Reducing Feedback Pollution

Sebastián Krynski
Czech Technical University
Prague, Czechia
skrynski@gmail.com

Michal Štěpánek
Czech Technical University
Prague, Czechia
stepam38@fit.cvut.cz

Filip Říha
Czech Technical University
Prague, Czechia
rihafili@fit.cvut.cz

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

Jan Vitek
Northeastern University
Boston, USA
vitekj@icloud.com

Abstract

Just-in-time compilers enhance the performance of future invocations of a function by generating code tailored to past behavior. To achieve this, compilers use a data structure, often referred to as a feedback vector, to record information about each function's invocations. However, over time, feedback vectors tend to become less precise, leading to lower-quality code – a phenomenon known as feedback vector pollution. This paper examines feedback vector pollution within the context of a compiler for the R language. We provide data, discuss an approach to reduce pollution in practice, and implement a proof-of-concept implementation of this approach. The preliminary results of the implementation indicate ~30% decrease in polluted compilations and ~37% decrease in function pollution throughout our corpus.

CCS Concepts: • **Software and its engineering → Just-in-time compilers**; *Dynamic analysis.*

Keywords: Feedback vector, JIT compilation.

ACM Reference Format:
Sebastián Krynski, Michal Štěpánek, Filip Říha, Filip Křikava, and Jan Vitek. 2024. Reducing Feedback Pollution. In *Proceedings of the 16th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages (VMIL '24), October 20, 2024, Pasadena, CA, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3689490.3690404

VMIL '24, October 20, 2024, Pasadena, CA, USA

ACM ISBN 979-8-4007-1213-5/24/10
https://doi.org/10.1145/3689490.3690404

1 Introduction

What's past is prologue. Imagine you need to generate code for the JavaScript function `add(a,b){return a+b;}`. Without more information, the operation may concatenate strings, add numbers, or fail in various ways. Yet, it is likely that a given program will use this function in a specific and predictable way. Perhaps there is a hot loop that adds all the integers in a large array, knowing that `add` was called with integer arguments in previous iterations of the loop, could enable compiling a version of the code specifically for integers to speed up subsequent iterations.

Just-in-time compilers (JIT) speculate that previously observed behavior predicts future behavior. By recording information about prior invocations of a function, JIT compilers can unlock optimizations that would otherwise be unattainable. This is particularly important for dynamic languages, where the types of variables and even the implementations of common functions are not explicitly specified in the program text. Compilers for these languages use data structures, commonly known as *feedback vectors*, to record information such as variable types and the targets of jumps and calls. These feedback vectors are utilized to specialize operations based on the types of their arguments, avoid unnecessary allocations, eliminate unused code, and inline functions.

One way to understand the role of a feedback vector is as a compact summary of the behavior of multiple executions of a function that the compiler uses to drive code generation. For example, one could summarize multiple calls of the above function by the set of types observed for each argument. For practical reasons, operations on vectors have to be fast and memory-efficient. Thus, as more behaviors are observed, feedback vectors become less precise. Consider the above example, if one were to call the sum functions with integers, floats, and strings, at different times, the resulting feedback vector would be of little use to the compiler. Pollution occurs when the information in a feedback vector ceases to be useful for generating efficient code. Over time, each new observed behavior causes the feedback vector to decay. While there are strategies to mitigate this problem, such as resetting the feedback vectors, their effectiveness depends on the program at hand.

This paper studies the issue of feedback pollution in the context of R, a dynamic language for statistical computing in the vein of Python and Matlab. We start by measuring pollution in real-world code and benchmarks, then we discuss potential strategies for reducing pollution based on our previous work on contextual dispatch, and propose to extend it with multiple feedback vectors. We have implemented a proof-of-concept of this approach in the Ř JIT. The preliminary results show that it reduces the number of polluted compilations by approximately 30% and function pollution by approximately 37%. We do not, yet, report on performance as that will take a significant implementation and evaluation effort.

2 Motivation

R is a challenging target for compilation due to its dynamic nature [3, 5]. Consider the function of Listing 1, written in R, that sums the elements of a vector. The script takes approximately 60 seconds in a bytecode interpreter and 6 seconds when compiled (on Intel Xeon Gold 6136 CPU running Ubuntu 20.04). Let's look under the hood for a moment.

```
sum = function(vec, init) {
    s = init
    for (i in 1:length(vec))
        s = s + vec[[i]]
    s
}

N = 100*1000*1000
for (x in 1:N) sum(floats, 0.0)
for (x in 1:N) sum(integers, 0L)
for (x in 1:N) sum(floats, 0.0)
```

Listing 1. Motivating example

The interpreter execution is fairly stable, taking about 20 seconds on each for loop. Running the example on a JIT compiler, we observe the following behavior. Invoking the function with an array of floats will quickly transition the execution from interpretation to a fast native-compiled code specialized for this type, as it was the only one observed so far. At this point, the function is running at its peak performance and finishes in about 0.15 seconds. The emitted code guards the input type to be floats. Later, **sum** is called with an array of integers, the speculation guard fails, and the execution continues in the interpreter while the system learns about the new behavior. After more invocations for integers, the system triggers a recompilation. The type-feedback information now indicates that two different types were observed, therefore preventing type-specific optimizations to unlock. The execution resumes for integers on the recompiled version and finishes in close to 3 seconds. Now, the last execution for the floats also takes 3 seconds as the initial performance is lost. See figure 1. Note, compilation times are omitted; only peak performance is measured. While the above example uses R, the same problem occurs in other language VMs. For example, running the above program in V8 results in a similar behavior[1].

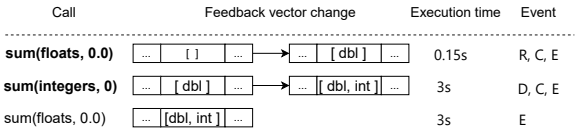


Figure 1. Execution trace for baseline compiler. R recording, C compilation, D deoptimization, E execution.

Let's run the example on Ř, a JIT compiler for the R programming language [3]. The Ř compiler uses contextual dispatch to specialize functions based on the observed types of arguments [2]. A context is a set of assumptions on function arguments. The system keeps multiple versions of a function, each optimized under different assumptions. Upon invocation, it dispatches to a version compiled under a context most suitable for the dynamic context of the call. Contexts can be partially ordered according to the level of specialization.

For floats, code starts running in the interpreter as the system learns the program's behavior. Some iterations later, the compiler is engaged and emits specialized code under a context $c1$ that assumes that the second argument is a float. The execution takes a similar 0.15 seconds as before. Now, we run the function for integers. Execution runs in the interpreter since there is no other suitable version available; the system now also learns about the new type. Subsequent invocations with integers trigger a new compilation under context $c2$ that states that the second argument is an integer. The current phase finishes in 3 seconds, again, as in the initial example. A different behavior is observed next in the last set of invocations for floats. The existing version for $c1$ is invoked and finishes in 0.15 seconds. Note that performance has not deteriorated this time as the code compiled for $c1$ used a feedback vector that had only observed floats. The overall execution took about 3 seconds. See figure 2.

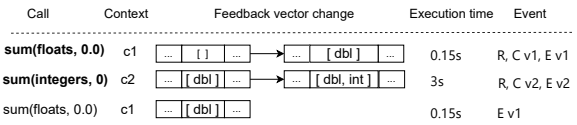


Figure 2. Execution trace for baseline compiler with contextual dispatch

In conclusion, while contexts already provide a level of specialization and show an improvement on the initial example, the feedback collected is still merged into a single vector, possibly preventing future code optimizations.

[1]Tested with Node v22.7.0 using floats and BigIntegers.

3 Feedback Pollution in R

What does a feedback pollution in R look like? To answer this, we create an experiment in which we look at function compilations in a sample corpus of R code. We observe the state of the feedback vectors and how they get polluted over the course of program execution.

3.1 Methodology

We start with an overview of the methodology used to gather the data: how a feedback vector looks like, what we are measuring, and how we are measuring it[2].

Feedback vector. The feedback is stored in a vector next to a function. Each element represents a slot that is connected with a particular instruction. We emit a feedback slot for each load, store, call, and test instructions. There is a partial ordering between the possible values that get stored in a slot. They are organized into a lattice, starting at $\bot$ to represent an empty slot. Recording a new value in a slot uses the least upper bound operation, climbing the lattice until $\top$ is reached. There are three types of feedback:

- *Observed call* stores pointers to function targets up to a certain size.
- *Observed test* is a flat lattice of `true` and `false` denoting whether the branch was always or never taken.
- *Observed value* contains a combination of types (up to a given size), flags (scalar, fast vector, object, attributes[3]), and whether it was a value, a promise, or an evaluated promise.

Pollution. Ultimately, we are interested in quantifying the feedback pollution. Pollution happens when a compiled function gets its feedback updated. In Ř it can happen by either a deoptimization when a failing assumption updates the corresponding feedback slot, or from an interpreter in the case a function is called in a context for which it has not been compiled yet. Given the monotonicity of the feedback vector, we use a feedback slot difference to measure feedback pollution. If two slots are not the same we have a pollution. Concretely, we use the following definitions to quantify the pollution in the corpus:

- *Polluted feedback slot.* A feedback slot whose value is different from its previous value (*i.e.* the value from the previous compilation).
- *Feedback pollution.* A ratio of the number of modified feedback slots to the total number of feedback slots.
- *Polluted compilation.* A compilation with feedback pollution greater than 0.
- *Function pollution.* A ratio of the number of polluted compilations to the total number of compilations.

[2]Data were gathered on the same machine as reported in Section 2.

[3]Most R values can have attributes, a dictionary of key-value pairs that can be used to store metadata.

Recordings. The Ř compiler can be instrumented to perform recording of runtime events such as compilation, updates to feedback vector, or deoptimizations. We use this feature to record the compilation events and together with the contents of their feedback vectors. We use the latest Ř compiler[4] with the default compilation heuristics: 100 invocations or 5,000 iterations for the on-stack replacement (OSR). The partial OSR compilations are ignored as they will be replaced by a full recompilation upon the next invocation.

Corpus. We use two codebases for our analysis: a collection of Ř benchmark suites[5] used to evaluate the performance of the compiler and a code from a Kaggle competition[6] that should represent a more realistic use of the language.

The benchmarks consist of 16 programs including the popular shootout benchmark suite[7]. Together, they contain 1.3K lines of code excluding comments and blank lines (79.2 on average). We ran each benchmark in the same instance of R 15 times. We have recorded 257 compilation events (16.1 on average per benchmark, 2.4 per function) across 139 compiled functions (8.7 on average per benchmark).

The Kaggle code has 108 lines extracted from 391 lines long data analysis notebook. Running the script triggered 970 compilations (on average 3.1 per function) for 315 functions over 515 contexts (on average 1.6 per function). Together, the feedback vectors contain 11,199 slots (on average 35.6 per function, with as few as 1 and as many as 497 slots). Splitting by type, there are 2,656 call slots, 1,010 test slots, and 7,533 type slots. As expected, the type slots dominate the feedback.

3.2 Pollution Analysis in Kaggle Code

We start with the analysis of the Kaggle code. Of the 315 compiled functions, 146 gets compiled more than once. There are 970 compilation events out of which 824 are recompilations (2.6 on average per function). Of these, 90 are recompilations with polluted feedback (10.9%). 19 recompilations have over half of their feedback slots polluted and 10 have all slots polluted. Looking at the functions, 66 out of the 315 are polluted (21%). 42 functions have over half of their compilations polluted and 26 have all compilations polluted.

Figure 3 shows the function pollution in the Kaggle code. The x-axis lists functions that are compiled multiple times. Each compilation is a dot and its position on the y-axis indicates accumulated feedback pollution.

The distribution of feedback pollution is illustrated in Figure 4, while the distribution of function pollution is illustrated in Figure 5.

[4]https://github.com/reactorlabs/rir/commit/2536eaf

[5]https://github.com/reactorlabs/RBenchmarking

[6]Exploring Survival on the Titanic, *cf.* https://www.kaggle.com/code/mrisdal/exploring-survival-on-the-titanic

[7]https://benchmarksgame-team.pages.debian.net/benchmarksgame/index.html

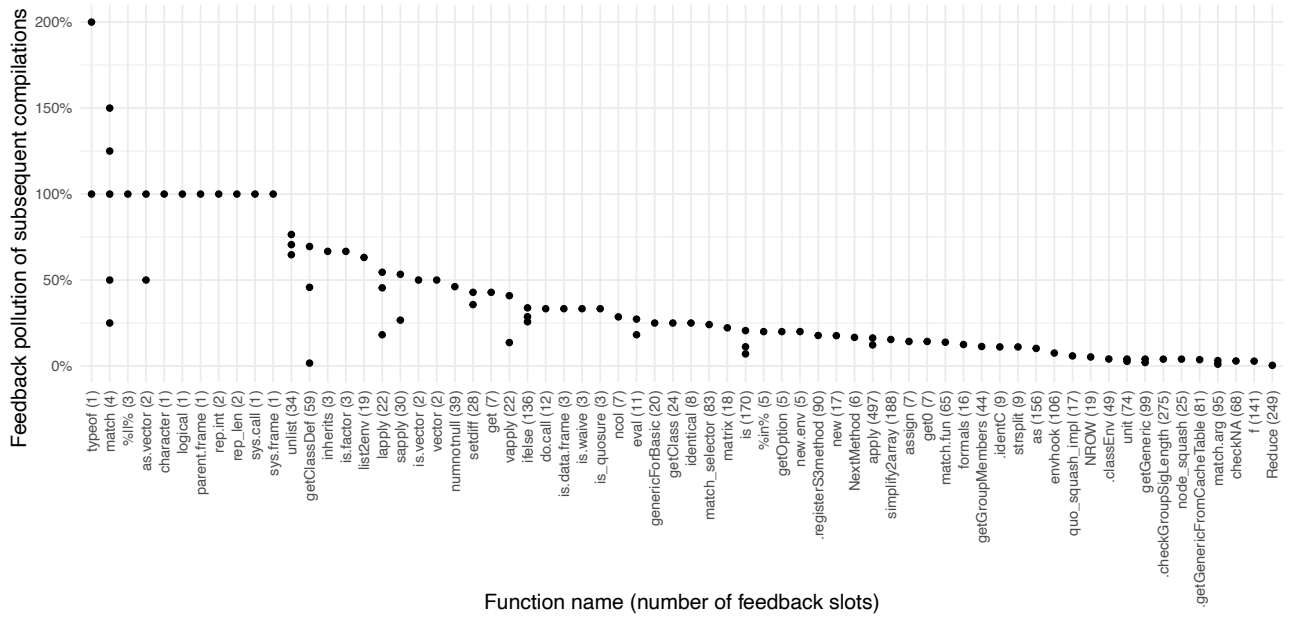


Figure 3. Function pollution in the Kaggle script. Each point represents a compilation. The y-axis shows an accumulated compilation pollution.

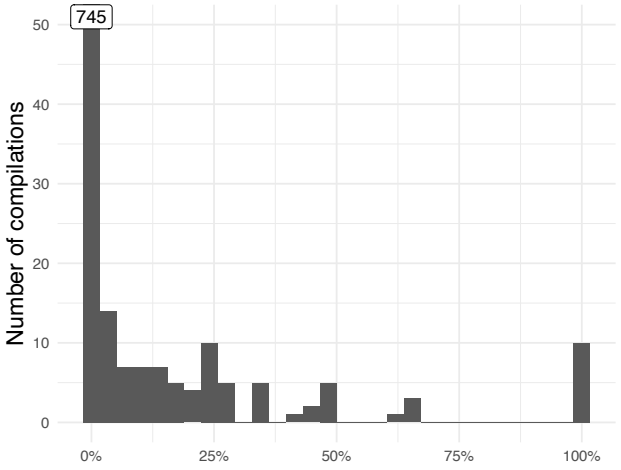


Figure 4. Distribution of feedback pollution across compilations in the Kaggle code.

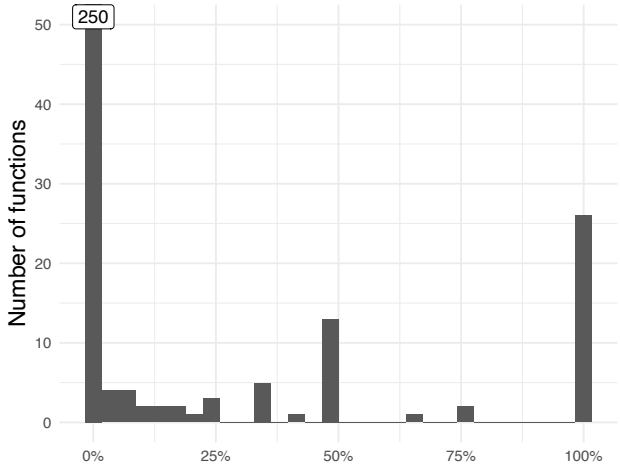


Figure 5. Distribution of the function pollution across the Kaggle code.

For example, the function `typeof` has three compilations in total. The plot shows that both the second and the third compilations use completely polluted feedback, *i.e.* all slots are different from the previous compilation. This is a simple R wrapper over the C builtin that determines the type of storage for a given value. It can be called with any type and thus will pollute all its slots with every new type.

Most functions that are compiled multiple times stabilize after a second compilation with about a quarter of slots polluted. The few that change over half of their feedback in a second compilation are very small functions with small feedback vectors. After manual inspection, most of these functions represent pollution caused by the fact that they are polymorphic, having some of the feedback slots connected to their parameters. But there are exceptions, for example, the `getClassDef` with over 50 feedback slots. It looks up a definition of a class that can be stored in a number of places (*e.g.* a package namespace or a local cache for frequently accessed classes). It contains a complex control flow to locate the correct definition. As it is called with different classes, it

exercises different execution paths, filling up the feedback as it proceeds. This is an example of pollution related to the global state.

3.3 Pollution Analysis in Benchmarks

Despite having more code, the benchmarks exercise fewer functions than the Kaggle code. This is unsurprising, as the benchmarks are small algorithmic programs that primarily do numerical computations. However, given that we use these benchmarks for the performance evaluation of the Ř JIT compiler, the question to ask is: *Do the benchmark programs suffer from feedback pollution?*

Figure 6 shows the *benchmark pollution*, *i.e.* the ratio of the compiled functions within each benchmark that have at least one polluted compilation.

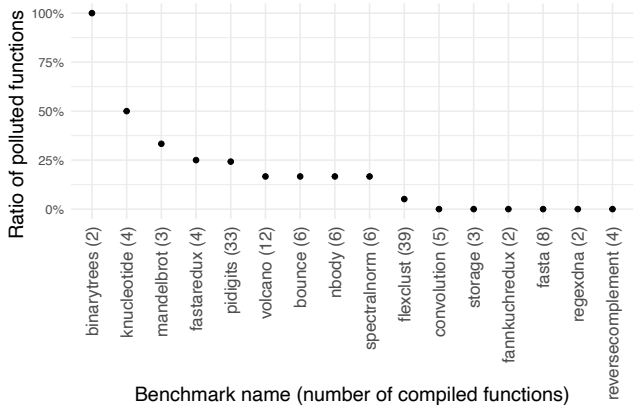


Figure 6. Benchmark pollution.

Out of the 16 benchmarks, 10 manifest feedback pollution (62.5%). The ratio between polluted feedback compilations is 8.2%, which is close to what we have seen in the Kaggle code. This is caused by a few functions from the standard library that get compiled a lot due to inlining. However, looking at the functions, from the 139 compiled functions, 21 are polluted (15.1%), suggesting that the benchmark code has a lower pollution rate than the Kaggle code. This is likely due to the nature of the benchmarks themselves. They are short algorithmic programs that are generally type-stable, and therefore not as likely to suffer from feedback pollution as other real-world code. Nevertheless, the pollution is present and should be addressed.

3.4 Feedback Slot Changes

From the three types of slots, the observed values are responsible for most of the pollution. Out of the 11,199 slots in the Kaggle code, 0.5% of observed calls, 2.7% of observed tests, and 5.7% of observed values are polluted during the execution of the program. This is expected as it is mostly types that vary.

The next question to ask is whether there are any patterns in the feedback pollution, and what type of changes are the most frequent. We find that there are 122 unique changes, but almost half of them (55) happen only once. Table 1 shows the top 10 most often observed changes. As expected, the most often changes are related to the types.

Table 1. Most often changes in feedback slots

Feedback slot change	Occurrence	Accumulated
add double	5%	5%
add logical, set is scalar	5%	9.9%
set always true	4.8%	14.7%
set has attributes	4%	18.7%
add NULL	3.8%	22.5%
add list	3%	25.5%
add integer, set is scalar	2.5%	28%
add NULL, add double	2.3%	30.3%
add character, set is scalar	2.3%	32.6%
change is scalar to is vector	2.3%	34.9%

3.5 Summary

Table 2 summarizes the feedback pollution in the corpus. The main takeaways are:

- The feedback pollution is present in both the benchmarks and the real-world code.
- The benchmarks have lower pollution rates than the Kaggle code mostly due to the different nature of the code.
- The pollution is mostly caused by polymorphic functions that are called with different types, but there are instances of pollution related to the global state.
- Slots representing the observed values get polluted the most.

Table 2. Summary of the feedback pollution in the corpus

	Kaggle	Benchmarks
Lines of code	108	1,268
Compilations	970	257
Polluted compilations	90 (10.9%)	21 (8.2%)
Compiled functions	315	139
Polluted functions	66 (21%)	21 (15.1%)

4 Context-Dependent Feedback in R

Acknowledging the feedback pollution, we propose an approach to reduce it by keeping separate copies of the type-feedback vectors, one per observed context. Let's revisit the motivating example from Section 2 under this lens. Executions in the interpreter for the context $c1$ will use a dedicated feedback vector that only observes the first argument to be an array of floats. Similarly, separate information is kept for $c2$ which only observes arrays of integers. When compilation

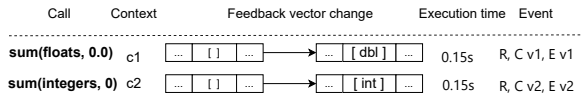


Figure 7. Execution trace for context-dependent feedback

triggers for each of the contexts, the relevant type-feedback vector is passed to the compiler, and, as a result, both get optimized for the fast case. The system has averted the previously observed type-feedback pollution and the overall execution decreased to less than a second, *cf.* Fig. 7.

For the following discussion, we introduce two definitions:

- *Compilation context* the context under which a function gets compiled. The compiler is free to use any assumptions present in the context to optimize the code.
- *Call context* has a dynamic nature and describes the arguments used at the call site. At invocation, the system uses the call context to find a suitable version that matches an existing compilation context. It either picks an already-compiled version or compiles a new one for the current call context. The operation always succeeds, since the top context (one that does not impose any restrictions) is always present.

Note that in the previous example the compilation and call contexts are the same. This is not always the case as it will be shown later.

Under the hypothesis that splitting the type-feedback vectors can avoid certain pathologies, we extend the Ř compiler with the new approach and describe the main changes to the default implementation.

4.1 Design

Extending a single feedback vector to multiple feedback vectors per function entails a number of challenges that need to be addressed:

- Where to store the additional feedback vectors?
- How to instruct the lazily evaluated promises to update the correct feedback vector?
- How to instruct inlined functions to update the appropriate feedback vector upon deoptimization?
- What to do with sparse feedback information?

For convenience of implementation and as a fail-safe mechanism, we record the profiled information twice: one for the current call context and one for the generic (or top) context. As such, the top context's feedback will combine all the information observed, similar to the single vector in the baseline implementation. This can prove useful when no other information is available. We are aware of the additional run-time cost the extra recording introduces, and it will be optimized in the future.

Additional feedback vectors. As with most JIT compilers, Ř keeps a single feedback vector per function. Additional vectors are now kept in an array, sorted by their corresponding call context, and starting from the most specialized context. This enables fast retrieval as a lookup simply returns the first satisfactory element.

Promises. R is a lazy language. A promise is created for each argument at call site and later forced by the callee when needed. As such, the execution of a promise in the bytecode interpreter can also record profile information. The baseline implementation will record this information in the mentioned single feedback vector of the function. In the new approach, we update the vector for the call context of the function that created the promise. But, as a promise can outlive the stack frame of its maker, this information might not be available anymore when the promise gets executed; therefore, promises now need to also remember the mentioned call context for a later use.

Call context of inlined code. In Ř, the inlined code is specialized for the static call site context inferred at compile-time, as there is no dynamic call context at the time of inlining. This context is now preserved in the IR to have it available should a deoptimization occur while executing inlined code. In other words, we want to preserve similar semantics as if the callee had not been inlined, and that means keeping additional information around that was previously not needed. Additional care is needed for OSR-compiled code.

4.2 Compilation Feedback

When a new compilation is triggered for a given context, a feedback vector is also an input to the compilation process. In a system that maintains multiple vectors, it is necessary to pick the most appropriate vector among all the available. The simplest would be to choose the vector that exactly matches the current compilation context. This, however, does not work well in practice. The observation is that the function being compiled could have been previously executed (interpreted) under several other call contexts than the current, but only a few times or none at all under the one being requested. In other words, the recorded information may be incomplete or scattered over a number of feedback vectors where some of the slots are empty. The trade-off is, empty feedback will hinder opportunities for optimization, whereas wrong or partial information will likely lead to deoptimizations.

Feedback substitution. We now address how missing feedback information is handled. The following example may seem rare, but was observed quite frequently. A function f calls g. Execution runs in the interpreter a few times while the feedback gets recorded. f runs in the interpreter and the inferred call context $c1$ is used to invoke g, therefore as the key for the recorded feedback information on each invocation to g. Sometime later, f gets compiled, which in turn triggers the compilation of g. When compiling f, the

optimizer picks up a compilation context $c2$, more specialized than $c1$, under which to compile (and later call) g. The problem is, there is no feedback information for the new context $c2$ as g was never run under it.

This poses an interesting situation where an inner compilation is requested and there is no relevant feedback information available. If we insist on compiling (and even inlining), we can borrow information from other contexts. This information, however, may be too generic leading to slow code, or too specific leading to deoptimizations. Alternatively, we can compile f and delay the compilation of g until subsequent invocations coming from the newly compiled f trigger a compilation for g. This means spending more time in the interpreter (or, worse, swinging back and forth from native to interpreted code), while ruling out inlining altogether.

We opted for the first option, namely forcing the inner compilation. The empty feedback for context C is substituted with the smallest C_{min}, such that $C < C_{\text{min}}$ and C_{min} contains sufficient information, defaulting at the top context. This works on the assumption that contexts closer to the top are more likely to contain information.

Feedback merging. This mechanism merges the feedback vectors recorded for different contexts. Merging is performed on a slot-by-slot basis and works by replaying the collected information in the source slot into the target slot. The following merging strategies were implemented and compared:

- no-merging strategy,
- merging of information of all smaller contexts, and
- merging of only the candidate contexts that would currently dispatch to the compiled version, therefore skipping the ones for which a compilation is already in place.

While this mechanism can unlock some optimizations, the main motivation for merging is to collect information recorded under call contexts which the newly compiled version could have been executed from. It addresses the problem of excessive deoptimization and deoptimization loops.

The no-merging strategy can lead to some well-known pathologies. As the compilation context can be different from the call context, a deoptimization event will update the feedback vector of the calling context, while also invalidating the current code for future executions. If no merging action is taken, recompiling the function under the same context as before will not include the recently learned behavior. This can lead to unwanted deoptimization and reoptimization loops.

Feedback filling. This mechanism fills the empty slots with feedback recorded for other call contexts. The following filling strategies were implemented and compared:

- no-filling strategy,
- filling with the top feedback, and
- filling feedback of context C with the information in contexts C_i, such that $C < C_i$, by traversing these contexts from the smallest to the top feedback.

Unlike the feedback merging, designed with a focus on minimizing deoptimizations, feedback filling focuses primarily on code optimization, no-filling being the most conservative.

In summary, the synthesis of the feedback information works as follows: a compilation is requested for a given context, and the corresponding feedback vector is selected if it exists; otherwise, the described feedback substitution mechanism is used. Only then are the set merge and filling strategies applied accordingly.

The mentioned strategies aim to strike a balance between having only one feedback vector and entirely separate feedback information for each context. As demonstrated in the examples above, the latter approach is impractical when taken to the extreme. The requested feedback information for a context might not always be available, so a compromise measure is needed. We experimented with combinations of merge and filling strategies for the R benchmark suite. Preliminary numbers indicate that merge candidates (3) and traversal filling (3) perform slightly better than other combinations, however this requires further exploration of real-world programs.

5 Preliminary Results

To experiment with the context-dependent feedback defined in the previous chapter, we have implemented an initial proof-of-concept in the Ř compiler. While the change itself is not big (~550 lines of C++ code), it cross-cuts most of the compiler and the runtime system code base.

In this section, we provide some preliminary results of the implementation using candidate merging strategy and traversal filling strategy. *How does our context-dependent strategy impact feedback pollution?* We focus our evaluation on the pollution itself rather than performance improvements. To measure performance, we would need to straighten the proof-of-concept implementation. For example, one shortcut we took was recording simultaneously the feedback for the top context and the current context.

Figure 8 shows the difference in the feedback pollution across compilations in the Kaggle code with the context-dependent feedback. In the majority of the functions, splitting the feedback reduced the pollution. There are a few cases where it got worse. Manually looking at the functions, we conclude that this is because the feedback is related to the global state. In this case, splitting the context amplifies the pollution. This is one of the issues that will be addressed in future work.

The impact of the change on the distribution of polluted compilations in the Kaggle code is illustrated in Figure 9,

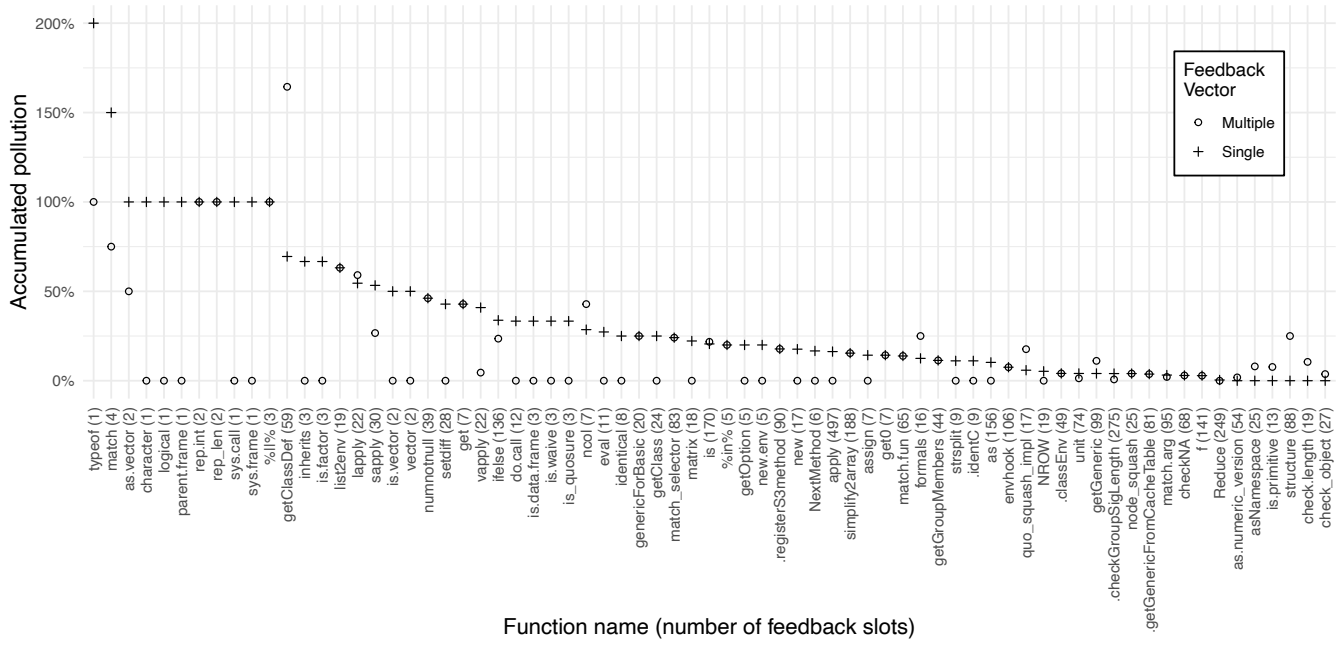
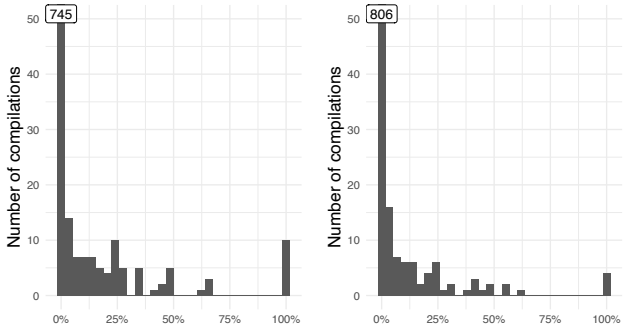
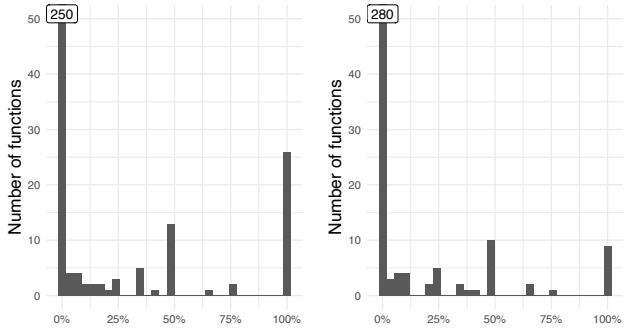


Figure 8. Difference in accumulated pollution in the Kaggle script with different approaches.



(a) Single feedback vector **(b)** Multiple feedback vectors

Figure 9. Distributions of feedback pollution across compilations in the Kaggle code for different approaches.



(a) Single feedback vector **(b)** Multiple feedback vectors

Figure 10. Distributions of the function pollution across the Kaggle code for different approaches.

while the impact on the function pollution is illustrated in Figure 10. Both figures show an improvement in their distribution for context-dependent feedback. In both cases, the most interesting changes are the decrease (of more than 50%) in the 100% polluted compilations and functions, and the increase of the non-polluted compilations and functions.

In Figure 11 we show the difference in feedback pollution for the benchmarks. Similarly as for the Kaggle code, in 7 cases, having multiple feedback contexts reduced the pollution. In one case, the pollution was amplified, again due to the global state.

Table 3 summarizes the pollution across the corpus for the proof-of-concept implementation. These results, compared to the results in Table 2, show ~30% decrease in the number of polluted compilations and ~37% decrease in function pollution.

6 Related Work

Feedback-directed compilers deal with profile data in different ways. The nature of the problem is, however, similar: compile for precision following the latest observed behavioral trends, or accept polluted information and compile for a more general case. The former strategy discards part of the

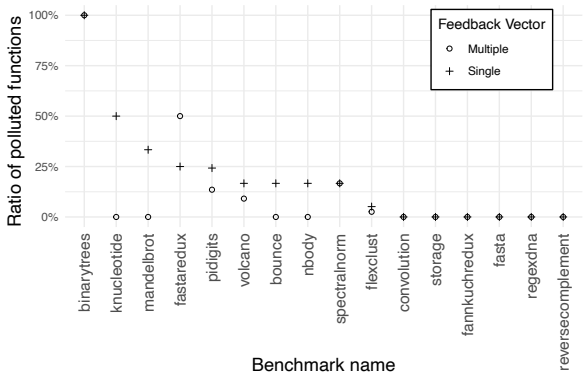


Figure 11. Difference in benchmark pollution with different approaches.

Table 3. Summary of the feedback pollution in the corpus with multiple feedback vectors

	Kaggle	Benchmarks
Compilations	987	275
Polluted compilations	68 (7.9%)	11 (4%)
Compiled functions	323	139
Polluted functions	44 (13.6%)	11 (7.9%)

observed information in favor of recent observations at the expense of additional recompilations and deoptimizations. *Context-sensitive profiling* is a technique used to reduce feedback pollution by keeping the profiled information separate based on the specific context the code is executed. The goal is to have precise information and thus enable more aggressive and specialized optimizations such as context-sensitive inlining. The context can take several forms: argument types, specific call sites, and even call stacks.

The Truffle Framework collects feedback through self-rewriting ASTs [8]. They deal with the problem of feedback pollution by tree cloning and inlining ASTs at call sites. As this increases the number of nodes in the system, a mechanism is needed to carefully choose the cloned targets. This differs from our new approach, as we keep only one feedback vector for all the call sites sharing similar specialization, therefore merging such information. It also alleviates the memory allocated for profile information.

The V8 Javascript engine uses feedback vectors to record runtime characteristics of the executing program. It shows similar behavior to Ř on the motivating example: calling the `sum` function with different types of arguments degrades its performance as more generic versions are compiled. In the past, V8 considered adding a time limit to the feedback vectors allowing to recover from pollution [7]. In JS, often a function is polymorphic only because it has been called with different types of arguments at the beginning of program execution. After some initialization, the program eventually stabilizes [6]. To prevent this kind of pollution, V8 does two things. First, it delays allocating the type feedback vector until the function is called at least 8 times. Second, unlike in Ř, it uses weak pointers in feedback vectors preventing it from keeping dead types or callee targets alive allowing for some recovery. In general, the worry in V8 is more in the deoptimization loops, not performance degradation.

Hotspot [4] uses method invocations and loop counts to trigger tiered compilation. Feedback is constantly refined as new information is collected. However, after too many deoptimizations, Hotspot gives up and compiles a more generic version. For inlining, Hotspot supports monomorphic, bimorphic, and mega-morphic callees. A method can be recompiled over time for a different set of callees in an attempt to capture the stable patterns.

Jalapeño's Adaptive Optimization System [1] accommodates different online analyses of feedback. It uses a cost-benefit analysis to determine whether a recompilation is profitable. Statistical samples of method calls are used to maintain an approximation of the dynamic call graph, and "hot" edges are then passed to the optimizing compiler. These inlining decisions are recomputed periodically. For example, a method can be flagged for recompilation if new opportunities for inlining have emerged since its last compilation.

7 Conclusion

Feedback pollution is a common issue in just-in-time compilers. The problem is more pronounced in dynamic languages, where functions are generally more polymorphic. The feedback collected for these different types gets merged into a single feedback vector, leading to less efficient code. This paper explored an alternative approach in which we keep separate feedback vectors for different invocation contexts. We implemented a proof-of-concept in the context of Ř JIT. The preliminary results are encouraging showing ~30% decrease in the number of polluted compilations and ~37% decrease in function pollution across the corpus. The new strategy can alleviate a family of existing pathologies; however, the general problem of pollution remains.

In summary, we have shown that context-dependent feedback can reduce feedback pollution. However, whether it will lead to performance improvements remains to be seen. As shown, there are functions whose performance can be improved by an order of magnitude. What is not clear is how many such functions are there in the real-world code. Analyzing that is part of our future work. Moreover, a polluted function does not necessarily mean the performance is lost in the compiled version, as some slots can have a meager contribution overall. This hints at a possible different approach that could statically identify and refine this information at compile-time.

The first step forward is to improve the current implementation which still has a ways to go. The recording overhead can be improved by avoiding redundant recording in the top context. New merge strategies and compilation heuristics for missing feedback information need to be further explored. Lastly, a finer-grained approach could be devised where part of the feedback information is context-dependent, while other is shared among other contexts. For example, the amount of feedback information could be also reduced by not storing slots that are fully determined by others. Without this engineering effort, it is hard to draw any conclusions.

Data Availability. Our code is open source and available online: https://gitlab.com/rirvm/splitfeedback-experiments/-/tree/artifact.

Acknowledgments

We greatly thank all the people contributing to Ř over the years. We thank the anonymous reviewers, for their insightful comments and suggestions to improve this paper. This work was supported by the Czech Science Foundation grant 23-07580X and a grant from Czech Ministry of Education, Youth and Sports under program ERC-CZ, grant agreement LL2325 as well as NSF grants CCF-1910850, CNS-1925644, and CCF-2139612 and by the Student Summer Research Program 2024 of FIT CTU in Prague.

References

[1] Matthew Arnold, Stephen Fink, David Grove, Michael Hind, and Peter F. Sweeney. 2011. Adaptive optimization in the Jalapeno JVM. *SIGPLAN Not.* 46, 4 (2011). https://doi.org/10.1145/1988042.1988048

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

[3] Olivier Flückiger, Guido Chari, Jan Jecmen, 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)*. https://doi.org/10.1145/3359619.3359744

[4] Sun Microsystem. 1999. The Java HotSpot Performance Engine Architecture. https://www.oracle.com/java/technologies/whitepaper.html.

[5] 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)*. https://doi.org/10.1007/978-3-642-31057-7_6

[6] Gregor Richards, Sylvain Lesbrene, Brian Burg, and Jan Vitek. 2010. An Analysis of the Dynamic Behavior of JavaScript Programs. In *Proceedings of the ACM Programming Language Design and Implementation Conference (PLDI)*. https://doi.org/10.1145/1809028.1806598

[7] Michael Stanton. 2016. V8 and How It Listens to You. https://www.youtube.com/watch?v=u7zRSm8jzvA.

[8] Thomas Würthinger, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Doug Simon, and Christian Wimmer. 2012. Self-optimizing AST interpreters. In *Proceedings of the 8th Symposium on Dynamic Languages* (Tucson, Arizona, USA) *(DLS '12)*. Association for Computing Machinery, New York, NY, USA, 73–82. https://doi.org/10.1145/2384577.2384587

Received 2024-07-26; accepted 2024-08-21