FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of master's thesis

Title:	Profiler for the R programming language
Student:	Bc. Karolina Hrnčiříková
Supervisor:	doc. Ing. Filip Křikava, Ph.D.
Study program:	Informatics
Branch / specialization:	System Programming
Department:	Department of Theoretical Computer Science
Validity:	until the end of summer semester 2024/2025

Instructions

The R language is slow. Because of this, R users tend to rewrite the performance-critical parts in C/C++ or Fortran. However, identifying which parts should be replaced by native code is not trivial. To prevent wasting time optimizing the wrong part of the codebase, developers use profilers to guide them to performance-critical and/or memory-sensitive code. Unfortunately, the existing R profilers are limited, providing just a snapshot of the call stack.

Analyze the current state of the Rprof and its implementation in the R virtual machine. Explore ways to implement a new profiler that would allow one to tease apart the time spent running in the R interpreter and executing native code. Explore ways to connect these measurements to the code lines. Prototype such a profiler, ideally with minimal changes to the R virtual machine.

Electronically approved by doc. Ing. Jan Janoušek, Ph.D. on 23 January 2024 in Prague.

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Master's thesis

Profiler for the R programming language

Bc. Karolina Hrnčiříková

Department of Theoretical Computer Science

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

May 9, 2024

Acknowledgements

I would like to express my gratitude and thanks to my supervisor Filip Křikava for his valuable insights and guidance.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as a school work under the provisions of Article 60 (1) of the Act.

In Prague on May 9, 2024

Czech Technical University in Prague

Faculty of Information Technology

Citation of this thesis

Hrnčiříková, Karolina. *Profiler for the R programming language.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2024.

Abstrakt

Jazyk R vyniká v průzkumu a analýze dat, ale často čelí výzvám v oblasti efektivity. R je dynamicky typováno, má automatický sběr paměti a co je nejdůležitější, jedna z jeho hlavních implementací, GNU R, interpretuje AST v kombinaci s kompilací just-in-time do bajtkódu. Všechny tyto faktory přispívají k tomu, že R je poměrně pomalý jazyk.

Pro zlepšení výkonu jsou uživatelé nuceni přepisovat kód citlivý na výkon v C, C++ nebo ve Fortranu prostřednictvím balíčků. Zjistit, které segmenty kódu jsou pomalé, protože jsou prováděny v interpretu R, však není snadné, protože současné metody profilování nerozlišují mezi prováděním kódu nativně a v R.

V této práci navrhujeme profiler, který dokáže rozlišit R a nativní vykonávání kódu. Inspirováni Scalene, profilerem pro Python, implementujeme do GNU R 4.3.3 prototyp našeho profileru. Profiler hodnotíme ve srovnání s Rprof, nejpoužívanějším R profilerem.

Klíčová slova profilování, R, GNU R, vzorkování, interpret, SEXP, Scalene, Rprof

Abstract

The R language excels in data exploration and analysis but often faces challenges regarding execution speed and efficiency. R is dynamically typed, has automatic memory collection, and, most importantly, one of its main implementations, GNU R, interprets AST in combination with just-in-time compilation into bytecode. All these factors contribute to R being a comparatively slow language.

To improve performance, users are forced to rewrite performance-sensitive code in C, C++, or Fortran through packages. However, finding out which code segments are slow because they are executed in the R interpreter is not easy because the current profiling methods do not distinguish between native and R execution.

In this thesis, we propose a profiler that can distinguish between R and native execution. Inspired by Scalene, a profiler for Python, we implement a proof-of-concept profiler into GNU R 4.3.3. We evaluate the profiler in comparison to Rprof, the most used R profiler.

Keywords profiling, R, GNU R, sampling, interpreter, SEXP, Scalene, Rprof

Contents

List of Figures

Chapter 1

Introduction

The R programming language, known for its robust capabilities in statistical computing and plotting, is widely adopted across various domains of data analysis in both academia and industry. R excels in data exploration and statistical modelling but often faces challenges regarding execution speed and efficiency. R is dynamically typed, it has automatic memory collection, and most importantly, one of its main implementations, GNU R [1], interprets AST in combination with bytecode. All these factors contribute to the fact that R is a comparatively slow language.

R allows for extending its capabilities via packages. While these can be written in R for performance-sensitive code, many of these packages are written in higher-performance languages like C and C++. These packages expand R's functionality while attempting to mitigate performance slowdown by leveraging faster execution at the native code level. Suppose we can identify which code is executed by R's interpreter and which parts run natively. Then, programmers could use this knowledge to rewrite areas that execute R with code that uses native libraries to enhance performance.

In the realm of profiling, a tool like Scalene [2] for Python offers insights by distinguishing between Python execution time and native execution time within Python programs. Scalene serves as an example of how profiling can aid in optimizing performance, particularly in a language that shares similar characteristics to R, such as being high-level, interpreted, and extensible by C code.

1.1 Aim of the Thesis

Inspired by the capabilities of Scalene, this thesis proposes the development of a similar profiling tool for the R language. This profiler aims to differentiate between time spent executing R code and time spent within native execution. The primary objective of this tool is to provide R programmers with a deeper understanding of where their R programs spend time, thereby identifying inefficient code, some of which can be rewritten in C/C++.

This profiler is a proof of concept. It has been developed for R version 4.3.3 under Linux.

1.2 Structure of the Thesis

The rest of the thesis is organized as follows. The second chapter introduces the R language and its mostly used profiler. Then, it overviews Scalene, a point of reference for our profiler. The third chapter describes R internals that are necessary to understand our implementation of a profiler. The fourth chapter shows in detail how the new profiler is implemented. It covers adding signal handling, assigning lines of code and more. The fifth chapter presents the assesment of our profiler. We analyze its outputs on their own and in contrast to Rprof. The last chapter summarizes the most important results of the thesis and offers further improvements.

Chapter 2

Preliminaries

In this chapter, we review some of the key concepts used in the thesis. First, we briefly describe the R language, its main features and how it is used. Then, we will review the current methods for profiling R code. Lastly, we will overview Scalene, a Python profiler that can distinguish between Python execution and native execution because it serves as an inspiration for our R profiler.

2.1 The R Language

R is a programming language designed with statistical computing and data visualization in mind. It originated as a GNU project based on the S language [3].

R is known for its robust ecosystem of extension packages, allowing users to perform a wide range of statistical techniques, from simple calculations to complex predictive modelling. Furthermore, the language is highly extensible, supporting seamless integration with other programming languages like C and C++, which enhances its performance and capabilities. That is especially useful since the GNU R implementation of R either interprets over an abstract syntax tree (AST) or bytecode[1], which means that executing R code is slow in comparison to compiled languages [1].

The language features dynamic typing and automatic memory management (garbage collection (GC)), simplifying coding by freeing the user from manual memory handling. As a functional programming language, R supports functions as first-class objects. Moreover, R employs lazy evaluation, meaning expressions are not computed until their values are actually needed. This can lead to efficiency improvements.

[1] GNU R has got just-in-time compilation from AST into bytecode. By default, a code is compiled into bytecode after 3 runs.

R has many different modes of usage. It can be used interactively in REPL environments and consoles, as scripts, as notebooks (for example, Jupyter notebooks [4]), or integrated with markdown and many more. It does not make much sense to profile the interactive use of R. Our profiler will focus on longer-running R code, especially without further user input.

2.2 Profiling R Code

Currently, there are not many ways to identify performance issues of R code. One way is to manually use the `system.time()` function [5]. This has the drawback that we might track only parts we believe might be expensive, and not analyze the program as a whole. And then there is Rprof, the most commonly used sampling profiler for R, which is built into R itself. Apart from profiling the CPU, it can also profile memory and the usage of GC. It aims to profile at a function level.

Rprof is a sampling profiler, i.e., it operates by periodically stopping the program execution and inspecting its call stack. Rprof developers chose the `ITIMER_PROF` alarm which generates `SIGPROF` signal [6]. `ITIMER_PROF` decrements the timer when the application is both in user space or kernel space [6]. The Rprof's timer is set by default to 20 ms and, on Linux systems, can not be set below 10 ms. Sending signals at an even faster rate is unrealistic due to the OS overhead and the pace at which the profilers can capture samples. The signal handler triggered at the reception of `SIGPROF` signal directly takes and processes a sample. It walks the stack frame, noting each function. If other options, such as memory profiling, are enabled, then it processes that as well.

Let us look at how we can work with Rprof. In Figure 2.1, we can see an example of code that computes factorials. On line 1, we start the profiling by calling the `Rprof` function [7]. It takes as an argument a file into which the output is written. Then, there is the code that we want to profile. Finally, to end profiling, the `Rprof` function is called yet again, this time with argument `NULL` [7]. There is a function `summaryRprof` [8] to inspect the data, to which we feed the output file.

For instance, if we run the code 2.1, the output might contain something like what we can see in Figure 2.2. Rprof measures for each function two types of data: how much time we spent in the function itself, denoted by `self`, and how much time we spent in the function, including nested calls to other functions, denoted by `total`. So, if we look at the row with `lapply`, we can see that we spend little time executing the function itself, specifically 0.8 s, which is 3.37% of the whole execution time. However, if we include the time we spent in the anonymous function, defined in lines 11 to 13, and the `facto` function, then we spent the whole execution time, 23.76 s, in the `lapply` function.

Although this information provides insight into where the execution spent the most time, we do not know much about the nature of the execution. For example, we do not know whether some code is executed in native or in R. This could help us identify R segments viable for being rewritten using packages.

```
1  Rprof("outputfile.txt")
2
3  facto <- function (n) {
4      if (n == 0) {
5          return(1)
6      } else {
7          return(n * facto(n - 1))
8      }
9  }
10
11 lapply(1:1000000, function(i) {
12     facto(30)
13 })
14
15 Rprof(NULL)
16
17 summary <- summaryRprof("outputfile.txt")
18 print(summary)
```

Figure 2.1: This snippet demonstrates how we can use the Rprof profiler. First, we need to start the profiler on line 1. Then, we stop it at line 15. The output file itself does not contain analyzed data, just some metadata and then the contents of the call stack taken at each sample. Therefore, we further analyze it using the `summaryRprof` function.

```
         self.time  self.pct  total.time  total.pct
"facto"      22.48     94.61       22.50      94.70
"lapply"      0.80      3.37       23.76     100.00
"FUN"         0.46      1.94       22.96      96.63
```

Figure 2.2: This is part of an output we got when running 2.1. The output sorts the functions in two ways, based on the `self` data and then based on the `total` data. This snippet is based on `self`. In general, we can have more functions in the `total` ordering because they transitively acquire time from the nested calls while having `self.time` equal to zero.

2.3 Scalene

Our profiler for the R language draws inspiration from Scalene, a profiler for Python, made by Berger et al. [2]. It offers comprehensive profiling capabilities spanning CPU, GPU, and memory usage analysis. Unsurprisingly, it became very popular and has got over 11,000 stars on GitHub [2] [2]. In Figure 2.3, we can see a little demo of Scalene's profiling output. Our objective is to adapt its principles to align with the specific requirements of our profiler for the R language.

Python's performance is notably impeded by its dynamic typing system and the overhead of GC. CPython, the predominant stack-based bytecode interpreter for Python implemented in C, commonly introduces a performance penalty ranging from one to two orders of magnitude when compared to native code

[2]Moreover, the paper introducing Scalene [9] won the OSDI 2023 Best Paper Award [10]

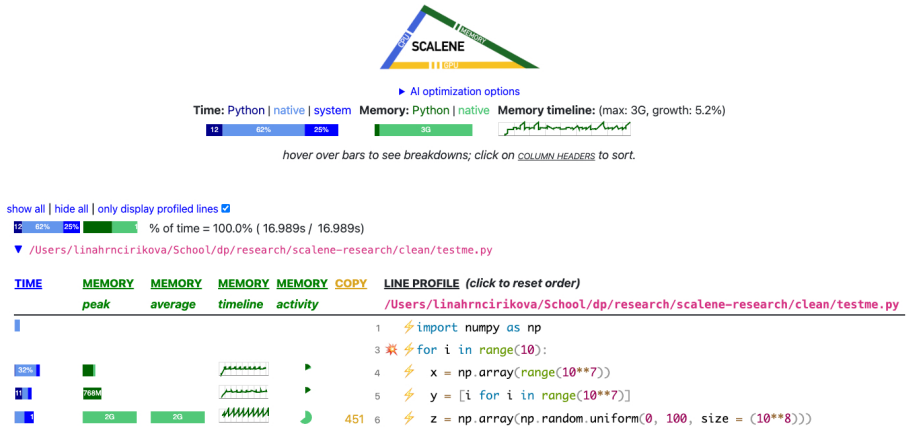

Figure 2.3: This figure shows an example of a profiling output for the code that can be seen in the lower right part of the picture [11]. In text, the high-level overview is that dark colours correspond to Python execution and lighter colours correspond to native execution. The first column represents CPU profiling. Then, there are multiple columns analyzing memory consumption; dark green shows that the memory was allocated within Python, and light green that it was allocated from native code. Scalene measures both peak and average memory consumption. Additionally, it introduces a new metric called *copy volume*, which notifies us of copying across the Python-native execution boundary. In our example, the copy volume spikes on line 5, which can be reduced by removing the outer `np.array()` call on the same line.

execution. In response to these challenges, programmers seek ways to identify segments of code executed within the Python interpreter and replace them with native code alternatives. However, existing profiling tools fail to adequately differentiate and report between code executed natively and within the Python interpreter. Thus, the emergence of Scalene addressed this critical gap, offering a new solution to the profiling challenge.

2.3.1 CPU Profiling

Scalene employs sampling to profile CPU usage. Sampling profilers operate by periodically stopping program execution and capturing the currently executing function, known as a sample. The underlying principle suggests that given enough samples, the time spent on a particular function correlates proportionally with the number of samples collected for that function [12].

To implement sampling, i.e. interrupt program execution and capture samples, we typically employ signals. In Python, signals are exclusively handled in the interpreter within the main thread. Scalene leverages this behaviour to differentiate between Python execution and native execution. By periodically sending signals, Scalene's signal handler detects the timing of signal reception. Immediate signal reception indicates time spent within the Python interpreter, while a delay suggests execution outside the interpreter [9].

To be more specific, Scalene maintains two counters for each profiled line: one for native execution and another for Python execution. Scalene dispatched signals at regular intervals, by default, every 10 ms (q) [3]. Upon receiving a signal, it records the current virtual time. When handling the following signal, it calculates the elapsed virtual time (T) since the previous signal handling. For Python execution, Scalene increments the counter by q. For native execution, it increments the counter by $T - q$. Diagram 2.4 illustrates the differentiation between Python execution and native execution.

This dual incrementation compensates for the lack of precise knowledge regarding the exact moment of transition from Python execution to native code. The profiler employs `ITIMER_VIRTUAL`, which is a timer mechanism that tracks CPU time consumed by the process under scrutiny [6, 9]. This prevents interference from other concurrently running processes.

Because signals are exclusively handled in the main thread, Scalene employs a different algorithm for profiling the CPU usage of children's threads. The key idea lies in monkey patching blocking functions so that they time out and set a flag informing whether the given thread is waiting/asleep or not. Then, when a signal is received, the profiler checks the bytecode instructions of the child threads. Suppose a child thread is not asleep and it is waiting on an instruction that corresponds to a call to a native function. Then, Scalene attributes the time to native execution and otherwise to Python execution. However, since the R language lacks support for multithreading, we will not delve into the specifics of this algorithm [9].

So, the high-level overview is that we input a piece of code we want to profile. Then, we receive output that attributes to each line the percentage of the execution time spent on it. Each line has information for both Python and native execution. For our R profiler, we want the same output. Given a piece of code, we want the output to show the percentages of time spent in R and native execution for each line.

However, our starting point is a little different than that of Scalene. Python already supports user-defined signal handling, but we have to add that to R. Therefore, we need to understand the architecture or R's interpreter, so that we know where how to add our signal handling. Moreover, there is a library, Frame Objects [14], to assess which line of Python code is currently executing. There is no such library in R. Therefore, we have to add it ourselves. For that, we need to understand R's internal representation and how the call stack works.

[3] The 10 ms period was chosen because it strikes a balance between good performance and accuracy (and it is a common choice for sampling profilers). Further information like the rate at which the Python interpreter executes instructions, how often is the Global Interpreter Lock released, or at what maximum pace Scalene is able to handle signals were not needed to be taken into account [13].

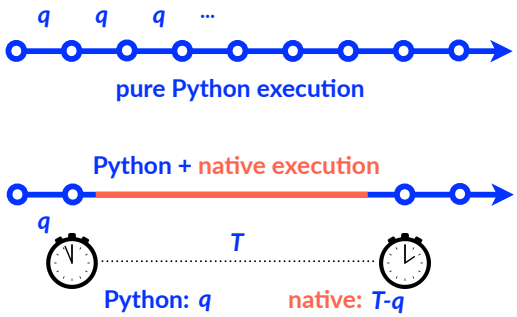

Figure 2.4: The diagram shows how Scalene differentiates between Python execution and native execution. The image is taken from [9].

2.3.2 Conclusion

To conclude, Scalene leverages the fact that in Python signals are only ever handled in the Python interpreter. The profiler periodically sends signals and finds the currently executing line. If there is a delay between sending a signal and handling it in the evaluation loop, then we know that time must have been spent in native execution. If there is no delay, then Python is being executed.

2.4 Conclusion

In summary, we briefly introduced the R language. Then, we looked at ways to identify performance bottlenecks in R and illustrated how Rprof, its most used profiler, works.

Currently, there is no easy way to determine which parts of the program run in R and which in native execution. Therefore, we analysed how Scalene, a profiler that is able to distinguish between Python and C execution, implements CPU profiling. The key idea lies in assigning delay in obtaining signal to native execution. Based on this observation, we want to implement our own R profiler that would be able to differentiate between R and native execution.

Chapter 3

Representation of R Code

In this chapter, we document some essential aspects of the GNU R internal representation necessary for the implementation of our profiler. Namely, we cover S-expressions (SEXPs) and the structure of R's call stack. Understanding it is necessary so that the profiler can asses which line of code is being executed and, therefore, attribute the execution times correctly.

3.1 SEXP

When profiling, we must work with the internal representation of R code so that we know how to assess what code is currently being executed. Therefore, in this section, we delve into how R code is internally stored and represented at C-level. However, there is a focus on concepts we will need later on in this thesis. For more comprehensive overview see [15], [16], and [17].

Let us consider an example depicted in Figure 3.1. As mentioned earlier, R is primarily a functional language. Therefore, basically, everything is a function call. For instance, the `{}` syntax that typically denotes a block is also a function call, and its arguments are the lines of the block, in our example, it is `2 + foo(3) + x`. The line is naturally also an expression which constitutes a function call, where `+` is the function's name, and its arguments are `2 + foo(3)` and `x`, and so forth. The overall structure can be seen in Figure 3.2.

```
1 # `{`(`+`(`+`(2, foo(3)), x))
2 {
3     2 + foo(3) + x
4 }
```

Figure 3.1: This code snippet is a brief example of R code. It is used multiple times in this section to illustrate the internal representation of R. On the first line, we can see a comment, which shows how the calls stack up.

```
"{"
 \------- "+"
          |------- "+"
          |         |------- "2"
          |         \------- "foo"
          |                    \----- "3"
          \------ "x"
```

Figure 3.2: This diagram represents the high-level overview of the structure of the code from Figure 3.1.

In the R language, all expressions are represented by SEXPs. Furthermore, all values are expressed as SEXPs and are understood by the runtime. Some of those are language objects such as function calls. These SEXPs are organized into nested lists and create a tree-like structure. The R interpreter traverses the SEXP representation of the code and executes the corresponding code segments.

```
typedef struct SEXPREC {
    // SEXPREC_HEADER
    struct sxpinfo_struct  sxpinfo;
    struct SEXPREC        *attrib;
    struct SEXPREC        *gengc_next_node;
    struct SEXPREC        *gengc_prev_node;
    // PAYLOAD
    union {
            struct primsxp_struct  primsxp;
            struct symsxp_struct   symsxp;
            struct listsxp_struct  listsxp;
            struct envsxp_struct   envsxp;
            struct closxp_struct   closxp;
            struct promsxp_struct  promsxp;
    } u;
} SEXPREC;
```

Figure 3.3: This code shows the definition of `SEXPREC` structure analogously to how it is defined in `Defn.h` in GNU R [1]. The structure contains a header and payload. In the actual implementation, the first four member variables are enclosed in the `SEXPREC_HEADER` macro.

```
struct symsxp_struct {
    struct SEXPREC *pname;
    struct SEXPREC *value;
    struct SEXPREC *internal;
};
```

Figure 3.4: This block of code shows the definition of `symsxp_struct` as it is written in `Defn.h` in GNU R [1]. Apart from `primsxp_struct` all the other structures in the union type in `SEXPREC` also contain three fields of the type `struct SEXPREC *`.

At the C-level, SEXPs are pointers to a `SEXPREC` structure. The contents of this structure are detailed in 3.3. The first four members are part of a header, and the variable `u` represents data. `gengc_next_node` and `gengc_prev_node` aid in GC by pointing to another SEXP in the same GC generation. Then, all objects in R can have attributes. Those can be either defined internally (such as `srcref` attribute), or we can add them as R programmers. They are represented by the `attrib` member variable. Lastly, the `sxpinfo` field contains multiple data. Most important for us is the kind of SEXP it represents, called the `SEXPTYPE`. Let us go over the relevant `SEXPTYPE`s in more detail:

- `NILSXP` represents a null value.
- `SYMSXP` represents symbols. The `SEXPREC` representing `SYMSXP` contains `symsxp_struct` structure as its payload, whose definition can be seen in 3.4. For us, it is significant that `symsxp_struct` contains a member variable called `pname`. The `pname` field is a pointer to a `CHARSXP`, where the printable name of the symbol is stored.
- `LISTSXP` is used to represent lists. The `SEXPREC` representing `LISTSXP` has `listsxp_struct` as its payload [15]. The `listsxp_struct` structure has three member variables: `carval`, `tagval`, and `cdrval`. The first two fields make up the value of the current SEXP node, `cdrval` points to the next item in the list, or if it is at the end, it is set to `NILSXP`. So, the SEXP is not the whole list. Instead, it is only one item on the list. Therefore, it might be better to think of `LISTSXP` as a *cons cell* [15]. There are macros such as `CAR`, `CDR`, `CADR`, and `CDDR` to simplify walking the list.
- `CLOSXP` represents closures. The `SEXPREC` representing `CLOSXP` contains `closxp_struct` as its payload. The `closxp_struct` structure has three member variables: `formals`, `body`, and `env`. If a closure has parameters, they are accessible via the `formals` field. The parameters are represented using `LISTSXP`. The `body` variable represents the body of the closure. Finally, the `env` field represents the enclosing environment of the closure. This is useful, for example, because R has lexical scoping.
- `LANGSXP`, language expression, represents a function call. The `SEXPREC` representing `LANGSXP` contains `listsxp_struct` structure as its data [15] (the same structure as for `LISTSXP`). The `carval` member variable represents the function that is called. For instance, it can be a `SYMSXP` when using the function's name directly (such as `foo()`), or it can be a `LANGSXP` when determining which function to call within a package (such as `package::foo()`). The `cdrval` field represents the arguments of the function call in the form of a `LISTSXP`. The `tagval` field embodies a named argument [16].
- `REALSXP` represents numeric vectors. Values of components of the vector correspond to the `double` data type in C [16].

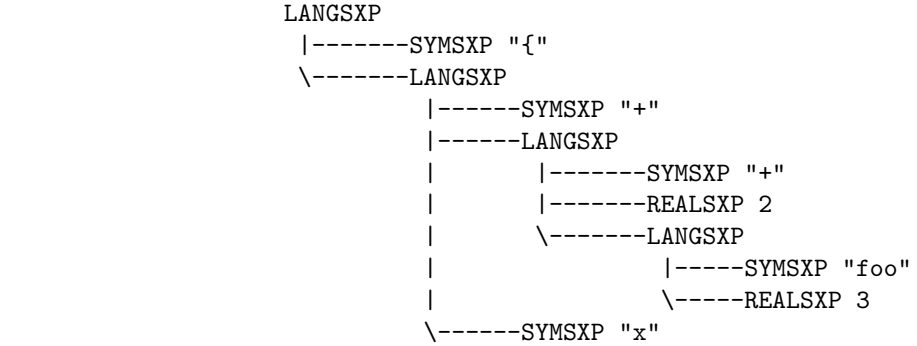

Figure 3.5: This diagram shows the code snippet from 3.1 translated into the SEXP structure.

```
@c63cb388a418 06 LANGSXP g0c0 [REF(1)]
  @c63cb33fe6c0 01 SYMSXP g0c0 [MARK,REF(3786),LCK,gp=0x5000]
                               "{" (has value)
  @c63cb388a450 06 LANGSXP g0c0 [REF(1)]
    @c63cb34098e8 01 SYMSXP g0c0 [MARK,REF(81),LCK,gp=0x5000]
                                 "+" (has value)
    @c63cb388a488 06 LANGSXP g0c0 [REF(1)]
      @c63cb34098e8 01 SYMSXP g0c0 [MARK,REF(81),LCK,gp=0x5000]
                                   "+" (has value)
      @c63cb59db778 14 REALSXP g0c1 [REF(2)] (len=1, tl=0) 2
      @c63cb388a4f8 06 LANGSXP g0c0 [REF(1)]
        @c63cb39808f0 01 SYMSXP g0c0 [MARK,REF(52)] "foo"
        @c63cb59db740 14 REALSXP g0c1 [REF(2)] (len=1, tl=0) 3
    @c63cb3464f30 01 SYMSXP g0c0 [MARK,REF(11534)] "x"
```

Figure 3.6: This figure shows an example of an output to the `.Internal(inspect(quote({2 + foo(3) + x})))` command in R without the attributes.

Now that we have gone over some of the SEXPs, we can look at how our example from 3.1 will look in its SEXP form in diagram 3.5. We can observe that all function calls are indeed represented by `LANGSXP`. In R, there is not a special SEXP for a single numeric value. Therefore, the numeric values in the example are represented by a vector of size equal to one by `REALSXP`.

SEXPs are mostly invisible to an R programmer. Nevertheless, there are functions that enable us to see the internal SEXP representation of a given R code. For instance, running `.Internal(inspect(quote({2 + foo(3) + x})))` in R provides insight into the internal structure and outputs something akin to 3.6.

3.2 Call Stack in R

At the C-level, R's stack frame is represented by the `RCNTXT` structure, which plays a critical role in the language's function call management. Each time a function is invoked, an `RCNTXT` structure is instantiated, effectively capturing the state of the function call. This context is then pushed onto the front of the call stack, ensuring that it becomes the active context. Upon function exit, this context is removed, *popped*, from the stack and subsequently deleted, maintaining the stack's integrity.

The `RCNTXT` structure is composed of several variables, which collectively manage and record the details of the function execution environment. Most notable among them are:

- `RCNTXT *nextcontext`: This pointer links the current context to the previous one on the stack, thus chaining together all active contexts. This linkage is crucial for the proper unwinding of the stack upon function exits.
- `int callflag`: This field holds the *type* of the context. It is the result of bitwise `or` over some predefined values. For the purpose of this thesis, we only need to know of `CTXT_TOPLEVEL`.
- `SEXP promargs`: This variable holds a pointer to a list of promises. In R, function arguments are mostly treated as promises, meaning that we defer the computation of function arguments until their values are required. This mechanism is essential for implementing lazy evaluation, allowing R to delay the computation of argument values until they are explicitly accessed within the function body.
- `SEXP callfun`: This holds a reference to a SEXP that represents the function that is being executed.
- `SEXP call`: This points to the SEXP representing the function call itself. This includes the function and its arguments, allowing the execution environment to access both the function to be called and the specifics of how it was called.
- `SEXP cloenv`: This is a reference to the environment in which the function is enclosed. In R, every function is associated with an environment that serves as the context for evaluating its variables, enabling lexical scoping.

3.2.1 Built-in Functions

Almost every language has built-in functions. Built-in functions typically offer basic functionality. They are pre-defined and provided by the language itself rather than either user-defined or imported from a library. Built-in functions in GNU R are implemented in C, and they are treated differently from user-defined functions. Built-ins bypass the creation of an `RCNTXT` structure, and instead, they are executed within the current execution context.

This is, of course, more efficient. For executing a built-in function, there is `PRIMFUN` macro, to which we supply a SEXP with the function, and it finds the corresponding C code meant to be executed. To be more specific, there is an array of `FUNTAB` structures. `FUNTAB` contains things like the string representation of the name, the arity of the function and most importantly, the associated C function.

Chapter 4

Implementation

In this chapter, we describe our implementation of the proof-of-concept R profiler. Inspired by Berger's Scalene, we aim to use our profiler to distinguish between R and native execution. We hope that with this knowledge, the output of our profiler can lead programmers to replace some inefficient R execution with optimized native execution.

The starting point for writing a profiler is slightly different for Python compared to R. Python already supports user-defined signal handling, but R does not. Therefore, we include a section on how we added it to the interpreter. Furthermore, in Python, there is a library to asses which line of code is currently executing. We must implement that for R ourselves.

Because this is a proof-of-concept and to support everything would be out of the scope of this thesis, we have taken a few shortcuts. We implement the profiler directly into GNU R version 4.3.3, we support AST interpretation only, and we aim at functions rather than profiling whole files.

We go step-by-step through our modifications. We made changes to three files: `src/main/eval.c`, `src/main/main.c`, and `src/include/Defn.h`. For simplicity, we will refer to them only by their name without the full path.

4.1 Signal Handling

Our goal for this R profiler is to distinguish between R execution and native execution. Inspired by Scalene, we aim to achieve this through sampling, typically facilitated by signals. However, since the R language lacks support for custom signal handling, we need to integrate our own signal handler into GNU R [1]. In this section, we provide a comprehensive description of the integration. The implementation of signal handling is platform-specific for Unix-like systems.

Signals serve as a communication mechanism in software systems by enabling notification of a process of an asynchronous event. Signals can be sent by the operating system, by another process, or by the process itself. Upon obtaining

the signal on Unix systems, the kernel interrupts the ongoing process and transfers control to the designated signal handler function. The signal handler can be either default or user-defined. Either way, it is executed in the context of the interrupted process [18]. After the signal handler terminates, the previous context is restored, and the process continues.

Signals occur asynchronously, meaning that the signal handler can be invoked at any point during the process's execution [19]. Because signals require minimal overhead and enable us to stop the program at any point, they are considered the state-of-the-art solution to sampling profilers.

In our implementation, we aim to measure the CPU time of a given process. To this end, we utilize the `SIGVTALRM` signal in conjunction with `ITIMER_VIRTUAL`, which triggers this signal. We need to set our signal handler once at the beginning of the execution. Therefore, we logged the handler within the `setup_Rmainloop` function, as illustrated in 4.1.

```
1     struct sigaction sa;
2     sa.sa_handler = shandler;
3     sa.sa_flags = SA_RESTART;
4     sigemptyset(&sa.sa_mask);
5
6     if (sigaction(SIGVTALRM, &sa, NULL) == -1) {
7         perror("sigaction");
8     }
```

Figure 4.1: The code shows how we logged our signal handler. This was added into the `setup_Rmainloop` function in the `main.c` file. This initialization happens conditionally only when our profiling is enabled.

The `sigaction` structure's `sa_flags` member variable modifies the behaviour of a signal. Using the `SA_RESTART` flag, we ensure that certain system calls (like write or open) interrupted by the signal handler can restart once the handler completes, allowing the process to continue the system call seamlessly [20]. The `sa_mask` member specifies which signals can be delivered while the signal handler is being executed, and the `sigemptyset` sets the variable so that all signals can be delivered.

Once a signal is received, our signal handler sets variable `R_GotSignal`, as can be seen in 4.2, serving as a notification to indicate the occurrence of the signal. We added the definition of the `R_GotSignal` variable into the `Defn.h` file. The configuration of the timer can be seen in 4.3. We always set the alarm for a single subsequent alarm that is set off in `SIGNAL_INTERVAL`, which corresponds to 10 ms.

In summary, we added a signal handler for `SIGVTALRM`, which is logged at the beginning of execution within the start-up of R. Then we added a function which sets the `ITIMER_VIRTUAL` to send a signal in `SIGNAL_INTERVAL` which is set to 10 ms.

```
void shandler (int signum) {
    R_GotSignal = 1;
}
```

Figure 4.2: The code snippet shows our signal handler. Signal handlers should execute as fast as possible, ideally not using many objects created out of their scope. Therefore, our signal handler simply sets the `R_GotSignal` variable to indicate an occurrence of a signal.

```
void reset_timer ( void ) {
    struct itimerval timer = {
        .it_value = { .tv_sec = 0, .tv_usec = SIGNAL_INTERVAL },
        .it_interval = { .tv_sec = 0, .tv_usec = 0 }
        };
    if ( setitimer(ITIMER_VIRTUAL, &timer, NULL) == -1 ) {
        perror("reseting timer");
    }
}
```

Figure 4.3: This piece of code shows how we set when the next signal should be triggered. We use the `ITIMER_VIRTUAL` timer, which works as follows. The following signal is set off in time configured by `it_value`. The period of how often the consequent signals are sent is determined by `it_interval`. By setting `it_interval` to zero, we set the timer to only one subsequent signal [6]. We added this function into the `eval.c` file since that is the only place where we need to access it.

4.2 Design of the Profiler

To do a sampling profiler was a relatively easy choice. The alternative would be instrumentation, and although there are benefits to both approaches, the main benefits of sampling profilers are that they have comparatively low performance overhead and they are not as prone to Heisenbug because they do not change the course of execution [21].

Therefore, we decided to write a sampling profiler, where we implement the trick that is used in Scalene to differentiate between R and native execution. Specifically, we attribute the delay in handling signals to native execution. The building ground for such a profiler is different for Python and R. Python already supports user-defined signals, while we needed to implement it into the R interpreter as is described in 4.1. Furthermore, there is a library to help us determine which line of Python code is executing, called Frame Objects [14]. There is no such library for R, so in Section 4.3.2, we describe in detail how we solved that. Briefly said we attribute time to `LANGSXP`s rather than lines when we take a sample. Regardless, we must find the appropriate line of code for each `LANGSXP` for the sake of having a readable output.

Moreover, in contrast with the Python profiler, we focus on profiling functions rather than files at this moment. This is a shortcut for the sake of simplicity, in future, this feature can be added. The code we want to profile must be in

`..my_profile..` function that is called at some point. If we insert a function's name into the arguments to the `..my_profile..` call, then we will profile that function as well.

Furthermore, this proof-of-concept profiler works so far only on the AST interpreter. The employed technique should work for bytecode interpretation in the same way. Therefore, we must disable just-in-time compilation into bytecode. That can be either done via environment variable `R_ENABLE_JIT` set to zero or in code with `compiler::enableJIT(0)`. To run our profiler we added a `R_SCALENE` environment variable. The user should set it to the name of the desired output file.

4.3 Initialization of the Profiler

When we run our profiler, there are a few things that must be done apart from the initialization of R.

Firstly, we extract from the environment variable `R_SCALENE` the name of the file, where we should store the profiling result. Sampling profilers, such as ours, periodically stop program execution to take samples. That is typically done using signals. Therefore, we log our signal handler for the `SIGVTALRM` and use the `ITIMER_VIRTUAL` which generates these signals.

Next, we need to set up the internal structures for the profiler. To obtain which function is currently executed, we need to use the info from the call stack. Because attributing lines to functions will merge together calls from different call sites, we decided to attribute time directly to the call sites which are represented by `LANGSXP`s. So, we need a container where we can access the R and native counters by indexing by `LANGSXP`s.

4.3.1 Profiler State

The container we used to represent the profiler state is a hashmap uthash [22]. It is a header-only implementation of a hashmap for C structures. The `map_entry_struct` structure, which can be seen in Figure 4.4, contains three values: key, value, and hash handle, which is used by uthash methods. In our case, the key is a SEXP, in other words, a pointer to `SEXPREC` structure. The value is a structure which consists of three values: R counter, native counter, and line number. Initially, R and native counters are equal to zero. Line number refers to which piece of R code is represented by the `LANGSXP`. Naturally, the line number does not change after initialization.

To work with the hashmap, we use macros defined in the `uthash.h` file. Some of the macros have multiple versions depending on the type of the key. Since our key is a pointer, we must use the pointer-specific macros. The `R_LANGSXPMap` variable is our entry point to the whole hashmap. We do not work with it directly, but instead only through the hashmap macros. For instance, in Figure 4.5, we can see the code for updating an entry in the hashmap, which demonstrates how the hashmap macros can be used.

```c
#include "uthash.h"
typedef struct {
    unsigned int r_counter;
    unsigned int c_counter;
    unsigned int line_number;
} counter_struct;

typedef struct {
    SEXP key;
    counter_struct value;
    UT_hash_handle hh;
} map_entry_struct;

map_entry_struct* R_LANGSXPMap = NULL;
```

Figure 4.4: This snippet shows the definition of our hashmap entry. Our key is a SEXP, i.e. a pointer to `SEXPREC` structure. Our value field contains the R and the native execution counter, and the line of code which corresponds to the SEXP key.

```c
void update_map_entry(SEXP key, counter_struct value) {
    map_entry_struct *s;
    HASH_FIND_PTR(R_LANGSXPMap, key, s);
    if (s == NULL) {
      s = (map_entry_struct*)malloc(sizeof *s);
      s->key = key;
      HASH_ADD_PTR(R_LANGSXPMap, key, s);
    }
    s->value = value;
}
```

Figure 4.5: This code snippet shows how we can update a value in our hashmap. If the value of the key argument is already present in our hashmap, then we only update its value to the value given as an argument. Otherwise, we make a new entry into our map with the key and value given by the arguments.

Now, it remains to initialize the hashmap with all the `LANGSXP`s we need to track. There is a switch over `SEXPTYPE`s in the `eval` function in the `eval.c` file. In the `LANGSXP` case of the switch, we check whether the name of the currently called function is `..my_profile...` If it is and profiling is enabled, then we walk the AST of the `..my_profile..`, and we add each `LANGSXP` that we encounter into our hashmap.

4.3.2 Determining Lines of Code

Walking a function's AST in order to obtain all its `LANGSXP` is fairly simple. However, this task gets slightly more complicated because we need to pair each `LANGSXP` with its corresponding line of R code.

To determine line numbers for each `LANGSXP`, we need to work with `srcref`. The `srcref` object is part of a source reference system for R code that helps us to keep track of code location that corresponds to given SEXP nodes. We

store the `srcref` as an attribute of an object. The `srcref` defines a span of characters of code in a `srcfile`, which represents the file name [1].

Because R does not keep track of `srcref` in every mode of use (for instance, in interactive mode, it does not make much sense), we run the profiled file with `source("filename", keep.source = TRUE)`. The `source` function is used to execute scripts, and the `keep.source = TRUE` parameter ensures we track the `srcref`s in the file.

The `srcref` object typically contains an `INTSXP` with 8 values, such as first line, last line, first column, and last column. We will only use the first value in the thesis, i.e. the number that denotes the first line.

Thus, there is a way to obtain the lines of code from the SEXP representation. However, it is not as if each SEXP node has a corresponding `srcref`. Functions do have the `srcref` attribute. Nevertheless, their `srcref` encompasses the whole function, and there is no way to extract more granular data. Luckily, we can use the `{` function call (in R code, it simply looks like a block of code).

The `{` function call takes as arguments the expressions within the block as was shown in Section 3.1. For each of its arguments, it contains a `srcref`. These `srcref`s contain the span of the characters of the individual expressions. So when we walk the function's AST to obtain all its `LANGSXP`s, we must pair each argument of the `{` function call with its `srcref`.

In Figure 4.6, we can see how we attribute lines of code within the profiler. This code is part of a function `make_map_from_AST` that we added. It takes two arguments: a SEXP and a line of code. If the current SEXP is not of the `LANGSXP` type then we simply skip. Otherwise, we check whether the current `LANGSXP`'s `CAR` value points to a `SYMSXP` that represents a `{` function call. If it does, then we further check that the `LANGSXP` contains `R_SrcrefSymbol` attribute (in other words, a `srcref`).

If we follow this branch of code, then, finally, we can extract the `srcref`s. The `Rf_getAttrib` function on line 2 in Figure 4.6 takes two SEXP arguments. The first argument is the element whose attributes we inspect (`e` at that point represent the `{` call); the second is the attribute we need (`R_SrcrefSymbol` represents `srcref`). For each argument of the `{` call and the call itself, there is a `srcref`. Because we need to attribute a line number to all of them, we store the `srcref`s in the `srcrefs` variable. Firstly, we inspect the `srcref` that belongs to the `{` call because we want to add it to the hashmap. We access the `srcref` at index 0 because it belongs to the call itself. We use for that the `VECTOR_ELT` function, which takes a vector-like SEXP and an index into the vector and returns the element found at the index. As was said before, `srcref` typically has an `INTSXP` vector with 8 values. We extract the value at index 0, since it denotes the first line of the expression (for simplicity for each `LANGSXP` we only store where it begins). Then, we add the `LANGSXP` for the `{` call into the hashmap.

For the arguments (lines of the block), we do it analogously, except we do not add it into the hashmap (that is done by the recursive call on line 19 in Figure 4.6). First, we extract the appropriate `srcref`, from which we obtain the first line of code that the SEXP corresponds to. Then, we recursively call the function on the arguments and the newfound line number.

For function calls other than `{`, we do not inspect their `srcref`. Instead we simply assign the line number that was passed as the argument to the `make_map_from_AST` function. Again, we recursively call the function with the same line number on all its arguments.

```
1  //extract all srcrefs
2  SEXP srcrefs = Rf_getAttrib(e, R_SrcrefSymbol);
3  //extract srcref for the "{" call
4  SEXP numbers = VECTOR_ELT(srcref, 0);
5  //extract the line number at which the "{" call starts
6  int line_number = INTEGER(numbers)[0];
7
8  counter_struct value = {.r_counter = 0,
9                          .c_counter = 0,
10                         .line_number = line_number};
11 update_map_entry(e, value);
12
13 //walk the arguments of the "{" call and assign them their lines
14 e = CDR(e);
15 int i = 1;
16 while (e != R_NilValue) {
17     SEXP numbers = VECTOR_ELT(srcref, i);
18     int line_number = INTEGER(numbers)[0];
19     ++i;
20     make_map_from_AST(CAR(e), line_number);
21     e = CDR(e);
22 }
```

Figure 4.6: This code segment is part of a function whose signature is `void make_map_from_AST (SEXP e, int line)`.

However, this algorithm has a drawback. If there is a function call with function calls as arguments and they span over multiple lines, then we assign the wrong line number to the argument function call. Figure 4.7 shows an example of such a case. We assign each of the argument function calls the line number where the outermost function call begins. To fix this, we would need to work with the parse table generated by the R interpreter.

In summary, we call the `make_map_from_AST` function on the body of the `..my_profile..` function and all functions that were passed as arguments to it. So, at the end of the initialization, we have a hashmap with `LANGSXP`s to which we want to attribute CPU time, with counters set to zero and the line numbers of code where the function calls started in the original R source code.

There are a few last things we need to do after we create the hashmap with `LANGSXP` keys. First, we log a function to print and then delete the hashmap to run at the end of the execution. We use the `atexit` function [23] for that.

```
1 foo ( bar(3),
2       "argument",
3       c(1,5,8) )
```

Figure 4.7: In this code segment we can see a function call which spans over multiple lines. The current implementation will not correctly assign line numbers to function calls that are arguments. To the `foo` function call we correctly assign line number 1. And then we assign it recursively to both `bar` and `c` function calls. For `bar`, it is coincidentally correct; for `c`, it is not, we would expect line number 3.

It takes one argument, a function pointer, and at the end of a successful exit, it runs the function.

Then, we set the `R_SubtractTime` variable to the current time. How and why we do that will be covered in the next section. Finally, we start the `SIGVTALRM` alarm for the first time and let the rest of the `..my_profile..` execute.

4.4 Taking Samples

In the `eval` function in the `eval.c` file we check whether the `R_GotSignal` is set. If it is, we proceed to take a sample, which we can see in Figure 4.8. We place the check at the end of the `eval` function. That is different from Scalene, which places it at the beginning of its eval function. This difference is due to the different granularity at which the eval function works. Python interpreter works with bytecode. R has got both AST and bytecode versions of the function, but so far, we only added profiling to the AST version. There, the signal handling needs to be at the end of the `eval` function because if it was at the beginning, we would attribute the execution time of the previous `LANGSXP` to the next `LANGSXP`.

First, we use the `GET_CURRENT_TIME_MS` macro to store the current time in milliseconds. The macro from line 3 in 4.8 can be seen in 4.9. Then we call the `get_current_entry` function which walks the call stack and tries to find a `LANGSXP` that is in our hashmap (the same hashmap as from Section 4.3.1).

4.4.1 Walking the Call Stack

To walk the R's call stack we use the `get_current_entry`. We wrote it based on the `R_GetTracebackOnly` function in `errors.c` file. We walk the stack until we reach the top-level context. At each level, we check whether the `c->call` SEXP is present in our hashmap. If it is, we exit the function and return a pointer to a `map_entry_struct` with the `c->call` as its key. Otherwise, the function returns `NULL`.

```
if (R_GotSignal == 1) {
    //log the current time
    long long new_signal_time;
    GET_CURRENT_TIME_MS(new_signal_time);
    //try to find a map entry using the call stack
    map_entry_struct *s = get_current_entry();

    //if we have not found an entry
    //we try the currently evaluated SEXP
    if ( s == NULL ) {
        s = find_map_entry(e);
    }
    if ( s != NULL ) {
        //the signal interval
        long long q = SIGNAL_INTERVAL / MICRO_IN_MS;
        //the elapsed time
        long long T = new_signal_time - R_SubtractTime;
        //attribute the times
        s->value.r_counter += q;
        s->value.c_counter += T - q;
        R_TotalSignalTime += T;
    }
    R_GotSignal = 0;

    // post-process signal
}
```

Figure 4.8: This code is placed at the end of the `eval` function. It checks whether a signal was received in the meantime, and if it was, then it proceeds to take a sample. First, it stores the current time. Then it walks the stack to find a `LANGSXP` to which it could attribute time. Finally, it calculates the additions to both counters. After this snippet there is some post-processing to restart the timer etc.

```
#define GET_CURRENT_TIME_MS(value) \
    do { \
        struct timeval tv; \
        gettimeofday(&tv, NULL); \
        (value) = (((long long)tv.tv_sec) * MS_IN_S) \
        (value) += (tv.tv_usec / MICRO_IN_MS); \
    } while(0)
```

Figure 4.9: This macro stores the current time in milliseconds in the `value` parameter. We use the platform-dependant `gettimeofday` function defined in `sys/time.h`. The time is the number of milliseconds since the beginning of Epoch. For the sake of best programming practises we encapsulate the code in `do-while` construct [24].

```
1  map_entry_struct * get_current_entry () {
2      RCNTXT *c;
3
4      for (c = R_GlobalContext ;
5           c != NULL && c->callflag != CTXT_TOPLEVEL;
6           c = c->nextcontext){
7              map_entry_struct *s = find_map_entry(c->call);
8              if (s != NULL){
9                  return s;
10             }
11     }
12     return NULL;
13 }
```

Figure 4.10: This function is defined in the `eval.c` file. It is based on the `R_GetTracebackOnly` function. At each stack frame, we check whether the function call that created it is in our hashmap.

If we do not find an entry in our hashmap when we walk the call stack, then we check whether the currently evaluated SEXP is in the hashmap, as can be seen on lines 6 and 7 in 4.8, where `e` is the current SEXP. That can happen when we just executed a `LANGSXP`. Therefore the call context is not on the stack anymore because it was completed. Luckily the `e` variable stores it in that case.

Finally, once we find an appropriate entry in the hashmap, we can attribute time to the counters as illustrated on lines 10 to 13 in 4.8. To the R counter, we add the period (q) at which we send signals, in our case, 10 ms. For native execution, we calculate the elapsed time (T) since we set the alarm by the `reset_timer` function and subtract from it the interval period. In other words, we add $T - q$ to the native execution counter. Finally, we update the total profiling time `R_TotalSignalTime`. We store it so that we can give the percentages of time spent at each function call.

However, as we mentioned earlier, built-in functions do not create their own context. Let us demonstrate the problem on code 4.11 (the code is not profileable, it runs too quickly). So, let us assume that we handle a signal at the `7 * 42`. But when we walk the call stack, we will only see the `..my_profile..`'s context. Therefore, we do not know where to attribute the time. Fortunately, Rprof had to solve the exact same problem. It introduced a flag `R_Profiling` that, if it is set, built-in functions do create their own contexts too. So at the initialization stage of our profiler, we set the variable.

```
result <- 0
..my_profile.. <- function (...) {
    result <- 7 * 42
}

..my_profile..()
```

Figure 4.11: This code snippet is an example to demonstrate the trickiness of profiling built-ins. This code is not intended for profiling; it would execute too quickly anyway.

After we processed a sample, we set up the next alarm. So firstly, we set the `R_GotSignal` flag to zero. Then we set the `R_SubtractTime` to the current time. This way we do not account for the time we spent handling the sample. And finally, we run the `reset_timer` function, which will trigger a signal in 10 ms.

4.5 Conclusion

To sum it up our profiler employs sampling. We implement the sampling mechanism using signals. Due to the lack of built-in support for signal handling in R, we implemented our own. When the environment variable `R_SCALENE` is set, we register our signal handler for `SIGVTALRM` during the initialization stage in `setup_Rmainloop`. This signal handler simply sets a flag to indicate that a signal was received.

During execution, when we encounter a function call on the `..my_profile..` function, we walk the AST of the function and of all functions that were passed as an argument to it. We add all nodes of type `LANGSXP` (which represent a function call) that we encounter to a hashmap. Then we set the first signal alarm. After this initialization we proceed normally, except for that at the end of the `eval` function, we check if the signal notification flag was set. If it was, we walk the stack until we find a `LANGSXP` that is key in our hashmap and add to the corresponding R counter the timing interval and to the native counter the difference between the elapsed time and the interval. Then, we restart the `ITIMER_VIRTUAL` timer and let the execution continue.

Finally, at the end of execution, we store the profiling results in a file. This file documents the absolute time spent in each function and the relative percentage of the consumed time.

Chapter 5

Assesment

In this chapter, we assess the proof-of-concept profiler implemented in this thesis. To answer whether the chosen approach could be useful, we need to look at the accuracy of the profiler by determining whether we get meaningful results. Furthermore, we must inspect the performance overhead to see whether the profiler could be used on real-world code.

We will inspect the outputs of the profiler on a program where we know approximately how it should behave. Next, we will compare the outputs of Rprof with those of RScalene and compare their performance.

All the measurements were taken on MacBook Pro 2021 with Apple M1 Pro chips and 32GB RAM on LIMA [25], a Linux container, with default configuration [4]. For clarity, we will call our profiler RScalene in this chapter.

5.1 Accuracy

Let us inspect the accuracy of the implemented profiler by studying its behaviour on one program in a little more depth. To this end, we needed to design a program where we could relatively accurately approximate where the execution spent time. The program can be seen at 5.1. It calls two main functions in a loop. The code loops because sampling profilers need the code to run for some time to obtain reasonable outputs. One of the functions is written in R and the second in C in our custom package (their implementation can be seen in Appendix B). Both functions simply loop until a certain time has elapsed. The implementations actively loop instead of calling the `sleep` function because its implementation can work with signals and interfere with our profiling. The profiling output of RScalene to this code can be seen at 5.2.

[4] OS: Ubuntu, CPU: 4 cores, Memory: 4 GiB, Disk: 100 GiB.

```
1  ..my_profile.. <- function (...) {
2      lapply(1:100, function(i) {
3          waitR(1000)
4          waitC(1000)
5
6          waitR(100)
7          waitC(100)
8
9          waitR(10)
10         waitC(10)
11
12         waitR(1)
13         waitC(1)
14     })
15 }
```

Figure 5.1: This is an example of code we used to better understand RScalene's behaviour. The `waitR` function takes as an argument the number of milliseconds it should wait in the function call. It does not use any library calls. Similarly, the `waitC` function takes the number of milliseconds for which it should be executing. Both functions are implemented without sleep function, as that might interfere with our signals. There are four blocks of two consecutive lines. Each pair always calls the functions with the same time period argument.

When we add up the percentages of time spent in both functions separately, we can see that the profiler attributes around 48.5% to `waitR` and 51.5% to `waitC` of the entire execution time. We know that both of the functions should take around the same time because we call both of them with the same time period arguments.

In total, each loop should take around 2.222 s to execute ($1+1+0.1+\cdots+0.001$). One second is about 45% in 2.222 s. If we add up the percentages for the first call of `waitR` and `waitC`, then we get 44.3% and 45.8%, respectively, values we could expect. Analogously, for the second call with 0.1 s argument, we expect 4.5% execution time for both calls. We got 4.2% for `waitR` and 4.8% for `waitC`.

Our accuracy falters on measurements for lines 8 and 9. In the 100 iterations, 99 times, we received the signal during `waitC` skipping the `waitR` call and attributing almost all the time to the C function. The remaining lines, 11 and 12, are systematically not detected because their span of execution is so short, and they run just after a signal is received. However, the percentage of the execution time of these four lines is so low that it does not tell us much about systematic errors in the profiler, and further evaluation and analysis of more shortly running functions would have to be done.

```
line,    name,      "r ms",   "c ms",      r%,       c%
1,       lapply,         0,        0,   0.000,    0.000
1,       :,              0,        0,   0.000,    0.000
1,       function,       0,        0,   0.000,    0.000
1,       {,              0,        0,   0.000,    0.000
2,       waitR,      61890,    35943,  28.027,   16.277
3,       waitC,       1000,   100112,   0.453,   45.335
3,       ::,             0,        0,   0.000,    0.000
5,       waitR,       6000,     3275,   2.717,    1.483
6,       waitC,       1000,     9599,   0.453,    4.347
6,       ::,             0,        0,   0.000,    0.000
8,       waitR,         10,       15,   0.005,    0.007
9,       waitC,        990,      992,   0.448,    0.449
9,       ::,             0,        0,   0.000,    0.000
11,      waitR,          0,        0,   0.000,    0.000
12,      waitC,          0,        0,   0.000,    0.000
12,      ::,             0,        0,   0.000,    0.000
```

Figure 5.2: This figure shows an example output obtained by using RScalene on the code from Figure 5.1. The first column shows the line of code at which the function call starts. The second column contains the name of the function called. The third has the R counter values in ms. The fourth shows the time spent in native execution in ms. The fifth has the percentage of time spent in the function call in R compared to the entire execution time, and the sixth is analogous but for native execution.

Unsurprisingly, most of the time spent in the `waitC` function is in native execution. Perhaps unexpectedly, the profile output claims some time is spent in R. That is because we always add to both R and native counters to make up for the fact that we do not know when exactly the execution boundary was crossed. The R counter will always contain numbers rounded to 10 because we always add to it the interval period (10 ms).

The execution time proportions for `waitR` are more interesting. We can see more than a third of the time was spent in C. However, we wrote `waitR` so that it does not use any packages written in C. So the question is, how come there is all that time spent in native? The answer lies in built-in functions. They are implemented in C for better performance. So, the native counter corresponds, in this case, to the time spent in built-ins rather than libraries.

The key outcome of this analysis is that currently, we cannot differentiate between built-ins and library functions. This is a shortcoming for the potential users of the profiler because it will be harder to identify which code segments are running R code and can potentially be replaced by native functions. For the profiler output to be more useful to R programmers, we would need to add a third counter to distinguish between built-ins and libraries.

	Name	RS R %	RS C %	RS T %	Rprof T%	Rprof S%
p01	`waitR`	30.81	18.32	49.13	50.03	1.84
	`waitC`	1.50	49.37	50.87	49.97	49.97
p02	`doit2`	62.31	37.61	99.92	100.00	84.67
	`stuff`	0.04	0.02	0.06	100.00	0.00
p03	`mean`	51.57	31.35	82.92	84.21	14.05
	`boot`	2.73	14.16	16.89	100.00	0.37
p04	`lapply`	2.28	86.92	89.20	100.00	0.00
	`rnorm`	4.4	6.39	10.79	10.73	10.73
	`matrix`	0	0	0	11.53	0.79
p05	`foo`	31.18	19.07	50.25	50.11	0.10
	`bar`	31.01	18.74	49.75	49.89	0.00

Table 5.1: This table shows the most time-consuming function calls in five programs according to the profilers. The second column shows a function name. The next three columns show the percentage of time spent in R execution, in native execution, and the total according to RScalene. The next two columns show the total time spent in a function including nested function, then the time spent in a function without nested calls according to Rprof.

5.1.1 Comparing Accuracy

In this section, we compare the outputs of RScalene and Rprof using a few examples. For this prototype of RScalene, we only implemented profiling for AST interpretation. Therefore, we run the code for both profilers with just-in-time compilation disabled so that the two results are comparable.

When comparing the outputs of the two profilers, we must keep a few things in mind. Rprof always profiles all function calls by inspecting the call stack. In contrast, RScalene only attributes time to the first function call that is logged for profiling. So, there are functions that might be on Rprofs output but can never be in RScalene's. Moreover, RScalene profiles lines, so different call sites of the same function are distinguished. That is not the case in Rprof.

The programs are simple for easier analysis. Here, we report only function calls that had above 1% of the execution time and are visible in the code.

Let us inspect Table 5.1. Mostly, the percentages of time spent in each function according to Rprof and RScalene align. All the programs are listed in Appendix C.

- p01: This program is similar to the code in Figure 5.1. The total percentages of the time spent in the functions are very similar according to both of the profilers. Furthermore, we can see that `waitR` calls further functions by the difference in Rprof's total and self percentage. RScalene notifies us that a significant amount of `waitR` execution is in R.
- p02: There is a big difference between the profilers for `stuff` function. That is because RScalene does not walk up the call stack further once

it finds the innermost function call it can track. Since `doit2` function is called from within `stuff`, we do not find the `stuff` function call. However, some postprocessing on the RScalene's output could easily track these transitive relations.

- p03: The results are similar to p02. The `boot` function call takes as an argument a function which contains the `mean` call. Therefore, RScalene finds `mean` on the call stack and does not walk it further. All the other functions that were called as a part of `boot` get attributed its counters in RScalene.

- p04: The outcome is yet again similar to p02 since `rnorm`, `matrix`, and `lapply` call each other. R supports lazy evaluation. Therefore, that is the case even for `rnorm`, which is not a part of `matrix` body but still gets called from within it.

- p05: The program calls two similar functions. We can see that both of the profilers attribute them to about the same time percentage.

To summarize, the results of RScalene seem accurate when compared with Rprof. Of course, we have to interpret them a bit differently.

5.2 Performance

To assess the performance of RScalene we compare it with Rprof, the existing R profiler. We used five programs. Some of them were runnable codes from R packages, and some were benchmarks (if a program was too short, we looped it)[5]. Then, we ran the programs in three ways: once with our profiler, once with Rprof, and without any profiling. All of the runs had disabled just-in-time compilation because RScalene does not support profiling bytecode. Both of the profilers had the interval period between signals set to 10 ms. The results can be seen in Table 5.2.

	Unprofiled [s]	RScalene [s]	Rprof [s]	RScalene %	Rprof %
p01	180.66	182.51	190.16	1.02	5.26
p02	44.88	45.95	47.02	2.38	4.78
p03	169.33	171.26	179.22	1.14	5.84
p04	21.06	21.76	22.32	3.32	5.98
p05	36.79	37.55	38.96	2.07	5.90

Table 5.2: This table summarizes how long each of the programs ran. The second, third, and fourth columns show how long the program executed when profiled with RScalene, profiled with Rprof, and unprofiled, respectively. Then there is the relative slow-down compared to unprofiled execution calculated as $\frac{\text{profiled} - \text{unprofiled}}{\text{unprofiled}} \times 100$.

[5]The actual programs can be seen in the implementation in `measurements/performance` folder.

It is unsurprising that profiled executions take more time. Apart from the overhead of taking samples, both of the profilers cause contexts to be made for built-in functions, which is not done in normal R execution.

As we can see, RScalene has a lower percentage overhead than Rprof. This is probably due to the fact that Rprof operates on storing the whole call stack at each sample. Although it employs buffering, the I/O operations to store call stacks are still relatively expensive. The current implementation of RScalene is unoptimized as it is a proof-of-concept. Therefore, the performance overhead can be further lowered.

5.3 Conclusion

In conclusion, during the analysis of the profiler stand-alone or in comparison to Rprof, significant inconsistencies in RScalenes outputs were not found. Of course, we would need to further investigate how exactly this would work for more programs. However, we found out that built-ins can sum up to report a significant amount of C execution. Since built-ins are an integral part of R, it is misleading for them to be in the same batch as library calls. This distinction seems an important addition to the implementation.

Chapter 6

Conclusions

In this chapter, we turn to the aims of this thesis formulated in Chapter 1 and examine their fulfilment. We also present suggestions for future work to extend and improve this work.

First, we overviewed Rprof, a sampling profiler for R, in Chapter 2. Then, we examined Scalene, a profiler for Python that can distinguish between Python and native execution and serves as an inspiration for our profiler. Finally, in Chapter 4, we implemented the profiler, including determining which line of code is currently executed.

The objective was to create a proof-of-concept profiler that distinguishes between R execution and native execution. We implemented a sampling profiler inspired by Scalene, a profiler for Python. This was fulfilled in Chapter 4, where we showed the implementation of such a profiler written into GNU R 4.3.3.

In Chapter 5, we analysed the behaviour of our profiler on a program where we knew how it should behave. Then, we compared it to Rprof, the de facto standard for profiling R code.

6.1 Future Work

Furthermore, there are many areas in which we can extend the current implementation of the profiler:

- Adding support for profiling bytecode would make this tool usable in practice. Disabling just-in-time compilation for the purpose of profiling significantly affects the course of execution and the profiling output can, therefore, be misleading.
- Extensively researching the accuracy of the profiler on real-world code.
- Distinguishing between execution of built-in functions and libraries written in C/C++. This would make it even easier for R programmers to

identify which code segments can be written using native libraries.

- Support profiling files, rather than functions, for simpler use.
- Analyzing whether randomizing when the signals are sent improves accuracy. A paper by Mytkowicz et al. [12] proposes that this should make the profilers more robust and "actionable".
- Then there are further smaller changes for easier use: add graphical output, add support for Windows, etc.

Bibliography

[1] R Core Team. *The R Project for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2023, r Language Implementation, Version 4.3.3. Available from: https://www.R-project.org/

[2] Berger, E. D.; Stern, S.; et al. Scalene: a high-performance, high-precision CPU, GPU, and memory profiler for Python with AI-powered optimization proposals. Available from: https://github.com/plasma-umass/scalene

[3] What is R? https://www.r-project.org/about.html, accessed: 2024-5-5. Available from: https://www.r-project.org/about.html

[4] Project Jupyter. https://jupyter.org/, accessed: 2024-5-5. Available from: https://jupyter.org/

[5] R Core Team. system.time: Measure Execution Time of R Expressions. R Documentation, 2019, accessed: 2024-05-07. Available from: https://www.rdocumentation.org/packages/base/versions/3.6.2/topics/system.time

[6] Unix. getitimer(2) - Linux manual page. https://man7.org/linux/man-pages/man2/setitimer.2.html, accessed: 2024-4-14. Available from: https://man7.org/linux/man-pages/man2/setitimer.2.html

[7] Rprof: Enable Profiling of R's Execution. https://rdrr.io/r/utils/Rprof.html, accessed: 2024-5-4. Available from: https://rdrr.io/r/utils/Rprof.html

[8] R: Summarise Output of R Sampling Profiler. https://stat.ethz.ch/R-manual/R-devel/library/utils/html/summaryRprof.html, accessed: 2024-5-4. Available from: https://stat.ethz.ch/R-manual/R-devel/library/utils/html/summaryRprof.html

[9] Berger, E. D.; Stern, S.; et al. Triangulating Python Performance Issues with {SCALENE}. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, 2023, ISBN 9781939133342, pp. 51–64. Available from: https://www.usenix.org/system/files/osdi23-berger.pdf

[10] Manning College of Information and Computer Sciences. Team Led by Emery Berger Wins OSDI 2023 Best Paper Award. https://www.cics.umass.edu/news/team-led-emery-berger-wins-osdi-2023-best-paper-award, 29 Aug. 2023, accessed: 2024-4-30. Available from: https://www.cics.umass.edu/news/team-led-emery-berger-wins-osdi-2023-best-paper-award

[11] Strange Loop Conference. "Python Performance Matters" by Emery Berger (Strange Loop 2022). 6 Oct. 2022. Available from: https://www.youtube.com/watch?v=vVUnCXKuNOg

[12] Mytkowicz, T.; Diwan, A.; et al. Evaluating the accuracy of Java profilers. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, New York, NY, USA: Association for Computing Machinery, 5 June 2010, ISBN 9781450300193, pp. 187–197, doi:10.1145/1806596.1806618. Available from: https://doi.org/10.1145/1806596.1806618

[13] Berger, E. scalene, Discussion #801. https://github.com/plasma-umass/scalene/discussions/801, accessed: 2024-5-4. Available from: https://github.com/plasma-umass/scalene/discussions/801

[14] Frame Objects. https://docs.python.org/3/c-api/frame.html, accessed: 2024-5-4. Available from: https://docs.python.org/3/c-api/frame.html

[15] Siek, K. Everything You Always Wanted to Know About SEXPs But Were Afraid to Ask. https://gitlab.com/kondziu/everything-you-always-wanted-to-know-about-SEXPs/-/blob/master/2017-03-12-everything-you-always-wanted-to-know-about-sexp-but-were-afraid-to-ask.md, 12 Mar. 2017, accessed: 2024-4-15. Available from: https://gitlab.com/kondziu/everything-you-always-wanted-to-know-about-SEXPs/-/blob/master/2017-03-12-everything-you-always-wanted-to-know-about-sexp-but-were-afraid-to-ask.md

[16] R Core Team. R Internals. https://cran.r-project.org/doc/manuals/r-release/R-ints.html, accessed: 2024-4-15. Available from: https://cran.r-project.org/doc/manuals/r-release/R-ints.html

[17] Wickham, H. r-internals: Documentation for R's internal C API. Available from: https://github.com/hadley/r-internals

[18] Northern Illinois University. LINUX Signals. https://faculty.cs.niu.edu/~hutchins/csci480/signals.htm, accessed: 2024-3-13. Available from: https://faculty.cs.niu.edu/~hutchins/csci480/signals.htm

[19] Ingargiola, G. Unix Signals. https://cis.temple.edu/~ingargio/cis307/readings/signals.html, accessed: 2024-5-3. Available from: https://cis.temple.edu/~ingargio/cis307/readings/signals.html

[20] Flags for sigaction. https://www.gnu.org/software/libc/manual/html_node/Flags-for-Sigaction.html, accessed: 2024-3-20. Available from:

https://www.gnu.org/software/libc/manual/html_node/Flags-for-Sigaction.html

[21] Sampling Profiler - DelphiTools. https://www.delphitools.info/samplingprofiler/, 25 Feb. 2009, accessed: 2024-5-4. Available from: https://www.delphitools.info/samplingprofiler/

[22] Hanson, T. D. uthash: C macros for hash tables and more. Accessed: 2024-2-18. Available from: https://github.com/troydhanson/uthash

[23] C++ Documentation. atexit. https://en.cppreference.com/w/c/program/atexit, accessed: 2024-5-2. Available from: https://en.cppreference.com/w/c/program/atexit

[24] gettimeofday(2) - Linux manual page. https://man7.org/linux/man-pages/man2/gettimeofday.2.html, accessed: 2024-5-3. Available from: https://man7.org/linux/man-pages/man2/gettimeofday.2.html

[25] Suda, A.; contributors. Lima: Linux-on-Mac. https://github.com/lima-vm/lima, 2023, software available from Lima GitHub repository. Available from: https://github.com/lima-vm/lima

[26] PRL@PRG. argtracer. Available from: https://github.com/PRL-PRG/argtracer

[27] Wickham, H. R's C interface, Advanced R. http://adv-r.had.co.nz/C-interface.html, accessed: 2024-5-4. Available from: http://adv-r.had.co.nz/C-interface.html

Appendix A

Acronyms

AST Abstract Syntax Tree

GC Garbage Collection

SEXP S-Expression

Appendix B

Implememntation of Waiting Functions

In Chapter 5, we mentioned functions that wait for a certain amount of time without using `sleep` functions. Here can be seen their implementation.

```
waitR <- function(interval ) {
    start_time <- Sys.time()
    elapsed_time <- 0
    while (elapsed_time < interval) {
        result <- sum(1:100000000)
        end_time <- Sys.time()
        elapsed_time <- end_time - start_time
    }
}
```

Figure B.1: This function is written in R that takes as an argument the number of seconds it should loop for.

```
SEXP waitC( SEXP num ) {
    struct timeval tv;
    gettimeofday(&tv,NULL );
    long long start = (((long long)tv.tv_sec) * 1000) + (tv.
    tv_usec / 1000);
    int should_end = 0;
    long long now;

    double interval = Rf_asReal(num);

    while (!should_end) {
            gettimeofday(&tv,NULL );
            now = (((long long)tv.tv_sec) * 1000) + (tv.tv_usec /
    1000);
            if (start + interval <= now){
                    should_end = 1;
            }
    }
    printf("baf\n");

    return R_NilValue;
}
```

Figure B.2: This function is written in C into an R package. It takes as an argument the number of milliseconds it should loop for. The implementation of the package is based on [26, 27].

Appendix C

Programs for Evaluating Accuracy

In Section 5.1.1, we compare profiling outputs for five different programs. The programs are listed here.

```
foo <- function(interval ) {
    start_time <- Sys.time()
    elapsed_time <- 0
    while (elapsed_time < interval) {
    result <- sum(1:100000000)
    end_time <- Sys.time()
    elapsed_time <- end_time - start_time
    }
}

..my_profile.. <- function (...) {
    lapply(1:50, function(i) {
        foo(1)
        argtracer::trace_code(1000)
        foo(0.5)
        argtracer::trace_code(500)
        foo(0.1)
        argtracer::trace_code(100)
        foo(0.05)
        argtracer::trace_code(50)
        foo(0.01)
        argtracer::trace_code(10)
        foo(0.005)
        argtracer::trace_code(5)
        foo(0.001)
        argtracer::trace_code(1)
    })
}

..my_profile..()
```

Figure C.1: Code p01.

```
#testme.py from scalene
..my_profile.. <- function (...) {
doit1 <- function(x) {
  y <- 1
  x <- (0:100000)[99999]
  y1 <- (0:200000)[199999]
  z1 <- (0:200000)[299999]
  z <- x * y * y1 * z1
  return(z)
}

doit2 <- function(x) {
  i <- 0
  z <- 0.1
  while (i < 100000) {
    z <- z * z
    z <- x * x
    z <- z * z
    z <- z * z
    i <- i + 1
  }
  return(z)
}

doit3 <- function(x) {
  z <- x + 1
  z <- x + 1
  z <- x + 1
  z <- x + z
  z <- x + z
  return(z)
}

stuff <- function() {
  x <- 1.01
  for (i in 1:20) {
    cat(i, "\n")
    for (j in 1:20) {
      x <- doit1(x)
      x <- doit2(x)
      x <- doit3(x)
      x <- 1.01
    }
  }
  return(x)
}
print("TESTME\n")
stuff()
}
..my_profile..()
```

Figure C.2: Code p02.

```
foo <- function () {
    lapply(1:25, function(i) {
        # Load the boot library
        library(boot)

        # Generating some random data
        data <- rnorm(1000)

        # Defining a simple statistic function
        statistic <- function(data, indices) {
          mean(data[indices])
        }

        # Timing the bootstrapping process
        start_time <- Sys.time()
        results <- boot(data, statistic, R=10000)
        end_time <- Sys.time()
    })
}
..my_profile.. <- function (...) {
    foo()
}

..my_profile..(foo)
```

Figure C.3: Code p03.

```
foo <- function () {
    lapply(1:30, function(i) {
        # Generating two large matrices
        matrix1 <- matrix(rnorm(1000*1000), nrow=1000)
        matrix2 <- matrix(rnorm(1000*1000), nrow=1000)

        # Timing the matrix multiplication
        start_time <- Sys.time()
        result <- matrix1 %*% matrix2
        end_time <- Sys.time()
    })
}


..my_profile.. <- function (...) {
    foo()
}

..my_profile..(foo)
```

Figure C.4: Code p04.

```
foo <- function() {
    for (i in 1:20000) {
         result <- sum(rnorm(1:i))
    }
}

bar <- function () {
    lapply(1:20000, function(i) {
         result <- sum(rnorm(1:i))
    })
}

..my_profile.. <- function (...) {
    lapply(1:5, function(i) {
        foo()
        bar()
    })
}

..my_profile..()
```

Figure C.5: Code p05.

Appendix D

Contents of Attachments

The project is also available at `https://github.com/sandbubbles/dp_code`.

- `README.md` ..... the file with media contents description
- `r` ..... R version 4.3.3. source code with our additions
- `measurements`
  - `performance` ..... scripts and data to compare performance
  - `r_vs_r` ..... scripts and data to compare profiling outputs
- `tests` ..... some programs for evaluation