BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

EFFICIENT TECHNIQUES FOR PROGRAM PERFORMANCE ANALYSIS

EFEKTIVNÍ TECHNIKY PRO MĚŘENÍ VÝKONU PROGRAMŮ

TERM PROJECT
SEMESTRÁLNÍ PROJEKT

AUTHOR **Bc. JIŘÍ PAVELA**
AUTOR PRÁCE

SUPERVISOR **doc. Mgr. ADAM ROGALEWICZ, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2020

Brno University of Technology
Faculty of Information Technology

Department of Intelligent Systems (DITS) Academic year 2019/2020

Master's Thesis Specification

Student: **Pavela Jiří, Bc.**
Programme: Information Technology Field of study: Information Technology Security
Title: **Efficient Techniques for Program Performance Analysis**
Category: Software analysis and testing
Assignment:

1. Get acquainted with the principles of measuring runtime and resource consumption of programs. Study the existing frameworks and libraries for effective program profiling (SystemTp, eBPF, etc.).
2. Study methods for modeling performance of programs based on statistical methods (regression analysis, non-parametric kernel estimates, etc.).
3. Get familiar with the Perun project: the performance profile manager. Study the *Tracer* tool that profiles the runtime of programs and that is implemented in the Perun suite.
4. Design and implement optimizations of the performance data collection process in the Perun within the existing (e.g. Tracer) or novel performance data collectors. Focus on optimizations that targets the precision of the measurement, the time of the measurement, or the size of the resulting measured data.
5. Demonstrate the functionality on at least two non-trivial case studies and evaluate the impact of optimizations on the process of performance profiling and resulting data models.

Recommended literature:

- Official site of the Perun project: https://github.com/tfiedor/perun
- Official site of the SystemTap framework: https://sourceware.org/systemtap/
- D. Calavera, and L. Fontana: Linux Observability with BPF: Advanced Programming for Performance Analysis and Networking, O'Reilly Media, ISBN-13: 978-1492050209, 2019.
- T. Hastie, R. Tibshirani, and J. Friedman: The Elements of Statistical Learning, Springer, 2001.

Requirements for the semestral defence:

- Items 1, 2 and 3.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor: **Rogalewicz Adam, doc. Mgr., Ph.D.**
Head of Department: Hanáček Petr, doc. Dr. Ing.
Beginning of work: November 1, 2019
Submission deadline: June 3, 2020
Approval date: May 14, 2020

Abstract

In this work, we propose optimization techniques focused on the data collection process of program performance analysis and profiling within the Perun framework. We enhance Perun (and especially its Tracer module) by extending their architecture and implementing novel optimization techniques that allow Perun to scale well even for large projects and test scenarios. In particular, we focus on improving the data collection precision, scaling down the amount of injected instrumentation, limiting the time overhead of the collection and profiling processes, reducing the volume of raw performance data and the size of the resulting profile. To achieve such optimization, we utilized statistical methods, several static and dynamic analysis approaches (as well as their combination) and exploited the advanced features and capabilities of SystemTap and eBPF frameworks. Based on the evaluation performed on two selected projects and numerous experiment cases, we were able to conclude that we successfully achieved significant levels of optimization for nearly all of the identified metrics and criteria.

Abstrakt

Tato práce představuje optimalizační techniky zaměřené na proces sběru výkonnostních dat v rámci výkonnostní analýzy a profilování programů v nástroji Perun. Rozšíření architektury a implementace těchto nových optimalizačních technik v nástroji Perun (a převážně pak v jeho modulu Tracer) zlepšuje jeho škálovatelnost a umožňuje tak provádět výkonnostní analýzu i nad rozsáhlými projekty. Zaměřujeme se především na zvýšení přesnosti sběru dat, redukci množství instrumentovaných bodů programu, omezení časové režie procesu sběru dat a výkonnostního profilování, snížení objemu sbíraných dat a velikosti výsledného výkonnostního profilu. Optimalizace je dosažena pomocí aplikace statistických metod, množství technik statické a dynamické analýzy (případně jejich kombinací) a využitím pokročilých možností a schopností nástrojů SystemTap a eBPF. Na základě vyhodnocení provedeného na dvou vybraných projektech a množství experimentů můžeme konstatovat, že se nám úspěšně podařilo dosáhnout značné optimalizace u téměř všech sledovaných metrik a kritérií.

Keywords

optimization techniques, performance analysis, dynamic analysis, static analysis, dynamic instrumentation, continuous integration, SystemTap, eBPF

Klíčová slova

optimalizační techniky, výkonnostní analýza, dynamická analýza, statická analýza, dynamická instrumentace, kontinuální integrace, SystemTap, eBPF

Reference

PAVELA, Jiří. *Efficient Techniques for Program Performance Analysis*. Brno, 2020. Term project. Brno University of Technology, Faculty of Information Technology. Supervisor doc. Mgr. Adam Rogalewicz, Ph.D.

Rozšířený abstrakt

Složitost a rozsah počítačových programů stabilně roste každým rokem, přičemž dnešní systémy jsou z hlediska velikosti a množství závislostí na externích nástrojích a knihovnách neporovnatelné se systémy předchozích desetiletí. Přestože tento trend umožňuje vytvářet stále sofistikovanější software a neustále posouvat hranice toho, co považujeme za možné, přináší s sebou i negativní důsledky. Vývojem takto navzájem propojených systémů s vysokou mírou závislostí vzniká hrozba, kdy i jedna kritická chyba — zavlečena do programu například v rámci pravidelných bezpečnostních záplat — může způsobit nejen pád samotného programu, ale i velké řady na něm závislých systémů.

Právě z těchto důvodů dnes hrají čím dál tím důležitější roli techniky *testování software*, nebo mnohem pokročilejší přístupy pro kontinuální sledování kvality vyvíjeného software (známé pod pojmem *CI: Continuous Integration*). Zatímco však automatizovaná a včasná detekce tzv. funkčních chyb (tedy chyb, které mění chování programu oproti jeho specifikaci nebo zamýšlenému účelu) pomocí nástrojů pro kontinuální sledování kvality software, jednotkového, systémového nebo regresního testování je v současnosti průmyslovým standardem, to samé neplatí v případě tzv. výkonnostních chyb. Výkonnostním chybám — tedy těm, které standardně nemění chování programu, ale způsobují značné zpomalení jeho běhu, vysokou odezvu na uživatelské akce nebo rovnou úplně znemožňují interakci s programem — je v praxi běžně přikládán význam až ve chvíli, kdy začnou být postřehnutelné jejich projevy při práci s daným programem. To vše přesto, že přítomnost, byť dočasná, výkonnostních chyb může mít drtivý dopad na důvěru zákazníků ve vyvíjený produkt; stejně jako je tomu v případě aplikací s přemírou funkčních chyb. Důvodem odlišného důrazu na hledání funkčních a výkonnostních chyb je nedostatek nástrojů pro integraci a podporu kontinuálního sledování kvality software v oblasti výkonnostního testování.

Nástroj *Perun* [20, 21], který vznikl a je vyvíjen v rámci výzkumné skupiny VeriFIT, má za cíl poskytnout vývojářům právě takové prostředky, jež jsou potřeba pro odhalení závažných výkonnostních chyb již ve stádiu jejich vzniku, což je možné díky přímému propojení nástroje Perun s verzovacími systémy správy kódu (známé pod pojmem *VCS: Version Control Systems*). Oproti jiným nástrojům zaměřeným na výkonnostní analýzu (jako je například *Valgrind* [46, 73], *OProfile* [50] a jejich komerční varianty) tak Perun navíc staví i na principech kontinuálního sledování kvality kódu aplikovaných v oblasti profilování výkonu programů. Přestože jsme v minulosti již demonstrovali [66], že Perun lze využít pro profilování malých a — do určité míry — i středně velkých programů, některé použité techniky analýzy a sběru výkonnostních dat neškálují dostatečně na to, aby bylo možné je aplikovat i na rozsáhlé programy.

Tato diplomová práce proto představuje několik nových, námi navržených optimalizačních technik z oblasti výkonnostní analýzy, a jejich implementaci v nástroji Perun. Každá z takto navržených technik je zaměřena na některé z definovaných optimalizačních kritérií, jako je například zvýšení přesnosti profilace, redukce množství instrumentovaných bodů programu, snížení velikosti výstupních souborů (obsahujících jak hrubá data získaná právě z instrmentačních bodů, tak jejich transformance do výsledného *profilu*), zrychlení procesu sběru výkonnostních dat a současně tak i zrychlení celého procesu profilování a lokalizace výkonnostních chyb. Nejdříve je však zapotřebí uzpůsobit jádro nástroje Perun navržením nové architekturu podporující optimalizační techniky, a současně rozšířit i aktuálně používaný sběrač výkonnostních dat: *Trace Collector*, označovaný také jako *Tracer*, který byl navržen, implementován a dále rozvíjen v rámci naší předchozí práce [53, 55, 54]). Cílem této práce je tak optimalizovat celý proces výkonnostní analýzy nástroje Perun (a obzvláště pak jeho segment sběru výkonnostních dat) tak, aby dostatečně škáloval i pro rozsáhlé

produkční systémy, a umožnil tak jejich efektivní automatizovanou a kontinuální kontrolu kvality z hlediska výkonnosti.

Ve své původní verzi byl sběrač dat, Tracer, implementován s využitím sady instrumentačních nástrojů *SystemTap*, která umožňuje dynamickou instrumentaci binárních spustitelných souborů. Možnosti instrumentace založené na nástroji SystemTap jsou však do určité míry omezené a pro mnou zkoumané techniky optimalizace nedostačovaly, a proto jsem v rámci práce provedl průzkum nové, velmi aktivně vyvíjené, instrumentační technologie *eBPF*. Na základě prvotních experimentů a získaných poznatků jsem se rozhodl rozšířit architekturu modulu Tracer tak, aby nově podporoval obě instrumentační techniky (tedy jak SystemTap, tak eBPF) a uživateli umožňoval mezi nimi volně přepínat.

Celkem jsem v této práci navrhl sedm optimalizačních metod: *Static Baseline*, *Dynamic Baseline*, *Call Graph Shaping*, *Diff Tracing*, *Dynamic Sampling*, *Timed Sampling* a *Dynamic Probing*, přičemž tyto techniky jsou založeny především na technikách statické a dynamické analýzy, nebo jejich vhodné kombinaci. Celý proces optimalizace pak spočívá v aplikaci vybrané podmnožiny technik a na základě jejich výstupů je učiněno rozhodnutí, které funkce budou nebo nebudou instrumentovány, případně s jakými instrumentačními parametry (například vzorkováním výstupních dat). V oblasti statické analýzy jsem úspěšně využil nástroj *Loopus* [63] zaměřený na analýzu mezí (*bounds analysis*) a amortizované složitosti (*amortized complexity analysis*) některých cyklů a funkcí ve zkoumaném programu (*Static Baseline*). Dále jsem také představil několik nových přístupů založených na průchodu grafu volání (*call graph*) a grafu toku řízení (*control flow graph*), z jejichž struktury je možné aproximovat některé vlastnosti vybraných uzlů (*Call Graph Shaping*), případně identifikovat změny napříč různými verzemi projektu (*Diff Tracing*). U dynamické analýzy se naopak spoléháme na informace a indikátory získané (a) z předchozích běhů nástroje Tracer pro danou konfiguraci profilování nebo (b) přímo za běhu profilování. Díky tomu jsem schopnen precizně identifikovat instrumentované funkce, které jsou volány příliš často a způsobují tak nezanedbatelnou časovou (*Dynamic Baseline*, *Dynamic Probing*) nebo paměťovou (*Dynamic Sampling*, *Timed Sampling*) režii spjatou s profilováním. Za účelem dosažení uživatelsky přívětivého rozhraní pro obsluhu optimalizací jsem se dále rozhodl vytvořit sadu předkonfigurovaných kombinací optimalizačních technik a jejich parametrů (tzv. *pipelines*), které mají umožnit jednoduchou volbu mezi různými úrovněmi preciznosti optimalizací.

Na závěr práce jsem provedl rozsáhlé vyhodnocení těchto navržených a implementovaných optimalizačních technik spolu s jejich předpřipravenými kombinacemi (*pipelines*) na dvou vybraných projektech: implementaci `CCSDS` [1] obrazového kompresního enkodéru a referenční impelemtace kompilátoru a interpretu jazyka Python, `CPython` [11]. Následně jsem výsledky provedených experimentů porovnal s výchozím stavem bez použití jakýchkoliv optimalizačních technik (pomocí sady předem vybraných evaluačních metrik) a na základě těcho výsledků lze konstatovat, že se mi úspěšně podařilo optimalizovat proces výkonnostní analýzy v rámci nástroje Perun a dosáhnout tak vytyčených cílů této práce.

Efficient Techniques for Program Performance Analysis

Declaration

I hereby declare that this Masters's thesis was prepared as an original work by the author under the supervision of doc. Mgr. Adam Rogalewicz, Ph.D. and Ing. Tomáš Fiedor. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

.....
Jiří Pavela
June 3, 2020

Acknowledgements

I would like to express my sincere gratitude to my supervisors Doc. Mgr. Adam Rogalewicz, Ph.D. and Ing. Tomáš Fiedor for their remarkable guidance and immensely helpful consultations, advice, recommendations, reviews and feedback during the course of this Thesis. Furthermore, I would like to thank Ing. David Bařina, Ph.D. for providing us with one of the evaluation projects used in this Thesis. Last but not least, a sincere thanks goes to my girlfriend, family and friends for their never-ending support and encouragement during the inevitable hard times.

The work presented in this thesis was supported by the ECSEL AQUAS: "Aggregated Quality Assurance for Systems" and Czech Science Foundation SNAPPY: "Scalable Techniques for Analysis of Complex Properties of Computer Systems" projects.

Contents

Chapter 1

Introduction

> "*Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%.*"
>
> Structured Programming with `go to` Statements
> — Donald E. Knuth

The complexity and scale of computer programs is steadily rising every year and Today's systems are hardly comparable to those of previous decades in terms of magnitude and the amount of dependencies on external tools or libraries. While such trend allows us to constantly build more sophisticated software and steadily push the limits of what is possible, it also comes at a price. By developing systems and programs so interconnected and dependent on (each) other, a critical failure — e.g. caused by an undiscovered bug, mistakenly introduced by the latest patch — may not only crash the affected program, but also a multitude of dependent systems.

For this reason, *software testing*, as well as more complex techniques such as *continuous integration (CI)*, begin to play an even more crucial role than ever before. However, while automated, early detection of functional bugs through CI, unit, system or regression testing has become an industry standard, performance defects are often not addressed until they fully manifest themselves — by a noticeable slowdown of services or even by rendering the system almost unusable. All the while poor software performance (even just a temporary) is known to have dire effects on customers' trust, as would a program ridden with functional bugs. The reason why performance bugs are widely omitted, lies in the lack of continuous monitoring and integration tools dedicated to performance analysis.

Perun [20, 21], authored and maintained by the VeriFIT group, attempts to fill the gap and aid developers in detecting severe performance degradations as soon as they emerge, i.e. whenever a new project version is created in *Version Control System (VCS)*. Unlike other various profiling frameworks, tools and tool suites (such as *Valgrind* [46, 73], *OProfile* [50] and many commercial ones), Perun also leverages the principles of CI and applies them in the field of performance analysis. Although Perun has been proven capable (as

demonstrated in [66]) of profiling small and — to a certain extent — medium sized projects, some of the analyses are still insufficient for large code bases.

In this work, we introduce several novel optimization techniques for performance analysis and their implementation within the Perun framework. The proposed techniques strive to scale down the amount of injected instrumentation, optimize the volume of collected raw performance data, reduce the size of resulting *profiles* or diminish the time overhead incurred by not only the performance data collection phase, but also the whole profiling and degradation detection process. In order to do so, first we focus on extending the Perun core with new optimization architecture, and enhancing the *Tracer* module (introduced in our previous work [53, 55, 54]) which implements the performance data collection. The goal of this Thesis is to optimize the workflow (and especially the data collection segment) of Perun so that even large programs and production systems can be efficiently and continuously analyzed for performance defects.

Structure of the Thesis. Chapter 2 introduces the topic of system observability using the *tracing*, *sampling* and *instrumentation* — techniques especially useful in the field of testing and, notably, performance analysis. In Chapter 3, we describe the difference between parametric and nonparametric statistical models, along with their examples. Specifically, only methods utilized by the Perun framework for performance data modelling are considered. An overview of the Perun framework, its workflow and architecture is briefly presented in the Chapter 4. Moreover, this Chapter also contains a detailed description of *Tracer* — a performance data collection module within the framework which is leveraged to evaluate the proposed optimization techniques. We argue that proper specification of the Thesis' requirements, goals and evaluation strategy is crucial for its success, hence, Chapter 5 identifies the Functional and Non-function Requirements of this work, introduces the selected Optimization Criteria used to design the optimization techniques and defines the Evaluation Metrics leveraged to assess the achieved results. In Chapter 6, we address the current shortcomings of Perun and Tracer architecture, as well as design and implement modules that aim to provide the missing features. Chapter 7 is the core of this Thesis — we introduce and thoroughly describe the proposed optimization techniques and their combinations, called *pipelines*, designed to present the user with easy-to-use, pre-configured optimization settings. During the evaluation, we generated an enormous amount of data and thus, Chapter 8 attempts to present the results in a compact, yet descriptive way. The final Chapter 9 summarizes the results achieved in this work, evaluates whether we satisfied all of the specified requirements and outlines potential improvements that could be addressed in the future work.

Chapter 2

System Observability

We begin with the introduction of static and dynamic analysis, system and application observability as well as the commonly leveraged observability techniques — *instrumentation*, *tracing* and *profiling*. These techniques are particularly useful in the field of performance analysis since they can provide the necessary data about the resource management or the execution time of the *system under test (SUT)*. First, we lay out the basic terminology regarding analysis and observability, followed by a description of state-of-the-art instrumentation approaches and solutions. Finally, we list a selection of publicly available tools and frameworks supporting dynamic instrumentation, tracing and profiling.

2.1 Static and Dynamic Analysis

Program analysis is nowadays widely exploited in many fields of computer science and software engineering, such as source code compilation, program debugging, testing, formal verification and especially in the area of performance tuning. Due to the often fundamentally different approaches in tackling the analysis task, we categorize the analysis tools into different groups. In particular, we will attempt to introduce one of the many possible (however, quite broadly used) classifications of analysis, as presented by [45].

Static analysis performs the analysis on the source code of the program, without the need to actually run it. This definition incorporates many traditional techniques such as *data flow analysis*, *constraint-based analysis*, *type-based analysis*, *abstract interpretation* or searching for *error patterns*; Sometimes even the more advanced techniques commonly leveraged by formal verification — such as *theorem proving* or *model checking* — are viewed to be part of the static analysis group (as described in [34]). Static analysis has found, e.g. in type checking, correctness analysis and optimization done by compilers, intelligent code completion, code highlighting or code transformation hints in development environments and many more. By definition, static analysis *can* be *sound* (a term originated in mathematical logic: a deductive system is sound with respect to a semantics if it only proves valid arguments [78]), unlike the dynamic analysis. Even though static analysis is rarely used in the field of performance analysis (apart from e.g. compiler optimizations), some approaches have been recently proposed: the *Mira framework* [43], for example, builds a performance model of the program based on the target architecture, source and machine code. The resulting model is subsequently analyzed in order to discover possible optimizations, without the need to actually run the program.

Dynamic analysis, on the other hand, focuses on examining the program behaviour during its execution — be it an execution on real processor, synthetic one or entirely simulated by a virtual machine (such is the case with tools like *Valgrind* [46], *Pin* [39] or *DynamoRIO* [5] in the performance analysis field). An essential element of most of the dynamic analyses (although not all of them, e.g. software testing) is *instrumentation*. Dynamic analysis is typically leveraged by tools such as *profilers*, *checkers*, *execution visualisers* or *performance analyzers* [45]. Software testing is also considered to be a part, or at least a closely related field, of the dynamic analysis scene. While static analysis generally covers all the possible execution scenarios, dynamic analysis is restricted only to the actually executed paths — however, the obtained results are guaranteed to be a close estimation of the actual runtime values. Measuring function execution time (in `real` elapsed time), for example, by using dynamic analysis yields more accurate values, compared to static analysis techniques.

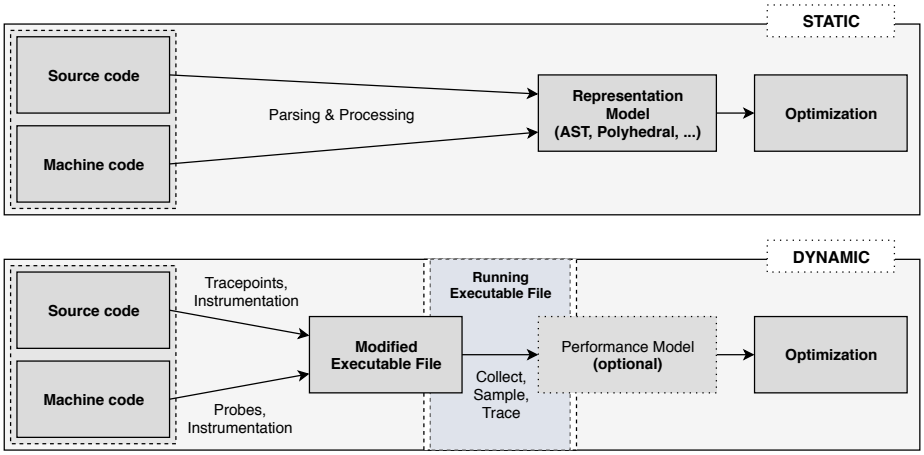

Figure 2.1: An illustration of the different analyses in the field of performance analysis. Both static and dynamic analyses usually leverage source and machine code to either parse and process or inject instrumentation, respectively. The biggest difference is that dynamic performance analysis needs to run the executable file in order to obtain performance data.

Apart from the dynamic and static categories described above, [45] also proposes another possible classification, orthogonal to the previous one:

- *Source analysis* is performed exclusively on the source code level. This type of analysis is generally utilizing common constructs of programming languages, such as functions, statements, expressions and variables. Typically, compiler optimizations and many IDE features are built around the source analysis.
- *Binary analysis* is focused solely on the intermediate (object) or machine code. Historically, the binary analysis had been limited to various proprietary virtual machines interpreting custom byte-codes (assembled from the original machine code) that simulated necessary instructions, registers and memory locations. However, with the origin of built-in HW and SW counters (used by, e.g. *Perf* [17]), *hooking* mechanisms, *probes* and *tracepoints*, binary analysis techniques has considerably shifted.

Figure 2.1 illustrates the different combinations of analysis approaches as utilized by the performance analysis. In this work, we will focus on the *dynamic* and *binary analysis*, however, we might also explore the possibilities of some static and source analysis techniques in the area of performance analysis.

2.2 Observability, Tracing, Sampling and Instrumentation

According to [6], system observability is explained as "*the capacity that we have to ask arbitrary questions and receive complex answers from any given system. A key difference between observability, logs and metrics aggregation is the data that you collect. Given that by practicing observability you need to answer any arbitrary question at any point in time [...]*". Additionally, [28] states the system observability refers to "*understanding a system through observation, and classifies the tools that accomplish this. These tools include tracing tools, sampling tools, and tools based on fixed counters*".

Based on the above definitions — albeit quite different — the observability is:

- a way to understand the given system by asking arbitrary questions whenever needed,
- achieved using tools for tracing, sampling and accessing fixed (SW or HW) counters.

The *tracing*, *sampling (profiling)* and *instrumentation* techniques are essential for dynamic observation and performance analysis of the target systems or applications. In order to better understand their use-cases and limitations (and consequently understand the capabilities of corresponding tools), we provide an explanation of the terms (based on [28, 4, 51, 39, 62]).

Tracing can be generally described as an event-based recording. This, rather broad, definition is in practice usually restricted to a handful of use-cases:

- recording the *syscall* or *signal* events (see Section 2.5.1 for details on *strace* [64]),
- recording the function — be it in kernel (such as the *ftrace* utility [59], see Section 2.5.1) or user space — and library function calls (e.g. the *ltrace* utility [36]),
- measuring events by fixed statistical counters or static points (e.g. the *top* utility [70]),
- recording network events, such as incoming or outgoing packets, e.g. *tcpdump* [68],
- and other, mostly custom and proprietary use-cases.

A typical characteristic of tracing is that it records large quantities of raw data and metadata, so post-processing techniques (e.g. creating snapshots or filtering and compressing the data, etc.) usually need to be utilized. The recovered call or event sequence trace can be traversed and examined in order to detect bugs in execution, unexpected or unwanted behaviour, performance changes, security risks or hazards, diagnose the cause of a failure, or be used in the process of optimization and assessing code or execution metrics (e.g. test coverage). The possible performance issues are usually identified through redundant calls or events, notable change in timestamp variance or average call hierarchy depth.

Sampling tools "*take a subset of measurements to paint a coarse picture of the target*" [28]. Unlike the tracing technique, the sampling only attempts to measure the relevant data or counters in (usually) fixed intervals — either elapsed time, CPU frequency or event count threshold. Although this approach can substantially reduce the amount of performance overhead imposed on the program under evaluation compared to the tracing, the accuracy and precision of obtained data is, more often than not, negatively affected.

A note on profiling:

Some authors (notably [28]) tend to view the *sampling* and *profiling* terms as identical, or at least very close to each other. However, in this thesis, we adopt the notion of *profiling* as an inherent technique (or a collection of techniques) for obtaining and assessing performance information of a program [51].

Instrumentation is a technique that allows for a detailed surveillance of selected application locations as they are executed. This surveillance is achieved by either adding extra code or software probes to the source code, or by utilizing some of the more advanced methods, such as: modifying directly the machine code or creating a new virtual layer between the application and the system, and thus intercepting system calls or kernel instructions, etc. Instrumentation is an essential component of tracing, sampling and profiling as it allows to specify what additional data will be produced by the program during its execution (e.g. timestamps to determine the time consumption of functions or code blocks).

The two following Sections 2.3 and 2.4 focus on two types of instrumentation techniques, as they are both heavily leveraged by the *Tracer* module of the *Perun* (see Section 4) project.

2.3 Static Instrumentation

Instrumentation is considered to be *static* if the *instrumentation code*[1] is injected before the program starts. The instrumentation can be performed either directly on the source code, or more advanced approach of modifying the machine code may be taken — these techniques are, however, usually not straightforward, nor reliable for production environment. Therefore, the dynamic or static source code instrumentation is often preferred. Nevertheless, some notable advancements have recently been made in the field of static binary instrumentation [81] and new tools for *reassembleable disassembling*, such as `Uroboros` [72, 76], are emerging. However, since the static binary instrumentation is not utilized in this work, it will not be further discussed.

One of the notable advantages of the source code instrumentation is the absence of *interface stability issue* and *inlining problem*, both of which manifest themselves during the dynamic instrumentation [28] (see Section 2.4 for more details). On the other hand, maintaining certain forms (primarily *tracepoints* and *user-space statically defined tracepoints*, but usually not the compiler-aided instrumentation, for example) of the source code instrumentation can be time-consuming and hinder good maintainability, if managed poorly.

2.3.1 Tracepoints and User-space Statically Defined Tracepoints

Tracepoints [13, 6, 28] are static markers scattered around the kernel code that handles virtual memory, networking, file system, drivers, etc. Tracepoints evolved from the previously used *kernel markers* [8] which were having issues due to some of their shortcomings, such as insufficient type checking. Each tracepoint serves as a *hook*[2] for function that can be provided at runtime, and if the tracepoint is enabled, the supplied function is called whenever the kernel execution reaches the tracepoint statement. The declaration of tracepoint

[1] The code that is being added to the program

[2] This expression usually refers to a code location, instruction or operation that can be accessed by tracing and profiling tools when needed, and we will use it as such.

also discloses which parameters or internal variables are accessible by the function, which often proves to be invaluable for debugging and monitoring.

Tracepoints are designed to be very *lightweight* and as such, disabled tracepoints incur only tiny performance overhead (in terms of both memory and execution time). Every tracepoint has to be added or modified manually by the kernel developers — which is one of the reasons why tracepoints are considered to be *stable API*, and thus change very rarely.

Listing 2.1: An example of kernel tracepoint declaration that utilizes the `TRACE_EVENT` macro. Detailed description of the individual macro parameters can be found at [60].

```
// include/trace/events/tcp.h
TRACE_EVENT(tcp_retransmit_synack,
  TP_PROTO(const struct sock *sk, const struct request_sock *req),
  TP_ARGS(sk, req),
  TP_STRUCT__entry(...),
  TP_fast_assign(...),
  TP_printk(...)
);
```

In order to define a new kernel tracepoint, one of the many available `C` macros has to be used (originally, only `DECLARE_TRACE` and `TRACE_EVENT` were proposed, however, the current kernel source code contains numerous other macros presumably built on top of the original ones). The Listings[3] 2.1 and 2.2 demonstrate the declaration and usage of one such kernel tracepoint that is used in the `ipv4` networking code.

Listing 2.2: An example of kernel tracepoint usage at `TCP` networking source code location. This tracepoint can be accessed by tracing or profiling tools (such as *ftrace*, *SystemTap* or *BPF*) in order to record what `TCP SYN-ACK` packets were retransmitted.

```
// net/ipv4/tcp_output.c
int tcp_rtx_synack(const struct sock *sk, struct request_sock *req)
{ ...
  if (!res) { ...
    trace_tcp_retransmit_synack(sk, req);
  } ...
}
```

The *User-space Statically Defined Tracepoints (USDT)* [28, 6] are based on the same principle as the kernel tracepoints with one key difference — USDT are meant for the user applications. This grants the developers of custom applications the power to create tracepoints in their own code at semantically sensible locations (e.g. *query___start* and *query___done* tracepoints in *mysql*). The necessary steps to add an user tracepoint are much easier than the kernel one, and roughly consist of linking the `sys/sdt.h` header file which contains the needed macros, such as `DTRACE_PROBE` (the macro was named this way since both USDT and tracepoints were greatly influenced by, and based on, the original mechanism introduced by the *DTrace* [15] framework developed by the *Sun Microsystems*). The Listing 2.3 shows an example of USDT locations found in the *Ruby*[4] project.

[3] Taken from https://github.com/torvalds/linux/blob/master

[4] More information available at: https://www.ruby-lang.org/en/

Listing 2.3: An example of USDT hooks available in the *Ruby*. The tracepoints are placed at code locations for e.g. array allocation, `C` methods calls and returns, garbage collection phases etc. The *SystemTap* (see Section 2.5.4) tool was used to list the available tracepoints.

```
$ stap -l 'process("./ruby").mark("*")'
process("./ruby").mark("array__create")
process("./ruby").mark("cmethod__entry")
process("./ruby").mark("cmethod__return")
process("./ruby").mark("find__require__entry")
process("./ruby").mark("find__require__return")
process("./ruby").mark("gc__mark__begin")
process("./ruby").mark("gc__mark__end")
process("./ruby").mark("gc__sweep__begin")
process("./ruby").mark("gc__sweep__end")
...
```

Tracepoints and USDT are currently supported by most of the relevant tracing and profiling tools or frameworks, such as `LTTng` [38], `perf`, `SystemTap`, or `eBPF`. Tracepoints can be leveraged by performance analysis, for example, to easily measure the duration of specific program operations spanning across multiple functions (e.g. garbage collection).

2.3.2 Compiler-aided Instrumentation

While the tracepoints have to be manually inserted by the developer, *compiler-aided* instrumentation takes a different approach: instrumentation is done by the compiler itself. Generally, the instrumentation is done in the intermediate and code-generation phases of the compilation by injecting new instructions (such as invoking a callback function) into target locations (e.g. function entry points). In particular, we will describe the compiler-aided instrumentation on two compilers: `GCC`[5] and `LLVM`[6].

GCC native instrumentation [22] is considerably limited. Even though GCC is capable of instrumenting many code locations (e.g. the `-fprofile-arcs` option adds instrumentation to code locations such as branches or calls), the flexibility is usually severely restricted, i.e. the user has no control over what instrumentation code is being injected. The only exception is the `-finstrument-functions` option which instruments the entry and exit points of all the user functions with new callback functions. Although this approach is quite cumbersome, it can be exploited to create an automated and viable — however a bit limited and inflexible — performance profiling tool, as demonstrated by our previous work [52]. Figure 2.2 depicts the result of the function instrumentation on an example binary file.

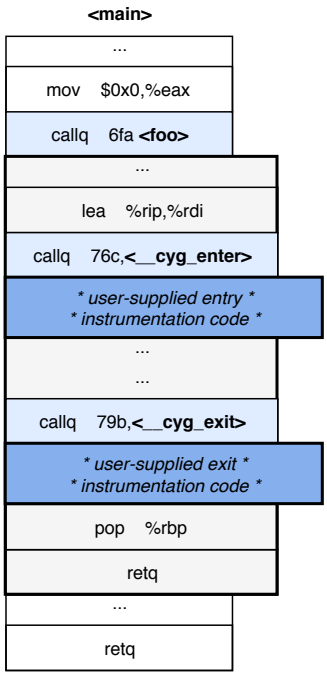

Figure 2.2: An example of the GCC instrumentation and its translation into the binary.

[5]The GNU Compiler Collection: https://gcc.gnu.org/
[6]The LLVM Compiler Infrastructure: https://llvm.org/

LLVM offers state-of-the-art support for instrumentation — mainly thanks to its modular design, plugin support and *Internal Representation (IR)* accessibility — which incentivized the creation of many instrumentation tools and frameworks, both highly purpose-specific and generic ones. Tools such as *XRay* [74], *LLVM Instrumentation Plug-in for Score-P* [71], *Loom* library [77] or the instrumentation tool proposed in [75] attempt to instrument myriad of basic source code blocks or IR elements (function entries and exits, memory reads and writes, variables etc.). Some of the tools (for example the XRay) even go as far as providing a way to dynamically enable and disable the instrumentation at runtime.

Although static instrumentation is powerful and robust, it still imposes some drawbacks. Namely, the source code has to be modified (tracepoints) or the compilation process has to be involved (compiler-aided instrumentation). This is especially limiting if, for some reason, source code of the application is not available, cannot be modified or the re-compilation is not an option — such is the case with many commercial or system applications.

2.4 Dynamic Instrumentation

Since static instrumentation proves to be limiting in certain scenarios, the dynamic instrumentation adopts a different approach which makes it possible to add instrumentation *during* the application runtime — without any preceding modifications done in the source code or by the compiler.

2.4.1 Early and Proprietary Solutions

The idea of dynamic instrumentation was initially introduced by Hollingsworth and others [30], who proposed an instrumentation technique that directly modifies the binary image of the running application. The so-called *base trampolines* and *mini trampolines* (essentially a blocks of instructions) are inserted into the executed machine code to invoke the instrumentation code (*primitive*). Using this approach (with some modifications), first tools for performance analysis and debugging, such as `KernInst` [67], have appeared.

However, since such direct modifications of running applications are perceived (and rightly so) as extremely risky, architecture specific and thus unreliable, more robust techniques were pioneered. These techniques revolved around utilizing synthetic processors (or virtual machines) that act as an additional virtual layer between the application and the system, thereby allowing such tools to instrument the binary code safely and control (or simulate) the execution flow.

The instrumentation is usually done using the *dynamic compilation* — a process, where the virtual machine dismantles the binary image into separate blocks which are then dynamically recompiled using a *just-in-time (JIT) compiler*, so that additional instrumentation code is injected. *Valgrind* [46], *Pin* [39] and *DynamoRIO* [5] are three most widely used frameworks based on this principle.

Nevertheless, these *heavyweight* frameworks are not well suited for potential deployment in the field of performance analysis due to the notable overhead (usually both in terms of memory and time) and the insufficient precision incurred by the recompilation and simulation processes.

2.4.2 State-of-the-art: `uprobes` and `kprobes`

The early attempts to make dynamic instrumentation work were quite "*obscure*" [28], yet, they helped to pave the way for more advanced, safe and reliant techniques used nowadays. One of such techniques are *Kernel Probes (kprobes)* (and its user-space counterpart, *uprobes*) which rely on native support from the operating system (the *Linux* kernel in this case).

A kernel probe is a set of handlers — the most important ones being the *pre-* and *post-* handlers — bound to a specific instruction address [23]. If a kernel probe handler is provided and registered (usually using *kernel modules*) for certain address, the kernel injects breakpoint, debug or one of many exception handling instructions at the given location (e.g. `int 3` or `debug-exception` for *x86* architecture).

Whenever execution reaches the trap instruction, control is passed to the user-supplied handlers (also, context is stored and interrupts are temporarily disabled, among other additional kernel operations). User probes leverage a similar mechanism as kernel probes, albeit with some implementation differences such as supporting a different set of handlers and exploiting virtual memory pages to inject trap instructions [10]. The Figure 2.3 illustrate the mechanism of a kernel probe being hit.

Compared to the its static counterpart, the latest dynamic instrumentation techniques offer "*visibility so deep and comprehensive that it can feel like a superpower*" [28]. One of the most appealing feature is the ability to instrument a running application without heavily impairing its execution time or stability.

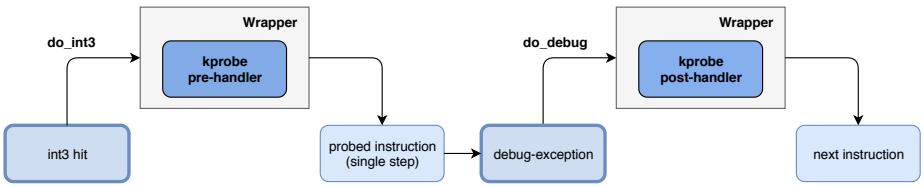

Figure 2.3: An illustration of kernel probe mechanism which utilizes breakpoint and exception interrupts (`int3` and `debug_exception`) and the corresponding handlers (`do_int3` and `do_debug`) to dynamically add instrumentation. Image adapted from [23].

However, dynamic instrumentation has its drawbacks, namely the *interface stability issue* and the *inlining problem*. The interface stability issue is encountered when the internals of the examined program or kernel change and previously available functions or events are renamed or removed which can be a complication for automated tracing tools. Similarly, the inlining problem is caused by the compiler optimization, which renders certain functions uninstrumentable since there is no such symbol to be found in the resulting binary image.

2.5 Performance Analysis Frameworks and Tools

We briefly introduce several tools and frameworks that support dynamic instrumentation and tracing, and could be leveraged for building a performance data collection tool. In this work, we focus on the *Tracer* — a performance data collection tool described in Section 4.2, already introduced in our earlier work [53, 54, 55] — which leverages the *SystemTap* framework. We also propose we can extend the `Tracer` with the *eBPF* framework in order to utilize some of its more advanced features, which are not supported by SystemTap. However, all of the following tools and frameworks had been considered as the underlying instrumentation layer when Tracer was originally proposed.

2.5.1 The Linux Tracing Family

Strace [64, 12] is a tool for tracing *syscall* and *signal* events related to a specific process. Strace relies on the *ptrace* system call (as do also debuggers for example, to implement the breakpoint functionality) to inject code which invokes `SIGTRAP` interrupt for the events (e.g. syscall entry and exit points) as well as to provide a way to access parameters or return values. Thus, when a system call is to be executed, kernel halts the traced process and notifies the tracer so that it can perform its operations, after which the process is resumed.

Ftrace [59, 16, 58] is a tool originally designed to trace kernel functions. However, since then ftrace has been heavily extended to also support call graph and kprobes tracing, tracepoints or various latency tracers. The function tracer leverages compiler-aided instrumentation (e.g. `-pg` option for GCC) to inject a `mcount` function call into *every* kernel function. During the system boot, all `mcount` calls are transformed into the `NOP` instructions. When the function tracer is enabled, these `NOP` instructions are dynamically changed back to the `mcount` function call. The communication (acessing trace records) is done through an internal kernel *ring buffer* structure and the `debugfs` file system.

Ltrace [36, 37] is primarily a dynamic library call tracing tool, however, it can also trace system calls and signals similarly to the *strace* tool. Ltrace relies on the *ptrace* system call which is used to inject breakpoint instruction into the *Procedure Linkage Table (PLT)* — a linkage table that keeps track of assembly instructions needed to call every external function (i.e. located in a dynamic library) since the address of the function is not known prior to the runtime. The injected breakpoint is then utilized in the same way as with strace.

Albeit the `strace`, `ftrace` and `ltrace` are certainly powerful tools for tracing specific operations and events, they all share one common shortcoming from our point of view: not being general-purpose. For this reason, a versatile tracing tool with the required tracing capabilities could be created only by incorporating all of the tracing tools.

2.5.2 `DTrace`

DTrace [15, 27, 9] stands for *Dynamic Tracing* — a complex instrumentation framework designed to provide observability across the whole software, i.e. the kernel as well as the user-space, libraries, signals, file systems, drivers and many more — and was originally developed for the *Solaris* OS. DTrace utilizes *probes* to specify instrumentation points. The means of probe attachment (i.e. the instrumentation) is dependant on the *providers* which implement the instrumentation according to the specifics of the target layer, e.g. kernel functions, syscalls, user application, tracepoints etc. When the probe *fires* (i.e. is triggered by reaching the instrumented instruction), a user-supplied probe handler is executed.

The biggest drawback of the DTrace — until recently, since official port from Oracle has already been released — had been its absence on Linux platforms, apart from some unofficial ports. However, currently the *eBPF* framework is considered to be superior in terms of available features and overall more capable[7].

Compared to the Linux tracing tools, `DTrace` is generic enough to be utilized as an underlying instrumentation framework, however, it is lacking proper Linux support and the language for defining probe handlers is somewhat restricted (e.g. not supporting loops or non-trivial branching) compared to e.g. *SystemTap* or *eBPF*.

[7] http://www.brendangregg.com/blog/2018-10-08/dtrace-for-linux-2018.html

2.5.3 Perf

`Perf` [17, 56, 26], also referred to as `perf_events`, is an event-oriented observability tool that can perform counting, tracing and profiling of events coming from different sources. Originally, Perf focused on exploiting the performance counters subsystem in Linux to implement a profiling tool based on the performance counters.

Hardware performance counters are CPU hardware registers managed by the *Performance Monitoring Unit (PMU)* which is a part of the CPU. These registers count events such as executed instructions, cycles, cache accesses or misses, mispredicted branches and so on. Software events, on the other hand, refer to pure kernel counters such as page faults, context switches, alignment faults or task clock.

However, the Perf tool has been enhanced to also support number of additional events such as kernel tracepoints, USDT, dynamic tracing (provided by the uprobes and kprobes) and timed profiling. Timed profiling is a technique that collects snapshots of counters, trace stacks or other events periodically at an arbitrary frequency (e.g. `perf record -99Hz`). The Figure 2.4 illustrates the available events and their categories in a map.

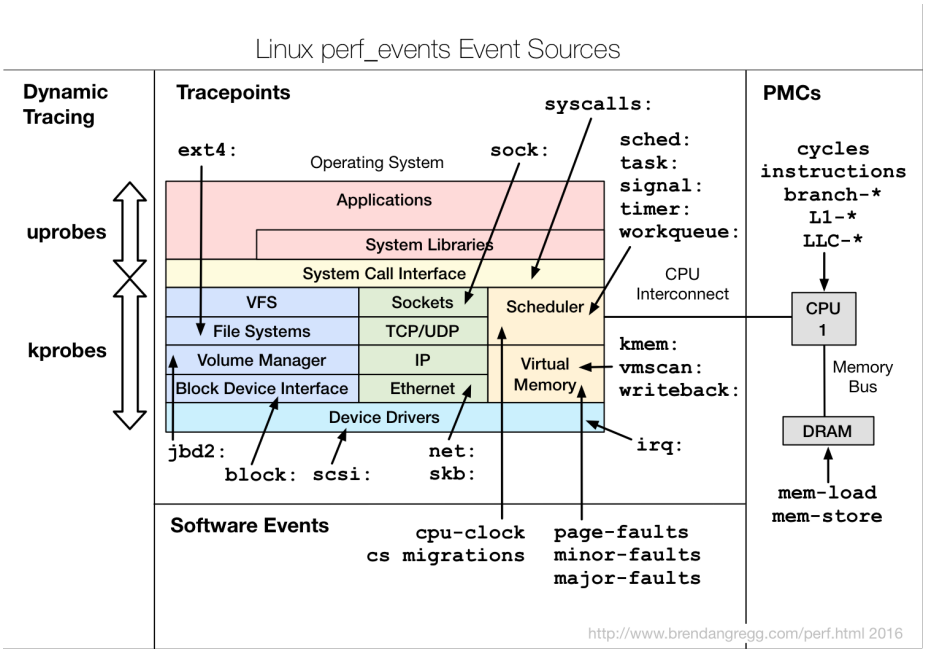

Figure 2.4: A scheme of the available `perf` events, their categories (such as *dynamic tracing*, *static tracing*, *hardware and software events*) and the corresponding tracing and sampling mechanisms (*kprobes*, *uprobes*, *tracepoints*, *performance monitoring counters*, etc.) arranged into a map. The map illustrates the essential system components, e.g. file system, virtual memory, networking, drivers, scheduler, communication buses, etc. Taken from [26].

Perf consists of a collection of commands such as `stat` (for counting events), `record` (for profiling, timed or regular), `record` (for static tracing using tracepoints), `probe` (for dynamic tracing using uprobes and kprobes) and other (e.g. `top` for tracing system calls by process, similar to the linux *top* utility). These commands produce output in a binary format so Perf supplies additional commands for presenting the recorded data (such as `report` or `script`).

Nonetheless, perf does not support user-supplied handlers for probes or events, i.e. it is not possible to precisely define operations that should be executed when a probe is fired or event is encountered, unlike e.g. the DTrace tool which uses its own C-like scripting language for defining probe handlers. Moreover, the binary output format makes it difficult to process the raw output data in a different way than what the built-in interpretation commands offer.

2.5.4 SystemTap

`SystemTap` [24] is a powerful, general-purpose, tracing and profiling framework. As stated in [33], "*SystemTap is not so much a tool as it is a system that allows you to develop your own kernel-specific forensic and monitoring tools*". It supports all of the current state-of-the-art dynamic instrumentation and probing mechanisms: kprobes, uprobes, kernel tracepoints, USDT, performance counters and — thanks to its rich scripting language — even some basic in-kernel programming.

SystemTap utilizes custom kernel modules to inject probes and their handlers, i.e. the actions that are performed when probe fires. The kernel modules are created from the user-supplied scripts in several steps:

1. User writes a SystemTap script containing probes and their corresponding handlers.
2. SystemTap then checks the script for any *tapsets* used (an abstraction that shields away the implementation details of probes and kernel layer, similarly to how libraries are used in programming languages), in which case it substitutes the tapsets with their definitions.
3. The script is translated into the `C` language.
4. The `C` code is compiled into a kernel module that is then loaded.

Due to its complexity, SystemTap used to have issues — with occasionally causing kernel panics or freezes on previous kernel versions — most of which nowadays seem to be resolved. Moreover, SystemTap needs *kernel debuginfo* packages (also depending on the Linux distribution, the installation process may get quite complicated) to function properly and utilize its full potential. However, at the time of *Tracer* (see Section 4.2) development, it still offered much more flexibility and power than perf, had official support (unlike the DTrace) and had stable development branch with most of the important features available (unlike *eBPF* at the time). For those reasons, SystemTap has been used as the underlying instrumentation and tracing layer for Tracer.

2.5.5 eBPF (BPF)

`BPF` [28, 6] stands for *Berkley Packet Filter* — until recently just a network packet filter tool that revolutionized [42], at its time, the way that packet filtering was done. Recently, the *extended Berkley Packet Filter (eBPF)* has made its appearance and turned the BPF into an impressive general-purpose engine inside the kernel that makes it possible to create advanced observability tools — similar to the SystemTap framework.

BPF is essentially a virtual machine inside the kernel that has its own instruction set, storage objects and helper functions. The BPF engine consists of an interpreter and *just-in-time (JIT)* compiler that translate the executed BPF instructions into a native system instructions. The BPF program can be supplied during kernel runtime, without the need

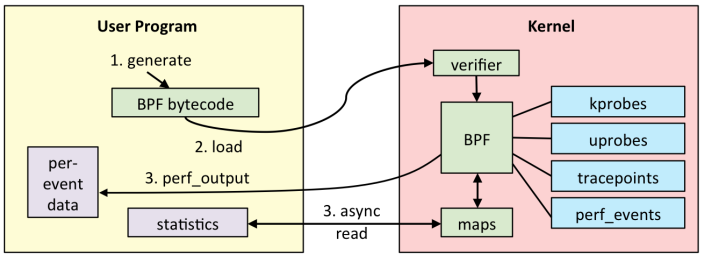

Figure 2.5: The eBPF workflow which consists of generating the BPF bytecode, verifying it, running it via the virtual machine (engine) within the kernel and storing the resulting raw data into the BPF maps or directly as a `perf` output. The illustration is taken from [25].

to recompile or restart any service. Moreover, any BPF program is first of all checked by a verifier that scans the code for potential stability issues that could lead to a crash or kernel panic making it a lot safer and robust than the kernel module approach used in SystemTap.

The communication between kernel and user-space is done through a *BPF maps* [44]: a generic *key:value* data structures created from user-space through a BPF system call. Since the maps are generic, the key and value types can be complex enough to allow for custom output formats. Figure 2.5 illustrates the whole BPF workflow in a simplified way.

This design greatly improves the means of dynamic in-kernel programming and allows the user to run custom mini programs in the kernel. However, a significant drawback is that BPF programming is difficult, since it has to be done using the BPF bytecode. Thus, a BPF frontends (e.g `BCC`, `bpftrace` or `ply`) were created to abstract the BPF internals.

We sumarize all the listed tools and frameworks in the Table 2.1. As shown, the SystemTap and eBPF frameworks support the most of the essential requirements for building a versatile profiling tool. The only difference lies in the support of *fully dynamic tracing* (i.e. the ability to enable or disable the probes and handlers even when a tracing process is running) which is presently supported only by the eBPF framework.

Table 2.1: A summary of all the tools and frameworks and their supported features. We considere the tool as *general-purpose* if it can be easily leveraged by another high-level tool to create versatile performance analysis tool. *Flexible handlers* ensure that specific event handlers support a wide range of operations and programming primitives (such as branching or loops). The *fully dynamic tracing* feature describes the ability to dynamically enable or disable probes and handlers even during an ongoing tracing.

	Linux Tracing Tools	Perf	DTrace	SystemTap	eBPF
Supports probes	✓	✓	✓	✓	✓
Supports tracepoints	✓	✓	✓	✓	✓
Supports counters	✗	✓	✓	✓	✓
General-purpose	✗	✓	✓	✓	✓
Define own handlers	✗	✗	✓	✓	✓
Flexible handlers	✗	✗	✗	✓	✓
Fully dynamic tracing	✗	✗	✗	✗	✓

Chapter 3

Statistics-based Performance Models

We will briefly familiarize the reader with the topic of data models utilized in the area of performance analysis. However, since the field of statistical modelling is a vast subject with great variety of methods, models, techniques or approaches and even their quick overview would be out this Thesis' scope, we decided to focus only on those currently leveraged by the *Perun* framework. We will first emphasize the difference between *parametric* and *non-parametric* models and then follow with a description of four kinds of models: *regression analysis*, *regressogram*, *moving average* and *kernel regression*. In our experimental evaluation, we will use these models to evaluate the impact of individual optimizations on the modelling precision as they are an important aspect of performance analysis.

3.1 The Categories of Models

In statistics, models can be categorized — among other possible classifications — as either *parametric* or *nonparametric*. The description of the model classes in this Section is based on [82, 83, 32, 7, 66].

Parametric models, generally, attempt to estimate an unknown parameter in a known parametric form. In order to do so, a sufficient amount of prior knowledge about the underlying data is required, i.e. the parameters (not their values though) of the model have to be known. Thus, if $\mathcal{M}_\theta$ is a known *parametric form* with unknown *parameter* θ, Θ is a *parameter space* and the studied model belongs to a *parametric family* $\{\mathcal{M}_\theta, \theta \in \Theta\}$, estimating the parameter θ is the main task of the model.

The drawback of a parametric model is its limited flexibility compared to the nonparametric approach and also its dependence on assumptions made about the data. However, the resulting parametric model is guaranteed to produce more reliable results for non-normal distributions or otherwise skewed data.

Nonparametric models, on the other hand, do not assume any specific parametric form $\mathcal{M}_\theta$ (since they cannot be parametrized by a fixed number of parameters), and instead rely on a certain smoothness assumption. By not utilizing a parametric form, we acknowledge that the underlying data *in fact have* parameters, but they are not fit for constructing a model due to their dynamic nature or due to our insufficient knowledge about them.

Nonparametric models are often considered to be more flexible and robust — mainly thanks to fewer assumptions being made about the data, unlike the parametric models. However, to infer a conclusion with high value of confidence, more data is usually required.

3.2 Regression analysis

Regression analysis [14, 55, 7] is a parametric statistical method that investigates the relationship between two (or generally more) related variables x and y. The variable x is often described as *independent*, *predictor* or *explanatory variable*, while the variable denoted as y is referred to as *dependent* or *response variable*. Regression analysis expresses the discovered relationship in the form of *regression function* which can be further used to *predict* the dependent variable (y) for any independent value (x).

For a random bivariate data $(x_1, y_1), (x_2, y_2) \dots (x_n, y_n)$, the most elementary deterministic mathematical relationship is linear and in order to describe it, a *Simple Linear Regression Model* is leveraged. This model is expressed as $Y = \beta_0 + \beta_1 x + \epsilon$, where β_0, β_1 denote the y-intercept and slope, respectively, of a linear function and ϵ is referred to as the *random deviation* or *random error term*. The *true regression line* stands for the resulting linear function — the *truthfulness* refers to the fact that $y(x_i) \neq x_i$, but instead x_i differs from the function exactly by the ϵ_i value. To express the *goodness-of-fit* of the regression function, a *coefficient of determination* (denoted by R^2, values in an interval of $[0, 1]$) is used. The coefficient of determination can be interpreted as the proportion of observed y variation that can be explained by the regression model [14]. Figure 3.1 depicts an example of the *Simple Linear Regression Model*.

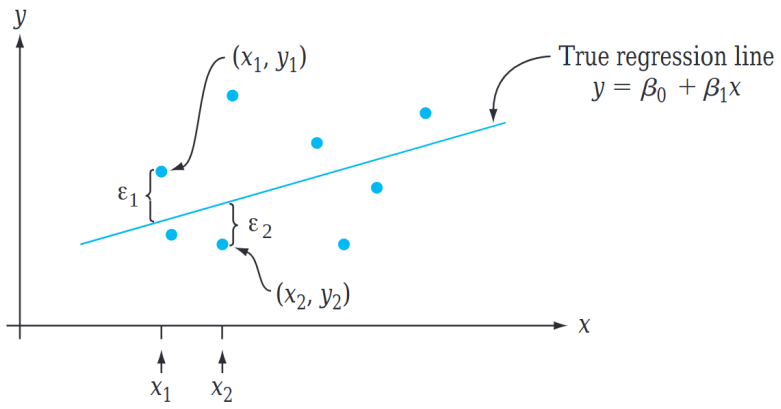

Figure 3.1: An illustration of the *Simple Linear Regression Model* (as found in [14]) that shows the *independent* (x) and *dependent* (y) variables, resulting *true regression line* and the *random deviation* of the value pairs (x_i, y_i).

The regression analysis can be successfully exploited in the field of program performance analysis (as shown, for example, by our previous work [52, 55] which utilized not only linear, but also logarithmic, quadratic or exponential regression to estimate the time complexity of functions) since the independent variable x can be mapped to e.g. workload volume, data structure size or cache limit and the dependent variable y can represent the time consumption of a function or an instruction block. However, in some cases, the independent variable is not easy to identify or cannot be measured — in cases like this, a nonparametric approaches prove themselves useful [66].

3.3 Regressogram

While regression analysis attempts to estimate the regression function $Y_i = m(x_i) + \epsilon_i$ using a parametric form such as $m(x) = \beta_0 + \beta_1 x$, nonparametric methods, like regressogram, do not impose any parametric form (arising from e.g. a linearity assumption) on the $m(x)$.

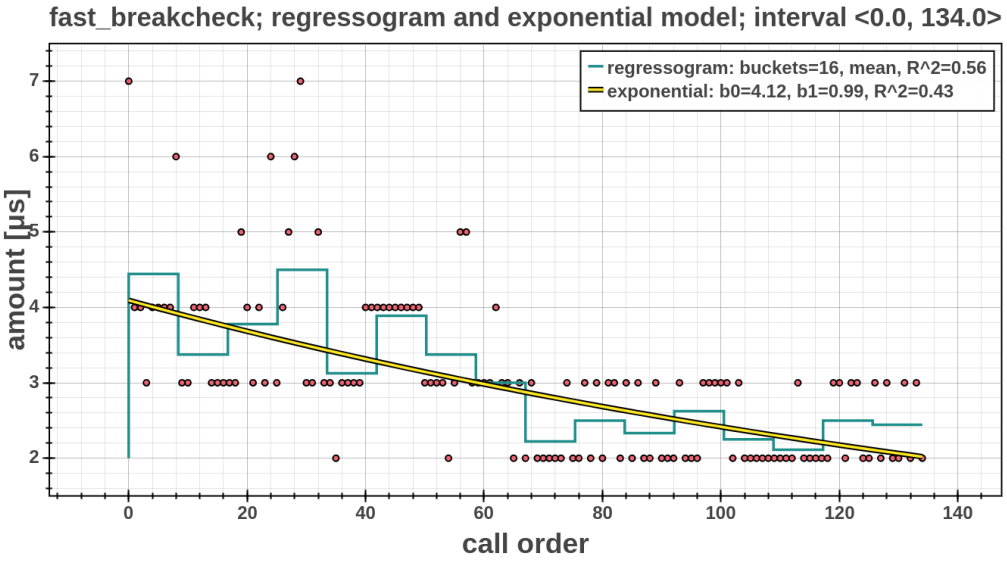

Figure 3.2: An example of the regressogram (cyan) and regression analysis (yellow) methods applied to performance data collected by the Trace collector (see Section 4.2). The x axis represents the call order of the function `fast_breakcheck` and the y axis shows the time consumed by every individual function call. The Figure was taken from [66].

Regressogram [7, 66] can be viewed as a method that utilizes both regression and histogram techniques. Similar to the histogram, the variable space of X is partitioned into a fixed number of equal-width bins — hence why regressogram is also sometimes labeled as *binning* — and then the bin value is estimated as the statistical function of all the (x_i, y_i) values that fall into the bin. Formally, if we assume the X distribution is over $[0, 1]$ and M is the number of bins, then the partition can be expressed as:

$$B_1 = \left[0, \frac{1}{M}\right), B_2 = \left[\frac{1}{M}, \frac{2}{M}\right), \ldots, B_{M-1} = \left[\frac{M-2}{M}, \frac{M-1}{M}\right), B_M = \left[\frac{M-1}{M}, 1\right]$$

The estimation of $m(x)$ can then be computed as:

$$\widehat{m}_M(x) = \frac{\sum_{i=1}^{n} y_i I(x_i \in B_\ell)}{\sum_{i=1}^{n} I(x_i \in B_\ell)}$$

where $I(x_i \in B_\ell)$ represents the membership of the (x_i, y_i) value in the bin (as 1 or 0). Similar to the regression analysis, the coefficient of determination (R^2) may be used to evaluate the goodness-of-fit of the model.

The number of bins used for partition affects the fitness of estimation, and it is thus imperative to choose it appropriately. Various rules for choosing fixed bin sizes were proposed (as mentioned in [66]), however, certain methods deal with this issue by recursively splitting previous bins along the axis directions, such as *regression trees* [47]. The Figure 3.2 illustrates the difference between the regressogram and regression analysis models.

3.4 Moving Average

When utilizing the dynamic analysis and instrumentation, the underlying data (e.g. measured execution time of functions) usually contain the so-called *outliers*, such as time records notably impacted by a scheduler or garbage collector running in the background. These outliers tend to skew the conclusions made about the data by statistical methods such as the regressogram. In order to diminish the impact of such outliers, a *moving average* can be used to smooth the data by averaging the deviations with other values in the proximity.

Generally, the moving average method [66, 80, 79] is a widely used indicator in the area of technical analysis. It averages y values in sub-intervals of x that fall into the data window — a fixed-width interval that continuously shifts from the beginning of x-axis towards the more recent data. The data window can either be centered — in which case the moving average accounts not only for the previous values, but also for the future ones (see Equation 3.1) — or right-aligned where only previous values are utilized (see Equation 3.2). In order to asses the goodness-of-fit, a coefficient of determination (R^2) can be used.

$$MA_t^c(n) = \frac{P_{t-k} + \cdots + P_t + \cdots + P_{t+k}}{n} \tag{3.1}$$

$$MA_t^r(n) = \frac{P_t + P_{t-1} + \cdots + P_{t-n+1}}{n} \tag{3.2}$$

3.4.1 Moving Average Variants

Orthogonal to the centered and aligned classification, multitude of moving average variants exist. In the following we list a brief overview of the selective variants:

Weighted Moving Average is a generic version of moving average where all the averaged data points are associated with a weight, without imposing any restrictions or conditions on the weight values distribution (such as done by the *linear* or *exponential* versions). The right-aligned Weighted Moving Average can be expressed as follows:

$$MA_t(k) = \frac{w_t P_t + w_{t-1} P_{t-1} + w_{t-2} P_{t-2} + \cdots + w_{t-k} P_{t-k}}{w_t + w_{t-1} + w_{t-2} + \cdots + w_{t-k}} \tag{3.3}$$

Simple Moving Average refers to the basic moving average in the Equation 3.1 or 3.2. It is, in fact, an equally-weighted moving average (where $w = 1$ for all data points) that computes the arithmetic mean of n data points.

Simple Moving Median is a modification of the Simple Moving Average that uses a median (of the values in the data window) instead of the mean for smoothing the data.

Exponential Moving Average uses a *decay factor* $0 < \lambda < 1$ to exponentially reduce the weight of the points the more distant they are, as shown in Equation 3.4. This approach guarantees that the recent data points have bigger impact on the new averaged point.

$$EMA_t k = \frac{P_t + \lambda P_{t-1} + \lambda^2 P_{t_2} + \cdots + \lambda^k P_{t-k}}{1 + \lambda + \lambda^2 + \cdots + \lambda^k} \tag{3.4}$$

Apart from choosing the correct weighted model, the width of the data window has a significant impact on the properties of the moving average method. By adjusting the

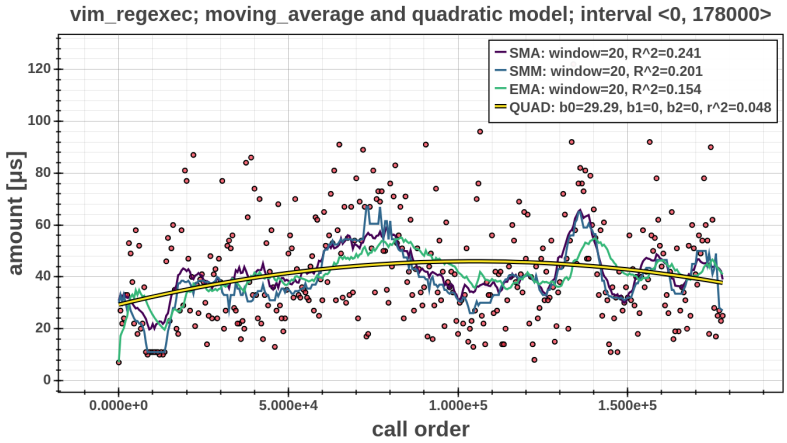

Figure 3.3: An example of moving average variants (*simple moving average (SMA)*, *simple moving median (SMM)* and the *exponential moving average (EMA)*), and the regression analysis quadratic model (QUAD). The x (resp. y) axis represents the call order of the function `vim_regexec` (resp. function runtime). Taken from [66].

width, a trade-off between the noise smoothing and the lag of the data trends detection arises. The narrower the window, the less effective the smoothing (and the smaller the detection lag) is and vice-versa. A comparison of three variations of the moving average technique and the regression analysis is illustrated in Figure 3.3.

3.5 Kernel Regression

Kernel estimate [32, 66, 7, 69, 48, 49] of function m at point x can be described as a weighted average of Y values in the close symmetric vicinity of x, denoted as $[x-h, x+h]$. Intuitively, this approach leverages the assumption of smoothness (as briefly mentioned in Section 3.1) to approximate the regression function $m(x)$ — thus given the function is smooth, the points in close proximity (specified by the *smoothing bandwidth* h) influence the point x. Formally, kernel estimates are expressed as[1]:

$$\widehat{m}(x;h) = \sum_{i=0}^{n-1} W_i^K(x;h) y_i \tag{3.5}$$

where n is the number of points in the kernel and W_i are weights parameterized by the point x, the bandwidth h and the kernel function K. As with the other parametric and nonparametric models implemented in Perun, the coefficient of determination (R^2) can be used to evaluate how well the model fits the data.

Kernel Estimators. Generally, in order to perform a kernel regression, a specific *estimator* and *kernel function* have to be chosen. The *Nadaraya-Watson* estimator (NW), for example, is one of the simplest *local polynomial estimators*, expressed as:

$$W_i^K(x;h) = \frac{K(\frac{x-x_i}{h})}{\sum_{\ell=0}^{n-1} K(\frac{x-x_\ell}{h})} \qquad \widehat{m}_{NW}(x;h) = \frac{\sum_{i=0}^{n-1} y_i K(\frac{x-x_i}{h})}{\sum_{\ell=0}^{n-1} K(\frac{x-x_\ell}{h})} \tag{3.6}$$

[1]The $(x;h)$ notation expresses that the weights, and consequently also the kernel estimation, depend on both the parameters x and h (as denoted e.g. by [32]).

However, the Nadaraya-Watson estimator does not estimate well certain data distributions or models, such as perfectly linear data with no regression error. Such purely linear data may (depending on the marginal distribution of x_i) lead to a nonlinear output due to the fact that the NW estimator approximates the regression function by a local constant: this explains why the NW is also called a *local constant estimator*. Thus, as the smoothing increases (based on the bandwidth parameter), the estimator simplifies to a constant (instead of a linear) function.

Kernel Functions. We have introduced kernel estimators without focusing much on the kernel function K. Every kernel function $K : \mathbb{R} \to \mathbb{R}$ has to satisfy the following conditions:

$$\int K(x)dx = 1 \qquad\qquad K(x) = K(-x) \tag{3.7}$$

The *Box* and *Gaussian* functions, for example, are amongst the most common ones:

$$K_B(x) = \begin{cases} \frac{1}{2} & \text{if } |x| \leq 1 \\ 0 & \text{otherwise} \end{cases} \tag{3.8}$$

$$K_G(x) = \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}x^2} \tag{3.9}$$

However, in practice, the resulting estimates of different kernel functions differ only slightly. On the other hand, choosing the correct bandwidth value h has a much greater impact on the quality of the approximation since the bandwidth controls the smoothing — the larger the bandwidth, the smoother the estimates.

3.6 Comparison of Models

We conclude with short comparison of models implemented in the Perun framework (see Section 4). The evaluation (taken from [66]) is conducted on two performance profiles, with focus on the goodness-of-fit and computation time of each technique.

Each profile contains performance data for 25 different functions in the *Vim* text editor, with an average of 773711 and 42038.55 records (for each function) in the first and the second profile, respectively. In general, compared to the regression analysis, the nonparametric models have achieved a superior *accuracy / computation time* ratio. Specifically, the results demonstrate an improved accuracy along with a significant speedup, or a much more precise results at the cost of a minor slowdown (as with e.g. the kernel regression). Table 3.1 shows a summary of the achieved accuracy and elapsed time for each model. See [66] for more details about the configuration of each methods, e.g. the number of regressogram bins, width of moving average window, etc.

Table 3.1: A short summary of accuracy (R^2) and computation time ($t[s]$) values for each model type. The emphasized values represent the minimum (red) and maximum (green) values in both metrics. This table was taken from [66].

	reg-analysis		**regressogram**		**moving-avg**		**kernel-reg**	
profile	R^2	$t[s]$	R^2	$t[s]$	R^2	$t[s]$	R^2	$t[s]$
#1	0.192	122.90	0.117	45.60	0.551	71.78	0.768	126.68
#2	0.222	11.01	0.415	3.86	0.572	5.28	0.775	76.71

Chapter 4

Perun

We will build our optimization methods within *Perun*: an open source lightweight Performance Version System for continuous performance monitoring [20]. First, we provide a brief overview of the whole Perun workflow and architecture, along with the description of the internal processes that we aim to optimize. Later, we describes the *Tracer* data collector in more details since Tracer is heavily utilized in this work. Last but not least, related work and projects are discussed, evaluated and compared to Perun.

4.1 Overview

Perun is as a wrapper over *Version Control Systems (VCS)*, such as *Git*, and keeps track of performance profiles for different project versions, and it also provides a tool suite for generating, processing and interpreting the performance profiles. Having access to both the project history and performance profiles, Perun attempts to continuously monitor the state of the project from the performance point of view and detect any potential performance changes the moment they appear: it measures and evaluates the performance metric every time a new project version is published (e.g. a *commit* or *pull-request*), and subsequently comparing the obtained results with the performance profiles assigned to the previous (stable) versions. Figure 4.1 illustrates the intended use-case model of Perun.

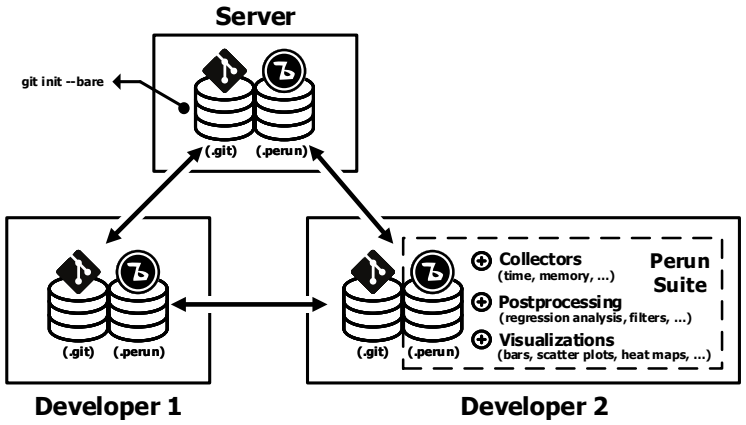

Figure 4.1: An illustration of the Perun framework coexisting with a *Version Control System (VCS)* [20], such as Git, during the project development.

Compared to the naïve approach of storing the performance profiles directly in DB or VCS, Perun offers an advanced performance management with the following advantages [21]:

Context. Perun stores the profiles alongside the VCS and maps them to a specific project version, so the context about the underlying codebase is preserved in performance profiles. Notably, the context can aid the developer in pinpointing the source of the discovered performance changes or help the *collectors*, with the optimized selection of the code locations to measure, based on the source code differences between various versions.

Automation. Instead of having to profile manually whenever a new version is released, Perun can automate the the whole process by the so-called *hooks* in the VCS and the concept of *jobs*: jobs are the sequence of Perun commands, while hooks are triggers of the defined jobs when, e.g. new project version being pushed. The automation model and file formats are inspired by the *continuous integration (CI)* tools, e.g. *Travis CI*[1].

Genericity. One of the core ideas of Perun is to be easily extensible, either by new data collectors, postprocessors or visualisations. Enhancing Perun with new methods is rather straightforward process with minimum requirements or restrictions. Also, Perun employs an unified format (based on a *JSON* notation) to store the performance data, so that every module can further process or interpret the profile. Furthermore, since every technique is implemented in a separate invocable module, Perun leverages this design by treating the modules as basic building blocks for the specification of automated jobs.

Easy-to-use. Since Perun is intended for deployment alongside VCS, the goal was to design a similar *Command Line Interface (CLI)* as used by the Git system — consisting of commands such as `add`, `status`, `log`, `init`, etc. — and thus giving the users a sense of familiarity when using Perun.

4.1.1 Workflow

The main Perun workflow is a series of steps that are executed whenever, e.g. a new version of a tracked project is created. The description of the workflow is based on our earlier work [55]. Figure 4.2 illustrates the workflow of the main Perun feature: automated detection of performance changes.

The workflow process can be described as follows:

1. User initializes a new Perun repository associated with certain project managed by a VCS — in this example, we assume the Git system is used. Furthermore, the user configures the Perun repository and specifies the desired collection, postprocessing and detection methods that should be used to analyze new project versions.

2. New changes to the source code are made, the developer checks them out in the Git repository and creates a new *commit*.

3. By creating a commit, new version of the project appears and triggers the cascade of tasks specified by the jobs. First of all, raw performance data are obtained using one of the available collectors, such as the Trace collector that measures the time

[1] https://travis-ci.com/

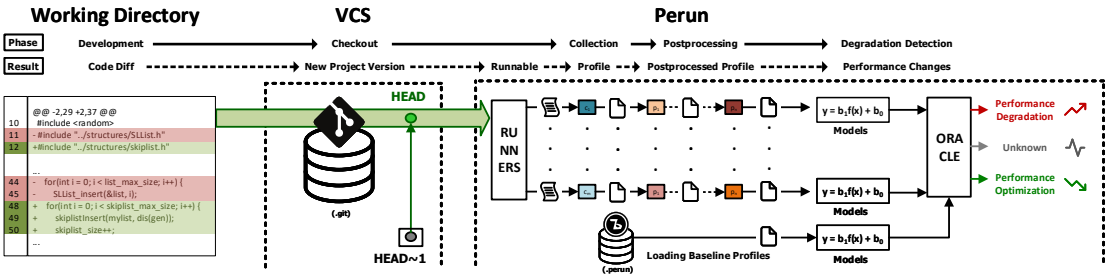

Figure 4.2: A graphical depiction of the Perun workflow [20]. Whenever new project version appears, Perun runs the cascade of tasks (also known as *job*) comprising of methods for performance data collection, raw data postprocessing and degradation detection with respect to the previous project version.

consumption of particular functions or code blocks. The raw data are stored as a *profile* in the Perun repository.

4. Since the raw data from collectors are not suitable for direct interpretation (even though it can be done for small projects), postprocessing techniques are used in order to transform the data or identify potential relations among them. Using the Regression Analysis, for example, we can build the performance models of measured functions — in terms of the model category (e.g. linear, quadratic, exponential) and its parameters. The results of the postprocessing methods are stored in the previously created performance profile, often called the *target profile* in the context of performance change detection.

5. The previous (stable) version of the project is determined and associated profiles (so-called *baseline profiles*) are retrieved from the repository. Such baseline profiles characterize the performance status of the previous project version.

6. The baseline and target profiles are paired with respect to their configuration and parameters (i.e. only profiles using the same executable file, parameters, methods, etc. are matched). For each matching pair, a detection of degradation is performed with the help of methods such as *Integral Method* or *Local Stats* (introduced in [66]).

7. The degradation detection methods provide the user with reports of the discovered potential performance changes. Each performance change is characterized by the *severity*, *location* and *confidence* parameters. While location helps to identify the source of the change, severity and confidence reflect the seriousness and the degree of certainty regarding the estimate, respectively.

4.1.2 Architecture

The architecture of Perun [20, 21, 66], illustrated in Figure 4.3, consists of four main components: *logic*, *data*, *view* and *check*. Apart from those core components, a lot of smaller helper or utility components exist, such as *vcs*, *templates*, *workload*, etc. However, in this section, we focus on a concise description of the four main components. Further, we will identify and discuss the most suitable opportunities for optimizations. schematically illustrates the described architecture.

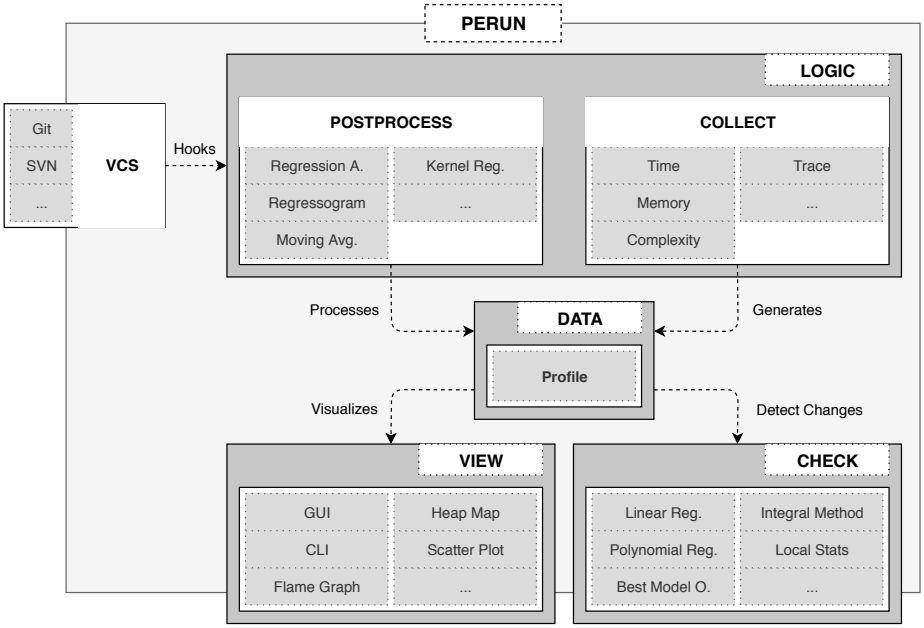

Figure 4.3: A simplified architecture schema of the Perun featuring four main components (*logic*, *data*, *view* and *check*) as well as the VCS module: interface to Git, SVN and similar systems. The Data layer is a core component that handles operations related to the generation and manipulation of profiles. The profiles are further used by the postprocess, view and check components to analyze, visualize or compare the raw performance data.

Logic handles most of the tasks related to the automation (e.g. jobs or *runners* controlling the invocation of the requested methods), CLI interactions, repository configuration, etc. Logic also encapsulates *collectors* and *postprocessors* responsible for measuring and further processing of the raw performance data, respectively. Notable collectors are:

- *Memory* collector gathers records of memory allocations, and overall heap memory usage, in `C` and *C++* programs with various associated attributes. The collector leverages the `libunwind`[2] and custom `libmalloc` libraries to perform the data collection.
- *Complexity* collector exploits the compiler-aided instrumentation of GCC (see Section 2.3.2) and a custom shared library to measure the time consumption and approximate structure size of functions operating on data structures within the executable.
- *Trace* collector, will be described in the following Section.
- *Bounds* collector focuses on estimating (and collecting) the integer bounds and heap-manipulating loops in `C` language (or more precisely, its subset). The collector is simply a wrapper over the *Loopus* tool [63].

Moreover, some of the commonly utilized postprocessors are:

- *Regression Analysis* attempts to fit the data with various regression functions and identify the best fit with respect to square error. The regression method requires dataset with independent and dependent variables.

[2] https://www.nongnu.org/libunwind/

- *Regressogram, Moving Average* and *Kernel Regression* which were already introduced in the Chapter 3.

Data component can be seen as the core of the Perun, since its contains interface for performance profiles, with which every other architecture component interacts. Namely, the data layer handles the profile generation and the resource queries.

View layer provides graphical interpretation of the collected data and models (obtained by postprocessors or checkers). Some of the currently supported view methods are:

- *Scatter Plot* uses two dimensional grid to display data as points and is best suited for outputting trace or complexity results.
- *Bars Plot* displays the profile resources in form of vertical bars which can be either grouped next to, or stacked on top of each other. Bars support large number of different resource types and are easily customized.
- *Heap Map* is best utilized in conjunction with memory collector output. It provides a visualization of the memory address map that keeps track of individual address locations, their usage, their allocated objects, or how often the address is utilized.

Check contains methods for the detection of performance changes which were mostly introduced in [66, 65]. Different methods are employed based on the particular resource type, since some methods work only on raw data, performance models or other specific data types. Some notable detection methods are:

- *Average Amount Threshold*, a simple heuristic, that groups the profile resources based on the *unique identifier (uid)*, computes the averages and compares the values for both baseline and target profiles. Potential change is classified as *optimization* or *degradation* based on the fixed thresholds.
- *Integral Method* uses parametric or nonparametric performance models to detect performance changes. In particular, it builds on the assumption that change occurs if the area below a specific performance model is different in the new target profile — thus the areas are computed using an integral and then the respective results are compared.
- *Local Statistics* divides the performance models into sub-intervals that are analyzed individually. The comparison is done using a collection of statistical metrics, such as integral, average, median, sum, etc. This approach allows for more precise identification of the potential degradation source.

4.2 Tracer

Tracer (also called *Trace collector*) — a successor of Complexity collector [52] — was first introduced in [53] and further enhanced in [54]. Tracer is built on top of the SystemTap framework (see Section 2.5.4) that provides the means to dynamically instrument kernel or executable files. However, Tracer is not just a wrapper over the SystemTap as it provides advanced techniques for full automation of performance data collection. The focus of Tracer is on measuring the time consumption of functions and custom code blocks (defined by the

static userspace probes), while also keeping track of the call hierarchy so that a partial trace of the whole run is obtained. Table 4.1 compares complexity and trace collector.

Table 4.1: Comparison of requirements and features of our old prototype *Complexity collector* and *Tracer*. Although Tracer requires kernel `dbgsym` version, it is not reliant on having access to the source code or compilation process. Also, unlike the complexity collector, Tracer supports dynamic instrumentation and full automation of the collection.

	Complexity	Trace
Requires access to source code	✓	✗
Requires project re-compilation	✓	✗
Requires `dbgsym` kernel version	✗	✓
Collects associated data `size`	✓	✗
Supports dynamic instrumentation	✗	✓
Supports USDT instrumentation	✗	✓
Fully automated collection process	✗	✓

Tracer works roughly in the following steps:

1. User invokes the Trace collector through the Perun interface and specifies the collection parameters: the target executable, code locations to measure, sampling, etc. The code locations (functions and USDT) can either be specified manually, identified and extracted automatically by a selected *collection strategy*, or user may choose a combination of both the automatic and manual approaches.

2. Tracer extracts the instrumentation locations based on the selected strategy (if necessary), builds the configuration object and assembles the SystemTap script file. The script contains *handlers* (blocks of code that will be executed) for every instrumented function or tracepoint.

3. In the next step, the script is compiled into a kernel module, SystemTap is initialized and launched as a background process that collects the performance data w.r.t script.

4. The executable file (either binary file or a script that calls the target executable) is invoked and being traced until the process terminates, or a timeout is reached. The collection data are obtained through kernel buffers and redirected to an output file.

5. Tracer transforms the raw data output from SystemTap into Perun resource records which are then stored in a profile. Before terminating the collection process, the Tracer performs a cleanup of all the used resources, that might have not been properly disposed — such as SystemTap processes, kernel modules, temporary files etc.

The already mentioned *collection strategies* are a key mechanism developed to enhance the automation of the Tracer. Every strategy defines what functions are to be instrumented in order to gather their performance data, without the need to specify them manually by the user. First, the list of available functions is extracted directly from the binary file using the linux utility `nm`[3]. Next, the function list is filtered to match the selected strategy and, optionally, the sampling is applied. Apart from the function symbols, userspace tracepoints are also automatically extracted using a builtin feature within the SystemTap framework. Figure 4.4 graphically illustrates, in a simplified way, the extraction process done by the various collection strategies.

[3] http://man7.org/linux/man-pages/man1/nm.1p.html

Currently, the following collection strategies are supported:

- **Custom** strategy relies solely on the user manual specification of function symbols without leveraging any automatic extraction. However, tracepoints can still be extracted automatically, if requested.

  + Suitable for profiling of a small subset of functions, e.g. the ones most likely to have a performance defect.

  - Unknown or subtle performance bugs are likely not to be discovered.
  - Tedious.

- **All** is a strategy that instruments all of the functions within the executable file with no filtering whatsoever. This strategy is also meant to instrument relevant functions from standard and user libraries, however, this feature has not yet been implemented. Collecting the performance data of all the functions can prove necessary when a covert performance bug has emerged. Nevertheless, the considerably increased overhead and profile size are the major drawbacks of this strategy.

  + Collecting performance data of all functions can prove necessary in case of covert performance bug.

  - A considerable increase in the overhead and resulting profile size.

- **Userspace** strategy filters function symbols that have not been defined by the user, such as various helper functions created by compilers etc. Also, user shared libraries should be instrumented similarly to the **all** strategy — a feature not yet available.

  + A sufficient strategy for detecting most of the performance bugs.
  + The overhead and profile size are not as substantial as in the case of the `all` strategy.

  - Heavyweight (in terms of overhead and profile size) for large projects.
  - Performance degradations associated with usage of standard and system libraries may not be detected.

- **Sampled all** and **userspace** strategies apply sampling to the instrumented functions. This ensures that the instrumentation code is not executed every time a corresponding event happens and thus sampled functions reduce the amount of raw data.

  + Considerably reduces the profile size.
  + Collects enough performance data for further processing.

  - Does not reduce the time overhead sufficiently.

4.3 Related Work

We conclude with a list of selected related works that address the same issues of profiling, performance analysis and performance data collection as Perun and Tracer. We concern ourselves only with projects that leverage dynamic instrumentation — with none or severely restricted access to the compilation process and its modification — and are available for Linux operating system.

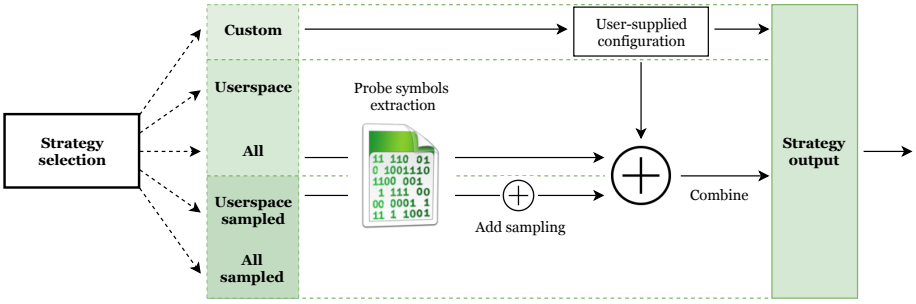

Figure 4.4: An overview of the Tracer strategies [54]. The *custom* strategy only instruments locations specified by the user; the *all* and *userspace* attempt to instrument all, or just userspace functions, respectively. Sampled methods include global sampling of probes.

Valgrind [46, 73] is a widely popular tool suite for, e.g., checking memory usage, cache and heap profiling, or thread debugging. The *Callgrind* tool focuses on profiling function calls and constructing CGs, and so has similar purpose as the Tracer. However, Callgrind is known for its substantial overhead due to its interconnection with the Cachegrind, which greatly hinders its usability in measuring the performance in terms of elapsed time.

OProfile [50] is a profiling tool based on the sampling principle: it periodically examines of various CPU registers, such as the *Program Counter (PC)*, in order to gather details about the hierarchy of running programs and their functions. Furthermore, OProfile also leverages HW counters to access useful performance metrics (see Section 2.5.3). Even though the sampling approach is undoubtedly light-weighted, the precision can be limiting and the amount of obtained data is not sufficient to construct any performance models.

PerfRepo [57] is a repository for performance results. It aims to capture and archive results of performance tests in order to automatically compare performance of subsequent project versions — similar to how Perun works. However, unlike Perun, the PerfRepo does not provide any specific performance tool suite. Moreover, it is designed as a web application and it is thus not possible to compare and archive the performance results without utilizing the server side code. As far as we know, it also does not provide deep VCS integration and the automation is instead performed through the properly configured client side. Still, it is a well-rounded tool for managing performance results generated by other tools.

Other similar tools and frameworks — e.g. the linux tracing family (*strace*, *ftrace*, *ltrace*), *DTrace*, *prof* and others — were already discussed. Multitude of commercial tools such as *Intel V-Tune Profiler*[4], *AMD μProf*[5], *Arm MAP*[6] or *Oracle Performance Analyzer*[7] exist. These tools are generally robust, versatile, packed with features and support broad range of possible use-cases. However, apart from being commercial, these tools usually require corresponding HW (e.g. Intel, AMD, Arm processors) or SW (e.g. Oracle IDE that integrates the profiling tool) to either work, or at least unlock its full potential.

To the best of our knowledge, no profiling framework or tool currently offers deep integration with version control systems as well as an extensible tool suite to collect, postprocess and visualize the performance data. This makes Perun a unique framework for continuous performance monitoring across different project versions in order to inform the user whenever a new severe performance degradation emerges.

[4] https://software.intel.com/vtune
[5] https://developer.amd.com/amd-uprof/
[6] https://www.arm.com/products/development-tools/server-and-hpc/forge/map
[7] https://docs.oracle.com/cd/E77782_01/html/E77798/index.html

Chapter 5

Analysis of Requirements

In the following, we list the selected metrics we will use for the evaluation, the optimization criteria we will focus on in our methods, and the key functional and non-functional requirements we will satisfy. These requirements have to be taken into account when designing the individual optimization methods, however, note that certain requirements may not be fully satisfied — and some may even contradict others — by some of the our proposed techniques.

5.1 Evaluation Metrics

While each individual method focuses on certain primary optimization criteria (with the goal of minimizing or maximizing it), the actual impact of methods on the performance analysis can be measured from multiple perspectives, known as the *metrics*. By selecting a suitable set of metrics, we can precisely capture a wide array of properties for each method, or even leverage a relevant subset of metrics to quantify the optimization criteria. In particular, we propose to assess each method individually based on the following metrics:

I) **M_PT: Profiling Time** [s] represents the duration of all profiling phases (i.e. including the **M_CPT**, **M_CRT**, **M_OO** and **M_ORE** times), i.e. the time spent by running one whole Perun job. Naturally, speeding up the entire profiling process makes measuring large-scale projects significantly more user-friendly and greatly reduces the time requirements for any deep Perun analysis of project repository (e.g., when examining the whole project history for possible performance degradations).

II) **M_CPT: Collection Phase Time** [s] represents the duration of the `collect` phase (e.g., in the Trace collector) only. In particular, we measure the time spent by attaching the instrumentation, running the profiled program and collecting the actual performance data. Note, that we exclude the setup or cleanup phase as well as some other Perun-related operations. In combination with the **M_CRT**, we can use this metric to observe the impact of various preparatory, supplementary or auxiliary `collect` operations, most notably the attaching and detaching of instrumentation.

III) **M_PRT: Program Run Time** [s] represents the duration of the profiled program run[1]. We strive to achieve program run time that is closer to the actual run time (without any instrumentation), so we can reduce the overhead of the profiling and obtain results that are closer to the real program performance.

[1] Note that this run is with the possible introduced instrumentation overhead.

IV) **M_RDV: Raw Data Volume** [MB] comprises the amount of raw performance data collected by underlying collection modules, before the data are transformed into the actual performance profile. Minimizing the volume of raw data speeds up the profile transformation process and, most importantly, prevents the excessive loss of performance data due to the data channels being full (both the SystemTap and eBPF internally utilize some form of kernel buffers that drop data records when full).

V) **M_PS: Profile Size** [MB] comprises the disk space needed to store the resulting Perun profile. Keeping the profiles compact is necessary since they are being stored in the internal Perun directory for future use (as opposed to the raw data, which are usually discarded), and smaller profiles can, naturally, be further processed much faster.

VI) **M_OO: Optimization Overhead** [s] represents the time overhead of utilizing the individual optimizations and all their associated operations, such as approximating the parameters or extracting the resources. Obviously, this overhead should not exceed the time saved by utilizing the optimizations. By minimizing the overhead, we improve the gain of optimizations, and, hence, improve the profiling speed.

VII) **M_ORE: Optimization Resources Extraction** [s] represents the duration of preparing the optimization resources (such as call graph or statistics of previous profilings) required by the individual optimization methods. Distinguishing the time required for the resource extraction from the overall optimization overhead (**M_OO**) offers a more detailed view of where the optimization code spends more time. Also, in many cases these resources can be precomputed, retrieved from internal Perun cache, or shared by several optimizations, so this would induce a one-time overhead.

VIII) **M_PL: Probe Locations** [probes count] represents the number of instrumentation sites within the SUT, i.e., the number of defined and attached probes, which can change, e.g. in case of various probe filtering techniques. Reducing the extent of instrumentation and targeting only the meaningful code locations is the primary approach for achieving profiling speedup (both in the form of collection and probe attaching times) and data volume reduction. Furthermore, it is imperative to balance the reduction of instrumentation points rather than minimize it, since it could lead to not profiling code locations that could still contain potential performance changes.

IX) **M_PLR: Probe Locations Reached** [probes count] represents the number of unique instrumentation points actually triggered during the profiling, since not all of the instrumented locations (as tracked by the **M_PL**) are necessarily reached during the specific profiling with concrete parameters and input workloads. Attaching and detaching excessive number of probes tends to be a time expensive operation (as well as useless, if majority of the attached probes are never reached). Hence minimizing the $M\_PL - M\_PLR$ difference is imperative in speeding up the profiling.

X) **M_HC: Hotspot Coverage** [time coverage %] is the inverse fraction of the total run time spent in the profiled functions that are in the `bottom-level` of the callgraph compared to the `main` function. We define the set of *bottom-level* functions as:

$$\bot = \{f \mid f \in measured \wedge \neg\exists g \in measured : g \neq f \wedge f \subseteq g\}$$

where $measured$ is the set of profiled functions and $\subseteq$ is a subsumption relation $f \subseteq g \Leftrightarrow f_{depth} < g_{depth} \wedge g \in f_{reachable}$, where $depth$ represents the depth of a function

within the call graph hierarchy (where 0 refers to the root level) and *reachable* is a set of functions reachable from f. The intuition behind the subsumption is that provided that g reaches the function f and has a higher-level, then the call duration of f may be included in the call duration of g, thus g *overapproximates* the performance coverage of f. Note, that Tracer currently measures only inclusive time, so we cannot properly compute the hotspot coverage for recursive functions, and hence we omit them from the `bottom-level` set during the raw data parsing. High hotspot coverage indicates that we can quite precisely pinpoint the location of performance bottlenecks. On the other hand, low coverage indicates that lower level functions excluded from profiling are likely to contain the performance hotspot, which we cannot precisely pinpoint.

XI) **M_HCD: Hotspot Coverage Depth** [average function depth] expresses the average call graph depth of all the $\bot$ functions. The average depth value reasonably estimates how thorough the hotspot coverage is: the deeper the depth, the more precise the coverage of the actual call graph nodes preceding the $\bot$ functions.

XII) **M_HCP: Hotspot Coverage Probes** [probes count] is the number of $\bot$ functions, which roughly assess how well the **M_HC** covers different subgraphs of the call graph.

XIII) **M_TLC: Top Level Coverage** [time coverage %] refers to the fraction of total run time spent in `top-level` functions compared to the `main` function itself. We define the *top-level* functions analogously to the **bottom-level** functions:

$$\top = \{f \mid f \in measured' \wedge \neg\exists f' \in measured : f' \neq f \wedge f' \subseteq f\}$$

where $\subseteq$ is defined the same way as in **M_HC**. Note, that we limit ourselves to the set of profiled functions $measured'$ that excludes those, which have lower or equal *depth* value than functions in the first call graph branching (e.g., the `main` functions and possibly other directly wrapped functions)[2]. Using this coverage metric, we obtain an estimate of how time-intensive are the computations directly within the main function or its wrapped functions (which we cannot measure in a more granular way as of now) as opposed to the rest of the functions. The higher the coverage is, the more precise our subsequent analysis (e.g., targeting certain segments of the program) will be, as we will not miss any important hotspot in the code. Figure 5.1 demonstrates the difference between the *top level* and *hotspot* coverage metrics.

XIV) **M_TLCP: Top Level Coverage Probes** [probes count] counts the number of $\top$ functions. The greater the amount of function probes, the more fine-grained top-level overview of the performance distribution is achieved.

XV) **M_FC: Function Complexity** [model categories number] counts the number of different complexity models (see Section 3) as obtained by, e.g., regression analysis. Typically, counting the occurrences of model differences (after applying an optimization method) is necessary to evaluate the potential data skew caused by the optimized profiling and for determining the actual impact of optimizations (since removing constant function has quite different impact than removing e.g. non-elementary function).

[2]Since otherwise, we would always obtain only one $\top$ function: the `main` function.

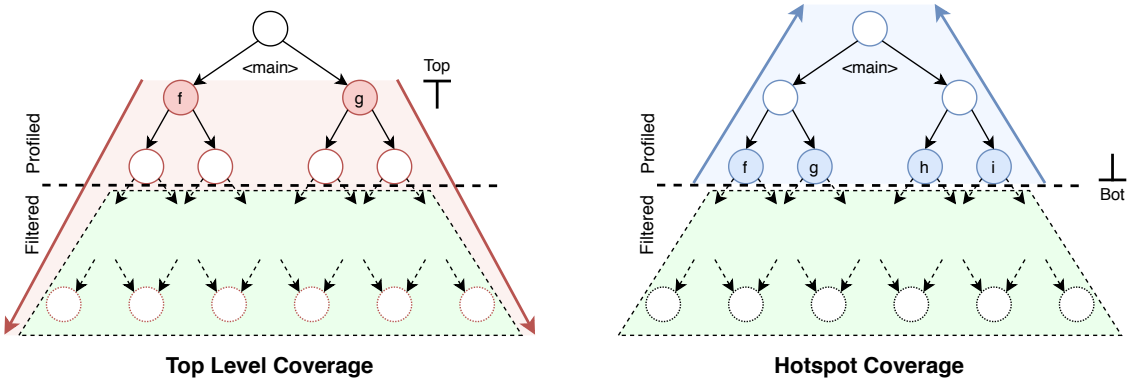

Figure 5.1: An illustration of the `Top Level Coverage` and `Hotspot Coverage` metrics, and their different, yet complementary, approach to quantifying the profiling coverage. While TLC uses `top` ($\top$) functions to assess the portion of the elapsed time that is covered by top-level overview functions, HC identifies `bottom` ($\bot$) functions that can be leveraged to express the portion of total elapsed time spent in actually profiled functions, so that possible performance hotspots can be precisely localized.

5.2 Optimization Criteria

Each optimization technique minimizes or maximizes a particular optimization criteria. Hence, we first select those that we will focus on in our optimizations. Naturally, each technique generally affects multiple parameters at once — we, however, usually pinpoint one (or a few) dominant criteria for each method.

I) **OC_T: Time** consumption of Perun profiling is one of our biggest concerns when analyzing large-scale projects, which includes both the instrumentation overhead and the total profiling time (including performance modelling, etc.). Metrics associated with the `time` criterion are: **M_PT**, **M_CPT**, **M_CRT**, **M_OO** and **M_ORE**.

II) **OC_DV: Data Volume** issues often manifest themselves when too many collection points are monitored (e.g., instrumented or probed) during the collection process, generating an excessive amount of data (up to `GBs` of raw data). However, similar problems can also occur when the program runs for too long even though the set of monitored locations is not particularly large[3]. This particular criterion can be evaluated using the **M_RDV** and **M_PS** metrics.

III) **OC_CP: Collection Points** are the amount of monitored code locations within the profiled program (regardless of the instrumentation, injection or attachment details for specific collection methods). The amount of collection points directly affects the profiling time as well as the data volume. Generally, we distinguish between the two following cases that differ in the severity of the imposed overhead:

- **Instrumented** locations generate overhead during the instrumentation (resp. cleanup) performed before (resp. after) the profiled program is run. Note, however, that these locations may or may not be reached during the actual profiling.

[3]Note, that the compact Profile format of Perun v.0.17 led to considerable reduction already.

- Actually **reached** collection points (*reached* $\subseteq$ *instrumented*) cause additional overhead during the program run by generating profiling data, and thus negatively affecting the precision of the performance data collection.

The **M_PL**, **M_PLR**, **M_TLC** and **M_HC** can be used to assess this criterion.

IV) **OC_F: Freshness** relates to how recently the functions were modified. We argue, that this criterion is, nowadays, the most critical, as the more fresh are the changes the more likely will the developers fix the underlying performance issues. In this work, however, we limit ourselves only to an approximation of the freshness — i.e., profile functions that are classified as *changed* since the last profiled project version. Optimizing the freshness greatly improves the profiling by precisely pinpointing only the relevant collection points. There are no metrics we can use for this criterion.

5.3 Functional Requirements

We believe, that our novel optimizations for performance analysis should aim at the wide developer community and also should be applicable to other performance suites. Hence, we propose they should comply with the following functional requirements:

1) **FR_OC (Optimize at least one Optimization Criteria):** each method must properly optimize at least one optimization criterion. Note, however, that achieving optimization of all the criteria simultaneously by a single method is highly unlikely.

2) **FR_O (Orthogonality):** every method must be implemented in a manner that allows users to utilize several different methods together. By leveraging various techniques at the same time, it should be possible to optimize multiple criteria at once.

3) **FR_A (Automation):** the selected methods must be appliable automatically (e.g. as a part of the Perun workflow), without the need for manual invocation from the user. Moreover, the basic usage of methods should require minimum of user configuration and they should offer as many default parameters as possible, which should be reasonably inferred, e.g., based on the project size and specifics.

4) **FR_MS (Method Separation):** the usage of certain optimization must not in any way limit the usage of another one, i.e, each method must be available as a standalone feature without the need to enable or disable other methods. However, note, we do not prohibit the methods from utilizing results from other selected methods.

5) **FR_PI (Position Independence):** each method has to be designed in an order-independent manner w.r.t combination of multiple methods. Note, however, that different ordering of methods within the optimization sequence can produce diverse results, since, generally, the combination of optimizations is not commutative.

6) **FR_PP (Predefined Pipelines):** the optimization module has to support at least two predefined configurations for automated optimization, the so called *pipelines*, including the selection of optimization methods to apply as well as their configuration.

7) **FR_P (Implemented in Perun):** the methods must be integrated into the Perun framework, so it is ensured the methods are easily leveraged by other Perun modules without the need for external calls to other programs.

8) **FR_SM (Utilize supported analysis methods):** the methods should be primarily based on the techniques described in the Chapter 2 or those already supported by Perun. Furthermore, the proposed techniques should be supported both by the SystemTap (see Section 2.5.4) and eBPF (more details in Section 2.5.5) frameworks.

5.4 Non-functional Requirements

The previous set of requirements focused on the intended behavior of the methods. However, if we want to apply our methods successfully in practice, we believe we should also take into account certain non-functional requirements, in particular:

A) **NR_G (Generic implementation):** even though the optimization methods will be mostly focused on the Trace collector, the implementation should be sufficiently generic to support any other data collectors (satisfying certain interfaces).

B) **NR_AIL (Acceptable information loss):** the information loss (expressed, e.g. as the precision of the performance models), caused by the reduction of profiles, should not skew the results of subsequent postprocessing beyond a reasonable threshold. Moreover, no optimization should lead to a loss of detectable performance change.

C) **NR_S (Scalability):** the optimization architecture and methods have to be designed and implemented such that Perun's profiling will scale well even for medium to large projects up to hundreds of thousands of `Lines of Code` (LoC).

D) **NR_M (Maintainability):** all of the methods should be properly tested and pass the code style checks utilized by the Perun *Pull Request* toolchain (see [20] for details). Furthermore, the implementation must be well documented, the resulting code should be well readable and easily extendable by other developers.

E) **NR_DL (Dependencies and Licensing):** even though the usage of third-party libraries, tools or frameworks for certain tasks can be expected, the number of introduced dependencies must be appropriate to the task being solved. The licensing of the external tools should be compatible with the Perun license.

Chapter 6

Extending Perun Architecture

The latest version of Perun (version 0.18.3), unfortunately, does not meet all of the requirements from Chapter 5, hence, we propose to extend the Perun as follows. First, we extend the architecture of Tracer to support multiple underlying instrumentation and tracing frameworks: using the so called *engines* which will allow us to utilize eBPF framework. Second, we enhance the Perun collection process with new optimization layer that will manage optimizations and their collections (the so called pipelines; see Sec. 6.2.1). The proposed layer will be irrespective of the collector specifics as long as the common interface is implemented (this will allow us to easily leverage the optimizations in any future collectors). At last, we implement several generic modules in the Perun core, such as the `stats` module for storing statistics, metrics and various proprietary data related to a specific project version or performance profile (this will allow us to e.g., utilize statistics from previous profilings to fine-tune the probes configuration, such as sampling, in future profilings).

6.1 Extending Tracer with Engines

The Tracer 0.18 (see Section 4.2) uses the SystemTap framework to inject probes into the profiled program. But, even though SystemTap is quite versatile, it lacks the means to dynamically enable or disable probes *during* the profiling as the SUT is being run. Based on the analysis of the state-of-the-art dynamic instrumentation practices and the possible optimization approaches, we noticed that the collection could be enhanced if we exploit additional framework, namely the eBPF, which does support probe runtime manipulation.

Compared to the SystemTap, the eBPF is based on a more dynamic approach (using lightweight in-kernel virtual machine as opposed to kernel modules utilized by SystemTap), and so it can provide interface for dynamic probe manipulation during the runtime, e.g., we could deactivate probes with too much overhead on the fly. Thus, we propose to implement a new data collector based on the eBPF as an underlying instrumentation framework, and on the *BCC (BPF Compiler Collection)* [3]: one of the currently developed BPF frontends with bindings for the *Python* programming language.

We noticed both SystemTap-based and eBPF-based collectors can share a considerable portion of common functionality. Hence, we decided to redesign the Tracer architecture to support multiple underlying instrumentation frameworks (such as SystemTap or eBPF) using the so called *engines*. Figure 6.1 compares the original Tracer architecture (based solely on the SystemTap framework) and the proposed architecture that is independent of the instrumentation and collection specifics.

A note on the eBPF probing mechanism:

When using BCC, the process of loading the BCC program (containing the definition of probe handlers) is separated from the process of attaching the probes through the API. Thanks to this design, probes can be attached or detached on the go considering the selected handlers are present in the BCC program — as opposed to the SystemTap approach where the loaded kernel module (built from the script with probe handlers definition) implicitly determines the used probes and attaches them automatically.

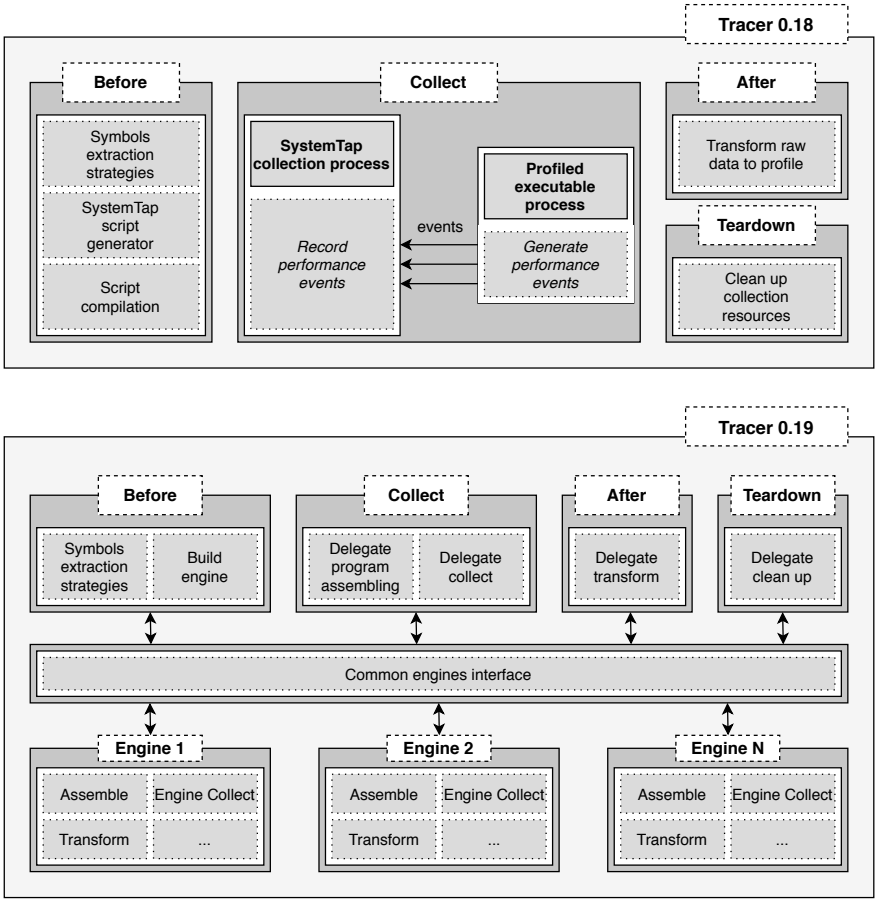

Figure 6.1: A schematic overview of (a) the old Tracer architecture (v0.18) with hard-coded SystemTap usage, and (b) the proposed architecture (expected as version 0.19) that abstracts away supported instrumentation frameworks. The collection process itself is divided into four separate stages: `before`, `collect`, `after` and `teardown`. Each light grey box represents an action, while the arrows represent callback of functions.

Using such design significantly improves the extensibility of Tracer: other instrumentation frameworks can now be easily incorporated into the Tracer in the future. However, each engine must implement the so called *Common Engines Interface* that unifies communication with concrete engines. The interface is defined as the following set of functions, where `**kwargs` refer to the *Python dictionary*[1] and → represents the returned type. We briefly illustrate each function and how it is handled by (1) SystemTap, and (2) eBPF.

[1]For more details, see the Python glossary: https://docs.python.org/3/glossary.html

- `check_dependencies()`: checks that all of the engine requirements are satisfied and all of the dependencies are available, e.g. (1) `stap` or `rmmod` external commands, (2) eBPF has no additional dependencies apart from the Python package. In case anything is missing, the user should be notified of any issues in advance.

- `available_usdt(**kwargs)` → `list`: extracts the USDT probes (see Sec. 2.3.1) from the SUT in a framework-specific manner, e.g., (1) SystemTap invokes external command to extract the probes and (2) eBPF engine utilizes the BCC library interface.

- `assemble_collect_program(**kwargs)`: assembles the collection program w.r.t the specification of profiled probes. Both (1) SystemTap and (2) eBPF engines attempt to assemble the collection program using the language-specific primitives of each framework. Refer to Listings A.1 and A.2 for example programs for both engines.

- `engine_collect(**kwargs)`: runs the actual data collection process. This usually involves running a designated process with elevated privileges — as (1) an external `stap` command, or (2) an internal proprietary Perun module — that injects the probes, captures the probe events, and runs the profiled executable itself.

- `transform(**kwargs)` → `generator`: transforms the engine-specific raw events to the unified Perun resources. Listings A.3 and A.4 show examples of the (1) SystemTap and (2) eBPF raw data format; the actual transformation is straightforward.

- `cleanup(**kwargs)`: frees the used set of resources that **have to** be properly cleaned-up. Failing to do so can lead to serious issues, or even performance data corruption. An example of such resources are: opened files (e.g., raw data file or command output capture file), running processes and child processes, or specific proprietary locks, e.g., for mutual exclusion of multiple running SystemTap instances.

6.1.1 The eBPF Collection Engine

We will now provide a brief overview of the newly implemented eBPF engine. The key difference between the SystemTap and BCC is that while SystemTap is available solely as a standalone system tool (invoked through the `stap` command), the BCC comes with bindings for `Python` language. Launching an external `stap` process has somewhat limited means to control the execution, in contrast, the eBPF engine can leverage the binding to configure and control the data collection directly through the Python API. Specifically, thanks to the direct Python binding, the eBPF virtual machine can be fully operated using only the library interface, thus making the engine more robust, maintainable or extendable, and avoiding numerous pitfalls associated with manipulating an external system process (such as polling the process status or terminating the elevated-privileges process from an unprivileged Perun process). Figure 6.2 illustrates the eBPF engine that operates as follows:

1. The BCC program is assembled (in a subset of the `C` language) through the API function `assemble_collect_program(**kwargs)` resulting into definitions of probe handlers (where *USDT* probes are obtained by the `available_usdt(**kwargs)` function) for every injected probe. Specifically, every profiled function has its own pair of probes (i.e. the `uprobes` and `uretprobes`) and handlers corresponding to the entry and exit function locations. Whenever the entry probe is hit, a timestamp is stored in a `BPF_ARRAY` kernel data structure at a position matching the function ID. When

the exit probe is triggered, the handler attempts to retrieve the previously stored timestamp, computes the elapsed time and generates a `perf_event` filled with raw data (see Listings A.2 and A.4 for detailed description). Note, that this process may be slightly altered by sampling when only on every n-th probe hit the timestamp is stored, thus reducing the number of generated events and the raw data volume.

2. The runtime configuration file is built as part of the `engine_collect(**kwargs)` function. The runtime configuration is loaded by the spawned eBPF instrumentation process and contains parameters relevant to the instrumentation, such as the profiled executable file, collection timeout, probes that should be attached etc. Listings A.6 shows an example of runtime configuration with more elaborate description.

3. The Tracer spawns a new process with elevated privileges that loads the runtime configuration and attaches the probes (as part of the `engine_collect(**kwargs)` function), because performing the eBPF instrumentation requires superuser privileges and Perun is not expected to be run in the superuser mode.

4. After the probes are attached, Tracer invokes the profiled command (through the `engine_collect(**kwargs)` function) as a separate process, polls the event buffer and stores the performance data into an output file (see Listings A.4). If timeout is specified, the profiled command is terminated upon reaching the threshold, otherwise the collection is run indefinitely, or until the process is killed.

5. When the data collection finishes, the elevated eBPF process detaches the probes and terminates. The original process transforms (`transform(**kwargs)`) the raw data from the output file into the Perun profile (for an example of the resulting profile resource see Listings A.5) and cleans up the engine resources (`cleanup(**kwargs)`).

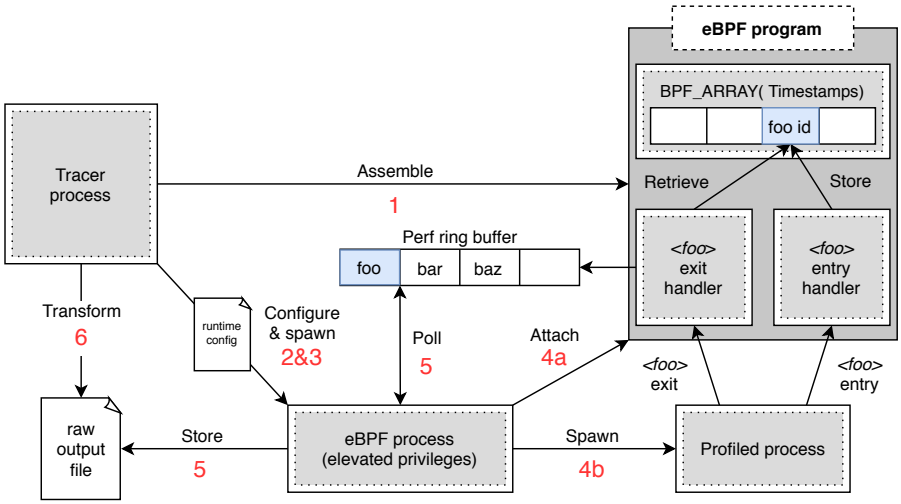

Figure 6.2: An illustration of the eBPF engine: (1) the Tracer assembles the eBPF program with probe handlers; (2) a runtime configuration is built; (3) the eBPF process is spawned with elevated privileges; (4a) the process attaches the probes and (4b) spawns the profiled process which fires events every time a probe is hit; (5) the eBPF process polls the performance events fired by probe handlers and stores the performance data into the raw output file; (6) finally, the Tracer process transforms the raw data into the Perun profile.

The eBPF supports Tracepoints, USDT and performance counters, however, their usage in BCC is still partially limited. It is, e.g., not possible to reliably attach USDT probes into processes that are not already running, and only a modest subset of all the kernel-supported performance counters are available through the BCC interface. However, it is expected that BCC will support the complete probing spectrum in the future, hence our eBPF engine is already prepared to handle such probes. We summarize the new engine as follows:

Dependencies:	`eBPF`	The `ebpf` engine depends on the underlying eBPF technology which must be supported by the kernel (at least version 3.18).
	`BCC`	The BCC provides an interface for operating the eBPF in-kernel virtual machine through a Python interface; programming directly in the eBPF bytecode is rather complicated.
Limitations:	`C/C++`	The engine currently supports instrumentation only for C/C++ programs.
	`Executable`	Only dynamic binary instrumentation is supported; only single-executable instrumentation is automated (e.g., shared libraries are not automatically instrumented).
	`Probe Limit`	The eBPF instrumentation has a limit on the number of simultaneously attached probes (approximately 1000 probes).
Resources:	`ebpf_process`	The `ebpf` process spawned with elevated privileges in order to operate the eBPF virtual machine.
	`program`	The *collection program* file with probe handlers.
	`runtime_conf`	The *runtime configuration* file used by the eBPF process with elevated privileges to configure itself.
	`data`	The raw data output file.
CLI		The proposed CLI for Tracer 0.19 can be found in Table A.1.

6.2 Optimization Architecture

The main workflow of Perun is implemented in the `runner` module responsible for running and managing the collectors and postprocessors. The actual collection is broken into four steps (`before`, `collect`, `after` and `teardown`), hence, we propose to interleave these phases with optimization phases. In particular, we propose to implement selected new phases that will be called before the actual runner phase. Such optimization layer will allow to run selected optimizations without the need to manually invoke them from the collectors. This architecture satisfies the **FR_A (Automation)**, **FR_P (Implemented in Perun)** and **NR_G (Generic implementation)** requirements (see Chapter 5).

We selected the following phase transitions to run selected optimizations:

- `before` → `collect` (*pre-optimize*): at this point, most of the collection parameters (e.g. the target executable, its parameters and workload, diagnostic modes, etc.) should already be configured, the collection program should be prepared for assembly[2], and the probe locations should be clearly identified (e.g. as the function and USDT locations obtained from the *collection strategies*). We expect that optimizations employed at this stage will mainly optimize the collection process itself, specifically, the set of instrumented probes (according to, e.g., the **OC_F: Freshness**). In case of Tracer, at this stage, the collection program is not yet assembled and thus filtering and eliminating some of the probe locations is still possible. Hence, limiting ourselves only to the subset of necessary probes can induce significant optimization.

[2]We assemble the collection program in the collect phase so that the optimization techniques can modify the set of instrumented probes or their configuration (such as the sampling) to avoid re-assembly.

This pre-optimization interface expects a `config` object (defined in the Listings A.7) within the collection configuration (since the whole configuration is being passed to the optimization layer as a *keyword arguments* `**kwargs`).

- `after` → `teardown` (*post-optimize*): at this point, the collection had been terminated and the profiling data had been generated. We expect that certain optimization techniques employed at this stage will either work iteratively across multiple profiling runs (and, hence, will need to gather profiling statistics *after* the collection is done) or further optimize the size of the resulting profile, thus targeting the **OC_DV: Data Volume** criterion. In particular, in this phase transition, the Tracer process had already transformed the raw performance data into the unified profile resources (available as a `profile` within the collection configuration) thus allowing to compute statistics (such as call count or total time per function, Q1, Q3 and median values of recorded duration, etc.) in a generic way. The computed statistics can be further exploited in subsequent profiling to, e.g., filter out functions with extreme call count.

At last, we noticed that some optimizations could potentially be employed during the collection itself. In particular, we will mention eBPF where we can leverage the dynamic probe manipulation to attach or detach probes during the profiling. Indeed, such optimization has to be, at least partially, managed directly from the engine during the collection.

6.2.1 Optimization Pipelines and Parameters Prediction

We designed the optimizations as a part of Perun architecture with their selection available through the CLI summarized in Table A.2. Usually, for efficient profiling session one uses several distinct optimizations, however, the manual configuration of numerous techniques can become a tedious job. Thus, we propose to enhance the optimization process with two additional techniques: (1) the concept of *optimization pipelines*, a fixed sequence of optimizations; and (2) automatic prediction of suitable parameters for regular profiling. Figure 6.3 illustrates the new optimization layer.

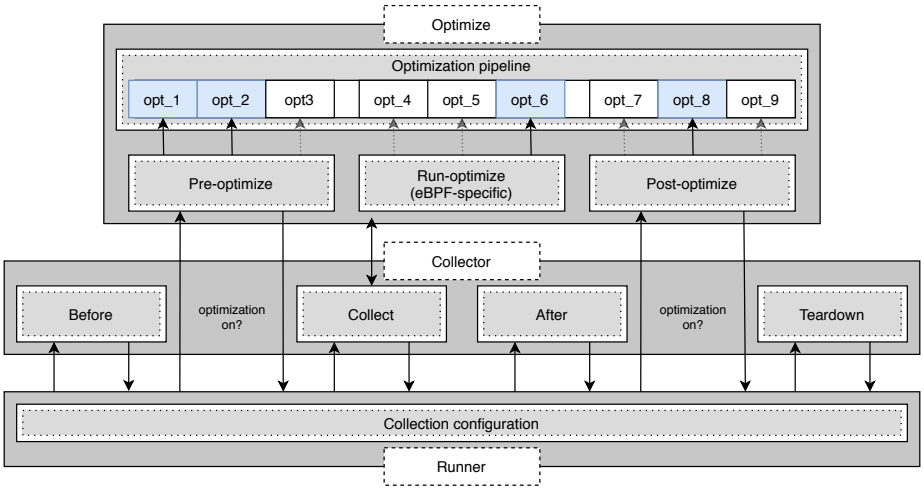

Figure 6.3: A schematic module overview of the new optimization layer and its interaction with the `runner` and collectors. The `runner` has been modified to automatically perform the optimizations (if enabled) between the selected collection phases, independently of the specific collector. Blue modules represent selected optimizations for some concrete run.

The first enhancement, the *pipelines*, are predefined sequences of optimizations that aim to be a quick user default to configure the profiling optimizations according to some criteria (e.g. reducing the profile size, collection time etc.). Note that the user still has the ability to alter the pipelines through the CLI and, thus, perform a more deeper profiling of SUT. We identified several pipelines and offer them in Perun interface (see Sec. 7.3).

While it would be possible to use fixed parameters for optimizations, we argue that for practical usage they should be inferred dynamically to scale well with the project size. Thus, the other enhancement is a *parameter prediction* module that estimates the optimization parameters based on the selected project metrics (both static and dynamic) such as the number of functions, maximal call graph depth, various statistics, etc. We include the full list of the optimization parameters and their proposed prediction formulae in Table A.4.

6.2.2 The `stats` module

At last, we extended the Perun core with the `stats`: a module designed to store and manipulate various developer-defined statistics for certain profiles and project versions. We chose such design, because these statistics are not optimization-specific; are related to project and repository; and, could be used by other modules (or user) for further usage.

For each project, we store `stats` as an internal directory tree within the `.perun` directory with an architecture similar to Git or Perun internals. Specifically, it uses the Secure Hash Algorithm (SHA) value of the project versions (as labeled in Git) to store the created files accordingly. Further manipulation of the stats files (updating and retrieving the stats contents; lookup of the created stats files across the project versions; or deleting) is executable through API or CLI described in Table A.3.

Since the `stats` module is not the main focus of this work, we will only briefly describe its internals. Selected optimizations utilize the `stats` to store collection statistics, extracted call graphs or the optimization parameters. Every created optimization stats file is stored in the appropriate version directory and identified by the profiling command, arguments and workload. Thus, when the same command is being profiled, the previous stats file can be easily leveraged. Figure 6.4 illustrates an example stats directory with several files corresponding to different project versions; Listings 6.1 shows an example of a stats file.

Figure 6.4: An example of the stats hierarchy, where folders are named after the SHA of the project version, each directory stores stats file for corresponding version.

```
"bio_read_bits": {
  "total": 164598,
  "Q3": 3.0,
  "sampled_count": 67591,
  "count": 67591,
  "min": 2,
  "median": 2.0,
  "Q1": 2.0,
  "IQR": 1.0,
  "max": 19,
  "sample": 1,
  "avg": 2.4352058706040745
}
```

Listing 6.1: An example of a stats file: statistics about the profiled functions in the previous profiling (see Sec. 7.1.2 for definitions) for function `bio_read_bits`.

Chapter 7

Optimization Techniques

Finally, we introduce the focus of this work: the seven optimization techniques for performance data collection. For each optimization we discuss their features, their requirements, limitations, and the criteria they target. First, we describe the *optimization resources* (such as *function call graph* or *control flow graph*) that are shared by certain techniques and are needed to optimize the collection. At the end, we elaborate on the concept of optimization pipelines (as already introduced in Sec. 6.2.1) in more details and present the three implemented pipelines with concrete sequence of configured optimizations.

7.1 Optimization Resources

The optimization module provides shared resources: internal data structures used in the specific optimization methods. While some of these resources can be optional for a given method and serve merely to improve the precision of the results, others are essential for successful optimization. The resources are either gathered by the optimization module itself during previous profiling, or extracted from the project using external tools. Currently, we utilize the *call graph*, *control flow graph* and *dynamic statistics* resources. In the following, we will briefly define these resources, describe how they are collected and how can they be utilized to optimize the collection process.

7.1.1 Call Graph and Control Flow Graph

Informally, a *call graph (CG)* represents the *caller-callee* relationship (i.e. calls from the *caller* to the *callee*) between program functions. In the context of profiling, call graphs express the inclusive consumption of resources and can be used to locate the performance bottleneck with more precision. Formally:

Definition 7.1.1. Given a program P, a *simple call graph* of P is a rooted directed graph $G = (V, E, root)$ where V is the set of program functions, edges $E = (caller, callee) \subseteq V \times V$ represent the function call from *caller* to *callee* in P and *root* is the root vertex of the graph.

Note, that such simple definition of CG is insufficient w.r.t a context-sensitive interprocedural analysis (as noted by [29]) and thus cannot accurately capture the exact caller-callee relation across different analysis-time states — mainly since no additional information about the call context (such as the stack trace) is available to distinguish the different states. Nevertheless, we argue that while using advanced context-sensitive extraction algorithms could

provide more precise CG, it would come with the trade-off of increased time and memory consumption. Still, we believe that even the limited CG will suffice to our use case.

Informally, a *control flow graph (CFG)* contains all of the execution paths that a program may traverse through, using a graph notation where nodes represent a set of non-jump instructions and edges represent jump instructions (i.e., jumps in the control flow). In the context of profiling, control flow graphs express the most performance-intensive execution paths and can be leveraged to pinpoint performance hotspots *within* a function. A *control flow graph (CFG)* and *basic block* can be defined as follows [2, 35]:

Definition 7.1.2. A *basic block* is a linear sequence of program instructions with at most one entry point and at most one exit point. Program entry blocks might not have predecessors that are in the profiled executable; program terminating blocks have no successors.

Definition 7.1.3. A *control flow graph* is a directed graph $G = (V, E)$ in which the nodes V represent basic blocks and the edges $E \subseteq V \times V$ represent control flow paths.

Since both the CG and CFG are extracted simultaneously and internally are stored in the same structure, we will denote the CG and CFG as *Call Graph Resource (CGR)*.

Extraction. We perform a static extraction of both the CG and CFG from the project binary using the `angr` analysis framework [61]. We also considered the `radare2`[1] or `gen-callgraph`[2] projects. However, while the former is a rather advanced reverse engineering framework, it is designed mainly as a command-line tool (although bindings to Python are available through additional dependencies), and the latter utilizes only naive extraction algorithm. Hence, the `angr` framework — designed as a Python library and exploiting advanced extraction algorithms — appeared to be the best option. Although, the static reconstruction of call graph tends to yield overapproximated results, compared to the dynamic approach, it considers all of the possible execution paths and, thus, the extraction has to be performed only once per binary executable file (which is especially important as the analysis can be significantly time consuming for large-scale projects, and for different combinations of program arguments and program workloads). The extraction results are directed graph representation of the application CG and CFG.

The `angr` framework. `Angr`[3] is a python framework for manipulation and analysis of executable binary. It exploits proprietary `CLE Loads Everything`[4] file loader and binary code lifter `PyVEX`, to hide the specifics of different binary formats while providing common analysis methods for the user. Generally, `angr` can be used for various manipulation and analysis tasks (such as *symbolic execution*, *hardening*, *exploit generation*, etc.), however, we leverage only the control flow reconstruction capabilities of the framework.

Angr supports multiple analyses with varying extraction speed and precision of the resulting CFG (faster approaches may yield a result faster at the cost of approximating certain computations). We, in particular, use the `CFGFast` analysis that attempts to build the CFG using lightweight approach and heuristics. This is opposed to the `CFGEmulated` which simulates the program execution and performs expensive data flow analysis, however, according to our experiments, it can take up to 20x more time to finish the extraction for

[1] See https://github.com/radareorg/radare2.
[2] See https://github.com/onlyuser/gen-callgraph.
[3] See http://angr.io/ for more details.
[4] See https://github.com/angr/cle.

large-scale projects, while also consuming considerably more memory in comparison with `CFGFast`. We limit the entry address of the `main` function as the starting point of the analysis and disable heuristics for automated lookup of other possible entry points. This way, we effectively obtain the CFG as a *rooted* directional graph with nodes representing only functions *reachable* from the `main` function. We observed, that this alone can provide up to two times extraction speedup while not affecting the precision of the relevant (and further utilized) part of the CFG. Furthermore, we restrict the analysis scope only to the executable binary file itself, not including shared libraries and references to external objects since the Tracer itself does not yet support automated configuration of probes in external libraries (however, support for manual configuration has been added in this work).

Although the accuracy of the CFG extracted by this configuration may be slightly inferior to the fully-fledged `CFGEmulated` analysis, our initial experiments with `angr` demonstrated that the `CFGFast` analysis can scale better for large projects (we experimented with the `CPython-3.8.2` executable binary).

The CGR construction. The `CFGFast` analysis reconstructs the CFG of the program as directed graph, and generates CG as a side product. The CFG is broken into basic blocks, and each block consists of sequence of `ASM` instructions. However, since certain node and graph properties are not part of the extracted CFG or CG by default, we pre-compute these properties so they may be easily accessed by our methods (without the need for potentially costly re-computation). Hence, we convert the CFG and CG to CGR with all the necessary properties of both the CG and CFG for the individual methods. We define CGR as follows:

Definition 7.1.4. Given $CG = (V_{CG}, E_{CG}, root)$ and $CFG = (V_{CFG}, E_{CFG})$, a *call graph resource* is a tuple $R = (\lambda_F, \lambda_C, \lambda_L, V_0, gd, \lambda_{CFG})$ where:

- $\lambda_F = \{(v, Callers, Callees) \,|\, v \in V_{CG}\}$, where $Callers = \{u \,|\, (u, v) \in E_{CG}\}$ and $Callees = \{w \,|\, (v, w) \in E_{CG}\}$, is the *function map* that keeps references to all callers and callees of the function vertex v.
- $\lambda_R = \{(v, reachable) \,|\, v \in V_{CG}\}$ is the *reachable map* where *reachable* is a set of reachable functions from v defined as follows:

$$\{r \,|\, r \in V_{CG} \wedge \forall r \in reachable : \exists \text{ sequence } v = v_0, v_1, v_2, \ldots, v_m = r \\ \text{such that } (v_j, v_{j+1}) \in E_{CG} \, \forall 0 \leq j \leq m-1\}$$

- $\lambda_L = \{(l, V_l) \,|\, l = 0, 1, \ldots, \omega\}$ is the *level map* where ω is the length of the longest detected acyclic directed path (starting in *main* with level 0), $V_l = \{v \,|\, v \in V_{CG} \wedge ELP(v) = l\}$ is the set of vertices (function nodes) that have the same *longest path* length estimate (matching the *level*) and ELP is the *Longest Path Estimate* function.
- $V_0 = \{v \,|\, v \in V_{CG} \wedge Callees[v] = \emptyset\}$ is the set of *leaf* vertices of the CG.
- $gd = max(Levels)$ is the maximal detected acyclic path length (*level*).
- $\lambda_{CFG} = \{(v, CF) \,|\, v \in V_{CG} \wedge CF \in CFG\}$ is the *control flow map* where $CF = (v, B, E)$ is the vertex control flow graph (i.e. the control flow graph of the given function v) with the corresponding *basic blocks* B and *edges* E.

Informally, λ_F stores the call graph structure (nodes and edges) in a dictionary-like structure which is more convenient from the implementation perspective. The λ_R is used to evaluate the significance of a function f based on the number of unique reachable functions r from f. Furthermore, λ_L allows for a quick access to all the functions with the same *level* estimate for optimization methods based on, e.g., traversing the call graph levels sequentially, while the *gd* is used mostly for the parameters prediction. The V_0 is the set of call graph leaf functions that, similarly to λ_L, allows the methods to traverse the call graph in the reversed order (i.e., from the bottom functions up to the `main`). At last, λ_{CFG} partitions the whole CFG into CF subgraphs that represent the control flow of each corresponding function, since internally the CFG is analyzed on a per-function basis. As per our previous experience with profiling numerous projects, we assume the following:

Assumption 7.1.1. Given a function call site, we expect that its total call count in this particular path-context increases proportionately to the (trace) path length from the root node to the call site in the given *call graph*. This implicates that the functions represented as leaf nodes within the call graph are likely to be invoked more often than other functions on the path from the root node to the leaf node.

This assumption — and the fact that we have exactly one handler for each function and distinguishing different path-contexts (e.g., by callers) within the handler is not a trivial *nor* time-inexpensive operation[5] — means that for each function we can effectively leverage only one call count value approximation (inferred globally across all the different path-contexts) for the optimization purposes.

However, estimating the actual call count value (although just approximate) solely on the call graph structure is extremely inaccurate, so we will instead (still unsoundly) operate mainly with the so-called *depth-based prediction* used to compare two call graph functions f and g as follows: $\psi(f) \leq \psi(g) \Leftrightarrow |LP(f)| \leq |LP(g)|$, where ψ is the call count approximation and LP is the longest path. Unfortunately, computing the exact length of the longest path is not feasible in our case, since the *longest path problem* is known to be NP-complete for directed graphs with loops [31]. Thus, while constructing most of the *CGR* properties is rather straightforward, computing the *level map* λ_L structure requires to estimate *longest paths*. We propose to use a heuristic based on a hierarchical *callers* inspection that computes for each call site a lower bound of its level.

Estimating the longest path for a callsite. We propose an algorithm that estimates the length of the longest path (also denoted as *level*) of each function within the call graph by leveraging *Breadth-First Search (BFS)* (interleaved with local, focused BFS that operates on cyclic subgraphs), and that exploits heuristics used for nodes ordering and placement employed in, e.g. graph visualization libraries and tools, such as `Graphviz`[6]. The Algorithm 1 illustrates the procedure of the level estimation as a pseudocode.

First, we start from the root `main` node and inspect the graph vertices in a global BFS manner at line 5. For each node, in case all of its callers were fully resolved (i.e. present in the *finished* set) we determine its *level* as the maximum of all callers' and callees' $levels + 1$. Once we fully resolve the node by assigning a *level*, we expand the *inspect_queue* with all of its outgoing edges $(node, callee_1), (node, callee_2), \ldots, (node, callee_n)$.

[5] Albeit, this is technically possible in our implementation, our experiments have shown that the overhead precise CG analysis is significant. We argue, such trade-off would not result into a better optimizations.

[6] https://www.graphviz.org/

Algorithm 1 Level Estimator for all call graph vertices

Input: Function map $FM = \lambda_F = \{(v, callers, callees) \mid v \in V_{CG}\}$
Output: The $finished$ set with *level* estimates needed to construct λ_L

```
1: procedure ESTIMATE_LEVEL(FM)
2:     finished ← candidates ← ∅        ▷ The level estimates; functions not fully resolved
3:     inspect_queue ← [(FM.root, root_callee, 0) | root_callee ∈ FM.root.callees]
4:     ▷ The global BFS phase
5:     while inspect_queue do
6:         caller, callee, level ← inspect.pop()
7:         if callee not in finished then
8:             candidates.add(callee)
9:             ▷ Everything is initialized with level ← 0, levels ← [], inspected_callers ← []
10:            callee.levels.append(level)
11:            callee.inspected_callers.append(caller)
12:            if callee.callers = callee.inspected_callers then
13:                callee.level ← set_level(callee)
14:                finished.add(callee)
15:                candidates.remove(callee)
16:                for all c_callee in callee.callees do
17:                    inspect_queue.push((callee, c_callee, callee.level))
18:        ▷ The focused BFS phase
19:        while not inspect_queue do        ▷ The set is ordered by the level estimation
20:            c ← lower_bound_representative(candidates)
21:            finished.add(c)
22:            candidates.remove(c)
23:            c.level ← set_level(c)
24:            for all c_callee in c.callees do
25:                inspect_queue.push((c, c_callee, c.level))
26:    return finished
27: procedure SET_LEVEL(func)
28:    ▷ Compute the maximum level of all callers and callees (even unresolved)
29:    callers_max ← max([max(cr.levels) | cr ∈ func.callers])
30:    callees_max ← max([max(ce.levels) | ce ∈ func.callees])
31:    return max(callers_max, callee_max) + 1
32: procedure LOWER_BOUND_REPRESENTATIVE(candidates)
33:    ▷ Representative must have the currently lowest level estimate
34:    min_level ← min([max(c.levels) | c ∈ candidates])
35:    representatives ← [c | c ∈ candidates ⇔ max(c.levels) = min_level]
36:    return sorted(representatives)[0]
```

The global BFS ends when the *inspect_queue* becomes empty. That either means that (1) all of the nodes were fully resolved and have been assigned a level, or (2) the graph contains loops and the remaining nodes cannot be fully resolved until the cycle is *broken*. We can break the cycle by fully resolving at least one of the node in the cycle. Then the estimation process can continue, again, in the global BFS manner.

We *break* the cycle in CG by the inner cycle on line 19 that traverses the sub-call graph in a focused BFS approach. We pick one or more *candidates* (i.e. unresolved) nodes that

are part of the cycle, approximate their *level* as a *supremum* of the current *levels* estimate (i.e., ignoring candidate's calleers and callees that were not fully resolved) and enqueue their outgoing edges. For example, if a candidate has the following *levels* estimate, where ? denotes unresolved level: $\{caller_1 : 10, caller_2 : 15, caller_3 :?, callee_1 : 12, callee_2 :?\}$, we estimate the resulting $level = 16$. The *candidate* nodes are basically lower bound representatives of the unresolved nodes of the cycle (a set ordered by the current *level* estimates). By selecting a node from the candidates' lower bound, we increase the chance of resolving more unresolved callers from other candidate nodes with a higher level estimate — although with a possible propagation of the estimation error.

The algorithm terminates when all nodes have been fully resolved with a *level* estimate. The estimator scales well even for enormous graphs with thousands of vertices, *strong components* with hundreds of vertices and up to tens of thousands of edges, so that the construction time of the λ_L is negligible with respect to the resource extraction process. Specifically, the λ_L construction takes only $\approx 0.046s$ for the `CPython` (see Section 8.1) CG which contains approximately 1800 vertices, 8000 edges and with more than 500 vertices in the largest strong component, while the call graph extraction itself takes about $48s$.

Using CGR in optimizations. We use the CGR of the input SUT throughout the *pre-optimize* transition phase (see Section 6.2). Methods that do directly leverage the CGR are primarily based on static analysis approaches that traverse the graph in different directions (e.g., root to leaves and vice versa) and identify functions that should (not) be profiled according to certain conditions or properties (e.g., the vertex level).

7.1.2 Dynamic Statistics

The *dynamic statistics* resource are summaries and statistics obtained from the collected data or from the collection process itself. Note, that this resource is created directly by the `Optimization` module during the *post-optimize* transition phase and as such cannot be utilized immediately in the same collection run. Instead, the resource is intended either for iterative optimizations[7] or, as an additional (and usually optional) parameter, for achieving more precise optimization results. Specifically, the statistics can be exploited to, e.g., adjust probe sampling according to the corresponding function call count, or even exclude functions that are invoked too frequently from the profiling altogether. Currently, the *dynamic statistics* resource contains the following metrics for each profiled function:

- **Sampled Count** is the number of collected performance records for the given profiled function in context of profiled program with given arguments and workloads. The count is independent of any kind of sampling settings.
- **Sample** stores the sampling configuration for the given function.
- **Count** is an estimate of the actual number of function calls, with the sampling settings taken into account. Since the first function call is always recorded, the estimation formula is $sampled\_count + ((sampled\_count - 1) * (sampling\_value - 1))$.
- **Total** is the sum of the collected records (e.g. the elapsed time of each function call).
- **Min, Max and Avg** store the minimum, maximum and average of the records.

[7] Iterative optimizations work by leveraging gathered data about the optimization effectiveness of the previous profiling to further improve the effectiveness of the next iteration.

- **Median, Q1 and Q3** are the three quartile values that split the record values into four equally-sized segments. Quartiles are generally the preferred indicators for data sets (compared to e.g. the maximum, minimum and average) since they are not skewed by the outliers and can be leveraged to approximate the data spread.
- **IQR** (Interquartile range) is the difference between the first and third quartile ($Q1 - Q3$) and thus covers the middle 50% of the data distribution.

An example of a Dynamic Statistics record for a specific function can be seen in Listings 6.1. Similarly to the *CGR*, we store the *dynamic statistics* resource within the `stats` directory in Perun to avoid recomputation and to use in subsequent analyses. However, unlike the CGR, we expect the dynamic statistics resource to be exploited by various Perun techniques, methods or analyses, all of which not necessarily related to this work.

7.2 Proposed Optimization Techniques

The goal of this work is to implement a number of optimization techniques that attempt to speed up the data collection process or reduce the data volume while also minimizing the inevitable information loss: a trade-off of the optimizations. We propose we can achieve this mainly by identifying locations (e.g. functions) that generate enormous volumes of data, and yet their impact on the program performance is insignificant. We claim, we can completely exclude such locations from the profiling, or at least sample their records. Such approach will definitely lead to a considerable optimization of performance testing.

Naturally, the core of each optimization problem is determining how to effectively select the candidate functions and how to configure the corresponding collection probes (e.g., what should the sampling ratio be) for the subsequent collection process.

For each proposed technique, we (1) discuss the underlying motivation and intuition, (2) describe the general idea of the method, (3) file the optimization to chosen optimization stage, and (4) list the possible implementation approaches (if the implementation can be approached in more than one ways). Moreover, we specify the inputs, the parameters, required and optional resources, expected output and targeted optimization criteria. In some cases, we provide a pseudocode of the method or a schematic diagram; in case the method is straightforward we omit such illustrations and only describe the general idea or interesting points. At last, we assess the advantages and disadvantages of every method, elaborate on the typical expected use-cases and discuss possible future work and extensions.

7.2.1 Static Baseline

Static Baseline optimization is based on the formal static analysis of the project source code. Specifically, we leverage the *resource bounds analysis* (with the focus on *amortized complexity analysis*), which is implemented in the Perun `Bounds` collector: a wrapper over the well-established `Loopus` tool [63].

Parameters:	`sources`	The source files of the project.
	`complexity`	The lowest complexity class used for filtering, i.e. we remove any function with complexity lesser or equal than `complexity`.
	`keep_top`	Protected top `keep_top` CGR levels.
Resources:	`CGR`	(Required) To access to the CG functions and levels.
Output:	`CGR`	(Modified) To remove filtered functions from the CG.
Dependencies:	`Loopus`	The underlying bounds analysis tool; requires LLVM toolchain to operate, and the `clang-3.5` compiler.

Limitations:	`C language`	The limitation of the `Loopus` tool.
Phase:	`pre-optimize`	
Application Scope:	`medium-scale`	Only for projects with simple compilation (e.g., no advanced Makefile or auto configuration features).
Primary OC:	`OC_T`	
Time Overhead:	`noticeable`	Due to the compilation process.

Motivation and Overview. `Loopus` is a static resource bounds analyser that converts the input source program to an abstract program model, and then leverages the symbolic execution and system of difference constraint equations to find a precise, amortized upper bound of loops and functions. The advantage of `Loopus` is that it can prove the time complexities[8] of program loops and functions. However, the Loopus still has its limitations: it cannot infer bounds for certain classes of non-trivial loops (notable cases, as described by the authors in [63], include when, e.g. we fail to find a termination proof or to infer an upper bound invariant for some variable reset) or loops that manipulate with the heap[9] .

Still, we believe we can exploit the `Bounds` collector (and the `Loopus` tool), in the *pre-optimization* phase, to statically determine the complexity of functions. Based on these complexities we can then subsequently filter out non-complex functions: since, naturally, the performance bottlenecks are mainly caused by function with non-constant complexity.

Implementation. We leverage the existing `Bounds` collector in Perun: a `Loopus` wrapper. Note, however, that we use `Loopus` with unsound heuristic, that extrapolates any traversal of dynamic data structure into a singly-linked list traversal. We argue, that in our case, using unsound heuristics in order to handle more program classes could be worth it.

In the optimization, we first internally call the collector, yielding the Perun profile. For each function, the profile contains so called *local* and *total* bound records, where local bounds represent individual loops and code chunks in the function, and total bound represents the bound of the whole function. Every bound is identified by its `class`, i.e. the polynomial degree of the bound in the worst-case asymptotic complexity notation (e.g. $\mathcal{O}(1), \mathcal{O}(n), \mathcal{O}(n^2), \ldots, \mathcal{O}(n^m)$), its `bound` in form of the ranking function and its location `uid`. Listings 7.1 and 7.2 show an example of a local and total bound records for a sample project. One can see, that both bounds belong to the same function, and while one of its local bound is constant, overall the function `encode_segment_bit_plane_coding` is linear in the size of the structure `bpe`.

[8] Note, however, these are upper bounds, so they cannot be effectively used to find performance bugs.

[9] There exist a well-established approach that transforms the input heap-manipulating program into integer program proposed, e.g. in the Ranger [19] or the Thor [40] tools. However, these approaches are heavily dependent on shape analyses and, thus, still cannot scale well even on medium sized projects.

Listing 7.1: Local bound record

```
"bound": "4",
"type": "local bound",
"class": "O(1)",
"uid": {
  "column": 6,
  "line": 2724,
  "source": "bpe.c",
  "function": "
    encode_segment_bit_plane_coding"
},
"time": "0.0"
```

Listing 7.2: Total bound record

```
"bound": "1 + max(Select(0, bpe), 0)",
"type": "total bound",
"class": "O(n^1)",
"uid": {
  "column": 4,
  "line": 2781,
  "source": "bpe.c",
  "function": "
    encode_segment_bit_plane_coding"
},
"time": "0.0"
```

By the construction, the `Bounds` collector returns exactly one total bound for each function. We use this total complexity as an indicator in our Static Baseline optimization. Currently, we distinguish between the `Constant`, `Linear`, `Quadratic`, `Cubic` and `Quartic` complexities (the `Loopus` expresses the complexities only in polynomial notation). Note, that we argue, that it is enough to limit ourselves to these classes, because any higher class (such as non-elementary, exponential, etc.) can never be omitted by the performance analysis as these have the considerable impact on the program complexity.

The main idea of the optimization is then straightforward: we identify functions with complexity below or equal to the provided threshold complexity, e.g., the `Constant`, and remove them from the set of profiled functions (unless they are in the *protected* CG levels specified by the `keep_top`). By default, we filter out only the `Constant` functions. The intuition is that proved constant functions are usually not the real location of the performance bottlenecks; the only case they are the source is when they are invoked many times by other functions (or new performance bugs are introduced by recent changes, which can, however, be covered by other optimizations). However, any such caller will have non-constant complexity and will not likely by omitted from the analysis, and hence, we will preserve the detection of potential performance issue. On the other hand, if the constant function is called many times, its overhead in the collection will be significant.

Method Assessment. The Static Baseline targets primarily medium-sized projects which contain significant amount of functions with constant (or even linear) time complexity that can be safely removed — as we remarked, such functions are usually not as significant from the performance point of view and only generate a considerable amount of data. Moreover, this method is restricted to projects that can be analyzed using the `Loopus` tool, i.e., the `LLVM` toolchain and compilation process should be supported on the target host. Large-scale projects can benefit from the Static Baseline method as well, however, with the current version of the `Bounds` collector, the codebase must support direct source compilation using the LLVM, thus no automated configuration or *Makefile* generation should be employed. Furthermore, we identified the following advantages, disadvantages and shortcomings:

+ Fast, scalable and sound[10] underlying static analysis technique.
+ Targets functions which have small individual performance impact but generate the majority of data.

- Requires access to the source files, compilation process and LLVM ≤ 3.5.
- Provides incomplete results, i.e., the analysis fails for some functions.
- Currently does not support Makefile compilation.

[10] For non-heap-manipulating programs

Future Work. Currently, the main issue that limits the effective usage of Static Baseline for large codebases is the missing support of Makefile compilation. Thus, the leveraged `Bounds` collector has to be extended to not only handle the source files, but also the Makefile, if any. Since not all Makefiles also support the LLVM toolchain, various options for transforming supplied Makefiles should be further explored. Other limiting issues stem from the limitations of the Loopus itself, which does not support some specific classes of loops as well as more precise and sound analysis of heap-manipulating loops.

7.2.2 Dynamic Baseline

Dynamic baseline optimization is an iterative method that leverages metrics gathered from previous collection runs (stored in the Dynamic Stats described in Section 7.1.2) of the same command configuration, i.e. the collection command (profiled executable), its arguments and workload — however, not necessarily the same optimization settings. In particular, the method attempts to identify functions that can be omitted from subsequent profiling.

Parameters:	`soft_threshold`	Minimal number of func. f calls when we start to check if f has constant time behaviour.
	`hard_threshold`	Maximal number of allowed func. f calls for profiling.
Resources:	`CGR`	(Required) To access the changed functions within the CG and to lookup wrapper candidates.
	`Dynamic Stats`	(Required) To access the selected function metrics gathered from the previous collections.
Output:	`CGR`	(Modified) To remove filtered and wrapped functions.
Phase:	`pre-optimize`	Detection and filtering process.
	`post-optimize`	Gathering of `Dynamic Stats` for next optimization.
Application Scope:	`large-scale`	Even for large-scale projects (unlike Static Baseline).
Primary OC:	`OC_T`	
Time Overhead:	`minimal`	Due to computation of Dynamic Stats.

Motivation and Overview. The *Static Baseline* method can identify how functions should, in theory, perform in terms of a speed, and then filter out those that will potentially not impact the performance at all. However, the upper bounds do not actually reflect how functions really behave on the input configurations and workloads. A non-complex (e.g. constant or linear) function can still be a considerable bottleneck if it exceeds order of minutes, and, on the contrary a complex (e.g. exponential) function can be insignificant, if the source of its complexity does not manifest at all. Hence, the *Dynamic Baseline* method aims to complement the Static Baseline: by covering most of the same classes, while also covering some additional ones. Note, that we ensure that the Dynamic Baseline can be used as a standalone method — in situations when Static Baseline cannot be leveraged — or as a complementary dynamic analysis that covers cases missed by the Static Baseline.

In particular, we utilize the *Dynamic Stats* resource to identify functions that have a *constant elapsed time* behaviour during the common SUT usages. Note that, we limit ourselves to detecting constant behaviour only, however, the method could be extended by using the *regression analysis* module implemented within the Perun. Yet, we decided to leverage the dynamic stats metrics and implement lightweight detection of constant functions (as described in the `Implementation` Section) to minimize the time overhead — which would otherwise be significantly higher if we exploited the least squares approach (i.e., as utilized in the regression analysis module) — in exchange of lower expressive power.

Moreover, we identified another use cases of Dynamic Stats. We can, e.g., filter out functions that cross a hard call count threshold. We argue that functions with excessive number of calls introduce significant overhead (the probe handling is a time-intensive operation if performed frequently) as well as generate enormous volume of raw data. At last, we detect the so-called *wrapper* functions, i.e. functions that only call one function with minimal own overhead[11]. Naturally, it is generally sufficient to measure only the wrapper.

Implementation. We propose to implement the Dynamic Baseline as a generic function that based on a list of filters removes functions from profiling. Algorithm 2 illustrates the Dynamic Baseline method as a pseudocode, including all of the proposed filters.

Algorithm 2 Dynamic Baseline Function Filtering

Input: Dynamic Stats (*stats*) and the set of filtering functions (*checks*) to apply, e.g. $check \leftarrow [(call\_limit\_filter, hard\_thr), (constant\_filter, soft\_thr), (wrapped\_filter, wrapper\_threshold\_ratio)]$.
Output: The set of functions ($filtered$) to exclude from profiling.

```
1:  procedure FILTER(stats, checks)
2:      filtered ← ∅
3:      for all func in stats do
4:          if func.is_changed() then    ▷ We obtain this indicator from the Diff Tracing.
5:              continue
6:          for all check, threshold in checks do
7:              if check(stats, func, threshold) then
8:                  filtered.add(func)
9:                  break
10:     return filtered
11: procedure CALL_LIMIT_FILTER(stats, func, threshold)
12:     return func.calls > threshold
13: procedure CONSTANT_FILTER(stats, func, threshold)
14:     if func.calls > threshold then
15:         ▷ Compute the resolution and IQR ratio to determine if func is constant.
16:         resolution ← func.median < median_resolution
17:         iqr_ratio ← func.iqr < func.median * constant_median_ratio
18:         return resolution ∨ iqr_ratio
19:     return False
20: procedure WRAPPED_FILTER(stats, func, threshold)
21:     ▷ Wrapper: function with a single callee func that has the same number of calls.
22:     for all wrapper in stats.find_wrappers(func) do
23:         if func.median < wrapper.median * threshold then
24:             return False
25:     return True
```

For each previously measured function with computed statistics (in the *stats* resource), we first check if it has been changed since the last project version[12]. We skip such functions, since we emphasize that they must be measured in order to detect new performance changes.

[11] We can mention, e.g. several functions in the CCSDS codec used in evaluation in Section 8.2.2, which contain switch that calls coding and encoding functions in either integer or floating point arithmetic.

[12] Note, that we obtain this indicator from the *Diff Tracing* method described in Section 7.2.4.

The generic implementation of Dynamic Baseline ensures that it can be easily extended with different filtering approaches based on the dynamic statistics. In its current implementation, we supply three filtering methods that correspond to the previously described filtering mechanisms: `call_limit_filter`, `constant_filter` and `wrapped_filter`.

- **call_limit_filter** compares the approximate number of function calls[13] (i.e., the `count` statistics) with the user-supplied `hard_threshold`: if the threshold is exceeded, the function is automatically filtered out.

- **constant_filter** compares the function calls (`count`) with the `soft_threshold`. In case it is exceeded, we perform the lightweight check for constant functions as follows. Using the `constant_median_ratio`, we verify if the `IQR` is a negligible fraction of the `median` — which indicates that the potential slope of the linear function approximating the elapsed-time behaviour (*if we would* construct such function) is below the set limit, thus manifesting a constant-like behaviour — or if the `median` is below the minimum `median_resolution`. Low median resolution implies that the used clock precision (in the utilized collector) is too low to adequately measure the duration of the function call (with respect to the introduced overhead and *observational error*) and further performance modelling of the function would thus be significantly biased. Note that this technique is an unsound heuristic that attempts to approximate partial results (i.e., only for the constant functions at the moment) of the regression analysis (moreover, since we work with dynamically obtained statistics, achieving *soundness* is rather difficult). Figure 7.1 attempts to illustrate this concept. The values of the internal parameters were experimentally chosen and adjusted based on the results obtained from the evaluation projects (see Section 8.1), however, we plan to further fine-tune the parameters based on additional future evaluation results. Note, that this is complementary to the approach we proposed in [52] that identifies constant functions based on the least square regression analysis.

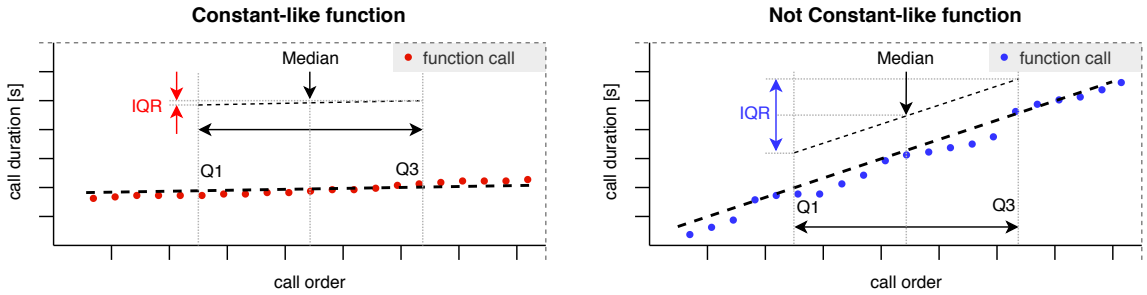

Figure 7.1: A comparison of constant (red) and non-constant (blue) behaviour. The idea is to compare the `IQR` (*interquartile range*) and *median* values of the function's data set: when the $IQR/median$ ratio is sufficiently small, we assume that the function is constant.

- **wrapped_filter** identifies (and filters) the wrapped functions as follows. We list all *callers* of the analyzed function `func` and check that each has only a single callee (`func`) with the same number of calls as the `caller` (`wrapper`) function. This indicates direct control flow from the `wrapper` function to the `func`, e.g., no iterative function calls in a cycle. Next, We compare `median` (see Dynamic Stats) of both the

[13] The number is approximated in case we use the call sampling.

`wrapper` and the `func`. If $func.median > wrapper.median * wrapper\_threshold\_ratio$ holds, i.e. the `func` elapsed time is close to the `wrapper` duration (specified by the parameter `wrapper_threshold_ratio`), we assume that the `wrapper-func` relation holds. The intuition is the caller `wrapper` function performs minimal additional performance-intensive computations. if the described constraints hold true for all `func's callers`, then the `func` is removed from the set of profiled functions.

Method Assessment. *Dynamic Baseline* is a universal optimization that can be successfully leveraged by projects of any size. Its lack of any requirements or dependencies makes it a viable alternative for the *Static Baseline* in cases when its usage is limited. However, the first iteration of the method (i.e., when a dynamic stats are not yet available) does not provide any optimization itself. Hence, we generally recommend to employ additional optimization methods for the first iteration.

One of the disadvantages is that functions excluded from profiling do not produce any statistics and new project versions do not trigger a reset of the Dynamic Stats, so functions excluded in the previous version will be still excluded in the new version. Thus, any potential performance bug, introduced by a new version, in an excluded function will not be detected (however, if the performance impact is severe, the slowdown will likely propagate to a higher-level function which is not excluded). One can then use the Diff Tracing method in combination with the Dynamic Baseline to solve this issue, as Diff Tracing guarantees that functions identified as *changed* will not be excluded from profiling.

+ Universal iterative method with no additional requirements or dependencies.
+ Precisely filters functions based on metrics from the previous profilings.
- If *dynamic stats* are not available, the first iteration filters nothing.
- Hard to handle function changes, since disabled probes produce no metrics.

Future Work. The primary shortcoming of the Dynamic Baseline method is missing built-in support for detecting function changes and subsequent automated re-enabling of the corresponding function probes in order to gather updated metrics. Currently, this issue is resolved either by a manual and user-triggered dynamic stats reset (thus forcing a collection of new baseline metrics), or enabling the Diff Tracing alongside Dynamic Baseline.

7.2.3 Call Graph Shaping

The *Call Graph Shaping* is a family of static analysis methods that exploits the structure of CG to identify and filter functions that are, e.g. more likely to be invoked more times than the remaining functions. Namely, we introduce three distinct Call Graph Shaping approaches: *top-down* (`Trimming`), *bottom-up* (`Pruning`) and *matching* (`Matching`). We base the approaches on the Assumption 7.1.1 (although it is mostly leveraged by the `Trimming` method), and, hence, we focus only on estimating the assumed call count without the need to measure (or at least approximate) the actual call count.

Parameters:	`keep_top`	The number of top call graph levels, counted from the root, that will be protected.
	`min_functions`	(Trimming) The minimum number of functions that have to remain in the call graph after the trimming.
	`keep_leaf`	(Trimming) The flag whether leaf functions should be kept in the remaining CG levels.
	`chain_length`	(Pruning) The maximal length of the pruning paths.
Resources:	`CGR`	(Required) To analyse the CG structure.
Output:	`CGR`	(Modified) Removed whole levels (trimming) or functions (pruning) from the call graph.
Phase:	`pre-optimize`	Modifying the call graph structure.
Notable Trade-off:	`time-precision`	Significant reduction of profiling time at the cost of diminished precision of hotspots (less functions are probed).
Application Scope:	`large-scale`	
Primary OC:	`OC_CP`	
Time Overhead:	`minimal`	

Motivation and Overview. Estimating the real call count for each function using static analysis is, indeed, a challenging task, but we believe we can achieve similar results by leveraging the *depth-based prediction* (see Sec. 7.1.1) instead. In particular, we can use the CGR of the SUT, and specifically its pre-calculated λ_L (*level*) property, as the base for the *depth-based predictor* to *over-approximate*[14] the assumed call counts. We can then use this prediction to, e.g., remove top N functions that have the highest predicted call count.

Generally, the Call Graph Shaping family of methods focuses on the CG structure and depending on the specified *depth* either (a) cuts out whole levels in a *top-down* manner or (b) filters specific functions in a *bottom-up* approach. The top-down approach (trimming) cuts a portion of the CG below the given depth. On the other hand, the bottom-up (pruning) approach traverses in reverse all the paths up to the *depth* length, starting from the $\perp$ functions (as defined in Section 5.1 in metric **M_HC**), and cuts all of the traversed functions that are not in the protected levels (`keep_top`).

We implemented the *top-down* and *bottom-up* techniques as a separate Call Graph Shaping modes called `trimming` and `pruning`, respectively.

Implementation of Trimming In this mode, we iterate the CG levels in a top-down order up to the specified `keep_top` threshold and remove lower functions. Note that we automatically remove the *leaf* functions as well, unless the `keep_leaf` parameter is set to `True`; we argue (based on our prior experience with profiling numerous projects) that leaf functions tend to be invoked more frequently than others, thus causing significant time overhead and generating large volumes of raw performance data. Since the trimming is implemented to keep at least `min_functions` remaining in the CG, we may keep some functions below the specified `keep_top` threshold — in particular, we select functions based on the *reachable criterion*, i.e. we select those that can reach most, unique functions, since they are more likely to cover a performance bug. We propose to use two pre-defined thresholds (`keep_top`) : `soft` and `strict` with the threshold being median (50%) and first quartile (25%), respectively, of the total number of CG levels. The values were experimentally chosen in order to achieve either severe or mediocre optimization by cutting up to a total of ¾ (`strict`), or ½ (`soft`) CG levels containing functions with the highest assumed call count. The Algorithm 3 illustrates the trimming.

[14]The predictor assumes the worst-case, i.e., the longest path from the root node.

Algorithm 3 Call Graph Trimming

Input: Call Graph *cg* and the remaining parameters are defined in the introductory Table.
Output: The *trim* set specifies functions that are excluded from profiling.

```
1: procedure TRIM(cg, keep_top, min_functions, keep_leaf)
2:     keep, trim ← ∅, ∅
3:     ▷ Iteration starts from root node, i.e., level = 0, 1, 2 . . . , ω.
4:     for all level in cg.levels do
5:         if level ≤ keep_top then
6:             ▷ Identify the leaf functions in the level, if necessary.
7:             k ← [f | f ∈ level.funcs ∧ (keep_leaf ∨ ¬f.leaf)]
8:             t ← [f | f ∈ level.funcs ∧ (¬keep_leaf ∧ f.leaf)]
9:             keep, trim ← keep ∪ k, trim ∪ t
10:        else
11:            if |keep| < min_functions then
12:                ▷ Select more functions to satisfy the min_functions condition.
13:                f ← sort(level.funcs, key = |f.reachable|)
14:                split_at ← min_functions − |keep|
15:                k, t ← f[: split_at], f[split_at :]
16:                keep, trim ← keep ∪ k, trim ∪ t
17:            else
18:                trim ← trim ∪ level.funcs
19:    return trim
```

Implementation of Pruning. In this mode we instead iterate the $\bot$ functions: starting from each function $f_i \in \bot, \forall i = 0, 1, \ldots, |\bot|$ we generate consecutive sets $S_i^0, S_i^1, \ldots, S_i^k$ such that $S_i^0 = \{f_i\}$ and $S_i^j = S_i^{j-1} \cup \{g \mid g \in V_{CG} \wedge \exists h : h \in S_i^{j-1} \wedge (g, h) \in E_{CG}\}, \forall 1 \leq j \leq k = chain\_length$. Informally, we construct sets of functions that can reach any $\bot$ function in upmost `chain_length` unique calls.

Note, that we further employ both an upper ($\sqcup = f_i.level + chain\_length/2$) and lower ($\sqcap = f_i.level - chain\_length/2$) limit for the caller level (i.e., $\sqcap \leq g_i.level \leq \sqcup : \forall g_i \in S_i^k - \{f_i\}$) to focus the pruning effect, which proved necessary for CGs with numerous loops and large *strong components* — otherwise, pruning originating from a single bottom-level function f_i has the potential to remove a disproportionate number of functions even from a distant CG sections (as experimentally tested on the `CPython` project). Moreover, we use the `keep_top` parameter to prevent the pruning algorithm from removing functions in the specified number of top call graph levels (since without such restriction, we could potentially prune even the `main` function[15]). The resulting set of pruned functions thus contains all functions from all paths up to the `chain_length` edges that terminate in some $\bot$ function and also satisfy the *upper*, *lower* and `keep_top` level constraints.

Algorithm 4 Call Graph Pruning

Input: Call Graph *cg* and the remaining parameters are defined in the introductory Table.
Output: The *prune* set specifies functions that are excluded from profiling.

```
1: procedure PRUNE(cg, chain_length, keep_top)
2:     prune ← ∅
```

[15] We want to keep some of the topmost functions to ensure we can cover the whole profiling with some precision. Moreover, these are usually not called as often and hence they induce only a minimal overhead.

3: ▷ Compute the *bottom-level* functions using the subsumption relation.
4: $\bot \leftarrow \{f \mid f \in cg.funcs \wedge \neg\exists f' \in cg.funcs : f' \neq f \wedge f \subseteq f'\}$
5: **for all** *bot* **in** $\bot$ **do**
6: **if** $bot.level < keep\_top$ **then**
7: **continue**
8: ▷ Compute the minimum and maximum level boundaries for the callers.
9: $max\_level, min\_level \leftarrow bot.level + chain\_length/2, bot.level - chain\_length/2$
10: $prune.add(bot)$
11: $inspect\_list \leftarrow [bot]$
12: $leaf\_candidates \leftarrow \{bot\}$
13: ▷ Iterate all the (in)direct steps in the call chain.
14: **for** $step \leftarrow 0$ to $chain\_length - 1$ **do**
15: $step\_callers \leftarrow \emptyset$
16: ▷ Find all callers that satisfy the constraints.
17: **for all** $func$ **in** $inspect\_list$ **do**
18: $callers \leftarrow \{caller \mid caller \in func.callers \wedge caller \notin leaf\_candidates \wedge$
$\wedge\, min\_level \leq caller.level \leq max\_level \wedge$
$\wedge\, caller.level \geq keep\_top\}$
19: $step\_callers \leftarrow step\_callers \cup callers$
20: $prune \leftarrow prune \cup step\_callers$
21: $leaf\_candidates \leftarrow prune \cup step\_callers$
22: $inspect\_list \leftarrow step\_callers$
23: **return** *prune*

Implementation of Matching. Finally, we also implement one additional mode: a simple `Matching` which does not modify the CG per se, but instead leverages it to filter out unreachable functions (in the set of functions extracted from the collection strategies, see Section 4.2) that would be otherwise needlessly instrumented. The Matching is, naturally, by default applied even during the Trimming or Pruning modes, however, while they employ additional filtering, the Matching keeps the CG structure intact. In our experience, even this simple restriction yields a considerable reduction of probes because lot of function symbols (obtained from the symbol table) are external to the SUT or simply unreachable. Consequently, a significant speed-up is achieved since the probe injection (i.e., instrumentation) is a time-expensive operation which scales noticeably with the number of probes.

Method Assessment. The Call Graph Shaping method represents a rather different optimization approach compared to the Static or Dynamic Baseline, which exploit the statistics and properties of functions. Instead of focusing on the complexities and filtering out functions according to their (expected) performance behaviour, we predict which functions are more likely to introduce significant overhead based solely on the structure of the CG. Thanks to the various modes (trimming and pruning) and pre-defined parameters (strict or soft), the method is considerably versatile in providing different degrees of optimization (from slight to rather harsh ones) or restricting the scope of profiling to certain levels even without the need for extensive manual configuration or combination with other methods.

+ Versatile, fast and effective technique; can be utilized as a standalone method.

- Manual configuration can be somewhat difficult due to the number of options.

+ Broad range of effectiveness based on its configuration (modes, parameters).	- Heavily relies on the CG *levels* estimation; significantly inaccurate estimate lead to inferior optimization results.

Future Work. We believe that the Call Graph Shaping family is currently in its final state. However, future efforts could focus on designing additional pre-defined configurations or fine-tuning the parameters, the default values and the parameter prediction process.

7.2.4 Diff Tracing

The *Diff Tracing* method is inspired by some of the recent applications of analysers (such as FBInfer [18]) in CI of large code-bases. In particular, these approaches optimize the analysis process for regular, day-to-day incremental updates to the project so that the developers can promptly analyze the found issues of (only) the freshly introduced changes. We propose we can achieve such precisely targeted optimization by leveraging the CG and CFG resources, as well as exploiting the integration of VCS within the Perun that grants us access to the project history and changes associated with specific project versions.

Parameters:	`keep_leaf`	The flag whether the changed leaf functions should also be profiled; in default we omit them to minimize the overhead.
	`inspect_all`	The flag that turns on a deep analysis of changes, which analyzes the whole CG.
	`cfg_mode`	Selected equivalence criterion (`Soft`, `Semistrict`, `Strict`) to determine what is to be considered a change.
Resources:	`CGR`	(Required) To identify CGR changes in new project version.
	`CGR (old)`	(Required) To comparewith `CGR` (for some `HEAD~i` version).
Output:	`CGR`	(Modified) With flagged changed functions so they will not be removed by other optimizations (if used in combination with other methods) or removed functions that have not changed (if used as a standalone method).
Phase:	`pre-optimize`	
Notable Trade-off:	`time-overview`	Speedup at the cost of almost no top-level overview.
Application Scope:	`large-scale`	
Primary OC:	`OC_F`	
Time Overhead:	`minimal`	

Motivation and Overview. When Perun is used to continuously monitor the performance of a project under development, it is usually not necessary to profile vast majority of the functions within the program after each project update (e.g., new *git commit*). Many of these updates usually have only a narrow impact in the code, and sometimes contain only non-semantic changes (such as comments, variable renames, etc.). So instead, precisely pinpointing and profiling only functions directly (and, ideally, semantically) affected by the latest changes is usually preferred by the developers, so that the performance may be evaluated quickly. Moreover, the more fresh are the found changes, the higher is the chance that developer is still in the context of the change and can fix the found error more quickly; we argue, that the bug fix ratio is more important metric than bug finding ratio.

Hence we propose the Diff Tracing: a lightweight method that utilizes multiple indicators to heuristically identify changes — (1) in the call graph structure, (2) in the function call order, or (3) in the function source code — that can lead to a different assembly (e.g., ignoring insignificant changes such as renaming variables, functions, comments, etc.). Specifically, we first compare the project's new CG with the previous one and detect new, removed,

renamed or moved functions. Next, we (optionally) parse the source code differences between the two versions using the project VCS (using *git diff*) and obtain intra-function changes that cannot be analysed from the call graph perspective only. Finally, we employ the CFG (as depicted in Figure 7.2) and check if these intra-function changes generated different assembly code (based on the selected *equivalence criterion*), thus filtering changes that have no impact on the behaviour or properties of the function. Note that this is an *unsound* heuristic — as automated and sound analysis of semantic machine code change is *not a trivial* task — and hence we restricted ourselves to pinpointing machine code changes, without further introspection of their nature or impact.

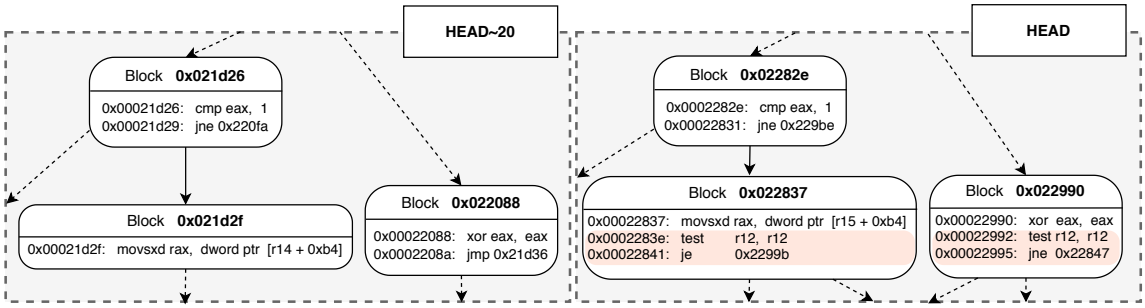

Figure 7.2: A reconstruction of CFG segment found in specific `CCSDS` (see Section 8.1 for more details) function across two different project versions, specifically `HEAD` and `HEAD~20`. As can be seen, both the CFG structure and *basic blocks* have changed since the bottom vertices have more edges and `ASM` instructions.

The Diff Tracing method has two different modes: one for standalone usage and one for its combination in *pipelines* (see Section 6.2.1). In the standalone mode, only the selected functions (and `main`) are profiled; in the combined mode, the functions are marked to be *protected* from removal by other optimizations ensuring they are always profiled.

Implementation. The Diff Tracing methods is, similarly to the Dynamic Baseline, implemented in a generic way that handles a set of comparison functions, and thus can be easily extended. In its current state, the method is supplied with three comparison functions (designed to identify changed functions): `Call Graph Comparison`, `VCS Difference` and `CFG Comparison`. The comparison functions are pipelined (not in the sense of *optimization pipelines* introduced in Section 6.2.1), so that each subsequent comparison function refines the output of the previous one. In the following, we will denote the VCS `HEAD` project version as the **current** version and the `HEAD~i` (i.e., the i-th previous version) as the **baseline** version. Furthermore, we classify the detected changes as: (1) **True Positive (TP)** if the function is classified as changed, and it indeed contains a change; (2) **False Positive (FP)** if the function is classified as changed, while, in fact, it has not changed; (3) **True Negative (TN)** if the function is classified as not changed, and it, truly, contains no change; and (4) **False Negative (FN)** if the function is classified as not changed, while the function has actually changed.

Generally, we aim for zero `FN` since excluding changed functions from profiling may prevent us from detecting newly introduced performance bug (although it might still manifest in certain higher-level functions). The `FP` cases are much less severe, although still resulting in (slightly) more time and memory overhead during the profiling.

The supported comparison functions are implemented as follows:

1. **Call Graph Comparison** expects a CG of the *current* (`cg_new`) and the *baseline* project version (`cg_old`), both loaded from the `stats` directory. Based on the difference of CG nodes only, we identify sets of `new` ($\{f \mid f \in cg\_new \wedge f \notin cg\_old\}$) and `removed` ($\{f \mid f \in cg\_old \wedge f \notin cg\_new\}$) functions. From these sets, we can try to estimate which functions were only renamed. We propose to compare the `callers` and `callees` of each new_i function with the `callers` and `callees` of each $removed_j$ function to find matching *new* and *removed* pairs — naturally we take other potential renames into account as well (by sorting the node comparison according to their level). Note that renamed functions are still checked for changes in the CFG comparison, as we merely leverage the rename computation to map *current* CG nodes missing in the *baseline* and vice versa. Furthermore, if the `inspect_all` is set, both the `cg_new` and `cg_old` CG are fully traversed (i.e., vertex by vertex) and the `caller` and `callee` sets of corresponding nodes are compared in order to find `caller-callee` differences, which would indicate that the `caller` function has been changed.

2. **VCS Difference** algorithm takes the set of functions from the *current* CG (`funcs`) and unique identification, such as *git sha*, of both compared versions (`current_sha`, `baseline_sha`). Using the Perun VCS API we obtain the version difference in a VCS-specific format — e.g., Listings 7.3 shows an example of the *git diff* output format. We then attempt to parse the version difference output in order to retrieve valid names (found in the `funcs`) of the changed functions regardless of the nature of the changes, i.e., whether it affects the functions behaviour or not. Functions identified as changed according to this approach are then further inspected in the CFG Comparison step. As of now, the algorithm implements only Git-specific *diff-parsing*, however, the algorithm can be easily extended to support multitude of other frequently used VCS.

Listing 7.3: An example of the git diff command output listing changes in the `sock_initobj` function source code between two versions of the `CPython` project.

```
@@ -5091,9 +5091,8 @@ sock_initobj(PyObject *self, PyObject *args, PyObject *kwds)
     }
     memcpy(&info, PyBytes_AS_STRING(fdobj), sizeof(info));

-    if (PySys_Audit("socket.__new__", "Oiii", s,
-                    info.iAddressFamily, info.iSocketType,
-                    info.iProtocol) < 0) {
+    if (PySys_Audit("socket()", "iii", info.iAddressFamily,
+                    info.iSocketType, info.iProtocol) < 0) {
         return -1;
     }
```

3. **CFG Comparison** analyzes functions (`funcs`) from the previous step excluding new functions and functions already marked as changed from the Call Graph Comparison algorithm. For each such function, we retrieve its corresponding CFGs of the *current* (`cfg_new`) and the *baseline* version (`cfg_old`). Next, we perform a quick check to verify that the structure of the `cfg_new` and `cfg_old` is the same, without inspecting the content of the nodes (the set of `ASM` instructions). If the graph structure matches, we then compare the assembly of each matching node pair, i.e., $(V_i^{cfg\_new}, V_i^{cfg\_old})$, using the selected equivalence criterion (`cfg_mode`). If the structure does not match,

the function is classified as changed. Assume nodes n_1, n_2, and their respective sets of instruction $\mathcal{I}^1$, $\mathcal{I}^2$; we will use $\mathcal{I}^i_j$ to denote set of instruction of type j corresponding to node n_i. Currently, we propose the following three criteria:

- **Soft** criterion: nodes n_1 and n_2 are considered to be equal when they contain the same number of assembly instructions, i.e.
$$n_1 = n_2 \Leftrightarrow \sum \mathcal{I}^1 = \sum \mathcal{I}^2$$
While, this is rough and imprecise criterion, it still may be useful in cases where the compiler reorders certain instructions (e.g., for optimization purposes) without altering the behaviour in any way.
- **Semi-strict** criterion: n_1, n_2 are considered to be equal when they contain the same number of assembly instructions of the same type, i.e.
$$n_1 = n_2 \Leftrightarrow \bigwedge_{k=0}^{l} \sum \mathcal{I}^1_k = \sum \mathcal{I}^2_k$$
Note, that we stil exclude the actual operands of the instructions. In our experience, the Semi-strict criterion offers a balanced `TP` and `FP` rate, since some of the direct memory addressing operands may have been changed due to other changes within the binary (e.g., by modifying the size of other memory segments) — hence not altering the function behaviour despite different operands.
- **Strict** criterion for two CFG nodes n_1, n_2 is defined as follows:
$$n_1 = n_2 \Leftrightarrow \bigwedge_{k=0}^{l} \mathcal{I}^1_k = \mathcal{I}^2_k$$
The Strict criterion, requires the instruction operands to match as well — with one exception being the (un)conditional jump instructions. Since the destination address of a jump instruction is likely to change while not altering the function behaviour (as described in the Semi-strict criterion), and the destination can be precisely obtained by the corresponding CFG edge, we decided to ignore the destination operand of the jump instructions. This criterion boosts the `TP` ratio (compared to the Semi-strict criterion) while also increasing the number of `FN`. However, on the other hand, due to such thorough comparison, low amount of `FP` are expected — mainly those caused by code changes external to the function itself (e.g., changing a global constant variable or macro definition). Thus, the Strict criterion is recommended when we strive to achieve low `FP` rates and the increased number of `FN` is unimportant.

At the end, the Diff Tracing algorithm flags the corresponding functions from sets `new` and `modified` as changed to ensure that they will be profiled. Note, that all of these detections could be realized by more advanced and much more precise, sound methods (such as, e.g., the DiffKemp project [41]), however, we believe that our lightweight heuristics can still lead to suitable results for our proposes with minimal performance overhead.

Method Assessment. The Diff Tracing is the only method that targets the freshness criterion and thus can ensure precisely directed — but narrow — performance profiling for projects that employ some form of a VCS. To achieve better top-level performance overview (compared to the standalone usage), combining the Diff Tracing with other optimization methods is strongly advised. Moreover, thanks to the multiple implemented light-weight equivalence criteria, we can achieve various degrees of precision for the change detection.

+ Fast and precisely targeted profiling for projects under active development.
+ Can achieve different true / false positive and negative rates thanks to the variety of equivalence criteria.
+ Supports a standalone and combined usage with slightly different behaviour.
- Does not provide sufficient top-level performance overview of the whole program in the standalone mode.
- Performance changes unrelated to the function source code changes may not be detected (e.g., different value of global variable, changed macros, etc.).

Future Work. The semantic comparison of function is still an area under wide research, where most approaches are not yet applicable in practice. Our method is based on a lightweight principles and several fast heuristics, with much room for improvements. First, the CFG comparison could be extended with more advanced detections, where only modifications severely impacting the performance would be considered (e.g. as some form of instruction pattern matching could be utilized). Next, the Strict equivalence criterion could be extended to semantically analyze the instructions and operands to detect changes in direct memory addressing values which, however, do not alter the behaviour. Finally, we could employ some more advanced (external) and precise analyser, such as `diffkemp` [41].

7.2.5 Dynamic Sampling

All of our previous methods were based mostly on the analysis of the code or the structure, followed by filtering of profiled functions. Now, we move into a different area, where instead we control how often the probes generate data (i.e., the sampling) or how often they are active (i.e., the dynamic probe switching). We first propose *Dynamic Sampling*: an iterative method that utilizes both the *CGR* and *Dynamic Stats* to optimize the data volume (`OC_DV`) by automatically estimating the appropriate sampling for each function, so that adequate amount of records is collected. That is, we want to prevent functions from over-generating millions of records, yet keep sufficient amount of data records for any further and analyses.

Parameters:	`step`	The default estimation of the function sampling based on the CG level fl of the function f ($step^{fl}$).
	`threshold`	The desired amount of performance records per each function.
Resources:	`CGR`	(Required) To access the level and *changed* status of functions.
	`Dynamic Stats`	(Optional) To access the number of collected records.
Output:	`CGR`	(Modified) Updated the sampling of profiled functions.
Phase:	`pre-optimize`	Estimation of the sampling for each function.
	`post-optimize`	Gathering of the Dynamic Stats after the profiling is done.
Application Scope:	`large-scale`	
Primary OC:	`OC_DV`	
Time Overhead:	`minimal`	

Motivation and Overview. Generally, the Perun collectors support some form of data sampling, i.e. they can store only every n-th raw performance record during the profiling (we can list e.g. the *specific* (manually selected probes) and *global* (all of the probes) sampling values implemented in the Tracer). However, so far neither of the specific sampling configurations were particularly practical or effective — it is either too tedious to manually configure the sampling for all the relevant functions, or too global, so, suitable only for a certain subset of functions (while the others generate too few or too many records).

Hence, the Dynamic Sampling solves this issue through automated, adaptive sampling estimation for each function, individually. Specifically, each function is first assigned an initial sampling estimate based on its *level* and the Assumption 7.1.1 (unless this initial sampling is not already available, e.g., from previous profiling). In every subsequent method iteration, we then further refine this estimate based on the Dynamic Stats, so that the number of collected records for each function reaches the user-specified `threshold`, i.e., we increase the sampling value for functions that are above the `threshold` and vice versa.

Implementation. The Dynamic Sampling algorithm is implemented in two phases: the `initial` and `refinement`. Note that functions may pass different phases. For example, new functions selected by the Diff Tracing will be assigned an *initial* sampling estimate, while the previously profiled (and sampled) functions will have their sampling *refined* only.

1. The `initial` phase is activated when the Dynamic Stats (`stats`) are either not available (for the same profiling configuration) or does not contain statistics for the given function, i.e., we have no information about the actual function call count. So, we leverage the *level* property of the function and the `step` parameter to estimate the sampling as $sample_f = step^{f.level}$. The intuition is that w.r.t to the Assumption 7.1.1 functions with higher level estimates (where *root* is level 0) are assumed to have higher number of total function calls. Furthermore, we also exploit the inferred theoretical complexity to further multiplicate this estimate using a complexity-specific `coefficients`. Currently, we set the coefficients to $constant = 2$ and $linear = 1.5$ since functions with constant or linear complexity are, as per our previous experience, more likely to have higher call count. However, these coefficients have not been thoroughly tested and as such, we plan to further fine-tune them in the future.

2. The `refinement` phase is activated when the given function has the corresponding statistics in the `stats` and thus, we can leverage them to adjust the sampling accordingly. Specifically, we utilize the `sampled_count` and `sample` records which express the number of recorded function calls with the previous sampling configuration. The refinement process consists of modifying the function sampling value according to the formula $sample_{new} = round(sample_{old}/(threshold/sampled\_count))$ so that the expected number of recorded calls is in close proximity of the `threshold` value, i.e., within a small ϵ deviation ($sampled\_count \approx threshold \pm \epsilon$) since exact match is highly unlikely. For example, if $threshold = 10, \epsilon = 2, sample\_old = 100$ and $sampled\_count = 20$, then $sample_{new} = round(100/(10/20)) = 200$ which should result in $sampled\_count = 10 \pm 2$ during the next profiling.

Finally, the resulting sampling is further *normalized* to fit between the minimum and maximum sampling range (i.e., $1 \leq sample_{new} \leq max\_sample$). Such normalization is necessary, since the languages used to construct the collection programs usually employ fixed-size types (such as `uint32` or `uint64`). The sampling estimate then is assigned to the profiled function within the CGR and the sampling itself is propagated to the collection program assembling. The Algorithm 5 illustrates the whole Dynamic Sampling pseudocode.

Algorithm 5 Dynamic Sampling

Input: Call Graph *cg*, Dynamic Stats *stats* and user-specified *threshold* parameter.
Output: Each profiled function has its *sample* property assigned.
1: **procedure** SET_SAMPLING(*cg*, *stats*, *threshold*)

```
    eps ← threshold * threshold_eps_ratio
    for all depth, level in enumerate(cg.levels) do
        for all func in level do
            if func not in stats then
                ▷ The initial phase.
                sample ← round(step^depth)
                sample ← sample * complexity_ratio(func.complexity)
            else
                ▷ The refinement phase.
                calls ← stats[func].sampled_count
                sample ← stats[func].sample
                if not (threshold − eps ≤ calls ≤ threshold + eps) then
                    ▷ Change the sampling so the calls are closer to the threshold.
                    sample ← round(sample/(threshold/calls))
            sample ← normalize(sample, min_sample, max_sample)
            func.sample ← sample
```

Method Assessment. The Dynamic Sampling is a simple and efficient method for reducing the amount of collected raw data. It extends the existing — albeit currently inefficient, crude and impractical — technique of data volume reduction with an automated parameter estimator, and thus making it easy to use without the need for extensive manual configuration. However, often the algorithm might need a few iterations to sufficiently refine the sampling, especially if the initial estimate is significantly inaccurate. Moreover, although the sampling can greatly reduce the amount of generated data, the effect on profiling time is usually negligible. The Tracer still has to inject all the probes, and each function call still triggers the corresponding probe — the only difference is that in most cases, it does not generate any data. Also, note that the sampling impacts the precision (quantifiable using the R^2) of the resulting complexity models, as fewer data are available.

+ Simple method that naturally extends the already supported sampling.	- Only negligible impact on the total profiling time.
	- The initial sampling estimation may be considerably inaccurate, requiring more refinement phases.
+ Significant reduction of data volume even without filtering collection points.	- Slight to significant impact on the precision of the performance models.

Future Work. In order to further enhance the Dynamic Sampling method, we propose to focus the efforts towards improving the *initial* phase of the algorithm. Specifically, we believe we could experiment with more advanced mathematical functions than the currently leveraged exponential progression ($step^{level}$) which could yield more fitting initial sampling distribution, especially for large-scale projects with up to hundreds CG levels. Also, performing more thorough study with multiple test projects could lead to refining the complexity ratios utilized in the initial phase.

7.2.6 Timed Sampling

The *Timed Sampling* is a *run-optimize* method that, similarly to Dynamic Sampling, tunes the sampling of collection probes to reduce the amount of generated data. However, compared to the *sampling of call count*, when we store only every n-th generated performance record, the Timed Sampling approach instead samples the data in the time domain, i.e. we generate records only in the specified periodic intervals.

Parameters:	`frequency [Hz]`	The frequency of the probe deactivation (resp. reactivation).
Resources:	`None`	
Output:	`None`	
Limitations:	`eBPF`	Only eBPF supports the Timed Sampling method.
Phase:	`run-optimize`	
Application Scope:	`medium-scale`	Due to the eBPF limit of simultaneously attached probes.
Primary OC:	`OC_DV`	
Overhead:	`minimal`	

Motivation and Overview. In the Dynamic Sampling (see Section 7.2.5), we have already stressed the motivation for adapting sampling techniques, their advantages and disadvantages. However, although the *sampling of the call count* is already effective in reducing the amount of resulting performance data, we suggest there may be an alternative approach that can yield a different distribution of the sampled performance data.

The proposed Timed Sampling is rather straightforward: based on the user-supplied `frequency` parameter, we alternate between *gather* and *dormant* profiling phases, where the *gather* phase generates performance records and the *dormant* phase prevents the probes from generating any data. Comparison of both approaches is illustrated on Figure 7.3.

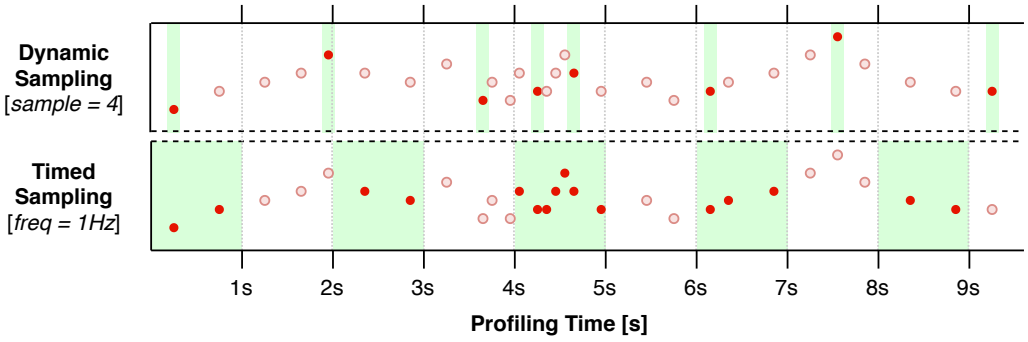

Figure 7.3: An illustration of the different sampling approaches employed by the *dynamic* and *timed* sampling — while Dynamic Sampling records only every n-th probe hit, the Timed Sampling records all probe hits in a given *time frame* specified by the frequency. Filled (resp. unfilled) points represent sampled (resp. missed) data.

Implementation. We proposed multiple candidate implementation approaches and examined their properties in terms of time overhead and performance data loss.

1. The initial prototype utilized a timer (in form of a separate `periodic_thread`) that systematically triggered the *gather* and *dormant* phase changes. The phase change would consist of (de)activating all of the injected probes at runtime, thus leveraging the eBPF capability of dynamic probe attaching and detaching. However, regardless

of whether we paused the profiled program (using the `SIGSTOP` signal) or kept it running (as is done in the Dynamic Probing method), the mass (de)activation of all the probes substantially impacted the time overhead and data loss — especially when hundreds of probes were being used.

2. In the second attempt, we decided to incorporate the probe (de)activation mechanism directly into the collection program: we implemented the timer as a standalone `perf_event` probe attached to the *CPU CLOCK software performance counter* (see Section 2.5.5) and set the tick frequency to the user-defined `frequency`. Then, whenever the probe is hit (i.e., the *CPU CLOCK* event is generated), it flips a boolean flag (in the collection program) that signalizes the currently active phase. Every probe handler (during the collection program assembly) then fetches the flag and, based on the phase, either generates a performance record or terminates the probe handler.

Compared to the first implementation, the second approach showed only a negligible performance data loss and time overhead (caused by the method itself). Hence, we decided to leverage the second implementation for the final version of the Timed Sampling.

Method Assessment. The Timed Sampling method utilizes a sampling technique and acts as a complementary method to the Dynamic Sampling. However, although both approaches target the same data volume optimization criterion (**OC_DV**), they differ in the distribution of the sampled performance data. Specifically, the Dynamic Sampling method samples the records uniformly for each function across the whole profiling run, whereas the Timed Sampling method samples all of the functions into the given periodic time windows.

+ Straightforward method that does not need any optimization resource or an extensive user configuration.
+ An alternative to the Dynamic Sampling method that noticeably reduces the amount of data and generates a different output distribution.
- Reduces the profiling time only negligibly since the probes are still being triggered even in the *dormant* phase.
- Currently limited to the eBPF engine.
- Skews the resulting complexity models.

Future Work. The current implementation does not rely on any advanced runtime probe manipulation, thus an additional effort should be made to decouple the Timed Sampling implementation from the eBPF engine and generalize it to other engines and collectors.

7.2.7 Dynamic Probing

The *Dynamic Probing* is a method utilized in *run-optimize* phase and designed to continuously monitor the call count of each function during the profiling, and subsequently disable those collection points that reach a certain call count threshold. Note that deactivating an attached probe at runtime is rather costly operation which can in some cases — according to our experiments — cause a minor loss of raw performance data, possibly due to unhandled probe triggers during the deactivation process. This method requires no optimization resources at all, however, it is usable only if the underlying instrumentation framework supports the runtime (de)activation of the instrumented collection points (such as the eBPF).

Parameters:	`threshold`	The number of function calls that triggers probe deactivation.
	`reattach`	The flag for enabling the `reattach` mode that periodically attempts to reactivate the disabled probes.
Resources:	`None`	
Output:	`Probe IDs`	Periodical identification of probes to deactivate or reactivate.
Limitations:	`eBPF`	Only frameworks that support runtime (de)activation of probes.
Phase:	`run-optimize`	
Application Scope:	`medium-scale`	Due to current eBPF limit of simultaneously attached probes.
Primary OC:	`OC_T`	
Time Overhead:	`noticeable`	Due to costly periodic probe (de)activation.

Motivation and Overview. Although the *pre-optimize* methods are generally versatile and efficient, there are situations when the optimizations either cannot be utilized (e.g., when the CGR may not be effectively obtained due to unsupported architecture type), or the result is unsatisfactory (e.g., the heuristics employed in various optimization methods fail). Hence, we propose a method that does not rely on any extracted resource. We believe, such solution could provide a sensible alternative to the *pre-optimize* methods.

The Dynamic Probing technique exploits the capability of instrumentation frameworks (such as the eBPF) to dynamically enable/disable probes at the runtime. Generally, we can distinguish two variants: the `detach` and `re-attach`. The former, systematically counts the number of probe hits and when it detects that the user-specified threshold has been reached by some function, the corresponding probe is deactivated, thus generating no more data and causing no further overhead. The latter works similarly, however, each deactivated probe is also reactivated after a certain time interval. We propose to double this reactivation interval every time the probe is deactivated again, so probes that generate significant amount of data during the whole profiling stay deactivated for most of the time, while some may still resume measuring after a brief deactivation (e.g., during a performance hotspot).

Implementation. Since the Dynamic Probing method runs *during* the profiling process, we experimented with multiple candidate implementations in order to minimize its impact on the profiling (e.g., noticeable increase of lost data records or slowdown of the profiling).

1. The first approach was eager: we check the amount of function calls for the given function every time its corresponding probe is hit, and subsequently deactive it if the `threshold` has been reached. However, the function used to store the generated raw performance data is being invoked so often (up to millions of calls per second), this caused a noticeable spike in the raw data loss[16].

2. The second attempt was based on a dedicated thread (`probing_thread`) that would periodically switch between a *check* and *sleep* phases. The thread performs no operations during the *sleep* phase, whereas during the *check* phase, the thread inspects probe hit counters for all the functions at once — while the profiled program itself is paused by the `SIGSTOP` signal (and later resumed using the `SIGCONT` signal). Although this way the observed data loss has been slightly smaller, the duration of the program has still notably increased. Moreover, it also led to a drawback that the probes were not disabled immediately after reaching the `threshold`, but rather during the next *check* phase since the *threshold* may be reached during the `probing_thread` sleep.

[16] Which, to the best of our knowledge, happens either (a) due to unhandled probe triggers since the eBPF VM performs other operations, or (b) the internal kernel buffer is full, thus additional records are discarded.

3. In the final implementation, we used an enhanced version of the dedicated thread. First, we incorporated the Dynamic Probing code directly into the Tracer (specifically, into the eBPF engine) to remove the additional layers of indirect function calls caused by calling functions from other modules. Next, we decided not to utilize any kind of program pausing — since according to our experiments, it causes more data loss than simply letting the program run while the `probing_thread` is active. Finally, we experimentally fine-tuned the duration of the thread *sleep* phase to balance the amount of data loss, time overhead and probe deactivation delay. As a result, the final implementation showed acceptable data loss rate and time overhead.

The `re-attach` is implemented on top of the *detach* mode: we extended the thread to keep track of the deactivated probes as well as the time remaining to their reactivation — each reactivation of the given probe location then doubles the future deactivation time. Figure 7.4 depicts the `probing_thread` dedicated to inspecting the *probe hit counters* and subsequently (de)activating any probe that hits the `threshold` number of calls. Moreover, it also shows the interaction between the `probing_thread` and the eBPF `profiling_thread` responsible for running the SUT and collecting raw performance data (see Section 6.1.1).

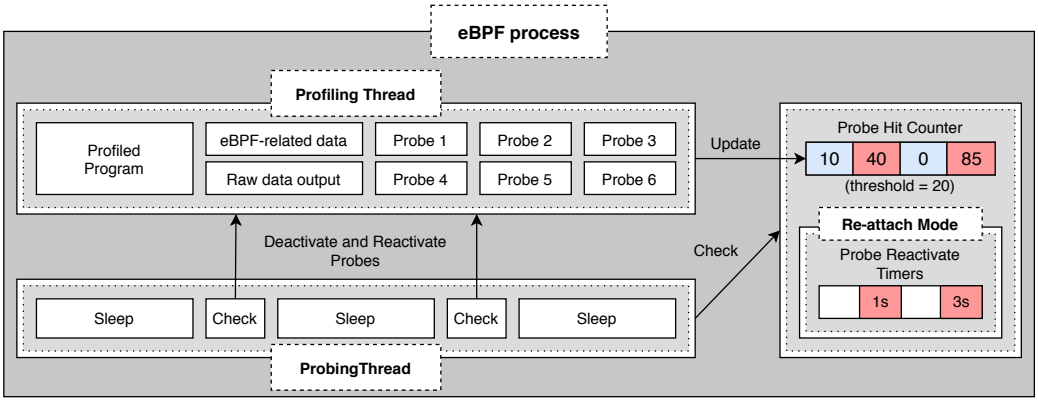

Figure 7.4: An illustration of the *probing* and *profiling* threads and shared dynamic structures. The `profiling_thread` performs the standard eBPF profiling operations such as running the SUT, attaching/detaching probes and storing the raw performance data. The `probing_thread` (active only when the Dynamic Probing is enabled), on the other hand, periodically switches between the *sleep* and *check* phases, where it checks the probe hit counts and deactivates (based on the `threshold`) or periodically reactivates probes.

Method Assessment. The Dynamic Probing is an attempt to design an alternative to the *pre-optimize* methods that require various optimization resources which might not always be obtainable. However, while it can precisely target the functions that generate the most performance data and cause the most overhead (without needing the Dynamic Stats as is the case with other precise methods), it is tightly linked to the underlying Tracer's engines. Specifically, it is currently only in the eBPF Tracer engine as a proof of concept, since the SystemTap does not support such dynamic runtime probe (de)activation.

+ No resources needed, making it a viable alternative to *pre-optimize* methods.
+ Precisely filters out the functions that cause significant profiling overhead.
- Noticeable inherent overhead compared to other methods.
- Currently implemented only as a proof of concept in the eBPF engine.

Future Work. The Dynamic Probing could be improved in two ways. First, additional approach to implementation could be examined to further reduce the average performance data loss and inherent time overhead. Second, the resulting implementation should not be tightly coupled with the eBPF engine, but rather be a module of the optimization architecture (see Section 6.2) so that other future engines or collectors can utilize it as well.

7.3 Proposed Optimization Pipelines

In order to optimize a profiling of a SUT, one usually does not employ individual optimizations by themselves and instead wants to maximize the gain. Hence, we propose the so called *Optimization Pipelines*: predefined sequences of configured optimizations. The pipelines are intended as a quick solutions that require minimal to none additional manual configuration, offer various degrees of precision and focus on different optimization criteria.

In total, we propose three pipelines within the Optimization architecture. Each pipeline was designed specifically to offer a different optimization precision and subsequently combines several of the optimization techniques using less strict modes and parameters.

7.3.1 Basic

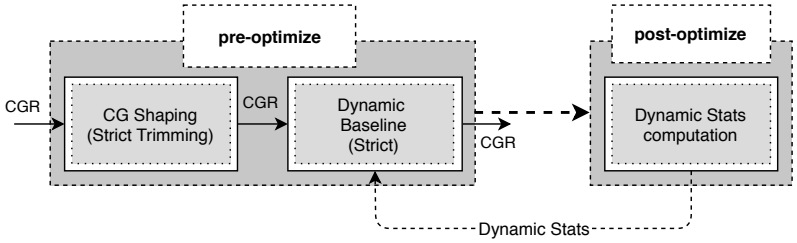

Figure 7.5: A schematic overview of the *Basic* pipeline that utilizes the Call Graph Shaping (in strict trimming mode) and Dynamic Baseline (in strict mode). The methods leverage both the Call Graph Resource (*CGR*) and Dynamic Stats, however, since the Dynamic Stats are computed after the profiling, it can be utilized only in the next profiling.

The `Basic` pipeline combines the Call Graph Shaping with the Dynamic Baseline: a simple but crude optimization approach. While Call Graph Shaping initially estimates and filters potentially performance-intensive functions based on the structure of SUT (and thus providing a significant speed-up of the first profiling run), Dynamic Baseline can further refine this estimate in the subsequent runs and filter additional performance-intensive functions. Hence, the Call Graph Shaping and Dynamic Baseline methods have an exceptional synergy in minimizing the profiling time overhead (starting from the initial profiling run) at the cost of diminished granularity.

We set the Call Graph Shaping to the *strict trimming*, i.e. we try to trim the CG levels more aggressively, thus keeping less functions (specifically, the *strict* and *soft* modes keep the top 25% and 50% of the CG levels, respectively). Moreover, we limit the `threshold` parameter of the Dynamic Baseline to *strict* mode as well, to filter out more functions. Since no additional optimization techniques are employed, we argue that using the *strict* modes lowers the chance of any remaining function causing a massive time or memory overhead (notably in the first profiling run), as the optimization is more aggressive and less — potentially performance-intensive — functions are kept.

Generally, the Basic pipeline is useful for an initial or overview profiling where only a small subset of all the functions is profiled (namely the top-level functions closest to the `main` function). This way, any possible performance changes will be detected and user will obtain its rough location. Figure 7.5 schematically depicts the `Basic` pipeline.

Call Graph Shaping:	`Strict Trimming`	The trimming is limited to the top 25% CG levels.
Dynamic Baseline:	`Strict`	The `soft_thr` and `hard_thr` thresholds have lower values in order to filter more functions.

+ Significant speedup and data volume reduction while still keeping sufficient top-level overview of the performance.
+ Requires only the CGR and Dynamic Stats, otherwise no restriction as to the used engine or collector.
- Filters out a large amount of functions, thus resulting in inferior profiling precision compared to other pipelines.

7.3.2 Advanced

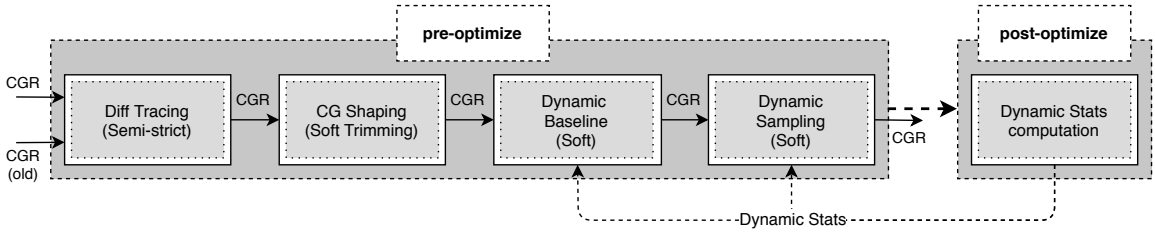

Figure 7.6: A schematic illustration of the `Advanced` pipeline. Compared to the `Basic` pipeline, both the Dynamic Baseline and Dynamic Sampling methods trigger the Dynamic Stats computation in the *post-optimize* phase. Moreover, apart from the CGR, the Diff Tracing method also leverages the CGR of the previously profiled project version.

The `Advanced` pipeline (Fig. 7.6) combines the Diff Tracing, Call Graph Shaping, Dynamic Baseline and Dynamic Sampling methods: a balanced combination in terms of profiling time, data volume and profiling precision. Compared to the `Basic` pipeline, the Diff Tracing method ensures that any previously filtered function f will be profiled as soon as we detect that f has been changed in the latest project version. Since some profiled functions might generate a disproportionately large volumes of raw performance data — as, e.g., linear functions with call count below `hard_thr` will not be filtered by the Dynamic Baseline — we further employ the sampling to control the amount of resulting raw data.

We run most of the methods in the `soft` mode in order to not filter out as many functions. We argue that this *softness* is then mitigated by other methods in the pipeline. Moreover, included Diff Tracing assures that the `Advanced` pipeline can be easily used to optimize continuous profiling of projects under active development.

Diff Tracing:	`Semi-strict`	Two CFG nodes are considered equal if their corresponding `ASM` instructions (excluding operands) are the same.
Call Graph Shaping:	`Soft Trimming`	The trimming is limited to the top 50% CG levels.
Dynamic Baseline:	`Soft`	Sets the `soft_thr` and `hard_thr` thresholds have higher values (compared to the `strict` mode) to filter less functions.
Dynamic Sampling:	`Soft`	The `threshold` has a higher value so less calls are sampled.

+ Balanced profiling time, data volume and profiling precision due to variety of methods and the usage of `soft` modes.

+ Can reliably profile changes across different project versions.

- May not precisely localize subtle performance changes caused by functions within the lower 50% levels of the CG.

7.3.3 Full

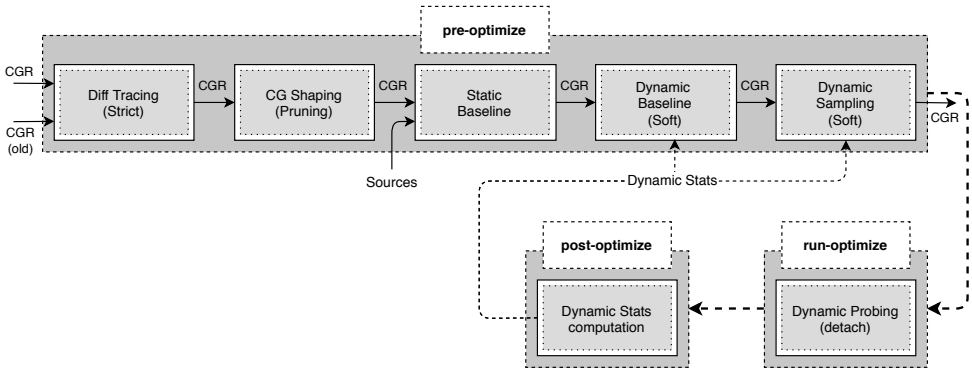

Figure 7.7: A schematic of the `Full` pipeline. Compared to the previous pipelines, the `Full` pipeline also utilizes the *run-optimize* phase by including the Dynamic Probing.

The `Full` pipeline (Figure 7.7) utilizes almost all of the currently implemented optimization methods: the Diff Tracing, Call Graph Shaping, Static Baseline, Dynamic Baseline, Dynamic Sampling and Dynamic Probing. Note that we excluded the Timed Sampling from the pipeline since the combination of Dynamic and Timed Sampling — although possible — may result in too severe reduction of data volume. Compared to the previous pipelines, the `Full` pipeline should filter out the least functions for profiling, thus generating the most data and guarantee the best precision at the cost of increased profiling time. Note that we achieve such behaviour mainly by selecting the appropriate modes (i.e., the least optimization-aggressive ones) for each optimization technique, thus, contrary to the naïve expectation: *more optimization techniques* $\neq$ *more aggressive optimization*, but rather *more optimization techniques* $=$ *more optimization control*.

Although the pipeline leverages methods that have additional requirements (e.g., the source code) and limitations (e.g., the eBPF Tracer engine), it can still be used even if those constraints are not satisfied by automatically switching the optimizations off.

Diff Tracing:	`Strict`	Two CFG nodes are considered equal if both the `ASM` instructions *and operands* are the same in compared basic blocks (see Sec. 7.2.4).
Call Graph Shaping:	`Pruning`	The CG is pruned in a bottom-up manner.
Dynamic Baseline:	`Soft`	The `soft_thr` and `hard_thr` thresholds have higher values (compared to the `strict` mode) in order to filter less functions.
Dynamic Sampling:	`Soft`	The `threshold` has higher value so that less calls are sampled.
Dynamic Probing:	`Detach`	The once deactivated probes stay disabled.

+ Complex combination of most of the methods.

+ Easier to precisely locate performance bugs, since more functions are profiled.

- Increased profiling time and data volume compared to the other pipelines.

- The full potential of pipelines at the cost of requirements and limitations.

Chapter 8

Experimental Evaluation

Overall, we have proposed seven optimizations and three pipelines. In order to evaluate them properly, we designed a set of experimental configurations covering every method, pipeline and most of their accompanying parameters (so called experimental cases). We then conducted the experiments on both medium and large-scale projects, in particular, the `CCSDS` [1], and `CPython` [11]. On each project we tested our implementation with respect to the *evaluation metrics* introduced in Section 5.1. Based on the metrics' values, we conclude whether the proposed technique truly optimized the target optimization criteria (see Section 5.2) and how significant the optimization is.

8.1 Methodology

We propose the following methodology for evaluating our set of optimizations. For the purpose of the evaluation, we select (1) the so called *experiment cases*, i.e. a set of optimization configurations (e.g., the enabled optimization methods and corresponding parameters), (2) *collector configurations*, i.e. collector-related parameters (such as the collection engine and caching mode), and (3) *test projects*, i.e. the profiled executable files and workload. Each experiment then corresponds to one combination of experiment case, collector configuration and test project.

Naturally, we run the whole set of experiments for each collector configuration multiple times: specifically, each case is repeated six times, where the first measurement is discarded (the warm-up) and the remaining five results are further considered[1]. So, in total, we run for each project $|collector\_configurations| * |experiment\_cases| * 6$ experiments. However, note that not every test project supports all of the collector configurations, nor can every collector (with given configuration) run all of the experiment cases (such as the Dynamic Probing), hence, the number of obtained measurements can vary for each project.

Test projects. We selected two *test projects*. For each project we list its size of the codebase, the workload we use to run the compiled executable, version we use as a baseline for comparison (in Diff Tracing) and limitations that it imposes (if there are any).

[1] We are aware, that increasing the number of warm-ups and repetitions would lead to a better precision of measurement. However, we believe, the proposed numbers are enough.

CCSDS:	`Description`	An implementation of a `CCSDS 122.0` [1] image compression standard that utilizes discrete wavelet transform coder and bitplane encoder for lossless to lossy compression. The implementation (although still under development) was provided to us by David Bařina, Ph.D.
	`Size`	Medium-scale (10000+ `C` LoC, 164 functions).
	`Workload`	A grayscale *Lenna.pgm* image with 512×512 pixels.
	`Diff Version`	`HEAD~20`
	`Compilation`	The executable is compiled with `-O3` optimization level.
CPython:	`Description`	`CPython` [11] is a reference implementation (written in the `C` language) of the Python programming language, its compiler and interpreter. For the purpose of evaluation, we used source-compiled `3.8` version with `debug` symbols.
	`Size`	Large-scale (500000+ `C/C++` LoC, $\approx$ 6400 functions).
	`Workload`	A reference implementation of *quicksort* [2].
	`Diff Version`	`CPython 3.7`
	`Limitations`	`CPython` cannot be profiled by the eBPF engine since the eBPF technology limits the number of probes that can be simultaneously attached (and the number of functions within `CPython` greatly exceeds the imposed limit).

Collector configuration. For the *collector configurations*, we identified two key parameters that have significant impact on the resulting profiling: the *engine* (stap or ebpf) and *cache* (on or off) — resulting in total of four combinations of parameters $\{(stap, cache\_on), (stap, cache\_off), (ebpf, cache\_on), (ebpf, cache\_off)\}$.

Engine:	`Description`	Since Tracer is currently the only data collector within Perun that exploits dynamic instrumentation, we evaluated the optimizations exclusively using the Tracer and its engines.
	`Values`	$\{stap, ebpf\}$ for SystemTap/eBPF based profiling.
Cache:	`Description`	The impact of cache is twofold: (1) the SystemTap can cache compiled scripts for significant speedup (however, eBPF has no such functionality), and (2) the once extracted resources are stored within the Perun internals and can be promptly retrieved without the need for repeated extraction.
	`Values`	$\{cache\_on, cache\_off\}$ for (de)activating both the engine and optimization cache at once.

Experiment cases. We created a set of *experiment cases* that cover all of the implemented optimization methods as well as most of their parameters. Since evaluating all of the parameter combinations is infeasible (mainly due to the time requirements for large-scale projects), we carefully selected the tested parameters (and their combinations) based on (1) how significantly it changes the method behaviour and (2) the expected severity of impact it would have on the method results.

No Optimization	
`no-opt`	Data collection run; no optimizations enabled.
CG Shaping:	
`cg:m`	Matching mode of CG Shaping.
`cg:p`	Pruning mode of CG Shaping.

[2] Taken from https://stackabuse.com/quicksort-in-python/

`cg:p-1`	Pruning mode of CG Shaping; $call\_chain\_length = 1$.
`cg:p-max`	Pruning mode of CG Shaping; $call\_chain\_length = 1000$, which effectively prunes until *keep_top* threshold is reached.
`cg:t-s`	Trimming mode of CG Shaping; $threshold = soft$.
`cg:t-s:l`	Trimming mode of CG Shaping; $threshold = soft$; $keep\_leaf = true$.
`cg:t-r`	Trimming mode of CG Shaping; $threshold = strict$.
`cg:t-r:l`	Trimming mode of CG Shaping; $threshold = strict$; $keep\_leaf = true$.
Static Baseline:	
`sb:c`	Filters *constant* functions using the Static Baseline method.
`sb:l`	Filters *linear* functions using the Static Baseline method.
`sb:q`	Filters *quadratic* functions using the Static Baseline method.
Dynamic Baseline:	
`db:s:<i>`	Iteration <*i*> of Dynamic Baseline; $threshold = soft$.
`db:r:<i>`	Iteration <*i*> of Dynamic Baseline; $threshold = strict$.
Dynamic Sampling:	
`ds:d:<i>`	Iteration <*i*> of D. Sampling; no initial Dynamic Stats; default $step = 2$.
`ds:d:10:<i>`	Iteration <*i*> of D. Sampling; no initial Dynamic Stats; custom $step = 10$.
`ds:d:1.1:<i>`	Iteration <*i*> of D. Sampling; no initial Dynamic Stats; custom $step = 1.1$.
`ds:s:<i>`	Iteration <*i*> of D. Sampling; initial Dynamic Stats; $threshold = soft$.
`ds:r:<i>`	Iteration <*i*> of D. Sampling; initial Dynamic Stats; $threshold = strict$.
Diff Tracing:	
`dt`	Default of Diff Tracing; *semi-strict* CFG comparison.
`dt:l`	Diff Tracing; $keep\_leaf = true$.
`dt:i`	Diff Tracing; $inspect\_all = false$; no CG comparison.
`dt:s`	Diff Tracing; *soft* CFG comparison mode.
`dt:r`	Diff Tracing; *strict* CFG comparison mode.
Dynamic Probing:	
`dp`	Default configuration of Dynamic Probing method; $threshold = 100000$; *re-attach* mode off.
`dp:l`	Dynamic Probing method; lower $threshold = 10000$.
`dp:h`	Dynamic Probing method; higher $threshold = 1000000$.
`dp:r`	Dynamic Probing method; *re-attach* mode on.
Timed Sampling:	
`ts`	Default Timed Sampling configuration; $frequency = 1Hz$.
`ts:2`	Timed Sampling; custom $frequency = 2Hz$.
`ts:4`	Timed Sampling; custom $frequency = 4Hz$.

Note that for most iterative experiment cases (except `ds:d:*`) we use pre-computed Dynamic Stats resource to save time and achieve more consistent results (since slightly different Dynamic Stats may be obtained in each initial iteration of iterative methods). Moreover, the Diff Tracing method utilizes results from the project version specified in the *Test projects* table.

We also prepared a set of experiment cases for all of the proposed pipelines, however, since pipelines *consist* of pre-defined configuration of methods *and* parameters, we tested only their default behaviour with no additional manual alteration of parameters or methods.

Basic:	
`p:d:b`	The `Basic` pipeline with no computed Dynamic Stats resource.
`p:b`	The `Basic` pipeline with pre-computed Dynamic Stats resource.
Advanced:	
`p:d:a`	The `Advanced` pipeline with no computed Dynamic Stats resource.
`p:a`	The `Advanced` pipeline with pre-computed Dynamic Stats resource.
Full:	
`p:d:f`	The `Full` pipeline with no computed Dynamic Stats resource.
`p:f`	The `Full` pipeline with pre-computed Dynamic Stats resource.

Machine Specification. We conducted the experiments on a machine with the following specification:

Arch	x86_64
CPU	i7-4600U 2.1GHz (3.3GHz Turbo), 2 Cores (4 Threads)
Cache	32K L1, 256K L2, 4096K L3
RAM	12GB
SSD	240GB SATA 3.0 6Gb/s

8.2 Evaluation Results

To simplify the presentation, we present only a limited subset of the results. The complete set of results can be found in Appendices B and C. Specifically, we present selected interesting results of both the `CPython` and `CCSDS` projects.

8.2.1 CPython Project Evaluation

We selected metrics (and their combinations) that, to the best of our knowledge, highlight the most significant differences between our methods. Note, however, that `CPython` is rather large-scale project that utilizes more advanced compilation pipeline, so, we were not able to run some optimizations: the Static Baseline (due to its compilation process restrictions), Dynamic Probing and Timed Sampling (due to their requirements of the eBPF engine that currently does not support such large amount of probe locations). Hence, we evaluated the `CPython` project using only the SystemTap configurations.

Impact on the profiling process. Figure 8.1 demonstrates how the number of injected probes — both the instrumented (`M_PL`) and the actually reached during the profiling (`M_PLR`) — affects the profiling (`M_PT`), collection (`M_CPT`) or program run (`M_PRT`) times without any caching. For the `no-opt` case, the probe locations (i.e., functions) are obtained using the Tracer strategies (see Section 4.2) which exploit the *symbol table* within the executable binary; the *pre-optimize* techniques rely on the extracted Call Graph structure (leveraging the `angr`) and contains only functions reachable from the `main` function and within the same executable (see Section 7.1.1). Note that the symbol table contains all of the functions referenced in the executable and, since our extracted CG is quite limited, we do not instrument certain reachable functions (e.g., functions outside of `main` or in external `libraries`). We consider these functions as a perspective future work.

Naturally, although all of the instrumented functions in *pre-optimize* are reachable from the `main` function, the actual number of reached probe locations (`M_PLR`) depends on the introduced workload, thus creating the $M\_PL - M\_PLR$ gap. The second graph, shows that the Program Run Time (`M_PRT`) is negligible compared to the Profiling (`M_PT`) or Collect Phase (`M_CPT`) times, which is mainly a consequence of no caching: the collect phase is influenced mostly by the probe injection (e.g., SystemTap script compilation), and in the rest of the phases, the most time-intensive operations are (1) the extraction of optimization resources, and (2) raw data parsing and transformation to the Perun profile format. The `dt:i` case achieved the fastest profiling time out of all the optimization cases since it also instrumented the least functions. The `cg:m` case, on the other hand, has not only instrumented the most functions, but the *cg matching* also employs no other form of optimization besides filtering functions that are not in the CG, thus generating the most raw performance data which must be further parsed.

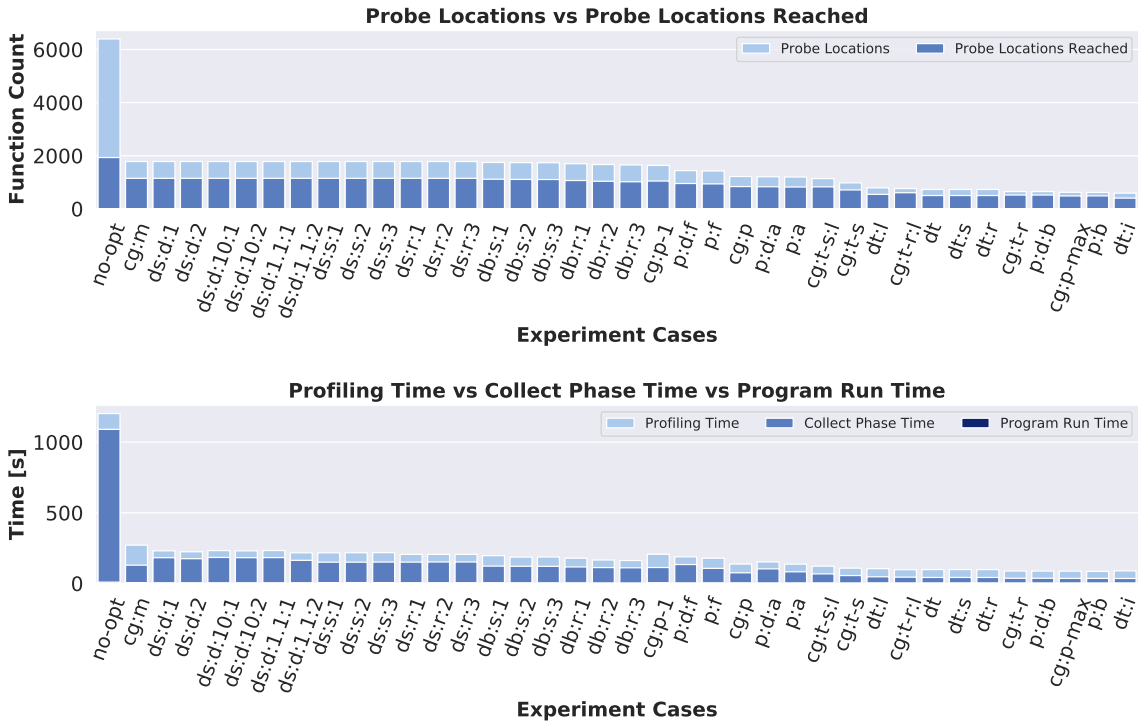

Figure 8.1: *Bar graphs* that compare (1) the number of instrumented (`M_PL`) and actually reached (`M_PLR`) probes (i.e., the total number of functions with probes versus the number of probes that generated at least one performance record), and (2) the profiling (M_PT), collect phase (`M_CPT`) and program run (`M_PRT`) times. We can see the number of instrumented probes contributes significantly, although not exclusively, to the overall runtime. Thus, minimizing the number of probes can substantially shorten the profiling.

When, we set the *cache_on* configuration then both the Profiling Time and Collect Phase Time are greatly reduced thanks to the cached results of SystemTap script compilation and optimization resources extraction as is shown in Figure C.1.

Impact of the Raw Data Volume. Figure 8.2 shows how the volume of Raw Data (`M_RDV`) influences the Profiling Time (`M_PT`) for every experiment case, as well as the Hotspot Coverage (`M_HC`). Methods with low Raw Data Volume and high Hotspot Coverage can precisely identify and filter out functions that generate considerable fraction of the total raw performance data while having only minor effect on the performance information gain.

Note, that in the bottom graph, we have removed those experiment cases that employ some kind of sampling. This is because the Hotspot Coverage computation currently excludes recursive function calls (as briefly described in 5.1) detected during the raw data parsing, and since employing any sampling technique may render this detection process impossible (as only the first call of the recursive sequence is recorded), the coverage values obtained for such experiment cases are generally invalid, e.g., in the case of `ds:*`, `p:f` or `p:a`. Full results are found in the Appendix B).

Moreover, we also decided to not include the `no-opt` case in both of the scatter plots, as it considerably skews the graphs due to its rather large raw data and profiling time values (approximately $1.2GB$ of raw data and $1200s$ of profiling time). Note the majority of `cg:*`

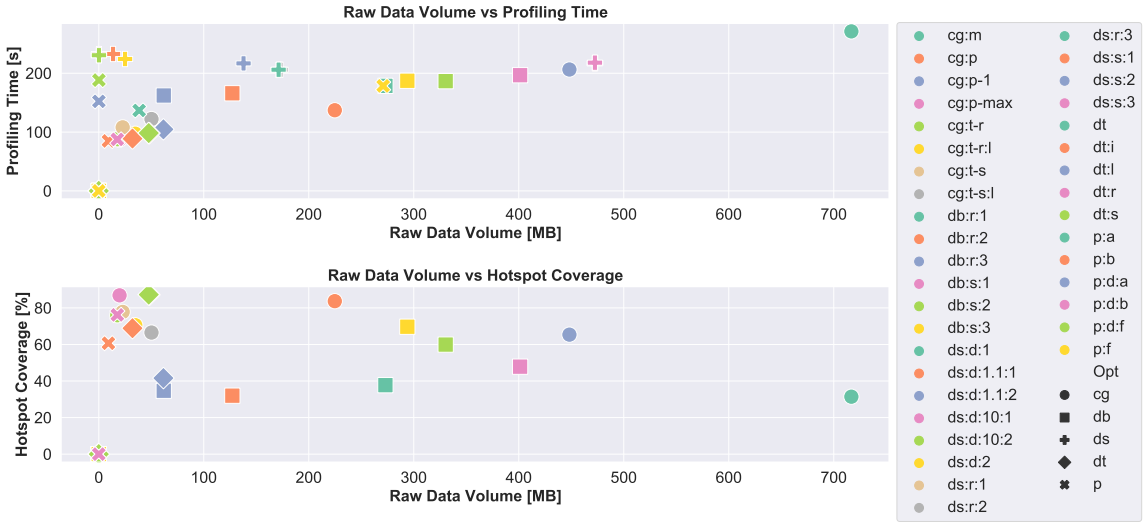

Figure 8.2: A relation between the Profiling Time (`M_PT`), Raw Data Volume (`M_RDV`) and Hotspot Coverage (`M_HC`) metrics for most of the experiment cases. Generally, methods that achieve short Profiling Time, low Raw Data Volume and high Hotspot Coverage should be always preferred. Note that we excluded the `no-opt` case (its large values skew the graphs considerably) from both graphs, as well as cases that employ any form of sampling (such as `ds:*` or *Advanced* and *Full* pipelines) from the Hotspot Coverage scatter plot; the reason being unreliable coverage computation (due to the record sampling) resulting in invalid values. The full results can be found in the Appendix B).

cases which achieve a reasonable `RDV` / `PT` / `HC` ratios. However, the Hotspot Coverage, in its current form, excludes the time spent in $\perp$ functions which can cause imprecise results (especially noticeable in the `cg:m` case).

Impact on performance modelling. Figure 8.3 shows how many functions we can successfully analyze by the regression analysis (`M_FC`) and, in how many cases the resulting *best model* is (un)reliable (denoted *Obtained* resp. *Unclear*). We say the model is *reliable* if its R^2 (coefficient of determination) is higher than 0.5, i.e., if the model can *explain* at least half the observed error). Note that the $M\_PLR - M\_FC$ gap is caused by reached functions that, have not met the requirements for regression analysis which expects a minimum of three records (since the regression cannot be reliably applied to one or two data points). Moreover, an extreme case of data sampling technique can cause the regression analysis to fail altogether, as demonstrated by the Dynamic Sampling method (or Advanced and Full pipelines) with no initial Dynamic Stats resource and large *step* parameter (i.e., cases with minimal to none `M_FC` records in Appendix B Tables). The other methods seem to have no significant impact on the modelling precision, in comparison to `no-opt` configuration.

8.2.2 CCSDS Project Evaluation

Contrary to the `CPython`, the `CCSDS` has no restrictions, and hence, we successfully measured the all evaluation metrics for all collector configurations. Note, that the difference between the *cache_on* and *cache_off* results was rather marginal, and, hence, we chose the results

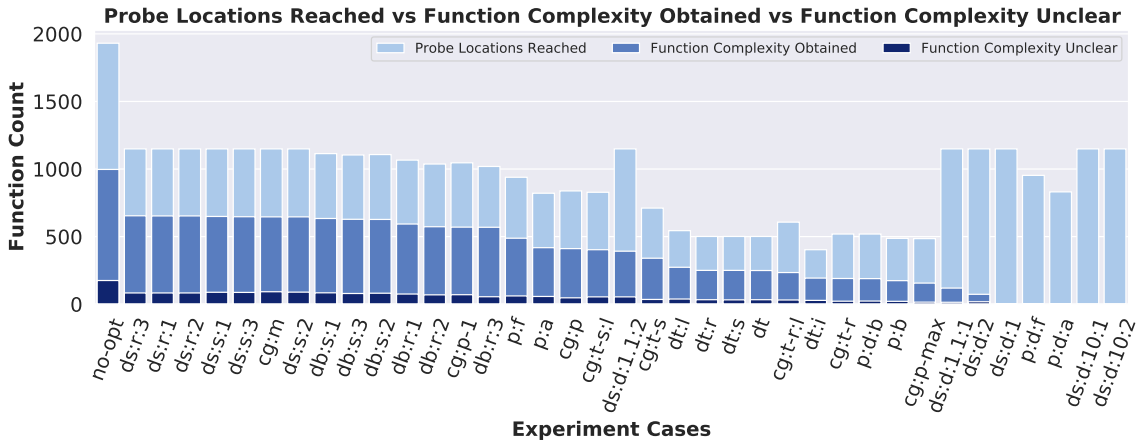

Figure 8.3: Numbers of function performance models that we can infer from data collected by each case with (1) $R^2 \geq 0.5$ (*Function Complexity Obtained*), or (2) $R^2 < 0.5$ (*Function Complexity Unclear*). Functions with less than three records were omitted.

of *cache_on* configurations to evaluate the aforementioned methods as well as showcase the differences between the SystemTap and eBPF Tracer engines.

Comparison of Tracer engines. Figure 8.4 compares the Profiling Time (`M_PT`) for both the SystemTap and eBPF Tracer engines. We particularly compare the cases which were not measured in the `CPython` evaluation (i.e., `sb:*`, `dp:*` and `ts:*`) as well as a selected subset of the remaining methods and pipelines. Note, that due to the number of total experiment cases, we filtered some of the cases for the sake of presentation. To be precise, we removed subsequent iterations of iterative methods as the initial iteration provides sufficient estimation of the method behaviour.

We noticed the considerable disparity in the Profiling Time of both engines, which is caused by the fundamentally different instrumentation approach: while majority of the SystemTap overhead is caused by the *script to kernel module* compilation, eBPF avoids such time-expensive operation by dynamically attaching probes through its internal kernel virtual machine. Although this introduces time overhead as well, it is not as significant as the compilation process utilized by the SystemTap. Still, one has to consider that eBPF has a hard limit on number of active probes, hence, both engines have complementary usage.

Impact of sampling on performance models. Figure 8.5 shows how certain methods affect the R^2 of linear and constant models, i.e. each point corresponds to a model of a function and its R^2 with or without optimizations. We believe this limitation is sensible, since (1) to the best of our knowledge, the `CCSDS` implementation should mostly consist of constant and linear functions anyway, and (2) both constant and linear models can be used as a rough estimate of the data and, hence, their precision should not change significantly. We show only selected optimizations that we believe cause the most significant difference in the R^2, specifically, the Dynamic Sampling, Dynamic Probing and Timed Sampling.

Ideally, the models should achieve a similar R^2 for both the unoptimized and optimized collection, and thus be located around the diagonal auxiliary line. In some cases (e.g. the Dynamic Probing), the method actually yielded a more precise models in most cases. However, as expected, the suspected methods have negatively impacted more than a half

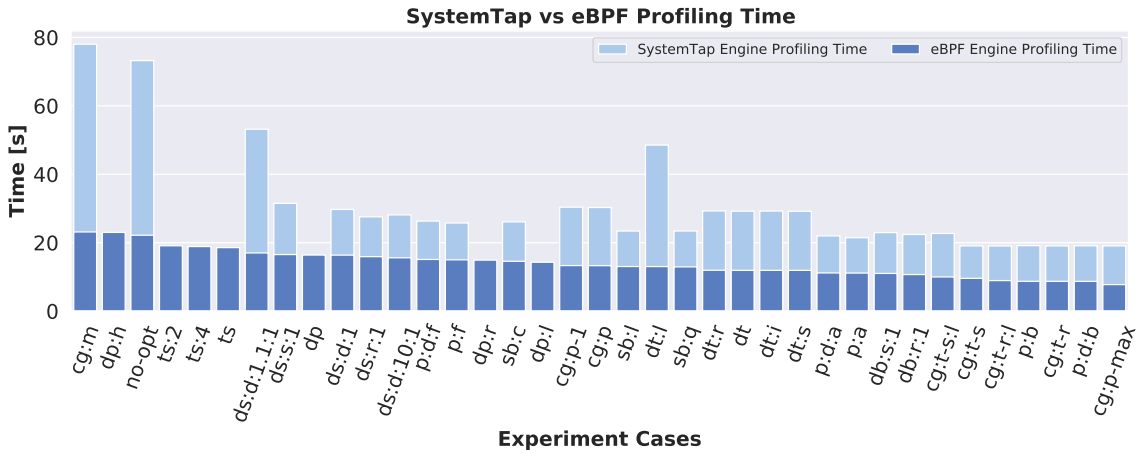

Figure 8.4: Comparison of the SystemTap and eBPF engine, featuring all of the methods and pipelines, and a selected subset of the total experiment cases. Specifically, we kept only the first iteration of every case since the values of subsequent iterations were rather similar.

of the function models, in both the constant and linear cases. Note, however, that even the cases where a superior R^2 values are achieved can lead to a negative outcome, especially when such models are then considered to be the *best fitting* ones.

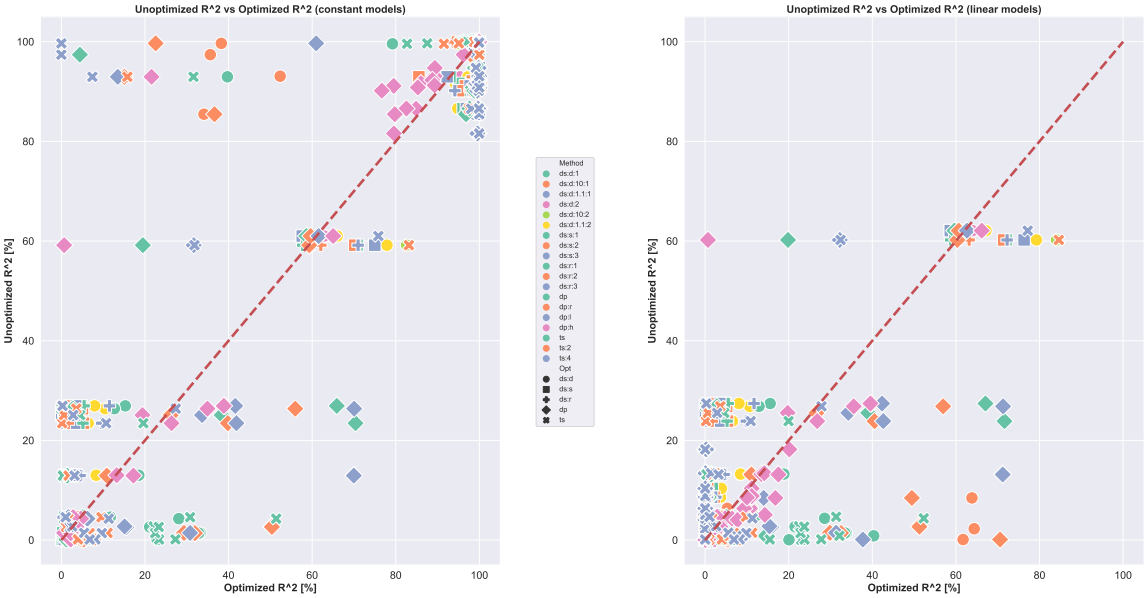

Figure 8.5: A comparison of the R^2 of constant (left) and linear (right) models with (*optimized*) and without (*unoptimized*) optimizations. For the comparison, we selected only the methods that were expected to significantly impact the modelling due to their nature. We can see, that sampling has indeed negatively affected all the models' quality.

8.2.3 Summary

We have demonstrated our methods on a particular subset of results chosen for comparison of the individual methods and pipelines with respect to a few carefully selected metrics. Now, we summarize and evaluate all of the proposed techniques, with the focus on the *primary optimization criterion* that is different for various methods.

Tables D.1 and D.2 in the Appendix D (to shorten the presentation) compare the `M_PL`, `M_PT`, `M_RDV` and `M_FC` metrics of selected *collector configurations* for both *test projects* — specifically, for each experiment case and selected metric, we evaluated by how much (in %) has the observed metric changed. We summarize the results in Table 8.6 that evaluates whether the individual methods are indeed capable of optimizing their primary optimization criterion (as defined in Section 7.2), based on the achieved magnitude of change.

We conclude that most of the proposed optimization techniques proved to be effective in reducing the number of collection points, speeding up the collection process or diminishing the excessive amount of collected raw performance data. As the Diff Tracing method employs a distinctive optimization approach, we were not able to properly quantify the degree of optimization using a single metric, however, our results indeed show that certain optimization of the observed metrics has been accomplished.

The Dynamic Probing method — although capable of achieving speed-up of the profiling process — can also cause a slight slowdown when using inappropriate parameters for the given test project. However, since the inherent overhead of the method decreases over time (as more and more probes are being deactivated), we believe that for experiments with longer Program Run Time (`M_PRT`), the method should show more promising results.

Table 8.6: A summary of all of the proposed and implemented individual optimization techniques. The complete results can be found in Appendix D.

Method	OC	Change [Δ%]	Verdict	Note
CG Shaping:	`OC_CP`	250% – 900% `[CPython]` 140% – 1400% `[CCSDS]`	✓	Significant reduction of Collection Points even for the *match* mode.
Static Baseline	`OC_T`	50% – 70% `[ebpf]` 180% – 210% `[stap]`	✓	Mediocre speed-up for both the SystemTap and eBPF Tracer engines in `CCSDS`.
Dynamic Baseline	`OC_T`	510% – 570% `[CPython]` 100% – 220% `[CCSDS]`	✓	Mediocre speed-up which, however, depends on the used engine.
Dynamic Sampling	`OC_DV`	170% – 9e5% `[CPython]` 80% – 2e6% `[CCSDS]`	✓	The Data Volume reduction is extremely dependant on the initial `step` parameter.
Diff Tracing	`OC_F`	n/a	✓	The OC_F is not easily quantifiable as its optimization is different from the other techniques, however, the `PL`, `PT` and `RDV` metrics were all optimized while keeping the `FC` ratio around the `no-opt` level.
Dynamic Probing	`OC_T`	−4% – 55% `[ebpf]` `[CCSDS]`	✓ / ✗	The speed-up and overhead depend greatly on the selected configuration.
Timed Sampling	`OC_DV`	120% – 190% `[ebpf]` `[CCSDS]`	✓	Mediocre Data Volume reduction compared to the Dynamic Sampling. The distribution of function call records can be fine-tuned by its `frequency` parameter.

Chapter 9

Conclusion

The goal of this work was to propose, implement and evaluate optimization techniques for efficient performance analysis and modelling, as implemented in the Perun framework. In particular, it focused on (1) improving the precision of data collection process, (2) reducing the volume of the generated raw data and subsequently the size of the resulting profile, (3) scaling down the amount of instrumented program locations, and (4) minimizing the time overhead caused by the profiling.

First, we had to enhance both Perun and Tracer collector architecture to allow (a) incorporating optimization routines within the Perun workflow and (b) leveraging both the SystemTap and eBPF technologies using the Tracer. This extension allowed us to further propose potential optimizations of profiling of projects with version control history.

We have noticed, that in general, the profiling could be optimized based on three areas: (i) semantic properties of profiled locations, (ii) structure of the SUT, and (iii) the profiling process itself. We explored each of these and proposed several suitable techniques. Specifically, the *Static Baseline* and *Dynamic Baseline* methods exploit static and dynamic analysis of program functions and estimate their complexities: we then avoid profiling functions that are expected to provide none to minimal information gain. Second, *Call Graph Shaping* and *Diff Tracing* leverage the structure of *CG* and CFG, combined with information about discovered machine code changes, to filter varying amount of functions of different depths of call traces. At last, instead of deactivating some collection probes completely, *Dynamic Sampling*, *Timed Sampling* and *Dynamic Probing* control how often the instrumented probes generate performance data.

Now, while all of these optimizations can be, indeed, applied by themselves, we noticed that there exist certain synergies between them, that make them suitable to be used together. Hence, we introduced the concept of *pipelines* — pre-configured combinations of optimization methods — to take advantage of these synergies.

Our experiments have demonstrated that nearly all of the proposed techniques and pipelines have achieved encouraging results, i.e., significant degree of optimization, thus making it possible to profile even otherwise infeasible scenarios and projects (e.g., `Cpython`). Also, we were able to satisfy all of the stated Functional and Non-functional Requirements, with the sole exception of *maintainability*, that we plan to resolve in the future.

We concluded each method with the potential future work, however, generally, extending the Tracer to handle highly modular programs (e.g., with numerous shared libraries) could push the profiling capabilities of Perun significantly further. Moreover, we plan to publish the results achieved in this work at relevant conferences (e.g., ICST or ISSTA).

Bibliography

[1] 122.0 B 2, C. *CCSDS Recommended Standard for Image Data Compression*. Standard. Washington, DC, USA: The Consultative Committee for Space Data Systems, september 2017.

[2] Allen, F. E. Control Flow Analysis. In: *Proceedings of a Symposium on Compiler Optimization*. New York, NY, USA: Association for Computing Machinery, 1970, p. 1–19. Available at: https://doi.org/10.1145/800028.808479. ISBN 9781450373869.

[3] *BCC: BPF Compiler Collection* [online]. [cit. 2020-3-11]. Available at: https://github.com/iovisor/bcc.

[4] Bhansali, S., Chen, W.-K., Jong, S. de, Edwards, A., Murray, R. et al. Framework for Instruction-level Tracing and Analysis of Program Executions. In: *Proceedings of the 2Nd International Conference on Virtual Execution Environments*. New York, NY, USA: ACM, 2006, p. 154–163. VEE '06. Available at: http://doi.acm.org/10.1145/1134760.1220164. ISBN 1-59593-332-8.

[5] Bruening, D. L. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation*. USA, 2004. Dissertation. Massachusetts Institute of Technology.

[6] Calavera, D. and Fontana, L. *Linux Observability with BPF: Advanced Programming for Performance Analysis and Networking*. 1st ed. O'Reilly Media, 2019. ISBN 1492050202.

[7] Chen, Y.-C. *Introduction to Nonparametric Statistics: Lecture 9*. 2018. Unpublished, online. Available at: http://faculty.washington.edu/yenchic/18W_425/Lec9_Reg01.pdf.

[8] Corbet, J. *Kernel markers* [online]. [cit. 2019-12-31]. Available at: https://lwn.net/Articles/245671/.

[9] Corbet, J. *On DTrace envy* [online]. [cit. 2020-1-3]. Available at: https://lwn.net/Articles/244536/.

[10] Corbet, J. *Uprobes in 3.5* [online]. [cit. 2020-1-1]. Available at: https://lwn.net/Articles/499190/.

[11] *CPython: The Python programming language* [online]. [cit. 2020-5-31]. Available at: https://github.com/python/cpython.

[12] Damato, J. *How does strace work* [online]. [cit. 2020-1-2]. Available at: https://blog.packagecloud.io/eng/2016/02/29/how-does-strace-work/.

[13] Desnoyers, M. *Using the Linux Kernel Tracepoints* [online]. [cit. 2019-12-31]. Available at: https://www.kernel.org/doc/Documentation/trace/tracepoints.txt.

[14] Devore, J. L. *Probability and Statistics for Engineering and the Sciences*. 8th ed. Cengage Learning, 2011. ISBN 0-8400-6827-1.

[15] *DTrace.org* [online]. [cit. 2019-12-31]. Available at: http://dtrace.org/blogs/about/.

[16] Dzyoba, A. *Ftrace* [online]. [cit. 2020-1-2]. Available at: https://alex.dzyoba.com/blog/ftrace/.

[17] Eranian, S., Gouriou, E., Moseley, T. and Bruijn, W. de. *Linux kernel profiling with `perf`* [online]. [cit. 2019-12-29]. Available at: https://perf.wiki.kernel.org/index.php/Tutorial.

[18] *Infer Static Analyzer: A tool to detect bugs in Java and C/C++/Objective-C code before it ships* [online]. [cit. 2020-5-30]. Available at: https://fbinfer.com/.

[19] Fiedor, T., Holík, L., Rogalewicz, A., Sinn, M., Vojnar, T. et al. From Shapes to Amortized Complexity. In: Dillig, I. and Palsberg, J., ed. *Verification, Model Checking, and Abstract Interpretation*. Cham: Springer International Publishing, 2018, p. 205–225. ISBN 978-3-319-73721-8.

[20] Fiedor, T. *Perun: Lightweight Performance Version System* [online, Github Repository]. [cit. 2020-1-13]. Available at: https://github.com/tfiedor/perun.

[21] Fiedor, T. and Pavela, J. *Perun Documentation: Release 0.17.3*. 2020. Unpublished, online. Available at: https://github.com/tfiedor/perun/blob/master/docs/pdf/perun.pdf.

[22] *GCC: Program Instrumentation Options* [online]. [cit. 2019-12-31]. Available at: https://gcc.gnu.org/onlinedocs/gcc/Instrumentation-Options.html.

[23] Goswami, S. *An introduction to KProbes* [online]. [cit. 2020-1-1]. Available at: https://lwn.net/Articles/132196/.

[24] Gregg, B. *Choosing a Linux Tracer (2015)* [online]. [cit. 2020-1-4]. Available at: http://www.brendangregg.com/blog/2015-07-08/choosing-a-linux-tracer.html.

[25] Gregg, B. *Linux Extended BPF (eBPF) Tracing Tools* [online]. [cit. 2020-1-4]. Available at: http://www.brendangregg.com/ebpf.html.

[26] Gregg, B. *Perf Examples* [online]. [cit. 2020-1-4]. Available at: http://www.brendangregg.com/perf.html.

[27] Gregg, B. *DTrace Topics: Introduction*. 2007. Unpublished, online. Available at: http://www.brendangregg.com/Slides/dtrace_topics_intro.pdf.

[28] Gregg, B. *BPF Performance Tools: Linux System and Application Observability*. 1st ed. Addison-Wesley Professional, 2019. ISBN 0136554822.

[29] Grove, D. and Chambers, C. A Framework for Call Graph Construction Algorithms. *ACM Trans. Program. Lang. Syst.* New York, NY, USA: Association for Computing Machinery. november 2001, vol. 23, no. 6, p. 685–746. Available at: https://doi.org/10.1145/506315.506316. ISSN 0164-0925.

[30] Hollingsworth, J. K., Miller, B. P. and Cargille, J. Dynamic program instrumentation for scalable performance tools. *Proceedings of IEEE Scalable High Performance Computing Conference*. 1994, p. 841–850.

[31] Karger, D., Motwani, R. and Ramkumar, G. On Approximating the Longest Path in a Graph. *Algorithmica*. september 1995, vol. 18.

[32] Koláček, J. *Jádrové odhady regresní funkce [czech]*. Czech Republic, 2004. Dissertation. Masaryk University, Faculty of Science.

[33] Krátký, R., Jahoda, M., Domingo, D. and Cohen, W. *SystemTap Beginners Guide* [online]. [cit. 2020-1-4]. Available at: https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/systemtap_beginners_guide/index.

[34] Křena, B. and Vojnar, T. Automated formal analysis and verification: an overview. *International Journal of General Systems*. 2013, vol. 2013, no. 42, p. 335–365. Available at: https://www.fit.vut.cz/research/publication/10284. ISSN 0308-1079.

[35] Li, Z. *Control Flow Graph Based Attacks: In the Context of Flattened Programs*. Stockholm, Sweden, 2014. Master's thesis. KTH, School of Computer Science and Communication.

[36] *Linux manual page for* `ltrace` [online]. [cit. 2019-12-30]. Available at: http://man7.org/linux/man-pages/man1/ltrace.1.html.

[37] *How does ltrace work?* [online]. [cit. 2020-1-3]. Available at: https://blog.packagecloud.io/eng/2016/03/14/how-does-ltrace-work/.

[38] *LTTng Documentation* [online]. [cit. 2020-1-24]. Available at: https://lttng.org/docs/v2.11/.

[39] Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A. et al. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In: *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*. New York, NY, USA: Association for Computing Machinery, 2005, p. 190–200. PLDI '05. Available at: https://doi.org/10.1145/1065010.1065034. ISBN 1595930566.

[40] Magill, S., Tsai, M.-H., Lee, P. and Tsay, Y.-K. THOR: A Tool for Reasoning about Shape and Arithmetic. In: Gupta, A. and Malik, S., ed. *Computer Aided Verification*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, p. 428–432. ISBN 978-3-540-70545-1.

[41] Malík, V. *DiffKemp: Tool for semantic Difference of Kernel functions, modules, and parameters.* [online]. [cit. 2020-5-30]. Available at: https://github.com/viktormalik/diffkemp.

[42] McCanne, S. and Jacobson, V. The BSD Packet Filter: A New Architecture for User-Level Packet Capture. In: *Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings*. USA: USENIX Association, 1993, p. 2. USENIX'93.

[43] Meng, K. and Norris, B. Mira: A Framework for Static Performance Analysis. In: *2017 IEEE International Conference on Cluster Computing (CLUSTER)*. Sep. 2017, p. 103–113. ISSN 2168-9253.

[44] Merey, A. *What are BPF Maps and how are they used in stapbpf* [online]. [cit. 2020-1-4]. Available at: https://developers.redhat.com/blog/2017/12/15/bpf-maps-used-stapbpf/.

[45] Nethercote, N. *Dynamic binary analysis and instrumentation*. UK, 2004. Dissertation. University of Cambridge.

[46] Nethercote, N. and Seward, J. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In: *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*. New York, NY, USA: Association for Computing Machinery, 2007, p. 89–100. PLDI '07. Available at: https://doi.org/10.1145/1250734.1250746. ISBN 9781595936332.

[47] *Lecture Notes 21: Nonparametric Regression*. 2018. Unpublished, online. Available at: http://www.stat.cmu.edu/~larry/=stat700/Lecture21.pdf.

[48] *Carnegie Mellon University: Nonparametric Regression slides*. 2017. Unpublished, online. Available at: http://www.stat.cmu.edu/~larry/=stat401/NonparametricRegression.pdf.

[49] *Wisconsin University: Nonparametric Regression slides*. 2009. Unpublished, online. Available at: https://www.ssc.wisc.edu/~bhansen/718/NonParametrics2.pdf.

[50] *OProfile - A System Profiler for Linux* [online]. [cit. 2020-1-15]. Available at: https://oprofile.sourceforge.io/news/.

[51] Patel, R. and Rajwat, A. A Survey of Embedded Software Profiling Methodologies. *International Journal of Embedded Systems and Applications*. december 2013, vol. 1.

[52] Pavela, J. *Library for Profiling of Data Structures of C/C++ Programs*. Brno, 2017. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology.

[53] Pavela, J. *Dynamic Analysis, Modeling and Prediction of Performance of Programs*. Brno University of Technology, Faculty of Information Technology, 2018. Project Practice.

[54] Pavela, J. *Dynamic Analysis, Modeling and Prediction of Performance of Programs*. Brno University of Technology, Faculty of Information Technology, 2019. Project Practice 2.

[55] Pavela, J. and Stupinský Šimon. Towards the detection of performance degradation. In: *Excel@FIT'18*. Brno, Czech Republic: [b.n.], 2018.

[56] *Perf Introduction* [online]. [cit. 2020-1-4]. Available at: https://perf.wiki.kernel.org/index.php/Main_Page.

[57] *Meet PerfRepo* [online]. [cit. 2020-1-26]. Available at: https://github.com/PerfCake/PerfRepo.

[58] Rostedt, S. *Debugging the kernel using Ftrace - part 1* [online]. [cit. 2020-1-2]. Available at: https://lwn.net/Articles/365835/.

[59] Rostedt, S. ***Ftrace*** *documentation* [online]. [cit. 2019-12-30]. Available at: https://www.kernel.org/doc/Documentation/trace/ftrace.txt.

[60] Rostedt, S. *Using the TRACE_EVENT() macro (Part 1)* [online]. [cit. 2019-12-31]. Available at: https://lwn.net/Articles/379903/.

[61] Shoshitaishvili, Y., Wang, R., Salls, C., Stephens, N., Polino, M. et al. SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In:. May 2016, p. 138–157.

[62] Shye, A., Iyer, M., Reddi, V. J. and Connors, D. A. Code Coverage Testing Using Hardware Performance Monitoring Support. In: *Proceedings of the Sixth International Symposium on Automated Analysis-Driven Debugging*. New York, NY, USA: Association for Computing Machinery, 2005, p. 159–163. AADEBUG'05. Available at: https://doi.org/10.1145/1085130.1085151. ISBN 1595930507.

[63] Sinn, M., Zuleger, F. and Veith, H. A Simple and Scalable Static Analysis for Bound Analysis and Amortized Complexity Analysis. In:. January 2014.

[64] *Linux manual page for* ***strace*** [online]. [cit. 2019-12-30]. Available at: http://man7.org/linux/man-pages/man1/strace.1.html.

[65] Stupinský Šimon. *Automatic Detection of Performance Degradation*. Brno University of Technology, Faculty of Information Technology, 2018. Project Practice.

[66] Stupinský Šimon. *New Models for Automatic Detection of Performance Degradation*. Brno, 2019. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology.

[67] Tamches, A. and Miller, B. P. Fine-Grained Dynamic Instrumentation of Commodity Operating System Kernels. In: *Proceedings of the Third Symposium on Operating Systems Design and Implementation*. USA: USENIX Association, 1999, p. 117–130. OSDI '99. ISBN 1880446391.

[68] *Linux manual page for* ***tcpdump*** [online]. [cit. 2019-12-30]. Available at: http://man7.org/linux/man-pages/man1/tcpdump.1.html.

[69] Tibshirani, R. *Advanced Methods for Data Analysis: Kernel Regression*. 2014. Unpublished, online. Available at: https://www.stat.cmu.edu/~ryantibs/advmethods/notes/kernel.pdf.

[70] *Linux manual page for* ***top*** [online]. [cit. 2019-12-30]. Available at: http://man7.org/linux/man-pages/man1/top.1.html.

[71] Tschüter, R., Ziegenbalg, J., Wesarg, B., Weber, M., Herold, C. et al. An LLVM Instrumentation Plug-in for Score-P. In: *Proceedings of the Fourth Workshop on the LLVM Compiler Infrastructure in HPC*. New York, NY, USA: Association for Computing Machinery, 2017. LLVM-HPC'17. Available at: https://doi.org/10.1145/3148173.3148187. ISBN 9781450355650.

[72] *Uroboros: Infrastructure for Reassembleable Disassembling and Transformation* [online, Github Repository]. [cit. 2020-1-24]. Available at: https://github.com/s3team/uroboros.

[73] *Valgrind's Tool Suite* [online]. [cit. 2020-1-15]. Available at: https://valgrind.org/info/tools.html.

[74] Veitch, A., Berris, D., Anderson, E., Heintze, N. and Wang, N. *XRay: A Function Call Tracing System* [online]. Google, Inc., 2016. [cit. 2019-12-31]. Available at: https://research.google/pubs/pub45287/.

[75] Vitovská, M. *Instrumentation of LLVM IR*. Brno, 2017. Master's thesis. Masaryk University, Faculty of Informatics.

[76] Wang, S., Wang, P. and Wu, D. Reassembleable Disassembling. In: *Proceedings of the 24th USENIX Conference on Security Symposium*. USA: USENIX Association, 2015, p. 627–642. SEC'15. ISBN 9781931971232.

[77] Wu, J., Cui, H. and Yang, J. Bypassing Races in Live Applications with Execution Filters. In: *Proceedings of the Ninth Symposium on Operating Systems Design and Implementation*. 2010. OSDI '10.

[78] Xie, Y., Naik, M., Hackett, B. and Aiken, A. Soundness and its Role in Bug Detection Systems (position paper). In: *Proceedings of Workshop on the Evaluation of Software Defect Detection Tools*. 2005. BUGS '05.

[79] Zakamulin, V. *Market Timing with Moving Averages: The Anatomy and Performance of Trading Rules*. 2015. Unpublished, online draft. Available at: https://pdfs.semanticscholar.org/7179/1c009a61f0f9d53077a2fbc72e808a31d5a0.pdf.

[80] Zakamulin, V. *Market Timing with Moving Averages: The Anatomy and Performance of Trading Rules*. 1st ed. Springer, 2017. ISBN 331960970X.

[81] Zhang, M., Qiao, R., Hasabnis, N. and Sekar, R. A Platform for Secure Static Binary Instrumentation. In: *Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. New York, NY, USA: Association for Computing Machinery, 2014, p. 129–140. VEE '14. Available at: https://doi.org/10.1145/2576195.2576208. ISBN 9781450327640.

[82] Zhao, Z. Parametric and nonparametric models and methods in financial econometrics. *ArXiv.org, Quantitative Finance Papers*. january 2008, vol. 1.

[83] Zhu, X. *Basics of Statistical Machine Learning*. 2018. Unpublished, online. Available at: http://pages.cs.wisc.edu/~jerryzhu/cs731/stat.pdf.

Appendix A

Miscellaneous

Here we present various Tables and Listings that were not included in the Thesis Chapters, mainly due to their scale and auxiliary nature, i.e., they are not fundamental for proper understanding of the presented concepts, but rather aim to show some of the underlying details.

Table A.1: The *Command Line Interface (CLI)* options of the new Tracer module, i.e., after introducing the engine architecture. Note that the `engine` option is related to the concept of *collection engines* (introduced in Section 6.1).

-e	**--engine** `[stap | ebpf]` Sets the data collection engine to be used: - stap: the SystemTap framework - ebpf: the eBPF framework
-s	**--strategy** `[userspace | all | u_sampled | a_sampled | custom]` Select strategy for probing the binary. See documentation for detailed explanation for each strategy. [required]
-f	**--func** `<func>` Set the probe point for the given function as `<lib>#<func>#<sampling>`.
-u	**--usdt** `<usdt>` Set the probe point for the given USDT as `<lib>#<usdt>#<sampling>`.
-d	**--dynamic** `<dynamic>` Set the probe point for the given code location as `<lib>#<line>#<sampling>`.
-g	**--global-sampling** `<int>` Set the global sample for all probes, sampling parameter for specific rules have higher priority.
--with-usdt, --no-usdt `<flag>`	The selected strategy will also extract and profile USDT probes.
-b	**--binary** `<path>` The profiled executable. If not set, then the command is considered to be the profiled executable and is used as a binary parameter.
-t	**--timeout** `<float>` Set time limit (in seconds) for the profiled command, i.e. the command will be terminated after reaching the time limit. Useful for, e.g., endless commands.
-z	**--zip-temps** `<flag>`

Zip and compress the temporary files (SystemTap log, raw performance data, watchdog log, etc.) into the Perun log directory before deleting them.

-k **--keep-temps** `<flag>`
Do not delete the temporary files in the file system.

-vt **--verbose-trace** `<flag>`
Set the trace file output to be more verbose, useful for debugging.

-q **--quiet** `<flag>`
Reduces the verbosity of the collector info messages.

-w **--watchdog** `<flag>`
Enable detailed logging of the whole collection process.

-o **--output-handling** `[default | capture | suppress]`
Sets the output handling of the profiled command:
- default: the output is displayed in the terminal
- capture: the output is being captured into a file as well as displayed in the terminal (note that buffering causes a delay in the terminal output)
- suppress: redirects the output to the `DEVNULL`

-i **--diagnostics** `<flag>`
Enable detailed surveillance mode of the collector. The collector turns on detailed logging (watchdog), verbose trace, capturing output etc. and stores the logs and files in an archive (zip-temps) in order to provide as much diagnostic data as possible for further inspection.

-sc **--stap-cache-off** `<flag>`
Disables the SystemTap caching of compiled scripts.

Table A.2: The new CLI options of the Perun `collect` command related to the optimizations, e.g., selecting the pipeline, enabling or disabling certain methods, setting parameters.

-op **--optimization-pipeline** `[basic | advanced | full]`
Pre-configured combinations of collection optimization methods.

-on **--optimization-on** `<optimization name>`
Enable the specified collection optimization method.

-off **--optimization-off** `<optimization name>`
Disable the specified collection optimization method.

-oa **--optimization-args** `<parameter name> <parameter value>`
Set parameter values for various optimizations.

--optimization-cache-off `<flag>`
Ignore cached optimization data (e.g., cached call graph).

--optimization-reset-cache `<flag>`
Remove the cached optimization resources and data.

Table A.3: The CLI of `stats` module consisting of several commands and numerous options. Note that the `stats` module has an internal Python API as well, designed primarily for more advanced usage by the developers.

Command: list-files

Show stat files stored in the stats directory (.perun/stats/). This command shows only a limited number of the most recent files by default. This can be, however, changed by the --top and --from-minor options.
The default output format is 'file size | minor version | file name'.

-N **--top** `<int>`
Show only stat files from top N minor versions. Show all results if set to 0. The minor version to start at can be changed using --from-minor. [default: 20]

-m **--from-minor** `<minor version>`
Show stat files starting from a certain minor version (default is HEAD).

-i **--no-minor** `<flag>`
Do not show the minor version headers in the output.

-f **--no-file-size** `<flag>`
Do not show the size of each stat file.

-t **--no-total-size** `<flag>`
Do not show the total size of all the stat files combined.

-s **--sort-by-size** `<flag>`
Sort the files by size instead of the minor versions order.

Command: list-versions
Show minor versions stored as directories in the stats directory (.perun/stats/). This command shows only a limited number of the most recent versions by default. This can be, however, changed by the --top and --from-minor options.
The default output format is 'directory size | minor version | file count'.

-N **--top** `<int>`
Show only top N minor versions. Show all versions if set to 0. The minor version to start at can be changed using --from-minor. [default: 20]

-m **--from-minor** `<minor version>`
Show minor versions starting from a certain minor version (default is HEAD).

-d **--no-dir-size** `<flag>`
Do not show the size of the version directory.

-f **--no-file-count** `<flag>`
Do not show the number of files in each version directory.

-t **--no-total-size** `<flag>`
Do not show the total size of all the versions combined.

-s **--sort-by-size** `<flag>`
Sort the versions by size instead of the minor versions order.

Command: sync
Synchronizes the actual contents of the stats directory with the internal 'index' file. The synchronization should be needed only rarely - mainly in cases when the stats directory has been manually tampered with and some files or directories were created or deleted by a user.

Command: clean
Cleans the stats directory by synchronizing the internal state, deleting distinguishable custom files and directories (i.e. not all the custom made or manually created files / directories can be identified as custom, e.g. when they comply the correct format etc.) and by removing the empty minor version directories.

-c **--keep-custom** `<flag>`
The custom stats directories will not be removed.

-e **--keep-empty** `<flag>`
The empty version directories will not be removed.

Command: delete file <NAME>
Deletes a stat file in either specific minor version or across all the minor versions in the stats directory.

-m **--in-minor** `<minor version>`
Delete the stats file in the specified minor version (HEAD if not specified) or across all the minor versions if set to ‘.

-k **--keep-directory** `<flag>`
Possibly empty directory of minor version will be kept in the file system.

Command: delete minor <VERSION>
Deletes the specified minor version directory in stats with all its content.

-k **--keep-directory** `<flag>`
Resulting empty directory of minor version will be kept in the file system.

Command: delete all
Deletes the whole content of the stats directory.

-k **--keep-directory** `<flag>`
Resulting empty directory of minor version will be kept in the file system.

Listing A.1: An example of a SystemTap script code featuring five probes, two functions and enabled sampling. Each function has a matching pair of entry (`call`) and exit (`return`) probes that measure the duration of each non-sampled function call. Note that the sampling is implemented using a global array that is initialized during the startup of the profiled program (the `begin` probe).

```
global sample_array[2]

probe process("ccsds/phase0/compress").begin {
    sample_array[0] = 19
    sample_array[1] = 19

    printf("begin ccsds/phase0/compress\n")
}

probe process("ccsds/phase0/compress").function("BitDepthAC").call? {
    sample_array[0] ++
    if (sample_array[0] == 20) {
        printf("0 %sBitDepthAC\n", thread_indent(1))
        sample_array[0] = 0
    }
}
probe process("ccsds/phase0/compress").function("BitDepthAC").return? {
    if (sample_array[0] == 0) {
        printf("1 %sBitDepthAC\n", thread_indent(-1))
    }
}

probe process("ccsds/phase0/compress").function("BitDepthAC_Block").call? {
    sample_array[1] ++
    if (sample_array[1] == 20) {
        printf("0 %sBitDepthAC_Block\n", thread_indent(1))
```

```
        sample_array[1] = 0
    }
}
probe process("ccsds/phase0/compress").function("BitDepthAC_Block").return? {
    if (sample_array[1] == 0) {
        printf("1 %sBitDepthAC_Block\n", thread_indent(-1))
    }
}
```

Listing A.2: An example of an eBPF collection program featuring two matching probes for the entry and exit points of `DitDepthAC` function. Similarly to the SystemTap script, the probe output is sampled. However, compared to the SystemTap, the eBPF program is more complicated and requires usage of more internal data structures (`BPF_ARRAY`). Moreover, the probe output is done by sending a `duration_data` structure with the raw data (the function *id*, program *pid*, *entry_ns* time of the entry probe event and *exit_ns* time of the exit probe event) through the `BPF_PERF_OUTPUT` which is then handled by the Python interface and can be polled by the developer.

```
#include <linux/sched.h>     // for TASK_COMM_LEN
#include <uapi/linux/bpf_perf_event.h>

struct duration_data {
    u32 id;
    u32 pid;
    u64 entry_ns;
    u64 exit_ns;
    char comm[TASK_COMM_LEN];
};

BPF_ARRAY(timestamps, u64, 1);
// timed sampling switch omitted
BPF_ARRAY(sampling, u32, 1);
BPF_PERF_OUTPUT(records);

int entry_BitDepthAC(struct pt_regs *ctx)
{
    // timed sampling code omitted
    u32 id = 0;

    u32 *sample = sampling.lookup(&id);
    if (sample == NULL) {
        return 0;
    }

    if (*sample == 0) {
        u64 entry_timestamp = bpf_ktime_get_ns();
        timestamps.update(&id, &entry_timestamp);
    }

    (*sample)++;
    if (*sample == 20) {
        (*sample) = 0;
    }

    return 0;
}
```

```
int exit_BitDepthAC(struct pt_regs *ctx)
{
    // timed sampling code omitted
    u64 exit_timestamp = bpf_ktime_get_ns();
    u32 id = 0;

    u64 *entry_timestamp = timestamps.lookup(&id);
    if (entry_timestamp == NULL || *entry_timestamp == 0) {
        return 0;
    }

    struct duration_data data = {};
    data.id = id;
    data.pid = bpf_get_current_pid_tgid();
    data.entry_ns = *entry_timestamp;
    data.exit_ns = exit_timestamp;

    (*entry_timestamp) = 0;

    bpf_get_current_comm(&data.comm, sizeof(data.comm));
    records.perf_submit(ctx, &data, sizeof(data));

    return 0;
}
```

Listing A.3: An example of a SystemTap raw data output that contains the records generated by injected probes. Specifically each probe record contains the probe type (e.g., *function entry* = 0, *function exit* = 1, etc.), number of microseconds elapsed from the top-most function entry (e.g., `frame_dummy` or `main`), the process *name*, *pid* and lastly, the function *name*.

```
begin ccsds/phase0/compress
0       0 compress(17462):frame_dummy
0      23 compress(17462): register_tm_clones
1      30 compress(17462): register_tm_clones
1      32 compress(17462):frame_dummy
0       0 compress(17462):main
0       6 compress(17462): frame_load_pgm
0    2511 compress(17462):  stream_skip_comment
1    2520 compress(17462):  stream_skip_comment
0    2526 compress(17462):  stream_skip_comment
1    2531 compress(17462):  stream_skip_comment
0    2535 compress(17462):  stream_skip_comment
1    2540 compress(17462):  stream_skip_comment
0    2544 compress(17462):  stream_skip_comment
1    2549 compress(17462):  stream_skip_comment
0    2553 compress(17462):  ceil_multiple8
1    2559 compress(17462):  ceil_multiple8
0    2563 compress(17462):  ceil_multiple8
1    2568 compress(17462):  ceil_multiple8
0    2584 compress(17462):  frame_read_pgm_data
0    2590 compress(17462):   ceil_multiple8
1    2596 compress(17462):   ceil_multiple8
0    2599 compress(17462):   ceil_multiple8
1    2605 compress(17462):   ceil_multiple8
1    3777 compress(17462):  frame_read_pgm_data
1    3786 compress(17462): frame_load_pgm
```

```
0    3789 compress(17462): frame_dump
```

Listing A.4: An example of a raw data output from the eBPF engine that contains records captured and stored by the Python eBPF process. Unlike the SystemTap, each line represents a paired function entry and exit record: the process *pid*, the function *id* (instead of a name), the *entry* point timestamp (in nanoseconds) and the inclusive *duration* of the function call (in *ns*).

```
18620 134 33156233850694 3815821
18620 128 33156237677393 619467
18620 148 33156238302007 4357
18620 98 33156238314380 9295248
18620 90 33156238310275 9303443
18620 129 33156248045668 553684
18620 87 33156248722397 2920
18620 135 33156247618424 4074280
18620 146 33156251699577 19552
18620 8 33156251732156 2991
18620 79 33156251751281 9583
18620 87 33156251809147 2458
18620 87 33156251879755 2023
18620 87 33156251948360 2035
```

Listing A.5: An example of a Perun profile resource obtained from transforming the raw data of one particular record into a profile structure. Specifically, `amount` represents the elapsed time of the first (`call-order`) function call (`uid`).

```
{
  'amount': 2920,
  'uid': 'BitDepthAC_Block',
  'type': 'mixed',
  'subtype': 'time delta',
  'workload': 'ccsds/phase0/data/Lenna.pgm',
  'thread': 18620,
  'call-order': 0,
}
```

Listing A.6: An example of a runtime eBPF configuration passed to the eBPF collection process spawned with elevated privileges. Generally, the configuration contains parameters required by the process to correctly inject the instrumentation probes (e.g., the `func` dictionary), run the profiled command (e.g., the `command`, `program_file`, `timeout` and others), apply the *run-optimize* methods (`optimizations` and `optimization_params`) and store the raw data output (`data_file`).

```
"binary": "ccsds/phase0/compress",
"command": "ccsds/phase0/compress ./phase0/data/Lenna.pgm",
"optimizations": ['dynamic-probing'],
"timeout": null,
"program_file": "ccsds/.perun/tmp/trace/files/collect_program_2020-05-26-23-35-35_18
    525.c",
"data_file": "ccsds/.perun/tmp/trace/files/collect_data_2020-05-26-23-35-35_18525.
    txt",
"optimization_params": {
  "probing-threshold": 100000,
  "timed-sample-freq": 1,
  "probing-reattach": false
```

```
},
"func": {
  "BitDepthAC": {
    "type": "F",
    "sample_index": 0,
    "sample": 20,
    "name": "BitDepthAC",
    "lib": null,
    "pair": "BitDepthAC",
    "id": 0
  },
  "BitDepthAC_Block": {
    "type": "F",
    "sample_index": 1,
    "sample": 20,
    "name": "BitDepthAC_Block",
    "lib": null,
    "pair": "BitDepthAC_Block",
    "id": 1
  },
}
```

Listing A.7: The documentation for Tracer `Configuration` class containing the necessary Tracer parameters, as well as implementing the interface for the optimization module consisting of `get_functions`, `prune_functions`, `get_target` and `get_stats_name` functions.

```
class Configuration:
  """ A class that stores the Tracer configuration provided by the CLI.

  :ivar Probes probes: the collection probes configuration
  :ivar bool keep_temps: keep the temporary files after the collection is
      finished
  :ivar bool zip_temps: zip and store the temporary files before they are
      deleted
  :ivar bool verbose_trace: the raw performance data collected will be more
      verbose
  :ivar bool quiet: the collection progress output will be less verbose
  :ivar bool watchdog: enables detailed logging during the collection
  :ivar bool diagnostics: enables detailed surveillance mode of the collector
  :ivar OutputHandling output_handling: store or discard the profiling
      command stdout and stderr
  :ivar CollectEngine engine: the collection engine to be used, e.g.
      SystemTap or eBPF
  :ivar float or None timeout: the timeout for the profiled command or None
      if indefinite
  :ivar str binary: the path to the binary file to be probed
  :ivar Executable executable: the Executable object containing the profiled
      command, args, etc.
  :ivar str timestamp: the time of the collection start
  :ivar int pid: the PID of the Tracer process
  :ivar str files_dir: the directory path of the temporary files
  :ivar str locks_dir: the directory path of the lock files
  """
def __init__(self, executable, **cli_config):
  """ Constructs the Configuration object from the supplied CLI configuration
  """

def engine_factory(self):
  """ Instantiates the engine object based on the string representation.
  """
```

```
def get_functions(self):
  """ Access the configuration of the function probes
  """

def prune_functions(self, remaining):
  """ Remove function probes not present in the 'remaining' set from the
      instrumentation
  """

def get_target(self):
  """ Obtain the target executable file.
  """

def get_stats_name(self, specifier=None):
  """ Create a 'stats' file name based on the specifier.
  """
```

Table A.4: A list of the *optimization parameters* that can be supplied by the `-oa` CLI option to change the parameters of various optimization methods. For each parameter, we list the corresponding method, default value and the prediction process, if any.

Constants $functions\_keep\_leaves = 20$ $top\_static\_ratio = 0.1$ $top\_prune\_ratio = 0.1$ $top\_strict\_ratio = 0.25$ $top\_soft\_ratio = 0.5$ $default\_keep\_top = 1$ $chain\_length\_ratio = 0.1$ $default\_chain\_length = 1$ $default\_min\_levels = 2$ $min\_functions\_ratio = 0.1$ $default\_min\_functions = 10$ $default\_sampling\_step = 2$ $threshold\_soft\_base = 10000$ $threshold\_strict\_base = 1000$ $hard\_threshold\_coefficient = 100$ $probing\_threshold = 100000$ $probing\_reattach\_coefficient = 0.2$
diff-keep-leaf `Methods:` Diff Tracing `Default:` `False` `Predict:` `True` $\Leftrightarrow |probed\_functions| <= functions\_keep\_leaves$
diff-inspect-all `Methods:` Diff Tracing `Default:` `True`
diff-cfg-mode `Methods:` Diff Tracing `Default:` `Semistrict`

`Predict:`	`Soft` $\Leftrightarrow pipeline = Basic$ `Semistrict` $\Leftrightarrow pipeline = Advanced$ `Strict` $\Leftrightarrow pipeline = Full$

source-files

`Methods:`	Static Baseline
`Default:`	`.c` files extracted from the directory (recursively) containing the profiled executable.

source-dirs

`Methods:`	Static Baseline
`Default:`	The directory containing the profiled executable.

static-complexity

`Methods:`	Static Baseline
`Default:`	`Constant`

static-keep-top

`Methods:`	Static Baseline
`Default:`	$default\_keep\_top$
`Predict:`	$max(cg.depth * top\_static\_ratio, default\_keep\_top)$

cg-mode

`Methods:`	Call Graph Shaping
`Default:`	`Match`
`Predict:`	`Strict` $\Leftrightarrow pipeline = Basic$ `Soft` $\Leftrightarrow pipeline = Advanced$ `Prune` $\Leftrightarrow pipeline = Full$

cg-keep-top

`Methods:`	Call Graph Shaping
`Default:`	$default\_min\_levels$
`Predict:`	$max(cg.depth * top\_prune\_ratio, default\_min\_levels) \Leftrightarrow cg\_mode = Prune$ $max(cg.depth * top\_strict\_ratio, default\_min\_levels) \Leftrightarrow cg\_mode = Strict$ $max(cg.depth * top\_soft\_ratio, default\_min\_levels) \Leftrightarrow cg\_mode = Soft$

cg-trim-min-functions

`Methods:`	Call Graph Trimming
`Default:`	$default\_min\_functions$
`Predict:`	$max(\|probed\_functions\| * min\_functions\_ratio, default\_min\_functions)$

cg-trim-keep-leaf

`Methods:`	Call Graph Trimming
`Default:`	`False`
`Predict:`	`True` $\Leftrightarrow \|probed\_functions\| <= functions\_keep\_leaves$

cg-prune-chain-length

`Methods:`	Call Graph Pruning
`Default:`	$default\_chain\_length$
`Predict:`	$max(cg.depth * chain\_length\_ratio, default\_chain\_length)$

dyn-sample-step

`Methods:`	Dynamic Sampling

`Default:` $default\_sampling\_step$

---

dyn-sample-threshold

`Methods:` Dynamic Sampling
`Default:` $threshold\_soft\_base$
`Predict:` $threshold\_strict\_base \Leftrightarrow threshold\_mode = Strict$

---

probing-threshold

`Methods:` Dynamic Probing
`Default:` $probing\_threshold$
`Predict:` $probing\_threshold * probing\_reattach\_coefficient \Leftrightarrow probing\_reattach = True$

---

probing-reattach

`Methods:` Dynamic Probing
`Default:` `False`

---

timed-sample-freq

`Methods:` Timed Sampling
`Default:` `1`

---

dyn-base-soft-threshold

`Methods:` Dynamic Baseline
`Default:` $threshold\_soft\_base$
`Predict:` $threshold\_strict\_base \Leftrightarrow threshold\_mode = Strict$

---

dyn-base-hard-threshold

`Methods:` Dynamic Baseline
`Default:` $threshold\_soft\_base * hard\_threshold\_coefficient$
`Predict:` $threshold\_strict\_base * hard\_threshold\_coefficient \Leftrightarrow threshold\_mode = Strict$

---

threshold-mode

`Methods:` Dynamic Sampling, Dynamic Baseline
`Default:` `Soft`
`Predict:` `Strict` $\Leftrightarrow pipeline = Basic$
`Soft` $\Leftrightarrow pipeline = Advanced$
`Soft` $\Leftrightarrow pipeline = Full$

Appendix B

Evaluation Tables

Here we present the tabulated results that were obtained in the evaluation phase. The results are divided into separate Tables according to the *test projects*, *collector configuration* and *experiment cases* grouped by the methods (see Section 8.1). Note that the column names refer to the evaluation metrics introduced in Section 5.1, where **FC [c, l, lg, q, p, e]** represents the amount of functions classified as *constant*, *linear*, *logarithmic*, *quadratic*, *power* or *exponential* by the regression analysis. Similarly, **FC [u]** counts the number of functions classified as *unclear*, i.e., where the coefficient of determination: $r^2 < 0.5$ for all supported model types.

Moreover, note that the **TLC** and **HC** values can, in some situations, exceed 100% or reach exactly 0%, respectively. The reason being a SystemTap glitch that causes raw data corruption in the output file — specifically, a duplicated blocks of records — when the collection records are generated too fast for SystemTap to handle. Thus, certain functions have more profiling records than actual function calls, resulting in skewed total time computation and, subsequently, invalid coverage percentage.

The Tables are presented in the following order:

1. `CPython:` $\{stap, cache\_off\}$
2. `CPython:` $\{stap, cache\_on\}$
3. `CCSDS:` $\{stap, cache\_off\}$
4. `CCSDS:` $\{stap, cache\_on\}$
5. `CCSDS:` $\{ebpf, cache\_off\}$
6. `CCSDS:` $\{ebpf, cache\_on\}$

where each segment contains five to eight tables according to the number of evaluated methods (note that not all configurations are able to run every method due to some limitations imposed by certain methods).

`CPython`: SystemTap — cache off

Table B.1: The evaluation results of *Dynamic Sampling* experiment cases under the $\{cpython, stap, cache\_off\}$ configuration.

	PT [*s*]	**CPT** [*s*]	**PRT** [*s*]	**RDV** [*MB*]	**PS** [*MB*]	**OO** [*s*]	**ORE** [*s*]	**PL** [*#*]	**PLR** [*#*]	**HC** [*%*]	**HCD** [*level*]	**HCP** [*#*]	**TLC** [*%*]	**TLCP** [*#*]	**FC [c, l, lg, q, p, e]** [*#*]	**FC [u]** [*#*]
no-opt	1202.7	1090.7	9.3	1272.8	10.7	0.0	0.0	6397	1931	0.0	0.0	0	0.0	0	[666, 1, 26, 234, 34, 31]	175
ds:d:1	230.8	182.2	2.9	0.2	0.4	47.7	47.2	1781	1149	99.8	84.9	162	100.0	2	[1, 0, 0, 2, 0, 0]	0
ds:d:2	224.2	175.1	3.0	25.0	1.0	47.6	47.1	1781	1149	98.4	84.9	162	100.0	2	[47, 0, 1, 19, 3, 1]	18
ds:d:10:1	232.5	183.9	2.9	0.1	0.4	47.6	47.1	1781	1149	99.8	84.9	162	100.0	2	[0, 0, 0, 0, 0, 0]	0
ds:d:10:2	230.6	182.3	2.9	0.1	0.4	47.6	47.1	1781	1149	99.8	84.9	162	100.0	2	[0, 0, 0, 0, 0, 0]	0
ds:d:1.1:1	232.7	182.9	2.9	13.6	0.7	47.9	47.4	1781	1149	84.3	84.5	163	100.0	2	[74, 0, 7, 32, 4, 1]	14
ds:d:1.1:2	216.7	163.5	3.4	137.9	2.3	47.7	47.0	1781	1149	86.5	84.5	163	100.0	2	[268, 2, 8, 86, 19, 10]	54
ds:s:1	216.8	150.4	4.7	472.4	6.5	48.3	47.5	1781	1149	10.9	84.5	163	100.0	2	[474, 0, 11, 119, 18, 20]	88
ds:s:2	217.5	150.6	4.7	472.8	6.5	48.4	47.6	1781	1149	14.9	84.5	163	100.0	2	[475, 1, 9, 117, 20, 19]	89
ds:s:3	218.0	151.4	4.7	472.5	6.5	48.5	47.6	1781	1149	19.2	84.5	163	100.0	2	[475, 1, 11, 117, 18, 19]	87
ds:r:1	205.6	150.9	3.5	171.2	2.3	48.1	47.5	1781	1149	0.0	84.1	164	100.0	2	[490, 1, 10, 116, 20, 12]	83
ds:r:2	206.0	151.9	3.5	173.1	2.3	48.2	47.5	1781	1149	78.5	84.5	163	100.0	2	[487, 0, 11, 113, 22, 14]	83
ds:r:3	206.0	151.9	3.6	171.3	2.3	48.1	47.5	1781	1149	0.0	84.1	164	100.0	2	[491, 0, 11, 113, 19, 15]	83

Table B.2: The evaluation results of *Call Graph Shaping* experiment cases under the $\{cpython, stap, cache\_off\}$ configuration.

	PT [*s*]	**CPT** [*s*]	**PRT** [*s*]	**RDV** [*MB*]	**PS** [*MB*]	**OO** [*s*]	**ORE** [*s*]	**PL** [*#*]	**PLR** [*#*]	**HC** [*%*]	**HCD** [*level*]	**HCP** [*#*]	**TLC** [*%*]	**TLCP** [*#*]	**FC [c, l, lg, q, p, e]** [*#*]	**FC [u]** [*#*]
no-opt	1202.7	1090.7	9.3	1272.8	10.7	0.0	0.0	6397	1931	0.0	0.0	0	0.0	0	[666, 1, 26, 234, 34, 31]	175
cg:m	271.1	129.3	5.7	716.8	9.7	48.5	47.6	1781	1149	31.4	84.5	163	100.0	2	[452, 1, 8, 147, 21, 18]	92
cg:p	137.1	75.4	2.4	224.8	4.9	48.2	47.3	1217	838	83.7	31.9	90	100.0	2	[285, 3, 5, 91, 9, 11]	47
cg:p-1	206.3	112.7	3.6	448.3	6.3	48.7	47.8	1634	1046	65.5	46.8	104	100.0	2	[391, 1, 10, 121, 17, 22]	70
cg:p-max	87.3	35.6	0.6	19.8	1.6	48.4	47.6	622	485	87.0	17.3	83	100.0	2	[109, 1, 3, 37, 4, 3]	15
cg:t-s	108.0	55.8	0.8	22.8	2.1	48.2	47.9	977	711	77.8	37.9	53	100.0	2	[247, 1, 8, 66, 12, 5]	35
cg:t-s:l	122.2	67.6	1.7	50.2	3.4	48.1	47.7	1139	828	66.9	43.5	141	100.0	2	[287, 0, 6, 87, 15, 7]	54
cg:t-r	87.9	36.6	0.7	17.6	1.6	48.0	47.8	648	519	76.3	36.0	67	100.0	2	[129, 1, 5, 45, 8, 2]	22
cg:t-r:l	97.2	44.3	1.1	34.7	2.6	48.0	47.7	765	607	70.6	31.1	128	100.0	2	[159, 1, 4, 53, 10, 2]	31

`CPython`: SystemTap — cache off

Table B.3: The evaluation results of *Dynamic Baseline* experiment cases under the $\{cpython, stap, cache\_off\}$ configuration.

	PT [s]	CPT [s]	PRT [s]	RDV [MB]	PS [MB]	OO [s]	ORE [s]	PL [#]	PLR [#]	HC [%]	HCD [level]	HCP [#]	TLC [%]	TLCP [#]	FC [c, l, lg, q, p, e] [#]	FC [u] [#]
no-opt	1202.7	1090.7	9.3	1272.8	10.7	0.0	0.0	6397	1931	0.0	0.0	0	0.0	0	[666, 1, 26, 234, 34, 31]	175
db:s:1	196.6	122.6	3.6	401.3	5.5	48.6	47.7	1746	1114	47.9	83.0	158	100.0	2	[444, 1, 13, 129, 17, 15]	84
db:s:2	186.5	121.5	2.9	330.4	4.4	48.5	47.8	1739	1107	60.3	82.0	157	100.0	2	[446, 1, 12, 120, 18, 21]	81
db:s:3	187.2	121.0	2.7	293.9	4.0	48.5	47.7	1736	1104	72.1	79.6	160	100.0	2	[458, 1, 10, 116, 17, 22]	79
db:r:1	178.6	116.8	2.3	273.0	3.9	47.9	47.3	1698	1066	37.9	75.7	147	100.0	2	[427, 2, 13, 114, 17, 19]	75
db:r:2	165.9	112.2	1.2	127.2	2.0	48.3	47.8	1669	1037	32.3	73.1	151	100.0	2	[406, 1, 10, 110, 17, 20]	69
db:r:3	162.2	110.7	0.6	61.9	1.2	48.3	47.7	1652	1019	34.9	72.9	160	100.0	2	[382, 0, 10, 124, 14, 25]	55

Table B.4: The evaluation results of *Pipeline* experiment cases under the $\{cpython, stap, cache\_off\}$ configuration.

	PT [s]	CPT [s]	PRT [s]	RDV [MB]	PS [MB]	OO [s]	ORE [s]	PL [#]	PLR [#]	HC [%]	HCD [level]	HCP [#]	TLC [%]	TLCP [#]	FC [c, l, lg, q, p, e] [#]	FC [u] [#]
no-opt	1202.7	1090.7	9.3	1272.8	10.7	0.0	0.0	6397	1931	0.0	0.0	0	0.0	0	[666, 1, 26, 234, 34, 31]	175
p:d:b	87.6	36.6	0.6	17.6	1.6	47.5	47.2	648	519	76.4	36.0	67	100.0	2	[128, 1, 4, 48, 6, 2]	23
p:b	85.0	35.6	0.4	9.2	0.9	47.5	47.2	616	487	60.8	35.6	66	100.0	2	[115, 1, 3, 44, 6, 1]	21
p:d:a	152.2	103.1	1.3	0.1	0.3	48.4	47.0	1208	831	99.8	48.7	55	100.0	2	[0, 0, 0, 1, 0, 0]	0
p:a	136.8	82.4	1.5	38.4	2.9	48.9	47.5	1198	821	92.6	49.3	56	100.0	2	[317, 1, 6, 75, 13, 6]	57
p:d:f	188.3	133.9	1.7	0.1	0.3	53.6	47.0	1439	953	99.8	36.9	93	100.0	2	[1, 0, 0, 1, 0, 0]	0
p:f	178.5	106.6	2.5	270.8	3.8	54.1	47.3	1425	939	70.0	36.9	93	100.0	2	[348, 0, 9, 89, 15, 15]	61

Table B.5: The evaluation results of *Diff Tracing* experiment cases under the $\{cpython, stap, cache\_off\}$ configuration.

	PT [s]	CPT [s]	PRT [s]	RDV [MB]	PS [MB]	OO [s]	ORE [s]	PL [#]	PLR [#]	HC [%]	HCD [level]	HCP [#]	TLC [%]	TLCP [#]	FC [c, l, lg, q, p, e] [#]	FC [u] [#]
no-opt	1202.7	1090.7	9.3	1272.8	10.7	0.0	0.0	6397	1931	0.0	0.0	0	0.0	0	[666, 1, 26, 234, 34, 31]	175
dt	98.3	42.3	1.5	47.6	3.7	48.8	47.5	729	501	87.3	47.8	39	100.0	2	[191, 1, 3, 40, 6, 5]	33
dt:l	104.6	46.9	1.9	61.6	4.8	49.0	47.6	790	544	41.6	54.4	62	100.0	2	[204, 0, 2, 47, 6, 5]	38
dt:i	89.0	35.1	1.1	32.1	2.5	49.1	47.8	591	403	68.9	42.9	34	99.9	3	[143, 1, 3, 37, 2, 5]	28
dt:s	98.3	42.3	1.5	47.6	3.7	49.0	47.7	729	501	87.3	47.8	39	100.0	2	[192, 1, 3, 40, 5, 5]	32
dt:r	98.1	42.3	1.5	48.3	3.7	49.0	47.7	729	501	87.3	47.8	39	100.0	2	[191, 0, 4, 42, 7, 5]	33

`CPython`: SystemTap — cache on

Table B.6: The evaluation results of *Dynamic Sampling* experiment cases under the $\{cpython, stap, cache\_on\}$ configuration.

	PT [*s*]	**CPT** [*s*]	**PRT** [*s*]	**RDV** [*MB*]	**PS** [*MB*]	**OO** [*s*]	**ORE** [*s*]	**PL** [#]	**PLR** [#]	**HC** [%]	**HCD** [*level*]	**HCP** [#]	**TLC** [%]	**TLCP** [#]	**FC [c, l, lg, q, p, e]** [#]	**FC [u]** [#]
no-opt	201.6	97.5	10.1	1250.1	10.7	0.0	0.0	6397	1921	0.0	0.0	0	0.0	0	[651, 1, 30, 197, 40, 23]	190
ds:d:1	17.3	14.8	2.9	0.2	0.4	1.6	1.1	1781	1149	99.8	84.9	162	100.0	2	[1, 0, 0, 1, 0, 0]	0
ds:d:2	18.8	15.2	3.1	25.0	1.0	1.7	1.2	1781	1149	98.4	84.9	162	100.0	2	[48, 0, 1, 19, 1, 0]	18
ds:d:10:1	17.0	14.7	2.9	0.1	0.4	1.6	1.1	1781	1149	99.8	84.9	162	100.0	2	[0, 0, 0, 0, 0, 0]	0
ds:d:10:2	17.1	14.7	2.9	0.1	0.4	1.6	1.2	1781	1149	99.8	84.9	162	100.0	2	[0, 0, 0, 0, 0, 0]	0
ds:d:1.1:1	18.1	14.7	3.0	13.5	0.7	1.6	1.1	1781	1149	84.5	84.5	163	100.0	2	[72, 0, 5, 35, 3, 1]	14
ds:d:1.1:2	22.7	15.4	3.6	137.6	2.3	1.9	1.2	1781	1149	86.5	84.5	163	100.0	2	[264, 0, 10, 83, 16, 12]	55
ds:s:1	37.5	17.8	5.0	471.9	6.5	2.0	1.2	1781	1149	12.3	84.5	163	100.0	2	[455, 0, 9, 129, 20, 24]	93
ds:s:2	37.6	17.5	5.0	472.2	6.5	2.0	1.2	1781	1149	13.5	84.5	163	100.0	2	[476, 0, 16, 116, 20, 20]	93
ds:s:3	37.5	17.4	5.0	471.9	6.5	2.0	1.2	1781	1149	15.7	84.5	163	100.0	2	[463, 1, 12, 118, 19, 17]	89
ds:r:1	23.8	15.8	3.7	171.2	2.3	1.8	1.2	1781	1149	0.0	84.1	164	100.0	2	[477, 1, 8, 120, 18, 15]	84
ds:r:2	24.0	16.0	3.7	173.1	2.3	1.8	1.2	1781	1149	75.8	84.5	163	100.0	2	[478, 1, 10, 112, 22, 18]	86
ds:r:3	23.9	16.0	3.7	171.3	2.3	1.8	1.2	1781	1149	0.0	84.1	164	100.0	2	[477, 0, 12, 128, 22, 14]	85

Table B.7: The evaluation results of *Call Graph Shaping* experiment cases under the $\{cpython, stap, cache\_on\}$ configuration.

	PT [*s*]	**CPT** [*s*]	**PRT** [*s*]	**RDV** [*MB*]	**PS** [*MB*]	**OO** [*s*]	**ORE** [*s*]	**PL** [#]	**PLR** [#]	**HC** [%]	**HCD** [*level*]	**HCP** [#]	**TLC** [%]	**TLCP** [#]	**FC [c, l, lg, q, p, e]** [#]	**FC [u]** [#]
no-opt	201.6	97.5	10.1	1250.1	10.7	0.0	0.0	6397	1921	0.0	0.0	0	0.0	0	[651, 1, 30, 197, 40, 23]	190
cg:m	88.5	17.6	6.5	716.7	9.7	2.0	1.1	1781	1149	33.1	84.5	163	100.0	2	[467, 2, 13, 124, 18, 15]	102
cg:p	24.4	10.8	2.7	225.5	4.9	2.1	1.1	1217	838	84.1	31.9	90	100.0	2	[284, 1, 7, 84, 9, 11]	48
cg:p-1	51.8	15.1	4.1	448.2	6.3	2.2	1.1	1634	1046	66.8	46.8	104	100.0	2	[404, 2, 10, 103, 17, 15]	81
cg:p-max	9.8	4.6	0.7	20.1	1.6	2.0	1.1	622	485	87.4	17.3	83	100.0	2	[112, 1, 2, 33, 5, 2]	13
cg:t-s	12.0	6.7	0.8	23.2	2.1	1.4	1.1	977	711	77.9	37.9	53	100.0	2	[245, 1, 7, 68, 12, 6]	36
cg:t-s:l	15.4	7.5	1.6	49.5	3.4	1.6	1.1	1139	828	66.8	43.5	141	100.0	2	[291, 1, 9, 79, 12, 7]	51
cg:t-r	9.2	4.6	0.7	17.3	1.6	1.4	1.1	648	519	76.7	36.0	67	100.0	2	[124, 0, 4, 46, 7, 2]	23
cg:t-r:l	13.5	7.3	1.1	34.7	2.6	1.4	1.1	765	607	70.7	31.1	128	100.0	2	[153, 1, 7, 58, 9, 2]	32

`CPython`: SystemTap — cache on

Table B.8: The evaluation results of *Dynamic Baseline* experiment cases under the $\{cpython, stap, cache\_on\}$ configuration.

	PT [*s*]	**CPT** [*s*]	**PRT** [*s*]	**RDV** [*MB*]	**PS** [*MB*]	**OO** [*s*]	**ORE** [*s*]	**PL** [#]	**PLR** [#]	**HC** [%]	**HCD** [*level*]	**HCP** [#]	**TLC** [%]	**TLCP** [#]	**FC [c, l, lg, q, p, e]** [#]	**FC [u]** [#]
no-opt	201.6	97.5	10.1	1250.1	10.7	0.0	0.0	6397	1921	0.0	0.0	0	0.0	0	[651, 1, 30, 197, 40, 23]	190
db:s:1	41.8	14.0	3.5	401.1	5.5	2.0	1.2	1746	1114	47.3	83.0	158	100.0	2	[462, 2, 13, 121, 17, 15]	83
db:s:2	131.9	112.6	2.7	330.5	4.4	1.9	1.2	1739	1107	61.5	82.0	157	100.0	2	[465, 1, 12, 116, 17, 19]	77
db:s:3	131.8	113.0	2.7	293.9	4.0	1.9	1.2	1737	1105	72.5	79.6	160	100.0	2	[472, 2, 11, 99, 18, 14]	84
db:r:1	29.5	12.7	2.3	277.3	4.0	1.8	1.2	1700	1068	36.7	76.8	148	100.0	2	[420, 1, 12, 114, 20, 21]	73
db:r:2	111.3	104.2	1.1	125.0	2.0	1.7	1.2	1669	1037	43.1	73.1	147	100.0	2	[398, 0, 10, 121, 20, 19]	66
db:r:3	107.5	102.6	0.7	63.0	1.3	1.7	1.2	1653	1021	36.9	72.9	160	100.0	2	[370, 1, 7, 126, 16, 28]	58

Table B.9: The evaluation results of *Pipeline* experiment cases under the $\{cpython, stap, cache\_on\}$ configuration.

	PT [*s*]	**CPT** [*s*]	**PRT** [*s*]	**RDV** [*MB*]	**PS** [*MB*]	**OO** [*s*]	**ORE** [*s*]	**PL** [#]	**PLR** [#]	**HC** [%]	**HCD** [*level*]	**HCP** [#]	**TLC** [%]	**TLCP** [#]	**FC [c, l, lg, q, p, e]** [#]	**FC [u]** [#]
no-opt	201.6	97.5	10.1	1250.1	10.7	0.0	0.0	6397	1921	0.0	0.0	0	0.0	0	[651, 1, 30, 197, 40, 23]	190
p:d:b	9.3	4.6	0.7	17.6	1.6	1.4	1.1	648	519	77.4	36.0	67	100.0	2	[127, 1, 6, 44, 7, 1]	23
p:b	8.1	4.6	0.4	9.8	1.0	1.4	1.2	620	491	61.4	35.6	66	100.0	2	[118, 0, 4, 45, 8, 1]	21
p:d:a	12.3	9.1	1.3	0.1	0.3	2.5	1.1	1208	831	99.8	48.7	55	100.0	2	[0, 0, 0, 0, 0, 0]	0
p:a	17.5	9.3	1.6	38.4	2.9	2.7	1.2	1198	821	92.5	49.3	56	100.0	2	[315, 1, 8, 74, 15, 7]	60
p:d:f	20.6	11.8	1.8	0.1	0.3	8.2	1.1	1439	953	99.8	36.9	93	100.0	2	[1, 0, 0, 1, 0, 0]	0
p:f	45.4	11.0	2.7	278.9	3.9	8.5	1.2	1426	940	71.8	36.9	93	100.0	2	[338, 1, 7, 98, 16, 16]	65

Table B.10: The evaluation results of *Diff Tracing* experiment cases under the $\{cpython, stap, cache\_on\}$ configuration.

	PT [*s*]	**CPT** [*s*]	**PRT** [*s*]	**RDV** [*MB*]	**PS** [*MB*]	**OO** [*s*]	**ORE** [*s*]	**PL** [#]	**PLR** [#]	**HC** [%]	**HCD** [*level*]	**HCP** [#]	**TLC** [%]	**TLCP** [#]	**FC [c, l, lg, q, p, e]** [#]	**FC [u]** [#]
no-opt	201.6	97.5	10.1	1250.1	10.7	0.0	0.0	6397	1921	0.0	0.0	0	0.0	0	[651, 1, 30, 197, 40, 23]	190
dt	17.0	7.8	1.6	48.4	3.7	2.5	1.2	729	501	87.2	47.8	39	100.0	2	[185, 0, 2, 47, 7, 5]	34
dt:l	19.5	7.9	2.2	61.6	4.9	2.6	1.2	790	544	42.3	54.4	62	100.0	2	[203, 0, 3, 47, 8, 5]	38
dt:i	12.4	5.1	1.2	32.1	2.6	2.4	1.2	591	403	69.2	42.9	34	99.9	3	[143, 0, 3, 38, 4, 5]	27
dt:s	17.0	7.8	1.7	48.4	3.7	2.5	1.2	729	501	87.0	47.8	39	100.0	2	[186, 1, 3, 43, 4, 6]	35
dt:r	17.0	7.8	1.7	48.4	3.7	2.5	1.2	729	501	87.6	47.8	39	100.0	2	[188, 1, 2, 41, 6, 5]	36

CCSDS: SystemTap — cache off

Table B.11: The evaluation results of *Dynamic Sampling* experiment cases under the $\{ccsds, stap, cache\_off\}$ configuration.

	PT [s]	CPT [s]	PRT [s]	RDV [MB]	PS [MB]	OO [s]	ORE [s]	PL [#]	PLR [#]	HC [%]	HCD [level]	HCP [#]	TLC [%]	TLCP [#]	FC [c, l, lg, q, p, e] [#]	FC [u] [#]
no-opt	73.2	27.6	10.5	216.9	29.2	0.0	0.0	164	67	0.0	0.0	0	0.0	0	[32, 0, 0, 3, 0, 0]	5
ds:d:1	29.7	24.0	7.2	7.0	1.0	3.7	3.6	67	63	97.5	3.8	25	99.9	15	[29, 0, 0, 4, 1, 0]	3
ds:d:2	31.3	23.4	7.3	17.8	2.2	3.7	3.6	67	63	94.3	3.8	25	99.9	15	[32, 0, 0, 3, 0, 0]	5
ds:d:10:1	28.1	23.9	7.1	0.0	0.0	3.6	3.6	67	63	100.0	3.8	25	99.9	15	[4, 0, 0, 2, 0, 0]	0
ds:d:10:2	29.2	22.9	7.2	10.2	1.3	3.7	3.6	67	63	97.1	3.8	25	99.9	15	[32, 0, 0, 3, 0, 0]	5
ds:d:1.1:1	53.2	24.4	8.7	118.6	15.6	4.3	3.6	67	63	66.8	3.8	25	100.0	15	[32, 0, 0, 2, 0, 0]	5
ds:d:1.1:2	31.5	23.4	7.3	18.2	2.2	3.7	3.6	67	63	94.3	3.8	25	99.9	15	[32, 0, 0, 2, 0, 0]	5
ds:s:1	31.5	23.4	7.3	18.2	2.2	3.8	3.6	67	63	94.3	3.8	25	99.9	15	[32, 0, 0, 3, 0, 0]	5
ds:s:2	31.5	23.4	7.4	18.2	2.2	3.8	3.6	67	63	94.3	3.8	25	99.9	15	[32, 0, 0, 3, 0, 0]	5
ds:s:3	31.3	23.4	7.3	18.2	2.2	3.7	3.6	67	63	94.3	3.8	25	99.9	15	[32, 0, 0, 3, 0, 0]	5
ds:r:1	27.6	22.9	7.1	1.9	0.2	3.7	3.7	67	63	99.4	3.8	25	99.9	15	[32, 0, 0, 3, 0, 0]	5
ds:r:2	27.6	22.9	7.1	1.9	0.2	3.7	3.6	67	63	99.4	3.8	25	99.9	15	[32, 0, 0, 3, 0, 0]	5
ds:r:3	27.6	22.9	7.1	1.9	0.2	3.7	3.6	67	63	99.4	3.8	25	99.9	15	[32, 0, 0, 3, 0, 0]	5

Table B.12: The evaluation results of *Dynamic Baseline* experiment cases under the $\{ccsds, stap, cache\_off\}$ configuration.

	PT [s]	CPT [s]	PRT [s]	RDV [MB]	PS [MB]	OO [s]	ORE [s]	PL [#]	PLR [#]	HC [%]	HCD [level]	HCP [#]	TLC [%]	TLCP [#]	FC [c, l, lg, q, p, e] [#]	FC [u] [#]
no-opt	73.2	27.6	10.5	216.9	29.2	0.0	0.0	164	67	0.0	0.0	0	0.0	0	[32, 0, 0, 3, 0, 0]	5
db:s:1	22.9	16.4	0.6	13.7	1.3	3.7	3.6	53	49	33.6	2.8	28	99.1	15	[21, 0, 0, 3, 0, 0]	2
db:s:2	20.7	15.9	0.2	2.8	0.3	3.7	3.6	50	46	28.8	2.6	27	96.5	15	[19, 0, 0, 3, 0, 0]	0
db:s:3	20.7	15.9	0.2	2.8	0.3	3.7	3.6	50	46	28.8	2.6	27	97.0	15	[19, 0, 0, 3, 0, 0]	0
db:r:1	22.4	16.4	0.4	11.1	1.0	3.7	3.6	49	45	33.0	2.5	26	98.6	15	[17, 0, 0, 2, 0, 0]	2
db:r:2	20.1	15.9	0.1	0.1	0.0	3.7	3.6	46	42	15.1	2.4	25	91.2	15	[16, 0, 0, 3, 0, 0]	0
db:r:3	20.2	15.9	0.1	0.1	0.0	3.7	3.6	46	42	15.2	2.4	25	91.1	15	[16, 0, 0, 3, 0, 0]	0

Table B.13: The evaluation results of *Static Baseline* experiment cases under the $\{ccsds, stap, cache\_off\}$ configuration.

	PT [s]	CPT [s]	PRT [s]	RDV [MB]	PS [MB]	OO [s]	ORE [s]	PL [#]	PLR [#]	HC [%]	HCD [level]	HCP [#]	TLC [%]	TLCP [#]	FC [c, l, lg, q, p, e] [#]	FC [u] [#]
no-opt	73.2	27.6	10.5	216.9	29.2	0.0	0.0	164	67	0.0	0.0	0	0.0	0	[32, 0, 0, 3, 0, 0]	5
sb:c	26.1	15.4	0.6	10.0	1.4	8.0	3.6	27	25	35.8	2.8	12	99.4	10	[11, 0, 0, 2, 0, 0]	3
sb:l	23.4	14.9	0.1	0.0	0.0	8.0	3.6	14	12	12.0	1.9	10	94.1	10	[1, 0, 0, 2, 0, 0]	0
sb:q	23.4	14.9	0.1	0.0	0.0	7.9	3.6	11	9	7.5	1.6	8	92.3	8	[0, 0, 0, 2, 0, 0]	0

CCSDS: SystemTap — cache off

Table B.14: The evaluation results of *Call Graph Shaping* experiment cases under the $\{ccsds, stap, cache\_off\}$ configuration.

	PT [s]	CPT [s]	PRT [s]	RDV [MB]	PS [MB]	OO [s]	ORE [s]	PL [#]	PLR [#]	HC [%]	HCD [level]	HCP [#]	TLC [%]	TLCP [#]	FC [c, l, lg, q, p, e] [#]	FC [u] [#]
no-opt	73.2	27.6	10.5	216.9	29.2	0.0	0.0	164	67	0.0	0.0	0	0.0	0	[32, 0, 0, 3, 0, 0]	5
cg:m	78.0	26.5	10.5	217.3	29.2	4.6	3.6	67	63	0.0	3.8	25	133.9	15	[30, 0, 0, 3, 0, 0]	6
cg:p	30.3	17.9	1.9	46.7	5.2	3.8	3.6	40	38	39.9	2.9	20	99.7	10	[15, 0, 0, 1, 0, 0]	4
cg:p-1	30.4	17.9	1.9	46.7	5.2	3.8	3.6	40	38	39.8	2.9	20	99.7	10	[15, 0, 0, 1, 0, 0]	4
cg:p-max	19.1	14.9	0.1	0.0	0.0	3.6	3.6	1	1	0.0	0.0	1	100.0	1	[0, 0, 0, 0, 0, 0]	0
cg:t-s	19.1	14.9	0.1	0.1	0.0	3.6	3.6	30	28	10.4	2.3	18	88.6	10	[11, 0, 0, 1, 0, 0]	0
cg:t-s:l	22.7	15.5	0.8	14.2	1.8	3.7	3.6	41	37	33.5	2.2	25	99.3	15	[13, 0, 0, 3, 0, 0]	0
cg:t-r	19.1	14.9	0.1	0.0	0.0	3.6	3.6	11	11	11.5	1.0	10	88.3	10	[0, 0, 0, 0, 0, 0]	0
cg:t-r:l	19.1	14.9	0.1	0.0	0.0	3.6	3.6	16	16	10.0	1.0	15	89.1	15	[0, 0, 0, 0, 0, 0]	0

Table B.15: The evaluation results of *Pipeline* experiment cases under the $\{ccsds, stap, cache\_off\}$ configuration.

	PT [s]	CPT [s]	PRT [s]	RDV [MB]	PS [MB]	OO [s]	ORE [s]	PL [#]	PLR [#]	HC [%]	HCD [level]	HCP [#]	TLC [%]	TLCP [#]	FC [c, l, lg, q, p, e] [#]	FC [u] [#]
no-opt	73.2	27.6	10.5	216.9	29.2	0.0	0.0	164	67	0.0	0.0	0	0.0	0	[32, 0, 0, 3, 0, 0]	5
p:d:b	19.1	14.9	0.1	0.0	0.0	3.6	3.6	11	11	11.1	1.0	10	88.4	10	[0, 0, 0, 0, 0, 0]	0
p:b	19.1	14.9	0.1	0.0	0.0	3.7	3.7	11	11	12.5	1.0	10	87.4	10	[0, 0, 0, 0, 0, 0]	0
p:d:a	22.0	17.4	1.4	1.5	0.2	3.7	3.6	34	32	95.0	2.6	19	99.3	10	[11, 0, 0, 0, 0, 0]	3
p:a	21.5	16.4	1.4	4.4	0.5	3.7	3.7	33	31	91.3	2.5	19	99.5	10	[11, 0, 0, 1, 0, 0]	2
p:d:f	26.3	17.4	1.4	1.5	0.2	8.0	3.6	26	24	95.1	3.1	14	99.2	8	[10, 0, 0, 3, 0, 0]	2
p:f	25.8	16.4	1.4	4.4	0.5	8.0	3.6	24	22	91.8	3.1	13	99.7	7	[13, 0, 0, 1, 0, 0]	1

Table B.16: The evaluation results of *Diff Tracing* experiment cases under the $\{ccsds, stap, cache\_off\}$ configuration.

	PT [s]	CPT [s]	PRT [s]	RDV [MB]	PS [MB]	OO [s]	ORE [s]	PL [#]	PLR [#]	HC [%]	HCD [level]	HCP [#]	TLC [%]	TLCP [#]	FC [c, l, lg, q, p, e] [#]	FC [u] [#]
no-opt	73.2	27.6	10.5	216.9	29.2	0.0	0.0	164	67	0.0	0.0	0	0.0	0	[32, 0, 0, 3, 0, 0]	5
dt	29.2	16.9	1.9	44.6	5.2	3.8	3.6	7	7	41.5	4.3	3	97.8	2	[4, 0, 0, 0, 0, 0]	2
dt:l	48.5	20.4	5.4	139.1	15.3	4.2	3.6	10	10	58.2	5.3	3	99.2	2	[7, 0, 0, 0, 0, 0]	2
dt:i	29.3	16.9	2.0	44.6	5.2	3.8	3.6	7	7	41.8	4.3	3	97.8	2	[4, 0, 0, 0, 0, 0]	2
dt:s	29.2	16.9	1.9	44.6	5.2	3.8	3.6	7	7	41.7	4.3	3	97.8	2	[5, 0, 0, 0, 0, 0]	1
dt:r	29.3	16.9	1.9	43.6	5.2	3.8	3.6	7	7	41.5	4.3	3	97.8	2	[4, 0, 0, 0, 0, 0]	1

CCSDS: SystemTap — cache on

Table B.17: The evaluation results of *Dynamic Sampling* experiment cases under the $\{ccsds, stap, cache\_on\}$ configuration.

	PT [s]	CPT [s]	PRT [s]	RDV [MB]	PS [MB]	OO [s]	ORE [s]	PL [#]	PLR [#]	HC [%]	HCD [level]	HCP [#]	TLC [%]	TLCP [#]	FC [c, l, lg, q, p, e] [#]	FC [u] [#]
no-opt	59.8	14.2	10.2	211.8	28.5	0.0	0.0	164	67	0.0	0.0	0	0.0	0	[32, 0, 0, 3, 0, 0]	5
ds:d:1	13.2	10.9	7.2	6.8	1.0	0.2	0.1	67	63	97.5	3.8	25	99.9	15	[28, 0, 0, 4, 1, 0]	3
ds:d:2	15.9	11.4	7.4	17.8	2.2	0.3	0.1	67	63	94.3	3.8	25	99.9	15	[32, 0, 0, 3, 0, 0]	5
ds:d:10:1	11.7	10.9	7.1	0.0	0.0	0.2	0.1	67	63	100.0	3.8	25	99.9	15	[4, 0, 0, 2, 0, 0]	0
ds:d:10:2	13.8	10.9	7.3	10.0	1.3	0.2	0.1	67	63	97.1	3.8	25	99.9	15	[32, 0, 0, 3, 0, 0]	5
ds:d:1.1:1	38.4	12.9	9.0	118.7	15.6	0.8	0.1	67	63	62.4	3.8	25	181.2	15	[32, 0, 0, 3, 0, 0]	5
ds:d:1.1:2	15.9	11.4	7.4	18.2	2.2	0.3	0.1	67	63	94.3	3.8	25	99.9	15	[32, 0, 0, 3, 0, 0]	5
ds:s:1	15.9	11.4	7.3	18.0	2.2	0.2	0.1	67	63	94.3	3.8	25	99.9	15	[32, 0, 0, 3, 0, 0]	5
ds:s:2	15.8	11.4	7.4	18.0	2.2	0.3	0.1	67	63	94.4	3.8	25	99.9	15	[32, 0, 0, 3, 0, 0]	5
ds:s:3	15.8	11.4	7.3	18.0	2.2	0.3	0.1	67	63	94.3	3.8	25	99.9	15	[32, 0, 0, 3, 0, 0]	5
ds:r:1	12.1	11.0	7.2	1.9	0.2	0.2	0.1	67	63	99.4	3.8	25	99.9	15	[32, 0, 0, 3, 0, 0]	5
ds:r:2	12.0	10.9	7.2	1.9	0.2	0.2	0.1	67	63	99.4	3.8	25	99.9	15	[33, 0, 0, 3, 0, 0]	4
ds:r:3	12.1	10.9	7.1	1.9	0.2	0.2	0.1	67	63	99.4	3.8	25	99.9	15	[32, 0, 0, 3, 0, 0]	5

Table B.18: The evaluation results of *Dynamic Baseline* experiment cases under the $\{ccsds, stap, cache\_on\}$ configuration.

	PT [s]	CPT [s]	PRT [s]	RDV [MB]	PS [MB]	OO [s]	ORE [s]	PL [#]	PLR [#]	HC [%]	HCD [level]	HCP [#]	TLC [%]	TLCP [#]	FC [c, l, lg, q, p, e] [#]	FC [u] [#]
no-opt	59.8	14.2	10.2	211.8	28.5	0.0	0.0	164	67	0.0	0.0	0	0.0	0	[32, 0, 0, 3, 0, 0]	5
db:s:1	7.5	4.4	0.6	13.7	1.3	0.2	0.1	53	49	33.8	2.8	28	99.0	15	[20, 0, 0, 3, 0, 0]	2
db:s:2	5.3	3.9	0.2	2.8	0.3	0.2	0.1	50	46	28.8	2.6	27	96.8	15	[19, 0, 0, 3, 0, 0]	1
db:s:3	5.3	3.9	0.2	2.8	0.3	0.2	0.1	50	46	28.7	2.6	27	96.7	15	[19, 0, 0, 2, 0, 0]	1
db:r:1	6.9	4.4	0.4	11.1	1.0	0.2	0.1	49	45	32.9	2.5	26	98.6	15	[17, 0, 0, 3, 0, 0]	2
db:r:2	4.7	3.9	0.1	0.1	0.0	0.2	0.1	46	42	15.2	2.4	25	89.8	15	[16, 0, 0, 3, 0, 0]	0
db:r:3	4.7	3.9	0.1	0.1	0.0	0.2	0.1	46	42	15.8	2.4	25	91.0	15	[16, 0, 0, 3, 0, 0]	0

Table B.19: The evaluation results of *Static Baseline* experiment cases under the $\{ccsds, stap, cache\_on\}$ configuration.

	PT [s]	CPT [s]	PRT [s]	RDV [MB]	PS [MB]	OO [s]	ORE [s]	PL [#]	PLR [#]	HC [%]	HCD [level]	HCP [#]	TLC [%]	TLCP [#]	FC [c, l, lg, q, p, e] [#]	FC [u] [#]
no-opt	59.8	14.2	10.2	211.8	28.5	0.0	0.0	164	67	0.0	0.0	0	0.0	0	[32, 0, 0, 3, 0, 0]	5
sb:c	11.7	4.4	0.6	10.0	1.4	4.5	0.1	27	25	35.4	2.8	12	99.4	10	[11, 0, 0, 2, 0, 0]	3
sb:l	8.9	3.9	0.1	0.0	0.0	4.4	0.1	14	12	12.2	1.9	10	93.9	10	[1, 0, 0, 2, 0, 0]	0
sb:q	8.9	3.9	0.1	0.0	0.0	4.4	0.1	11	9	7.5	1.6	8	92.2	8	[0, 0, 0, 2, 0, 0]	0

CCSDS: SystemTap — cache on

Table B.20: The evaluation results of *Call Graph Shaping* experiment cases under the $\{ccsds, stap, cache\_on\}$ configuration.

	PT [s]	CPT [s]	PRT [s]	RDV [MB]	PS [MB]	OO [s]	ORE [s]	PL [#]	PLR [#]	HC [%]	HCD [level]	HCP [#]	TLC [%]	TLCP [#]	FC [c, l, lg, q, p, e] [#]	FC [u] [#]
no-opt	59.8	14.2	10.2	211.8	28.5	0.0	0.0	164	67	0.0	0.0	0	0.0	0	[32, 0, 0, 3, 0, 0]	5
cg:m	61.2	14.4	10.2	211.8	28.5	1.1	0.1	67	63	22.0	3.8	25	112.9	15	[32, 0, 0, 3, 0, 0]	5
cg:p	14.9	5.9	1.9	46.7	5.2	0.3	0.1	40	38	39.9	2.9	20	99.6	10	[15, 0, 0, 1, 0, 0]	4
cg:p-1	14.9	5.9	1.9	46.7	5.2	0.3	0.1	40	38	39.6	2.9	20	99.7	10	[15, 0, 0, 1, 0, 0]	4
cg:p-max	4.6	3.9	0.1	0.0	0.0	0.1	0.1	1	1	0.0	0.0	1	100.0	1	[0, 0, 0, 0, 0, 0]	0
cg:t-s	4.6	3.9	0.1	0.1	0.0	0.1	0.1	30	28	10.3	2.3	18	87.6	10	[11, 0, 0, 1, 0, 0]	0
cg:t-s:l	8.2	4.4	0.7	13.9	1.8	0.2	0.1	41	37	33.3	2.2	25	99.3	15	[13, 0, 0, 3, 0, 0]	0
cg:t-r	4.6	3.9	0.1	0.0	0.0	0.1	0.1	11	11	11.3	1.0	10	88.5	10	[0, 0, 0, 0, 0, 0]	0
cg:t-r:l	4.6	3.9	0.1	0.0	0.0	0.1	0.1	16	16	9.5	1.0	15	90.5	15	[0, 0, 0, 0, 0, 0]	0

Table B.21: The evaluation results of *Pipeline* experiment cases under the $\{ccsds, stap, cache\_on\}$ configuration.

	PT [s]	CPT [s]	PRT [s]	RDV [MB]	PS [MB]	OO [s]	ORE [s]	PL [#]	PLR [#]	HC [%]	HCD [level]	HCP [#]	TLC [%]	TLCP [#]	FC [c, l, lg, q, p, e] [#]	FC [u] [#]
no-opt	59.8	14.2	10.2	211.8	28.5	0.0	0.0	164	67	0.0	0.0	0	0.0	0	[32, 0, 0, 3, 0, 0]	5
p:d:b	4.6	3.9	0.1	0.0	0.0	0.2	0.1	11	11	11.1	1.0	10	88.3	10	[0, 0, 0, 0, 0, 0]	0
p:b	4.6	3.9	0.1	0.0	0.0	0.2	0.1	11	11	11.0	1.0	10	87.6	10	[0, 0, 0, 0, 0, 0]	0
p:d:a	6.4	5.4	1.4	1.4	0.2	0.2	0.1	34	32	94.9	2.6	19	99.3	10	[10, 0, 0, 1, 0, 0]	3
p:a	6.9	5.4	1.4	4.3	0.5	0.2	0.1	33	31	91.6	2.5	19	99.5	10	[11, 0, 0, 1, 0, 0]	2
p:d:f	10.8	5.4	1.4	1.4	0.2	4.5	0.1	26	24	95.2	3.1	14	99.2	8	[10, 0, 0, 3, 0, 0]	1
p:f	11.2	5.4	1.4	4.3	0.5	4.5	0.1	24	22	92.0	3.1	13	99.7	7	[13, 0, 0, 1, 0, 0]	1

Table B.22: The evaluation results of *Diff Tracing* experiment cases under the $\{ccsds, stap, cache\_on\}$ configuration.

	PT [s]	CPT [s]	PRT [s]	RDV [MB]	PS [MB]	OO [s]	ORE [s]	PL [#]	PLR [#]	HC [%]	HCD [level]	HCP [#]	TLC [%]	TLCP [#]	FC [c, l, lg, q, p, e] [#]	FC [u] [#]
no-opt	59.8	14.2	10.2	211.8	28.5	0.0	0.0	164	67	0.0	0.0	0	0.0	0	[32, 0, 0, 3, 0, 0]	5
dt	14.9	5.9	1.9	44.6	5.2	0.3	0.1	7	7	41.6	4.3	3	97.8	2	[5, 0, 0, 0, 0, 0]	1
dt:l	35.1	9.9	5.5	144.8	15.7	0.7	0.1	10	10	57.3	5.3	3	99.2	2	[7, 0, 0, 0, 0, 0]	2
dt:i	15.0	5.9	1.9	44.6	5.2	0.3	0.1	7	7	41.5	4.3	3	97.8	2	[4, 0, 0, 0, 0, 0]	2
dt:s	14.9	5.9	1.9	44.6	5.2	0.3	0.1	7	7	41.6	4.3	3	97.7	2	[4, 0, 0, 0, 0, 0]	2
dt:r	14.8	5.9	1.9	44.6	5.2	0.3	0.1	7	7	41.6	4.3	3	97.7	2	[4, 0, 0, 0, 0, 0]	2

CCSDS: eBPF — cache off

Table B.23: The evaluation results of *Dynamic Sampling* experiment cases under the $\{ccsds, ebpf, cache\_off\}$ configuration.

	PT [s]	CPT [s]	PRT [s]	RDV [MB]	PS [MB]	OO [s]	ORE [s]	PL [#]	PLR [#]	HC [%]	HCD [level]	HCP [#]	TLC [%]	TLCP [#]	FC [c, l, lg, q, p, e] [#]	FC [u] [#]
no-opt	22.2	17.9	7.2	48.9	23.0	0.0	0.0	164	59	0.0	0.0	0	0.0	0	[20, 0, 1, 1, 0, 0]	12
ds:d:1	16.4	12.5	5.6	1.6	0.8	3.7	3.6	67	56	98.0	3.9	21	99.8	14	[18, 0, 0, 7, 0, 0]	6
ds:d:2	16.6	12.4	5.6	3.5	1.5	3.7	3.6	67	51	95.5	3.7	18	99.9	14	[14, 0, 0, 1, 1, 0]	14
ds:d:10:1	15.6	11.9	5.0	0.0	0.0	3.7	3.6	67	56	100.0	3.9	21	99.8	14	[1, 0, 0, 3, 0, 0]	0
ds:d:10:2	16.3	12.4	5.5	1.8	0.8	3.7	3.6	67	52	92.2	3.7	18	99.8	14	[15, 0, 0, 1, 1, 0]	14
ds:d:1.1:1	17.0	12.5	5.7	6.7	3.1	3.8	3.6	67	52	69.2	3.7	18	99.9	14	[15, 0, 0, 0, 1, 0]	14
ds:d:1.1:2	16.4	12.3	5.5	2.7	1.2	3.7	3.6	67	49	95.9	3.6	17	99.9	14	[12, 0, 0, 0, 1, 0]	14
ds:s:1	16.5	12.4	5.6	3.4	1.5	3.7	3.6	67	51	95.5	3.7	18	99.9	14	[14, 0, 0, 1, 1, 0]	14
ds:s:2	16.5	12.4	5.6	3.4	1.5	3.7	3.6	67	51	95.5	3.7	18	99.9	14	[14, 0, 0, 1, 1, 0]	14
ds:s:3	16.6	12.4	5.6	3.4	1.5	3.7	3.6	67	51	95.5	3.7	18	99.9	14	[14, 0, 0, 1, 1, 0]	14
ds:r:1	15.9	12.1	5.3	0.4	0.2	3.7	3.6	67	51	98.2	3.0	26	99.9	14	[13, 0, 0, 1, 1, 0]	14
ds:r:2	15.8	12.1	5.3	0.4	0.2	3.7	3.6	67	51	98.2	3.0	26	99.9	14	[14, 0, 0, 0, 1, 0]	14
ds:r:3	15.9	12.1	5.3	0.4	0.2	3.7	3.6	67	51	98.2	3.0	26	99.9	14	[14, 0, 0, 1, 1, 0]	13

Table B.24: The evaluation results of *Call Graph Shaping* experiment cases under the $\{ccsds, ebpf, cache\_off\}$ configuration.

	PT [s]	CPT [s]	PRT [s]	RDV [MB]	PS [MB]	OO [s]	ORE [s]	PL [#]	PLR [#]	HC [%]	HCD [level]	HCP [#]	TLC [%]	TLCP [#]	FC [c, l, lg, q, p, e] [#]	FC [u] [#]
no-opt	22.2	17.9	7.2	48.9	23.0	0.0	0.0	164	59	0.0	0.0	0	0.0	0	[20, 0, 1, 1, 0, 0]	12
cg:m	23.1	14.0	7.3	47.4	23.0	4.8	3.6	67	57	49.5	4.0	22	99.9	14	[20, 0, 1, 4, 0, 0]	10
cg:p	13.3	8.0	2.4	14.7	6.7	4.0	3.6	40	36	49.0	2.9	19	99.7	9	[7, 0, 0, 0, 0, 0]	12
cg:p-1	13.3	8.0	2.4	14.7	6.7	4.0	3.6	40	36	49.0	2.9	19	99.7	9	[6, 0, 0, 0, 1, 0]	12
cg:p-max	7.8	4.1	0.0	0.0	0.0	3.6	3.6	1	1	0.0	0.0	1	100.0	1	[0, 0, 0, 0, 0, 0]	0
cg:t-s	9.6	5.9	0.6	0.0	0.0	3.6	3.6	30	26	14.9	2.4	17	88.4	9	[1, 0, 0, 0, 1, 0]	9
cg:t-s:l	10.0	6.3	0.6	0.0	0.0	3.6	3.6	41	32	12.7	2.1	23	90.8	14	[1, 0, 0, 0, 1, 0]	9
cg:t-r	8.7	5.0	0.6	0.0	0.0	3.6	3.6	11	10	14.2	1.0	9	85.6	9	[0, 0, 0, 0, 0, 0]	0
cg:t-r:l	8.9	5.2	0.6	0.0	0.0	3.6	3.6	16	15	14.7	1.0	14	84.4	14	[0, 0, 0, 0, 0, 0]	0

CCSDS: eBPF — cache off

Table B.25: The evaluation results of *Dynamic Baseline* experiment cases under the $\{ccsds, ebpf, cache\_off\}$ configuration.

	PT [s]	CPT [s]	PRT [s]	RDV [MB]	PS [MB]	OO [s]	ORE [s]	PL [#]	PLR [#]	HC [%]	HCD [level]	HCP [#]	TLC [%]	TLCP [#]	FC [c, l, lg, q, p, e] [#]	FC [u] [#]
no-opt	22.2	17.9	7.2	48.9	23.0	0.0	0.0	164	59	0.0	0.0	0	0.0	0	[20, 0, 1, 1, 0, 0]	12
db:s:1	11.0	7.1	0.8	1.9	0.8	3.7	3.6	54	44	42.3	2.7	25	97.9	14	[7, 0, 0, 0, 1, 0]	14
db:s:2	10.7	6.9	0.7	0.9	0.4	3.7	3.6	53	43	37.8	2.7	25	95.9	14	[5, 0, 0, 1, 1, 0]	15
db:s:3	10.7	6.9	0.7	0.9	0.4	3.7	3.6	53	43	38.0	2.7	25	96.2	14	[5, 0, 0, 2, 1, 0]	10
db:r:1	10.7	6.8	0.7	1.0	0.5	3.7	3.6	50	40	38.7	2.4	23	95.7	14	[2, 0, 1, 0, 1, 0]	14
db:r:2	10.3	6.6	0.6	0.0	0.0	3.7	3.6	49	39	20.8	2.4	23	88.6	14	[3, 0, 0, 1, 0, 0]	13
db:r:3	10.4	6.7	0.6	0.0	0.0	3.7	3.6	49	39	20.4	2.4	23	88.3	14	[3, 0, 0, 1, 1, 0]	12

Table B.26: The evaluation results of *Pipeline* experiment cases under the $\{ccsds, ebpf, cache\_off\}$ configuration.

	PT [s]	CPT [s]	PRT [s]	RDV [MB]	PS [MB]	OO [s]	ORE [s]	PL [#]	PLR [#]	HC [%]	HCD [level]	HCP [#]	TLC [%]	TLCP [#]	FC [c, l, lg, q, p, e] [#]	FC [u] [#]
no-opt	22.2	17.9	7.2	48.9	23.0	0.0	0.0	164	59	0.0	0.0	0	0.0	0	[20, 0, 1, 1, 0, 0]	12
p:d:b	8.7	5.0	0.6	0.0	0.0	3.6	3.6	11	10	13.5	1.0	9	86.0	9	[0, 0, 0, 0, 0, 0]	0
p:b	8.7	5.0	0.6	0.0	0.0	3.7	3.6	11	10	12.9	1.0	9	85.8	9	[0, 0, 0, 0, 0, 0]	0
p:d:a	11.2	7.5	1.9	0.2	0.1	3.7	3.6	34	29	93.1	2.6	18	99.3	9	[3, 0, 0, 5, 0, 0]	5
p:a	11.1	7.3	1.9	0.3	0.2	3.7	3.6	34	27	7.0	2.4	17	99.5	9	[2, 0, 0, 0, 1, 0]	10
p:d:f	15.1	7.1	2.0	0.2	0.1	8.0	3.6	26	22	93.2	3.1	13	99.3	7	[3, 0, 0, 5, 0, 0]	4
p:f	15.0	6.9	1.9	0.3	0.2	8.0	3.6	25	19	7.3	2.8	12	99.7	7	[1, 0, 0, 1, 1, 0]	8

Table B.27: The evaluation results of *Diff Tracing* experiment cases under the $\{ccsds, ebpf, cache\_off\}$ configuration.

	PT [s]	CPT [s]	PRT [s]	RDV [MB]	PS [MB]	OO [s]	ORE [s]	PL [#]	PLR [#]	HC [%]	HCD [level]	HCP [#]	TLC [%]	TLCP [#]	FC [c, l, lg, q, p, e] [#]	FC [u] [#]
no-opt	22.2	17.9	7.2	48.9	23.0	0.0	0.0	164	59	0.0	0.0	0	0.0	0	[20, 0, 1, 1, 0, 0]	12
dt	11.9	6.7	2.4	13.7	6.6	4.0	3.6	7	7	50.5	4.3	3	98.0	2	[4, 0, 0, 0, 0, 0]	2
dt:l	13.0	7.3	2.9	17.5	8.4	4.1	3.6	10	9	55.3	5.0	3	98.4	2	[6, 0, 0, 0, 0, 0]	2
dt:i	11.9	6.7	2.4	13.7	6.6	3.9	3.6	7	7	50.5	4.3	3	97.9	2	[4, 0, 0, 0, 0, 0]	2
dt:s	11.9	6.7	2.4	14.2	6.6	3.9	3.6	7	7	50.5	4.3	3	97.9	2	[4, 0, 0, 0, 0, 0]	2
dt:r	11.9	6.7	2.4	14.2	6.6	4.0	3.6	7	7	50.5	4.3	3	97.9	2	[4, 0, 0, 0, 0, 0]	2

CCSDS: eBPF — cache off

Table B.28: The evaluation results of *Dynamic Probing* experiment cases under the $\{ccsds, ebpf, cache\_off\}$ configuration.

	PT [s]	**CPT** [s]	**PRT** [s]	**RDV** [MB]	**PS** [MB]	**OO** [s]	**ORE** [s]	**PL** [$\#$]	**PLR** [$\#$]	**HC** [$\%$]	**HCD** [$level$]	**HCP** [$\#$]	**TLC** [$\%$]	**TLCP** [$\#$]	**FC [c, l, lg, q, p, e]** [$\#$]	**FC [u]** [$\#$]
no-opt	22.2	17.9	7.2	48.9	23.0	0.0	0.0	164	59	0.0	0.0	0	0.0	0	[20, 0, 1, 1, 0, 0]	12
dp	16.4	13.8	3.2	23.6	10.4	0.6	0.0	164	59	0.0	0.0	0	0.0	0	[18, 0, 0, 3, 1, 1]	12
dp:l	14.3	12.8	2.5	14.8	6.4	0.4	0.0	164	59	0.0	0.0	0	0.0	0	[18, 0, 0, 4, 4, 1]	7
dp:h	23.0	17.5	6.8	49.2	22.3	1.2	0.0	164	59	0.0	0.0	0	0.0	0	[19, 0, 1, 3, 0, 0]	13
dp:r	14.9	13.1	2.7	16.0	6.9	0.4	0.0	164	59	0.0	0.0	0	0.0	0	[18, 0, 0, 4, 3, 2]	5

Table B.29: The evaluation results of *Static Baseline* experiment cases under the $\{ccsds, ebpf, cache\_off\}$ configuration.

	PT [s]	**CPT** [s]	**PRT** [s]	**RDV** [MB]	**PS** [MB]	**OO** [s]	**ORE** [s]	**PL** [$\#$]	**PLR** [$\#$]	**HC** [$\%$]	**HCD** [$level$]	**HCP** [$\#$]	**TLC** [$\%$]	**TLCP** [$\#$]	**FC [c, l, lg, q, p, e]** [$\#$]	**FC [u]** [$\#$]
no-opt	22.2	17.9	7.2	48.9	23.0	0.0	0.0	164	59	0.0	0.0	0	0.0	0	[20, 0, 1, 1, 0, 0]	12
sb:c	14.6	6.1	1.1	3.8	1.8	8.0	3.6	27	23	46.1	2.9	10	99.3	8	[4, 0, 0, 0, 1, 0]	9
sb:l	13.0	5.1	0.6	0.0	0.0	7.9	3.6	14	10	16.2	1.9	8	91.9	8	[1, 0, 0, 1, 0, 0]	0
sb:q	12.9	5.0	0.6	0.0	0.0	7.9	3.6	11	7	9.5	1.5	6	90.3	6	[0, 0, 0, 0, 1, 0]	0

Table B.30: The evaluation results of *Timed Sampling* experiment cases under the $\{ccsds, ebpf, cache\_off\}$ configuration.

	PT [s]	**CPT** [s]	**PRT** [s]	**RDV** [MB]	**PS** [MB]	**OO** [s]	**ORE** [s]	**PL** [$\#$]	**PLR** [$\#$]	**HC** [$\%$]	**HCD** [$level$]	**HCP** [$\#$]	**TLC** [$\%$]	**TLCP** [$\#$]	**FC [c, l, lg, q, p, e]** [$\#$]	**FC [u]** [$\#$]
no-opt	22.2	17.9	7.2	48.9	23.0	0.0	0.0	164	59	0.0	0.0	0	0.0	0	[20, 0, 1, 1, 0, 0]	12
ts	18.6	16.7	5.7	17.1	7.6	0.4	0.0	164	45	0.0	0.0	0	0.0	0	[17, 0, 0, 2, 0, 0]	7
ts:2	19.1	16.6	5.5	22.3	9.9	0.5	0.0	164	55	0.0	0.0	0	0.0	0	[19, 0, 0, 1, 0, 0]	14
ts:4	18.9	16.4	5.5	21.3	9.5	0.5	0.0	164	50	0.0	0.0	0	0.0	0	[18, 0, 0, 1, 0, 0]	16

CCSDS: eBPF — cache on

Table B.31: The evaluation results of *Dynamic Sampling* experiment cases under the $\{ccsds, ebpf, cache\_on\}$ configuration.

	PT [s]	CPT [s]	PRT [s]	RDV [MB]	PS [MB]	OO [s]	ORE [s]	PL [$\#$]	PLR [$\#$]	HC [$\%$]	HCD [$level$]	HCP [$\#$]	TLC [$\%$]	TLCP [$\#$]	FC [c, l, lg, q, p, e] [$\#$]	FC [u] [$\#$]
no-opt	22.6	18.2	7.3	50.6	23.0	0.0	0.0	164	59	0.0	0.0	0	0.0	0	[20, 0, 1, 1, 0, 0]	14
ds:d:1	13.0	12.5	5.6	1.6	0.8	0.2	0.1	67	56	98.1	3.9	21	99.8	14	[18, 0, 1, 7, 0, 0]	6
ds:d:2	13.1	12.5	5.6	3.5	1.5	0.2	0.1	67	51	95.5	3.7	18	99.9	14	[14, 0, 0, 1, 1, 0]	14
ds:d:10:1	12.2	12.0	5.1	0.0	0.0	0.2	0.1	67	56	100.0	3.9	21	99.8	14	[1, 0, 0, 3, 1, 0]	0
ds:d:10:2	12.9	12.5	5.6	1.9	0.8	0.2	0.1	67	52	92.2	3.7	18	99.9	14	[15, 0, 0, 1, 1, 0]	14
ds:d:1.1:1	13.6	12.5	5.7	6.9	3.1	0.3	0.1	67	52	69.0	3.7	18	99.9	14	[15, 0, 0, 1, 1, 0]	14
ds:d:1.1:2	12.9	12.4	5.5	2.8	1.2	0.2	0.1	67	49	95.9	3.6	17	99.9	14	[12, 0, 0, 1, 1, 0]	14
ds:s:1	13.1	12.5	5.6	3.5	1.5	0.2	0.1	67	51	95.5	3.7	18	99.9	14	[14, 0, 0, 1, 1, 0]	14
ds:s:2	13.1	12.5	5.6	3.5	1.5	0.2	0.1	67	51	95.5	3.7	18	99.9	14	[14, 0, 0, 1, 1, 0]	14
ds:s:3	13.1	12.5	5.6	3.5	1.5	0.2	0.1	67	51	95.5	3.7	18	99.9	14	[14, 0, 0, 1, 1, 0]	14
ds:r:1	12.5	12.2	5.3	0.4	0.2	0.2	0.1	67	51	94.7	3.0	26	96.3	14	[13, 0, 0, 1, 1, 0]	13
ds:r:2	12.4	12.2	5.3	0.4	0.2	0.2	0.1	67	51	98.2	3.0	26	99.9	14	[13, 0, 0, 1, 1, 0]	14
ds:r:3	12.4	12.2	5.3	0.4	0.2	0.2	0.1	67	51	98.2	3.0	26	99.8	14	[13, 0, 0, 1, 1, 0]	14

Table B.32: The evaluation results of *Call Graph Shaping* experiment cases under the $\{ccsds, ebpf, cache\_on\}$ configuration.

	PT [s]	CPT [s]	PRT [s]	RDV [MB]	PS [MB]	OO [s]	ORE [s]	PL [$\#$]	PLR [$\#$]	HC [$\%$]	HCD [$level$]	HCP [$\#$]	TLC [$\%$]	TLCP [$\#$]	FC [c, l, lg, q, p, e] [$\#$]	FC [u] [$\#$]
no-opt	22.6	18.2	7.3	50.6	23.0	0.0	0.0	164	59	0.0	0.0	0	0.0	0	[20, 0, 1, 1, 0, 0]	14
cg:m	19.8	14.1	7.3	49.1	23.0	1.3	0.1	67	57	50.4	4.0	22	97.4	14	[19, 0, 1, 1, 0, 0]	14
cg:p	9.9	8.1	2.4	14.2	6.7	0.5	0.1	40	36	49.2	2.9	19	99.6	9	[6, 0, 0, 0, 1, 0]	13
cg:p-1	9.9	8.1	2.4	14.2	6.7	0.5	0.1	40	36	49.0	2.9	19	99.6	9	[6, 0, 0, 0, 0, 0]	13
cg:p-max	4.3	4.1	0.1	0.0	0.0	0.1	0.1	1	1	0.0	0.0	1	100.0	1	[0, 0, 0, 0, 0, 0]	0
cg:t-s	6.0	5.8	0.6	0.0	0.0	0.1	0.1	30	26	14.1	2.4	17	87.5	9	[2, 0, 0, 0, 1, 0]	9
cg:t-s:l	6.5	6.3	0.6	0.0	0.0	0.1	0.1	41	32	13.2	2.1	23	87.9	14	[1, 0, 0, 1, 1, 0]	9
cg:t-r	5.2	5.0	0.6	0.0	0.0	0.1	0.1	11	10	14.7	1.0	9	84.3	9	[0, 0, 0, 0, 0, 0]	0
cg:t-r:l	5.4	5.2	0.6	0.0	0.0	0.1	0.1	16	15	11.6	1.0	14	87.8	14	[0, 0, 0, 0, 0, 0]	0

CCSDS: eBPF — cache on

Table B.33: The evaluation results of *Dynamic Baseline* experiment cases under the $\{ccsds, ebpf, cache\_on\}$ configuration.

	PT [s]	CPT [s]	PRT [s]	RDV [MB]	PS [MB]	OO [s]	ORE [s]	PL [#]	PLR [#]	HC [%]	HCD [level]	HCP [#]	TLC [%]	TLCP [#]	FC [c, l, lg, q, p, e] [#]	FC [u] [#]
no-opt	22.6	18.2	7.3	50.6	23.0	0.0	0.0	164	59	0.0	0.0	0	0.0	0	[20, 0, 1, 1, 0, 0]	14
db:s:1	7.6	7.2	0.8	1.9	0.8	0.2	0.1	54	44	41.9	2.7	25	98.2	14	[7, 0, 0, 0, 1, 0]	14
db:s:2	7.3	7.0	0.7	0.9	0.4	0.2	0.1	53	43	38.0	2.7	25	96.8	14	[6, 0, 0, 1, 1, 0]	14
db:s:3	7.3	7.0	0.7	0.9	0.4	0.2	0.1	53	43	36.9	2.7	25	94.8	14	[5, 0, 0, 1, 0, 0]	15
db:r:1	7.2	6.9	0.7	1.0	0.5	0.2	0.1	50	40	39.6	2.4	23	97.2	14	[3, 0, 1, 0, 1, 0]	12
db:r:2	7.0	6.8	0.6	0.0	0.0	0.2	0.1	49	39	20.7	2.4	23	89.4	14	[2, 0, 0, 1, 0, 0]	12
db:r:3	6.9	6.7	0.6	0.0	0.0	0.2	0.1	49	39	7.8	2.4	23	35.5	14	[2, 0, 0, 1, 1, 0]	14

Table B.34: The evaluation results of *Pipeline* experiment cases under the $\{ccsds, ebpf, cache\_on\}$ configuration.

	PT [s]	CPT [s]	PRT [s]	RDV [MB]	PS [MB]	OO [s]	ORE [s]	PL [#]	PLR [#]	HC [%]	HCD [level]	HCP [#]	TLC [%]	TLCP [#]	FC [c, l, lg, q, p, e] [#]	FC [u] [#]
no-opt	22.6	18.2	7.3	50.6	23.0	0.0	0.0	164	59	0.0	0.0	0	0.0	0	[20, 0, 1, 1, 0, 0]	14
p:d:b	5.2	5.0	0.6	0.0	0.0	0.2	0.1	11	10	14.3	1.0	9	85.4	9	[0, 0, 0, 0, 0, 0]	0
p:b	5.2	5.0	0.6	0.0	0.0	0.2	0.1	11	10	14.1	1.0	9	85.7	9	[0, 0, 0, 0, 0, 0]	0
p:d:a	7.7	7.4	1.9	0.2	0.1	0.2	0.1	34	29	93.0	2.6	18	99.3	9	[3, 0, 1, 4, 0, 0]	6
p:a	7.6	7.3	1.9	0.3	0.2	0.2	0.1	34	27	7.1	2.4	17	99.4	9	[2, 0, 0, 0, 1, 0]	9
p:d:f	11.6	7.1	2.0	0.2	0.1	4.5	0.1	26	22	93.2	3.1	13	99.2	7	[3, 0, 0, 6, 0, 0]	4
p:f	11.5	7.0	1.9	0.3	0.2	4.5	0.1	25	19	7.3	2.8	12	99.7	7	[1, 0, 0, 0, 1, 0]	9

Table B.35: The evaluation results of *Diff Tracing* experiment cases under the $\{ccsds, ebpf, cache\_on\}$ configuration.

	PT [s]	CPT [s]	PRT [s]	RDV [MB]	PS [MB]	OO [s]	ORE [s]	PL [#]	PLR [#]	HC [%]	HCD [level]	HCP [#]	TLC [%]	TLCP [#]	FC [c, l, lg, q, p, e] [#]	FC [u] [#]
no-opt	22.6	18.2	7.3	50.6	23.0	0.0	0.0	164	59	0.0	0.0	0	0.0	0	[20, 0, 1, 1, 0, 0]	14
dt	8.5	6.7	2.4	13.3	6.6	0.5	0.1	7	7	50.6	4.3	3	98.0	2	[4, 0, 0, 0, 0, 0]	2
dt:l	9.6	7.3	2.9	16.9	8.4	0.6	0.1	10	9	55.3	5.0	3	98.3	2	[6, 0, 0, 0, 0, 0]	2
dt:i	8.5	6.7	2.4	13.3	6.6	0.5	0.1	7	7	50.5	4.3	3	98.0	2	[4, 0, 0, 0, 0, 0]	2
dt:s	8.5	6.7	2.4	13.3	6.6	0.5	0.1	7	7	50.5	4.3	3	97.9	2	[4, 0, 0, 0, 0, 0]	2
dt:r	8.5	6.7	2.4	13.3	6.6	0.5	0.1	7	7	50.6	4.3	3	98.0	2	[4, 0, 0, 0, 0, 0]	2

CCSDS: eBPF — cache on

Table B.36: The evaluation results of *Dynamic Probing* experiment cases under the $\{ccsds, ebpf, cache\_on\}$ configuration.

	PT [s]	CPT [s]	PRT [s]	RDV [MB]	PS [MB]	OO [s]	ORE [s]	PL [$\#$]	PLR [$\#$]	HC [$\%$]	HCD [$level$]	HCP [$\#$]	TLC [$\%$]	TLCP [$\#$]	FC [c, l, lg, q, p, e] [$\#$]	FC [u] [$\#$]
no-opt	22.6	18.2	7.3	50.6	23.0	0.0	0.0	164	59	0.0	0.0	0	0.0	0	[20, 0, 1, 1, 0, 0]	14
dp	17.1	14.5	3.7	22.8	10.4	0.6	0.0	164	59	0.0	0.0	0	0.0	0	[18, 0, 0, 4, 1, 1]	10
dp:l	14.8	12.9	2.6	14.4	6.3	0.4	0.0	164	59	0.0	0.0	0	0.0	0	[19, 0, 0, 4, 3, 2]	8
dp:h	23.6	18.2	7.3	50.6	23.0	1.2	0.0	164	59	0.0	0.0	0	0.0	0	[20, 0, 1, 5, 0, 0]	10
dp:r	15.0	13.2	2.7	15.9	6.9	0.4	0.0	164	59	0.0	0.0	0	0.0	0	[18, 0, 0, 3, 4, 3]	5

Table B.37: The evaluation results of *Static Baseline* experiment cases under the $\{ccsds, ebpf, cache\_on\}$ configuration.

	PT [s]	CPT [s]	PRT [s]	RDV [MB]	PS [MB]	OO [s]	ORE [s]	PL [$\#$]	PLR [$\#$]	HC [$\%$]	HCD [$level$]	HCP [$\#$]	TLC [$\%$]	TLCP [$\#$]	FC [c, l, lg, q, p, e] [$\#$]	FC [u] [$\#$]
no-opt	22.6	18.2	7.3	50.6	23.0	0.0	0.0	164	59	0.0	0.0	0	0.0	0	[20, 0, 1, 1, 0, 0]	14
sb:c	11.2	6.3	1.1	3.7	1.8	4.5	0.1	27	23	45.6	2.9	10	99.4	8	[4, 0, 0, 0, 1, 0]	10
sb:l	9.6	5.1	0.6	0.0	0.0	4.4	0.1	14	10	16.1	1.9	8	91.9	8	[0, 0, 0, 0, 0, 0]	0
sb:q	9.5	5.0	0.6	0.0	0.0	4.4	0.1	11	7	9.4	1.5	6	90.6	6	[0, 0, 0, 0, 0, 0]	0

Table B.38: The evaluation results of *Timed Sampling* experiment cases under the $\{ccsds, ebpf, cache\_on\}$ configuration.

	PT [s]	CPT [s]	PRT [s]	RDV [MB]	PS [MB]	OO [s]	ORE [s]	PL [$\#$]	PLR [$\#$]	HC [$\%$]	HCD [$level$]	HCP [$\#$]	TLC [$\%$]	TLCP [$\#$]	FC [c, l, lg, q, p, e] [$\#$]	FC [u] [$\#$]
no-opt	22.6	18.2	7.3	50.6	23.0	0.0	0.0	164	59	0.0	0.0	0	0.0	0	[20, 0, 1, 1, 0, 0]	14
ts	18.9	16.8	5.7	17.0	7.5	0.4	0.0	164	48	0.0	0.0	0	0.0	0	[19, 0, 0, 2, 0, 0]	13
ts:2	19.5	16.9	5.5	22.4	10.0	0.5	0.0	164	55	0.0	0.0	0	0.0	0	[19, 0, 0, 1, 1, 0]	14
ts:4	19.1	16.6	5.5	21.2	9.4	0.5	0.0	164	50	0.0	0.0	0	0.0	0	[19, 0, 0, 1, 0, 0]	16

Appendix C

Evaluation Graphs

In this Appendix we present some of the evaluation plots and graphs that have been omitted from the evaluation Chapter 8 due to the space limitations.

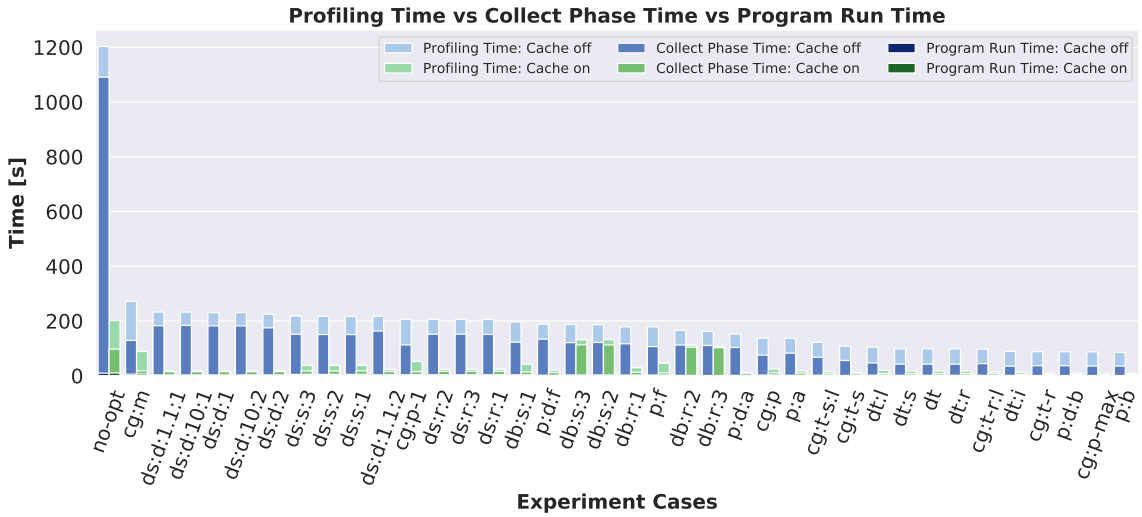

Figure C.1: Comparison of the `PT`, `CPT` and `PRT` metrics for both *cache_off* and *cache_on* configurations of the `CPython` project. Note that the Dynamic Baseline technique does not benefit from the caching technique nearly as much as other methods. The reason lies in the dynamic nature of the method — although we use pre-computed initial Dynamic Stats resource (so that the results are comparable), each subsequent iteration of Dynamic Baseline method usually achieves unique results (caused by the system interference) which translates to slightly different SystemTap scripts, thus the whole compilation process must be performed (instead of loading a cached kernel module).

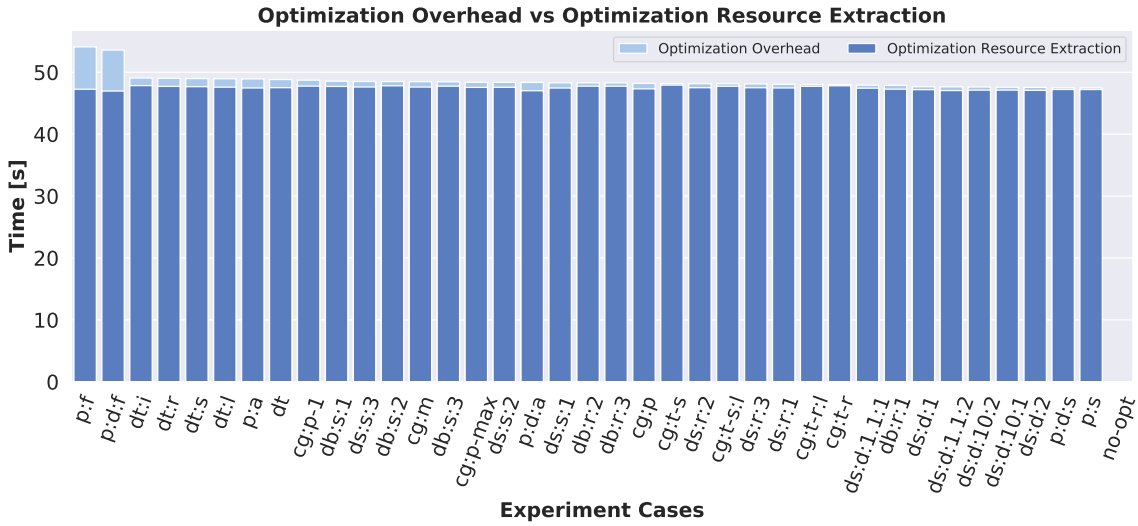

Figure C.2: An illustration of the Optimization Overhead (`OO`) and the duration of Optimization Resource Extraction (`ORE`) in the *cache_off* configuration and `CPython` project. It is evident that the optimization overhead itself is negligible compared to the time requirements of resource extraction. However, note the increased `OO` values in `p:f` and `p:d:f` cases which are caused by the failed Static Baseline compilation (which takes a few seconds before crashing) — this is also the reason why we excluded the `sb:*` cases evaluation in the `CPython` project.

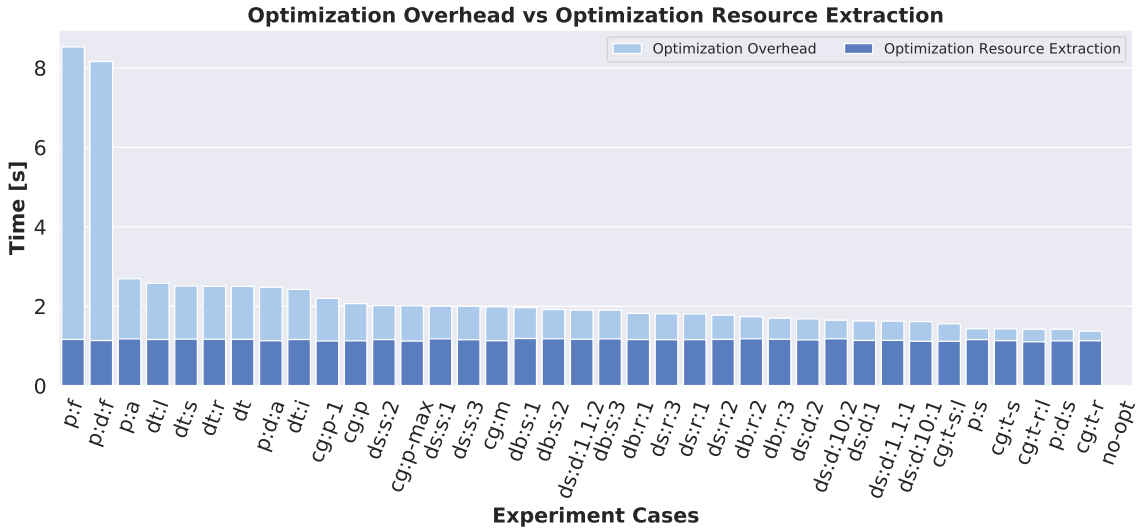

Figure C.3: A complementary Figure to the Figure C.2 that shows the same `OO` and `ORE` metrics, however, measured with the *cache_on* configuration. Note especially the significant reduction of `ORE` values.

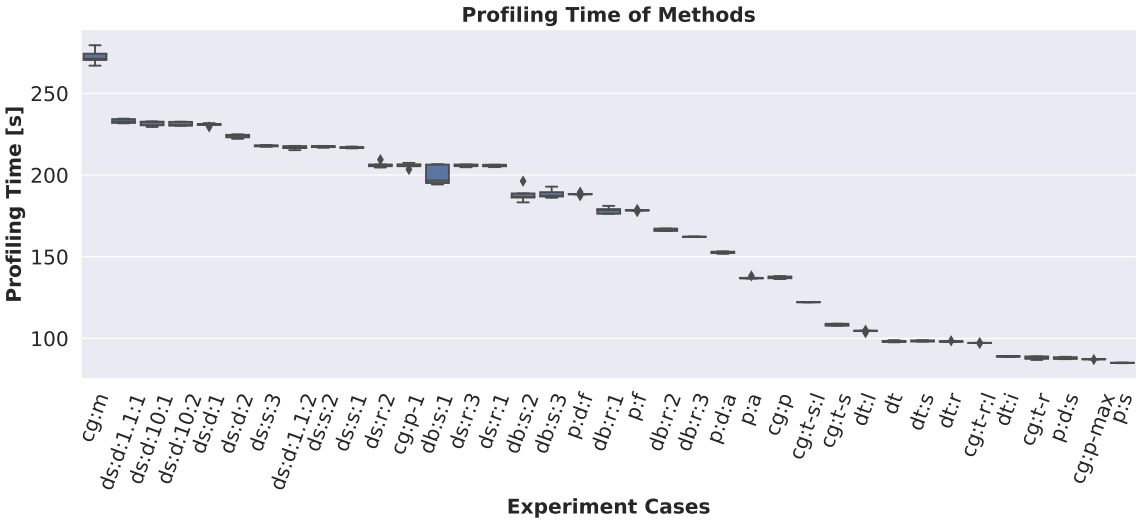

Figure C.4: In Section 8.1, we have explained the employed methodology for the experimental evaluation, and especially the technique of aggregating the results from multiple runs into a single *median* value. However, we were also interested in examining the spread of the results obtained in individual runs. Thus, we created several *Box plots* that show the *median*, *first quartile*, *third quartile*, *minimum* and *maximum* values for various metrics. Since the different box plot values can be hardly seen, we conclude that the Tracer collection process is surprisingly reliable in terms of consistent results.

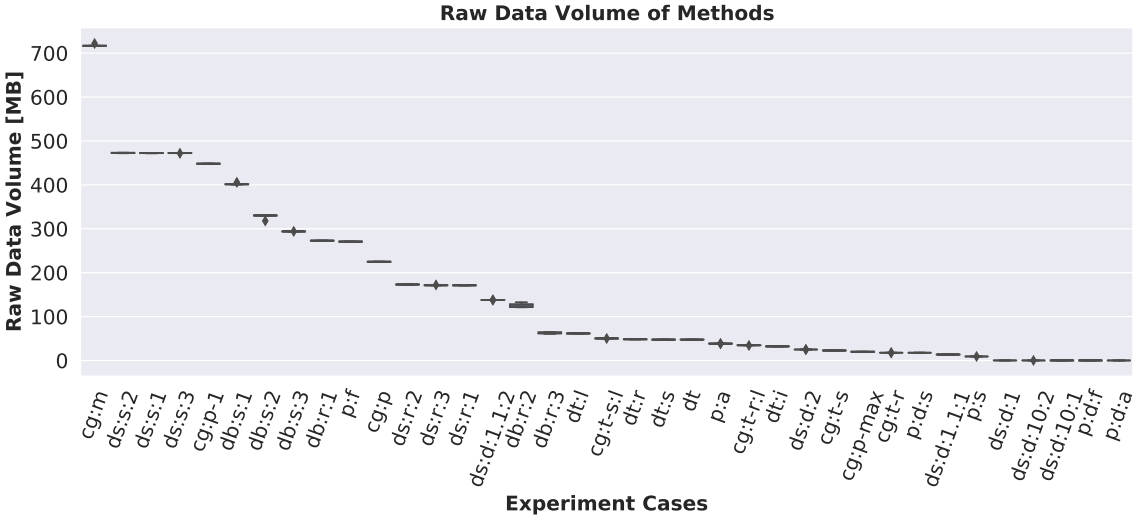

Figure C.5: Similarly to the Profiling Time (`PT`) results in the previous box plot C.4, also the variance of Raw Data Volume (`RDV`) is extremely low — however, such behaviour is expected, and as such, this Figure is more of a sanity check.

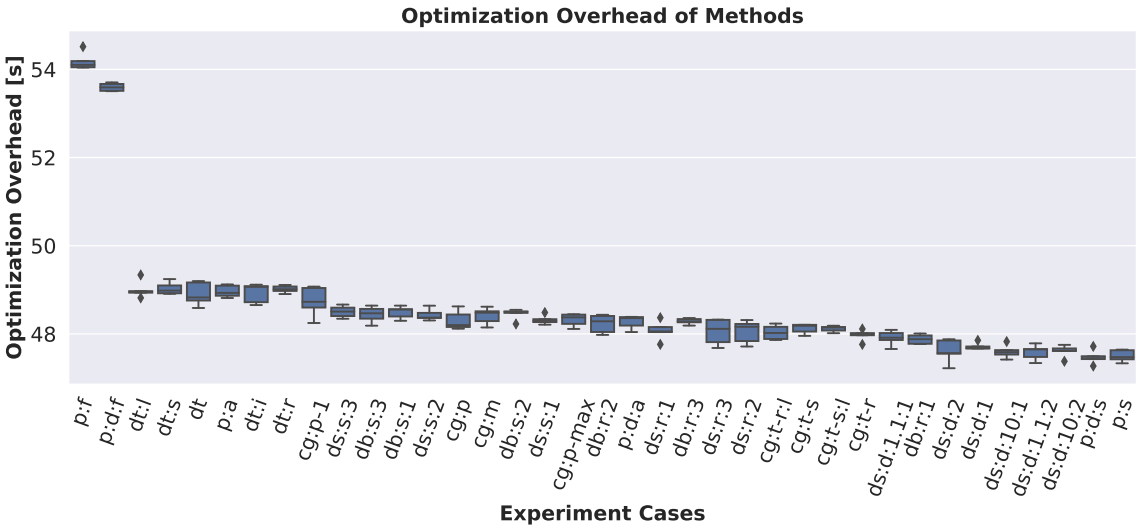

Figure C.6: Compared to the previous two box plots, the Optimization Overhead (OO) may seem to show much more varying results, which can be, however, upon closer inspection explained by the notably more detailed scale on the Y axis.

Appendix D

Comparison Tables

Here we present the full comparison Tables that compare the **M_PL**, **M_PT**, **M_RDV** and **M_FC** values for non-optimized collection and a majority of the experiment cases (excluding only the subsequent iteration steps of iterative methods). Note that $\Delta\%$ represents by how much % has the metric been improved compared to the `no-opt` case.

Table D.1: Comparison of the selected metrics for the $\{stap, cache\_off\}$ and $\{stap, cache\_on\}$ configurations of the `CPython` evaluation results.

Method	SystemTap: cache off				SystemTap: cache on			
	PL [Δ%]	**PT** [Δ%]	**RDV** [Δ%]	**FC** [%]	**PL** [Δ%]	**PT** [Δ%]	**RDV** [Δ%]	**FC** [%]
no-opt	0.0	0.0	0.0	51.6	0.0	0.0	0.0	49.8
cg:m	259.2	343.6	77.6	56.2	259.2	127.7	74.4	55.2
cg:p	425.6	777.2	466.3	49.0	425.6	726.6	454.3	49.0
cg:p-1	291.5	482.9	184.0	54.5	291.5	289.2	178.9	53.1
cg:p-max	928.5	1278.3	6332.5	32.2	928.5	1953.9	6116.5	32.4
cg:t-s	554.8	1013.3	5494.4	47.8	554.8	1578.6	5298.4	48.0
cg:t-s:l	461.6	884.5	2435.6	48.7	461.6	1209.1	2425.8	48.9
cg:t-r	887.2	1268.3	7125.3	36.6	887.2	2084.5	7125.4	36.2
cg:t-r:l	736.2	1137.4	3564.3	38.6	736.2	1394.4	3500.3	38.2
db:s:1	266.4	511.7	217.2	56.9	266.4	382.4	211.7	57.0
db:r:1	276.7	573.5	366.3	55.6	276.3	583.1	350.8	55.7
ds:d:1	259.2	421.2	846695.0	0.3	259.2	1062.9	831597.1	0.3
ds:d:10:1	259.2	417.2	907376.5	0.0	259.2	1085.3	905852.5	0.0
ds:d:1.1:1	259.2	416.8	9256.1	10.4	259.2	1011.6	9133.3	10.3
ds:s:1	259.2	454.8	169.4	56.5	259.2	437.0	164.9	55.8
ds:r:1	259.2	485.0	643.3	56.8	259.2	746.5	630.2	56.7
dt	777.5	1123.7	2575.8	49.7	777.5	1087.0	2485.0	49.5
dt:l	709.7	1049.5	1967.7	50.2	709.7	933.6	1928.7	49.3
dt:i	982.4	1251.1	3866.7	47.9	982.4	1525.9	3790.4	47.9
dt:s	777.5	1123.4	2576.0	49.9	777.5	1085.0	2484.0	49.1
dt:r	777.5	1126.0	2534.3	49.9	777.5	1085.1	2484.0	49.3
p:d:b	887.2	1273.3	7125.3	36.4	887.2	2064.0	6996.4	36.2
p:b	938.5	1315.3	13787.9	35.7	931.8	2378.7	12626.5	36.3
p:d:a	429.6	690.0	1325130.4	0.1	429.6	1535.8	1279479.9	0.1
p:a	434.0	779.2	3211.6	50.9	434.0	1055.3	3151.8	50.7
p:d:f	344.5	538.6	1046599.1	0.3	344.5	878.1	1027937.1	0.3
p:f	348.9	573.9	369.9	52.0	348.6	344.4	348.2	51.6

Table D.2: Comparison of the selected metrics for the $\{stap, cache\_off\}$ and $\{ebpf, cache\_on\}$ configurations of the `CCSDS` evaluation results.

Method	SystemTap: cache off				eBPF: cache off			
	PL [Δ%]	**PT** [Δ%]	**RDV** [Δ%]	**FC** [%]	**PL** [Δ%]	**PT** [Δ%]	**RDV** [Δ%]	**FC** [%]
no-opt	0	0	0	52.2	0.0	0.0	0.0	37.3
cg:m	144.8	-6.1	-0.2	54.0	144.8	-4.2	3.2	43.9
cg:p	310.0	141.8	364.5	42.1	310.0	66.9	232.4	22.2
cg:p-1	310.0	141.0	364.6	42.1	310.0	66.6	232.4	19.4
cg:p-max	16300.0	284.0	124637452.9	0.0	16300.0	186.1	148242154.5	0.0
cg:t-s	446.7	284.1	330211.5	42.9	446.7	131.5	232365.0	7.7
cg:t-s:l	300.0	222.6	1424.1	43.2	300.0	122.1	226843.5	6.2
cg:t-r	1390.9	284.1	22900569.7	0.0	1390.9	154.9	14516203.9	0.0
cg:t-r:l	925.0	284.4	15186828.7	0.0	925.0	148.6	8612566.2	0.0
sb:c	507.4	180.7	2076.5	52.0	507.4	52.3	1174.4	26.1
sb:l	1071.4	212.8	2522810.0	25.0	1071.4	70.2	2260525.9	20.0
sb:q	1390.9	212.7	19225905.5	22.2	1390.9	71.5	15780527.1	14.3
db:s:1	209.4	219.1	1481.1	49.0	203.7	101.3	2437.5	20.5
db:r:1	234.7	226.8	1858.9	44.4	228.0	107.5	4678.6	12.5
ds:d:1	144.8	146.4	2998.8	54.0	144.8	35.5	2992.2	46.4
ds:d:10:1	144.8	160.3	1018448.5	11.1	144.8	42.3	2242905.2	8.9
ds:d:1.1:1	144.8	37.8	82.8	55.6	144.8	30.4	631.3	32.7
ds:s:1	144.8	132.5	1089.0	55.6	144.8	34.2	1343.1	31.4
ds:r:1	144.8	165.7	11272.8	55.6	144.8	39.4	12289.2	31.4
dt	2242.9	150.7	386.2	57.1	2242.9	86.0	255.9	57.1
dt:l	1540.0	50.9	55.9	70.0	1540.0	70.3	179.2	66.7
dt:i	2242.9	150.3	385.8	57.1	2242.9	86.1	255.9	57.1
dt:s	2242.9	150.9	386.2	71.4	2242.9	86.2	243.6	57.1
dt:r	2242.9	150.0	397.1	57.1	2242.9	85.7	243.6	57.1
dp	-	-	-	-	0.0	35.4	106.9	40.7
dp:l	-	-	-	-	0.0	55.2	230.6	47.5
dp:h	-	-	-	-	0.0	-3.6	-0.5	39.0
dp:r	-	-	-	-	0.0	48.8	206.6	49.2
ts	-	-	-	-	0.0	19.5	186.6	46.7
ts:2	-	-	-	-	0.0	16.2	119.3	38.2
ts:4	-	-	-	-	0.0	17.5	130.0	38.0
p:d:b	1390.9	283.5	23496036.7	0.0	1390.9	155.1	14516203.9	0.0
p:b	1390.9	282.6	22900569.7	0.0	1390.9	154.8	15480894.9	0.0
p:d:a	382.4	233.0	14634.5	37.5	382.4	98.8	27307.4	27.6
p:a	397.0	241.3	4873.8	38.7	382.4	99.4	14974.4	11.1
p:d:f	530.8	178.2	14737.7	54.2	530.8	46.9	27133.3	45.5
p:f	583.3	184.3	4845.7	63.6	556.0	47.9	15085.3	21.1

Appendix E

Storage Medium

We have enclosed a storage medium (DVD) with this Thesis. The DVD contains two notable folders:

- The `latex` folder contains an electronic copy of the Thesis in both PDF and source code format.
- The `perun` folder. Since the focus of the Thesis — designing and implementing optimization techniques within the Perun framework — is tightly interconnected with the Perun internals, it is not feasible to extract and present only the source code created within this work. Hence the `perun` folder contains the full source code of the Perun framework. However, we also included the project respository (Git) with only two relevant branches: *develop* and *collect-optimizations* where the latter contains only the code created during the course of this Thesis. Using the appropriate tools (e.g., git show <commit>), one can precisely inspect the individual changes relevant to this Thesis. Moreover, the *collect-optimizations* branch is also available online at: https://github.com/JiriPavela/perun/tree/collect-optimizations.