What Do R Benchmarks Measure

Toward a Performance-Characterized Benchmark Suite for R

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

Filip Říha
Czech Technical University
Prague, Czech Republic
filip.riha@fit.cvut.cz

Oliver Tušla
Czech Technical University
Prague, Czech Republic
tuslaoli@fit.cvut.cz

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

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

Jan Vitek
Czech Technical University
Prague, Czech Republic
jan.vitek@fit.cvut.cz

Abstract

Many programming language implementations have a benchmark suite that consists of a fixed set of programs with a harness and clear maintenance. R has none. Instead, R implementations are evaluated on ad hoc sets of programs and harnesses with different timing conventions. This paper is a step toward a benchmarking suite for R. We collect 118 R programs from 8 repositories and put them under a single harness. Next, we characterize their *performance profiles*, which show the share of the run spent in different regions of the R virtual machine (*e.g.*, bytecode interpreter, memory management, builtin functions). This allows us to classify benchmarks by what they exercise so we can create suites that target a specific use such as compiler optimizations. On three R compilers, the speedups follow the profiles. They concentrate in the interpreter-bound programs and vanish in the ones dominated by native code. The profiles also cut the cost of running a suite. Dropping the 53 programs dominated by native code or builtins leaves 65 programs and saves 51.7% of the running time of the corpus.

CCS Concepts: • **Software and its engineering** → **Software performance**; **Just-in-time compilers**; **Interpreters**; **Dynamic analysis**; • **General and reference** → **Measurement**; **Performance**; *Empirical studies.*

Keywords: benchmarking, performance profiling, R, virtual machines, dynamic languages, benchmark suites, workload characterization

ACM Reference Format:
Matěj Kocourek, Filip Říha, Oliver Tušla, Pierre Donat-Bouillud, Filip Křikava, and Jan Vitek. 2026. What Do R Benchmarks Measure: Toward a Performance-Characterized Benchmark Suite for R . In *Proceedings of the 18th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages (VMIL '26), October 4–9, 2026, Oakland, CA, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3840562.3844972



This work is licensed under a Creative Commons Attribution 4.0 International License.

VMIL '26, Oakland, CA, USA
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2932-4/2026/10
https://doi.org/10.1145/3840562.3844972

1 Introduction

Every performance claim about a language implementation is a claim about a set of programs. Many communities therefore maintain that set deliberately, as a shared benchmarking suite. On the JVM, implementers share DaCapo [2], its industrial counterpart SPECjvm2008 [25] and Renaissance [18], which adds concurrent and functional workloads. JavaScript engine implementers share Speedometer [23] and JetStream [7], revised under a consensus model by the teams behind all three major engines. CPython maintains `pyperformance` [19]. Similarly, Ruby has `ruby-bench` [22]. There are also cross-language benchmark suites such as *Are We Fast Yet?* [13] or the *Computer Language Benchmarks Game* [6] (formerly the *Language Shootout*). These suites differ in scope, but they share three properties: the set of programs is fixed and maintained, every program runs under one harness, and the suite was constructed with a specific purpose in mind.

R has no such suite. What it has is a set of scattered repositories, each with its own driver, timing convention and experimental setup. There are Radford Neal's `pqR` speed tests [17], the ATT `R-benchmark-25` [27], ports of the Computer Language Benchmarks Game [6] and of the vector benchmarks of Riposte [26]. The nearest thing to a common collection is the R Benchmark Suite repository (*RBS*) [20], which repackages some of these but has been left unmaintained since 2016. Every project that works on R performance draws its own subset from this scatter, from the early evaluations of the interpreter [9, 14] to FastR [24], Riposte [26], the Ř JIT [3] and the RCP compiler [10].

This paper is a step toward a curated benchmarking suite for R. First, we collect the existing programs and put them under a common framework, so that every program runs the same way in the same controlled environment. Next, we

characterize the programs by what they exercise in the R virtual machine. The premise is simple. A program that spends most of its time in a linear algebra kernel cannot demonstrate anything about an interpreter, however carefully it is timed. A tight scalar loop cannot claim anything about the performance of that kernel. Both can be good benchmarks. What matters is knowing which question each one answers.

Concretely, we focus on just-in-time compilation. The programs of interest are those where the R interpreter itself is the bottleneck, *i.e.*, where time goes into bytecode dispatch, environment lookup and function calls rather than into a linear algebra kernel or a regular expression engine. To tell the two apart we compute a program's *performance profile*. We sample the running interpreter with `perf` and attribute each sample to one of 11 handcrafted categories, each of which names a region of the virtual machine. A sample is attributed by matching the symbols on its call stack against a set of rules.

We test the profiles against three R compilers. Speedups track the share of the run a compiler can reach, and they vanish once a program is dominated by native code. This turns the profiles into a selection criterion, applied in two stages. Dropping first every program that spends more than 10% of its run in native code, which no R compiler can reach, and then every program that spends more than 80% in builtins, which a compiler can reach only through the dispatch to them, leaves 65 of the 118 programs and cuts 51.7% of the running time of the corpus, from 36.3 min to 17.5 min (in the case of the Ř JIT). A suite can then start small and grow, one profile at a time, as the regions a compiler targets change. This paper makes the following contributions:

1. A corpus of 118 R programs from 8 sources, unified under a single harness so that every program runs identically.
2. A method for computing performance profiles using vanilla R.
3. The performance profiles of the whole corpus with an evaluation on three R compilers.

Availability. The corpus, the harness, the profiling scripts and the analysis notebooks are available at https://github.com/PRL-PRG/r-benchmarks.

2 The Corpus

We collected 8 suites.[1] Two of them, `shootout` and `areWeFast`, are ports of cross-language suites. The rest are R specific. Some suites contain multiple variants of the same program written in a different style (*e.g.*, with explicit loops or vectorized operations).

- `shootout` [6] (37 benchmarks): Programs from the well-known Computer Language Benchmarks Game, taken from *RBS*, and covering eleven algorithms (*e.g.*, binary-trees, fannkuch-redux, fasta, mandelbrot, n-body). Most of the original programs use explicit loops, so the suite also ships more idiomatic variants that rely on vectorized operations and builtin functions.
- `SpeedTest` [17] (25): Programs from Radford Neal's `pqR` speed tests. Small statistical and numerical applications and short algorithmic programs, each mixing R-level loops with vector and matrix primitives (*e.g.*, Cholesky decomposition, Gaussian-process hyperparameter search, Hamiltonian Monte Carlo, kernel PCA, Q-learning).
- `riposte` [26] (16): The vector benchmarks of the Riposte project. It contains data-parallel R written without loops (*e.g.*, Black-Scholes option pricing, k-means, logistic regression, z-score outlier detection). Some of them pair hand-written vector code with an equivalent version that calls a builtin or a package. One program of the suite, `qr`, is not in our corpus: it depends on Riposte's own vector semantics and does not run under GNU R.
- `R-benchmark-25` [27] (15): The ATT `R-benchmark-25`, in three groups of five kernels: matrix calculations, matrix functions that go through the linear algebra library, and small programs written with explicit loops.
- `mathkernel` [27] (11): Two math kernels, vector addition and matrix multiplication, each shipped in a loop-based and a builtin form so that the pair isolates the interpreter from the operation.
- `RealThing` [4] (6): Three representative applications, some of them in more than one variant. Flexclust implements a k-centroids clustering algorithm from the flexclust package [12]. Convolution performs vector updates through nested loops. Volcano conducts a ray-casting simulation across terrain using the builtin volcano elevation dataset, based on code by Morgan-Wall [15].
- `areWeFast` [9] (5): Three programs translated to R from the established *Are We Fast Yet?* suite [13]: Bounce (a bouncing balls physics simulation), Mandelbrot, and Storage (a tree creation program). Bounce comes in three different variants.
- `misc` [27] (3): Ross Ihaka's 2D random walk in three forms.

Methodology. Every benchmark is one R file exporting one function, `execute(size)`, called once per measured iteration. Two optional functions handle the cases that otherwise force per-benchmark drivers. The `setup(size)` function loads any libraries and prepares inputs before the measured region. The `doctor()` function reports any missing dependency. 38 benchmarks define the former and 9 the latter. The `size` parameter controls the cost, *i.e.* the execution time of a single benchmark iteration. Only the `execute` function runs inside the timed region, so it must not load data, generate input, or import any package. It returns a work checksum so different implementations can be compared for correctness.

[1] The programs are redistributed under their original licences: MIT for `areWeFast`, GPL-2 for `SpeedTest` and `RealThing`, and BSD-3-Clause for the five suites taken from *RBS*.

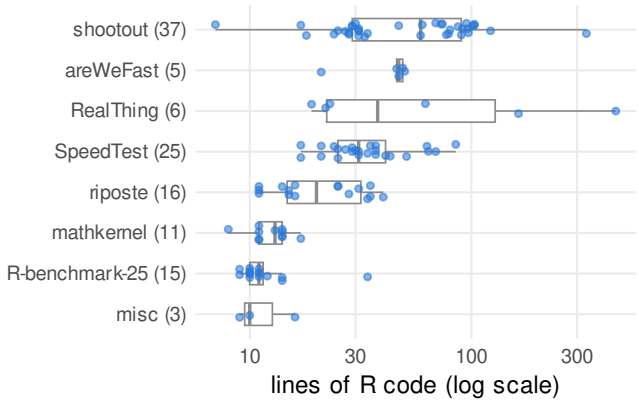


Figure 1. Size of each benchmark as source lines of code in the `execute()` function.

The default benchmark size is chosen so that one iteration runs in about one second on vanilla R. The median runtime is 0.97 s, excluding setup and interpreter startup, and the elapsed wall times range from 0.14 s to 8.81 s.

Benchmarks run under a common harness. It sources the benchmark in its own directory, checks `doctor()`, calls `setup(size)` once outside the measured region, and then repeats `execute(size)`. Every iteration is preceded by a full garbage collection, so the garbage one iteration leaves behind is not attributed to the next. That collection is timed on its own, as are the collections R runs inside `execute()`. Each iteration measures both CPU and wall time, which is how we spot an accidental use of a multithreaded library. We also control OpenBLAS threading through environment variables.

The harness itself runs under `bench` [21], a framework for repeatable performance experiments, which keeps the experimental setup under control by reducing the noise on the host machine.

Result. The corpus contains 5K lines of R across 118 programs. They are mostly micro-benchmarks: the median benchmark is 28.5 lines and 76.3% of them are under fifty (Figure 1). Only the `RealThing` suite contains real application code.

Environment. All experiments ran on a dedicated machine with a four-core i7-7700K CPU at 4.2 GHz (no simultaneous multithreading) and 32 GB of RAM, running Ubuntu 24.04 with the 6.17.0-23 kernel and `perf` 6.17. We used GNU R version 4.5.2 built from source with GCC 14.2.0, linking OpenBLAS 0.3.26 for both BLAS and LAPACK.

The machine is quiesced for measurement: the frequency governor is pinned to `performance`, turbo is disabled, address-space randomization and transparent huge pages are off, and swappiness is zero. OpenBLAS is pinned to single thread via `OPENBLAS_NUM_THREADS=1`. Benchmarks run sequentially, one benchmark per process, and one process at a time. R is single-threaded and no multi-threaded system libraries are used.

3 Benchmark Performance Profiles

Once we have a corpus, we can ask what each benchmark exercises. The question is how much of a run goes into which part of the virtual machine. We call this the benchmark's *performance profile*.

3.1 Categories

Decomposing an R run is not a new idea. Morandat *et al.* built such a decomposition to evaluate the design of R [14]. Inspired by that we define the following 11 categories:

Category	Description
`interpreter`	executing R code, bytecode or AST
`call`	calling R closure: matching actuals to formals, building an environment, creating promises
`dispatch`	R object system dispatch (S3 and S4)
`lookup`	symbol resolution
`primitives`	the native code of R builtins and specials
`mem-alloc`	any memory allocation
`mem-duplicate`	copy-on-write copying
`mem-gc`	garbage collection
`native-ffi`	the native code in R packages
`native-blas`	the code of the BLAS and LAPACK libraries
`other`	all other work

The grouping is meant to be actionable. Our focus is on the parts relevant to JIT compilation. `interpreter`, `call`, `dispatch` and `lookup` are the interpreter's own machinery, and they are what a compiler targets. `primitives` is harder to act on, because part of that work the bytecode interpreter already does itself. Adding two scalar integers happens in the eval loop, but `+` in general has complex semantics and may even dispatch to arbitrary user code, so it often remains as a call to a builtin. How much a compiler can reclaim differs from case to case. Memory allocation, garbage collection and the two native categories are largely out of a compiler's reach, but they are not out of reach in general. They are what one would look at to evaluate a different collector or a different linear algebra library. Another interesting category is `mem-duplicate`: a smart compiler could for example prevent useless copying and do operations in place.

A profile is one share per category, the fraction of the run attributed to it. Shares sum to one, so profiles of benchmarks of different lengths are comparable. Each share is also a bound. A benchmark that spends a fifth of its run in the bytecode loop cannot remove more than a fifth of its runtime through a change to that loop, however good the change.

3.2 Methodology

Profiling requires knowing which region of the interpreter is executing at a given instant. We obtain that with `perf`,

the Linux sampling profiler. It interrupts the process at a fixed rate and records a call stack. The advantage is that it needs no change to the interpreter's source. The disadvantage is that sampling introduces run-to-run variance, yielding slightly different profiles across executions. Furthermore, the sampling frequency and the unwinding method decide the quality of the recorded call frames.

Profiling with perf. Frame pointers unwind more precisely than DWARF when they are present. Ubuntu 24.04 LTS default `CFLAGS` include `-fno-omit-frame-pointer`, so 100.0% of stack samples reach `main`, yet that alone is not enough. At `-O2` functions are inlined into their callers. For example `Rf_applyClosure`, the frame that says an R closure was called, appears 79 times per thousand samples against 2533 on a build with inlining disabled (in both C and Fortran). That build costs a median 1.35 slowdown over our benchmarks.

An alternative to frame pointers is DWARF-based unwinding which copies a slice of user stack per sample and recovers the stack offline from debug data. Because a DWARF unwind lands inside an inlined body rather than at a return address, `perf` can also expand inlined callees from the line table. For example the `Rf_applyClosure` frame appears 2525 per thousand samples. We use DWARF with 32k stack samples at which 100.0% reach `main`. This costs on average 60 MB per recording totalling of 34 GB for the corpus. Holding the binary fixed and varying nothing but `--call-graph`, DWARF costs 1.1% of the profiled interval against frame pointers' 0.9%: the stack copy is a memcpy into the ring buffer, and the unwinding happens offline.

A share is a proportion estimated from a finite number of samples, so sampling faster makes it steadier: over three repetitions the least stable category share of a benchmark moves 1.44% at 999 Hz and 1.14% at 1,999 Hz. The cost is linear in the rate, in runtime and in bytes alike — 0.9% and 22 MB per recording at 999 Hz against 1.4% and 45 MB at 1,999 Hz — so what bounds the rate is disk rather than the measurement. We record at 1,999 Hz because that is where the samples become as steady as in the frame pointer case with disabled inlining.

Harness. Recordings are driven by `bench` [21], a benchmarking framework. It prepares the inputs, forces a full collection, and calls `execute()` exactly once, with no iteration loop, so the recorded process is self-contained. Each phase brackets itself with a pair of timestamps taken from the system monotonic clock, the same clock `perf` uses for its samples. Without that cut, a profile would describe interpreter startup, library loading and input construction as much as the program itself (the median share of `execute()` in a recording is 86.3%). Marking the phases from inside the process is simpler than driving `perf`'s enable and disable through a control pipe. The recording is therefore unconditional and the window is applied offline. Each benchmark is recorded 5 times, one process per recording.

3.3 Mapping

Sampling stacks are converted into profile categories. We unwind the stack until we find a frame that matches a rule. That is because the function that is actually executing often does not name a region. Also, 46.2% of samples have a leaf symbol that occurs in more than one category, and the worst offender occurs in 9 of the 11. These are the shared accessors, whose category is a property of their caller rather than of themselves. The unwinding should nevertheless stay short, and we show below that it does.

Rules. Three kinds of rule classify a frame, applied in this order of precedence.

1. 2 *shared object rules* identify code by the dynamic shared object it lies in rather than by its name, because some code cannot be named (*e.g.* the symbols of a package's shared library). A frame in a BLAS or LAPACK object belongs to `native-blas`, and a frame in a package's own object is foreign code, `native-ffi`. These rules come first, which is what separates a linear algebra kernel from the builtin that entered it.
2. 52 handcrafted *symbol rules* name one C function each, one per region boundary, read off the interpreter's source. Some of them exist because the bytecode loop carries its own fast paths for arithmetic, subscripting, variable lookup and closure calls, which never reach the functions the interpreted path goes through. Each fast path is mapped to the category its interpreted counterpart resolves to, so that an operation is charged the same way whichever path executed it.
3. 2 *shape rules* match a naming convention instead of a list of names. R's primitives are hundreds of C functions that follow one convention. So does the layer that materializes compact vectors, whose work belongs to the primitive that asked for the elements.

For example, the following shows the top of the stack of a sample from the `attr3_1` benchmark:

```
1 __ieee754_pow_fma (libm.so)
2 R_pow (R)
3 R_POW (R)
4 real_binary (R)
5 R_binary (R)              ← primitives
```

The sample fell inside the C library's exponentiation routine. Neither that routine nor the three frames of R's own arithmetic path that called it carries a rule, so we unwind to the first frame that does, `R_binary`, which is a primitive.

3.4 Result

We recorded 590 profiles, 5 runs of each of the 118 benchmarks. They hold 66.5M stack frames, which reduce to 2.2M

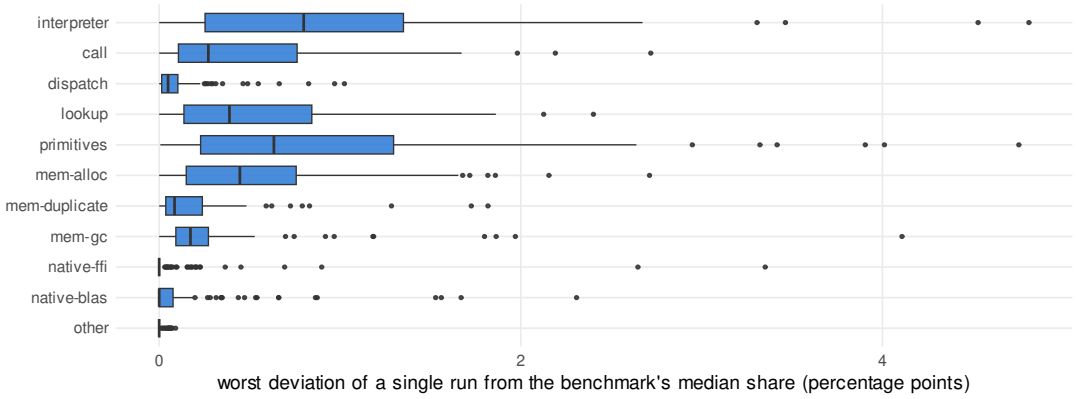


Figure 2. Performance profile stability across the 5 runs. Each box is one category over every benchmark. The value is the worst of the runs, *i.e.*, how far the share read from the unluckiest run lies from that benchmark's median share.

samples, 1.8M of them inside `execute()`. A run yields a median of 2,510 samples and never fewer than 1,027. The fewest any single run has inside `execute()` is 281.

Unwinding distance. The unwinding is short. It stops at the frame that was executing for 33.7% of samples and one or two frames out for a further 50.5%, which leaves 15.8% crossing three frames or more. How far it travels is also the diagnostic for what the rules do not cover, since the long chains are precisely the sequences of frames that no rule names.

Unattributed frames. A rule matches almost every sample. Across the corpus, 0.02% of the samples inside `execute()` carry no category at all. What is left is not a gap in the taxonomy. The dominant cause is an unresolved stack (37.2%). The unwinding reached frames the debug information cannot name, which no additional rule could recover.

Profile variations. Five runs of a benchmark give five profiles; the question is how stable they are. Read from the least representative of the five, a category share is off from that benchmark's median by 0.1% and by 1.2% at the ninetieth percentile. Figure 2 shows the distribution category by category. The worst single pair in the corpus, one of 118 by 11, moves 4.8%. Taking a profile whole rather than category by category, two runs of one benchmark are a median 1.1% apart in total variation distance and 3.2% at the ninetieth percentile.

Performance profiles. Figure 3 is the corpus seen through the categories. Two shapes recur. One is a benchmark split between the interpreter and the primitives it drives, with memory taking whatever its working set costs. The other is a benchmark that is essentially a single category, and those are the ones that decide what a suite can be used for. Native linear algebra is only 2.6% of the corpus as a whole, but it is concentrated rather than spread out. A program of such shape has almost no interpreted code to speed up and can therefore say very little about a compiler, however carefully it is timed.

4 Evaluation

To evaluate the usefulness of the categories, we want to see how well they can predict speedups for three different R compilers: (*bytecode*) R bytecode compiler vs the R AST interpreter, (*Ř*) Ř JIT compiler [4] vs vanilla R, and (*rcp*) RCP baseline JIT compiler [10] vs vanilla R. For the last two comparisons, the bytecode compiler is always on, so the speedup is over the bytecode compiler. We use the same environment as for the `perf` experiments. The only difference is that Ř is based on 4.1.1 compiled with gcc-9 and so it is compared against the same 4.1.1 vanilla R built with the same compiler and options.

Speedups are reported here by the ratio of the geometric mean of the baseline runtimes over the geometric mean of the contender runtimes. We run ten iterations in one process all sequentially. For *bytecode* and *rcp* the first five are discarded as cold and the geometric mean is over the remaining five. Ř is an optimizing JIT which needs more warm-up time, so five warm-up iterations are run and discarded and we compute the geometric mean over an additional ten timed ones.[2]

In Table 1, we show the geometric mean speedup per comparison, over the whole corpus and within each benchmark suite. Some benchmark suites, such as `riposte` or `misc`, have little speedup for some or all of the three compilers. Looking at how the speedups change across categories can help understand the performance of the compilers and identify potential performance problems in the benchmarks.

4.1 Predicting speedups from category shares

In Figure 5, we show the speedups for each of the compiler comparisons across unions of categories, taken in turn below.

[2] Ř is configured to start optimizing a function after 5 invocations.

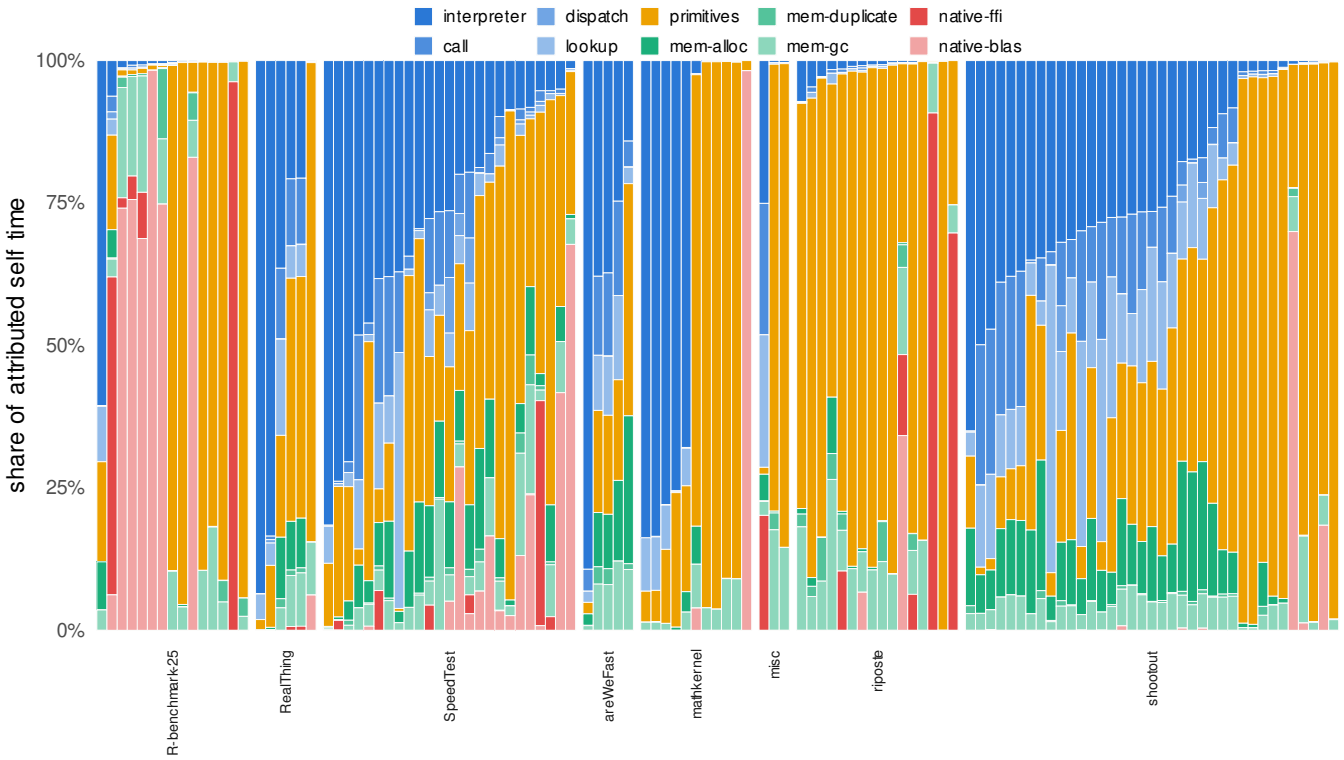


Figure 3. Where each benchmark spends its time, by the 11 categories of Section 3. One bar per benchmark, median of 5 runs, grouped by suite and ordered within a suite by decreasing `interpreter` share.

Table 1. Geometric-mean speedup per comparison, over the whole corpus and within each suite. Each contender is measured against its own baseline, so columns are not comparable.

Suite	bytecode vs AST	Ř vs bytecode	RCP vs bytecode
shootout	1.59	1.86	1.15
SpeedTest	1.91	1.04	1.09
riposte	1.00	0.96	1.01
R-benchmark-25	1.13	1.20	1.02
mathkernel	3.04	3.03	1.17
RealThing	2.74	1.82	1.16
areWeFast	2.55	3.31	1.19
misc	1.30	1.00	0.99
whole corpus	1.65	1.50	1.10

The speedups for each fine-grained category can be found in Figure S1 of the supplementary material.

***Interpreter overhead* (`interpreter + lookup + call + dispatch`).** This is the overhead of the interpreter a compiler can remove. Figure 5a shows that the speedup increases for two of the three: the Spearman correlation between that share and the log speedup is +0.87 for *bytecode* and +0.74 for *rcp*. Ř is the exception at +0.16, but Ř does not only aim at removing the interpreter overhead, unlike *bytecode* and *rcp*. It also performs function inlining and specializations.

***Native libraries* (`native-blas + native-ffi`).** Figure 5b shows the speedups against the share of time spent in the native categories. R compilers cannot optimize external libraries, so we expect to see a negative correlation as the share of native libraries increases. Indeed, the Spearman correlation between the share of those native categories and the log speedup is -0.29 for *bytecode*, -0.26 for *Ř* and -0.36 for *rcp*. Primitives, although also native code and not directly optimizable by an R compiler, could be replaced by optimized versions by the compiler as they are owned by the R interpreter. This happens only in a few specific cases in Ř.

***Memory* (`mem-alloc + mem-duplicate + mem-gc`).** Figure 5c shows the speedups against the share of time spent in the memory categories. The 3 compilers do not have specific memory optimizations compared to their baselines, hence no correlation is expected. The Spearman correlation between the share of those memory categories and the log speedup is -0.02 for *bytecode*, +0.05 for *Ř* and -0.05 for *rcp*. The share is well spread across the corpus, with a median of 12.0% and 79.7% of benchmarks spending at least 5% of their runtime in

these categories. The corpus does not contain a benchmark *dominated* by memory work: the largest share is 40.9%. A compiler could still improve performance here by reducing the number of memory copies (`mem-duplicate`) that R makes because of its copy-on-write semantics.

4.2 Diagnosing performance anomalies

Benchmarks that have a high share of an optimizable category but exhibit a low speedup or even a performance regression are candidates to be investigated. The categories can be seen as a diagnostic tool to detect performance anomalies.

For instance, for the combined interpreter categories (`interpreter + lookup + call + dispatch`), we can look at the number of performance regressions for the share of the 40 benchmarks that spend more than 50% in those categories. The bytecode compiler is never slower than the AST interpreter on those. RCP has only 2 regressions, but Ř is slower on 12 — 30.0% of the group. The worst slowdowns are large: `matmult_triplp`, `MMM-T1` and `heat` run at 0.31 × to 0.45 × on programs that are two thirds interpreter work. More generally, given a threshold on a category, we can rank the performance anomalies by the share of that category and how large the regression is.

If we denote by x the speedup and by s the normalized share of a category, we can obtain f, the factor by which the compiler can change the performance:

$$1/x = (1 - s) + s \cdot f \quad \Longrightarrow \quad f = \frac{1/x - (1 - s)}{s} \qquad (1)$$

$1/x$ is the duration of the contender by definition of x. We assume that the compiler does not touch the non-interpreter categories, which corresponds to $1 - s$. s is the part that the compiler targets. A higher factor f means that the interpreter regions are about f times slower.

Ranked this way, the three worst of Ř's regressions are also `matmult_triplp`, `MMM-T1` and `heat`, at f of 4.00, 3.20 and 2.70 respectively: on the worst of them Ř leaves the interpreter regions 4.00 times slower than the baseline runs them. For scale, Ř's median f across the whole corpus is 0.60: on the median benchmark it makes those regions faster rather than slower, which is what marks these three as anomalies rather than as the norm.

4.3 Selecting a smaller suite

Running all the benchmarks in the corpus takes a long time but some of the benchmarks are not relevant for a compiler evaluation. We can use thresholds on the share of some categories to filter out or include benchmarks. We do it here in two stages.

The first stage is `native-*`: a benchmark that spends most of its run inside a compiled library cannot be sped up by an R compiler by more than what Amdahl's law leaves it, whatever the compiler does. Dropping the 23 benchmarks whose `native-*` share is above 10% leaves 95 of 118, and moves the three compilers from 1.65, 1.50 and 1.10 to 1.82, 1.66 and 1.12.

The `primitives` category is not directly optimizable by a compiler either, and unlike `native-*` it is widespread: it covers functions as essential as arithmetic, so most of the corpus has some. The second stage consists of dropping the 30 of the remaining 95 benchmarks that spend more than 80% of their run in builtins. It leaves 65 benchmarks and saves 51.7% of the corpus's running time. Figure 4 shows what it does to the corpus: the `interpreter` categories' share increases a lot and `primitives` are not in 100% of the benchmarks anymore.

The second stage is not neutral between compilers though. On the reduced suite *bytecode* measures 2.37 and RCP 1.17, both higher than after the `native-*` cut alone (1.82 and 1.12), and the reason is that neither gains anything on the benchmarks this stage removes: over the 30 dropped programs they score 1.03 and 1.01.

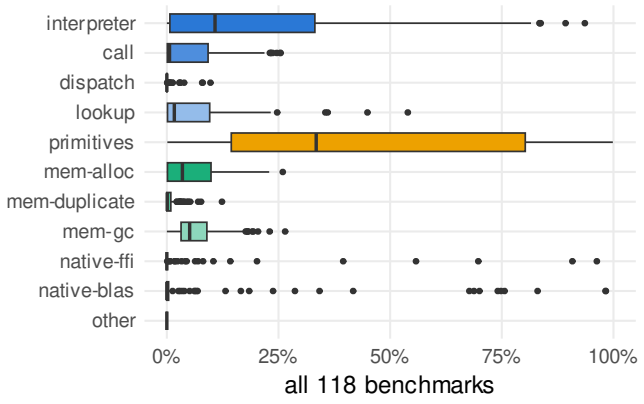
Ř behaves differently. It scores 1.66 on the dropped programs — the same as it scores on the suite before the stage and on the suite after it, all three agreeing to two decimals. The stage is invisible to Ř's column because Ř is the one compiler here that can specialize some builtins, so a builtin-dominated program is not a lost cause for it. A suite meant to measure builtin specialization should therefore stop after the first stage.

4.4 Pruning equivalent programs

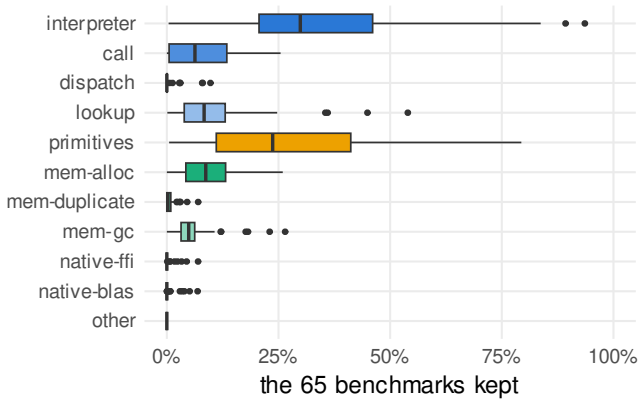
A seemingly low-hanging fruit is reducing the number of duplicate implementations of the same algorithm. The hypothesis is that implementations with similar profiles are equivalent therefore we can select a single representative candidate without losing any important data point. Yet, even with a very similar profile, the program behavior can be vastly different. If we look at the profiles of the `spectralnorm` programs in Figure 6, we can see all the implementations with varying profiles. The closely matching `spectralnorm` and `spectralnorm_naive` have significantly different performance on Ř, with speedups of 64.1 × and 12.6 ×, respectively. Similarly, `spectralnorm_alt_3` and `spectralnorm_alt_4` also exhibit different performance, achieving speedups of 11.6 × and 4.1 ×. The problem mostly comes from the categories. They are too coarse-grained to identify duplicates, and do not separate differently behaving programs sufficiently. To more accurately capture the nature of the program, we would need to break these categories further.

5 Related Work

Two questions place this work: what a benchmark suite measures, and how the time of a run is attributed to interpreter regions.



(a) All 118 profiled benchmarks.



(b) The 65 the two stages keep.

Figure 4. Category composition before and after the reduction: the spread of each category's share over the benchmarks, on a common scale. The cut lifts the interpreter categories off zero and takes the median benchmark out of `primitives`.

5.1 Benchmarking and suite design

Much work has addressed benchmarking and the setup of the experimental environment. Georges *et al.* [5] and Kalibera and Jones [8] address how to measure a benchmark rigorously. Mytkowicz *et al.* [16] show how easily measurement bias survives careful work, and Barrett *et al.* [1] show that even the assumption that a VM reaches a steady state often fails. That literature takes the choice of benchmarks as given. We ask the prior question of what a benchmark measures, and whether a given collection of programs fits what it is used to measure.

DaCapo [2] argued for real applications over micro-benchmarks and characterized its programs by object demographics and code complexity. Renaissance [18] did the same for concurrent and functional JVM workloads. *Are We Fast Yet?* [13] restricted itself to language constructs available across languages to make cross-language comparison meaningful. All three characterize their suites, and all three characterize a suite they designed themselves. In this work we characterize a corpus collected from across the R ecosystem, which we did not design.

5.2 Region profiles in CPython

The Faster CPython team has been computing performance profiles for the `pyperformance` suite continuously since April 2023. As in our work, each benchmark is recorded with `perf` and reduced to a share per category. They started with six categories over 28 symbol patterns and have at the time of writing extended these to 20 categories over 184 matching rules, following new CPython features such as the JIT, coroutines and free threading. Their categories mix activities such as lookup and calls with data types such as `dict` and `str`.

The main difference is in the profiling approach. Their profile is flat: `perf` records without call stacks, so a sample is one leaf address and no stack is unwound at either end.[3] The recording is gated live through `perf`'s control FIFO, enabled and disabled around each timed value, whereas we cut ours down to `execute()` using monotonic clock markers.

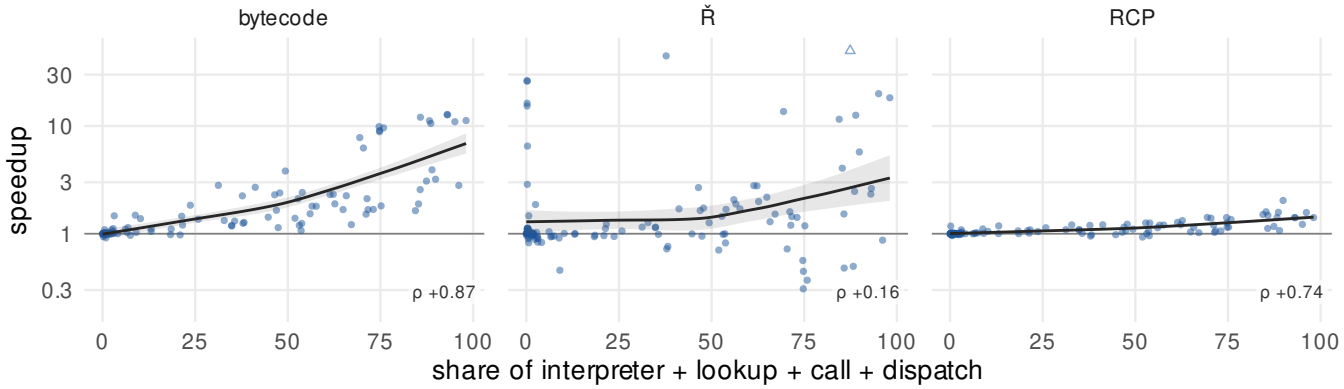
A flat sample carries one symbol, so a leaf outside the interpreter binary can name no region. Their table therefore keys on the object file first. The kernel, `libc`, the JIT and every `.so` become categories of their own, and only symbols in the `python` binary reach the 184 patterns. What is left there is *unknown*. Across the 76 profiles shipped with the tool, 13.0% of self time is bucketed by object file and a further 2.9% is unknown. The loss is skewed rather than uniform. The median benchmark gives up 8.6% this way, while `pickle_list` gives up 81.2%. Their patterns are also ordered and the first match wins, so a generic rule can outrank a specific one. 165 of 5,047 symbols match more than one category and carry 10.8% of self time. A symbol table is not lossless even for CPython, and the order of the rules decides what is lost.

A flat profile suffices when the executing function names the work. CPython's runtime is composed of many small functions under systematic prefixes, so a pattern over a symbol names a region. In R this does not work. Its hottest leaves are generic accessors that every region calls, *e.g.*, `DATAPTR`, `REAL_ELT`, `CHKVEC`, `CAR`, and `Rf_protect`. Each names a mechanism, not a region. Over our corpus 46.2% of samples sit on a leaf symbol that resolves to more than one category. There are 453 such symbols out of 1,362, and `DATAPTR` alone spans 9 of our 11 categories. A leaf-to-category table for R would not merely be lossy; it would be undefined.

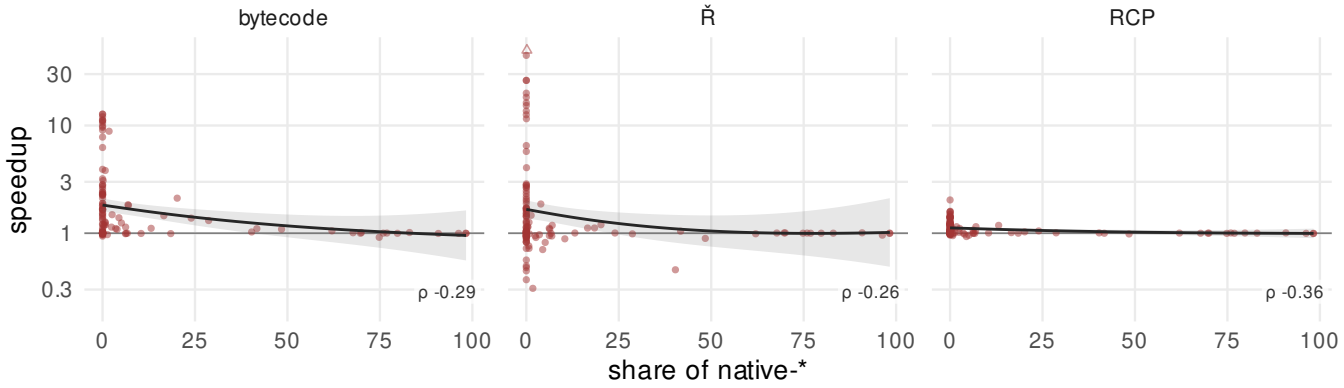
5.3 Deterministic profiling with `timeR`

An alternative to sampling is to instrument the interpreter. `timeR` [11] descends from the instrumentation Morandat *et al.* used to evaluate R's design [14]. It brackets chosen call sites in R's C source with a `BEGIN_TIMER`/`END_TIMER` pair. Each

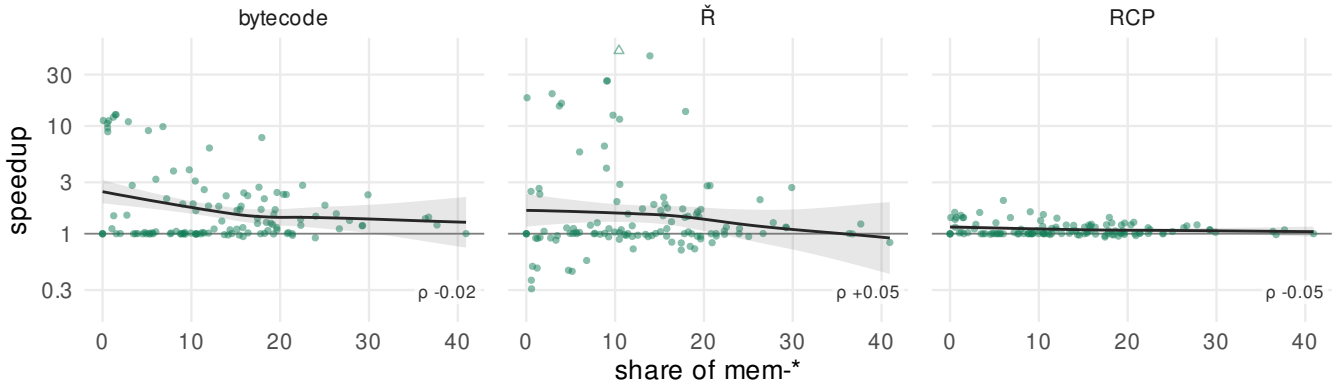
[3] Unwinding was in their first design and was dropped in 2024 because DWARF made the recordings large and the runs slow.



(a) `interpreter` + `lookup` + `call` + `dispatch`: the interpreter overhead a compiler can remove.



(b) `native-*`: `native-blas` + `native-ffi`: work inside a compiled library, which none of the three compilers can enter.



(c) `mem-*`: `mem-alloc` + `mem-duplicate` + `mem-gc`: what the allocator, copy-on-write and the collector cost.

Figure 5. Speedup against the share of `execute()` spent in each of the three unions of categories. Each point is one benchmark, and its speedup is the ratio of two steady-state geometric means: the baseline's over the contender's, each taken over that side's measured iterations after warm-up. Each contender is measured against its own baseline — bytecode against the AST interpreter, Ř and RCP against vanilla R — so the columns are not comparable to each other in level, only in slope. The line is a local regression through the points, fitted in log speedup, with its standard-error band. ρ is the Spearman correlation between share and log speedup. The triangle point in the scatter plot in the Ř comparison is one outlier, with a speedup of 819×.

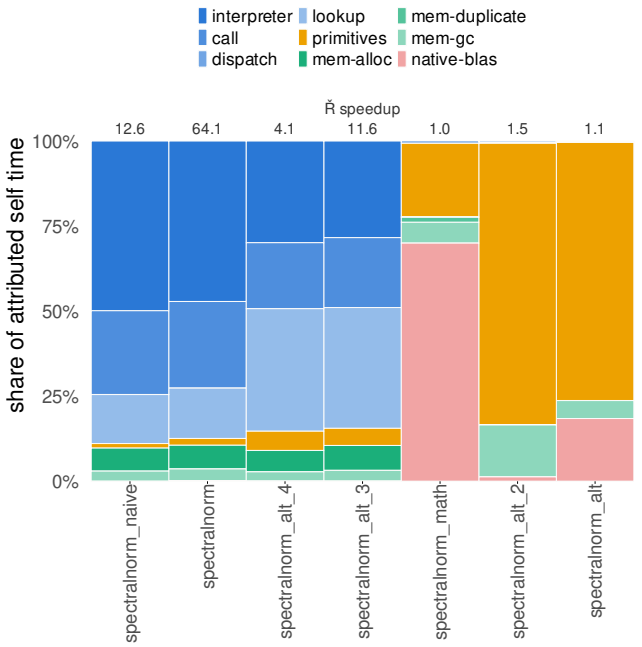


Figure 6. Figure 3 zoomed into the different implementations of the `spectralnorm` benchmark from the `shootout` suite. Above each column is the speedup of Ř against vanilla R on that benchmark, as described in Figure 5.

pair feeds a bin that counts calls and accumulates self time, nested timers subtracted. About two dozen sites are placed by hand: the allocator, the collector, duplication, symbol lookup, argument matching, the bytecode loop and the foreign-call boundary. The rest are generated, one bin per primitive and one per R closure. Attribution is exact, and a bin can name an R function. There are however several shortcomings.

`timeR` patches R 3.4.0 (released in April 2017). Moving it forward is not a rebase. Placing the timers correctly is challenging. Years of bytecode work added paths that carry no timer. For example, when an opcode's fast path calls the primitive's C implementation directly, no timer opens and the work is charged to the interpreter.

R unwinds with `longjmp`, which jumps past `END_TIMER` and leaves the timer open. `timeR` force-closes it with the enclosing timer's end time, so that timer's self time swallows all the nested work. The R object dispatch function never returns normally. It jumps to the caller of the generic, so every dispatch aborts a timer. On `convolution_slow` that leaves 86.2% of the run in dispatch, against `perf`'s 0.6%.

The remaining shortcomings are smaller but still structural. A timer pair costs 16 ns and charges it to the bin it opens, so the busiest bins inflate the most, and the allocator bins fire 3.5M times in a median run. Bins accumulate over the whole process, so a profile cannot be cut down to `execute()` afterward, unlike a timestamped sample.

6 Conclusion

R implementations are evaluated on ad hoc collections of programs. We collected 118 of them from 8 sources, put them under one harness, and characterized each one by a performance profile. It represents the share of its run spent in each of 11 categories of the virtual machine. They are obtained by sampling with `perf` and mapping call frames to category boundaries. Across three compilers, speedup tracks the share of the run a compiler can reach and vanishes once a program is dominated by native code.

The profiles are useful in two ways. First, as a selection criterion: dropping every program that spends more than 10% of its run in native code and then every one that spends more than 80% in builtins leaves 65 programs and cuts 51.7% of the suite's running time. Second, as a diagnostic: of the 40 programs that spend more than 50% of their run in the interpreter categories, Ř is slower on 12. Those are the programs where a compiler should win and does not, and the profile says which category to look at.

This is a step toward a curated suite for R, not the suite itself. The categories are still coarse. For example, `interpreter` and `primitives` hold work that a finer decomposition would separate, which is why programs with matching profiles can still behave differently. Splitting them is the next step. Nothing in the method is specific to R. Any implementation that dispatches into native builtins faces the same distinction between what a compiler can reach and what it cannot. We expect a suite to start small and grow, one profile at a time, as the regions a compiler targets change.

The benchmarks are hand-written to study R performance, but they are not necessarily representative of *real-world* R programs, which often have an emphasis on data wrangling and analysis. The *mathkernel* suite exercises some of the statistical functions, but we plan to extend our corpus with programs from R packages and notebooks. Another extension is to pair equivalent programs systematically, the same kernel written with explicit loops and with vectorized builtins, since the gap between a pair's profiles bounds what a native-code compiler can recover from R-level code.

Use of AI. Generative AI was used in preparing this work, in two ways. Claude (Anthropic), driven through Claude Code, wrote and refactored the R analysis notebooks that compute the categories from the profiles and emit every table and figure. It was also used to edit prose the authors had drafted. The authors reviewed all generated code and text and are responsible for the content of the paper, including the accuracy of its numbers and citations.

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] Edd Barrett, Carl Friedrich Bolz-Tereick, Rebecca Killick, Sarah Mount, and Laurence Tratt. 2017. Virtual Machine Warmup Blows Hot and Cold. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017). doi:10.1145/3133876

[2] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. 2006. The DaCapo benchmarks: Java benchmarking development and analysis. In *Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*. doi:10.1145/1167473.1167488

[3] 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

[4] Olivier Flückiger, Guido Chari, 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

[5] Andy Georges, Dries Buytaert, and Lieven Eeckhout. 2007. Statistically rigorous Java performance evaluation. In *Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*. doi:10.1145/1297027.1297033

[6] Isaac Gouy. 2025. The Computer Language Benchmarks Game. https://benchmarksgame-team.pages.debian.net/benchmarksgame/. Version 25.03, accessed August 2026.

[7] JetStream Contributors. 2026. JetStream 3.0: JavaScript and WebAssembly Benchmark Suite. https://browserbench.org/JetStream/. Announced 31 March 2026 at https://browserbench.org/announcements/jetstream3/. Repository: https://github.com/WebKit/JetStream.

[8] Tomáš Kalibera and Richard Jones. 2013. Rigorous benchmarking in reasonable time. In *International Symposium on Memory Management (ISMM)*. doi:10.1145/2464157.2464160

[9] 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

[10] 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

[11] Ingo Korb and Helena Kotthaus. 2017. timeR: an R interpreter with instrumentation for deterministic profiling. https://github.com/allr/timeR. Patches GNU R 3.4.0, last updated May 2017.

[12] Friedrich Leisch. 2006. A Toolbox for K-Centroids Cluster Analysis. *Computational Statistics & Data Analysis* 51, 2 (2006), 526–544. doi:10.1016/j.csda.2005.10.006

[13] 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

[14] 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

[15] Tyler Morgan-Wall. 2018. Throwing Shade at the Cartographer Illuminati: Raytracing the Washington Monument in R. https://www.tylermw.com/posts/rayverse/throwing-shade.html. Published 11 May 2018.

[16] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. 2009. Producing wrong data without doing anything obviously wrong!. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. doi:10.1145/1508244.1508275

[17] Radford M. Neal. 2020. pqR: a pretty quick version of R. https://github.com/radfordneal/pqR. Release pqR-2020-07-23.

[18] Aleksandar Prokopec, Andrea Rosà, David Leopoldseder, Gilles Duboscq, Petr Tůma, Martin Studener, Lubomír Bulej, Yudi Zheng, Alex Villazón, Doug Simon, Thomas Würthinger, and Walter Binder. 2019. Renaissance: benchmarking suite for parallel applications on the JVM. In *Programming Language Design and Implementation Conference (PLDI)*. doi:10.1145/3314221.3314637

[19] pyperformance contributors. 2026. pyperformance: the Python benchmark suite. https://github.com/python/pyperformance. Results tracked at https://speed.python.org.

[20] R Benchmark Suite contributors. 2016. The R Benchmark Suite: collections of benchmarks of R. https://github.com/rbenchmark/benchmarks. Last updated May 2016.

[21] Filip Říha and Filip Křikava. 2026. bench: A Programmable Command-Line Framework for Complex Benchmarking Workflows. In *Proceedings of the 2026 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity (SPLASH Companion '26)*,. doi:10.1145/3837729.3840502 Available at https://github.com/PRL-PRG/bench.

[22] ruby-bench contributors. 2026. ruby-bench: a set of benchmarks for the Ruby programming language. https://github.com/ruby/ruby-bench. Results tracked at https://speed.ruby-lang.org.

[23] Speedometer Contributors. 2025. Speedometer 3.1: Browser Responsiveness Benchmark. https://browserbench.org/Speedometer3.1/. Developed under a shared governance model by the Blink, Gecko and WebKit teams. Announced 31 March 2025 at https://browserbench.org/announcements/speedometer3.1/. Repository: https://github.com/WebKit/Speedometer.

[24] Lukas Stadler, Adam Welc, Christian Humer, and Mick Jordan. 2016. Optimizing R language execution via aggressive speculation. In *International Symposium on Dynamic Languages (DLS)*. 84–95. doi:10.1145/2989225.2989236

[25] Standard Performance Evaluation Corporation. 2008. SPECjvm2008. https://www.spec.org/jvm2008/.

[26] Justin Talbot, Zachary DeVito, and Pat Hanrahan. 2012. Riposte: a trace-driven compiler and parallel VM for vector code in R. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*. doi:10.1145/2370816.2370825

[27] Simon Urbanek. 2008. R Benchmark 2.5 (R-benchmark-25). https://mac.R-project.org/benchmarks/R-benchmark-25.R. Based on the R benchmark of Philippe Grosjean.

Received 2026-08-15; accepted 2026-08-27