Characterizing Type Feedback in Just-In-Time Compilation

Sebastián Krynski
Czech Technical University, Prague, Czechia

Filip Říha
Czech Technical University, Prague, Czechia

Filip Křikava
Czech Technical University, Prague, Czechia

Jan Vitek
Charles University, Prague, Czech Republic
Czech Technical University, Prague, Czechia

Abstract

Just-in-time (JIT) compilers leverage type feedback to optimize dynamic languages by recording runtime observations during interpretation and using them to generate specialized code. However, this recording process introduces significant execution time overhead to program warmup. This paper presents a characterization study of how feedback information is utilized by JIT compilers, providing motivation for future work on reducing recording overhead. We instrument the Ř compiler for the R language to track feedback slot usage throughout compilation, analyzing well-known benchmarks and real-world programs. Our measurements reveal that recording adds overhead to the interpreter up to 1.6× (mean 1.2×), yet at least 59% of non-empty slots are not used by the compiler. A large fraction of these unused slots can be attributed to dead code that the compiler eliminated. Most of the remaining slots are type-stable – meaning the inferred type matches the observed type – making speculation futile. To quantify the potential for improvement, we evaluate two oracle-based configurations that establish upper bounds on achievable reduction: a conservative set approach that preserves optimization quality while improving warmup, and an optimistic set approach that sacrifices some compiled code performance for faster interpretation.

2012 ACM Subject Classification Software and its engineering → Just-in-time compilers; Software and its engineering → Dynamic analysis

Keywords and phrases Feedback vector, JIT compilation, type speculation, deoptimization

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2026.16

Supplementary Material *Software (ECOOP 2026 Artifact Evaluation approved artifact)*: `https://doi.org/10.4230/DARTS.12.1.4`

Funding This work was supported by the Czech Science Foundation grant No. 23-07580X.

1 Introduction

Just-in-time compilers face a fundamental tradeoff: they must spend time learning about program behavior during interpretation before they can generate optimized code. This learning process records feedback about types, call targets, and branch decisions – information that enables specialization and optimization. However, this recording comes at a cost. The execution time during interpretation increases significantly, slowing down program warmup. This overhead matters: many functions never become hot enough to compile, programs spend time in the interpreter after deoptimizations, and short-running scripts do not amortize the recording cost. For interactive environments and serverless deployments, startup time is critical.

The feedback mechanism works as follows. During interpretation, when executing an operation like `x + 1` where the type of `x` is statically unknown, the runtime records the observed type of `x` – for instance, that `x` held an integer. Later, during compilation, this observation guides the generation of specialized integer addition code, guarded by a type check. Should the guard fail – for instance, if `x` later holds a string – a deoptimization event is triggered, execution jumps back to interpretation, the new behavior is learned, and a future recompilation adapts the code.

Feedback information is stored in a per-function data structure called a *feedback vector*, consisting of *slots* that summarize observed information about program behavior. Due to memory and performance constraints, what can be recorded is kept minimal so that it can be cheaply obtained and compactly encoded. Since the feedback vector summarizes all invocations of a function, it starts precise when few executions have occurred and becomes less specific as different values are observed across calls; this phenomenon is known as feedback vector pollution.

In this paper, we focus on *type feedback* – information about the runtime types of values. For a variable `x`, the feedback evolves as the function executes with different inputs. Initially, the feedback slot is empty, indicating no observations yet. After the first call, where `x` holds an integer, the feedback records this precise type. If a subsequent call provides a floating-point value, the feedback merges these observations, recording that `x` may be either an integer or a double. We call this merged summary the *observed type* – the union of all types encountered across different call contexts.

Despite the language runtime aggressively recording information about the program being executed, the compiler uses only a subset of it. Much feedback is unused for three reasons:

1. Many functions never become hot enough to compile, leaving their feedback unused.
2. Optimizations eliminate code paths whose feedback was recorded – for instance, constant folding removes branches, making their feedback obsolete.
3. Some type distinctions cannot be exploited by the compiler – if both observed and inferred types already match, speculation offers no benefit.

In all these cases, the recording overhead is wasted.

This observation motivates our work. We ask two questions: *How does the compiler utilize feedback information?*, and *Is it possible to reduce the overhead of recording feedback without impacting peak performance?* For the latter, we need to be able to distinguish necessary feedback from unnecessary feedback. Beyond the immediate performance gain, reducing recording overhead would also allow the compiler to record more detailed information about the program, such as the values of variables.

To investigate this question, we study Ř, a JIT compiler for the R language. R provides an ideal case study: its combination of lazy evaluation and extensive reflection capabilities requires particularly aggressive feedback recording, making it representative of the challenges – and opportunities for improvement – faced by dynamic language compilers more broadly. We analyze 14 programs from established benchmark suites, 3 Kaggle notebooks, and the Recommenderlab library, tracking feedback slot usage throughout the compilation pipeline.

Our findings. We instrumented Ř to track feedback usage during just-in-time compilation and measured performance impacts of various configurations of the system. Our measurements reveal that recording adds execution time overhead to the interpreter up to 1.6×, with a mean of 1.2×. We also report that at least 59% of the slots that had values were not used by the compiler. A large fraction of these unused slots can be attributed to dead code that the compiler eliminated. Most of the remaining slots are type-stable – meaning the type that the

compiler inferred for the slot matches the type observed by the recording; such slots are not used by the compiler. This characterization reveals a substantial gap between recording cost and actual utilization, suggesting that significant overhead could be eliminated by identifying which slots the compiler is likely to use.

To establish upper bounds on achievable improvements, we evaluate two oracle-based configurations that use retrospective knowledge of compiler behavior. The *Conservative sets* configuration includes all slots consumed during compilation – either by speculation or by optimization heuristics – and maintains full optimization capability while reducing interpreter overhead. The *Optimistic sets* configuration records only slots appearing in final speculation, trading compiled code quality for faster warmup. These configurations quantify what could be gained with perfect prediction, providing concrete targets for future practical implementations.

Our contributions. This paper presents a characterization study that makes the following contributions:

- We provide a detailed empirical analysis of how feedback information is utilized by an optimizing JIT compiler, identifying which recorded information remains unused and analyzing the reasons why. We propose a categorization of feedback slots based on their usage patterns during compilation.
- We evaluate two oracle-based configurations – *Conservative set* and *Optimistic set* – that establish upper bounds on achievable recording reduction, providing quantitative targets for future practical implementations.

Our results demonstrate that recording overhead could be substantially reduced while preserving code quality, motivating future work on predictive strategies for selective feedback recording in JIT compilers.

Paper organization. The rest of this paper is structured as follows. Section 2 provides background on type feedback in JIT compilers and describes Ř's feedback mechanism. Section 3 describes our experimental methodology and corpus. Section 4 presents our empirical analysis of feedback slot utilization and categorizes slots by usage patterns. Section 5 evaluates two oracle-based configurations that establish upper bounds on recording reduction. Section 6 discusses related work. All code, data, and analysis scripts are available under an open-source license[1].

2 Background

Dynamic languages lack type annotations, rely on reflection, and use late-bound polymorphic functions, making efficient compilation challenging. Just-in-time compilers address this by deferring compilation until the runtime provides behavioral feedback. During interpretation, the runtime records observations at key bytecode instructions or AST nodes. Once a function becomes *hot* (frequently executed), the compiler speculates that past behavior will persist and specializes the code accordingly. When behavior changes, feedback is updated and the code is adapted to the new execution pattern.

[1] https://doi.org/10.5281/zenodo.19551252

2.1 Feedback Recording

Feedback recording collects runtime observations during program execution. The JIT compiler later uses this information to generate specialized code, inserting guards to verify the types remain consistent.

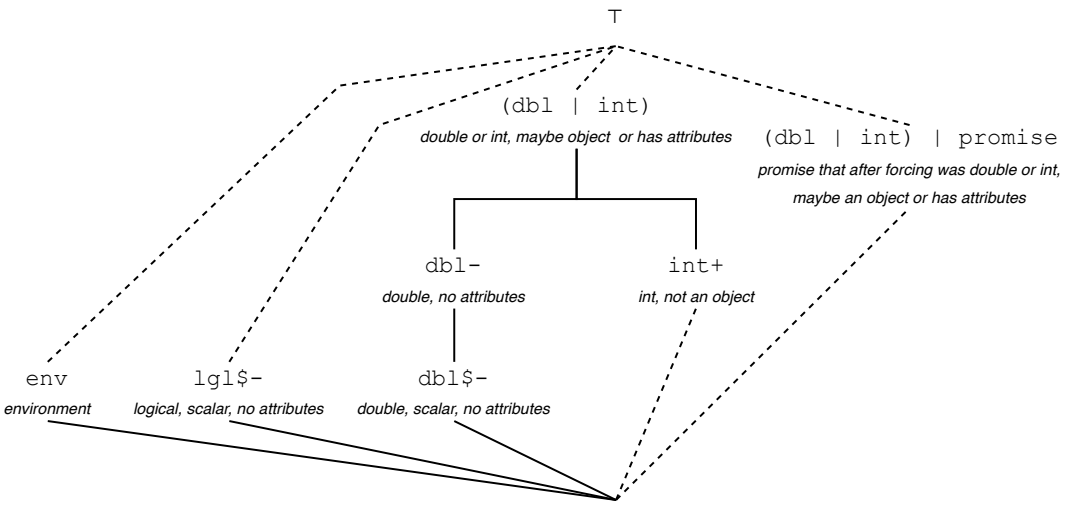

Figure 1 Observed values lattice: types lower in the diagram are more specific; arrows show the $\sqsubseteq$ ordering; multiple observations are merged using least upper bound ($\sqcup$). The suffix indicates scalar (`$`), no attributes (`-`), maybe an object or has attributes (`+`).

Categories of recorded information. The Ř runtime records three feedback categories: *observed callees* (closure pointers at call sites), *observed types* (expression types), and *observed tests* (guard outcomes). Values in each category form a lattice, with multiple observations merged via least upper bound. Up to three closure pointers per call site distinguish monomorphic (single target), polymorphic (few targets), and megamorphic (many targets) cases. Test feedback records guard outcomes, *e.g.* whether a builtin has been overloaded. Type feedback is more involved. R has primitive types (int, double, string) that are vectorized, plus heterogeneous vectors, expressions, environments, closures, and others. Almost any value may carry attributes (name–value pairs), some affecting dynamic dispatch, essentially turning values into objects. R also has two builtin object systems: S3 and S4. Beyond nominal type, the runtime tracks two properties. First, whether a value is scalar (`$`) – a vector of length 1 – or possibly non-scalar (no suffix). Second, its *objectness*: no attributes (`-`), has attributes but not an object (`+`), or possibly an object. These combine into type descriptors like `int$-` (scalar integer, no attributes) or `dbl+` (double vector, not object). Finally, R's lazy semantics pass arguments as promises – deferred computations with cached results. The runtime records the state before the last force: `promise`, `evaluated promise`, or plain `value`. Fig. 1 shows a simplified observed value lattice.

Compilation pipeline. Fig. 2 shows the compilation pipeline. An R closure's AST is first compiled into Ř bytecode. During bytecode generation, a *feedback vector* is allocated – a sequence of slots for storing observed runtime information. Each slot corresponds to a bytecode instruction that produces a value, such as variable loads, function calls, or arithmetic operations.

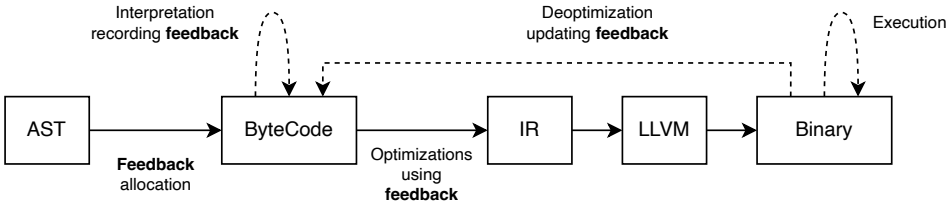

Figure 2 Ř compilation pipeline highlighting feedback lifetime.

Recording and speculation. Feedback is primarily recorded during bytecode interpretation. Once a function becomes hot, the bytecode is compiled into a higher-level SSA-style CFG IR [11]. The compiler leverages type feedback, code analysis, and optimizations to generate specialized code. Speculation is eager: type checks are inserted early to specialize code to assumed types, with each speculation guarded by an `Assume` instruction referencing its feedback slot. The optimized IR is then lowered to LLVM bitcode for the backend.

Deoptimization and pollution. When a guard fails at runtime, control returns to the interpreter and the corresponding slot is updated with the new observation. Future recompilations account for this updated behavior. Over time, feedback vectors become less precise as more behaviors are observed, degrading code quality – a phenomenon known as feedback vector pollution [17].

When recording occurs. During bytecode interpretation, each instruction that produces a value is followed by a recording instruction that updates an associated feedback slot. This update merges the current observation with the slot's existing content using the lattice least upper bound operation. For example, if a variable load instruction with slot `@1` first observes `int$-`, the slot records this precise type. On a subsequent execution with a different input, if the same instruction observes `dbl$-`, the slot is updated to `(int|dbl)$-`, their least upper bound. This merging continues across all executions, gradually generalizing the recorded type.

Recording mechanisms. To illustrate feedback recording, consider the R code in Fig. 3a. Following the compilation pipeline from Fig. 2, the R closure `f` is first compiled into bytecode. The resulting closure object contains bytecode and a feedback vector with 5 slots. Since R is lazy, each argument is compiled separately into a promise – `p0` in this example. Calling `f(42)` involves loading function `g`, creating and pushing a promise, calling the loaded function with one argument, pushing the result, and returning.

A promise is a triplet of code, environment, and cached value (initially unset). When forced, the code executes in its environment and caches the result. For `p0` representing `x + x`, this involves looking up variable[2] `x` using the promise's environment (*i.e.* `f`'s environment), forcing it if needed, and placing its value on the stack. This repeats for the second operand before the values are added and the result returned.

During execution, the runtime records feedback about program behavior. In `f`, it records `g`'s target (slot `@0`) and the return type of `g(x + x)` (slot `@4`) – here a scalar double with no attributes, `[ dbl$- ]`. In `p0`, it records `x`'s type twice (slots `@1` and `@2`) and the result of

[2] Ř distinguishes function loading (`LdFun`) from variable loading (`LdVar`) due to semantic differences.

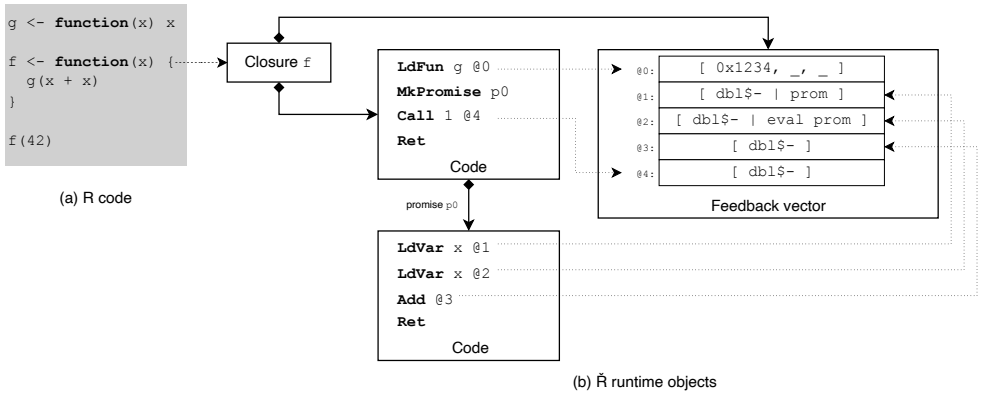

Figure 3 Feedback vector.

`x + x` (slot `@3`). The notation `[ dbl$- | prom ]` indicates `x` was bound to an unevaluated promise whose forced value was `[ dbl$- ]`. The second load of `x` also observes a scalar double, but since the promise has already been forced, it is recorded as an *evaluated promise*, `[ dbl$- | eval prom ]`. The three promise states are: `prom` (unevaluated), `eval prom` (already forced), and `value` (not wrapped in a promise at all).

Ř's recording is more extensive than in typical JIT compilers. In other languages, recording the second load of `x` might be omitted as redundant. However, R's reflection permits inspecting and modifying any environment on the call stack. Consider function `h`:

```
h <- function() { assign("x", 1L, sys.frame(-2)); 42 }
f(h())
```

which modifies variable `x` two frames up. Calling `f(h())` yields 43 for `x+x`: the first load sees 42 (the return value of `h`), while the second sees integer 1 after `h` modifies `x` in `f`'s environment. The type feedback becomes `[ dbl$- | prom ]` for the first load and `[ int$- | value ]` for the second. Without recording both, speculations on `x` would be invalidated, causing unnecessary deoptimizations.

2.2 Feedback Use in Compilation

The compiler uses an intermediate representation (IR) in Static Single Assignment (SSA) form, organized as a control-flow graph where each instruction has an *inferred type* drawn from the same lattice as observed types (*cf.* Fig. 1). Type feedback enables the compiler to speculate on more precise types than static analysis alone could determine, generating specialized code guarded by runtime checks.

Consider the function `f` and its recorded feedback from Fig. 3. The compiler produces the optimized version shown in Fig. 4. The `Assume` instructions represent speculation points. The first assume verifies that `g` matches the expected target, allowing the compiler to inline both `g` and promise `p0`. The second assume checks that the argument promise has type `[ dbl$- | prom ]`, enabling specialization of the addition in BB_3. The final assume verifies that the function environment remained unchanged after forcing the promise – ensuring no reflective modification (like the `assign` example above) has occurred. This guarantee makes the second load of `x` redundant, since its value must be unchanged, allowing the compiler to eliminate it. (One Ř optimization further elides call frames by creating stubs that materialize only when needed.) If any assume fails, the function deoptimizes: execution returns to the interpreter, and the corresponding feedback slot is updated.

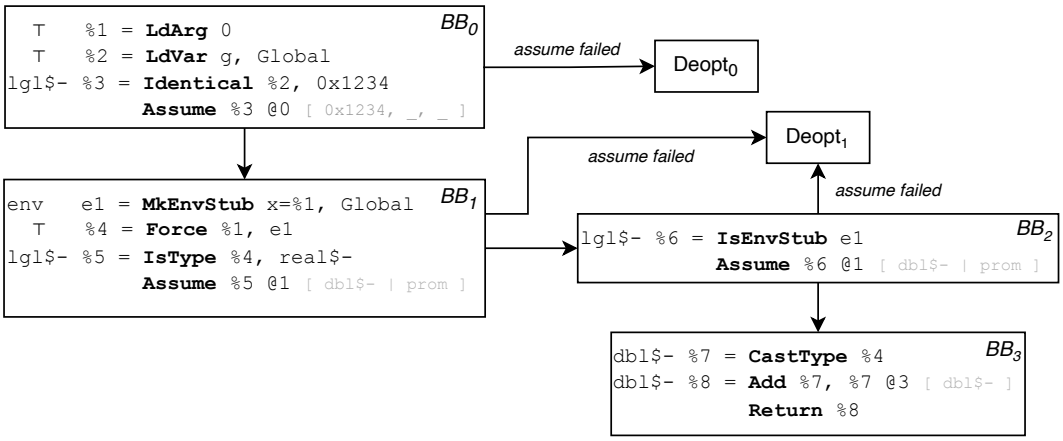

Figure 4 IR for closure `f` from Fig. 3 (*simplified*).

Feedback slot categories. Of the 5 feedback slots recorded during interpretation, only 2 are *used in speculation*: `@0` for speculating on `g`'s target, and `@1` for speculating on the type of `x`; the rest of the slots are *unused*. Note that some slots influence compilation without appearing in final speculation – for instance, by guiding inlining decisions or other heuristics (as we explore later in § 5). We say a slot is *present* if its associated instruction remains in the final optimized IR, and *not present* if the instruction was eliminated during optimization. Slots `@2` and `@4` (the second load of `x` and the result of calling `g`) are not present – they belong to dead code. Slot `@3` is present (still attached to an instruction) but remains unused because the inferred type of the `Add` instruction already matches the observed type, making speculation unnecessary.

2.3 Inferred, Observed, and Expected Types

Every IR instruction has an inferred type and possibly attached type-feedback information. We distinguish three types:

- *Inferred Type* (IT) known statically through dataflow analysis,
- *Observed Type* (OT) stored in a feedback slot,
- *Expected Type* (ET) IT $\sqcap$ OT, the type we can safely assume.

As stated before, the inferred type is the statically known type of an IR expression or instruction. It is computed during JIT compilation using dataflow analysis. Through speculation, inlining, and other optimizations, the inferred type evolves iteratively as subsequent passes unlock further refinements. The observed type is the runtime type observed and recorded during interpretation into a feedback slot. The compiler recovers precision by speculating on the expected type, which combines both the inferred and the observed type. The expected type must be more precise than the inferred type – for example, speculating on ET = `int$-` when IT = `any`.

The observed type may not always be included in the inferred type. When a function inlines callees, it incorporates their instructions along with their feedback vectors, which record observations from various call sites. If we take the following code:

```
f <- function(x, flag) { y <- x; if(flag) g(y) }
g(42); g(42L)
vec <- c(42L, 42L)
f(42L, TRUE); f(vec, FALSE)
```

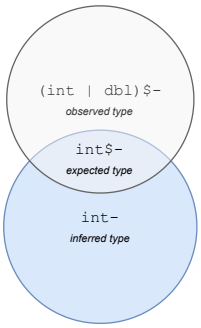

Figure 5 Inferred, observed, and expected types.

the standalone calls `g(42)` and `g(42L)` cause `g` to record an observed type of `(int|dbl)$-` on its argument (the union of both calls). Meanwhile, the function `f` records `int-` on the variable `y`, since `f` is only called with integer arguments. When `g` gets inlined into `f`, the `dbl` component of `g`'s observed type becomes irrelevant – it originated from a different calling context where `g` was called standalone. Fig. 5 illustrates this: the gray zone shows the observed type portion not included in the inferred type. Narrowing yields the expected type, computed as ET = IT ⊓ OT. Speculation operates on this expected type.

3 Experimental Setup

This section describes the experimental infrastructure for our empirical study. We present the compiler configuration, benchmarking hardware, and corpus of programs used to evaluate feedback utilization and recording overhead.

3.1 Compiler Setup

We use the latest Ř compiler[3]. We have modified the default behavior of the compiler in the following ways:

1. We lowered the compilation threshold from 100 to 10 invocations to collect more data points (compiled closures).
2. On-stack replacement (OSR) is disabled to avoid partial compilations that would complicate the analysis without adding significant value.
3. Ř features function specialization using Contextual Dispatch [12], where each function is specialized for different combinations of argument types, leading to multiple compiled versions of the same function. As this feature is not commonly present in other JIT compilers, we disabled it to make the results more generalizable. We experimented, however, with leaving this feature on and the results only changed in small ways. Note, only function specialization is disabled; guarded specialization with deoptimization remains enabled.
4. Upon compilation, Ř fills in empty slots with a default type so that speculation can still be issued on them. We disabled this behavior for a cleaner analysis.

[3] `github.com/reactorlabs/rir/commit/d1081b5`

Benchmarking Setup. Our experiments are run on a dedicated benchmark machine, with all background tasks disabled. The machine features an Intel i7-6700K CPU, stepping 3, microcode 0xea with 4 cores and 8 threads, 32 GB of RAM and Ubuntu 18.04 on a 4.15.0-151 Linux kernel. Experiments are built as Ubuntu 20.04.1 based containers, and executed on the Docker runtime 20.10.7, build `f0df350`.

3.2 Corpus

We evaluate three distinct codebases: the Ř benchmark suite, Kaggle notebooks, and the Recommenderlab library.

Benchmarks. The Ř benchmark suite[4] includes 14 programs from the well-known *shootout*[5] and *Are We Fast Yet*[6] suites, plus real-world programs (flexclust, volcano, convolution). The benchmarks contain 1,410 lines of code and run 20 times in the same Ř instance. We recorded 472 compiled functions with 17,119 type feedback slots (7,126 or 42% non-empty) across 556 compilations. Per benchmark, this averages 34 functions, 40 compilations, and 1,222 slots.

Kaggle notebooks. We selected notebooks for Titanic dataset exploration[7], Bolt taxi driver earnings analysis[8], and London Airbnb dataset analysis[9]. These 428 lines of code use various popular packages and operate on large datasets. Each notebook runs 15 times in the same instance to warm up occasionally invoked functions. We observed 3,748 compiled functions with 94,402 slots (44,227 or 47% non-empty) across 4,458 compilations. Per notebook, this averages 1,249 functions, 1,486 compilations, and 31,467 slots.

Recommenderlab. The Recommenderlab [15] examples from the CRAN repository span 222 lines of code. Execution yielded 1,060 compiled functions with 34,818 slots (15,188 or 44% non-empty) across 1,359 compilations. All examples run in a single Ř instance, each executed 15 times.

4 Empirical Study of Type Feedback

This study examines how an optimizing compiler utilizes feedback information. We investigate whether all recorded information is actually used and, if not, what factors prevent its use. We address the first research question:

- **RQ1**: How does an optimizing compiler utilize feedback information, and is any part of this information left unused?

To answer this question, we instrumented the Ř compiler to collect information about all compilations during program execution. For each feedback slot in a compiled closure, we record:

- Whether the slot is *empty* (instruction never executed) or *non-empty*

[4] `github.com/reactorlabs/RBenchmarking`
[5] `benchmarksgame-team.pages.debian.net/benchmarksgame`
[6] `github.com/smarr/are-we-fast-yet`
[7] `kaggle.com/code/mrisdal/exploring-survival-on-the-titanic`
[8] `kaggle.com/code/dannymumo/bolt-driver-earnings-analysis`
[9] `kaggle.com/code/godofoutcasts/eda-on-london-airbnb-r`

- Whether the slot is *used in speculation* or *unused*
- Whether the slot originates from the *top-level compilation* or an *inlinee*
- Whether the slot is *present* in the final optimized code or eliminated
- The *inferred type*, *observed type*, and *expected type*

We group the information per function compilation, where we collect the slot information of not only the top-level function, but also all of its inlinees. Inlining complicates the analysis, since it brings callees' instructions – along with their feedback slots – into the caller's code. To handle this, we track feedback separately for top-level slots (belonging to the main function being compiled) and for slots from each inlined function. We record only one set of information per inlined function rather than tracking each call site separately. We have observed that this does not impact the analysis, as most inlinings of one function behave similarly.

Since feedback slots change during program execution as more observations accumulate (during interpretation or after deoptimization), we snapshot each slot's state only during compilation when the program is quiescent. This ensures a consistent view of feedback information. We focus exclusively on non-empty slots, as only these can participate in speculation.

The collected information is aggregated per function compilation before being summarized for the whole program.

4.1 Feedback Slots in Speculation

We categorize feedback slots in Ř based on their usage in the speculation process, as illustrated in Fig. 6.

The following definitions use a helper function *widened* that widens the expected type into a form suitable for speculation when possible, otherwise returning the inferred type:

$$widened(IT, ET) = \begin{cases} ET & \text{if goodForSpeculation}(ET), \\ S & \text{if } ET \sqsubset S \sqsubset IT, \\ IT & \text{otherwise,} \end{cases}$$

where $S = \text{subsetForSpeculation}(IT, ET)$.

It attempts to normalize the expected type into a useful form for speculation, with the invariant $ET \sqsubseteq widened(IT, ET) \sqsubseteq IT$. If the expected type is a good speculation candidate, it is used directly. Otherwise, the compiler computes a more precise subset of the inferred type that can be exploited. If neither succeeds, it defaults to the inferred type.

goodForSpeculation checks whether the type is a good target for speculation as-is. This refers to non-object atomic types such as integer, double, or logical.

subsetForSpeculation computes a proper subset of the inferred type that can be used for speculation while being more general than the expected type. It takes the inferred type and attempts to strengthen properties such as non-object or no-attributes. For instance, if the inferred type is `any` and the expected type is `string+`, it returns `any+`, which is more widely applicable.

As a concrete example, consider an instruction with IT = `any` and OT = `int$-`. Since `int$-` is a primitive scalar type, goodForSpeculation returns true, so *widened* returns `int$-` – the compiler speculates on the exact observed type.

Used slots. A slot is used when the compiler speculates on it based on the inferred and observed types, refining the inferred type to unlock new optimizations. As described above, speculation may use the expected type directly (*e.g.*, slot `@1` in Fig. 4), or apply widening to find a more exploitable type. We distinguish two subcategories of used slots:

- *Exact match*, observed type is used directly in a speculation, and
- *Partial match*, observed type is narrowed or widened before participating in a speculation.

Note that narrowing discards observations inconsistent with the inferred type, but this is rare since the observed type typically falls within the inferred type.

Unused slots. Unused slots are either present or not present in the final version of the code. Slots that are not present correspond to instructions that were eliminated during optimization (*e.g.*, the slots `@2` and `@4` in Fig. 4), so no speculation on them is possible. Speculation on dead code is impossible. Unused present slots can be further divided into two subcategories based on the reason speculation did not happen:

- *Type stable slots*, formally $widened(IT, ET) = IT$, capture slots where speculation cannot improve precision. Either the widened expected type is not a meaningful speculation target, or the inferred type already matches it and speculation would not refine the type further (*e.g.*, slot `@3` in Fig. 4 is attached to the `Add` instruction with $OT = IT =$ `[dbl$- ]`).
- *Speculation blocked by non-type constraints*, formally $widened(IT, ET) \sqsubset IT$, contains slots where types align favorably but speculation is prevented by other factors. For instance, speculation requires an available deoptimization checkpoint; without one, speculation cannot be inserted. Additionally, optimization heuristics may intentionally defer speculation to allow subsequent passes to rewrite or simplify code first.

Summary. Fig. 6 summarizes this taxonomy. The categorization provides a systematic framework for understanding how feedback slots are utilized during compilation, distinguishing between slots that actively contribute to speculation and those that remain unused for various reasons.

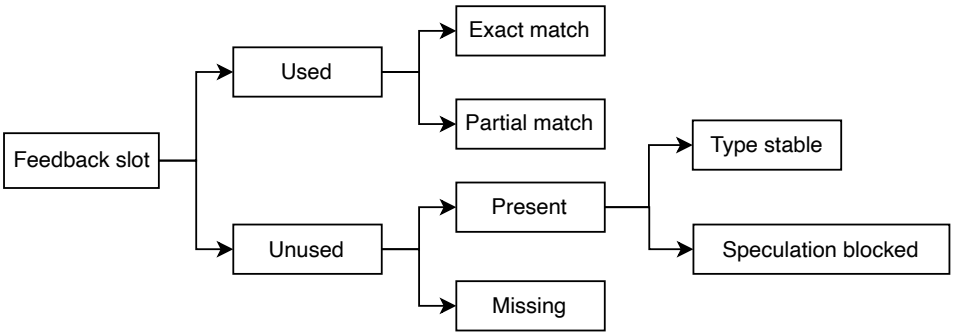

Figure 6 Categorization of feedback slots based on participation in speculation.

4.2 Results

We evaluate the above categorization by running the Ř compiler over a collection of benchmarks and real-world programs with instrumentation enabled. This allows us to quantify how feedback information is used in practice and derive insights from the empirical data.

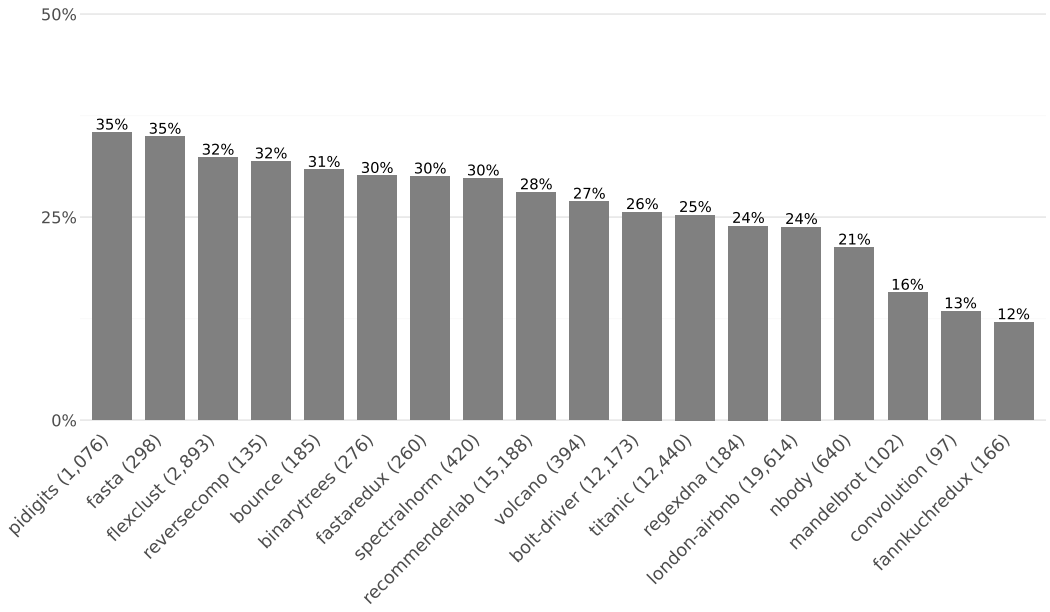

Figure 7 Ratio of used slots out of non-empty (number of non-empty slots in parentheses).

We define a slot as used if it was speculated on in at least one compilation and appears in the final version of the code. Fig. 7 shows the ratio of used slots out of all non-empty slots. On average, programs use 26% of their slots. Some benchmarks use as few as 12%. Manual inspection reveals that benchmarks with usage under 20% (convolution, fannkuchredux, and mandelbrot) primarily perform numerical operations on integers or doubles with many dependencies between computations. Since the resulting types of these operations are known for primitive types, the compiler can infer most program types from just a few speculations.

Among unused slots, on average 43% are not present in the final code. This is stable across all programs, ranging between 35% and 55%. R's lazy semantics generate significant code for delayed evaluation. The compiler often converts this to strict evaluation during optimization, eliminating many instructions along with their attached feedback slots.

For unused but present slots, we expect type stability to be the main reason speculation is skipped. Indeed, on average 74% of unused present slots are type stable. However, some programs show as low as 56%, meaning almost half of their unused present slots have types that could enable speculation, but speculation was blocked by non-type constraints. Programs with low type stability ratios (under 60%: nbody, regexdna, and reversecomp) primarily perform I/O operations (file loading, printing) and string processing. These operations are built into the virtual machine (VM) and can alter the calling environment, preventing compiler optimization.

Revisiting the low-usage outliers (convolution, fannkuchredux, and mandelbrot), we observe very high type stability percentages, ranging from 75% to 80% of unused present slots. This confirms our hypothesis: only a few speculations suffice to infer types for most of the program.

Examining how slots are used in speculation reveals that most usage is exact match, as seen in Fig. 8. On average, 64% of speculations use precisely the observed type. These are mostly speculations on primitive unboxable types: plain logical, integer, and double scalar types without attributes. This pattern is stable even for Kaggle programs, averaging around

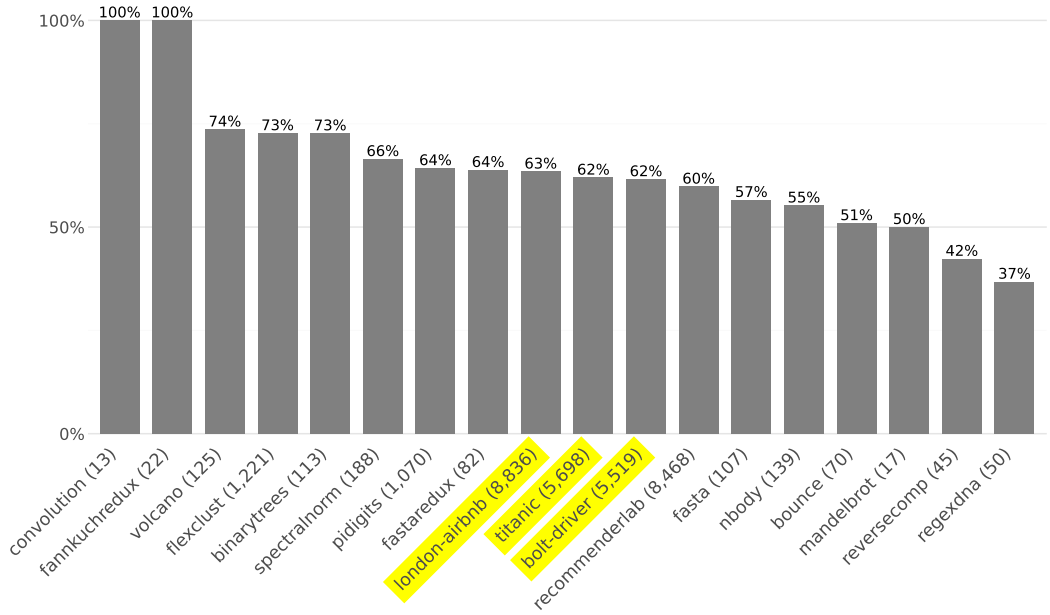

Figure 8 Ratio of exact matches out of used slots (number of used slots in parentheses).

62% (Kaggle programs are highlighted in Fig. 8). Even for programs using a large number of libraries, more than half of speculations use exactly the observed information, enabling precise speculations in non-trivial code.

Finally, we examine partially matched slots – speculations issued on types that were narrowed or widened. Narrowing is rare: out of 10,468 partially matching slots, only 14 are narrowed. These mostly involve properties statically inferred by the compiler that are too expensive to track at runtime, resulting in trivial narrowing.

The remaining partially matched slots involve widening. On average, 49% of these contain the `string` type, the most common reason for widening. This type is widely used in R programs, yet the compiler lacks targeted optimizations for it. These types are widened to retain only the additional observed properties. Programs with the highest widening ratios (regexdna with 63% and reversecomp with 58%) primarily process strings, where speculating on the exact type provides little benefit.

Table 1 Summary of type feedback usage.

Used slots	26%
Exact match	64%
Partial match	36%
Unused slots	74%
Present	57%
Type stable	74%
Speculation Blocked	26%
Not Present	43%

Summary. Table 1 summarizes what we found:

- Slot usage behavior is similar between synthetic benchmarks and real-world code.
- Almost 75% of slots are not used for speculation, mostly because they are type stable.
- Over half of the speculations use exactly the observed type.

5 Bounding Recording Reduction

Recording feedback information adds significant overhead to the interpretation phase of a program. This motivates our second research question:

- **RQ2**: What are the upper bounds on reducing recording overhead while preserving peak performance?

To evaluate recording overhead, we measured interpreter-only execution time with and without feedback recording enabled. We disabled compilation entirely to evaluate the interpreter in isolation.

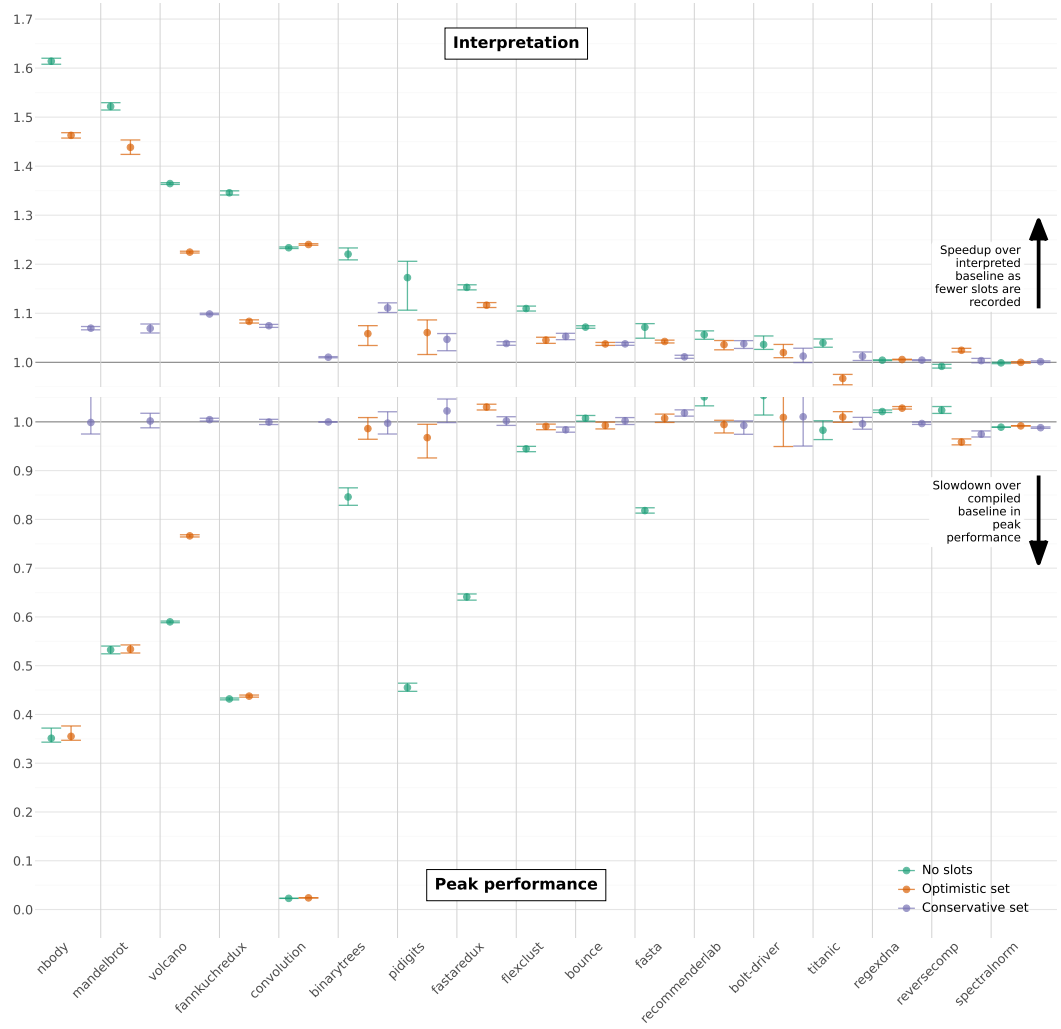

Figure 9 Comparison of execution times for different configurations, normalized to baseline.

Fig. 9 (upper panel) shows interpreter execution speedup for the programs described in the previous section, normalized to baseline (interpreter recording all slots).[10] The dots labeled *No slots* show the recording overhead that feedback profiling introduces. In this configuration, no profiling instructions are emitted in the bytecode, compared to the baseline where all are emitted. Recording overhead varies significantly: some programs experience overheads up to 1.6×, while others are barely affected (average overhead is 1.2×). Programs with high overhead primarily perform arithmetic operations in pure R code without native functions. Each sub-expression in the code's AST requires a record instruction (due to R's semantics), adding considerable cost to the interpreter.

The obvious solution to reduce recording overhead is to record less information. However, this requires identifying which slots are actually needed. Specifically, we must distinguish slots required for compilation from those that are unnecessary or redundant.

5.1 Slots Used in Compilation

We need to clarify what it means for a slot to be *used in compilation*. When a program runs on a given input, it records information in a set of slots S, and the compiler generates a result R (a set of optimized compiled closures). However, not all slots in S are relevant for reaching result R. We seek the smallest subset $S' \subseteq S$ such that recording only S' produces the same result R. As we show, considering only slots used in speculation in the final code may be insufficient and can degrade peak performance.

This definition is broader than *slots used in speculation* (Section 4), which counts only slots whose speculations persist in the final optimized code. *Slots used in compilation* additionally includes slots that influence the compilation process even if their speculations are later eliminated. A slot may guide an optimization pass (such as enabling loop-invariant code motion) even if the speculation it enabled is later subsumed or eliminated. This encompasses:

- Slots attached to speculative instructions emitted during compilation (whether or not they reach the final code)
- Slots consumed by optimization heuristics (*e.g.* inlining decisions based on call-site feedback)

This broader definition captures all slots needed to reach result R, though it may overapproximate – some slots might not be strictly necessary.

5.2 Reducing Recorded Slots

We now address RQ2: reducing the set of recorded slots. The goal is two-fold: (1) improve interpreter execution time (and thus program warmup), and (2) preserve the same peak performance as the baseline compiler that records all slots.

To evaluate the potential benefits, we instrument the compiler to track which slots are actually used during compilation. This provides an oracle that retrospectively identifies relevant slots – information not available in practice without running the full compilation. A practical implementation would need to predict which slots will be used, perhaps through heuristics or lightweight static analysis, without the benefit of this oracle.

[10] The london-airbnb Kaggle was removed, as it could not reach steady state in the given number of iterations.

Limitations. Our evaluation uses a fixed set of training inputs to determine which slots are used. Different inputs may require different slots, and the configurations may underperform when encountering code paths not seen during training. This limitation is inherent to feedback-directed compilation, where compilers specialize code based on observed behavior.

Additionally, we do not include slots that were empty during training. These might prove useful when different inputs explore new code paths. Handling such cases remains future work.

Configurations. We evaluate four configurations that vary the number of recorded slots. For each, we measure interpreter-only execution to estimate reduced overhead, and peak performance to assess optimization impact. The configurations, ordered by increasing slot count, are:

- *No slots*: no feedback information is recorded (baseline for comparison)
- *Optimistic set*: slots used in speculation (as previously defined)
- *Conservative set*: slots used in compilation (as previously defined)
- *Baseline*: all slots recorded (default compiler behavior)

The *Optimistic set* and *Conservative set* configurations provide bounds on achievable improvements: they show what could be gained if we could perfectly predict which slots the compiler will use. The *No slots* and *Baseline* configurations establish the extreme points for comparison.

We ran the same set of programs with each of the above configurations.

Results. On average, *Optimistic set* includes 26% of slots, while *Conservative set* includes 41%. In Fig. 9 (upper panel), *Optimistic set* demonstrates clear advantages over *Conservative set*, most prominently for nbody, mandelbrot, and convolution. Note that, in some cases (*e.g.* binarytrees, flexclust), the dots do not appear in the expected ordering; the *Optimistic set* is always smaller (or equal) than the *Conservative set*, and therefore the recording should be faster (or on par). The difference is within measurement noise: it is due to either the program running for too little time or the mentioned sets being too similar, or both.

Fig. 9 (lower panel) shows peak performance for each configuration, normalized to baseline (optimized code compiled under complete feedback information). One group of benchmarks reaches baseline performance even with no type feedback slots. For these programs, adding more slots provides no benefit, making all configurations equivalent. These programs also show minimal interpreter differences across configurations because they spend most time in native code, which Ř cannot optimize.

The remaining programs make significant use of type feedback, as evidenced by *No slots* showing substantial performance degradation. These programs appear in Table 2. With *Conservative set*, all programs reach baseline performance. However, *Optimistic set* falls short for some programs. Specifically, convolution, fannkuchredux, mandelbrot, nbody, and volcano perform equivalently to *No slots*, indicating that speculation-only slots are insufficient.

Understanding underapproximation. Why does *Optimistic set* fail for these programs? The issue is that some slots influence compilation even though they do not appear in final speculations. To understand this phenomenon, we examine the convolution benchmark as a representative example.

The convolution benchmark in Listing 1 illustrates how the *Optimistic set* configuration underapproximates the necessary slots. This simple program performs compute-intensive operations within a doubly-nested loop. In the baseline configuration (all slots recorded),

Table 2 Peak performance and slot usage for programs that benefit from feedback.

Program	Slots used		Speedup
	Optimistic set	Conservative set	Optimistic set
convolution	13%	53%	2%
nbody	21%	50%	36%
fannkuchredux	12%	51%	44%
mandelbrot	16%	41%	53%
volcano	27%	59%	77%
pidigits	35%	49%	97%
binarytrees	30%	39%	99%
fasta	35%	44%	101%
fastaredux	30%	45%	103%

the compiler emits assumptions on all intermediate steps of the computations in the loop – an element access on an object can execute arbitrary code which could modify the current environment. These assumptions are later pulled out of the loop and subsumed by speculation on the function arguments, allowing the compiler to infer the type of all variables. Only speculation on the function arguments survives in the final code; these become the Optimistic set. However, when the slots within the loop are not available, the compiler cannot propagate the information from the arguments to the loop section. This example highlights an important detail: even if an assumption is not present in the final code, its presence at some point may unlock critical optimizations.

```
convolve <- function(a, b) {
    a <- as.double(a); b <- as.double(b)
    na <- length(a); nb <- length(b); ab <- double(na + nb)
    for(i in 1 : na)
        for(j in 1 : nb)
            ab[i + j] <- ab[i + j] + a[i] * b[j]
    ab
}
```

Listing 1 Code of the convolution benchmark

Conversely, for binarytrees, fasta, fastaredux, and pidigits programs, the *Optimistic set* was sufficient to reach baseline performance. This is the ideal case: the compiler reaches full performance without tracking all intermediate uses of slots. The ratio of slots preserved in the *Optimistic set* for these programs is 33% of all non-empty slots, which is on average 26% fewer slots than the *Conservative set*.

Toward the minimal set. These results show that *Optimistic set* works for some programs but not others, while *Conservative set* succeeds in all cases. However, even *Conservative set* likely overapproximates: it includes all slots consumed during compilation, but some may not be strictly necessary. Neither configuration yields the minimal set of slots. Finding the true minimum would require a more sophisticated tracking mechanism than we have implemented. The fundamental difficulty is that when feedback information is consumed, it is impossible to predict whether it will affect the final result or be eliminated during optimization. The compiler runs many optimization passes iteratively until a fixed point is reached, and the order in which code is transformed is unpredictable.

An alternative approach would be brute-force search: start with the full baseline set and iteratively remove combinations of slots, checking each time whether the resulting configuration yields the same compilation result. However, this approach is computationally infeasible due to the combinatorial explosion of possible subsets, even when performed offline.

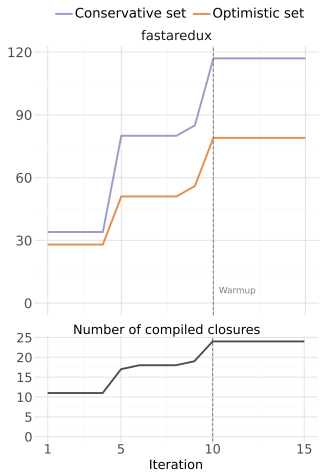

Figure 10 Example of the most common case of used slots saturation.

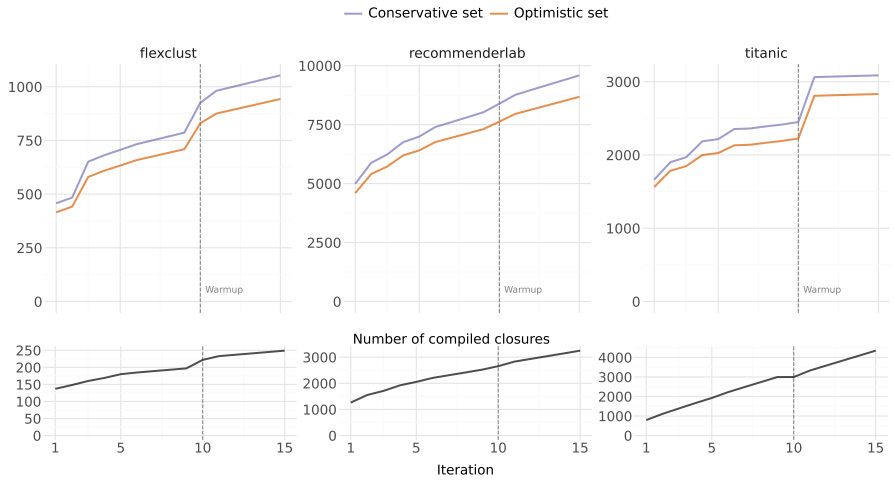

Figure 11 Examples of exceptional cases of used slots saturation.

Saturation of the *Conservative set* and *Optimistic set*. We conducted an additional experiment to evaluate the stability of the *Conservative set* and *Optimistic set* configurations. The goal is to evaluate how fast these sets saturate, meaning how many iterations are needed before no new slots are added to the set. This indicates how quickly we could identify the necessary slots for a given program. We ran the same set of programs while tracking the number of new slots discovered in each iteration, as well as the number of unique closures that were compiled. Fig. 10 shows the most frequent result: since programs typically have an inner loop, after a certain number of iterations (here 5) there is a pronounced jump in the number of compiled functions, as well as in the number of discovered used slots. The

second jump occurs after all the functions are warmed up (*i.e.* the iteration number is equal to the warmup counter; here 10). After that, the set is saturated. Notice that new slots only get discovered during compilation.

In a few programs, however, the set is not saturated even after warmup (flexclust, titanic, and recommenderlab). Fig. 11 shows this case – the number of closures and used slots is always increasing. This is because in each iteration a new function is created (*e.g.* using non-standard evaluation) which cannot be matched with an already existing definition. Therefore, in each iteration new referenced slots are created and new used slots discovered.

Summary. Our evaluation establishes upper bounds on achievable gains from selective recording. The *Conservative set* configuration demonstrates that profiling costs could be substantially reduced without sacrificing optimization quality, though a practical implementation would need to predict compiler decisions – a significant challenge given iterative optimization. The *Optimistic set* configuration shows more aggressive reductions are possible by accepting degraded performance for certain programs. These oracle-based bounds quantify the design space and tradeoffs, providing motivation and concrete targets for future practical implementations.

6 Related Work

Historical foundations. Feedback-driven adaptive optimization in virtual machines was pioneered in Smalltalk-80 [8] and extended in SELF [16], where type information was recorded to eliminate dynamic dispatch. Arnold *et al.* [1] provide a comprehensive overview of profiling techniques in modern VMs, highlighting the challenge of accurate recording with low overhead. The correctness and implementation of speculation with deoptimization has been formalized and studied extensively [13, 4, 14], establishing the theoretical foundations for safe speculative optimization.

Profile-guided optimization. While JIT compilers perform online profiling, static compilers employ profile-guided optimization (PGO) [9, 10, 7, 3], which shares the insight that runtime observations improve code quality beyond static analysis. However, PGO differs fundamentally in operation mode and recorded information. PGO operates in batch mode: programs are profiled with training inputs then recompiled. It primarily records control-flow information – branch directions, edge frequencies, basic block execution counts, and call-site frequencies – to guide branch prediction, code layout, and inlining decisions. Work on reducing profiling overhead through sampling [6] and instrumentation frameworks [2] has made PGO more practical. In contrast, JIT compilation for dynamic languages requires continuous online profiling of type information as well as control-flow, with immediate feedback and deoptimization support when assumptions are violated – a more challenging setting where both profile accuracy and overhead directly impact user-perceived performance.

Feedback in dynamic language VMs. Dynamic language VMs adopt different strategies for recording type information. Most systems maintain separate feedback structures. HotSpot [18] stores feedback per method in a method data object, with the C1 compiler collecting profiles that summarize bytecode instruction behavior – for instance, operations on references collect receiver types and null reference information. Wade *et al.* [20] examined the impact of these profiles on code quality. V8 [19] similarly collects feedback during bytecode interpretation, storing it in a feedback vector linked to each closure with over 20 slot types for different kinds of observations (*e.g.* hidden classes, binary operation types, loop jumps). V8 records at use sites for lazy speculation, deferring type checks until optimization opportunities arise.

Ř's feedback mechanism resembles V8's but differs in two key aspects. First, Ř records more extensively due to its pervasive reflection capabilities. Second, Ř records eagerly and speculates early – a design choice that simplifies implementation in the presence of lazy evaluation and reflection. Early speculation allows the compiler to gradually eliminate effects, which may unlock further optimizations such as variable loading (*cf.* the convolution example in Listing 1). Since trivially redundant speculations are eliminated, later optimizations can reduce guards to check only the difference between type expectations. For example, if an observed type is `int$-` and the inferred type is `int$`, the speculation guard checks only for `-` (*i.e.*, the absence of attributes).

Rather than maintaining separate feedback structures, some systems integrate profiling more tightly with execution. Truffle [21] performs specialization inline during AST interpretation through node rewriting: when a node encounters a new type, it rewrites itself to a specialized version, integrating feedback collection with specialization. PyPy's tracing JIT [5] similarly interleaves profiling with compilation by recording traces of hot loops, capturing type information implicitly in the trace rather than in separate feedback structures.

7 Conclusions

This paper presented a characterization study of type feedback utilization in the Ř optimizing JIT compiler for R, analyzing 14 benchmark programs, 3 Kaggle notebooks, and the Recommenderlab library. Our measurements reveal that recording adds up to 1.6× overhead to the interpreter, yet much of the recorded information does not influence the final compiled code. On average, more than 59% of non-empty feedback slots are not used by the compiler and do not affect compilation decisions. Only about a quarter of non-empty slots are used in speculation. Among the remainder, nearly half correspond to dead code, while most others are type-stable – the observed type does not refine the inferred type. These findings expose a substantial gap between the cost of feedback collection and its practical benefit.

To establish upper bounds on achievable improvements, we evaluated two oracle-based configurations that use retrospective knowledge of compiler behavior. Since these configurations rely on oracle knowledge – retroactively identifying which slots influenced compilation – they represent upper bounds rather than deployable solutions. The *Conservative set* configuration includes all slots that affected compilation decisions, preserving optimization quality while modestly improving interpreter performance. The *Optimistic set* configuration includes only slots appearing in final speculations, yielding greater interpreter improvements but degrading compiled code performance for some programs.

Our characterization demonstrates that recording far less feedback could be beneficial without sacrificing optimization quality, suggesting that current feedback mechanisms are overly conservative in practice. These upper bounds quantify the potential gains and tradeoffs, providing concrete motivation and targets for future work on practical selective recording strategies. The results are intended to be applicable across different contexts, languages, and virtual machine implementations. Within the R ecosystem, we have analyzed a diverse corpus of benchmarks, libraries, and real-world programs, covering different programming patterns. Across dynamic languages, R stands out as an upper-bound case for aggressive recording due to its laziness and reflective features; simpler languages are likely to exhibit even greater opportunities for reduction. Lastly, the fundamental insights – dead code elimination and type stability – broadly affect all JIT compilers (*e.g.* V8 and HotSpot face similar challenges).

Future work. Our approach reduces recording overhead based on a fixed set of training inputs. Whether a slot set exists that generalizes across all possible inputs remains an open question.

Implementing a practical reduced recording strategy requires predicting which slots the compiler will use without running the full compilation pipeline. Our findings demonstrate two key opportunities for optimization: slots in dead code and type-stable slots are unnecessary and could be eliminated. Dead code – code that is redundant or that can be optimized away under certain conditions – could be detected through heuristics based on program control-flow analysis, such as tracking dependencies across variables via use-def chains. Detecting type-stable slots requires lightweight static analysis to identify operations whose resulting type can be inferred from their operands' types. This challenge is particularly pronounced in R, where laziness and reflection complicate the analysis.

Once such a reduced recording strategy is in place, it opens the door to a complementary optimization: reallocating the saved recording budget toward more targeted fine-grained recording. By selectively capturing richer information about operations that matter most for optimization, the strategy can potentially enable more aggressive and precise specializations. In this way, reducing recording overhead not only lowers cost but also creates opportunities to improve optimization precision and effectiveness.

References

1 M. Arnold, S. Fink, D. Grove, M. Hind, and P. Sweeney. A Survey of Adaptive Optimization in Virtual Machines. *Proceedings of the IEEE*, 93(2), 2005. doi:10.1109/JPROC.2004.840305.

2 M. Arnold and B. G. Ryder. A Framework for Reducing the Cost of Instrumented Code. In *Conference on Programming Language Design and Implementation (PLDI)*, 2001. doi:10.1145/378795.378832.

3 T. Ball and J. R. Larus. Efficient Path Profiling. In *International Symposium on Microarchitecture (MICRO)*, 1996. doi:10.5555/243846.243857.

4 A. Barriere, O. Flückiger, S. Blazy, D. Pichardie, and J. Vitek. Formally verified speculation and deoptimization in a JIT compiler. *Proc. ACM Program. Lang.*, 5(POPL), 2021. doi:10.1145/3434327.

5 C. F. Bolz, A. Cuni, M. Fijałkowski, and A. Rigo. Tracing the Meta-Level: PyPy's Tracing JIT Compiler. In *International Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems (ICOOOLPS)*, 2009. doi:10.1145/1565824.1565827.

6 M. D. Bond and K. S. McKinley. Continuous Path and Edge Profiling. In *International Symposium on Microarchitecture (MICRO)*, 2005. doi:10.1109/MICRO.2005.16.

7 P. P. Chang, S. A. Mahlke, and W. W. Hwu. Using Profile Information to Assist Classic Compiler Code Optimizations. *Software Practice and Experience*, 21(12), 1991. doi:10.1002/spe.4380211204.

8 P. Deutsch and A. Schiffman. Efficient Implementation of the Smalltalk-80 System. In *Conference on Principles of Programming Languages (POPL)*, 1984. doi:10.1145/800017.800542.

9 J. A. Fisher. Trace Scheduling: A Technique for Global Microcode Compaction. *IEEE Trans. Comput.*, 30(7), 1981. doi:10.1109/TC.1981.1675827.

10 J. A. Fisher and S. M. Freudenberger. Predicting Conditional Branch Directions from Previous Runs of a Program. In *Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 1992. doi:10.1145/143365.143493.

11 O. Flückiger, G. Chari, J. Jecmen, Ming-Ho Yee, J. Hain, and J. Vitek. R melts brains: an IR for first-class environments and lazy effectful arguments. In *International Symposium on Dynamic Languages (DLS)*, 2019. doi:10.1145/3359619.3359744.

12 O. Flückiger, G. Chari, M.-H. Yee, J. Jecmen, J. Hain, and J. Vitek. Contextual dispatch for function specialization. *Proc. ACM Program. Lang.*, 4(OOPSLA), 2020. doi:10.1145/3428288.

13 O. Flückiger, G. Scherer, Ming-Ho Yee, A. Goel, Amal Ahmed, and J. Vitek. Correctness of speculative optimizations with dynamic deoptimization. *Proc. ACM Program. Lang.*, 2(POPL), 2018. doi:10.1145/3158137.

14 O. Flückiger, J. Ječmen, S. Krynski, and J. Vitek. Deoptless: Speculation with dispatched on-stack replacement and specialized continuations. In *Programming Language Design and Implementation Conference (PLDI)*, 2022.

15 M. Hahsler. Recommenderlab: An R framework for developing and testing recommendation algorithms. *CoRR*, 2022. doi:10.48550/arXiv.2205.12371.

16 U. Hölzle and D. Ungar. Optimizing dynamically-dispatched calls with run-time type feedback. In *Conference on Programming Language Design and Implementation (PLDI)*, 1994. doi:10.1145/178243.178478.

17 S. Krynski, M. Štěpánek, F. Říha, F. Křikava, and J. Vitek. Reducing Feedback Pollution. In *International Workshop on Virtual Machines and Intermediate Languages (VMIL)*, 2024. doi:10.1145/3689490.3690404.

18 M. Paleczny, C. Vick, and C. Click. The Java HotSpot Server Compiler. In *Java Virtual Machine Research and Technology Symposium*, 2001. URL: https://dl.acm.org/doi/10.5555/1267847.1267848.

19 V8 Team. Launching Ignition and TurboFan, 2017. URL: https://v8.dev/blog/launching-ignition-and-turbofan.

20 A. Wade, P. Kulkarni, and M. Jantz. Exploring Impact of Profile Data on Code Quality in the HotSpot JVM. *ACM Trans. Embed. Comput. Syst.*, 19(6), 2020. doi:10.1145/3391894.

21 T. Würthinger et al. Practical partial evaluation for high-performance dynamic language runtimes. In *Programming Language Design and Implementation (PLDI)*, 2017. doi:10.1145/3140587.3062381.