Understanding Feedback Pollution in the R Programming Language

Ing. Filip Říha Supervisor: doc. Ing. Filip Křikava Ph.D.

FACULTY OF INFORMATION TECHNOLOGY CTU IN PRAGUE

Motivation

Many modern dynamic languages are executed on a *virtual machine* (VM). To speed up the programs, VMs traditionally include *Just-in-Time* (JIT) compilers, allowing them to improve the performance of frequently executed pieces of programs by compiling them to native code.
To further enhance performance, during the execution of unoptimized code (in the *interpreter*) VMs record information about the runtime (called *feedback*, stored in a *feedback vector* composed of *slots*), allowing them to predict future behavior. If the feedback information is useful, the compiler speculates on it, leading to a more optimized code if the assumption holds.

Let's take the following JavaScript function:

```
function sum(vec) {
    let acc = vec[0]
    for (let i = 1; i < vec.length; i++) {
        acc = acc + vec[i]
    }
    return acc
}
```

By calling

```
sum(rangeDouble(1, 1e8)) // Peak execution time: 100ms
```

we observe the type `double`. After a few invocations, a compilation will be triggered, assuming the input is of type `double`. This allows the JIT to generate optimized native code that approaches C code performance.

If we then call

```
sum(rangeBigInt(1, 1e8)) // Peak execution time: 500ms
```

we update the observed types to both `double` and `BigInt`. The assumption on the type is broken, thus a new native version is compiled with weaker assumptions, resulting in less optimized code.

Even if we call again

```
sum(rangeDouble(1, 1e8)) // Peak execution time: 500ms
```

we do not reach the previously observed performance – this is called **feedback pollution**.

Goals

We look at feedback pollution in the context of *R programming language*, a high-level programming language specialized for statistical computing and data visualization.

More concretely, we analyze the pollution in the *Ř Just-in-Time compiler*, developed at CTU Prague and Northeastern University, Boston.

Recording Tool

In order to observe the behavior of Ř, we developed a tool for capturing and recording various events happening in runtime. This resulted in a tool that seamlessly integrates into the compiler, not impacting it at all if not required.

The result of a recording is illustrated graphically in figure 1. We can see the function invocations, compilations, deoptimizations, and observed types.

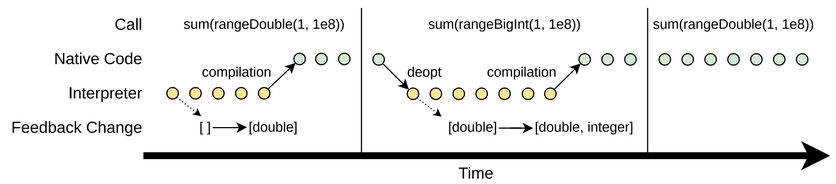


Figure 1: Graphical illustration of the recording tool result

Design Goals

- Capture the behavior of the compiler — function invocations, compilations, deoptimizations, and feedback updates
- Minimize the impact on the compiler performance
- Keep the changes in the compiler code minimal

Assessment
The implementation is in less than 1700 lines of code, excluding blank lines and comments.
Apart from 40 calls to the recording functions, no changes were implemented in the compiler.
The tool needs to be explicitly enabled in compilation; otherwise, it has no impact on the compiler and its performance.

Analysis of Feedback Pollution

This was the first usage of the recording tool, published in a 2024 VMIL paper. *(Krynski, Sebastián; Štěpánek, Michal; Říha, Filip; Křikava, Filip; Vitek, Jan. Reducing Feedback Pollution. VMIL '24. DOI: 10.1145/3689490.3690404.)*

Research Question
How does feedback pollution manifest in R?

Methodology
We define a *polluted feedback slot* as a slot whose value at the point of compilation has changed from previous compilation.
The experiment was run on Ř benchmarks and one real-life script.
Data was collected using the recording tool.

Results
Most functions compiled multiple times have about a quarter of slots polluted. Pollution mostly manifests in polymorphic functions or in functions that heavily use the global state.

Analysis of Feedback Usage

After observing that feedback can be polluted, our next step is to understand how the feedback is used in a compilation. The interpreter spends significant time on recording runtime information in hopes that it will make up for it by JIT compiling more optimized code. We want to analyze whether this hypothesis is true and whether there is room for improvement by reducing the amount of information recorded.

These partial results are part of an upcoming publication.

Research Questions

RQ1: How much of the recorded information is used?

RQ2: Why a slot is not used?

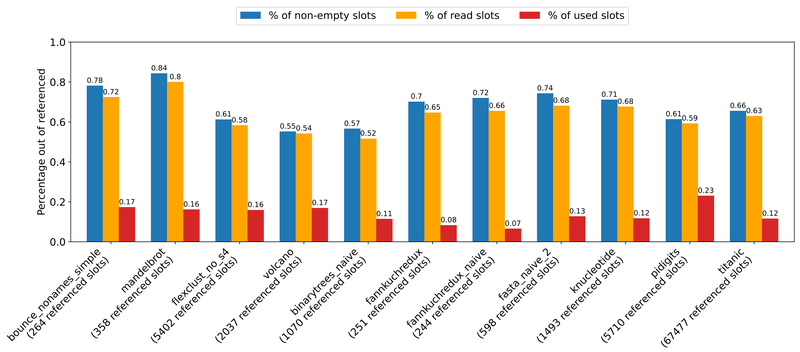
RQ3: How does pollution affect the slot usage?

Methodology
We define:

- *non-empty slot* – has at least one observation
- *referenced slot* – it is part of a compiled function
- *read slot* – the information is observed during compilation
- *used slot* – there is a speculation on this slot

The experiment was run on a few selected Ř benchmarks and one real-life script.

Results
The usage of slots on average per compilation:



On average, a compilation uses 21% of non-empty slots. (**RQ1**)

The main reason for not using a slot is redundancy — it contains the same information as some other slot. (**RQ2**)

Pollution of slots does not affect whether a slot is used, but it heavily weakens the assumptions the compiler can make. (**RQ3**).