Bachelor's Thesis

AN EVALUATION OF DIFFERENT MEASUREMENT APPROACHES FOR FEATURE PERFORMANCE ANALYSIS

JONAS KAUFMANN

December 7, 2022

Advisor:
Florian Sattler Chair of Software Engineering

Examiners:
Prof. Dr. Sven Apel Chair of Software Engineering
Prof. Dr. Sebastian Hack Compiler Design Lab

Chair of Software Engineering
Saarland Informatics Campus
Saarland University

UNIVERSITÄT DES SAARLANDES

Erklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis

Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken,_____ _____

(Datum/Date) (Unterschrift/Signature)

ABSTRACT

With an ever more growing user base, software systems require more and different functionalities. Developers implement features to offer functionality and satisfy these requirements. Additionally, modern software systems are highly configurable, allowing the user to tune the offered features to their concrete use-case. During execution, features can interact and lead to complex functional, as well as, performance bugs, which may only manifest when using a specific configuration [12] and are therefore hard to fix [19].

For the resolution of performance bugs, experts conduct performance analysis to identify slow code parts. The first step typically involves the use of an off-the-shelf profiler to gain an overview where time is spent during execution. The profile shows, in detail, which functions consume the most time. The issue is that off-the-shelf-profilers are unaware of features and their interactions. Developers struggle "looking for relevant information to identify influencing options [...] and trace the cause-effect chain [19]. In contrast, feature performance analysis uses measurements of the time spent per feature to aggregate the measured timing information into known concepts to users of configurable software systems, including non-developers like system administrators. To enable the measurement of features, current work on white-box performance analysis for configurable software systems [17, 18] inserts simple and static measurement code for regions. This instrumentation can lead to high overhead when these regions are executed with high frequency [17]. Additionally, current feature performance analysis approaches [17, 18] are designed for offline analysis of configurable software systems and don't allow measurements to be dynamically turned on or off or to be adjusted during execution without restarting the software system.

In this thesis, we evaluate two fundamental measurement approaches, namely tracing and sampling in the context of measuring feature performance. To solve the before mentioned issues, we leverage flexibility offered by state-of-the-art eBPF-based profiling to implement these. Finally, we conduct case studies to evaluate accuracy and overhead of these implementations using a mixture of real world and synthetic configurable software systems. We show that we can enable on-the-fly feature performance analysis with only 5 % overhead when measurements are inactive. Furthermore, we show that sampling is a promising approach to reduce high measurement overhead observed in current work.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

1 INTRODUCTION

Users of software systems require different functionality. To satisfy these requirements, developers add functionality in the form of features. For example, offering the functionality to print out a web page in a browser or to compress and encrypt data sent by a web server are features. In addition, modern software systems are highly configurable, offering configuration options, e.g., command-line switches, to turn certain features on or off and customize their run-time behavior. This configurability however, has a flip-side. It increases the complexity of a software system. Each configuration option adds additional possible paths in the source code, which the program's execution can take. Furthermore, configuration options can interact, for example, by executing some specific code only if both features are simultaneously active [2]. Features and configuration options are not only relevant when testing and debugging functional correctness, but also influence a software system's performance. Performance is a non-functional characteristic of software and typically quantified as throughput or response time to fulfill a certain task or total execution time. By enabling the features compression and encryption through configuration options in our example of a web server, we increase the response times for web requests.

Performance is an important non-functional quality of modern software systems. In a post-Moore era where hardware on its own isn't able to significantly increase performance from generation to generation without requiring changes to the software anymore, performance gains require writing more efficient code [3].

The industry is investing increasing resources to prevent performance regressions across releases [4]. Zaman, Adams, and Hassan [21] found that performance bugs take on average significantly more time to fix, are fixed by more experienced developers, and require changes to larger parts of the code than non-performance bugs. Compared to functional bugs, performance bugs don't produce a wrong result or crash the configurable software system. Additionally, they may only occur when using a certain configuration [12] and only be discovered once running in production using large workloads that highlight the issue. When discovered, they require performance debugging starting with performance analysis to figure out where time is spent during execution. The first step in performance analysis typically involves the use of an off-the-shelf profiler to measure time the CPU spends executing each function. This grants a quick yet detailed overview and provides places where to investigate further. Off-the-shelf profilers, however, are unaware of features and their interactions, which are involved in more than half of the performance bugs Han and Yu [12] investigated. Using feature unaware tools, developers struggle to find relevant information to identify features or their interactions causing a performance bug, as well as, where and how these affect the performance of the configurable software system [19].

Feature performance analysis aims to address this need for relevant information by measuring time spent per feature instead of function to provide information on how the active features contribute to the observed performance of a configurable software system. Features are a concept directly visible to all users of a software system. This also means that

they can be understood even by non-developers. Feature performance analysis aggregates timing information in this high-level concept.

To realize feature performance analysis, current work [17, 18] inserts measurement code, a process called instrumentation, to log the current timestamp whenever the execution enters or leaves a region in the source code relating to a feature. This instrumentation can lead to high overhead when such a region is executed with high frequency and slow the system under measurement down up to orders of magnitude [17]. When high overhead causes total execution time to be noticably longer, an otherwise significant performance difference between two configurations (one of them being slower due to a performance bug) can become insignificant compared to the total execution time. In addition to the issue of overhead, current work on white-box performance analysis for configurable software system [17, 18] is designed for offline analysis. For production use, we on one hand want to be able to measure at any time to collect information about short-term performance issues, for example, a momentary spike in observed latency. On the other hand, our software should execute as efficiently as possible to reduce operating costs. To achieve this, we need to be able to disable measurements most of the time to avoid paying the overhead. Current instrumentation approaches for feature performance analysis don't offer the functionality to dynamically turn measurements on or off. Moreover, the user has no way to influence overhead, for example, by reducing accuracy.

Profiling tools, which only induce a small amount of overhead and which allow flexible on-demand measurements, do already exist but not for feature performance analysis. We are going to use their fundamental ideas to offer the same flexibility and small overhead for feature performance analysis.

1.1 GOAL OF THIS THESIS

We aim to solve the current limitations of feature performance analysis by first discussing two fundamental measurement approaches state-of-the-art profilers use, namely tracing and sampling. Both fundamentally differ in terms of measurement overhead and accuracy, as well as the controllability they offer to the user. We then implement both and evaluate them using case studies on a mixture of real world and synthetic configurable software systems. Additionally, we also investigate the compromises we have to accept to achieve the flexibility state-of-the-art profilers provide.

Our results show that we can indeed offer on-demand feature performance analysis but at the cost of around 5 % overhead when measurements are inactive. This flexibility comes at the cost of higher overhead when measuring though. If we are willing to sacrifice accuracy, we can use sampling to enable feature performance analysis with negligible measurement overhead.

1.2 OVERVIEW

In Chapter 2, we discuss the necessary concepts we use throughout the thesis. This includes an introduction to features and configuration options. Feature performance analysis uses feature regions, which group statements in code that implement a feature. We want to measure the time spent in feature regions and therefore also talk about the two measurement

approaches tracing and sampling and the important concept of measurement overhead. We continue with state-of-the-art eBPF-based profiling and an introduction to VaRA, a framework which can automatically detect and insert measurement code into feature regions, which is required for performing feature performance analysis.

Chapter 3 describes how we can implement the introduced measurement approaches and minimize measurement overhead while doing so. Sampling is particularly hard to implement because we don't have direct access to feature regions and instead use call stacks to infer them. During our implementation, we also encountered potential limitations for the use of the eBPF-based profiling and discuss these.

We evaluate our implementations on a mixture of real-world and synthetic case studies in Chapter 4 to determine whether we have been successful in offering on-demand feature performance analysis while also reducing high overhead seen in current work [17]. We point out related work that inspired our ideas and used similar approaches in Chapter 5. We conclude the thesis in Chapter 6 and describe open future work to improve our implementations.

2 BACKGROUND

In this thesis, we implement two fundamental measurement approaches for feature performance analysis. Before diving into concrete details, we first discuss the concept of software features and configuration options. Based on that we introduce the concept of feature regions, a model to capture the statements in code, which implement a feature. Our goal is to measure the time a program spends in a feature region during its execution. We therefore continue by discussing two fundamental approaches how to measure them and highlight differences as well as theoretical limitations. We conclude the chapter by introducing additional tools we use for the implementation of the measurement approaches.

2.1 FEATURES AND CONFIGURATION OPTIONS

In today's world, users have many individually different requirements in terms of functionalities the software they work with has to offer. Developing customized software systems for each of them is expensive and error-prone. Instead, users turn towards off-the-shelf software, which is designed to serve a broad mass of customers. This approach allows the company building the software to focus its resources on a single instead of many individually tailored systems and hence reduces individual acquiring costs for customers while overall offering better quality due to the drastically larger user and developer base. [2] To appeal to a large range of users, off-the-shelf software has to offer the required functionalities while not limiting their use to just single individuals. To achieve this, today's software implements features and configuration options. "A *feature* is a characteristic or end-user-visible behavior of a software system" [2]. In 1998, Kang et al. defined it as "a distinctively identifiable functional abstraction that must be implemented, tested, delivered, and maintained". There are multiple other definitions, however, we aim to use features to explain observed performance, even to non-developers. As such, we care about their property of being end-user-visible, which includes all stakeholders of a software system.

As we have discussed, implementing just the required features is not enough to serve a wide range of users. For example, we can use different algorithms to implement a feature to compress payloads sent by our web server. There is a trade-off depending on which one we choose. Algorithm *a* may be faster but algorithm *b* achieves better compression ratios. Users may want one algorithm over the other for their concrete use-case. The solution is to implement both algorithms as features and then offer the choice to users through configuration options.

Modern software systems are highly configurable, allowing users to enable and disable certain features and tune their run-time behavior to their needs. In this thesis, we investigate load-time configuration options. Examples include command-line options and entries in configuration files. As the name implies, the software system loads the options when it is started and does not change them afterwards.

We now look at an example of features and configuration options in a real software system. Listing 2.1 shows some configuration options related to the *export* feature of a note-taking application. Configuration options control the features a software system offers. We can therefore look at configuration options to discover features a software system implements.

```
$ xournalpp --help-export
Usage:
  xournalpp [OPTION...]

Advanced export options
  -p, --create-pdf=PDFFILE          Export FILE as PDF
  -i, --create-img=IMGFILE          Export FILE as image files (one per page)
                                    Guess the output format from the extension
                                    of IMGFILE
                                    Supported formats: .png, .svg

  --export-no-ruling                Export without ruling
                                    The exported file has no paper ruling

  --export-range                    Only export the pages specified by RANGE
                                    (e.g. "2-3,5,7-")
                                    No effect without -p/--create-pdf or
                                    -i/--create-img
  --export-png-dpi=N                Set DPI for PNG exports. Default is 300
                                    No effect without -i/--create-img=foo.png
  --export-png-width=N              Set page width for PNG exports
                                    No effect without -i/--create-img=foo.png
                                    Ignored if --export-png-dpi is used
```

Listing 2.1: Export feature and some of its subfeatures in the note taking application Xournal++. The feature `export-range` depends on the feature to export an image or pdf file. `export-png-dpi` and `export-png-wdith` are mutually exclusive.

The example also shows that features can have relationships. This software system offers a subfeature for the feature *export* to create a PDF or an image file. Configuration options don't have to be just binary simply offering an on/off switch. The configuration option to export a specific range of pages takes a string as its input. Another observation is that features and by extension also configuration options can be mutually exclusive. The user can either provide `export-png-dpi` or `export-png-width`. Using both at the same time logically speaking makes no sense. There can also be other forms of dependencies. The two just mentioned configuration options require the feature to export to an image file to have an effect. The relations and dependencies of features can be captured using feature models. Figure 2.1 shows the feature model for the export feature of Xournal++.

For any configurable software system, we provide a configuration when executing it. A *configuration* contains configuration options and their assigned values. In the case of configuration options that toggle a feature, this value is limited to *on* or *off*. As a user of a software system, we usually don't specify all configuration options because the developer provides a default configuration. We only indicate the changed configuration options.

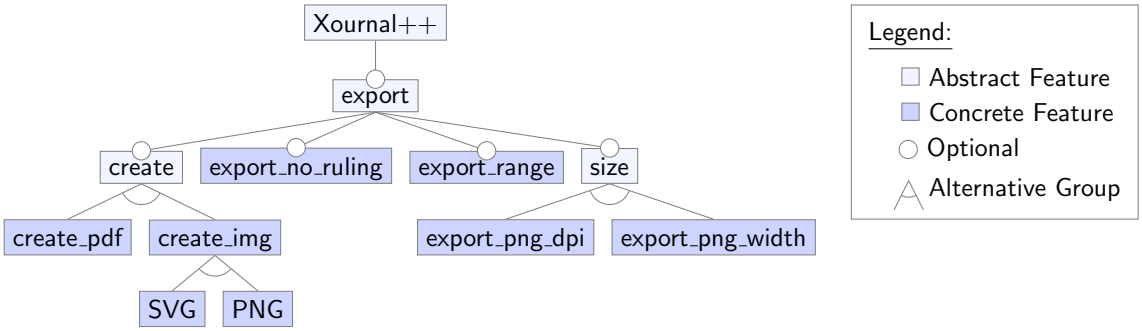


Figure 2.1: Feature model for the export feature of Xournal++. It encodes relationships and mutually exclusive groups. Abstract features are inserted to logically group functionality and to enable the expression of mutually exclusive features (alternative group). They don't actually represent a feature of the software system. We can form other constraints in addition to those encoded in the feature model from the configuration options. For example, `size` $\implies$ `PNG` and `export_range` $\implies$ `create`.

2.2 FEATURE REGIONS AND FEATURE INTERACTIONS

In the last section, we introduced the user-visible concept of features and configuration options. We now continue on the developer side discussing how features are implemented in code and how their behavior is influenced through configuration options. We introduce feature regions, which span a continuous range of source code and implement a feature. They are the basis for conducting feature performance analysis. We conclude the section by discussing how features can interact with each other and cause a performance issue that only occurs for specific configurations.

Load-time configuration options are parsed when the program starts and control the behavior of features by changing the control-flow during execution. The *control-flow* of a program is the order in which the individual instructions, statements, or function calls are executed. Source code isn't necessarily executed line after line. Instead, the execution can jump using control-flow statements like if-else, for-loops, break, or continue. To be able to influence the control-flow of a program, the parsed value of a configuration option is stored in some location to later be accessible from control-flow statements. Examples of these locations are local or global variables, struct fields, and data structures. For the software systems we are going to look at, we consider only variables and struct fields for simplicity. As discussed in Section 2.1, configuration options enable and control features. We therefore call variables and struct fields containing a configuration option's parsed value *feature variable*.

Listing 2.2 shows an example of a software system with two configuration options A and B, which can be enabled independently. Both options are represented in the program's source code with the feature variables `a` and `b`. The corresponding feature regions can be found using data-flow analysis by looking for control-flow statements that use a feature variable in their control-flow decision. All code that is executed as a consequence of this decision forms a feature region. In the case of an if-statement, this not only includes the if- but also the else-block because having a configuration option not enabled also influences the

runtime behavior by executing different code. If there is no else block, then the configuration option won't have an effect at the examined location when disabled.

```
1  int main(int argc, char *argv[]) {
2      bool a = getOpt("A", argc, argv);
3      bool b = getOpt("B", argc, argv);
4
5      bool x = false;
6
7      // Begin R1 (Feature A)
8      if (a) {
9          ...  // 2 seconds
10         x = true;
11     } else {
12         ...  // 1 second
13     }
14     // End R1
15
16     // Begin R2 (Feature A, B)
17     if (a && b) {
18         ...  // 3 seconds
19     }
20
21     // Begin R3 (Feature B)
22     if (b) {
23         ... // 1 second
24
25         // Begin R4 (Feature A)
26         if (x) {
27             ... // 1 second
28         }
29         // End R4
30     }
31     // End R3
32 }
```

Listing 2.2: A constructed example showing features A and B interacting in source code and their feature regions. The total execution time depends on which features are active. We can't simply add the execution time values of the two configurations $\{A\}$ and $\{B\}$ to predict the performance for $\{A, B\}$ due to the feature interaction in the regions R2 and R4.

The feature regions are marked with comments in Listing 2.2. This example shows some possibilities of how features can interact and cause performance issues to only arise in certain configurations. Regions R2 and R4 are only executed in the case where both features are active. R2 contains an explicit and wanted feature interaction. Region R4, however, is an example of a feature interaction which may be unwanted and easily overseen. We are not able to identify this relationship by investigating control-flow only, as the decision isn't directly based on the feature variable. When looking at possible control-flow paths the execution can take, we see that the value of x is influenced by that of the configuration option

A. Actually, it is only `true` if A is enabled. As a consequence, if `x` is used in control-flow decisions, it should form additional feature regions which belong to A. *data-flow analysis* performs the reasoning we just discussed algorithmically and can therefore be used to detect feature regions. We are going to use an existing framework to find feature regions.

2.3 ANALYZING PERFORMANCE THROUGH MEASUREMENTS

We discussed the concept of feature regions and challenges when trying to identify them. For performing feature performance analysis, we want to find out how much time we are spending per feature region. We now introduce two fundamentally different measurement approaches for this task. Both differ in how they collect the measurement data and, more importantly, how the system under measurement is influenced. The main comparison property is the overhead-accuracy trade-off. High overhead slows down the system under measurement and can also alter (perturb) its observed behavior.

2.3.1 *Overhead*

A central concept when comparing measurement approaches is *overhead*, the amount of additional execution time incurred to perform the measurements compared to running *dry* (i.e. without any measurements). The obvious consequence is a slowdown in performance or, in other words, that our system under measurement will take longer to perform certain tasks. Additionally, overhead can perturb the system under measurement and also our measurement results.

Imagine we want to trace the duration of two tasks. Task A takes a long time to complete, while task B is much shorter. Taking the measurements requires time, which means that the two tasks will take longer to finish. Suppose the measurement overhead is constant. From a relative view, the long task's execution time will increase only slightly, while the increase for the short task is much more significant.

The measurement overhead can perturb the overall time the system spends on each of its tasks when B is executed with a high frequency, for example, because it performs some basic operation. Task A executes only rarely. Due to the higher frequency, the measurement overhead is also paid often by task B. The time the software system spends on task B over the course of its whole execution therefore increases significantly while the time spent on A sees nearly no difference. We are now in a situation where the measurement overhead perturbs the system behavior.

To minimize the influence of this perturbation on our measurement results, we need to make sure that we only measure the original time spent on the two tasks and don't include measurement overhead itself in our results. In practice, even if we managed to realize the time measurements in a single instruction just before the first or last instruction of the code we want to measure, there would always be a small delay between the CPU executing these two. This means that we can't prevent perturbation from happening. The question is whether it is relevant in practice. Our goal when performing measurements therefore is to keep overall measurement overhead small since this also reduces the amount of perturbation.

2.3.2 *Tracing*

After discussing overhead, we now introduce the first fundamental measurement approach named tracing. Its measurements are event-based. An *event* is a "change in the system state that is relevant for the measurement of a given metric" [14]. By this definition, we as the user of a measurement tool decide what is considered an event. If we were managing a web server, we could choose the arrival of a request or a detected denial of service attack as an event. In the case of this thesis, we are interested in the metric execution time per feature region. For this metric, events are the program's execution entering and leaving a feature region. The idea of tracing is to execute measurement code whenever an event occurs. For example, we can write out the current timestamp to a file, indicate whether we entered or left a feature region and which region it was. By subtracting the timestamps we gain the amount of time we spent per feature region. An example of the result can be found in Figure 2.2. We can also access local variables, for example, inside the current function's scope, and write out their values as well. In the tracing context, the collected information is called *trace*, which is generally a list of events plus the measured information. [14]

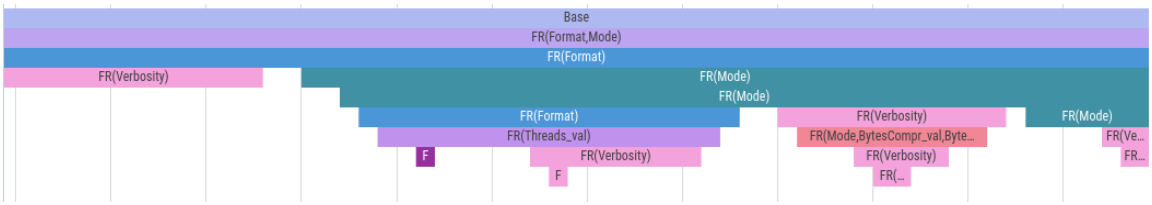


Figure 2.2: A zoomed-in excerpt of the *xz* feature region trace visualized using Perfetto. Feature regions are represented as start and end events including timestamps. The trace viewer is then able to automatically determine the nesting of complete (having start and end) events. The feature `Base` is active from beginning to the end of the program execution. The feature region above a feature region is its parent. By clicking on a single bar, we can see additional information, for example, the feature region's UUID and its measured duration.

For performing the measurements we need to insert measurement code. This process is called *instrumentation* and can be performed either statically or dynamically. For the former, the measurement code is inserted into the program's binary before starting its execution, while dynamic instrumentation modifies instructions during runtime. In this thesis, we are going to start off with static instrumentation for the measurement of feature regions, which we perform during the compilation process of a program. As a next step, we investigate how we can use state-of-the-art profiling tools to implement dynamic instrumentation which gives us greater flexibility since we are able to dynamically add and remove measurement code. However, this flexibility could come at the cost of additional overhead.

For tracing, the overhead is determined by the events' frequencies and the time necessary to execute the instrumented measurement code. Even if a single measurement code invocation takes negligible time, due to high event frequencies, overhead can become large enough to slow down the system under measurement up to orders of magnitude [17]. One possibility to combat overhead when we can't make the measurement code more efficient is to trace less frequent events at the cost of accuracy. Whether this is acceptable or not depends on the concrete situation (*overhead-accuracy trade-off*). Additionally, we may not be

able to find or define a lower frequency event. In that case, tracing doesn't offer any further adjustment possibilities. The dependency of overhead on the events' frequencies therefore poses a fundamental limitation.

2.3.3 *Interrupts*

In the following sections, we discuss the second fundamental measurement approach sampling and state-of-the-art profiling tools. Both use interrupts to execute measurement code, a concept we now introduce. Using interrupts also has implications on how overhead is induced and which information is available during measurements compared to inserting the measurement code directly into the software system, as seen for tracing.

Interrupts make the CPU immediately transfer execution to a pre-defined address in the kernel. Interrupts can be invoked from hard- and software. An example of software interrupts are system calls. When a user application performs a system call, it writes information like the system call number and additional information the kernel needs to pre-defined registers and the stack. To invoke the system call, an interrupt instruction is executed. The interrupt takes care of switching the CPU into kernel mode such that the kernel's code has full privileges to respond.

The operating system also uses timer interrupts to periodically interrupt the currently executing user program and regain control. They are used for performing periodic tasks, for example, scheduling another user application on the CPU. [1] Timer interrupts can also be used to periodically execute measurement code. The measurement approach we present in the following section uses this idea.

2.3.4 *Sampling*

Sampling aims to solve tracing's fundamental limitation by sacrificing accuracy. The idea of sampling is to execute external measurement code in equidistant time intervals. Its overhead is determined solely by the sampling frequency and the time to execute the measurement code. Consequently, sampling can conceptually offer lower overhead than tracing in the case of high event frequencies.

To realize sampling in practice, we need a way to periodically invoke the measurement code. This is done using timer interrupts. Due to this design, the measurement code is executed asynchronously to the system under measurement's code. This makes it impossible to measure events in the sense of tracing, where they represent a single point in time. Instead, we can only measure state, which has a certain duration. We need a way to expose the software system's current state to the measurement code.

One possibility is to manage externally visible state, for example, some constant address in the memory which is accessible by the external measurement code and whose content we continuously update during execution. Doing so slows down the system under measurement though and hence causes overhead. A popular alternative that doesn't suffer this overhead and that we are going to use is *call stack sampling*. It uses the call stack information automatically maintained on the stack by the CPU when executing a program. When a function is invoked, the return address is written to the stack. This is the address of the next instruction to execute in the current function once the invoked one returns. It is possible to

traverse these return addresses and doing so reconstruct all currently active functions or in other words the call stack [7]. Figure 2.3 gives an example of the result obtained by call stack sampling.

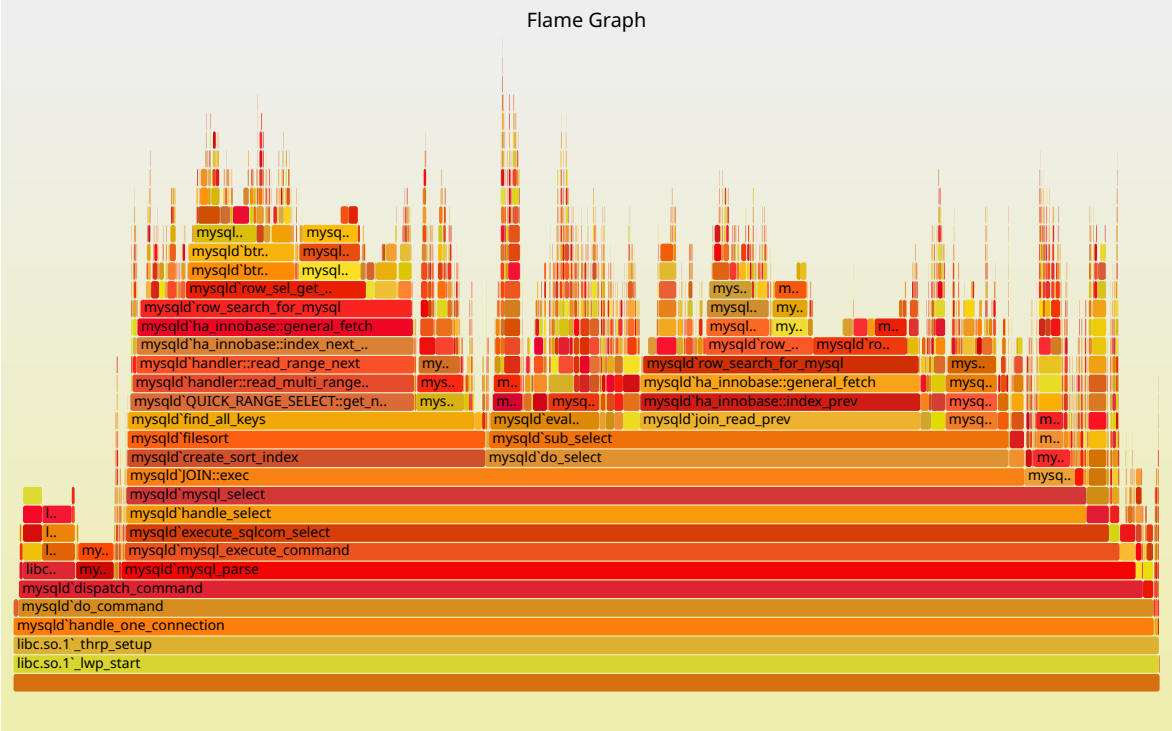


Figure 2.3: Flame Graph for the visualization of sampled call stacks from MySQL. Each horizontal bar represents a function. The bar directly beneath is its parent function in the call stack. The graph is colored from yellow to red, where red highlights functions that execute for a long time.
Source: https://www.brendangregg.com/FlameGraphs/cpu-mysql-updated.svg, visited 4th December 2022

Whatever we choose to represent our system's current state, we can't just access it asynchronously from the measurement code. This can lead to data races since the program might be in the process of writing updating the state while the measurement code simultaneously reads it. As a consequence, we need to synchronize to ensure that only one of the two can access the state at a time. To do so, we temporarily interrupt the execution of the program, execute the measurement code and then resume program execution [7]. This means that the amount of execution time required to execute the measurement code influences the measurement overhead.

In the context of tracing, overhead can lead to measurement perturbation. For sampling it exists as well, but for different reasons. We need to be careful when choosing the sampling rate in the presence of periodically occurring state to ensure we collect meaningful samples. Values like 97 Hz and 997 Hz are used to avoid sampling in lock-step with common frequencies of periodic tasks. If we were call stack sampling with 100 Hz and a short function was also invoked at the same rate, we might end up in a situation where all samples only show this single call stack, because we might always sample at the moment the function is invoked. [8]

Having discussed the conceptual idea of sampling and some important implementation details, we now turn towards its fundamental limitation, accuracy. Through the sampling frequency we not only control the overhead but also resolution. We can miss program states, which last a shorter amount of time than $\frac{1}{\text{sampling frequency}}$. Increasing the sampling rate therefore also increases the resolution, but we sacrifice overhead doing so. This is the *accuracy-overhead trade-off*.

Comparing sampling to tracing, sampling offers controllability of the accuracy-overhead trade-off, while tracing only allows us to measure less events, which might not be feasible in the concrete case. When sampling in practice, we are often interested in a *profile* of the software system, which provides a high-level summary of the execution behavior [14]. Since we can in principle always encounter state durations, which are shorter than a single sampling period, sampling can only provide an approximation. In addition, we might see different profiles when sampling the exact same execution of a program but when starting with a slight offset in time for taking the first sample. If we require to capture all events, we need to turn towards tracing.

2.4 STATE-OF-THE-ART PROFILING TOOLS

For the tracing of feature performance, our goal is to reduce overhead and offer more flexibility compared to the static instrumentation used in current work [17, 18]. As found by Velez et al. [17], the tracing of feature regions can induce significant overhead. When running a software system in production, performance issues may only be observable in a limited time frame because they depend on the current workload [11]. The possibility for high tracing overhead poses two issues though: First, we can't keep the instrumentation active all the time or else we have to pay additional operating costs and second, we may perturb the measured data noticeably. Even if we were able to restart the software system to activate the static instrumentation for detailed analysis on demand, the problematic workload would probably be gone. To solve these issues we investigate state-of-the-art profiling tools for tracing with two design goals: First, dynamic instrumentation with insignificant overhead when inactive and second, the option to only trace selected features with lower event frequencies to reduce measurement overhead.

We can use eBPF in conjunction with USDT probes to implement dynamic tracing. In addition, eBPF offers programmability of the measurement code to change the collected information on demand. It therefore provides a high level of flexibility to the user, which is useful for performance debugging. We explain all of these concepts in the following sections.

The idea of eBPF is to provide the ability to load programs which are executed in the kernel-space whenever a specified event occurs. Examples for these events are system calls, function entries and exits either in kernel- or user-space, network events, and also developer-defined locations in code called trace points. Like tracing, we have access to local information when the eBPF code is invoked, for example, when instrumenting a function entry we have access to the arguments the function is called with. Since eBPF programs are executed in the kernel, they need to be safe to run, that is, not crash or hang the kernel by executing indefinitely. A verifier is used to restrict what can be done in an eBPF program, for

example, it doesn't allow for unbounded loops. For what we want to do, mainly measuring time and writing out the data to a trace file, these limits don't impact us. [5]

For our implementation, we make use of trace points since we can place them at arbitrary locations in the source code, which especially includes entry and exit points of feature regions. We insert them during compilation into the software system of interest and make use of *USDT* probes (User Statically Defined Tracing) for that task. To add them to our software, we use macros provided by the header `sys/sdt.h` in the SystemTap project as shown in Listing 3.2. USDT probes work by placing a single `nop` instruction at the location where they are declared. When attaching an eBPF program during execution, the kernel will replace this instruction by an interrupt [9]. This process is called *activation of a probe*. The code handling the interrupt in the kernel then looks up and executes the attached eBPF program for the probe.

We already introduced the concept of interrupts in Section 2.3.4. For USDT probes, the code invoked in the kernel as a consequence of the interrupt has to identify which probe caused it and then invoke the attached eBPF program or function. When attaching an eBPF program, for example, the one shown in Listing 2.3, we need to specify the identifier of the USDT probe and what to execute as a response. The identifier of a probe and its location are stored inside the notes section in a program's executable binary with the ELF format. *ELF* is an abbreviation for Executable and Linking Format, which is a file format for executable binaries under Linux [13].

Figure 2.4 shows an excerpt of the contents in this section, which can be viewed on the command-line through `readelf -n <path_to_binary>`. We see four attributes which are of interest to us. `Provider` and `Name` are used to identify the USDT probe. We can choose both freely. The attribute `Provider` is useful to differentiate between probes with the same name, for example, in the situation where we are using common probe names and link against a library also declaring probes with the same name. `Location` stores the address where we can find the `nop` instruction. Using `objdump -d <path_to_binary>` to disassemble the code segments in the ELF file indeed shows a `nop` instruction at that address.

The attribute `Arguments` contains the location where the arguments passed to the USDT probe's can be found. Finally, there is the attribute `Semaphore` which we don't use. Suppose the arguments passed to our USDT probes aren't directly available and instead have to be constructed using a time-intensive operation first. Without a semaphore, we have to pay this cost independent of whether the probe is active or inactive. The semaphore allows us to check the status beforehand.

2.4.1 BPFTRACE

To write eBPF tracing code, we use a high-level language frontend called BPFTRACE. Writing eBPF code directly is cumbersome and makes the contents presented here unnecessarily hard to understand. We introduce BPFTRACE and some additional details of how dynamic tracing is realized, which is relevant for the later discussion on overhead.

When writing BPFTRACE code, we describe two components. One is the eBPF component running in the kernel, which collects tracing information and writes it to a storage object called map. The other is a user application, which asynchronously consumes entries in the

```
// RUN WITH: sudo bpftrace <path to this script> <path to binary to trace>

BEGIN {
    // Initialize maps.
    @num_probes_fired["begin"] = 0;
    @num_probes_fired["end"] = 0;
}

usdt:$1:vtrace:feature_begin
{
    @num_probes_fired["begin"]++;
}

usdt:$1:vtrace:feature_end
{
    @num_probes_fired["end"]++;
}
```

Listing 2.3: A simple BPFTRACE program counting the number of begin and end events of feature regions. `@num_probes_fired` is an eBPF map, which can also be read from user-space. `$1` inserts the first argument provided to the bpftrace program, in our case the path to the binary to trace.

```
$ objdump -d <binary>
[...]
0000000000226c40 <__vtrace_usdt_start>:
  226c40:   48 89 74 24 f0          mov    %rsi,-0x10(%rsp)
  226c45:   48 89 7c 24 f8          mov    %rdi,-0x8(%rsp)
  226c4a:   90                      nop
  226c4b:   c3                      ret
  226c4c:   0f 1f 40 00             nopl   0x0(%rax)
[...]
```

```
$ readelf -n <binary>
[...]
Displaying notes found in: .note.stapsdt
  Owner                Data size    Description
  stapsdt              0x00000044   NT_STAPSDT (SystemTap probe descriptors)
    Provider: vtrace
    Name: feature_begin
    Location: 0x0000000000226c4a, Base: 0x000000000020b730, Semaphore: 0x0000000000000000
    Arguments: 8@-8(%rsp) 8@-16(%rsp)

[...]
```

Figure 2.4: This figure shows how the USDT probes from Listing 3.2 are compiled into the ELF binary. A `nop` instruction is inserted at the location where the probe is declared. Additionally, an entry is added to the ELF notes section. It contains the locations of all USDT probes together with their provider and name, which form the identifier. It also specifies where the supplied arguments can be found.

map and writes them to a file. The application accesses the data in the map via system calls. Figure 2.5 visualizes the two components and their interactions.

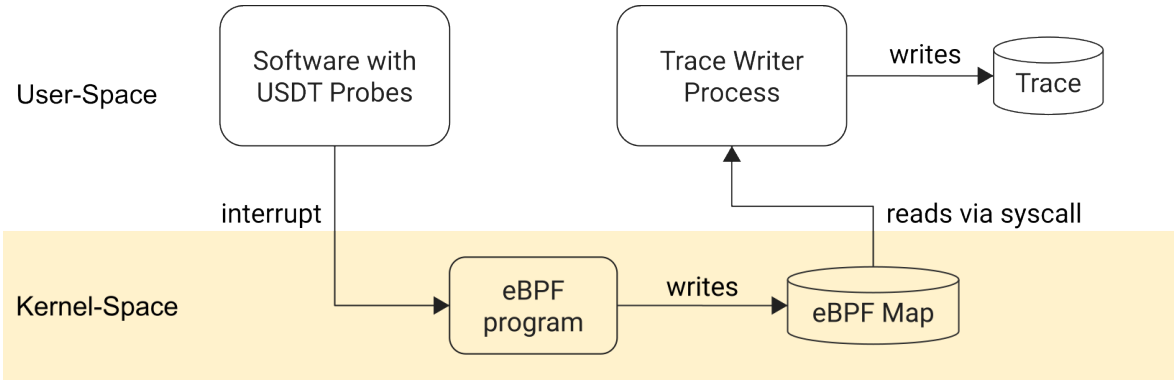


Figure 2.5: This figure shows the two components BPFTRACE produces when tracing an arbitrary software system with USDT probes and how they interact.

The design just described is intentional to reduce overhead while offering high flexibility. First, data is written asynchronously to a file. The execution of the software system under measurement can therefore resume faster after it invoked the interrupt for the USDT probe. Second, eBPF can reduce the overhead when computing statistical metrics which require a lot of data to be written to a file. The metric itself is then computed in a post-processing step after the execution of the software system under measurement has finished. eBPF can instead compute it efficiently in the kernel without writing loads of data to a file first. [10]

The overhead for eBPF tools comes from performing the interrupt to invoke eBPF code and then the execution time required for the eBPF code itself.

2.5 VARA

To measure feature performance, we use an analysis framework based on LLVM called VaRA (Variability-aware Region Analyzer). It offers the necessary data-flow analysis to determine feature regions along with an interface to instrument measurement code at entry and exit locations of feature regions.

VaRA is based on LLVM and modifies the clang compiler to perform feature region analysis and instrument feature regions during the compilation of a software system. [16] The feature region analysis itself needs a starting point in the form of the feature variable. We supply the location where the feature variable is declared in code using a feature model for the concrete software system. VaRA then uses data-flow and taint analysis to detect feature regions by tracking how values of these feature variables propagate throughout the program and where they are used in control-flow decisions.

VaRA also offers an interface to instrument the entry and exit points of feature regions. VaRA inserts calls to functions, which we implement. An example for this is shown in Listing 3.1. We can choose the provided arguments freely. Concretely, we use them to expose information about the current feature region like its unique identifier (UUID) and name.

3 IMPLEMENTATION

3.1 OVERVIEW

Our objective in this thesis is to evaluate different measurement approaches for feature performance analysis on a mixture of synthetic and real world software systems. We now discuss how to implement these approaches and which obstacles and potential issues we encountered doing so.

The basis for all measurement approaches is that we identified feature regions and have a way of instrumenting them. This task is handled by VaRA and its built-in data-flow analysis. The data-flow analysis requires the information which variables are feature variables. We provide this information through a feature model containing the declaration locations of the corresponding feature variables for each feature.

Next, to add instrumentation, VaRA already offers an interface by inserting function calls at the entry and exit locations of feature regions. We implement these functions by adding our measurement code as the functions' body. We can also decide which information about the feature region to provide as the functions' arguments and therefore expose to our measurement code. In the following sections, we implement tracing and sampling using this instrumentation capability and show concrete examples of how this can be done with VaRA's instrumentation interface.

3.2 STATIC TRACING

In this section, we describe the implementation of our tracing baseline, which uses static instrumentation. As discussed in Section 2.3.2, tracing is event-based. The events in our case are the execution entering and leaving feature regions. The conceptual idea for the instrumented code is the following: When an event occurs, write the current timestamp, a feature region identifier and whether it is an entry or exit event to a trace file. We use Google's *Trace Event Format*[1] because it allows us to later visualize the trace file using their viewer Perfetto[2]. A good example of a resulting trace can be found in Figure 2.2.

A naive implementation of tracing can suffer from unnecessary overhead. Writing to a file is performed using a system call, which in turn means writing values to specific registers or pushing them to the stack and issuing an interrupt [1]. As discussed in Section 2.3.3, this not only consumes processor time, but can also have other side effects, for example, cache pollution. To minimize overhead in the application's worker thread, we only push the information about an occurred event to a queue and use a separate thread to issue the system calls for writing to the trace file. Listing 3.1 shows the implemented functions for VaRA's instrumentation interface.

1 https://docs.google.com/document/d/1CvAClvFfyA5R-PhYUmn5OOQtYMH4h6I0nSsKchNAySU/preview
2 https://perfetto.dev/

```
1  static MeasureManager *MM = nullptr;
2  static const char *BaseFeatureName = "Base";
3
4  void __vtrace_tef_init() {
5    MM = new MeasureManager(GetFilename());
6    MM->enqueue_start(0, BaseFeatureName, GetPid(), GetTid());
7  }
8
9  void __vtrace_tef_finalize() {
10   MM->enqueue_end(0, BaseFeatureName, GetPid(), GetTid());
11   // wait for consumer thread to write all events to trace file
12   MM->report();
13 }
14
15 void __vtrace_tef_start(uint64_t ID, char *RegionName) {
16   MM->enqueue_start(ID, RegionName, GetPid(), GetTid());
17 }
18
19 void __vtrace_tef_end(uint64_t ID, char *RegionName) {
20   MM->enqueue_end(ID, RegionName, GetPid(), GetTid());
21 }
```

Listing 3.1: Main functionality of the instrumented measurement code for static tracing. `MeasureManager` internally manages a queue and uses a consumer running in a separate thread to write the tracing data to a file. The current timestamp is measured when enqueuing.

3.3 TRACING WITH DYNAMIC INSTRUMENTATION

Next, we implement tracing but with dynamic instrumentation using USDT probes in conjunction with an attached eBPF program. The high-level idea for the measurement code is to produce the same trace file as the static counterpart.

We insert the required USDT probes by adding a declaration macro through VaRA's instrumentation interface (see Listing 3.2). We provide two arguments to the USDT probes, the feature region's unique identifier and region name. The current timestamp is measured on the eBPF side. To write the eBPF code, we use BPFTRACE[3], a high-level eBPF frontend language designed to allow developers and system administrators to quickly write arbitrary eBPF code that should be executed in response to passing a USDT probe during execution. As a result, tracing tools can be created as needed, for example, the simple script shown in Listing 2.3 to count the number of occurred events.

```
1  #include <sys/sdt.h>
2
3  // inserted at the start of the program's main() function
4  void __vtrace_usdt_init() {
5    DTRACE_PROBE(vtrace, feature_init);
6  }
7
8  // inserted at exit points of main() function
9  SANITIZER_INTERFACE_ATTRIBUTE void __vtrace_usdt_finalize() {
10   DTRACE_PROBE(vtrace, feature_finalize);
11 }
12
13 // inserted at entry points of feature regions
14 void __vtrace_usdt_start(uint64_t ID, char *RegionName) {
15   DTRACE_PROBE2(vtrace, feature_begin, ID, RegionName);
16 }
17
18 // inserted at exit points of feature regions
19 void __vtrace_usdt_end(uint64_t ID, char *RegionName) {
20   DTRACE_PROBE2(vtrace, feature_end, ID, RegionName);
21 }
```

Listing 3.2: Adding USDT probes to feature regions in VaRA. The `sys/sdt.h` header contains macros to declare USDT probes. There are variants to pass multiple arguments to the attached tracer. The first and second argument of the macro form the identifier of the probe. The first is the provider name and the second the probe name. Both can be chosen freely. Notice however, that they aren't strings and as such can't be chosen dynamically. The available characters that can be used are restricted in the same way as identifiers of variables in code.

3 https://github.com/iovisor/bpftrace

3.3.1 *Minimizing Overhead and Selective Activation of Probes*

The instrumentation just described has two issues, suboptimal overhead, especially when the probes are inactive, and the user can't selectively choose which feature regions to measure. This is helpful in a situation where certain feature regions are executed often and hence their measurement incurs high overhead. The user can then selectively turn off measurement of these regions.

The reason for suboptimal overhead is the implementation of VaRA's instrumentation interface. VaRA inserts function calls at the location of entry and exit points of feature regions. This means that, when inserting the USDT probe as shown in Listing 3.2, we wrap it inside an additional function, which means the potential for more overhead compared to inserting the probe directly at the event location. This higher overhead might be shadowed by the time required to perform the interrupt to execute the eBPF program when the probe is active. In the case where the probe is inactive though, the function call pushes arguments to the stack, jumps to the function and jumps back for the return. We suspect that this takes more time than executing a single `nop` instruction.

The other issue is that a user of the instrumented USDT probes can't currently choose to only activate a subset. The reason is that we can't change the probe's identifier when inserting the instrumentation during compile time using the macro to declare the USDT probe. The two arguments for provider and name aren't strings and by our observations behave like identifiers of variables in C code.

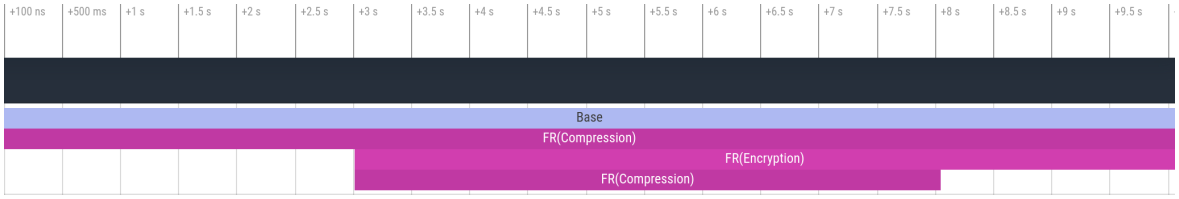
To fix both issues, we investigated how the macro works internally and especially how it adds the necessary entry to the notes section of the compiled ELF binary to provide information about the inserted `nop` instruction. Both the `nop` instruction and the notes section entry are added using a series of assembly instructions. We can replicate these on VaRA's side because the framework is based on LLVM, which does support inserting assembly code directly. This assembly code is defined using a string containing the instructions, which we can modify to use a custom probe identifier.

We realize the just described optimizations separately from the simpler approach of adding the USDT probes using the pre-defined macro to evaluate the effectiveness of our optimization later. We denote the optimized variant *raw USDT instrumentation*.

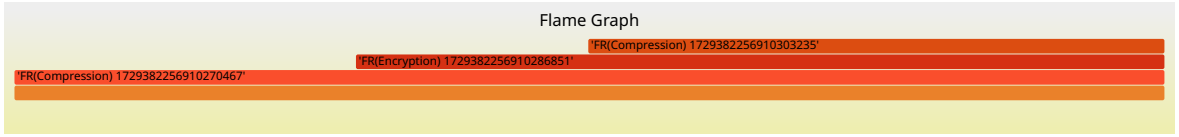
3.4 SAMPLING

The second measurement approach we implement is sampling. The goal is to produce a sampling profile for feature regions, which can be visualized as a flamegraph. Flamegraphs are for example used for visualizing sampled call stacks (see Figure 2.3). They highlight where the most amount of time is spent using the bar length and colors and are therefore a good starting point to gather a high-level understanding when conducting performance analysis.

The resulting flamegraph should match the trends of feature region durations seen in a trace for the same software system. We also want to preserve the nesting of feature regions. For Figure 3.1 we used a constructed software system called *SimpleFeatureInteraction* with three nesting feature regions to do a proof of concept and show a resulting flame graph we aim for.



(a) Trace in PERFETTO



(b) Flamegraph of sampling profile

Figure 3.1: This figure shows the visualized trace file and sampling flamegraph for a constructed software system, which nests three feature regions. It represents the result we aim for when sampling where the resulting flamegraph matches the duration trends and nesting of the feature regions we see in the trace.

To construct a flamegraph visualizing nested feature regions, we need a call stack pendant. Our high-level idea is to sample the call stack and then translate this to a feature region stack. This idea has the that advantage we don't need to spend any time to construct a feature region stack during execution, hence we minimize measurement overhead. This kind of overhead would also be susceptible to the frequency of events and therefore void the big advantage sampling has over tracing.

To obtain the call stack samples, we use PERF, a profiling tool for Linux, which already offers this functionality and is used widely. To give an idea of the content of a sampled call stack, Listing 3.3 shows a single PERF sample collected from *SimpleFeatureInteraction*. The first address in the array we see there is the currently executing instruction and also top of the call stack. All other values are return address [7] and form the rest of the call stack. Return addresses indicate the next instruction to execute in the calling function when returning from the callee.

We now describe how we can translate a sample of the call stack to a feature region stack sample. To make explanations easier in the beginning, suppose we only have the address of the current instruction and only want to translate that to a single feature region, hence ignoring nesting. Furthermore, suppose every feature region only has a single entry and exit location and that we know the addresses of these. To implement a simple algorithm, we assume the following (A_1): The execution does only execute instructions in this address range while in the corresponding feature region and leaves it once the feature region is completed.

With this assumption, we can determine feature regions by checking whether the current instruction's address lies inside the address range for each feature region. The issue is that we can construct examples, for which A_1 doesn't hold at all. One such example is shown in Listing 3.4. The execution enters the feature region, after a few instructions jumps to another location outside the address range and later jumps back in to invoke the exit probe. This pattern causes us to miss parts of, or even the full time, a feature region executes. The result is that we under-approximate the actual execution time.

```
[...]
{
    "timestamp": 112718368898239,
    "pid": 161850,
    "tid": 161850,
    "comm": "SimpleFeatureInteraction",
    "callchain": [
        {
            "ip": "0x227ccf"
        },
        {
            "ip": "0x228084"
        },
        {
            "ip": "0x228882"
        },
        {
            "ip": "0x228a8c"
        },
        {
            "ip": "0x7fc17c54a510",
            "symbol": "__libc_start_call_main",
            "dso": "libc.so.6"
        }
    ]
},
[...]
```

Listing 3.3: An excerpt of a call stack sample, which has been collected using PERF. The array for `callchain` contains the addresses on the call stack. The current instruction's address is the first one in this array. PERF attempts to translate the addresses to function names if the necessary information is available.

```
[...]
  nop       // entry point of FR(Foo)
  jump do_something
exit_feature_region:
  nop       // exit point of FR(Foo)

do_something:
  [...]     // do work for FR(Foo)
  jump exit_feature_region
  [...]
```

Listing 3.4: This example shows that we can construct cases where the address range spanned by entry and exit points of a feature region has no correlation with the addresses of the executed instructions at all.

We can also construct an example to show the other way around, where we wrongly detect time spent in a feature region. The example is shown in Listing 3.5.

```
1   [...]
2   nop       // entry point of FR(Foo)
3   jump main_code_of_Foo
4 main_code_of_Bar:
5   nop       // entry point of FR(Bar)
6   [...]
7   nop       // entry point of FR(Bar)
8 main_code_of_Foo:
9   [...]     // do work
10  nop       // exit point of FR(Foo)
```

Listing 3.5: A constructed example in which we wrongly translate to feature region *Foo* instead of *Bar* under assumption A_1.

We did not encounter this pattern when conducting experiments on small synthetic software systems though. We hope that compiler optimizations arrange assembly code in such a way, that as few `jump` instructions as possible are used and A_1 therefore only rarely breaks in practice.

Continuing with the description of the algorithm to translate call to feature region stack samples, we need to get the addresses of entry and exit locations. For that, we can use the data in the ELF notes section because it contains the locations of the inserted `nop` instructions. It is important that we use the raw USDT instrumentation though, since the regular USDT instrumentation inserts an intermediate call to the same function for all entry and exit probes.

When inspecting the ELF notes section for the *SimpleFeatureInteraction* example, we notice that we can have multiple entry and exit locations for a single feature region. We need to combine them in a sensible way to retrieve multiple address ranges for each feature region. The algorithm is to form an address range for each entry address and end it at the closest larger exit address. We can discard ranges that overlap and only keep the larger one.

So far, we discussed how to translate only the current instruction address to a single feature region for the current instruction. Now, we include nesting. The high-level idea is to iterate over addresses in the call stack sample from first return address inserted to the current instruction to construct the feature region stack bottom-up. We need to handle two important edge cases. The address we are currently investigating may have no matching feature region. We ignore that one. The second case is that we may see multiple matching address ranges, for example, due to nested feature regions in the same function. The outer region can only be equal or larger than the regions it nests. We therefore add the feature regions ordered ascending in address range size to the feature region stack sample.

It is possible to encounter a situation, where a feature region is already contained in the translated stack because a feature region can in principle span multiple functions. We don't insert such a feature region into the feature region stack again.

When applying the algorithm just described to our example *SimpleFeatureInteraction*, it fails. When investigating the assembly instructions, we see the pattern shown in Listing 3.6.

The reason is not that assumption A_1 does not hold, but an issue in our algorithm to derive address ranges in the presence of multiple entry and exit points.

```
1   [...]
2  // BEGIN address range
3     nop       // entry point of FR(Foo)
4     [...]     // do some basic work
5     conditional_jump do_something
6     [...]
7     nop       // first exit point of FR(Foo)
8  // END address range
9     [...]
10 do_something:
11    [...]     // do heavy work
12    nop       // second exit point of FR(Foo)
13    [...]
```

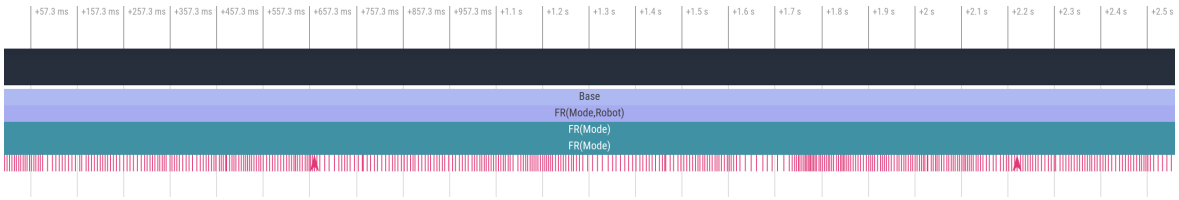
Listing 3.6: This example shows how our algorithm to extract address ranges when a feature region has multiple entry and exit locations fails. It won't produce an address range for capturing the instructions at label `do_something`.

Our high-level idea to solve the issue is to perform static analysis of instructions in the binary to take conditional and non-conditional jumps into account. We iterate over each feature region and its entry locations. We start the static analysis at the location of the concrete entry location and follow the execution path. When encountering a non-conditional `jump`, we insert an exit location for its address and an entry location at its target address. In the case of a conditional jump, we only add its target address to the list of entry locations for the current feature region. We stop when we encounter an exit location for the current feature region and then proceed to perform the same analysis for the next entry location. We repeat the whole algorithm until we no longer insert new entry locations.

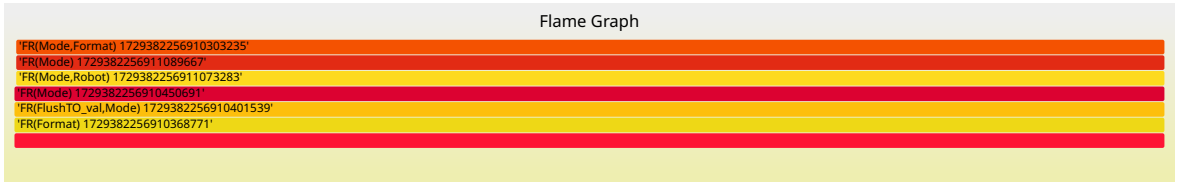
Due to limited time, we did not implement this idea. Instead, we executed the *SimpleFeatureInteraction* twice. In the first run, we collected the addresses of the probes hit during execution. Using this information as a filter for the entry and exit locations, we only produce a single entry and exit location for each feature region in the concrete case. This was sufficient to produce our aimed for result visible in Figure 3.1.

We then also tried the algorithm on the real world case-study *xz* (introduced later in evaluation). The result is shown in Figure 3.2. The sampling profile contains the three most notable bars seen in the trace, which are *FR(Mode,Robot)*, and twice *FR(Mode)*. The issue though is that we completely miss the small periodically appearing, purple region called *FR(Size_val,Threads_val)*. We expect to see a fraction of samples that includes and another that doesn't include this region. The sampling profile also contains wrong feature regions, which provides evidence that assumption A_1 does not hold in practice.

To conclude, the presented algorithm for translating call to feature region stack samples does not work for real world case studies yet. We see evidence that assumption A_1 does not hold in practice. We propose to use static analysis to determine the control flow of the binary's assembly instructions and doing so produce more precise address ranges of the instructions implementing a feature region.



(a) *xz* trace in PERFETTO



(b) Flamegraph of *xz* sampling profile

Figure 3.2: This figure shows the result of the algorithm we proposed to translate call stack to feature region stack samples for the real world case-study *xz*. The flamegraph contains the feature regions visible in the trace but also additional ones that didn't actually execute.

3.5 POTENTIAL LIMITATIONS OF EBPF-BASED TRACING

When using eBPF to trace feature regions, we encountered three potential limitations, which can prohibit the application of this measurement approach in certain situations.

The first issue is that the number of concurrently active probes is limited. In BPFTRACE the default limit is 512 probes and can be extended. However, in their documentation[4] they mention that using more probes "will consume more memory, increase startup times and incur high performance overhead or even freeze or crash the system". We didn't encounter this limit during implementation. For one of the case study *xz* used in the evaluation, however, we require up to 450 concurrently active probes. We didn't notice any slowdown in performance though. Depending on the size of the software system under measurement and the number of feature regions, we imagine that the pre-defined limit can definitely be surpassed in practice when using the raw USDT instrumentation. If this happens and poses issues, we propose to either measure only a subset of feature regions and therefore also only activate a subset of USDT probes or use the regular instead of raw USDT instrumentation because it wraps the USDT probe in a function call meaning that there will be only the four probes shown in Listing 3.2 in total.

Another potential issue is that events are dropped when the maps for passing information from the kernel- to the user-space component of a BPFTRACE program run out of space. The size of an eBPF map is fixed and specified during its initialization[5]. The eBPF verifier also doesn't allow eBPF programs to block execution for an indefinite amount of time [5] to wait for space to become available in the map. During implementation, we actually encountered a situation where simply increasing the map size to compensate for sudden spikes in event frequency did not suffice. We performed an experiment on the case study *gzip*, where a single feature region was invoked with high frequency throughout the whole program execution and overwhelmed the user-space consumer. Even though we might be able to

4 https://github.com/iovisor/bpftrace#bpftrace_max_probes, visited on December 1st 2022
5 https://www.kernel.org/doc/html/latest/bpf/maps.html, visited on December 1st 2022

speed up the consumer by using a lower-level language than BPFTRACE, the tracing overhead was also significant in the concrete case. The desired solution therefore is to reduce the amount of events.

We discussed this case with the VaRA developers and they implemented nested feature regions merging with the high-level idea of discarding inner feature regions when they don't add additional information compared to the outer one. To provide an example, if we encounter a feature region containing the feature *Foo* nested inside a feature region containing *Foo* and *Bar*, then measuring the inner in addition to the outer region does not add new information. This solved our issue but in principle we can always encounter dropped events, which in the case of tracing means we can no longer determine the duration of certain feature region. This is especially problematic when this feature region executes for a significant amount of time. Suppose we are conducting feature performance analysis and interested in which features the CPU spends most of its time on. Missing a significant feature region can lead to wrong conclusions. Sampling also isn't accurate but in comparison to this issue, it only drops details and the overall statistical information in the sampling profile isn't skewed. BPFTRACE will therefore warn when events are dropped. When dealing with high event frequencies, this might be an exclusion criterion for the usage of eBPF though.

The third issue is that running eBPF programs for tracing currently requires root privileges. In addition, BPFTRACE offers the feature to invoke arbitrary binaries on the command-line. This can be a huge security risk in practice, not only in the sense of malicious code but also because it eliminates the isolation guarantees between processes enforced by the operating system. We could, for example, accidentally influence other software systems concurrently running in the production environment and cause them to enter a faulty state. The environment we plan to measure in may therefore not offer the ability to run privileged software.

In principle, we don't require access to internal kernel information to execute a function in an external program whenever we hit a USDT probe. We can wrap sensitive functionality in a system call to limit the ability of user-space applications and preserve the operating system's isolation guarantees. eBPF is designed to offer programmability and do so with high performance though [6]. Adding invocations of additional system calls in measurement code means introducing more overhead.

We found no other tool to attach arbitrary measurement code to USDT probes. The issue of requiring root privileges in our view does not have fundamental reasons and a solution might be developed in the future.

4 EVALUATION

In this chapter, we describe the experiments we conducted to compare the overhead and accuracy properties of tracing and sampling in practice. We answer the following research questions to evaluate the discussed theoretical limitations:

- **RQ1**: Can state-of-the-art profiling tools provide feature-specific on-demand measurements with insignificant overhead when inactive?
- **RQ2**: How much do state-of-the-art tools compromise measurement overhead to offer flexibility?
- **RQ3**: By how much can feature-specific measurement overhead be reduced when switching from profiling traces to sampling?
- (**RQ4**: How much measurement error is introduced by sampling-based feature performance analysis compared to tracing?)*

*We did not evaluate the accuracy of our call stack to feature region stack translation to answer RQ4 due to limited time. Still, we highlight the research question here because it is a deciding factor whether sampling feature regions is feasible in practice.

4.1 CASE STUDIES

To answer our research questions, we conduct experiments on real world and synthetic software systems. We now briefly introduce them.

For the real world software systems we decided on *xz*, *gzip*, and *bzip2*. All of them are file compression tools implementing a different compression format each. We chose to limit ourselves to file compression projects because their execution time on a fixed input file is strongly influenced by the compression level we choose using the configuration options. Consequently, we expect to see a noteworthy amount of time spent on the features responsible for the compression level when conducting feature performance analysis. Compression tools are also used in the evaluation of feature performance analysis in current work [17, 20] Furthermore, the code size and amount of configuration options of these compression tools is limited such that we were able to investigate issues we encountered during implementation in a reasonable amount of time. One example for these issues is VaRA's feature region analysis, which is still under development and can miss feature regions.

Originally, we also planned to include the compression tool *brotli* as a case study. VaRA's data-flow analysis to detect feature regions is expensive and in the case of this software system, we ran out of memory before obtaining a result. We didn't investigate further and decided to drop this software project.

By only studying a few real world software projects, we may not observe the fundamental limitations of tracing and sampling like unavoidable high overhead due to high event frequencies and inaccurate measurements. We therefore complement the real world software projects with a synthetic one, which allows us to construct a certain amount of feature regions with deliberately chosen execution properties like their execution frequencies. Furthermore, we can form feature regions which purposefully span certain statements and which VaRA can reliably detect.

The synthetic case study is called *SimpleBusyLoop* and designed to stress-test the overhead property of the measurement approaches. It is used for answering RQ1 to 3. We use a single small feature region, which we place in a loop, meaning it is constantly entered and left. Inside the feature region we perform busy sleeping by continuously incrementing an integer until it reaches a certain value. This value is chosen such that the feature region executes only for a few microseconds. The idea of the busy sleeping is to simulate a heavy computation which will take longer to complete when inserting measurement code. The details are shown in Listing 4.1. We expect this case study to stress differences in instrumentation overhead since even if small, they should become noticeable due to many events in a short amount of time.

```
1  long __attribute__((feature_variable("NumIterations"))) NumIterations;
2  long CountTo;
3
4  int main(int argc, char *argv[]) {
5    if (!parseParams(argc, argv)) {
6      return 1;
7    }
8
9    for (long i = 0; i < NumIterations; ++i) {
10     // conceptual entry point of feature region
11     for (long Counter = 0; Counter < CountTo; ++Counter) {
12       // prevent compiler optimization
13       asm volatile("" : "+g"(i), "+g"(Counter) : :);
14     }
15     // conceptual exit point of feature region
16   }
17 }
```

Listing 4.1: Main logic of *SimpleBusyLoop*. `NumIterations` is declared as a feature variable through an annotation processed by VaRA's feature region analysis. The inner loop continously increments `Counter` to simulate a heavy computation.

4.2 OPERATIONALIZATION

To answer the presented research questions, we designed multiple experiments we ran on the presented case studies. In this section, we describe these and the efforts we have taken to eliminate external influences, for example, from other processes running on the system.

In all experiments, we measure the execution time of the software system with the goal of comparing the overhead of the different feature performance analysis approaches. For this purpose, we wrap the execution of the case studies in a call to GNU TIME. The measured execution time has limited precision though as GNU TIME reports seconds with only two decimal places. Even though this means we can't measure small differences, we are interested in general trends and perceivable differences. For this intention, the precision suffices as we don't see a difference of only 10 ms to be meaningful.

To make the obtained results comparable between experiments, we use the same workload for the same case study throughout. The reason is that the applied workload or input of a software system influences the execution time. Compressing a larger file takes more time than compressing a smaller one. We choose the workload for each case study such that it produces an execution time of at least 30 seconds to reduce the impact of external influences or variations in startup time. For the same reasons, we repeat each experiment 10 times and compute mean and standard deviation for the measurements.

The experiments are performed on a machine with a six-core AMD Ryzen 5 5600X processor and 16 GB of RAM. We disable turbo boost and lock all processor cores at a frequency of 2.8GHz. The machine is running a clean Ubuntu server install. We ensure that no other processes are actively executing to minimize a possible influence on the results.

In the following, we explain the operationalization for each experiment in detail.

4.2.1 *Dry Experiments*

We conduct three dry experiments, which are designed to answer **RQ1**. The case studies are executed *dry*, meaning we are not performing any measurements. We first execute the case studies without any instrumentation. The measured execution time is the baseline to later determine the amount of overhead. Comparing the execution time in the other experiments to this baseline gives the amount of induced overhead. We also conduct two further dry experiments to determine the amount of overhead the inactive USDT probes or inserted `nop` instructions induce. One of them uses the regular and the other the optimized raw USDT instrumentation.

4.2.2 *Tracing Experiments*

For answering RQ2, we perform three experiments to compare the amount of overhead that tracing with each of the presented implementations induces. In the first experiment, we trace the case studies using the static instrumentation. We then run two further experiments in which we instrument the case study with the two USDT instrumentation variants and attach a BPFTRACE script to produce the same trace file as the static instrumentation.

4.2.3 *Sampling Experiments*

In RQ3, we are interested in how much the overhead for conducting feature performance analysis can be reduced by switching to sampling. We conduct two experiments for which we compile the case studies with the raw USDT instrumentation. This is necessary to determine

the addresses of entry and exit points of feature regions for producing a flamegraph. We sample the call stacks of the case studies using PERF and the two sampling frequencies, 97 Hz and 997 Hz. Measuring the execution time of the case studies when sampled with the two frequencies gives us an idea of how much additional overhead we have to pay for the higher accuracy.

4.3 RESULTS

In this section, we present the results we obtained during our experiments. We summarize the observations in the context of each research question.

RQ1: Can state-of-the-art profiling tools provide on-demand measurements with insignificant overhead when inactive?

Table 4.1 presents the results of the experiments conducted for this research question. Our synthetic case study *SimpleBusyLoop* shows a 3 % increase in execution time when adding the regular USDT instrumentation. *bzip2* shows a more noticeable increase of 5-6 %. *gzip* and *xz* both show a smaller increase.

Comparing the execution times of regular and raw USDT instrumentation, we don't see a clear picture and observe cases where the optimized version is actually slower. The difference is less than 1 % though so for executing dry, our optimization doesn't have a meaningful effect.

Case Study	Dry	Dry USDT	Δ USDT	Dry Raw USDT	Δ Raw USDT
SimpleBusyLoop	59.27 (0.02)	61.12 (0.00)	+3.12 %	61.10 (0.00)	+3.8 %
bzip2	35.23 (0.03)	37.08 (0.11)	+5.25 %	37.16 (0.07)	+5.48 %
gzip	43.50 (0.04)	43.75 (0.01)	+0.58 %	43.55 (0.02)	+0.11 %
xz	48.30 (0.35)	48.50 (0.18)	+0.41 %	48.83 (0.25)	+1.09 %

Table 4.1: A comparison of the dry (no active measurements) execution times. The shown values are in seconds and averages from 10 runs including their standard deviation inside the parentheses. The 0 variance is caused by GNU TIME's limited measurement precision.

Based on our observations, we can answer RQ1: Using state-of-the-art profiling tools to provide feature-specific on-demand measurements does come at the cost of about a 5 % overhead when inactive.

RQ2: How much do state-of-the art tools compromise measurement overhead to offer flexibility?

Table 4.2 shows the concrete results of all experiments we conducted to answer RQ2. Static tracing serves as our comparison baseline.

We see 42 % more overhead in the case of *SimpleBusyLoop*. For the real world software projects the difference in overhead is less than 1 %. These results confirm that our design intention for *SimpleBusyLoop* to stress overhead is indeed working.

When comparing raw USDT tracing numbers to regular USDT tracing to evaluate whether our optimization reduces overhead, we see a similar picture to the results in RQ1. The differences are at most around 1 % and close to the measurement standard deviation. We therefore conclude that raw USDT tracing causes the same amount of overhead as regular USDT tracing. Our optimization is therefore ineffective.

Case Study	Static Tracing	USDT Tracing	Δ USDT	Raw USDT Tracing
SimpleBusyLoop	66.52 (0.20)	94.15 (0.33)	+41.53 %	94.65 (0.24)
bzip2	37.31 (0.09)	37.42 (0.10)	+0.29 %	37.16 (0.24)
gzip	44.19 (0.04)	44.52 (0.01)	+0.75 %	43.58 (0.04)
xz	48.85 (0.35)	48.70 (0.19)	-0.3 %	48.78 (0.23)

Table 4.2: A comparison of the execution time for the tracing experiments and running dry. The shown values are seconds and averages from 10 runs including their standard deviation inside the parentheses. Static tracing is the comparison baseline.

Based on our observations, we can answer RQ2: We are able to construct examples where state-of-the-art tools can significantly compromise measurement overhead to offer flexibility while tracing. In practice, this depends on the concrete software system. We saw an insignificant difference below 1 % for the real world software systems we evaluated.

RQ3: By how much can measurement overhead be reduced when switching from profiling traces to sampling?

The results are denoted in Table 4.3. The execution times from the static tracing experiment denote our comparison baseline as we found them to be generally lower than tracing with eBPF. The basis to compute the relative amount of overhead are the execution times from the dry experiment.

Starting with our synthetic project *SimpleBusyLoop*, we see a decrease in measurement overhead to just 3 % compared to 12 % when tracing. For the real world case studies, sampling also produces less overhead throughout, the difference is generally smaller though and largest for *gzip* with between 1 and 2 %. Increasing the resolution of sampling by using a higher sampling frequency doesn't see any change in overhead for *SimpleBusyLoop* and *xz*, while *bzip2* and *gzip* are slightly affected. The difference is less than 1 % of the dry execution time though. We conclude that increasing the sampling frequency from 97 to 997 Hz does not see a perceivable difference in overhead.

Based on our observations, we can answer RQ3: We are able to reduce the overhead by sampling instead of tracing in all cases. The concrete amount differs though and depends on whether we are dealing with high tracing overhead in the first place. The largest difference we see was a reduction from 12 to just 3 % measurement overhead.

Case Study	Dry	Static Tracing	Sampling 97 Hz	Sampling 997 Hz
SimpleBusyLoop	59.27 (0.02)	66.52 (0.20)	61.13 (0.00)	61.14 (0.00)
bzip2	35.23 (0.03)	37.31 (0.09)	37.12 (0.02)	37.27 (0.02)
gzip	43.50 (0.04)	44.19 (0.04)	43.58 (0.01)	43.64 (0.02)
xz	48.30 (0.35)	48.85 (0.35)	48.80 (0.31)	48.80 (0.20)

Table 4.3: Execution time during sampling compared to the experiments dry and static tracing. The shown values are in seconds and averages from 10 runs including their standard deviation inside the parentheses.

4.4 DISCUSSION

In this section, we discuss the implications of our results for applying our presented measurement approaches in practice. We also conduct two subsequent experiments to explore interesting findings further.

By answering **RQ1**, we saw that inactive USDT probes, even in our synthetic case study designed to stress overhead, only increase the execution time up to 5 %. It is therefore sensible to compile them in by default to allow on-demand feature performance analysis and when not worrying about every percent of performance.

In **RQ2**, we did not observe a noticeable difference when tracing with static instrumentation vs tracing with USDT probes for the real world case studies. The synthetic software system hints though that there can indeed be a significant difference. This difference can be explained by the additional time we require to perform an interrupt and determine the eBPF program to execute in response.

To investigate the negligible difference observed for the real world case studies further, we wrote a small BPFTRACE script to count the number of events for each case study. The result is shown in Table 4.4. The highest number of events for the real world case studies is only roughly 42000, which is low when taking into account that execution time is at least 30 seconds. This number is also by orders of magnitude lower than *SimpleBusyLoop*. The code size and amount of features for our real world case studies is small. If we were tracing feature regions in a larger software system, which offers more functionality and consequently more features and configuration options, we probably see higher overhead.

Case Study	Number of Events
SimpleBusyLoop	20000002
bzip2	115
gzip	13496
xz	42528

Table 4.4: Number of events for all case studies during tracing.

In general, we were surprised to see a roughly 40 % larger execution time when tracing our constructed case study with BPFTRACE instead of static instrumentation. We wondered

how much overhead would decrease when using a lower level eBPF frontend like BCC[1]. BCC allows us to write the measurement eBPF code in C. We can remove additional operations BPFTRACE performs, for example, checking whether events are dropped and communicating that to the user. We performed another tracing experiment using regular USDT probes with the BCC script attached.

The results are shown in Table 4.5 and show a significant reduction in overhead to only a 6.3 % larger execution time than performing static tracing. As expected, the real world case studies don't show a meaningful difference because for them BPFTRACE wasn't slower than static tracing to begin with.

Case Study	Static Tracing	USDT Tracing	USDT Tracing with BCC
SimpleBusyLoop	66.52 (0.20)	94.15 (0.33)	70.73 (0.11)
bzip2	37.31 (0.09)	37.42 (0.10)	37.41 (0.05)
gzip	44.19 (0.04)	44.52 (0.01)	44.51 (0.02)
xz	48.85 (0.35)	48.70 (0.19)	48.64 (0.26)

Table 4.5: Tracing with regular USDT instrumentation but using a lower-level eBPF frontend called BCC instead of BPFTRACE can reduce measurement overhead.

We can also draw another conclusion from RQ2. If we don't require on-demand measurements but need to use tracing to not miss events, static instrumentation offers lower overhead than using dynamic instrumentation and should therefore be the preferred choice when measuring in an offline environment, for example, during development.

The experiments conducted for **RQ3** show that sampling can offer significantly smaller overhead than tracing when only interested in a statistical summary of feature performance. It is therefore a possible approach to reduce measurement overhead during feature performance analysis, for example, when deriving performance influence models as done by Velez et al. [17]. Additionally, when the goal is to influence the system under measurement as little as possible, for example, while investigating performance issues of applications running on production servers, sampling is the obvious choice. As presented in this thesis, it is limited in its applicability though because using our implementation to translate call stacks can currently miss or even translate wrong feature regions. It definitely requires future work to become more reliable.

4.5 THREATS TO VALIDITY

In this section, we discuss internal and external threats to the results of our experiments.

We begin with internal threats to validity. We did not have any variations in the workloads used for each case study. The workload chosen influence the time required to compute the output to given inputs. In addition, they can also change the concrete code paths the software system takes during execution. Furthermore, we only evaluated the software domain of compression tools due to limited time. Other software systems may be much more susceptible to overhead during tracing due to a higher event rate. The case studies

1 https://github.com/iovisor/bcc

we chose only offer a few features and are also small in terms of code size. We expect the number of events on more complex software systems with hundreds of features to be significantly larger. This causes more overhead and can create a significant difference between eBPF-based and static tracing contrary to our observations in the context of RQ2.

Another threat to internal validity is the quality of VaRA's feature analysis. Since it determines the feature regions we instrument, it decides whether we collect meaningful information for feature performance analysis using our measurement approaches. The quality of the analysis also controls the number of events during execution, which directly influences the overhead of the tracing implementations. VaRA's analysis is still under active development. As a consequence, feature regions may be missing or may be determined wrongly. While working on this thesis, we were indeed able to identify such missing regions. Additionally, the collected trace files for all case studies have changed significantly due to updates to VaRA since we tested our tracing implementation for the first time. Despite the unclear level of reliability of VaRA's analysis, we used a synthetic case study to stress the fundamental difference in overhead between the measurement approaches. Doing so we expect our results to generalize to situations with more events, for example, larger software systems.

Our evaluation also has threats regarding external validity. We ran every experiment in a controlled environment with only few other processes executing concurrently. For a server in a production environment, this is typically not the case. Multiple processes are competing for resources like CPU and storage bandwidth. Since our measurement code uses these resources as well, the observed overhead may be larger. Additionally, other processes may be slowed down due to measurements as well.

Furthermore, we did not evaluate the accuracy of the necessary call stack to feature region stack implementation on real world case-studies due to limited time. The accuracy is an important property and decides whether sampling feature regions is feasible in practice. We showed ways to improve our preliminary experiments though and belief that accuracy can improve to a reliable state.

5 RELATED WORK

In this chapter, we put our work into the context of related work. We provide examples of what motivated our work and highlight the concrete contributions we make.

One of the main motivations for our work was a user study conducted by Velez et al. [19] to identify information needs of developers while debugging the performance of a configurable software system. They found that even though developers compare the execution times of a problematic and non-problematic configurations, and analyze call stacks, they still "struggle for a substantial amount of time looking for relevant information to identify influencing options, locate option hotspots, and trace the cause-effect chain". To solve the need for relevant information, Velez et al. [19] provide local performance influence models at the method level to developers. Performance influence models capture how the execution time is influenced when changing the values for certain configuration options. The authors found evidence that providing this information to developers does indeed help during performance debugging.

To determine performance influence models, two approaches do currently exist, black- and white-box analysis. An example for a black-box approach is the work from Siegmund et al. [15]. Their goal is to create a performance influence model for the whole configurable software system. They measure total execution time using many configurations and learn a machine learning model using "the influence of individual configuration options and their interactions from the differences among the measurements". This machine learning model predicts performance of unseen configurations and hence not all possible configurations have to be measured. Measuring all possible configurations using brute-force is not feasible in practice due to the size of the configuration space. Each configuration can potentially interact with each other. The significant drawback of black-box approaches is that depending on the concrete software system it can take a lot of executions with different configurations and therefore long time to build a reliable model.

White-box approaches promise to solve this limitation. Their idea is to analyze source code and doing so reduce the number of required executions, while increasing the precision of the obtained performance influence models.

Weber, Apel, and Siegmund [20] trace methods to learn a performance influence model. They generally use only a few measurements and analyze the code in methods to determine those potentially showing a lot of execution time variance when using a different configuration. The authors then improve the accuracy of the model predictions by extending the amount of measurements to selectively measure additional configurations for methods with potentially significant variance.

Velez et al. [18] also want to reduce the amount of necessary measurements by determining influence of options on fixed regions in code. They chose methods as their region granularity and decompose the performance influence model for a whole system into multiple smaller ones for each region and then combine them later. These smaller performance influence models can be built in parallel during every execution of the configurable software system.

Using white-box analysis, the authors only execute configurations that actually influence the execution time in each region. Interestingly, this work uses call stack sampling to measure the time spent per method with the reasoning to induce a small amount of overhead.

Velez et al. [17] use a similar idea to the work in our thesis. First, they detect regions in code that are influenced by configuration options, which have the same idea as feature regions used throughout our work. Compared the approach from Velez et al. [18], these regions are not fixed to certain structures in code like methods. They then trace the found regions to build the performance influence model. This enables reliable pinpointing of which regions are responsible for observed performance afterwards.

Our work aims to contribute to the current state in the following ways. All white-box approaches we presented here are concerned with high measurement overhead, which could perturb the obtained results. We provide ways of minimizing this, for example, by using sampling. Additionally, we specifically measure the time spent in feature regions with the goal of pinpointing how and where configuration options influence the observed performance.

All discussed approaches are built for offline analysis of configurable software system. We propose a way to enable on-demand feature performance analysis and explain the performance observed in the concrete configuration of the software system deployed in production. There is no need to build a performance influence model beforehand. Furthermore, our approach is developed for high-performance software systems written in C or C++. All before-mentioned work conducts their experiments on Java systems.

6 CONCLUDING REMARKS

In this chapter, we summarize what we learned and which aspects we can improve in future work.

6.1 CONCLUSION

During this thesis, we implemented an approach for enabling on-demand feature performance analysis. We increased the execution time of the compiled binaries by a small amount of 5 % to add the necessary instrumentation to enable these on-demand measurements. We also showed that the flexibility of conducting feature performance analysis on-demand can come at the cost of significantly more overhead compared to static tracing. The reason is the necessary interrupts to invoke the measurement code, which takes more time than inserting measurement code directly into the binary, but enables us to disable them in practice to save operating costs. When on-demand measurements are not required, for example, because feature performance analysis is conducted in an offline environment, we therefore recommend static instrumentation to reduce necessary measurement time when dealing with a complex configurable software system.

During our evaluation, we were able to confirm that using sampling instead of tracing can significantly reduce the overhead during measurements when interested in a statistical summary instead of exact event information. The challenge in the context of feature performance analysis is that we need to enable the measurement of feature regions, which don't correspond to functions found in call stacks. We proposed an algorithm to translate call stacks to feature region stack. It requires future work though to become reliable. We presented an idea to do so using a static analysis of the compiled instructions in the binary.

Finally, we highlight that our work is not limited to measuring feature regions and can be applied to measure other regions of interest in the source code as well.

6.2 FUTURE WORK

While working on this thesis, we either discovered or left opportunities for future work, which we present in this section.

Regarding the tracing implementations we introduced, our results don't show meaningful differences between static instrumentation and dynamic instrumentation using eBPF. We were able to construct a synthetic example that produces significant differences though. In practice, software systems can be more complex offering more features and consisting of more lines of code. We therefore wanted to the experiments we presented on larger software systems but did not have the opportunity due to a limited amount of time.

Furthermore, *VaRA*'s feature region analysis is still under active development to improve the reliability and precision of the found regions. It can either miss feature regions completely or their detection is wrong. An example for this, we encountered during our

evaluation, is *gzip*. The chosen compression level has significant influence on execution time. The corresponding feature does not appear in the trace though. If we were conducting performance debugging and the reason for high observed execution time was a configuration error, which set the compression level to the highest value, the trace offered is of limited use to identify this relationship. This is the motivation for conducting feature performance analysis in the first place though. This example shows that analysis needs more work to become more precise to ensure we provide helpful and trustworthy information to users.

As we saw in our evaluation, sampling is a promising approach to reduce measurement overhead to a small amount compared to tracing-based implementations. The reason is that feature regions are reconstructed after taking the measurements and the measurement code samples state that is maintained automatically without any additional overhead by the CPU. We did not have the time to evaluate the accuracy of our idea to translate call stack samples into feature regions samples and based on these results improve it further. We instead focused on tracing because it can reliably measure feature regions and doesn't introduce a source of measurement error compared to sampling. One idea to improve sampling is to statically analyze the produced binary to reliably identify the address ranges of instructions implementing the feature regions.

BIBLIOGRAPHY

[1] Greg Gagne Abraham Silberschatz Peter Baer Galvin. "Operating System Concepts, 8th Edition." In: Wiley, 2008. Chap. 1.2.1 Computer-System Operation, 1.5 Operating-System Operations.

[2] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. "Feature-Oriented Software Product Lines: Concepts and Implementation." In: Springer, 2013, pp. 6–10, 18, 213–218.

[3] Anindya Banerjee, Sankar Basu, Erik Brunvand, Pinaki Mazumder, Rance Cleaveland, Gurdip Singh, Margaret Martonosi, and Fernanda Pembleton. "Navigating the Seismic Shift of Post-Moore Computer Systems Design." In: *IEEE Micro* 41.6 (2021), pp. 162–167.

[4] David Daly. "Creating a Virtuous Cycle in Performance Testing at MongoDB." In: *Proceedings of the ACM/SPEC International Conference on Performance Engineering*. Association for Computing Machinery, 2021, pp. 33–41.

[5] eBPF.io. *Introduction to eBPF*. visited on November 21st, 2022. URL: https://ebpf.io/what-is-ebpf/#introduction-to-ebpf.

[6] eBPF.io. *Why eBPF?* visited on December 4th, 2022. URL: https://ebpf.io/what-is-ebpf#why-ebpf.

[7] Brendan Gregg. "BPF Performance Tools." In: Addison-Wesley Professional, 2019. Chap. 2.4 Stack Trace Walking.

[8] Brendan Gregg. "BPF Performance Tools." In: Addison-Wesley Professional, 2019. Chap. 18.2 Sample at 49 or 99 Hertz.

[9] Brendan Gregg. "BPF Performance Tools." In: Addison-Wesley Professional, 2019. Chap. 2.10 USDT.

[10] Brendan Gregg. "BPF Performance Tools." In: Addison-Wesley Professional, 2019. Chap. 2.3.1 Why Performance Tools Need BPF, 2.3.3 Writing BPF Programs.

[11] Brendan Gregg. *Analyzing a High Rate of Paging*. visited on November 21st, 2022. URL: https://brendangregg.com/blog/2021-08-30/high-rate-of-paging.html.

[12] Xue Han and Tingting Yu. "An Empirical Study on Performance Bugs for Highly Configurable Software Systems." In: *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. Association for Computing Machinery, 2016.

[13] Linux Man-Pages. *Executable and Linking Format*. visited on November 21st, 2022. URL: https://man7.org/linux/man-pages/man5/elf.5.html.

[14] Jóakim von Kistowski Samuel Kounev Klaus-Dieter Lange. "Systems Benchmarking - For Scientists and Engineers." In: Springer, 2020, pp. 131–133, 138–140.

[15] Norbert Siegmund, Alexander Grebhahn, Sven Apel, and Christian Kästner. "Performance-Influence Models for Highly Configurable Systems." In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. Association for Computing Machinery, 2015, pp. 284–294.

[16] *VaRA Documentation*. visited on December 1st, 2022. URL: https://vara.readthedocs.io/en/vara-dev/research_tool_docs/vara/vara.html.

[17] Miguel Velez, Pooyan Jamshidi, Florian Sattler, Norbert Siegmund, Sven Apel, and Christian Kästner. "ConfigCrusher: towards white-box performance analysis for configurable." In: *Automated Software Engineering* 27.3 (2020), pp. 265–300.

[18] Miguel Velez, Pooyan Jamshidi, Norbert Siegmund, Sven Apel, and Christian Kästner. "White-Box Analysis over Machine Learning: Modeling Performance of Configurable Systems." In: *Proceedings of the 43rd International Conference on Software Engineering*. IEEE Press, 2021, pp. 1072–1084.

[19] Miguel Velez, Pooyan Jamshidi, Norbert Siegmund, Sven Apel, and Christian Kästner. *On Debugging the Performance of Configurable Software Systems: Developer Needs and Tailored Tool Support*. 2022.

[20] Max Weber, Sven Apel, and Norbert Siegmund. "White-Box Performance-Influence Models: A Profiling and Learning Approach." In: *Proceedings of the 43rd International Conference on Software Engineering*. IEEE Press, 2021, pp. 1059–1071.

[21] Shahed Zaman, Bram Adams, and Ahmed E. Hassan. "Security versus Performance Bugs: A Case Study on Firefox." In: *Proceedings of the 8th Working Conference on Mining Software Repositories*. Association for Computing Machinery, 2011, pp. 93–102.