A Sampling Profiler for a JIT Compiler

Master Thesis

Andreas Markus Wälchli
from
Madiswil BE, Switzerland

Faculty of Science
at the University of Bern

September 4, 2020

Prof. Dr. Oscar Nierstrasz
Olivier Flückiger

Software Composition Group
Institute for Computer Science
University of Bern, Switzerland

Abstract

For efficient execution of dynamically typed languages, many implementations use a two-tier architecture. The first tier is used for low-latency startup and collects dynamic profiles, *e.g.,* the types of all program variables. The second tier provides high throughput through the use of an optimizing compiler. This compiler specializes the code for the type information recorded in the first tier. If a program suddenly changes its behavior and presents the compiled code with types that have not been seen before and that are incompatible with the compiled version, that specialization becomes invalid. It is deoptimized and control is transferred back to the first tier where new profiles are gathered and specialization can start anew. But if the program behavior becomes more specific, for instance, if a variable suddenly becomes monomorphic (*i.e.,* only takes on one single type) this will not trigger a deoptimization as it is still compatible with the compiled version. If the program were recompiled with that monomorphic variable in mind, performance could be improved. Once the program is running in an optimized form there are no means to notice such optimization opportunities.

We propose the use of a sampling profiler to monitor native code without instrumentation. With the absence of instrumentation we incur no overhead when the profiler is inactive and can control the active profiler overhead by limiting the sampling rate. It also allows sampling at random points in the program and not just at predefined locations. Our implementation is in the context of the optimizing Ř JIT-compiler for the R language. Based on the collected profiles we are able to detect when the native code produced by Ř is specialized for stale type information and trigger recompilation for more specific type information. We show that sampling with our profiler adds an overhead of less than 3% in most cases and up to 9% in some cases when active. We also show that it reliably detects stale type information within milliseconds.

Aknowledgements

I would like to extend a special thanks to Olivier Flückiger whose continuous support and previous work on Ř was invaluable for my thesis. I also thank Prof. Oscar Nierstrasz for his patience and for the opportunity to prepare a second thesis at the Software Composition Group. Finally I thank everyone at the Ř team for providing me with their tools, testing infrastructure and support.

Contents

1 Introduction

1.1 Motivation

Efficient execution of dynamic languages is a challenging endeavor. This is well demonstrated by the many articles, conferences and resources dedicated to this topic. There are many challenges as solutions need to account for dynamic typing of variables, dynamic object layouts, and for introspection and reflection, just to name a few. The best established approach is to use virtual machines with multi-tier JIT compilers. This approach combines different execution engines with different trade-offs of compile-time versus execution-time. Early tiers, typically implemented as interpreters, favor low latency. Late tiers favor execution speed and are realized as optimizing native compilers. To deal with the dynamic nature of the source language, the virtual machine monitors the program execution and collects profiles. These are propagated from one tier to the next. This way the compiler can use information from the previous runs to optimize code for future runs in a way that is suited for the previously observed behavior. Programs are generally assumed to behave in such a way that the observed properties stabilize over time. After a so-called warmup phase the execution reaches stable behavior at peak performance. But unfortunately, as noted empirically by Barrett et al. [7], in reality this is not always the case. Changes in program behavior can degrade performance over time. Because the new behavior may not conform to the behavior observed during the warmup phase, existing optimizations may no longer be adequate. This may be because the new behavior is no longer compatible with the existing optimizations and the virtual machine has to fall back to an earlier execution tier with worse performance. This is known as deoptimization. Alternatively profiles recorded early in the execution may affect performance of later tiers by limiting and degrading their optimization choices, even if that profile information may be stale. This is due to the fact that highest tier, reserved for the busiest functions, no longer collects any profile information to avoid overheads that would reduce peak performance. Once a function reaches the highest tier, the execution flies effectively blind and the optimizations stay, unless a deoptimization event occurs, fixed for the remainder of the program execution.

We propose the use of a sampling profiler to address optimization for stale type feedback. With a sampling profiler no instrumentation is necessary and an overhead is only introduced while sampling. Our aim is to detect when for compiled code the feedback collected during the warmup phase is more generic than what could be observed by the profiler in later executions.

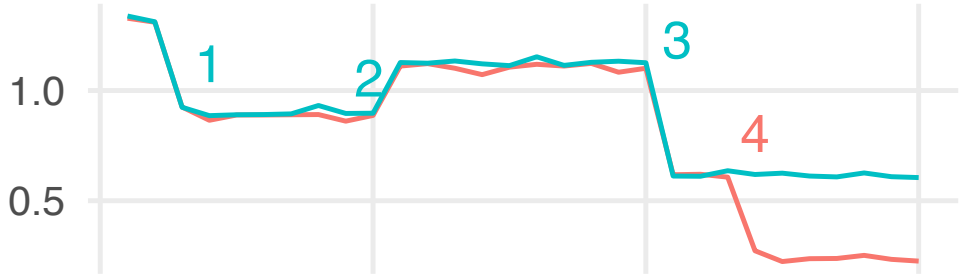
As a concrete example of the benefits of our profiler, consider an R user trying to compute values produced by a linear congruential generator. This could be implemented as follows:

```
lcg <- function(n)
  for (i in 1:n)
    state <<- (state * 48271) %% 0x7fffffff
```

Say the user is interested in the billionth value after an initial seed of `1`, `2` and `3`. In that case the following queries could be run in the REPL:

```
> state <- c(1,2,3)
> lcg(1e7)
> state
[1] 1901417813 1655351979 1409286145
```

As demonstrated here R is a vectorized language. Therefore multiple instances of this pseudo random generator can be run in parallel. The language also encourages an interactive exploration style: The user might continue to query different values for `lcg`, using differently sized vectors for `state`. Assume we record the run times in seconds of each invocation of `lcg` during an interactive session in Ř:



Let us focus on a number of interesting change points in that graph. First at (1) the `lcg` function is compiled and execution speeds up by a factor of 1.3. Then at (2) the user switches from three-dimensional vectors to six-dimensional vectors. The the execution takes longer again, but not twice as much, thanks to the built in vectorization. At point (3) the user starts querying `lcg` with a scalar `state` and performance improves again. Because scalar vectors are so common in R, the Ř compiler has special support for them and can treat them as machine floating-point numbers. But the stale type feedback from previous runs has caused the `lcg` to be optimized to the least upper bound of all observed types so far: a vector of numbers. At (4) we see that the two lines start to differ. The one staying flat is our baseline (*i.e.,* the current behavior), the red line dropping below includes our sampling profiler. The profiler has detected that `state` is now scalar and that `lcg` should be reoptimized with that information. The subsequent specialization leads to an over 2 times faster execution.

1.2 Outline

The remainder of this work is structured as follows: First we will highlight the technical background relevant to this thesis in chapter 2. We will then introduce the problem we are trying to solve as well as provide an overview of related work in chapter 3. In chapter 4 we will explain our implementation. We will evaluate our implementation in chapter 5. The thesis will conclude with chapter 6 where we will present our conclusion and future work.

1.3 Contribution

This work represents an initial effort in trying to use a simple variable type recording sampling profiler to improve JIT compiler output quality by providing more refined type information than was available during the initial compilation. We will show how such a profiler can be added to a virtual machine for general use without significant performance losses. We also show how such the data from the profiler can be used with an existing JIT compiler without adding additional optimization algorithms.

2 Technical Background

In this chapter we will introduce the technical concepts central to this work. We will start with JIT compilers and profilers. We will also take a look at the concept of feedback-directed optimization and finally introduce R and the Ř compiler and virtual machine that this thesis is built on.

2.1 JIT Compilation

Whenever a program should be executed it must be translated into a form the target platform can work with. Traditionally there are two ways such a translation can be done: compilation or interpretation [6].

Compilers, in the most general sense, are programs that convert code from one language into a second language. A compiler may take the source code and translate it into machine code that is directly executable on the target machine. It is also possible for a compiler to not emit machine code directly but instead an intermediate bytecode format that is easier to be used by further compilers or interpreters. An example for such a compiler is *javac*, the standard Java compiler. *Javac* accepts Java source code (`*.java` files) and emits Java bytecode (`*.class` files) that can then be interpreted by a Java Virtual Machine (JVM).

An interpreter does essentially the same analyses as a compiler but instead of emitting machine code for a later execution, the actions represented by the source code are directly performed and the program is in that way interpreted (*i.e.,* executed).

A JIT or *just-in-time* compiler bridges the gap between these two approaches by compiling the program (or just parts of a program) just before (or even during) the program execution. Combining a JIT compiler with an interpreter also allows for parts of a program to be interpreted while other parts can be compiled.

2.1.1 Comparison between Compilation and Interpretation

As explained by Aycock, compiled programs tend to run faster than interpreted ones as the program can be directly executed by the underlying hardware [6]. The decoupling of compilation and execution also allows the compiler to spend a theoretically arbitrary amount of time for optimizations that would not be feasible if they need to be done during the execution. This also highlights one central constraint of JIT compilers: Since they need to compile the code while the program is running they need to be relatively fast. This limits the amount and quality of optimizations that can be done in a JIT compiler.

Interpreted code on the other hand is often slower than compiled code as it cannot directly use the underlying hardware and much of the code analysis is performed during the execution and not beforehand. This can be alleviated through the introduction of an intermediate code format where much of the analysis is performed by a compiler beforehand. If designed right such an intermediate format can then be more or less directly interpreted without the need for many further analyses at run time. Such an interpretable intermediate format is usually called a *bytecode* format. Interpreted code is often more portable than compiled code. Since compiled code is generated with one specific processor hardware and operating system in mind it can only really be used in that exact combination. Interpreted code on the other hand can be used on potentially any hardware as long as a compatible virtual machine (VM) exists.

JIT compilers can be used to augment a VM by allowing it to compile parts of the program as needed. Interpreted code is relatively slow compared to compiled code. If we are able to compile just the parts of a program that are heavily used we can retain the benefits of an interpreter while also benefiting from the performance of compiled code. Portability can also be retained as when the code is compiled during execution it can be compiled for the exact environment it is running in at that moment. JIT compilers can also make use of analyses performed on the code running in the interpreter. This information can then be used for optimizations that would not be possible using only static analysis methods available to classical compilers [21, 25]. This is especially important for dynamically typed languages where type information for the variables used in a program is not explicitly available and may only be known at run time [17, 21]. Here the program can be initially interpreted and the type information can be recorded. This is known as a *warmup* phase. We remain in this warmup phase until we are confident enough that we have enough information for a compilation. But modern JIT compilers usually do not immediately compile a function as soon as sufficient data is available. Instead they wait with compilation until a function is considered *hot* (*i.e.,* heavily used). This is done to prevent unnecessary compilation of rarely used functions where the potential gains from compilation would not justify the added compilation overhead. If a function is identified as hot we can then compile it and optimize the compiled code to work with the recorded types only. If multiple combinations of types have been observed it is also possible to generate multiple compiled versions (or *specializations*) of the code — one for each configuration. At run time we can then simply choose the version we need or generate new specialized versions as needed [21]. It is also possible for JIT compilers to use multiple optimization levels. In a such a setup an initial compilation emits unoptimized code. This minimizes compiler overhead for rarely used functions. More heavily used functions can then be recompiled with increasing levels of optimization where the added overhead is justified by the potential performance gains [25].

2.2 Profilers

A profiler in general is a system that collects *run-time profile information* about a running program. These can include for example run times for different functions, observed variable types, memory usage. Profilers can be used for program analysis where the data is collected at run time and can then later be processed and evaluated. Profilers can also be used to provide information about the running program to its runtime environment itself. Such a profiler may for example provide data to a JIT compiler to improve the compiler output.

Profilers can be roughly broken down into two categories: Many profilers rely on instrumentation code others on sampling for their data collection [26]. We will discuss each category in the following sections. It should however be noted that these two approaches are not mutually exclusive but can be used in combination with each other [25].

2.2.1 Instrumenting Profilers

Most profilers rely on instrumentation [2, 5, 14, 25, 26]. That means that they insert additional code into the running program. This code can be inserted at specific points where information should be collected. The inserted code will — when executed — record some information about the program. For example if you want to measure the time a program spends in each function, instrumentation code can be inserted at the very beginning and at every exit point of the function to measure the time that expired between these two points.

As noted by Whaley a fundamental problem with this approach is that the frequency at which the profiling code runs — and thereby its overhead — is entirely controlled by the profiled program [26]. If we use the hypothetical profiler described above in a program with very long functions we have a relatively small overhead. But if we use the same profiler in another program with many short functions we have a lot of transitions between functions and because of that an increased overhead. It is also not possible to dynamically control the profiling frequency as it is baked into the profiled code itself and changing the profiling frequency would require the code to be — at least partially — replaced.

Instrumenting profilers are also very heavyweight and therefore not really applicable for general use. Depending on the implementation they are also often less precise than sampling profilers as they can only record information at the exact point where the instrumentation code has been injected [20]. Additional problems are introduced when the profiled program is multithreaded: Profilers often use a centralized data structure for their data recording. When code running on multiple threads is instrumented the profiler implementor must ensure thread safety of the data structure. This can also introduce additional profiling overhead and further decrease performance.

2.2.2 Sampling Profilers

Other profilers use a sampling approach [3, 25, 26]. Sampling profilers do not use instrumentation code injected at specific points in the program. Instead they use some mechanism to interrupt the program execution at random points and record the information available wherever the program was interrupted. As such they are using a probabilistic approach. As with this approach the sampling rate is not coupled to the running program it can be freely controlled and the profiler will show a relatively constant overhead. Sampling profiles can show sufficiently low overhead to justify their general use even in production environments. The data recorded and therefore their overhead varies highly between different profilers. Some walk through the entire stack [18] while others only look at the currently running function [13].

Existing profilers use different approaches to trigger a sample. Some use system timers that send a signal to the program at regular intervals. Others use hardware performance counters to send signals to the program. In both cases the profiling action is performed inside a signal handler. Such an approach is useful when the profiling action cannot be performed concurrently with the main program: While the signal handler is running the program is effectively frozen in place and the profiler may inspect any data structures it desires without potential concurrency issues. But interrupting the program repeatedly increases the overhead of the profiler as the program is not progressing during the profiling action. Other profilers that are capable of performing their actions concurrently with the main program can work by using an independent profiling thread using a simple busy loop [26]. This reduces the overhead by removing the need for signal handlers and therefore regular program interruptions.

2.3 Feedback-Directed Optimization

As described by Smith, feedback-directed optimization (FDO) is generally used to describe techniques and strategies used to alter the execution of a program based on tendencies observed during the program execution [24]. This may be information from the current run or even from previous runs. With this broad

view of FDO most JIT-compilers are on their own already an FDO technique. Compiling interpreted code significantly alters the execution of a program and most modern JIT compilers include a warmup period that is used to gather information about the running code. These compilers therefore use previously collected information to influence the compiler output. Going by Smith's definition of FDO any reoptimization event triggered by some information about the running program is also an FDO event as there must have been something in the running program that motivated a reoptimization event to be triggered and that reoptimization is used to further alter the execution of the running program. Smith also notes that FDO does not need to be based in software but some hardware implementations can also be considered to practice FDO. For example modern hardware often handles register allocation or branch prediction in hardware. These can in a broad sense also be considered FDO techniques as they too influence and change program execution based on previous execution. While not listed by Smith any form of speculative execution can with the same reasoning also be considered an FDO technique. The main benefit of FDO as noted by Smith is that it allows existing code to change:

- Optimizations can be applied whenever new information becomes available to improve performance for future runs. New information may allow a JIT-compiler to apply further optimizations that it could not show to be valid or beneficial during the warmup phase.
- New optimizations can be introduced to adapt to new situations: A program can often be divided into several phases with distinct behavior [24]. It is very possible that optimizations that are beneficial to one phase may not be as beneficial or may even be detrimental to the next phase. FDO techniques allow the detection of these phase transitions and triggering of reoptimizations beneficial to the next phase. This way optimizations must no longer be static and globally valid but are allowed to change over time to adapt to the running program. The information needed to determine when such reoptimizations should be done can for example originate from a profiler continuously running in the background [5].

2.4 R and Ř

R is a programming language mainly used for statistical and computational data science applications [9]. GNU R — the reference implementation — is a virtual machine including a bytecode compiler that compiles the raw R source code into an interpreted bytecode. R code usually only runs in a single thread[1] and is lazily evaluated. The R environment also supports the use of libraries, that is existing software packages that may be used by the main R program. These may be implemented in R itself or in C or Fortran.

R is very hard to compile mainly because of — but not limited to — its rich reflective interface that allows modification of most runtime structures [8]. For example it is possible for a function to modify or even add or remove local variables of the *calling* function. This makes compilation very hard as basically everything can be modified at any time.

Ř[2] is a relatively new JIT compiler for R [8]. It integrates into the GNU R virtual machine and effectively replaces the existing bytecode compiler and logic for function evaluation. It introduces a new single static assignment (SSA) [22] intermediate representation (IR) called PIR.

[1] native libraries may use multi-threading internally
[2] https://github.com/reactorlabs/rir

2.4.1 Function Lifecycle in Ř

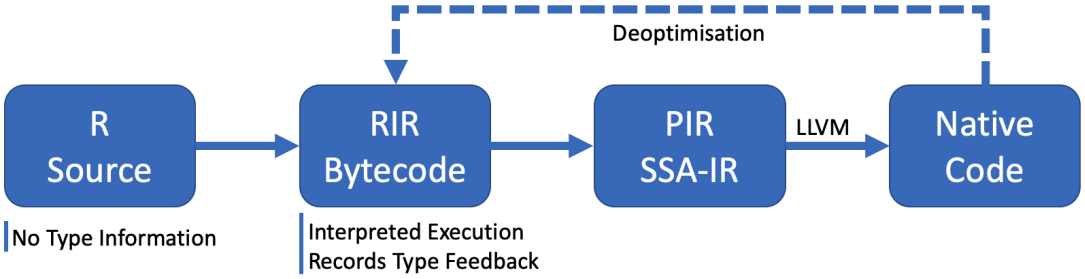


Figure 2.1: Graphical representation of the Ř compilation pipeline.

Function Declaration

Newly declared functions are compiled into a first intermediate representation called RIR. This first IR is similar to the standard bytecode used by GNU R.

Warmup

Initially when a function is called the RIR bytecode is directly interpreted. This is similar to the way GNU R works normally. The main difference is that during these interpreted function executions information about the function is gathered using instrumentation. This includes type information for variables used by the function and other information that is relevant for JIT compilation. For example during the RIR interpretation it is determined whether there are some values that are always evaluated before the function call or the function must lazily evaluate that value. This gathered information is attached to the instrumentation instructions in the RIR bytecode itself. These instrumentation instructions serve as annotations following the generation of a new value and contain the information about the possible types that value may have. An example of this is shown in subsection 4.2.2.

Compilation

When a function is hot (*i.e.,* heavily used) compilation is triggered. The RIR bytecode is passed to a JIT compiler. This compiler raises the RIR bytecode to the PIR IR. From there it is compiled to native code using an LLVM backend. The current implementation considers a function to be hot and compiled on its third call.

Compiled Execution

Following compilation all subsequent function calls are executed using the previously compiled version. The compiled code has assertions added at critical points that validate that the compiled code supports for example the values passed to it. If for example a function has been compiled to only support already evaluated parameters but now suddenly receives a parameter that is not yet evaluated that assertion would no longer hold. Any violation of such an assertion would lead to a deoptimization event:

Deoptimization and Recompilation

When an assertion in the compiled code is violated execution cannot continue and falls back to interpreting the RIR bytecode directly. After a deoptimization event we start a new warmup phase. It is to be noted that this second warmup phase does not start completely from scratch but uses the information already gathered in previous warmup phases. But because some type information or assumptions present in the RIR code were obviously violated the assumptions get relaxed (*i.e.,* no longer valid assumptions are removed and too narrow type information will be extended to also cover newly observed types). After completing the necessary number of warmup runs the function can then once again be compiled using the Ř compiler.

2.4.2 Boxed Values in Ř

Since R is dynamically typed, any variable may hold a value of any type and the variable type may change at any time. Since the variable itself does not give any information about the type of its value, the value cannot be simply stored as a raw value: The type information must be attached to the value itself. This is done through the use of *boxes*. Boxes are simple objects containing the type information and the value. In the GNU R implementations this is not an issue as the programs are fully interpreted and the virtual machine works solely with boxed values. Even primitive values don't really exist as such. These are represented as one-dimensional vectors.

But in Ř this is an issue because if boxed values could be *unboxed*, (*i.e.,* used as a raw value) more optimizations may be possible: When it is known that a value only ever takes on one type, this value can be stored as a raw value. Once compiled all operations on raw values could be performed directly with intermediate values being stored in CPU registers. Only the final result would need to be boxed for further use outside the function. But when values cannot be unboxed (*e.g.,* because multiple types have been encountered) such optimization is not possible and even intermediate values need to be boxed. This creates some overhead because each calculation needs to create a new box. Additionally this increases the load on the garbage collector as unused boxes need to be destroyed.

3 Problem Statement

In this section we will introduce the problem we are trying to solve. Then we will highlight some existing work that is in some aspects related to this thesis.

3.1 Problem Description

In Ř type information collected in the RIR bytecode is persistent: The type information is recorded into a simple structure essentially serving as a bit set. Each so-called `TypeFeedback` object basically consists of a collection of flags indicating all types that a variable has been seen to hold. It simply indicates to the Ř compiler what types have to be supported but does not track any information about how often each given type has been seen or how recently a given type has been observed. This presents an interesting problem when the type of a variable suddenly changes.

But where does such stale feedback actually originate from in Ř? If we return to the example in section 1.1 we can see that one such origin can be changing global state. R is designed to be used as an interactive environment. During an explorative session it is very common for values to change and for functions to be executed repeatedly with different values, or even variable types, present. In such a scenario the traditional idea of a warmup phase is no longer really applicable as the program execution never stabilizes.

A second very common problem in R programs is accidental polymorphism. Consider for instance the following implementation of a counter using a closure:

```
counter <- function() {
  count <- 1L
  function()
    count <<- count + 1
}
```

The variable `count` was intended to be an integer. This is suggested by the `1L` suffix. However, (accidentally) on the first increment a floating-point `1` is added. This converts the variable `count` to a double from then on. Therefore, the observed types of `count` are integer and double. But dynamically the integer only

occurs up to the first invocation. From then on, the function might as well be specialized for the double case.

A third very common case for erroneous feedback is not really a case where the feedback is stale, but instead a case where the dynamic analysis is not precise enough and merges unrelated profiles. A typical situation occurs when polymorphic library code is called from different locations. If we consider the following `add` function:

```
add <- function(a,b) a+b
add(1,2)
add(c(1,1),c(2,2))
```

The function `c` is a standard helper function in R that builds a vector containing the arguments passed to it: The expression `c(1,1)` builds a two-dimensional vector with both elements having the value `1`. As can be seen, the `add` function is called both with scalars — `add(1,2)` — and vectors — `add(c(1,1),c(2,2))`. Therefore the profiling information for `add` records both arguments to be either scalar or vectors. When this `add` function is inlined, the PIR optimizer imports type feedback from the inlinee into the caller. Therefore at both call sites `add(1,2)` and `add(c(1,1),c(2,2))` the merged type feedback of both invocations will be imported. The core problem is that in general type feedback lacks dynamic context. Of course in this simple example the variable type can be inferred from the static context. But in general this is a problem that can easily be observed in practice and does have negative effects on performance. It is one of the reasons for the appeal of three-tier architectures, used by some systems. The second tier performs inlining, and at the same time still features instrumentation to record profiles. This allows a virtual machine to record context sensitive profiles for inlined functions. But the Ř VM lacks this architecture and only uses two tiers: an interpreter with instrumentation and native code without instrumentation.

To demonstrate how the Ř VM could benefit from our profiler we will take a look at a concrete benchmark: In the following listing the flag `POLLUTE` causes the function `f` to be invoked twice with the global variable `x` being a double. All subsequent invocations are on integers.

```
f <- function() x+x+x+x+1L
if (POLLUTE) {
  x <- 1
  f(); f()
}
x <- 1L
for (i in 1:1000000) x <- x+f()
```

When the `POLLUTE` flag is unset, then this snippet executes 25% faster in Ř, due to the variable `x` being fully unboxed in `f`. If on the other hand, the `POLLUTE` flag is set, the variable `x` cannot be fully unboxed as it may hold a double or an integer.

3.2 Related Work

Several works have combined a profiler profiling compiled code with a JIT compiler in what is essentially a feedback-directed optimization loop:

- Arnold et al. use a low-level instrumenting profiler to build a control-flow graph for a Java program. This information is then used for traditional optimizations in a JIT compiler [5].
- Whaley has created an instrumenting profiler for Java programs that can be continuously used [26]. It builds profile information that is fed back to a dynamic compiler.

- An interesting hybrid solution combining different techniques is presented by Suganuma et al. [25]: They use a sampling profiler to decide where to add or remove instrumentation code dynamically. This instrumentation code is then used to collect profile information that is provided to a JIT compiler for additional optimizations. Their approach yields modest performance improvements at minimal cost.
- Kistler and Franz propose two techniques for continuous program optimization [16]. One approach attempts to continuously improve cache usage to minimize cache misses over time. The second approach reschedules instruction to improve instruction-level parallelism. They also issue a reminder that continuous optimizations have to be well-tuned and used sensibly as it is easy for the gained performance to be entirely consumed by the optimization mechanisms that caused them.
- Ammons et al. have created a system where an instrumenting profiler is used in combination with hardware performance counters [2]. While the profiler performs a relatively standard data flow analysis this information is augmented with the data from the hardware performance counters for further optimizations.
- Schneider et al. also created a solution where hardware performance measurements are used to guide optimizations in a Java JIT compiler [23].
- Zaparanuks et al. and Moore have both performed extensive research into the accuracy of hardware performance measuring equipment. While Moore compares different configuration modes [19], Zaparanuks highlights potential problems with the overall accuracy of hardware performance measurements [27].

4 Implementation

In the following sections we will present the concepts behind our implementation. We will start with the way we trigger the profiler to record samples. Then we will cover how these samples are recorded and what they contain. Then finally we will explain how we decide when to recompile a function and how the recorded data is used by the compiler in order to improve the compiler output.

4.1 Sample Triggering

There are multiple solutions for triggering the recording of a sample. Since we desired random samples we had to find a trigger that is not dependent on the instructions being executed at a given time. An instrumentation approach is therefore not applicable. We also cannot simply have a second thread that continuously records information about the first thread as this would lead to many synchronisation issues at various locations inside the Ř virtual machine. An obvious approach at this point was the use of timers and signals.

4.1.1 POSIX Timers

The trivial approach to repeatedly running code independent from the running program is to use a timer that periodically issues a signal. That signal interrupts the running program and jumps to a signal handler responsible for recording a sample. While this approach seemed promising early on, we quickly identified some potential issues that can appear whenever the program performs a system call. UNIX system calls can be broadly grouped into 3 types [12]:

- *non-blocking* (*e.g.,* `getpid`): These system calls always return quickly and are not affected by signals.
- *blocking restartable* (*e.g.,* `wait`): These system calls may block for an indefinite amount of time and are aborted by a signal. They can however be automatically restarted upon return from the signal handler. This restart is usually transparent to the issuer of the system call.

- *blocking non-restartable* (*e.g.,* `pause`): Just like the restartable system calls these may block for indefinite amount of time and are also aborted by a signal. They can however not be automatically restarted and fail with `EINTR` (interrupt error) upon return from the signal handler.[1]

For our use case the blocking system calls pose a problem as non-restartable system calls will be aborted by our signals. This may change the behavior of the program in a way we cannot influence. The system call `pause` for example suspends the program execution until a signal is received. We could conceivably patch the places performing system calls from within the Ř back-end in order to fix the altered behavior but there is no such solution for system calls performed by R packages based on compiled code.

Restartable system calls also pose a problem to us as we have no way of knowing for how long they will block upon restarting. If we ever encounter a situation where we interrupt a blocking restartable system call once and after restarting it remains blocked for longer then our sampling interval we will just interrupt and restart it repeatedly. In this situation the program may starve simply because we never let the system call complete. Here we could disable the signal handler before performing the system call and re-enable it after the system call completes. But once again we are in a position where we could patch Ř but cannot do anything about R packages. A potential solution is to detect when we are stuck in a system call. It is possible to register a signal handler in such a way that it is provided with additional information including the user time and system time consumed [11]. This information could be used to implement a back-off strategy: If we detect that we are trapped inside a system call we temporarily increase the time interval between samples until the call completes. While in principle solving the starvation issue (when repeatedly increasing the time interval there must come a point where it sufficiently long for the system call to complete) in many cases this is not really a viable strategy. A simple example is a program waiting for input on `STDIN`. If there is no data available the `read` call will block. Now if for example the user needs even just a few seconds for the input our back-off strategy would already have increased the sampling interval to several seconds between samples. While solving the starvation issue this does nothing to solve the issues of interference with non-restartable system calls.

4.1.2 Hardware Performance Monitoring Unit

An alternative to the use of POSIX timers is to use the Performance Monitoring Unit (PMU) present on modern Intel processors. The PMU allows monitoring of a multitude of information on the performance of a processor — *e.g.,* counting the number of retired instructions. The PMU was first introduced in Intel Pentium processors in 1993 and expanded in capability over time. All Intel Core and Intel Xeon processors since the *Nehalem* architecture (released in 2008) and all Intel Atom processors provide a very capable PMU with most modern-day features supported [15]. Other processor manufacturers offer similar capabilities. For example AMD Zen processors provide Performance Monitor Counters (PMC) [1] and some ARM processors also provide similar capabilities [4]. In the following sections we will simply refer to this hardware as the PMU as the development and evaluation have been done with Linux on Intel processors. The content should however be transferrable to other CPU architectures as well.

Counting and Sampling

To enable modern PMU features the PMU is very configurable. On Linux the configuration is performed through a system call [10]. In general the different configurations can be categorized into 2 operational modes: *counting* and *sampling* [19]:

- *counting mode*: In this mode the PMU simply counts simple events occurring in the CPU. The exact type of events counted can be configured. Possible configurations include counting of retired

[1] `sleep` is an exception and terminates successfully even when aborted through a signal, but returns the time remaining.

instructions or number of cache misses [2]. The data is collected in simple counters on the CPU and can be retrieved through polling via a system call.

- *sampling mode*: In this mode the PMU can record information about a running process on the CPU. This may be simple information like in the counting mode, or it may be more complex metrics. The main difference to a counting configuration is that the recorded data is not polled by the program itself but written into a buffer. This buffer can be either read by the program (polling) or it can be configured to send a signal to the program whenever the buffer is written to.

  The sampling mode is an extension of the counting mode: The PMU is still configured to count some metric. Additionally a *sampling period* is configured and a sampling configuration is added. Whenever the counter exceeds the value set as the sampling period a sample is taken and written to the buffer. This may for example be a timestamp or the current instruction pointer. The PMU holds a counter indicating how many samples should be taken. This counter starts positive and decrements with every recorded sample. When it reaches zero the PMU will no longer take samples. Initially this value is set to zero. To manipulate this value the PMU can be refreshed with an integer value. This value will be *added* to the value present in the PMU. By initially refreshing the PMU with a value of -1 the sample counter will be set to the maximum value (usually 2^{40}). Assuming a 40-bit counter, the PMU will support ca. 1.1×10^{12} signals. Even assuming 1 sample per microsecond this counter would not run out for 12.7 days and would in most cases never require subsequent refreshes (unless of course a program would run for longer than that).

Use for Sample Triggering

Since we want the PMU for sample triggering we need it to run in the sampling mode even though we are not actually recording any real data. We can configure the PMU to generate an event whenever a given number of instructions are retired. In order to use the sampling mode we need to chose something to be sampled: We simply chose to sample the current instruction pointer. While this data is irrelevant to us we use it to force a change in the sample buffer that we can detect and use to generate a signal. It is also possible to configure filters so that events issued while the CPU is executing instructions in kernel or hypervisor mode are ignored. This way we only generate samples while executing code in user mode [10]. Given our configuration (Listing 1) samples should be recorded immediately when the instruction counter exceeds the predefined value. If we assume that the signal generated by the sample is issued as soon as the data becomes available the signal should not interfere with system calls. Listing 1 shows how the PMU can be configured for our use case.

Security Considerations

One potential problem with the PMU-based approach has to be noted: On some Linux distributions PMU access is disabled for users by default for security reasons. It is possible to attach the PMU to a process or thread other than your own. This imposes a security risks as a malicious program monitoring a running process may potentially be able to extract sensitive data from the monitored process.

4.2 Sampling Native Code

In this section we will outline how we detect valid sampling locations and how samples are actually taken. The profiler is triggered at random points in the program. As a consequence not every time the profiler is triggered will we be able to record a sensible sample. We are only interested in native code produced by the Ř JIT compiler. If at the point of the sample we are executing interpreted RIR code or even library code we will not be able to take a sample. We must therefore be able to detect valid sampling locations.

```
static void handler(int signal) {
    /* signal handler */
}

static void initProfiler() {
   // Setup signal handler
    struct sigaction sa = {};
    sa.sa_handler = handler;
    sa.sa_flags = 0;
    // register handler for SIGUSR1
    if (sigaction(SIGUSR1, &sa, NULL) < 0) {
      exit(EXIT_FAILURE);
    }

    // Configure PMU
    struct perf_event_attr pe = {};
    pe.type = PERF_TYPE_HARDWARE;
    pe.size = sizeof(struct perf_event_attr);
    // count retired hardware instructions
    pe.config = PERF_COUNT_HW_INSTRUCTIONS;
    // disable PMU by default
    pe.disabled = 1;
    // sample the instruction pointer
    pe.sample_type = PERF_SAMPLE_IP;
    // sample every 100k instructions
    pe.sample_period = 100000;
    // ignore kernel-mode instructions
    pe.exclude_kernel = 1;
    // ignore hypervisor
    pe.exclude_hv = 1;
    // force the sample to be taken exactly when triggered
    pe.precise_ip = 3;

    // initialize PMU and verify
    int fd = perf_event_open(&pe, 0, -1, -1, 0);
    if (fd == -1) {
        exit(EXIT_FAILURE);
    }

    // Setup event handler for PMU samples
    fcntl(fd, F_SETFL, O_NONBLOCK | FASYNC);
    // trigger SIGUSR1 when a new sample is ready
    fcntl(fd, F_SETSIG, SIGUSR1);
    fcntl(fd, F_SETOWN, getpid());
    // reset the event counter to 0
    ioctl(fd, PERF_EVENT_IOC_RESET, 0);
    // enable and allow maximum number of events
    ioctl(fd, PERF_EVENT_IOC_REFRESH, -1);
}
```

Listing 1: `profiler.cpp` (extract showing PMU configuration)

4.2.1 Detecting Native Code

The task of the profiler is to observe and record types of boxed values inside native code originating from the Ř compiler. It is also important that we not simply sample whenever native code is being executed but only when that native code actually originated from the Ř compiler. There is no trivial way to detect from within a signal handler if the currently running code is such code. To achieve this we decided to use a relatively simple approach: We needed to add a marker into the stack frames of code emitted by the Ř compiler. Additionally the profiler needs to access the code object representing the currently running code. Each compiled function has a code block object associated with it. It is possible to use that code object itself as the marker. We can therefore simply pass the code block of a function into that function. To achieve that the Ř compiler backend was extended. At the start of each code block the PIR code block object representing the currently running code would be pushed into the first slot of the current stack frame. In order to detect relevant code the signal handler must simply take the current stack frame and test if the first slot contains a code object. If so, we have found relevant code and can start taking a sample. If not the profiler simply returns and the program can continue execution.

4.2.2 Stack Slot Mapping

There is a certain disconnect between the original RIR bytecode and the native code emitted by the Ř compiler. While the RIR code contains many instrumentation instructions to record type information, not all of them are carried over into the PIR. And upon compilation using the LLVM backend these instructions are erased completely and transformed into type assertions in the native code.

One aspect of the way Ř currently works is that boxed values are always located on a shadow stack. In the current implementation stack slots are never reused. This simplifies compilation as liveness of different variables does not have to be considered. Once a value has been written to the stack it will remain there until the function terminates. If a value on the stack is modified the new version is assigned a new slot. We were able to leverage this implementation detail to create a simple mapping. Since stack slots are never reused and slot assignments are static the compiler can determine the associated type feedback in the original RIR code. We extended the Ř compiler to build a mapping indicating for each stack slot whether — and if so, where — a corresponding type marker exists in the RIR code. This mapping was attached to the code object containing the compiler output. This way that mapping would be accessible to the profiler as that code object will be present on the stack.

If we take a look at the function `f` as presented in section 3.1 we see that this is compiled to the following RIR code:

```
 0   ldvar  x
 9   [double(s), integer(s)]
14   ldvar  x
23   add
     ...
```

At offset 9 there is a type recording instruction and it has recorded the top of the operand stack to be either a double scalar or an integer scalar. When this function is optimized in PIR, the variable `x` loaded from the global environment is speculated to be an integer or a double. The optimized function is also annotated with a map as metadata, associating slots in the shadow stack, with original type feedback locations. In this case after compilation the map contains the following entries:

```
- #2->9 : [<?>] (0), [dbl(s), int(s)]
- #3->9 : [<?>] (0), [dbl(s), int(s)]
- #4->29: [<?>] (0), [dbl(s), int(s)]
- #7->81: [<?>] (0), [dbl(s), int(s)]
```

For instance, the first row reads as follows: In the second slot of the shadow stack, we find a value that corresponds to a type feedback that was recorded at bytecode offset 9 (see previous listing). Since the code has not yet been run we were not yet able to collect any profile information. This is shown by the sequence before the comma indicating an unknown type with zero samples taken. But the type feedback used during compilation (listed after the comma) shows that in each slot we can expect doubles and integers.

4.2.3 Recording a Sample

In order to limit overhead we only look at the boxed values of the currently running function. While sampling we can go through all slots in the current stack frame and record the type of the value present there. While we have no guarantee that any slot has already been written to it is trivial to check this during the sampling as empty slots are zero-initialized and after a value has been written to the stack it can no longer hold a value of zero. This is because the stack holds references to boxed values. And once a reference to a box has been written the reference will no longer be a null-reference. Since each code block may be specialized for different call (or inlining) sites we keep track of the recorded types for each specialization separately by storing the data in the code block object associated with that specialization.

While we could record the types of all non-zero stack slots we only require records for those that are associated with a type marker in the RIR code. These are the only records that we can use in the JIT compiler. we could record the other slots as well but this does not yield any benefit to us. We therefore check for each slot if we find a type marker in the previously constructed mapping and only record the type if the slot is present in that mapping.

There are some situations where multiple stack slots may be mapped onto the same RIR type marker. For example in our function `f` we see two slots that are both associated with type feedback recorded at bytecode offset 9:

```
- #2->9 : [<?>] (0), [dbl(s), int(s)]
- #3->9 : [<?>] (0), [dbl(s), int(s)]
```

These situations could create ambiguity in the compiler and therefore we needed to find a solution for this: We created a secondary mapping that mapped each RIR type marker onto one so-called `TypeFeedback` object. We chose to use the same internal representation for your type records as was already used by the interpreter. This representation is already well-optimized. Since all our profiling happens inside the signal handler it was important to keep the performance impact as low as possible. Therefore it seemed logical to use the same representation that was already deemed sufficiently performant for the interpreter itself. With the use of this secondary mapping we can ensure that all slots associated with a single RIR bytecode offset are recorded into a single `TypeFeedback` instance.

If we go back to the function `f`, after some iterations we may see the following profile:

```
- #2->9 : [int(s)] (4), [dbl(s), int(s)]
- #3->9 : [int(s)] (4), [dbl(s), int(s)]
- #4->29: [int(s)] (2), [dbl(s), int(s)]
- #7->81: [<?>]    (0), [dbl(s), int(s)]
```

Now for some slots we were able to record type information. In the first three slots (2, 3 and 4) we were able to record a scalar integer twice. Because slots 2 and 3 are both associated with the same RIR bytecode offset they are mapped onto the same `TypeFeedback` instance. As a consequence, that instance has twice as many samples as the instance for offset 29 that was used by slot 4. As can be seen in this example, during optimization it is possible for a value to appear multiple times (in which case said values will be aggregated when taking decisions), or to be absent from the table if it cannot be traced back to the original type feedback. For some slots we will be able to collect many samples; for others, the samples will be sparse or even completely missing.

4.3 Triggering Recompilation

Once sufficient data have been recorded we need to make sure that the function is recompiled. But first we have to define what we consider to be *sufficient* data: If we recompile too soon we may have incomplete data. This could lead to a deoptimization event soon after recompilation. Such a situation must be avoided because it would significantly reduce the program performance.

We needed to define a sample count T_1 that could be considered sufficient for recompilation. After recording a sample we check each slot for its sample count. If this number exceeds T_1 samples, the slot is marked as *ready*. It is obvious that there exists a trade-off here: While lower values of T_1 lead to slots being *ready* more quickly this also increases the risk of having incomplete information. Higher values increase the chance for all possible types to be recorded at least once but doing so also delays slot readiness.

Due to the random nature of our sampling there is also an imbalance in the sample count for different stack slots: Stack slots become live at different points in time but remain live until the function terminates. As a consequence of this early stack slots will naturally have more samples than late stack slots. If, for example, a slot becomes live after 30% of the execution time of a function, it remains live for the remaining 70% of the time and will therefore be present in approximately 70% of the samples.

The imbalance between the sample counts of different stack slots cannot be prevented. We must make sure that we do not recompile the function if only a few slots have sufficient data while all others still have no or insufficient data. Not doing so could again lead to overly eager recompilation with insufficient data. We needed to find a simple way to decide for each function if the recompilation effort was worth it. Global variables are usually pushed to the stack at the beginning of the function. This means that they are sampled with basically every sample. The remaining slots hold intermediate values of calculations in the function or the return value. They become live at different times during the execution. A simple approach for deciding when a function is *ready* for recompilation is to define a fraction of slots that need to be *ready* in order for the function to be *ready*.

The ideal fraction is obviously different for every function: If a function only uses global variables and parameters for the calculations, the types of all the intermediate values can be inferred from the slots that are live from the beginning. In this case the function should be *ready* as soon as all the initially filled slots are *ready*. Sampling intermediate values is not necessary. But if, for instance, a function calls other functions where the return type is not known sampling the very early slots may not be enough. Here the function should be *ready* as soon as the last slot that cannot be inferred from previous slots (*i.e.,* the slot holding the result of the last function call) is *ready*.

It is obviously impractical to tune the profiler for every function individually. We therefore need a general rule that works well enough with most functions. We chose to consider a function as *ready* when at least 50% of all slots were *ready*. This value should — in most cases — be high enough that sampling only the initially filled slots should not be sufficient to make the function *ready*. But it is still low enough that less frequently sampled slots late in the function that are not yet *ready* do not prevent recompilation based on the early slots. Late slots can often be inferred from earlier slots (*i.e.,* if they hold results of calculations based on the values in earlier slots) or optimizations could be done on much of the function even if some slots have types that are not well known. It is also set such that a majority of the types need to be known in order for a function to be *ready*. It is clear that our value of 50% is somewhat arbitrary. We have not selected this value experimentally. Much of the early testing was done with relatively simple functions where most types could be inferred from global variables alone. Further research could be done to find a better value but as we highlighted above any value will be somewhat arbitrary as any value would only be ideal for some functions.

To decide if a function is *ready* for compilation is not enough: Even if a function is *ready* it makes no sense to recompile it if the recorded type information is identical to the type information the function was initially compiled with. Doing so would — since the compiler is deterministic — produce the identical code that is already present. We therefore consider a function *ready* by the following rule:

> *A function is "ready" if at least 50% of all stack slots with type feedback are "ready"* — i.e., *have at least T_1 samples each — and at least one stack slot has recorded type information that differs from the type information used during compilation.*

When this condition is met, a flag is set on the function marking it for recompilation.

Preventing Pollution

The main job of the profiler is to detect polluted type information in the RIR code by finding. To accomplish that goal we must make sure that the type information gathered by the profiler cannot be itself polluted. Polluting the profiler type information would make it useless. The source of this pollution is stale type information that does not get removed: If we recorded one type just a single time but never again we should discard that type as it obviously is no longer relevant. But because we use the same infrastructure for tracking the recorded types as is used in the RIR code we are unable to count the number of occurrences of each type and the time since the last occurrence. To solve this issue we decided to follow the trivial approach. If a stack slot shows a high sample count that means that while that slot is *ready* the function as a whole is not *ready*. This may be either because an insufficient number of stack slots are *ready* or — more interestingly — because the recorded types match the compile-time types completely. This is where potential pollution comes in: When the types all match because they are actually all used it is correct that no recompilation has been triggered as it would not change anything. But it could also be that we see some pollution that matches the compile-time types. If we retain this recorded type information indefinitely we will never be able to recompile the function. But if we at some point decide to discard the type information and start a fresh recording we will be able to determine if the previous recording was correct or was in fact polluted. If after clearing the recorded type information we once again see the same types they are correct. But if after clearing some types no longer appear in the recordings they were probably just pollution. The act of clearing the type information alone is in this case sufficient to remove the pollution and allow eventual recompilation.

The challenge lies in deciding when to clear the recorded type information. If we clear too eagerly we may be clearing early slots before late slots were able to become *ready*. In this situation we just prevent sensible recompilations from taking place by preventing the function from becoming *ready*. But if we wait too long before clearing we also simply delay sensible recompilations. We introduced a second threshold T_2 and the following rule:

> *When a stack slot has more than T_2 samples while the function itself is not "ready", the slot is considered potentially polluted and is cleared.*

The exact value for T_2 is less important than that of T_1 and the ratio of stack slots that need to be *ready* for a function to be considered *ready*. It is important that T_2 is chosen large enough that early slots do not exceed it before the function is *ready*. Early stack slots exceeding it too quickly could prevent the function from ever becoming *ready*. But larger values are not as important as they would simply delay recompilation.

In our preliminary testing we observed that for simple functions (*e.g.*, the function `f` from section 3.1) early slots would not exceed $10 * T_1$ samples before the function becomes *ready*. Therefore we chose to scale T_2 together with T_1 with $T_2 = 10 * T_1$ for our evaluation.

4.4 Using Profiler Data

In the previous sections we have covered how we gather type information from compiled code. Now that we have introduced a flag on the code object marking *ready* functions we have to actually perform the

recompilation. There is already existing code in the Ř VM that periodically checks if the assumptions a function has been compiled with change. If that check passed the function would be recompiled and the existing version would be replaced with the new one. In this check we also already knew which code object the recompilation check was performed for. It was therefore trivial to simply extend the existing check to also include the newly introduced flag on the existing code object marking it as *ready* for recompilation. That way simply setting that flag in the profiler would trigger a recompilation relatively soon after setting it. But simply triggering a recompilation is not enough as the compiler would still use the same type information gathered during the warmup phase. It is obvious that in a deterministic compiler repeated compilations with the identical information will not yield any difference in the output.

As explained in previous sections we introduced a bidirectional mapping between the type information in the RIR code and the type information form the profiler. Using this mapping we can for each RIR offset retrieve the corresponding `TypeFeedback` object from the previously generated code. The existing compiler implementation already made a copy of all type information coming from the RIR. This copy would then be used for type-dependent optimizations and in the compiler. We extended this copying code to try and retrieve the corresponding type information from the previously generated code instead. If successful we would check the type information. If it is *ready* we would provide that to the compiler. If it is not *ready* we would provide the information from the RIR code instead. If no corresponding type information can be found we also provide the RIR information to the compiler.

The fact that not all RIR markers could be replaced with data from the previously generated code is an issue that we cannot solve: The further the optimizations go the more the native code distances itself from the original RIR version. If we have a long calculation like for example `x <- a * b + c`, a generic compilation would have stack slot assignments for `a`, `b`, `(a*b)`, `c` and `x`. It is also likely that the RIR would have type markers for all those values. If however the same statement were compiled only for integers, then only `a`, `b`, `c` and maybe `x` would have their own stack slots while the intermediate term `(a*b)` would very likely be simply stored in a CPU register. In this situation the RIR type marker for that value could not be corrected by the profiler. But this appears to be a non-issue as with sufficiently constrained type information on the input variables the compiler could very likely correctly infer the correctly constrained type for the intermediate values even when that information could not be provided externally.

5 Evaluation

To evaluate our solution we investigate three facets of our implementation, each of these targeting one aspect of the profiler. The first evaluation measures the overhead introduced by merely running the profiler. The second, attempts to find a sensible re-optimization threshold. Finally, the third evaluation highlights the examples where our profiles produces useful performance improvements.

The ŘŘ benchmark suite used in this paper consists of 46 programs that range from trivial code fragments to small algorithmic problems, and real-world code. Some programs are variants; they use different implementations to solve the same problem. This pre-existing suite was also expanded and three benchmarks were added to show potential performance improvements to be gained through use of the profiler.

To deal with warmup phases of the virtual machine (*i.e.,* iterations of a benchmark during which compilation events dominate performance), we run each benchmark fifteen times in the same process and discard the first five iterations. We ran experiments on a dedicated i7-6700K CPU, clocked at 4 GHz, stepping 3, microcode version 0xd6, with 32 GB of RAM and Ubuntu Bionic on Linux kernel version 4.15.0-88.

5.1 Overhead Evaluation

This first evaluation was used to measure the pure performance overhead introduced by running the profiler. To be able to perform these measurements we modified the profiler so that while it continues to record type information, that information is never used in the compiler and a recompilation is never triggered. Doing so, we avoid measuring potential compiler overhead caused by the profiler triggering a recompilation.

The measurements were performed with four different configurations: The first was a baseline run with the profiler completely disabled. This was followed by three runs with the profiler enabled with different sampling periods: one triggered a sample every 100,000 instructions, one with a sample every 500,000 instructions and the last one with a sample every 1 million instructions. For each benchmark in these runs the median run times were normalized against those of the baseline run. While the data is somewhat noisy we can still gather important information. In some configurations we see apparent improvements compared to the baseline values. These can be attributed to the inherent noisiness of the benchmark environment as

well as some additional noise introduced through the randomized sampling locations hit by the profiler and do not indicate actual reproducible performance improvements.

Figure 5.1 shows the profiling overhead for each benchmark for the three sampling periods with each data point representing an individual benchmark. With a sampling period of 1 million instructions the overhead is minimal. Most benchmarks had a slowdown of less than 3% compared to the baseline. Only 4 benchmarks had a slowdown above 3%. This can be seen by the 4 blue data points that lie above the dotted black line. The worst-performing benchmark was one called *Storage*. It suffered a slowdown of 9.9%. The mean slowdown over all benchmarks was just 0.6%.

With a sampling period of 500,000 instructions we observed very similar behavior as before, with only 5 benchmarks showing a slowdown above 3%. The worst-performing benchmark was once again *Storage* with a slowdown of 9.1%. The mean slowdown was 0.3%.

Using a sampling period of 100,000 instructions we see a significant slowdown compared to the previous configurations. Now, 19 benchmarks showed a slowdown of more than 3% and *Storage* even exceeded 10% slowdown at 12.7%. The mean slowdown was 2.7%.

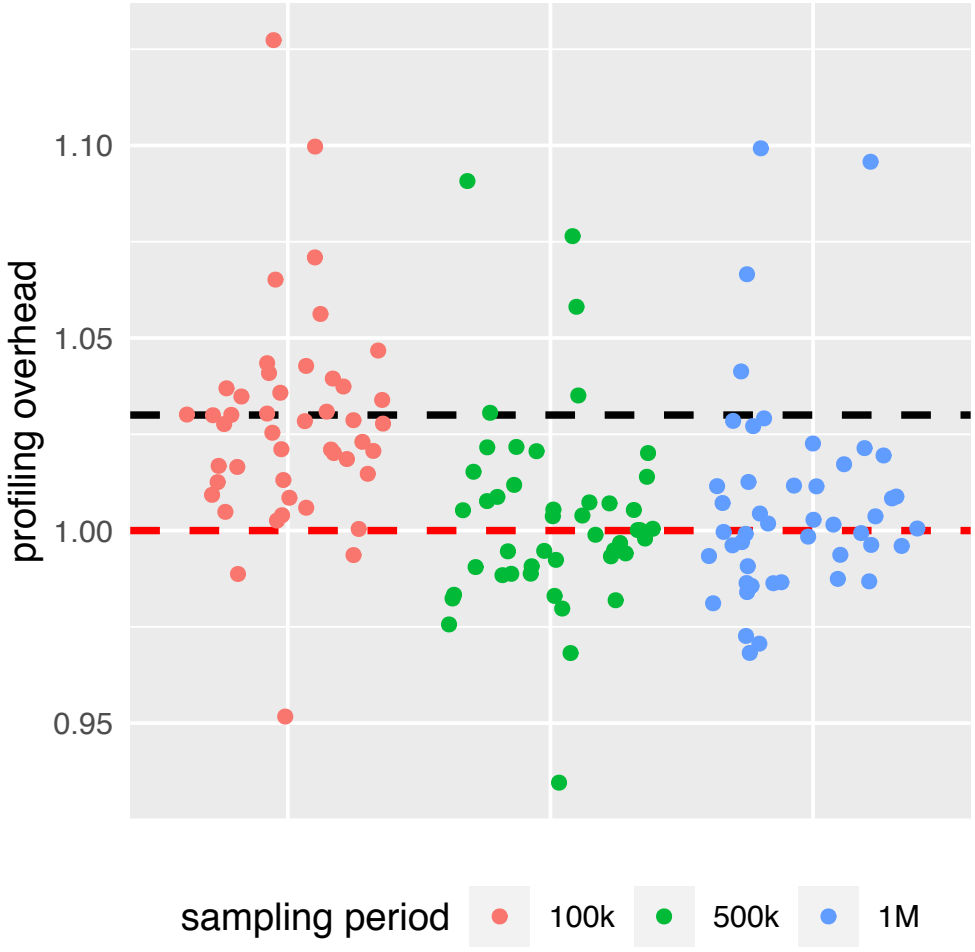


Figure 5.1: Profiling overhead per benchmark. (The dotted black line indicates a slowdown of 3%.)

5.2 Threshold Configuration

This second part of the evaluation was designed to find sensible recompilation thresholds T_1 for the different sample period configurations: Low recompilation thresholds allow for quick action by the profiler. This can improve performance in situations where the resulting compilation produces a good result. But with too low a threshold we introduce a higher risk of having incomplete data at our disposal when the profiler decides to trigger a recompilation. This can lead to the newly compiled version being incompatible with what actually is required at that time. This will invariably lead to subsequent deoptimization events. At too low thresholds this behavior can even become cyclic where the profiler repeatedly tries to step in and trigger recompilations but never manages to gather a complete picture before this happens. In such a situation we could observe extreme performance impacts. An obvious approach to preventing this situation is to use high thresholds. But while this helps to reduce detrimental recompilations it also delays useful ones. This reduces the performance gains we can expect in cases where the profiler can theoretically improve performance. We need to find thresholds that are large enough to avoid having too many detrimental recompilations while still keeping it as small as possible in order to maximize potential benefits.

We started with a very low threshold of 10 samples and raised it until significant performance impacts were no longer observed. This was done for the three sampling periods of 1 million, 500,000 and 100,000 instructions individually. The main difference in profiler configuration compared to the first part of the evaluation is that now the profiler was allowed to trigger recompilation and the compiler was allowed to use data collected by the profiler during compilation.

The collected data was then compared to the overhead measurements gathered in section 5.1. Each run that had a slowdown of more than 10% compared to the median run time of the overhead measurements was flagged as an outlier. Figure 5.2 shows the outlier frequencies for different thresholds. In general for the same threshold the higher sampling rate produced more outliers. We can likely attribute that to the fact that with higher sampling rates we effectively shorten the time required to reach the recompilation threshold. This shortened sampling time increases the risk of premature recompilation. Such premature recompilations lead to deoptimization events soon after. When this happens often enough it is even possible to have a recompilation and a deoptimization event in each run. This significantly impacts performance.

With a sampling period of 1 million instructions we observed 25 (5.4%) outliers with a threshold of 10 samples. This dropped to 12 (2.6%) outliers for a threshold of 50 samples. Increasing the threshold further does not seem to improve things: At 100 samples we recorded 14 (3%) outliers. In fact starting at a threshold of 20 no significant changes can be observed anymore.

In addition to the overall observations we can look at specific benchmarks for more information: The *nbody_naive_2* benchmark for example had a mean run time of 290 seconds and a median run time of 5 seconds with a threshold of 10 samples. These values dropped to 1 second each when using a threshold of 20 samples. This shows both a significant reduction in run time and increased consistency: With a threshold of 20 the closely matched mean and median indicate the absence of significant spikes. Of course the extreme outliers in the lower threshold must be mitigated by back-off strategies in practice.

There are however a few benchmarks where the increased threshold actually reduced the measured performance: For example the *reversecomplement_naive* benchmark had a mean run time of 323ms at a threshold of 10. This increased to 393ms at a threshold of 20. This is attributable to the fact that at larger thresholds a recompilation event is measured in the ten benchmark runs that were hidden inside the warmup phase for smaller thresholds.

Overall for a sampling period of 1 million instructions and thresholds starting at 20 we observed exactly one slow run that significantly increased mean run times. The median run times however remained essentially unchanged.

With a sampling period of 500,000 instructions the behavior is very similar to before. While at a threshold of 10 we observed 43 (9.3%) outliers this dropped to just 14 (3%) at a threshold of 20. And

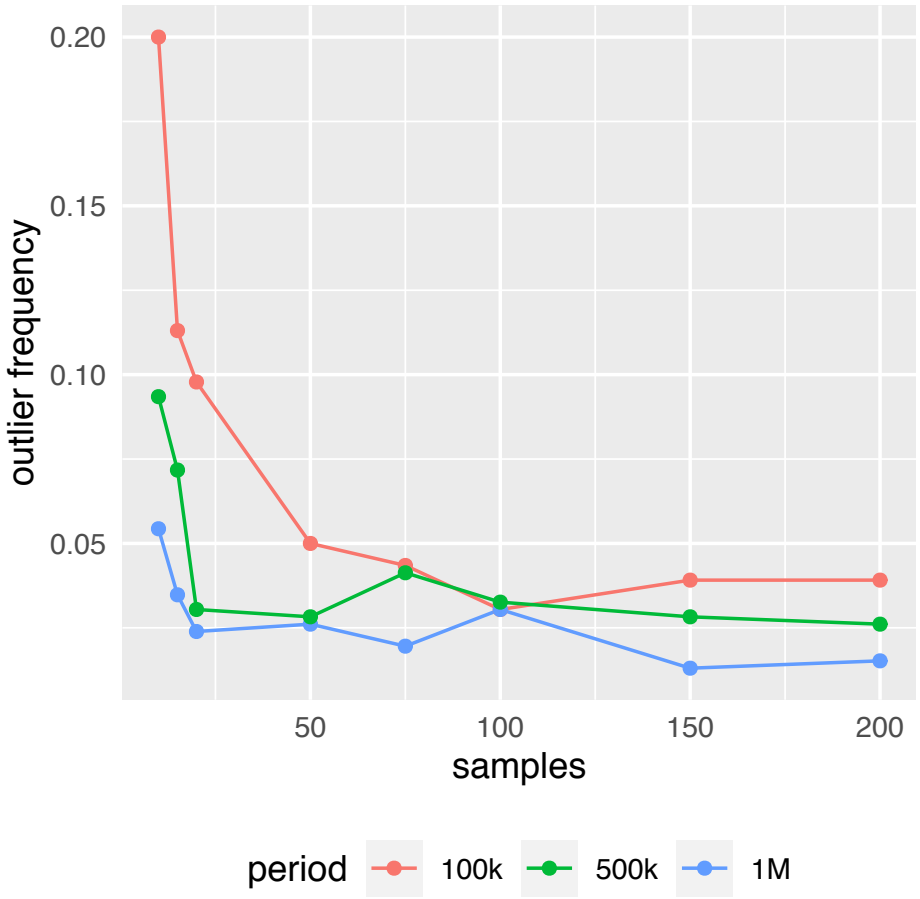


Figure 5.2: Outlier frequencies as fractional values relative to the total number of runs recorded.

above 20 samples we observed basically identical outlier counts as with a sampling period of 1 million instructions.

With a sampling period of 100,000 instructions we see a high outlier count of 92 (20%) for a threshold of 10. This drops down to 20 (4.3%) at a threshold of 75. At a threshold of 10 it behaves clearly worse than with a sampling period of 1 million instructions. Due to the significantly lower sampling period we will collect samples quicker. When using the same threshold we effectively make the profiler more eager in its optimizations since it only needs to measure over a shorter time frame.

5.3 Performance Improvements

The third part of the evaluation is designed to show potential performance improvements gained through use of the profiler. To that end we added three additional benchmarks to the Ř benchmark suite: *profiler_microbenchmark*, *profiler_rsa* and *profiler_shared*.

The *profiler_microbenchmark* is essentially the same as the example function `f` in section 3.1. It is designed to test a situation where a single stale type was introduced before compiling the function. Here

we start with a floating-point number. But after two calls we change to an integer. The function will initially be specialized to both integer and floating-point numbers. The profiler should detect that only integers are present from now on and should at some point trigger a re-optimization.

The *profiler_rsa* benchmark contains a simple RSA encryption implementation with small key numbers:

```
p1 <- 971
p2 <- 383
n1 <- p1 * p2        # floating-point
e <- 17
encrypt <- function(msg) {
  p <- 1
  a1 <- msg
  for(i in 1:e) {
    p <- p*a1
    p <- p%%n1
  }
  p
}
for(i in 1:n) encrypt(i)
n1 <- 371893L        # integer
for(i in 1:n) encrypt(i)
```

In this function we start with the key element `n1` as a floating-point number (calculated from `p1` and `p2`). The function is repeatedly called and is therefore compiled expecting floating-point numbers in `n1`. At some point, however, `n1` changes and is set to an integer. After that, the function is again called several times. This will lead to a deoptimization followed by compilation for floating-point *and* integers. At this point the profiler is supposed to step in and detect the stale feedback for floating-point and cause function `encrypt` to be recompiled to only support integers in `n1`. This benchmark shall serve as an example for a program with a phase change: After a stable behavior in a first phase, some datatypes change as the program transitions into its second phase.

The third benchmark (*profiler_shared*) consists of the following functions:

```
id <- function(a) {
  # prevent inlining
  while (F) a;
  while (F) a;
  a
}
add <- function(a, b) {
  id(a)+id(b)
}
```

Initially the `add` function is called with a number that has a class attribute attached to it.

```
poison=structure(1, class="foo")
add(poison, poison)
add(poison, poison)
```

Because `poison` is an object and `add` does not use it directly, it is passed as a promise to `id` where it is in fact evaluated. `id` is also written in such a way that it will not be inlined. This prevents `add` from knowing that `poison` is in fact a number. In the following part of the benchmark, the `test` function is called repeatedly which in turn repeatedly calls the `add` function with simple numbers. The `add` function is inlined into `test` importing with it the unrelated type feedback of the earlier calls with `poison` as arguments. We lose performance improvements we could have had if `add` had the information that it was

```
test <- function() {
  s = 0
  for (num in 1:500000) {
    s = add(s, num)
  }
  s
}
for (i in 1:n)
  test()
```

dealing with normal numbers. This is, however, a fact the profiler should be able to detect and trigger reoptimization for. This is a second example of a scenario where previously relevant type information is no longer needed. But other than in the *profiler_microbenchmark* this time it is the combination of an object and a number type instead of two number types. And instead of a global variable that changes, it is the method's parameter in a situation where the function would normally not be specialized.

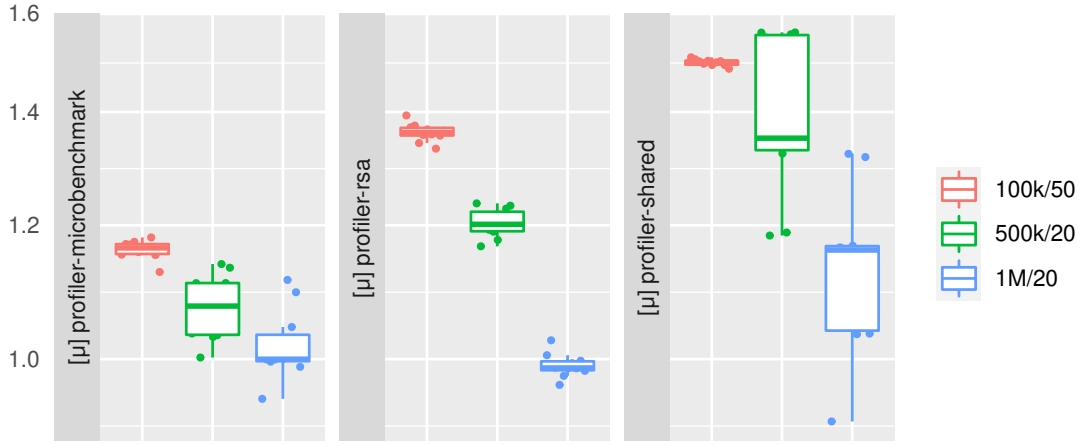


Figure 5.3: Performance Improvements

For the evaluation we used the threshold values determined in section 5.2. For each sampling period configuration, we used the smallest measured threshold above which no significant improvements in the outlier count could be observed. This is 20 for periods of 1 million and 500,000 instructions and 75 for 100,000 instructions. Tuning the compilation threshold is not easy and there is a clear trade-off: With a lower threshold, optimizations can be applied much quicker but we risk overly eager optimization. With a higher threshold, we reduce the risk of overly eager optimization but delay optimization where they would make sense.

Figure 5.3 shows that for all three benchmarks, the 1 million instruction sampling period performed the worst of all three. In fact, for each benchmark, the smaller sampling periods are always performing better. This makes sense, since the difference in thresholds is smaller than the difference in the sampling period. This leads to shorter sampling times and quicker reoptimization when using smaller sampling periods.

We also observed a sampling period of 1 million instructions and a threshold of 20 to not yield any substantial performance improvements. In all three benchmarks there are runs that show a reduction in performance. However, with a sampling period of 500,000 instructions at the same threshold, we are capable of reoptimization soon enough that we see a significant increase in performance in all three

benchmarks.

While using a sampling period of 100,000 instructions with a threshold of 50 yields better performance, it is also the configuration that produces the most outliers in section 5.2 of the configurations presented here. This once more highlights the trade-off present in tuning the profiler for a specific application. To conclude, taking into account both the outliers produced in section 5.2 and the benefit gained in these three examples, we consider a sampling period of 500,000 instructions and a threshold of 20 to be the best combination.

6 Conclusion and Future Work

6.1 Conclusion

We present a sampling-based profiler for a virtual machine, capable of monitoring native code without instrumentation, used to detect inefficient code with missed optimization opportunities. The absence of instrumentation has the advantage that when the profiler is not engaged, no additional overhead is incurred. We evaluate the approach on an implementation for the Ř research virtual machine for the R language. Our profiler uses the Performance Monitoring Unit (PMU) to trigger samples, allowing us to reliably interrupt the program only when running in user-level code and not during system calls.

Preliminary measurements indicate that it is possible to accurately detect stale or too generic specialization in native code and improve peak performance by recompilation with subsequent specialization to the sampled information. The profiling was found to incur small overheads of typically below 3% and up to 13% for some benchmarks and configurations. The main trade-off observed concerns the recompilation threshold, which needs to be tuned to trigger as early as possible while avoiding the use of incomplete data leading to wrong speculation. Our evaluation considers many combinations of sampling intervals and recompilation thresholds, which allowed us to determine a sweet spot for our implementation. Three short running benchmarks representative for real-world situations with stale type feedback showed encouraging improvements of 1.1–1.5×.

It still remains to be seen if the proposed technique can be tuned and made robust enough to improve the performance of real world R programs without causing unexpected performance behavior. As future work, we proposed to explore more robust heuristics, sampling of more properties besides types and the recording of context sensitive samples.

6.2 Future Work

6.2.1 Deoptimization Loop Break-Out

While the current implementation works well and demonstrated the potential for significant performance gains in programs with stable types[1] there is the potential for the profiler to get caught in a deoptimization loop: If a variable frequently alternates between two types, it is likely that the profiler will see both of these types. In this situation the profiler will not attempt a reoptimization based on the variable types. But if the alternation frequency is low enough it is possible that the profiler decides to reoptimize for one type just before the variable type changes. This leads to a deoptimization soon after the reoptimization. It is obvious that such a situation actually decreases the performance as there are not only two compiler runs close together, but the deoptimization forces the VM to run the code in the interpreted mode before attempting a recompilation.

In order for our profiler implementation to be used in a production environment a mechanism has to be introduced to detect these situations and essentially force the profiler to capitulate and to no longer attempt a recompilation. This could be a relatively simple approach where we never try to reoptimize a function that has experienced a deoptimization after we triggered a reoptimization. It would also be possible to only block the profiler from performing reoptimizations temporarily with growing cool-down timers. This would essentially create a backoff strategy that would still allow the profiler to come in at a later point after the variable type alternations have cased. It is however unclear whether such a strategy would bring any benefit compared to the simpler implementation, as in programs where type alternations persist we would still repeatedly attempt recompilation.

Another approach may be to not prevent the profiler from triggering reoptimization but instead to increase the threshold for reoptimization after each deoptimization. This approach would increase the sampling time required for each reoptimization and thereby increase the change for all types to be seen before a potential reoptimization. The sampling period would grow as long as required to record all possible types and would then stabilize. This may yield better performance than the backoff strategy described above as as soon as the sampling period has sufficiently increased no more reoptimization will be triggered until the observed types change again. This approach should theoretically initially show similar performance to the backoff strategy above but should show better performance once the sampling period has been sufficiently increased as the backoff strategy would still attempt reoptimizations albeit increasingly sporadically.

6.2.2 Burst Sampling

While we were able to demonstrate minimal performance overhead with our current implementation it may still be interesting to investigate another approach at minimizing overhead: By repeatedly enabling and disabling the profiler we could use high sample rates for limited periods of time while keeping the overhead minimal. By using such a *burst sampling* approach we could for example record at 10 samples per millisecond for one second every 2 seconds while observing similar overhead to recording at 5 samples per millisecond continuously. This approach may however introduce some additional overhead as the sample triggering must be enabled and disabled repeatedly. It is unclear whether our implementation would actually benefit from such an approach but we believe it worth some investigation.

6.2.3 Runtime Configuration

The current implementation requires the profiler to be fully configured through environment variables at the time of Ř VM startup. It is also initialized during VM startup and starts running at that point. By adding

[1] *i.e.,* programs where variable types change rarely and remain constant for long periods of time

the possibility for programatic control over the profiler by code running in the VM programmers could make use of the profiler as they see fit. For example if it is known that a program would not benefit from the profiler during a certain execution phase, it could be explicitly disabled during that period and only be enabled where it may benefit the performance. Runtime configurability would also facilitate further research as currently for each configuration to be tested at least the Ř VM needs to be restarted, and in some cases the VM even needs to be recompiled.

Bibliography

[1] Advanced Micro Devices, Inc. *Processor Programming Reference (PPR) for AMD Family 17h Model 01h, Revision B1 Processors, Sec. 2.1.13*, April 2017.

[2] Glenn Ammons, Thomas Ball, and James R. Larus. Exploiting Hardware Performance Counters with Flow and Context Sensitive Profiling. *SIGPLAN Notices (ACM Special Interest Group on Programming Languages)*, 32(5):85–96, 1997.

[3] Jennifer M. Anderson, Lance M. Berc, Jeffrey Dean, Sanjay Ghemawat, Monika R. Henzinger, Shun Tak A. Leung, Richard L. Sites, Mark T. Vandevoorde, Carl A. Waldspurger, and William E. Weihl. Continuous Profiling: Where Have All the Cycles Gone? *ACM Transactions on Computer Systems*, 15(4):357–390, 1997.

[4] ARM. *ARM Architecture Reference Manual, Ch. C12*, ARMv7-A and ARMv7-R edition, May 2014.

[5] Matthew Arnold, Michael Hind, and Barbara G. Ryder. Online feedback-directed optimization of Java. *ACM SIGPLAN Notices*, 2012.

[6] John Aycock. A Brief History of Just-In-Time. *ACM Computing Surveys*, 35(2):97–113, June 2003.

[7] Edd Barrett, Carl Friedrich Bolz-Tereick, Rebecca Killick, Sarah Mount, and Laurence Tratt. Virtual machine warmup blows hot and cold. In *Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, 2017.

[8] Olivier Flückiger, Guido Chari, Jan Ječmen, Ming Ho Yee, Jakob Hain, and Jan Vitek. R melts brains: An IR for first-class environments and lazy effectful arguments. In *DLS 2019 - Proceedings of the 15th ACM SIGPLAN International Symposium on Dynamic Languages, co-located with SPLASH 2019*, pages 55–66. Association for Computing Machinery, Inc, October 2019.

[9] The R Foundation. *R: What is R?* https://www.r-project.org/about.html.

[10] Free Software Foundation, http://man7.org/linux/man-pages/man2/perf_event_open.2.html. *perf_event_open(2) — Linux manual page*, October 2019.

[11] Free Software Foundation, http://man7.org/linux/man-pages/man2/sigaction.2.html. *sigaction(2) — Linux manual page*, October 2019.

[12] Free Software Foundation, http://man7.org/linux/man-pages/man7/signal.7.html. *signal(7) — Linux manual page*, August 2019.

[13] Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick. gprof: a Call Graph Execution Profiler. In *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction, SIGPLAN 1982*, pages 120–126. Association for Computing Machinery, Inc, June 1982.

[14] Robert J. Hall. Call path profiling. In *Proceedings - International Conference on Software Engineering*, number 14, pages 296–306. Publ by IEEE, May 1992.

[15] Intel Corporation, https://software.intel.com/en-us/articles/intel-sdm. *Intel 64 and IA-32 Architectures Software Developer Manuals, Vol. 3B, Ch. 18–19*, October 2019.

[16] Thomas Kistler and Michael Franz. Continuous program optimization: A case study. *ACM Transactions on Programming Languages and Systems*, 25(4):500–548, July 2003.

[17] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. Numba: a LLVM-based Python JIT compiler. *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC - LLVM '15*, pages 1–6, 2015.

[18] Sheng Liang and Deepa Viswanathan. Comprehensive Profiling Support in the Java Virtual Machine. In *Proc. COOTS*, page 17, 1999.

[19] Shirley V. Moore. A comparison of counting and sampling modes of using performance monitoring hardware. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 2330 LNCS, pages 904–912, 2002.

[20] Carl Ponder and Richard J. Fateman. Inaccuracies in program profilers. *Software: Practice and Experience*, 18(5):459–467, 1988.

[21] Igor Rafael De Assis Costa, Henrique Nazaré Santos, Péricles Rafael Alves, and Fernando Magno Quintão Pereira. Just-in-time value specialization. *Computer Languages, Systems and Structures*, 40(2):37–52, 2014.

[22] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88, pages 12–27, New York, NY, USA, 1988. Association for Computing Machinery.

[23] Florian T. Schneider, Mathias Payer, and Thomas R. Gross. Online optimizations driven by hardware performance monitoring. *ACM SIGPLAN Notices*, 42(6):373–382, June 2007.

[24] Michael D. Smith. Overcoming the challenges to feedback-directed optimization. In *Proceedings of the ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization, DYNAMO 2000*, pages 1–11. Association for Computing Machinery, Inc, January 2000.

[25] Toshio Suganuma, Toshiaki Yasue, Motohiro Kawahito, Hideaki Kornatsu, and Toshio Nakatani. A dynamic optimization framework for a Java just-in-time compiler. *ACM SIGPLAN Notices*, 36(11):180–194, November 2011.

[26] John Whaley. A portable sampling-based profiler for Java virtual machines. In *ACM 2000 Java Grande Conference*, pages 78–87, 2000.

[27] Dmitrijs Zaparanuks, Milan Jovic, and Matthias Hauswirth. Accuracy of performance counter measurements. In *ISPASS 2009 - International Symposium on Performance Analysis of Systems and Software*, pages 23–32, 2009.