuration serves as a comparison against R's built-in profiler, which samples only the R stack and, at best, every 10 ms. We ran each benchmark 15 times and report relative slowdown as ratios of medians with bootstrapped 95% confidence intervals. Figure 6 summarizes the results. As expected, both `Rprof` and RCP with profiling introduce overhead, but the overhead is small in practice. Since RCP itself runs the benchmarks about 1.09 faster than GNU R, RCP with `perf` remains, on average, slightly faster (1.04) than GNU R.

Results. The native profiling provides a unified view of the execution, interleaving R and native code. This helps us to identify where time is typically spent. Figure 7 shows the breakdown of the execution time in the 32 benchmarks across 6 categories[4]. We use a classification similar to Morandat *et al.* [32]. A significant portion of time is spent in R builtins (38.7%). The memory management (24.3%) can be divided into the time spent in the garbage collector (12.8%) and the time spent in allocation (11.5%). About 17.5% is spent in the JIT-compiled code. R has a complicated calling convention (among others, named arguments use partial matching and can be passed in any order) which contributes to 9.9% overhead. The system category (7.5%) includes time spent in shared libraries such as libc, regular expressions, or blas/lapack for linear algebra. There are some outliers; for example, `spectralnorm_math` spends most of its time in R builtins.

[4]We have manually classified 827 that appeared at the top of the call stack while running the benchmarks with `perf` at 999 Hz and 16 kB stack sampling.

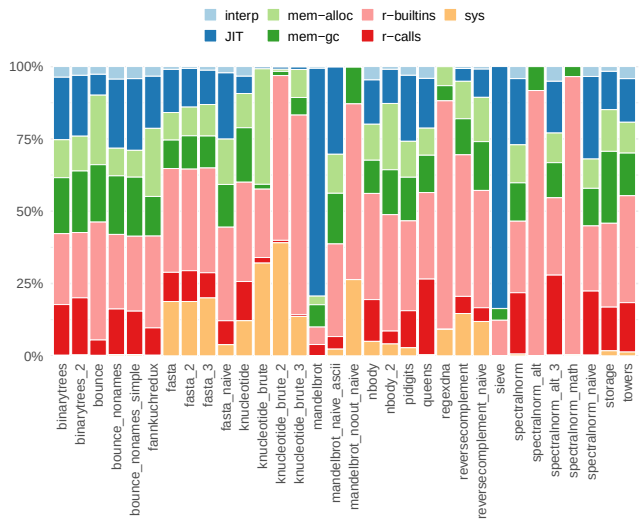

Figure 7. Breakdown of benchmark runtimes by category.

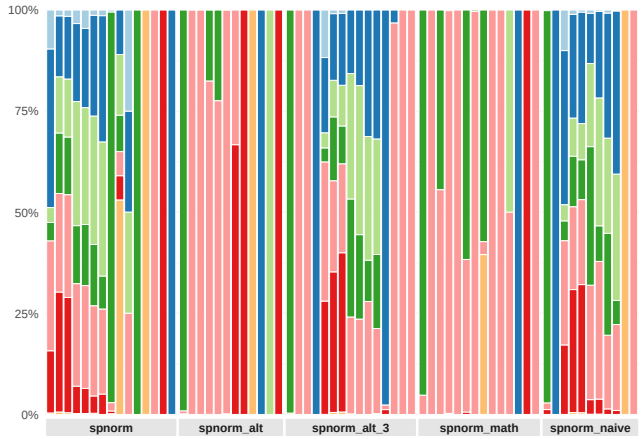

Figure 8. Breakdown of compiled functions by category.

The main advantage of the implemented profiling support is that it allows us to perform a similar classification per compiled function, as shown in Figure 8. We can therefore zoom in on the JIT-compiled code and see the breakdown on a finer, per-function level. For example, for `spectralnorm_alt`, we can see that it contains 12 compiled functions, out of which only one spends significant time in the JIT code, while the others spend most of their time in R builtins or memory management. This kind of information is not available with R's built-in profiler, which only samples the R stack and cannot see into native code.

6 Related Work

Dynamic analysis tools for R. R provides several built-in facilities for dynamic analysis. The `Rprof` profiler samples the R call stack at regular intervals and `profvis` [5] provides interactive visualizations of the results. For debugging, R offers `debug`, `trace`, and `browser`, all of which operate within the AST interpreter. Code coverage is provided by the `covr` [19] package, which instruments R code by modifying parsed expressions to insert trace calls. For instrumentation and tracing, `instrumentr` [16] provides a callback-based framework, and `genthat` [27] traces function calls to automatically generate unit tests. All of these tools operate at the R level, through the AST interpreter, and therefore cannot observe native code executed by built-in functions or C/Fortran extensions. R-dyntrace [17] is a modified R interpreter with manually inserted hooks for various events such as function entry and exit, promise forcing, or GC allocations and deallocations. `traceR` [24] is a modified R interpreter with inserted performance counters. Both require running a modified R interpreter and constantly updating the tracing logic as the R interpreter evolves; as such, both have been unmaintained for 5 and 9 years, respectively.

Cross-language profiling and debugging. The difficulty of profiling across the boundary between a dynamic language and its native extensions is well recognized. Scalene [2] addresses this for Python by separately attributing CPU time to Python code and native C code, using a combination of sampling and bytecode-level accounting. PieProf [35] uses hardware performance and debug registers to identify redundant memory accesses; it uses `libunwind` and hooks into the Python interpreter to map native calls to Python calls. Feature-specific profiling [1] takes a different approach by attributing costs to individual language features rather than functions. PolyDebug [20] leverages the Debug Adapter Protocol to coordinate language-specific debuggers. PolyDebug has a large overhead when crossing language boundaries, which happens often in R because the interpreter and performance-critical code are written in C. Our approach sidesteps these difficulties: because the copy-and-patch JIT compiles R bytecode to native code, a single native profiler such as `perf` can profile both R and C code in a unified view.

JIT-based instrumentation. The GraalVM Truffle framework [39] provides a rich instrumentation API that allows building debuggers, profilers, and other tools for any Truffle-hosted language. While powerful, it requires adopting the entire Truffle framework and rewriting the language implementation. The Ř JIT compiler for R [10, 11] generates LLVM IR and can thus leverage LLVM-based debugging infrastructure, but it is a substantially more complex system. In the WebAssembly space, Titzer et al. [37] instrument Wasm programs by rewriting modules at the bytecode level to insert probes before they are JIT-compiled by the Wizard research

engine. Whamm [15] extends this with a declarative DSL that compiles instrumentation rules into Wasm bytecode rewrites, also targeting Wizard's JIT compiler. Like our work, these approaches instrument before or during compilation rather than on emitted machine code, and benefit from bytecode-level semantic information. However, they target a portable, typed bytecode with well-defined instrumentation points, whereas R bytecode is untyped and tightly coupled to the C runtime. Our approach is simpler: it piggybacks on the existing copy-and-patch compiler, requiring only the addition of new stencils for each analysis, with no separate rewriting pass.

Binary instrumentation. Binary instrumentation frameworks such as Pin [30] and DynamoRIO [4] are general-purpose tools capable of instrumenting any native code, including JIT-compiled code. However, they are language-unaware: they operate at the machine code level and have no knowledge of the source language's semantics, making it difficult to map results back to the original program. They also incur significant overhead, typically 2–5× slowdown. By contrast, our approach is language-aware and lightweight, since the instrumentation is inserted at copy-and-patch compile time directly alongside the compiled bytecode.

7 Conclusion

In this paper, we explored copy-and-patch compilation as a practical foundation for dynamic analysis, using R as a concrete target. We implemented four analyses on top of an existing copy-and-patch JIT: code instrumentation, with type tracing and code coverage, performance profiling, and native debugging. The key property is that copy-and-patch produces native code through a standard C toolchain. As a result, the system can reuse native metadata such as DWARF, while the bytecode stream can be edited before code generation to inject analysis hooks in a direct and predictable way. This allowed us to implement native profiling with `perf` and source-level debugging with `gdb` with relatively little effort.

The other analyses exploit the bytecode-level structure of copy-and-patch. Inserting instrumentation and coverage probes reduces to injecting new stencils into the bytecode stream at copy-and-patch compile time, which keeps the implementation self-contained and lightweight. The evaluation confirms that this design yields low overhead: instrumentation adds no measurable cost compared to uninstrumented RCP; code coverage is roughly 2× faster than the established `covr` package; and profiling with `perf` at 999 Hz leaves RCP faster than GNU R on average.

Acknowledgments

This research was supported by the Czech Ministry of Education, Youth and Sports under program ERC-CZ (no. LL2325) and the Czech Science Foundation (no. 23-07580X).

References

[1] Leif Andersen, Vincent St-Amour, Jan Vitek, and Matthias Felleisen. 2019. Feature-Specific Profiling. *ACM Trans. Program. Lang. Syst.* 41, 1 (2019). doi:10.1145/3275519

[2] Emery D. Berger, Sam Stern, and Juan Altmayer Pizzorno. 2023. Triangulating Python Performance Issues with SCALENE. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[3] Felix Berlakovich and Stefan Brunthaler. 2024. Cross Module Quickening - The Curious Case of C Extensions. In *European Conference on Object-Oriented Programming (ECOOP)*. doi:10.4230/LIPIcs.ECOOP.2024.6

[4] Derek Bruening, Qin Zhao, and Saman Amarasinghe. 2012. Transparent Dynamic Instrumentation. In *Conference on Virtual Execution Environments (VEE)*. doi:10.1145/2151024.2151043

[5] Winston Chang and Javier Luraschi. 2024. profvis: Interactive Visualizations for Profiling R Code. https://cran.r-project.org/package=profvis.

[6] Maxime Chevalier-Boisvert, Takashi Kokubun, Noah Gibbs, Si Xing (Alan) Wu, Aaron Patterson, and Jemma Issroff. 2023. Evaluating YJIT's Performance in a Production Context: A Pragmatic Approach. In *International Conference on Managed Programming Languages and Runtimes (MPLR)*. doi:10.1145/3617651.3622982

[7] DWARF Standards Committee. 2010. *DWARF Debugging Information Format, Version 4.* Technical Report. Free Standards Group.

[8] Michael J. Eager. 2012. Introduction to the DWARF Debugging Standard. https://dwarfstd.org/doc/Debugging%20using%20DWARF-2012.pdf.

[9] Stephane Eranian. 2016. perf JIT Dump Specification, Version 2. https://github.com/torvalds/linux/blob/master/tools/perf/Documentation/jitdump-specification.txt.

[10] 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

[11] Olivier Flückiger, Guido Chari, Jan Ječmen, Ming-Ho Yee, Jakob Hain, and Jan Vitek. 2019. R melts brains: an IR for first-class environments and lazy effectful arguments. In *International Symposium on Dynamic Languages (DLS)*. doi:10.1145/3359619.3359744

[12] Free Software Foundation. 2026. *gcov — a Test Coverage Program.* GNU Project. https://gcc.gnu.org/onlinedocs/gcc/Gcov.html Part of the GNU Compiler Collection (GCC)..

[13] GCC Developers. 2021. Consider not omitting frame pointers by default on targets with many registers (Bug#100811). https://gcc.gnu.org/bugzilla/show_bug.cgi?id=100811.

[14] GDB Developers. [n. d.]. JIT Compilation Interface. https://sourceware.org/gdb/current/onlinedocs/gdb.html/JIT-Interface.html.

[15] Elizabeth Gilbert, Matthew Schneider, Zixi An, Suhas Thalanki, Wavid Bowman, Alexander Y Bai, Ben L Titzer, and Heather Miller. 2025. Debugging webassembly? put some whamm on it! *Proceedings of the ACM on Programming Languages* 9, OOPSLA2 (2025), 2058–2086.

[16] Aviral Goel. 2022. instrumentr: Instrumentation Framework for R. https://github.com/PRL-PRG/instrumentr.

[17] Aviral Goel. 2022. r-dyntrace: Dynamic Tracing for R. https://github.com/PRL-PRG/R-dyntrace/tree/r-4.1.0.

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

[19] Jim Hester. 2024. covr: Test Coverage for Packages. https://cran.r-project.org/package=covr.

[20] Philémon Houdaille, Djamel Eddine Khelladi, Benoit Combemale, Gunter Mussbacher, and Tijs van der Storm. 2025. PolyDebug: A Framework for Polyglot Debugging. *The Art, Science, and Engineering of Programming* 10, 1 (2025), 13–1.

[21] Tomáš Kalibera. 2019. Use of C++ in Packages. https://blog.r-project.org/2019/03/28/use-of-c-in-packages/.

[22] 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

[23] Matěj Kocourek, Filip Křikava, and Jan Vitek. 2025. Copy-and-Patch Just-in-Time Compiler for R. In *International Workshop on Virtual Machines and Intermediate Languages (VMIL)*. doi:10.1145/3759548.3763370

[24] Helena Kotthaus, Ingo Korb, Michel Lang, Bernd Bischl, Jörg Rahnenführer, and Peter Marwedel. 2015. Runtime and memory consumption analyses for machine learning R programs. *Journal of Statistical Computation and Simulation* 85, 1 (2015), 14–29. doi:10.1080/00949655.2014.925192

[25] Filip Křikava, Aviral Goel, and Pierre Donat-Bouillud. 2026. injectr - R package to inject code to existing functions. https://github.com/PRL-PRG/injectr.

[26] Filip Křikava, Aviral Goel, and Pierre Donat-Bouillud. 2026. runr - R package for running R code artifacts with various preprocessing. https://github.com/PRL-PRG/runr.

[27] Filip Křikava and Jan Vitek. 2018. Tests from traces: automated unit test extraction for R. In *International Symposium on Software Testing and Analysis (ISSTA)*. doi:10.1145/3213846.3213863

[28] Linux Kernel Developers. [n. d.]. perf JIT Interface. https://github.com/torvalds/linux/blob/master/tools/perf/Documentation/jit-interface.txt.

[29] Linux Kernel Developers. 2025. perf: Linux profiling with performance counters. https://perf.wiki.kernel.org/.

[30] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoffrey Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Conference on Programming Language Design and Implementation (PLDI)*. doi:10.1145/1065010.1065034

[31] 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

[32] 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

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

[34] Markus Stange. [n. d.]. samply: Command-line sampling profiler for macOS and Linux. https://github.com/mstange/samply.

[35] Jialiang Tan, Yu Chen, Zhenming Liu, Bin Ren, Shuaiwen Leon Song, Xipeng Shen, and Xu Liu. 2021. Toward efficient interactions between Python and native libraries. In *European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2021)*. Association for Computing Machinery, 1117–1128. doi:10.1145/3468264.3468541

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

[37] Ben L Titzer, Elizabeth Gilbert, Bradley Wei Jie Teo, Yash Anand, Kazuyuki Takayama, and Heather Miller. 2024. Flexible non-intrusive dynamic instrumentation for webassembly. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. 398–415.

[38] Alexi Turcotte, Aviral Goel, Filip Křikava, and Jan Vitek. 2020. Designing Types for R, Empirically. *Proc. ACM Program. Lang.* 4, OOPSLA (2020). doi:10.1145/3428249

[39] Michael Van De Vanter, Chris Seaton, Michael Haupt, Christian Humer, and Thomas Würthinger. 2018. Fast, Flexible, Polyglot Instrumentation Support for Debuggers and other Tools. *The Art, Science, and Engineering of Programming* 2, 3 (2018). doi:10.22152/programming-journal.org/2018/2/14

[40] Alex Villazón, Haiyang Sun, Andrea Rosà, Eduardo Rosales, Daniele Bonetta, Isabella Defilippis, Sergio Oporto, and Walter Binder. 2019. Automated Large-Scale Multi-Language Dynamic Program Analysis in the Wild (Tool Insights Paper). In *European Conference on Object-Oriented Programming (ECOOP)*. doi:10.4230/LIPIcs.ECOOP.2019.20

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

A Corpus

Table 1. Corpus for the overhead analysis: extracted R files by package and category

Package	Description	Vignettes	Examples	Tests	Total
`ggplot2`	Data visualization	5	159	128	292
`dplyr`	Data manipulation	10	80	86	176
`Matrix`	Sparse and dense matrices	5	117	19	141
`stringr`	String manipulation	4	36	31	71
`fs`	File system operations	1	25	23	49
`jsonlite`	JSON parsing and generation	1	11	35	47
`zoo`	Time series data structures	5	30	6	41
`glue`	String interpolation	4	8	10	22