VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INTELIGENTNÍCH SYSTÉMŮ
DEPARTMENT OF INTELLIGENT SYSTEMS

ANALÝZA VÝKONU PROGRAMŮ V JAZYCE PYTHON

PERFORMANCE ANALYSIS OF PYTHON PROGRAMS

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — **MATEJ ALEXEJ HELC**
AUTHOR

VEDOUCÍ PRÁCE — **Ing. JIŘÍ PAVELA**
SUPERVISOR

BRNO 2024

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

Zadání bakalářské práce

155172

Ústav:	Ústav inteligentních systémů (UITS)
Student:	**Helc Matej Alexej**
Program:	Informační technologie
Název:	**Analýza výkonu programů v jazyce Python**
Kategorie:	Analýza a testování softwaru
Akademický rok:	2023/24

Zadání:

1. Seznamte se s projektem Perun (správcem výkonnostních profilů) a s principy analýzy výkonu programů.
2. Seznamte se s metodami instrumentace programů, s možnostmi měření spotřeby zdrojů (doba běhu funkcí, spotřeba paměti, apod.) a existujícími nástroji pro měření výkonu programů v jazyce Python.
3. Navrhněte a implementujte nástroj, který bude měřit spotřebu alespoň jednoho zdroje v Python programech. Rozhraní nástroje navrhněte a implementujte s ohledem na požadavky projektu Perun.
4. Prozkoumejte možnosti asociace naměřené spotřeby vzhledem ke konkrétním prvkům programu (např. k funkcím, základním blokům nebo řádkům kódu).
5. Navrhněte a implementujte vhodnou vizualizaci pro interpretaci naměřených dat (např. flame graph nebo tree view), případně využijte a rozšiřte alespoň dvě z již existujících vizualizací v nástroji Perun.
6. Demonstrujte řešení na alespoň jednom netriviálním projektu.

Literatura:

- Oficiální stránky projektu Perun: https://github.com/Perfexionists/perun
- Bryant M. and Dobke A. (2020). FunctionTrace, a graphical Python profiler: https://functiontrace.com/
- Gregg, B. (2020). Systems Performance, (2nd ed.). Pearson. ISBN: 9780136821694.

Při obhajobě semestrální části projektu je požadováno:
První dva body zadání.

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

Vedoucí práce:	**Pavela Jiří, Ing.**
Konzultant:	Fiedor Tomáš, Ing., Ph.D.
Vedoucí ústavu:	Hanáček Petr, doc. Dr. Ing.
Datum zadání:	1.11.2023
Termín pro odevzdání:	9.5.2024
Datum schválení:	6.11.2023

Abstrakt

Cieľom tejto práce je rozšíriť verzovací systém Perun o nový modul, ktorý je navrhnutý na profilovanie Python programov. Tento modul profiluje a analyzuje kľúčové metriky, ako je čas vykonávania jednotlivých funkcií programu, vrátane funkcií z volaných knižníc. Profiler dokáže tieto dáta nielen zbierať, ale aj vizualizovať formou FlameGraphu. Tento prístup umožňuje vývojárom hlbšie pochopenie výkonnosti ich programov a podporuje efektívnejšiu optimalizáciu.

Abstract

The goal of this work is to extend the Perun versioning system by adding a new module designed for profiling Python programs. This module profiles and analyzes key metrics, such as the execution times of individual program functions, including functions from called libraries. The profiler can not only collects this data, but also visualize it in the form of a FlameGraph. This approach provides developers with a deeper understanding of their programs' performance and facilitates more efficient optimization.

Kľúčové slová

Profilovanie, Python, výkon, Perun, analýza programov, sys.monitoring, inštrumentácia

Keywords

Profiling, Python, performance, Perun, program analysis, sys.monitoring, instrumentation

Citácia

HELC, Matej Alexej. *Analýza výkonu programů v jazyce Python.* Brno, 2024. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Jiří Pavela

Analýza výkonu programů v jazyce Python

Prehlásenie

Prehlasujem, že som túto bakalársku prácu vypracoval samostatne pod vedením Ing. Jiřího Pavelu. Ďalšie informácie mi poskytol Ph.D. Tomáš Fiedor. Uviedol som všetky literárne pramene, publikácie a ďalšie zdroje, z ktorých som čerpal.

....
Matej Alexej Helc
8. mája 2024

Poďakovanie

Chcel by som vyjadriť vďaku vedúcemu tejto práce, Ing. Jiřímu Pavelovi a technickému konzultantovi, Ing. Tomášovi Fiedorovi, Ph.D., za ich pomoc, trpezlivosť, odborné rady a konzultácie. Ich cenné pripomienky k textu práce a k samotnému vývoju nástroja výrazne prispeli k tvorbe tejto práce.

Obsah

1 Úvod **2**

2 Analýza programov **4**
2.1 Statická analýza a dynamická analýza 4
2.2 Profilovanie 5

3 Perun **9**
3.1 Úvod 9
3.2 Architektúra 12

4 Možnosti inštrumentácie Python programov **15**
4.1 Štandardné moduly Pythonu 15
4.2 Vzorkovacie profilery 18
4.3 `Pyflame` 19
4.4 `Yappi` 20
4.5 `Scalene` 20
4.6 `FunctionTrace` 20
4.7 Ostatné profilery 21

5 Návrh a implementácia **23**
5.1 Analýza požiadaviek 23
5.2 Návrh 24
5.3 Implementácia 26
5.4 Známe obmedzenia 31

6 Vizualizácia **33**
6.1 Návrh 33
6.2 Implementácia 34
6.3 Známe obmedzenia 35

7 Experimenty **36**
7.1 Metodológia 36
7.2 Experiment 1: Réžia profilovania 36
7.3 Experiment 2: Profilovanie modulu parser.py 37
7.4 Experiment 3: Výkon a optimalizácie 38

8 Záver **43**

Literatúra **44**

Kapitola 1

Úvod

Vo svete softvérového inžinierstva je Python uznávaný pre svoju jednoduchú čitateľnosť a flexibilitu, čo z neho robí veľmi produktívny jazyk v porovnaní s inými jazykmi, ako sú napríklad C alebo Java. Dôvodom je jeho schopnosť použiť menej kódu a ponúka množstvo užitočných knižníc, najmä v oblastiach akými sú strojové učenie alebo vývoj webov. Jeho popularita vzrástla vďaka dynamickým vlastnostiam, rozsiahlej štandardnej knižnici, bohatému spektru frameworkov z rôznych oblastí podporených obrovskou komunitou.

Aj napriek tomu, že Python prináša množstvo výhod, vývojári sa často stretávajú s výkonnostnými výzvami, najmä čo sa týka spotreby pamäte a rýchlosti vykonávania programu. Tieto problémy sú obzvlášť kritické v aplikáciách, ktoré si buď vyžadujú rýchle spracovanie dát, alebo v aplikáciach v oblasti strojového učenia.

Vývoj produktu nekončí len pri výbere vhodného programovacieho jazyka. Tvorba výsledného produktu spočíva v rôznych fázach. Jednou z dôležitých fáz je testovanie, ktoré hrá dôležitú rolu vo vývojovom procese. Testovaním vývojári alebo testeri overujú nie len funkčnosť a bezpečnosť produktu, ale aj výkonnosť a spotrebu zdrojov programu (efektívnosť), na ktorých identifikáciu slúži tzv. profilovanie.

Ide o metódu dynamickej analýzy, ktorá sa používa pri meraní rôznych aspektov programu, ako je napríklad časová náročnosť alebo spotreba pamäte. Na túto analýzu nám slúžia profilovacie nástroje, tzv. profilery, ktoré dokážu zaznamenávať potrebné informácie o vykonávaní programu, ktoré následne interpretujú. Na základe týchto informácii vývojári dokážu odhaliť výkonnostné nedostatky. Umožňuje im identifikovať kusy kódu, na konkrétnych miestach v zdrojovom kóde v závislosti od granularity daného profileru, čo môže viesť k výraznému zlepšeniu celkového výkonu programu.

Profilovanie je kľúčové pri identifikácii skrytých výkonnostných problémov, ako napríklad pri probléme s `ctypes` v Python 3.11, ktorý spôsobil až 8% zhoršenie réžie vo výkone pri volaní funkcie oproti predchádzajúcej verzii[1]. Prípadne výkonnostné problémy spojené s *Global Interpreter Lock* (GIL, globálny zámok interpretu)[2], ktorý slúži na synchronizáciu prevádzania kódu v jednotlivých vláknach tak, že v jednom momente je kód vykonávaný iba v jednom vlákne.

V uvedení novej verzie Python 3.12 bolo predstavené nové monitorovacie *API*API (aplikačné programovacie rozhranie) `sys.monitoring`. To prináša viaceré výhody oproti existujúcim profilerom. Tento nástroj umožňuje flexibilné monitorovanie rôznych udalostí pri vykonávaní programu a umožňuje nízkoúrovňovú granularitu sledovania. Týmto spôsobom

[1] `https://github.com/python/cpython/issues/92356`

[2] Napríklad `https://github.com/pytorch/pytorch/issues/77139`

minimalizuje vplyv na výkon aplikácie, čo vývojárom pomáha efektívnejšie identifikovať výkonnostné problémy a umožňuje lepšie prispôsobenie zbieraných dát. Bližie informácie o tomto nástroji a porovnanie s existujúcimi profilermi je popísané v kapitolách 4 a 5.

Cieľom tejto bakalárskej práce je vytvorenie nové modulu, ktorý bude rozširovať systém pre verzovanie výkonnostných profilov nástroja Perun, ktorý je vyvíjaný skupinou *VeriFit* na Fakulte informačných technológií VUT v Brne. Okrem toho sa táto práca zaoberá aj analýzou existujúcich riešení (kapitola 4) s cieľom identifikovať oblasti, v ktorých môže byť Perun unikátny. Vzhľadom na to, že Perun je multiplatformový, je nevyhnutné, aby touto vlastnosťou disponoval aj nový modul. To znamená, že modul je navrhnutý a implementovaný tak, aby fungoval na rôznych operačných systémoch, ako sú Windows a Linux. Práca sa tiež sústredí na praktické aspekty implementácie, vrátane výberu vhodných nástrojov a technológií pre vývoj modulu, jeho testovania a validácie.

Kapitola 2

Analýza programov

Táto kapitola poskytuje detailný prehľad metodík na analýzu programov s dôrazom na rozbor ich efektivity. Detailne popisuje princípy profilovania, ktoré zahrňujú rôzne metódy zberu dát potrebných na meranie výkonnosti programu. Ďalej popisuje kľúčové parametre a metriky, ktoré je možné získať profilovaním.

Analýza programov je komplexný proces, ktorého cieľom je získať hlbšie poznatky o kóde a jeho vlastnostiach. Rôzne prístupy k analýze môžu odhalovať chyby, identifikovať bezpečnostné riziká a prispieť k celkovému zlepšeniu kvality programu. Táto analýza je nevyhnutná pre údržbu programu a taktiež hrá kľúčovú rolu pri jeho optimalizáci.

Analýza programu môže byť vykonávaná bez spustenia programu (statická analýza programu) alebo počas jeho vykonávania (dynamická analýza) [15].

2.1 Statická analýza a dynamická analýza

Statická analýza je proces, ktorý umožňuje analyzovať zdrojový kód bez jeho spustenia. Jednou z jej hlavných výhod je schopnosť detekovať chyby ešte pred spustením programu a nezávisle od jeho vstupu. Zaoberá sa analýzou zdrojových alebo strojových kódov, prípadne nejakým medzikódom (napríklad LLVM IR). Pomáha k lepšiemu porozumeniu štruktúry a závislostí v kóde, čo môže usmerniť vývojárov k dodržiavaniu osvedčených postupov a štandardov v oblasti písania kódu (tzv. štábna kultúra), ktoré sú nevyhnutné pre vývoj kvalitného softvéru.

V rámci pokročilejších prístupov, formálna statická analýza pridáva hlbšiu vrstvu kontroly, zameranú na matematické overovanie správnosti kódu. Táto technika používa logické vzorce a formalizované metódy overenia na detekciu potenciálnych problémov, ktoré by mohli byť prehliadnuté pri bežnej analýze. Medzi nástroje patriace do tejto kategórie sa radia Coq alebo Z3, ktoré umožňujú vývojárom vykonať formálne dôkazy správnosti algoritmov a systémových modelov. Tento druh analýzy je dôležitý predovšetkým v oblastiach, kde sú požiadavky na bezpečnosť a spoľahlivosť na najvyššej úrovni, ako napríklad v oblasti letectva, kozmonautiky a inej kritickej infraštruktúry.

Statickú analýzu využívajú najmä nástroje na kompiláciu kódu–kompilátory, ktoré najčastejšie odhaľujú syntaktické (porušenie pravidiel jazyka) a niektoré sémantické chyby (napr. sčítanie reťazca a čísla v jazyku, ktorý túto operáciu nepodporuje). Statická analýza dokáže odhaliť napríklad nadbytočné výpočty, nadmernú alokáciu pamäte, neoptimalizované slučky, mŕtvy t.j. nepoužívaný kód a podobne.

Na analýzu výkonu programu existujú nástroje, ktoré dokážu identifikovať mŕtvy kód, ako napríklad Python nástroj Vulture [1], alebo nástroj Radon [2], ktorý dokáže vypočítať napríklad cyklomatickú zložitosť [3].

Dynamická analýza je proces zbierania dát priamo počas vykonávania programu. Zozbierané dáta (namerané metriky ako čas behu, spotreba pamäte a podobne) môžu byť následne analyzované buď v priebehu, alebo po skončení vykonávania programu. Na rozdiel od statickej analýzy, ktorá poskytuje analýzu kódu bez jeho spustenia, dynamická analýza odhaľuje správanie softvéru v reálnom čase. Dynamická analýza sa obmedzuje len na scenáre, ktoré boli aktívne vykonané počas konkrétneho vykonávania programu, zatiaľ čo statická analýza dokáže pokryť všetky možné scenáre programu.

Hlavnou výhodou dynamickej analýzy je jej schopnosť odhaľovať chyby a problémy, ktoré nie sú viditeľné pri samotnom pohľade do zdrojového kódu, ako napríklad úniky pamäte. Dynamická analýza tiež pomáha pri odhaľovaní bezpečnostných hrozieb.

Pri dynamickom testovaní je možné využiť nástroje zvané profilery, ktoré poskytujú vývojárom podrobné informácie o vykonávaní programu, ako napríklad čas trvania jednotlivých operácií alebo využitie pamäte.

Dynamická analýza je kľúčovým nástrojom v procese vývoja softvéru, ktorý zabezpečuje nielen jeho bezchybnosť, ale aj efektívnosť. Okrem profilovania zahŕňa aj rôzne druhy integračných, systémových a akceptačných testov. Tieto testy poskytujú širší pohľad na možné chyby a nedostatky.

Hlavnou nevýhodou dynamického testovania je zvýšená réžia v podobe časovej a zdrojovej náročnosti, ktorá závisí od použitého nástroja. Medzi ďalšie nevýhody patria nepresnosti spôsobené sledovaním alebo nutnosť pokrytia kódu [2].

2.2 Profilovanie

Profilovanie je proces, ktorý sa zaoberá zbieraním dát o vykonávaní programu pomocou nástrojov známych ako profilery. Tieto nástroje vykonávajú vzorkovanie a iné formy sledovania, aby získali podrobné informácie o výkonnosti a efektivite softvéru. Samotné profilovanie je zamerané na zber dát, ktoré môžu byť následne vizualizované a analyzované, v závislosti od použitého profileru. Výstup profileru môžu následne vývojári použiť na optimalizáciu [4].

Efektívne profilovanie poskytuje základ pre zlepšenie výkonu softvéru, čo vedie nielen k rýchlejšiemu a efektívnejšiemu programu, ale aj k zlepšeniu užívateľského zážitku a zníženiu réžie spôsobenej samotným profilovaním. Profilovanie teda predstavuje veľmi cenný krok v procese vývoja softvéru, ktorý pomáha zabezpečiť, že výsledný produkt je nielen kvalitný, spoľahlivý a bezpečný ale aj efektívny. Profilovanie a následná optimalizácia kódu sú nevyhnutné pre dosiahnutie týchto výsledkov.

2.2.1 Profilery

Profiler je nástroj používaný v softvérovom inžinierstve na sledovanie vybraných metrík, ako sú napríklad počet volaní funkcií a čas ich vykonávania. Existuje niekoľko spôsobov,

[1] `https://github.com/jendrikseipp/vulture`

[2] `https://github.com/rubik/radon`

[3] Softvérová metrika, ktorá určuje zložitosť programu meraním počtu lineárne nezávislých ciest v zdrojovom kóde [1].

ako môže profiler získavať požadované dáta, pričom metóda ich zberu určuje jeho typ. Toto sú tri najrozšírenejšie metódy:

1. **Sledovanie udalostí (tzv. *Event-based* alebo *Tracing* profilery):** Táto metóda zbiera dáta o konkrétnych udalostiach, ktoré sa vykonajú počas vykonávania programu, ako napríklad volania funkcií, alokácia objektov, výjimky a podobne. Má vyššiu detailnosť zberu dát v porovnaní so vzorkovacími profilermi, čo vedie k potenciálne podrobnejším poznatkom, avšak zároveň aj k vyššej réžii. Napríklad, pri sledovaní vykonávania nejakej funkcie, tracing profiler zachytáva jej vstup a výstup (prípadne výjimky) a na základe týchto udalostí vypočíta metriky. Táto metóda zberu dát nezasahuje priamo do zdrojových kódov a je obzvlášť užitočná v komplexných viacvláknových aplikáciách alebo aplikáciách s vysokým výkonom. Kvôli vysokej réžii tracing profilera je odporúčané najskôr využiť vzorkovací profiler a až v prípade, že s jeho pomocou nie je možné problém vyriešiť, siahnuť po detailnejšom tracing profileri [20].

2. **Vzorkovanie (*Sampling*):** Táto metóda využíva periodické zachytávanie stavu programu prostredníctvom vytvárania vzoriek (*snapshots*). Tieto vzorky umožňujú analyzovať, ktoré časti programu spotrebujú najviac prostriedkov. Avšak je dôležité zdôrazniť, že metóda vzorkovania nemusí byť vždy presná. Výkonnostné problémy, ktoré nastanú medzi vzorkami, môžu uniknúť detekcii, a malé funkcie, ktoré sú volané a ukončené v rámci intervalov vzorkovania, môžu byť nesprávne interpretované ako dlhšie funkcie, čo vedie k potenciálnym chybám v identifikácii výkonnostných problémov. Výsledkom tejto metódy je štatistický odhad správania programu, pričom je potrebné vziať do úvahy, že výsledky môžu byť ovplyvnené externými faktormi, ako sú systémové prerušenia (*system interrupts*), vyrovnávacia pamäť (*cache*) a ďalšie. Pri opakovaných spusteniach programu sa výsledky môžu líšiť. Profilery využívajúce túto metódu zberu dát sú tiež známe aj ako štatistické profilery. Hlavnými výhodami tejto metódy je, že nezasahuje do zdrojového kódu, má nízku réžiu, čo zaisťuje minimálny vplyv na rýchlosť vykonávania programu [22].

3. **Inštrumentácia programu:** Ide o vkladanie dodatočného kódu do programu. Existujú dva hlavné typy inštrumentačných profilerov:

   (a) **Profiler binárne spustiteľného kódu (*binary profiler*):** Funguje priamo v prostredí vykonávania programu, čo umožňuje jeho analyzovanie bez nutnosti prístupu k jeho zdrojovému kódu. Pri kompilácii programu sa inštrumentačný kód vkladá buď priamo do binárne spustiteľného kódu (tzv. *binary code*), alebo pri načítaní obrazu do pamäte. Je ideálnym na analýzu programov, pri ktorých nie je užívateľovi dostupný zdrojový kód.

   (b) **Profiler zdrojového kódu (*source-code profiler*):** Vyžaduje priamy prístup ku zdrojovému kódu. To umožňuje užívateľovi upravovať kód podľa vlastných potrieb a špecifikácií, čo je nevyhnutné na presnú a efektívnu analýzu jeho výkonu. Inštrumentácia musí prebiehať pred spustením programu, teda pred jeho kompiláciou, prípadne v počas jej priebehu tzv. *compile–time instrumentation.*

   Pridávanie dodatočného kódu do programu zvyšuje jeho réžiu, čo môže v niektorých prípadoch zapríčiniť nie celkom presné výsledky. Táto metóda aj napriek tomu môže byť veľmi presná s vysokou mierou detailov v závislosti od efektivity akou je vykonávaná.

Príklady jednotlivých druhov profilerov na analýzu programov v jazyku Python sa nachádzajú v kapitole 4.

2.2.2 Granularita

V oblasti profilovania programov je možné analyzovať výsledky na rôznych úrovniach. Je však dôležité poznamenať, že nie každý profilovací prístup je schopný dosiahnuť požadovanú jemnosť profilovania, kvôli obmedzeniam spojeným s typom použitého profileru, čo je popísané v časti 2.2.1. Významným aspektom, ktorý si treba uvedomiť je, že rýchlosť profilovaného programu vždy predstavuje kompromis medzi úrovňou detailu a požadovanou presnosťou. Granularitu môžeme rozdeliť do viacerých úrovní:

- **Riadková (*line-by-line*) granularita:** Táto úroveň granularity sa zameriava na detailné skúmanie jednotlivých riadkov programu. Poskytuje extrémne detailné informácie, avšak toto detailné sledovanie môže mať svoju cenu v podobe vysokej réžie, čo môže ovplyvniť celkovú výkonnosť profilovaného programu, a teda aj samotné výsledky profilovania, najmä pri komplexných aplikáciách.
- **Funkcionálna granularita:** Profilovanie na úrovni funkcií (metód) je menej detailné v porovnaní s riadkovou granularitou, čo má za následok menšiu réžiu, to ale neznamená, že nemôže ovplyvniť výsledok profilovania podobne ako riadková granularita.
- **Inštrukčná granularita:** Profilovanie na úrovni jednotlivých inštrukcií umožňuje analyzovať nízkoúrovňové operácie vykonávané procesorom. Tento prístup poskytuje veľmi presný pohľad na vykonávanie programu, môže však výrazne zvýšiť réžiu a spomaliť profilovaný program.
- **Granularita základných blokov (tzv. *basic-blocks*):** Táto úroveň sa zaoberá skupinami inštrukcií, ktoré spolu vstupujú a vystupujú bez rozhodovacích bodov, čo umožňuje efektívnejšie sledovanie programu v porovnaní s inštrukčnou granularitou. Je užitočná na optimalizáciu výkonu, pretože identifikuje horúce miesta (tzv. *hotspots*) v kóde.
- **Modulová granularita:** Profilovanie na úrovni modulov alebo knižníc umožňuje sledovať výkon a správanie väčších celkov programu, čo je ideálne na identifikáciu problémov na vyšších úrovniach architektúry aplikácie. Táto granularita je menej detailná, ale poskytuje širší pohľad na systémové interakcie a závislosti.
- **Ďalšie úrovne granularity:** Medzi ďalšie úrovne granularity patrí profilovanie na úrovni vlákien alebo procesov, systémová granularita, prípadne užívateľom definovaná granularita . . .

2.2.3 Metriky

Pri výkonnostnej analýze je dôležité poznať a monitorovať rôzne metriky, ktoré môžu ovplyvniť rýchlosť vykonávania programu a efektívnosť využitia zdrojov. Profiler by mal zbierať a analyzovať tieto metriky, aby umožnil identifikáciu možných problémov a optimalizačných príležitostí. Tu sú niektoré základné metriky:

- **Čas *Wall-clock*:** Tento parameter zahŕňa celkový čas potrebný na vykonanie určitej funkcie, vrátane času stráveného vykonávaním funkcie a času stráveného v systémo-

vých volaniach (napríklad I/O), uspaním plánovačom, čakania na sieťové odpovede alebo na uvoľnenie zdrojov od iných procesov ...

- **CPU čas:** Tento čas predstavuje dobu, počas ktorej procesor aktívne vykonáva inštrukcie danej funkcie, bez zahrnutia času stráveného v systémových volaniach alebo iných réžijných operáciách.
- **Počet volaní jednotlivých funkcií, základných blokov a inštrukcií ... :** Táto metrika poskytuje informácie o volaniach jednotlivých funkcií, základných blokov, inštrukcií a podobne, ktoré sú volané počas vykonávania programu.
- **Využitie zdrojov (tzv. *utilization*):** Táto metrika uvádza mieru zaťaženia zdrojov, napríklad procesoru alebo grafickej karty. Je typicky udávaná v percentách.
- **Využitie pamäte (*memory usage*):** Monitorovanie využitia pamäte je kľúčové pri identifikácii únikov pamäte (*memory leaks*) a efektívnosti alokácie pamäte. Príliš vysoké využitie pamäte môže viesť k problémom s výkonnom alebo k vyčerpaniu dostupnej pamäte.

Okrem týchto základných metrík je možné tiež monitorovanie ďalších metrík podľa špecifických požiadaviek projektu. To môže zahŕňať napríklad:

- **Spotrebu energie:** Napríklad pre mobilné aplikácie je dôležité monitorovať spotrebu energie, aby sa minimalizoval vplyv na batériu používateľa.
- **Vyťaženie siete:** Pri softvéri, ktorý komunikuje cez sieť, je kľúčové monitorovať vyťaženie siete, vrátane rýchlosti nahrávania a sťahovania (*upload* a *download*). Tieto údaje pomáhajú identifikovať potenciálne úzke miesta v sieťovej komunikácii a umožňujú optimalizáciu prenosovej kapacity a odozvy.

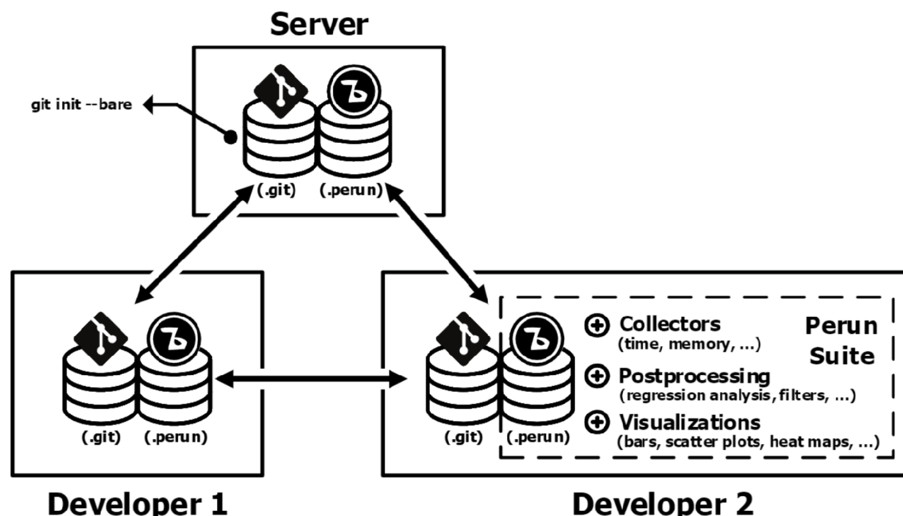
Kapitola 3

Perun

Táto kapitola sa zameriava na nástroj Perun, pokročilý verzovací systém na správu výkonnosti projektov. Cieľom je predstaviť Perun nie len ako nástroj na sledovanie výkonu aplikácií, zároveň aj objasniť jeho architektúru a mechanizmy, ktoré umožňujú efektívne testovanie a analýzu programov naprieč verziami. Perun je nie len správca profilov, ale aj sada nástrojov na profilovanie, trasovanie, výkonnostné testovanie, následné spracovanie a vizualizáciu. Základným zdrojom faktov v tejto kapitole je najnovšia dokumentácia Perunu [9] a ostatné práce, ktoré prispeli k jeho vývoju [16, 14, 13]. V úvode kapitoly je uvedený stručný prehľad funkcií, možností a výhod, ktorý tento nástroj poskytuje. Nasledujúca sekcia je venovaná popisu jeho architektúry, je v nej vysvetlené, ako sú jednotlivé komponentny Perunu prepojené a ako spolu spolupracujú.

3.1 Úvod

Perun je open-source nástroj na správu výkonnosti softvéru, ktorý je vyvíjaný výskumnou skupinou VeriFIT za účelom automatizácie sledovania výkonnosti softvérových projektov naprieč verziami. V súčasnosti má podporu iba pre pár programovacích jazykov (C/C++ a C#), ale naďalej sa rozširuje. Aj napriek tomu, že je neustále v aktívnom vývoji, poskytuje širokú škálu funkcionalít. Jeho hlavnou prednošťou je integrácia s VCS (*Version Control System*, tj. verzovací systém napr. Git), kde Perun slúži ako rozhranie na automatické testovanie a správu výkonnostných profilov jednotlivých verzií projektu, následné spracovanie vygenerovaných profilov a ich efektívne vyhodnotenie. Perun zbiera a vyhodnocuje výkonnostné metriky každej verzie projektu, tj. v prípade zmeny v projekte v podobe *commitu* alebo *pull-requestu* s následným porovnaním s predchádzajúcou verziou. Vďaka automatizácii analýzy výkonnosti je možné jednoducho vykonávať regresné testy, ukladanie výkonnostných profilov každej verzie projektu umožňuje rýchle odhlaľovanie potenciálnych problémov bez nutnosti manuálneho zásahu. Perun je škálovateľný ako pre malé projekty, vyvíjané malými tímami prípadne jednotlivcami, ako kompletné riešenie na ukladanie, automatizáciu a interpretáciu výkonnosti softvérového produktu, tak aj pre komplexné projekty ako úložisko. Obrázok 3.1 ilustruje model zamýšľaného využitia nástroja Perun pri vývoji projektu.



Obr. 3.1: Ilustrácia fungovania nástroja Perun paralelne s *VCS* (v tomto prípade Git) v priebehu vývoja projektu [9].

V porovnaní s manuálnym prístupom k správe výkonnosti projektu, Perun prináša niekoľko výhod oproti separátnemu použitiu databázy a *VCS*:

- **Zachovanie kontextu.** Perun integruje výkonnostné profily priamo do systému na správu verzií (*VCS*), čím zabezpečuje väzbu každého profilu s príslušnou verziou projektu. Tento prístup uchováva kľúčový kontext o konkrétnych verziách zdrojového kódu. Integrácia umožňuje vývojárom nie len jednoduchšie rozpoznať výkonnostné rozdiely naprieč verziami, ale aj možnosť efektívnejšej optimalizácie zdrojového kódu. Vďaka začleneniu historických dát o výkonnosti a zmenách v kóde (napr. *commit* alebo *pull-request*), je možné lepšie určiť oblasti vhodné na meranie a analýzu. To všetko vedie k podrobnejšiemu porozumeniu príčin výkonnostných problémov čo pomáha k ich efektívnejšiemu riešeniu.

- **Možnosť automatizácie.** Perun umožňuje automatizáciu zberu výkonnostných profilov prostredníctvom integrácie do *VCS*, čo umožuje automatizovať generovanie nových profilov s každou verziou projektu. Tento mechanizmus je inšpirovaný systémami na kontinuálnu integráciu a využíva konfiguračné súbory vo formáte *YAML*[1], čo predstavuje intuitívny formát na definovanie automatizovaných úloh. Integrácia Perunu do *VCS* umožňuje nastavenie tzv. háčikov (*hooks*), čo zaisťuje, že nebudú vynechané žiadne verzie projektu. Na základe týchto háčikov Perun dokáže identifikovať uverejnenie novej verzie projektu, vďaka čomu automaticky vykoná meranie výkonu a porovná profil s predchádzajúcimi verziami.

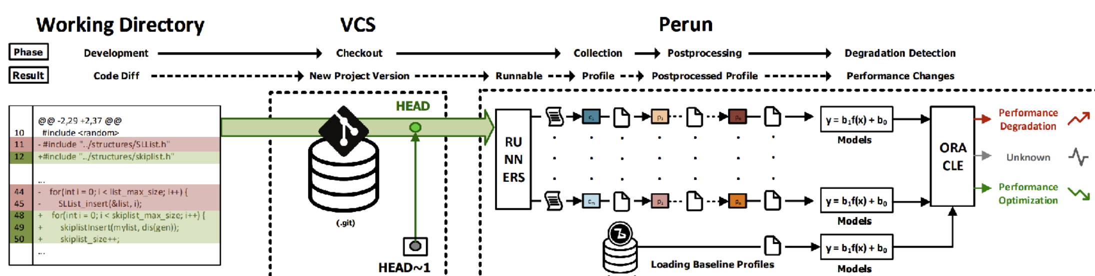
- **Vysoká genericita.** Kľúčovou prednosťou Peruna je jeho flexibilita a ľahká rozšíriteľnosť o nové moduly na zber dát, ich následné spracovanie alebo vizualizáciu, čo umožňuje jeho adaptáciu na rôzne typy projektov. Jeho modulárna architektúra podporuje jednoduchú integráciu nových komponentov, čím zvyšuje jeho využiteľnosť. Modularita a dizajn, ktoré každý modul považuje za základný stavebný blok, uľahčujú automatizáciu procesov. Minimálne požiadavky na rozšírenie a použitie jednotného

[1]Formát na serializáciu štruktúrovaných dát `https://yaml.org/`

formátu (*JSON*[2]) na ukladanie dát zjednodušujú integráciu nových modulov, bez zbytočných komplikácií.

- **Jednoduché použitie.** Perun je navrhnutý s dôrazom na intuitívne ovládanie pre užívateľov, ktorí sú zvyknutí na prácu s verzovacími systémami, predovšetkým na Git. Zahŕňa bežne používané príkazy ako napríklad *add*, *status*, *log* a *init*, čo umožňuje rýchle pochopenie tohto nástroja. Aktuálna verzia Peruna je prístupná pomocou príkazového riadku, ale prebieha aj vývoj grafického užívateľského rozhrania.

Obrázok 3.2 ilustruje, ako Perun pri každej novej verzii projektu vykonáva sériu úloh na zber, spracovanie a analýzu výkonnostných dát, a ich následné porovnanie s predchádzajúcimi verziami. Na základe výsledkov vyhodnotí, či došlo k degradácii alebo optimalizácii výkonu analyzovaného projektu.

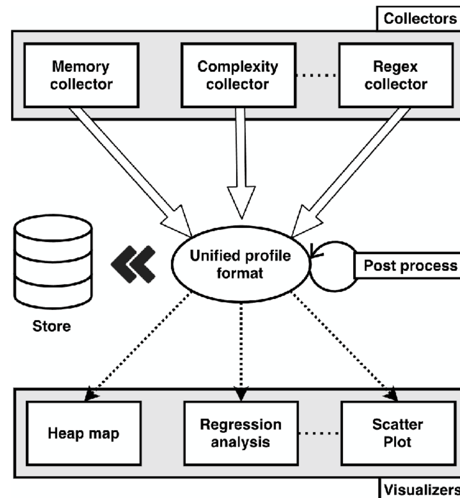


Obr. 3.2: Grafické zobrazenie bežného pracovného procesu nástroja Perun [16]. Pre každú novú verziu projektu spúšťa Perun radu úloh (tzv. *jobs*), ktoré zbierajú, spracovávajú a analyzujú dáta s cieľom určiť výkonnostné zmeny v projekte oproti predchádzajúcej verzii.

Životný cyklus výkonnostných profilov (obrázok 3.3) vyzobrazuje ako je profil vygenerovaný pomocou kolektorov (*collectors*) a následne ukladaný do úložiska (*store*, paralelne s *VCS*), ktoré je aktuálne v súborovom systéme, kde je komprimovaný pomocou *zlib*[3]. Uložené výkonnostné profily môžu byť ďalej upravované pomocou postprocesov alebo priamo interpretované pomocou rôznych vizualizačných metód. Podrobnejšie informácie nájdete v dokumentácii [9].

[2]Univerzálny formát na zápis dát `https://www.json.org/json-en.html`

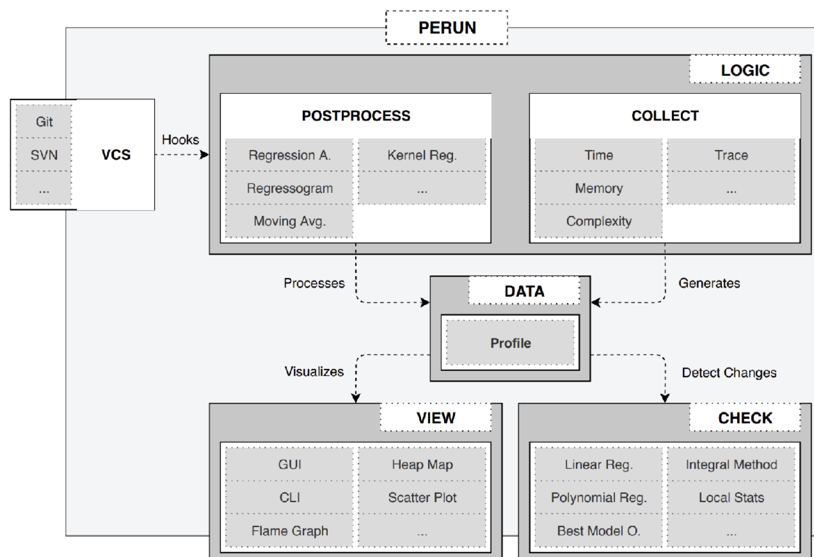
[3]Softvérová knižnica používaná na kompresiu súborov `https://www.zlib.net/`



Obr. 3.3: Grafické zobrazenie životného cyklu výkonnostného profilu [9]. Profil je vygenerovaný sadou kolektorov (*collectors*) ako napríklad *Memory, Complexity, Time collector*, … , ktoré sú ukladané v súborovom systéme. Uložené profily je možné následne interpretovať pomocou vizualizačných metód, napríklad *heap–map*, *flame graph*, …

3.2 Architektúra

Architektúra nástroja Perun je postavená na štyroch hlavných komponentoch: logika (*logic*), dáta (*data*), kontrola (*check*) a pohľad (*view*). Okrem týchto hlavných komponentov obsahuje Perun aj ďalšie menšie komponenty, ako sú napríklad *VCS*, šablóny a podobne. Táto sekcia sa zameriava na popis hlavných komponentov, ktoré sú ilustrované na obrázku 3.4.



Obr. 3.4: Zjednodušená schéma architektúry nástroja Perun ilustrujúca 4 hlavné komponenty spolu s *VCS* modulom [16].

Data. Tento komponent predstavuje základnú časť architektúry, poskytujúcu rozhranie na správu výkonnostných profilov, na ktoré sa opierajú všetky ostatné komponenty. Dáta (výkonnostné profily) sú unifikované vo formáte *JSON*, čo uľahšuje komunikáciu s ostatnými komponentmi.

Logic. Tento komponent je určený na tvorbu výkonnostných profilov s využitím radu kolektorov a v prípade potreby aj na ich ďalšie spracovanie pomocou postprocesov. Má na starosti automatizáciu a vyššiu logiku riadenia a generovania profilov. Medzi hlavné kolektory, ktoré Perun obsahuje patria:

- *Trace collector*: Kolektor trasovania zaznamenáva časovú náročnosť jednotlivých funkcií. Architektúra tohto kolektoru poskytuje užívateľom možnosť výberu z rôznych tzv. *engines*, ktoré na zber dát využívajú rozličné inštrumentačné frameworky, napríklad `SystemTap`[4] alebo `eBPF`[5].
- *Memory collector*: Kolektor spotreby pamäte zhromažďuje informácie o alokáciách pamäte a o jej celkovom využití. Zaznamenáva rôzne atribúty, ako sú typy alokácií pamäte alebo ich cieľové adresy. Na zber informácií využíva knižnicu `libunwind`[6] spolu s vlastnou knižnicou `libmalloc`.
- *Time collector*: Kolektor času zhromažďuje celkové doby vykonávania ľubovoľných príkazov. Je implementovaný ako jednoduchý obal nad nástrojom `time`.
- *Bounds collector*: Predstavuje metódu statickej analýzy, vypočítava amortizovanú časovú zložitosť v najhoršom prípade. Využíva techniku nástroja `Loopus`, ktorý je však obmedzený iba na analýzu celočíselných (*integer*) programov. Pre každú funkciu a každú slučku vypočíta symbolickú hranicu (napríklad $2 * n + max(0, m)$) a následne posúdi ich zložitosť použitím O-notácie (*Big O notation*)[7].

Medzi postprocesory, ktoré Perun obsahuje patria:

- *Normalizer Postprocessor* škáluje dáta do intervalu $(0, 1)$, so zámerom porovnať profily s rozdielnou záťažou programu alebo vstupnými parametrami.
- *Regression Analysis* ponúka niekoľko výpočtových metód a modelov na nájdenie vhodných lineárnych regresných modelov pre trendy v zachytených profilovacích zdrojoch.
- *Clusterizer* zhromažďuje zdroje do skupín na ďalšie spracovanie (napríklad Regresnou analýzou) alebo ich zhromažďuje do skupín s podobným využitím zdrojov.
- *Regressogram method* hľadá vhodné modely pre trendy v zachytených profilovacích zdrojoch s použitím konštantnej funkcie na jednotlivých častiach intervalu.
- *Moving Average Methods* je metóda štatistickej analýzy, ktorá analyzuje dátové body v zachytených profilovacích zdrojoch vytvorením série hodnôt na základe špecifickej agregačnej funkcie (najčastejšie priemer, prípadne medián) nad rôznymi podmnožinami celého súboru dát.

[4] `https://sourceware.org/systemtap/`
[5] `https://ebpf.io/what-is-ebpf/`
[6] `https://www.nongnu.org/libunwind/`
[7] `https://en.wikipedia.org/wiki/Big_O_notation`

Check. Obsahuje metódy na detekciu potenciálnych zmien vo výkonnosti projektu. Na vstupe očakáva dva výkonnostné profily: novú verziu projektu a jej predchodcu, ktoré sú následne porovnané. Na základe tohto porovnania vyhodnotí, či nové zmeny priniesli optiomalizáciu alebo degradáciu výkonu projektu. Aktuálne Perun obsahuje tri stratégie na detekciu výkonnostných zmien:

- *Best Model Order Equality*: Metóda založená na výsledkoch regresnej analýzy, ktorá skúma každú unikátne indetifikovanú skupinu zdrojov určí, či sa najlepší výkonnostný (alebo predikčný) model zmenil (berúc do úvahy lexikografické usporiadanie typov modelov), napríklad že sa najlepší model zmenil z lineárneho na kvadratický."[9]
- *Average Amount Threshold*: Metóda založená na porovnávaní priemerov. Porovnáva priemer cieľa s priemerom referenčnej hodnoty, v prípade, že pomer týchto priemerov prekročí stanovený prahový interval, metóda signalizuje zmenu.
- *Exclusive Time Outliers*: Zameriava sa na identifikáciu odchýlok v rozdieloch exkluzívnych časov funkcie. Na identifikáciu týchto odchýlok využíva tri štatistické metódy, ktoré umožňujú klasifikovať zmeny do troch rozličných kategórií v závislosti od metódy, ktorá odchýlku zaznamenala.

View. Poskytuje užívateľovi grafickú reprezentáciu nazbieraných dát v podobe rôznych grafov alebo tabuliek. Perun obsahuje niekoľko vizualizačných metód, napríklad: *Bars Plot*, *Flow Plot*, *Flame Graph*, *Scatter Plot* alebo výpis v podobe tabuľky.

Kapitola 4

Možnosti inštrumentácie Python programov

Táto kapitola poskytuje prehľad a analýzu najznámejších nástrojov na profilovanie Python programov, konkrétne Python programov bežiacich v interprete `cPython`. Ide o predvolenú a najpoužívanejšiu implementáciu jazyka Python, ktorá je napísaná v jazyku C. Táto kapitola je zameraná predovšetkým na profilery, ktoré monitorujú čas strávený volaniami funkcií. Je v nej popísaná funckionalita jednotlivých profilerov, ich obmedzenia, metódy, akými sú zbierané dáta a spôsoby ich interpretácie [23].

4.1 Štandardné moduly Pythonu

Táto sekcia sa zaoberá integrovanými (tzv. *built-in*) nástrojmi Pythonu, ktoré poskytujú základné možnosti profilovania a merania výkonu. Štandardné moduly umožňujú vývojárom identifikovať výkonostne úzke miesta vo svojich aplikáciách, monitorovať čas vykonávania funkcií a analyzovať správanie programu počas ich vykonávania. Primárne ide o *tracing (event-based)* profilery, okrem `time` a `timeit`, ktoré sú formou inštrumentácie zdrojového kódu. Podrobný popis ako tieto profilery fungujú je v sekcii 2.2.1.

4.1.1 `Profile` a `cProfile`

Moduly `profile` a `cProfile` sú základné nástroje na profilovanie Python programov. Ide o nástroje, ktoré sledujú všetky udalosti, akými sú volania funkcií, ich návraty, výjimky a presne určujú časové intervaly medzi týmito udalosťami. `cProfile` je efektívnejší a má menší vplyv na výkon profilovanej aplikácie, vďaka svojej implementácii v jazyku C založenej na `lsprof`. Je vhodný na podrobnejšie profilovanie malých aj veľkých aplikácií. `Profile` je implementovaný čisto v jazyku Python. Je možné považovať ho za malú podmnožinu `cProfile`. Je vhodnejší prevažne pre menšie skripty, v ktorých réžia nie je kritickým faktorom. Hlavnou nevýhodou týchto nástrojov je chýbajúca podpora viacvláknových (tzv. *multithreading*) a spôsob merania času vykonávania jednotlivých funkcií, ktorý sa vypočíta ako pomer celkového času trvania danej funkcie a počtu volaní tejto funkcie.

Tieto nástroje možno používať rôznymi spôsobmi, napríklad vkladaním inštrumentácie priamo do kódu alebo spustením profilera nad celým programom. Tento prístup umožňuje programátorom získať flexibilitu pri kontrole rozsahu profilovania. Napríklad, kým použitie `cProfile.run('my_function()')` umožní profilovať špecifickú funkciu, spustenie modulu

`cProfile` na úrovni príkazového riadku profiluje celý skript, čím poskytuje komplexnejší pohľad na výkon aplikácie [18].

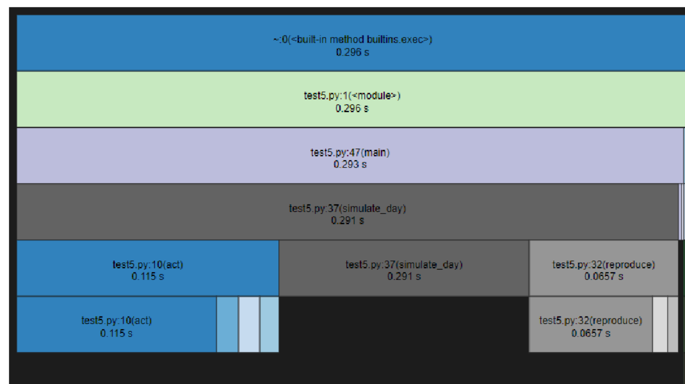
```
1940793 function calls (1940765 primitive calls) in 0.302 seconds

   Ordered by: internal time

   ncalls tottime percall cumtime percall filename:lineno(function)
       50 0.110 0.002 0.294 0.006 test5.py:37(simulate_day)
   596979 0.083 0.000 0.110 0.000 test5.py:10(act)
   596979 0.061 0.000 0.074 0.000 test5.py:32(reproduce)
   228146 0.010 0.000 0.010 0.000 test5.py:24(eat_plant)
   206153 0.009 0.000 0.009 0.000 test5.py:28(eat_herbivore)
   162680 0.007 0.000 0.007 0.000 test5.py:21(photosynthesize)
   ...
```

Výpis 4.1: Príklad výstupu cProfile.

Nevýhodou týchto nástrojov je, že ich výstup je vo forme zoznamu ako je ukázané vo výpise 4.1, ktorý nie je priamo vhodný na vizuálnu analýzu. Existujú však knižnice, ako napríklad `SnakeViz`[1], ktoré umožňujú vizualizáciu týchto dát a zjednodušujú interpretáciu výsledkov, ukážku je možné vidieť na obrázku 4.1.



Obr. 4.1: Vizualizácia nazbieraných dát pomocou `SnakeViz`.

4.1.2 `Time` a `timeit`

Moduly `time` a `timeit` sú nástroje Pythonu na meranie času vykonávania užívateľom definovaných úsekov kódu (stopky). `Time` poskytuje jednoduché funkcie na meranie celkových časových intervalov, ako napríklad `time.perf_counter()`, ktoré sú volané jedenkrát pred a jedenkrát po akcii, čím získava rozdiel medzi nimi. Toto je nenáročný, ale nie príliš sofistikovaný spôsob merania času vykonávania kódu. Výsledky musí užívateľ explicitne spracovať a interpretovať, pretože `time` poskytuje iba surové časové údaje.

`Timeit` je špecializovaný na presné meranie trvania malých fragmentov kódu, funkcií, či jednotlivých výrazov, je optimalizovaný na minimalizáciu vplyvu externých faktorov na meranie. `Timeit` sa odporúča na hodnotenie výkonnosti (tzv. *benchmarking*) a testovanie výkonu kódu vzhľadom na jeho schopnosť poskytovať konzistentné a opakovateľné výsledky.

[1] `https://jiffyclub.github.io/snakeviz/`

Jeho výstup je možné ľahko spracovať a interpretovať, keďže poskytuje priemer z viacerých meraní. Naproti tomu, `time` je vhodný na jednoduché, menej formálne merania.

Obidva moduly sú obmedzené na presné úseky kódu, ktoré vývojári musia explicitne definovať ako napríklad vo výpise 4.2, a nezahŕňajú komplexné profilovacie schopnosti ako napríklad `cProfile`, ktorý poskytuje podrobné informácie o vykonávaní celého programu. Tieto nástroje sú ideálne na rýchle overenie a optimalizáciu špecifických funkcií alebo operácií bez potreby podrobnej analýzy výkonnosti celej aplikácie [10, 17].

```
import timeit

...

def main():
    # Meranie casu inicializacie ekosystemu
    init_time = timeit.timeit \
        ('init_ecosystem(100)', globals=globals(), number=1)
    print(f"Initialization time: {init_time} seconds")

    ecosystem = init_ecosystem(100)
    day_times = []
    for day in range(50):
        # Meranie casu pre jednotlive dni
        day_time = timeit.timeit \
            ('simulate_day(ecosystem)', globals=globals(), number=1)
        day_times.append(day_time)
        print(f"... {day_time} ...")

    print(f"Total simulation time over 50 days: {sum(day_times)} seconds")

if __name__ == "__main__":
    total_main_time = timeit.timeit('main()', globals=globals(), number=1)
    print(f"Total time for main function: {total_main_time} seconds")
```

Výpis 4.2: Príklad možnej inštrumentácie pomocou modulu `timeit`.

4.1.3 Sys.monitoring a sys.settrace

Nový modul `sys.monitoring` zavedený v Python 3.12 prináša vylepšené možnosti na monitorovanie udalostí počas vykonávania programu s nižšou réžiou oproti tradičným metódam, ako sú `sys.settrace` a `cProfile`. Tento modul umožňuje vývojárom špecifikovať, ktoré udalosti chcú sledovať, čím poskytuje podrobnejší pohľad na vnútorné operácie aplikácie s minimálnym vplyvom na výkon.

`sys.settrace` je účinný nástroj na trasovanie programových operácií, ako sú volania funkcií a návraty z nich, spoločne s výnimkami, ale je spojený s vyššou réžiou, čo môže negatívne ovplyvniť výkon aplikácie, najmä pri rozsiahlejšom použití.

Na rozdiel od `cProfile`, ktorý poskytuje komplexné profilovanie aplikácií s detailnými metrikami o čase strávenom v jednotlivých funkciách, `sys.monitoring` ponúka efektívnejší spôsob monitorovania špecifických udalostí bez zbytočného zaťaženia systému. Tým sa stáva vhodnou voľbou v situáciách, kedy je potrebné získať detailné informácie o správaní prog-

ramu s minimálnym dopadom na jeho výkon. Na rozdiel od `cProfile` umožňuje filtrovať udalosti už počas vykonávania programu, čím značne znižuje réžiu vykonávania programu. Podrobnejšia analýza je v kapitole 5.

`sys.monitoring` je teda ideálnym nástrojom pre aplikácie vyžadujúce neustále monitorovanie a analýzu počas ich vykonávania, zatiaľ čo `cProfile` je vhodnejší na hĺbkové profilovanie a optimalizáciu kódových úsekov počas vývoja. Na rozdiel od nástroja `sys.settrace` sa udalosti vzťahujú ma interpret, nie na vlákno (túto nevýhodu je ale možné ošetriť). `Sys.monitoring` umožňuje vývojárom nielen špecifikovať, ktoré udalosti chcú sledovať, ale aj definovať typy dát, ktoré chcú získavať, ako sú časy vykonávania funkcií, návratové hodnoty, spracovanie výnimiek a podobne. Modul sám o sebe neponúka interpretáciu získaných dát, čo znamená, že vývojári musia explicitne definovať, ako budú dáta spracované a interpretované. Umožňuje to vysokú mieru prispôsobenia a možnosť zamerania sa na špecifické potreby aplikácie [19].

4.2 Vzorkovacie profilery

Táto sekcia poskytuje prehľad nástrojov na profilovanie, ktoré sú špeciálne navrhnuté na efektívne sledovanie a analýzu výkonu programov bez priameho zásahu do ich kódu. Medzi takéto nástroje patria napríklad `PyInstrument`, `py-spy` a `Scalene`, ktoré sú vhodné predovšetkým pre výkonnostne náročné aplikácie, kde je dôležité zachovať minimálne zaťaženie systému a vplyv na výkon samotného programu, čo umožňuje profilovanie alebo debugovanie programov v produkčnom nasadení. Detaily o tom, ako tieto profilery fungujú, sú popísané v sekcii 2.2.1.

4.2.1 `PyInstrument`

`PyInstrument` funguje podobne ako `cProfile`, ale ponúka dve hlavné výhody. Prvou je, že namiesto sledovania každej jednotlivej inštancie vykonávania funkcie, PyInstrument využíva metódu vzorkovania. Táto metóda získava vzorky zásobníka volania každú milisekundu, pričom frekvenciu získavania vzoriek si môže užívateľ nastaviť podľa vlastných potrieb. Druhou hlavnou výhodou je, že vďaka štatistickej metóde profilovania je jeho výstup zredukovaný a umožňuje vývojárom sústrediť sa na analýzu najkritickejších miest (tzv. *hotspots*).

`PyInstrument` je možné použiť buď na analýzu vybraných častí programu podobne ako `time` alebo `timeit`, alebo na profilovanie celého programu (pri použití z príkazového riadka). Výstup môže byť priamo do príkazového riadka, vo formáte JSON, alebo ako vizuálna reprezentácia v HTML. Aj napriek tomu má určité obmedzenia: Nemusí správne fungovať pri volaní priamo z príkazového riadka, hoci vložené inštrukcie v zdrojovom kóde obvykle fungujú ako očakávané. Navyše, nedokáže profilovať viacvláknové aplikácie [21].

Obr. 4.2: Ukážka výstupu `PyInstrument` spusteného z príkazového riadku.

4.2.2 Py-spy

`Py-spy` podobne ako `PyInstrument` sleduje stav zásobníka vykonávania programu v pravidelných intervaloch. Na rozdiel od `PyInstrumentu`, `py-spy` bol napísaný v jazyku Rust a jeho vykonávanie je mimo procesu s profilovaným programom. Vďaka svojej architektúre dokáže `py-spy` profilovať viacvláknové a viacprocesové Python aplikácie.

`py-spy` ponúka tri hlavné príkazy: `record`, `top`, a `dump`. Príkaz `record` generuje SVG súbory, v ktorých sú nazbierané dáta vizualizované pomocou skriptu `flamegraph.pl` od B. Gregga[2]. Príkaz `top` vyvolá počas vykonávania aktualizované interaktívne zobrazenie vnútra aplikácie, zobrazené rovnakým spôsobom ako Unixový nástroj `top`. Príkaz `dump` slúži na zobrazenie aktuálneho zásobníka vykonávania programu. Najväčším nedostatkom `py-spy` je, že neumožňuje inštrumentáciu užívateľom vybraných častí kódu, ale iba celého programu alebo jeho komponentov [11].

Obr. 4.3: Ukážka výstupu `py-spy` v podobe *flamegraphu*.

4.3 Pyflame

`Pyflame` je založený na Linuxovom nástroji `ptrace`, čo ho obmedzuje na použitie v prostredí Linux. Je inšpirovaný projektom `djdt-flamegraph`[3]. Efektívnosť `Pyflame` vo viacvláknových aplikáciách z neho robí vhodný nástroj na profilovanie komplexnejších systémov, avšak jeho použitie je limitované na prostredia, kde je dostupný nástroj `ptrace`. Podobne ako `py-spy`, jeho vykonávanie je mimo procesu s profilovaným programom. `Pyflame` generuje

[2] `https://github.com/brendangregg/FlameGraph`
[3] `https://github.com/blopker/djdt-flamegraph`

dáta na štandardný výstup vo formáte, ktorý očakáva skript `flamegraph.pl` od B. Gregga[2] podobne ako `py-spy` [7].

4.4 Yappi

`Yappi` (*Yet Another Python Profiler*) zdieľa niektoré vlastnosti s `cProfile`, ako je efektívnosť a implementácia v jazyku C, avšak je špeciálne navrhnutý na riešenie niektorých obmedzení `cProfile`, najmä v oblasti viacvláknových aplikácií. `Yappi` poskytuje podrobné informácie o výkone jednotlivých vlákien a umožňuje profilovanie asynchrónneho kódu, čo ho robí ideálnym nástrojom pre moderné asynchrónne aplikácie. Ako jeden z mála profilerov umožňuje profilovanie tzv. *greenlets* a *coroutines*. `Yappi` podporuje rôzne spôsoby výstupu, vrátane možnosti exportu profilovacích dát do formátov ako HTML, CSV, alebo priamo do SQL databáz, čo zjednodušuje ďalšie spracovanie a analýzu [6].

Podobne, ako `cProfile`, má výstup v podobe zoznamu, ktorý je náročný na analýzu výkonu aplikácie, ale existujú nástroje na vizualizáciu jeho výstupu, ako je napríklad `gprof2dot`[4].

4.5 Scalene

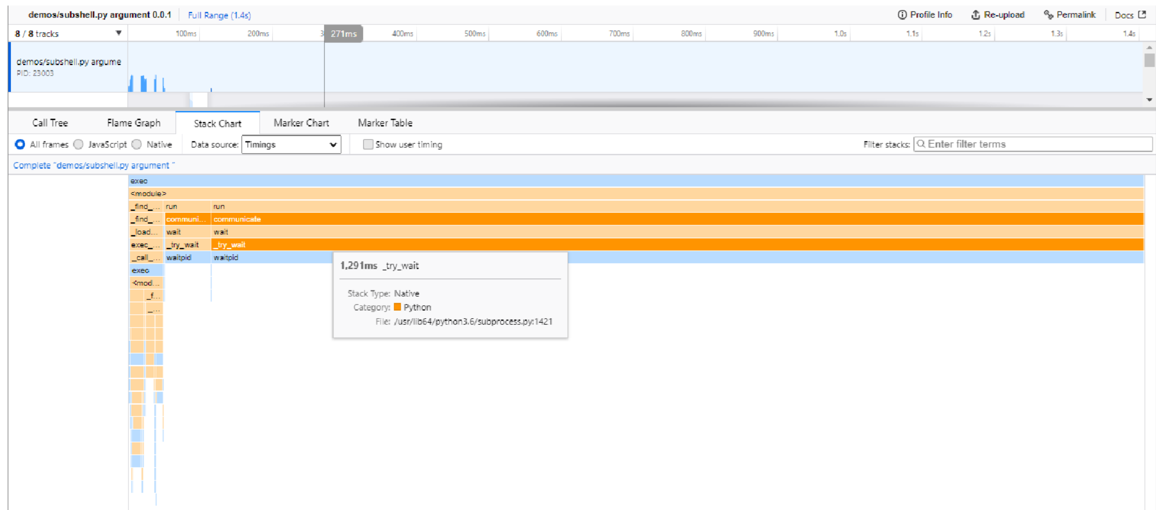
`Scalene` je profilovací nástroj, ktorý používa kombináciu inštrumentácie a vzorkovania na analýzu výkonu a používania pamäte Python aplikácií, a zahŕňa aj schopnosť profilovania GPU (obmedzený na grafické karty NVIDIA). `Scalene` sa líši od tradičných nástrojov ako `cProfile` tým, že poskytuje rozšírené metriky nielen pre CPU, operačnú pamäť, ale aj pre GPU, čo umožňuje podrobnejšiu analýzu v aplikáciách, ktoré využívajú grafické procesory. Naviac, integruje umelú inteligenciu, ktorá navrhuje optimalizácie v zdrojovom kóde. V porovnaní s ostatnými profilermi má `Scalene` nižšiu réžiu. Vďaka týmto vlastnostiam je ideálny na použitie vo vývojovom prostredí, poskytuje vizuálne prístupné a ľahko interpretovateľné výstupy v podobe grafického užívateľského rozhrania v HTML na efektívnejšiu analýzu výkonu [3] ako je možné vidieť na obrázku 4.5.

4.6 FunctionTrace

`FunctionTrace` je nástroj určený na detailné sledovanie a analýzu výkonu Python aplikácií prostredníctvom trasovania funkcií. Vývojárom umožňuje sledovať tok programu a interakcie medzi volaniami funkcií v reálnom čase, bez nutnosti modifikácie kódu. Poskytuje úplný a presný prehľad o celkovom vykonávaní aplikácie, vrátane záznamov vykonávania funkcií, kedy a ktorá informácia bola zaznamenaná, miesta alokácií a ďalších dôležitých udalostí. `FunctionTrace` podporuje Python 3.5+, je kompatibilný s Linuxom a macOS a umožňuje jednoduché online zdieľanie profilov. Používa `sys.setprofile` háčiky na zaznamenávanie vykonaných funkcií. Výstupy `FunctionTrace` je možné jednoducho vizualizovať v podobe *Stack Charts*, *Flame Graphs* a *Call Trees* vďaka integrácii s `Firefox Profilerom`[5] ako je možné vidieť na obrázku 4.4. Dáta sú ukladané vo formáte JSON, podobne ako pri nástroji `perf`[6] z operačného systému Linux. Tento nástroj je efektívny aj pri profilovaní viacvláknových a viacprocesových aplikácií, pričom jeho řéžia je obvykle nižšia ako 5%. Umožňuje

[4]https://github.com/jrfonseca/gprof2dot
[5]https://profiler.firefox.com/docs/
[6]https://perf.wiki.kernel.org/index.php/Main_Page

tiež sledovanie alokácií pamäte v celej aplikácii, čím pomáha identifikovať oblasti vhodné na optimalizáciu. Profily možno ľahko nahrávať a zdieľať, čo umožňuje efektívnejšiu spoluprácu pri riešení výkonnostných problémov alebo chýb [5].



Obr. 4.4: Príklad vizualizácie profilu vygenerovaného nástrojom `FunctionTrace` vizualizovaného v podobe *Stack Chart*.

4.6.1 Porovnanie s podobnými nástrojmi

Hoci `FunctionTrace` poskytuje robustné funkcionality na sledovanie Python aplikácií, nástroje ako `VizTracer` a `Palanteer` prinášajú niektoré unikátne vlastnosti. `VizTracer` sa zameriava na vizualizáciu dát pomocou interaktívnych grafických reportov, ktoré umožňujú zobrazovať aj zdrojový kód profilovanej funkcie v HTML dokumente [12]. Zároveň ponúka aj množstvo filtrov, ktoré môžu pomôcť zefektívniť samotný proces profilovania. Na druhej strane, `Palanteer` kombinuje profilovanie s monitorovaním správania programu, poskytujúc komplexnejší pohľad na výkon a správanie aplikácie. `Palanteer` umožňuje profilovanie Python a C++ programov, čo môže byť užitočné v prípade Python programu, ktorý obsahuje C++ knižnice. Ďalším rozdielom je, že obsahuje separátne grafické užívateľské rozhranie na interpretáciu nazbieraných dát. Syntax je podobná modulu `cProfile` [8].

4.7 Ostatné profilery

Táto práca je zameraná na nástroje, ktoré by mohli byť integrované do nástroja `Perun` 3 s cieľom rozšíriť jeho schopnosti o profilovanie Python programov. Okrem spomínaných nástrojov existujú komerčné nástroje ako napríklad `GProfile` a `VTune`, avšak tieto nástroje nie sú podrobne analyzované, pretože sú mimo záujmu tejto práce o rozšírenie nástroja `Perun`. Python 3.12 okrem `sys.monitoring` ponúka aj podporu pre `Linux perf profiler`[7], ktorý umožňuje detailnejšie profilovanie až na úroveň natívnych funkcií a procedúr napísaných v jazyku C. Využitie nástrojov špecifických pre Linux by navyše obmedzilo použitie tohto nástroja na zariadenia s iným operačným systémom. Okrem spomínaných nástrojov existuje mnoho ďalších, ktoré však v rámci tejto práce neboli podrobne analyzované.

[7]`https://docs.python.org/3.12/howto/perf_profiling.html`

/mnt/c/Users/Matej/Desktop/Test/test5.py: % of time = 100.00% (116.821ms) out of 116.821ms.

```
      Time    ------  ------  Memory  ------  -----------  Copy
Line  Python  native  system  Python  peak    timeline/%   (MB/s)  /mnt/c/Users/Matej/Desktop/Test/test5.py

   1                                                               import random
   2                                                               import time
   3
   4                                                               class Organism:
   5                                                                   def __init__(self, type, energy):
   6                                                                       self.type = type
   7                                                                       self.energy = energy
   8                                                                       self.alive = True
   9
  10   11%                                                                 def act(self, ecosystem):
  11                                                                       if self.type == "plant":
  12                                                                           self.photosynthesize()
  13                                                                       elif self.type == "herbivore":
  14                                                                           self.eat_plant(ecosystem)
  15                                                                       elif self.type == "carnivore":
  16                                                                           self.eat_herbivore(ecosystem)
  17
  18                                                                       if self.energy <= 0:
  19                                                                           self.alive = False
  20
  21                                                                   def photosynthesize(self):
  22                                                                       self.energy += 1
  23
  24                                                                   def eat_plant(self, ecosystem):
  25                                                                       # Eat plant logic here
  26                                                                       self.energy += 1
  27
  28   10%      5%                                                         def eat_herbivore(self, ecosystem):
  29                                                                       # Eat herbivore logic here
  30                                                                       self.energy += 1
  31
  32   31%     12%      1%                                     61          def reproduce(self, ecosystem):
  33                                                                       if self.energy > 10:
  34                                                                           ecosystem.append(Organism(self.type, self.en
  35                                                                           self.energy -= self.energy // 2
  36
  37                                                               def simulate_day(ecosystem):
  38                                                                   for organism in ecosystem:
  39                                                                       organism.act(ecosystem)
  40    4%      3%     12%                                    168              organism.reproduce(ecosystem)
  41   11%                                                                 return [organism for organism in ecosystem if organi
  42
  43                                                               def init_ecosystem(num_organisms):
  44                                                                   types = ["plant", "herbivore", "carnivore"]
  45                                                                   return [Organism(random.choice(types), random.randin
  46
  47                                                               def main():
  48                                                                   ecosystem = init_ecosystem(100)
  49                                                                   for day in range(50):
  50                                                                       ecosystem = simulate_day(ecosystem)
  51                                                                       print(f"Day {day+1}: {len(ecosystem)} organisms"
  52
  53                                                               if __name__ == "__main__":
  54                                                                   start_time = time.perf_counter()
  55                                                                   main()
  56                                                                   print(f'Code execution: {round(time.perf_counter() -

                                                                   function summary for /mnt/c/Users/Matej/Desktop/Test/te…
  10   11%                                                         Organism.act
  28   10%      5%                                                 Organism.eat_herbivore
  32   31%     12%      1%                                     61  Organism.reproduce
  37   14%      5%      9%                                    168  simulate_day
```

generated by the scalene profiler

Obr. 4.5: Ukážka výstupu Scalene vo formáte HTML.

Kapitola 5

Návrh a implementácia

Táto kapitola sa zameriava na návrh a popis implementácie nového kolektoru pre profilovanie Python programov, ktorý je integrovaný do nástroja Perun (sekcia 3), ako aj na prepísanie už existujúceho modulu na vizualizáciu profilov, `flamegraph`. Kolektor využíva nové aplikačné rozhranie `sys.monitoring`, ktoré bolo predstavené vo verzii Python 3.12, na sledovanie behu programu. V tejto kapitole sú popísané jednotlivé požiadavky, návrh a spôsob implementácie týchto modulov.

5.1 Analýza požiadaviek

Predtým, než začne samotný proces návrhu a implementácie nového nástroja na profilovanie, je nevyhnutné dôkladne analyzovať a definovať požiadavky, ktoré tento nástroj musí splniť. Táto analýza poskytuje predstavu o cieľoch a očakávaniach, ktoré majú byť dosiahnuté. V tejto časti sú rozanalyzované požiadavky, ktoré sú nevyhnutné na efektívne profilovanie Python aplikácií.

- **Presnosť merania**: Kolektor musí byť schopný poskytnúť presné merania času stráveného v jednotlivých funkciách profilovaného programu.
- **Minimálna réžia**: Profilovanie by nemalo výrazne spomaliť vykonávanie aplikácie. Kolektor by mal byť navrhnutý tak, aby spôsoboval čo najmenšiu réžiu.
- **Granularita**: Profiler by mal byť schopný poskytnúť detailné dáta na úrovni jednotlivých funkcií, vrátane volaní vnútorných funkcií a možnosti sledovania rekurzívnych volaní.
- **Viacvláknové aplikácie**: V prípade aplikácií, ktoré využívajú viacvláknové procesy, je nevyhnutné, aby profiler mohol správne sledovať a zaznamenávať dáta pre jednotlivé vlákna.
- **Graf volania funkcií**: Profiler by mal byť schopný zaznamenávať poradie volania funkcií na následnú interpretáciu.
- **Filtrovanie dát**: Užívatelia by mali mať možnosť nastaviť filtre na profilovanie dát, vylúčiť napríklad špecifické moduly, knižnice alebo funkcie.

5.2 Návrh

Prvým krokom pri implementácii profileru je zaistenie zberu profilovacích dát. Na základe analýzy v kapitole 4 bol za týmto účelom vybraný nástroj `sys.monitoring`. Toto *API*, ktoré je súčasťou štandardnej knižnice Pythonu, ponúka širokú škálu možností na výber zbieraných dát, vrátane ich filtrovania a nastavenia granularity. Tento výber umožňuje efektívne sledovanie výkonnostných metrík bez potreby externých závislostí, čo znižuje riziko komplikácií pri nasadení profileru v rôznych prostrediach. `sys.monitoring` je navrhnuté tak, aby minimalizovalo vplyv na výkon aplikácie, čo je kritické na profilovanie aplikácií v produkčnom nasadení. Ide o novinku, ktorá bude pravdepodobne v budúcnosti rozširovaná[1], čím sa otvárajú ďalšie možnosti na jeho využitie.

V tejto časti sa venujeme návrhu a architektúre kolektora, ktorý je integrovanou súčasťou profilovacieho nástroja založeného na *API* `sys.monitoring`. Kolektor je rozdelený na dva hlavné moduly: *collect* a *parser*. Toto rozdelenie, kde modul *collect* zaznamenáva udalosti ako napríklad vstupy do funkcií a ich návraty, a modul *parser* zodpovedá za vytváranie výsledného profilu, minimalizuje réžiu v porovnaní s priamym spracovaním udalostí.

5.2.1 Získavanie dát

Počas vykonávania programu dochádza k vyvolávaniu udalostí, pričom `sys.monitoring` umožňuje zachytenie týchto udalostí prostredníctvom *tracingu*. Táto metóda poskytuje vysokú presnosť a má nižšiu réžiu v porovnaní s inštrumentáciou programu, pričom nevyžaduje vkladanie dodatočného kódu do sledovaného programu. `Sys.monitoring` navyše umožňuje rôzne stupne granularity, od sledovania jednotlivých funkcií až po sledovanie základných blokov programu, čo závisí od typu sledovaných udalostí. Okrem toho je možné filtrovať zachycované udalosti a tým znížiť réžiu preskočením ich ďalšieho spracovania.

V prvom prototype kolektoru boli sledované vstupy do funkcií a ich návraty. Avšak, výsledný profiler by mal byť schopný ponúkať rozšírené možnosti, vrátane zachytávania výnimiek a správneho sledovania uvoľňovania zásobníka volaní spolu s výnimkami. Okrem toho by mal byť schopný efektívne monitorovať aj generátory a ich špecifické správanie.

5.2.2 Ukladanie dát

Jedným z kľúčových faktorov je spôsob ukladania dát. V pôvodnom návrhu sme predpokladali ukladanie dát v operačnej pamäti, čo však môže viesť k problémom v prostredí s obmedzenou operačnou pamäťou alebo pri profilovaní rozsiahlych aplikácií. Z týchto dôvodov sme od tohto prístupu upustili a zvolili sme priebežné ukladanie nazbieraných dát do externého súboru.

Tento prístup má však niekoľko nevýhod. Kvôli periodickému ukladaniu meraných dát môže dôjsť k drobným oneskoreniam, pretože vykonávanie programu je počas ukladania dát krátkodobo pozastavené. Toto môže nepriaznivo ovplyvniť presnosť meraného času vykonávania funkcií. Ukladané dáta obsahujú typ vyvolanej udalosti, unikátny kľúč zložený z cesty k funkcii, jej názvu, vlákna, v ktorom bola udalosť vyvolaná, a čísla riadku, na ktorom sa funkcia nachádza. Ďalej zahŕňajú časovú pečiatku, kedy bola udalosť vyvolaná a v prípade výnimiek, aj informácie o nich. Záznamy sú ukladané v chronologickom poradí, v akom boli udalosti vyvolané, čo následne umožňuje zostaviť graf vykonávania funkcií.

[1] V priebehu tvorby tejto práce boli pridané nové zaujímave udalosti, ktoré pridávajú možnosť monitorovania základných blokov (tzv. *basic–blocks*).

5.2.3 Spracovanie zachytených udalostí

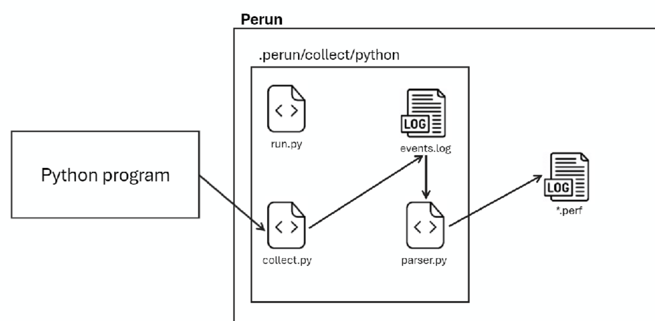
Správne spracovanie zachytených udalostí je kľúčové na vytvorenie presných profilov aplikácie. Keďže samotné udalosti neposkytujú priamo analyzovateľné dáta o výkone, je nevyhnutné ich správne spojiť, aby presne reprezentovali vykonávanie jednotlivých funkcií. Každá funkcia je v profiloch reprezentovaná dvojicou udalostí: štart a koniec. Koniec funkcie môže nastať návratom do rodičovskej funkcie, vyhodením výnimky, alebo iným ukončením vykonávania funkcie.

Dôležité je tiež vypočítať exkluzívny čas strávený vo funkcii a ďalšie metriky podľa potreby. Parser musí efektívne rekonštruovať trasu vykonávania funkcií, pričom zohľadňuje možnosť, že program môže byť vykonávaný vo viacerých vláknach. Na presné stvárnenie profilu viacvláknového vykonávania je potrebné udržiavať zásobníky pre každé vlákno a záznamy o vzťahoch medzi vláknami, aby bolo možné správne zreprodukovať kompletné trasy vykonávania jednotlivých funkcií.

Parser po úspešnom spracovaní agreguje dáta a exportuje ich do súboru vo formáte JSON, čo uľahčuje ich ďalšie spracovanie a analýzu.

5.2.4 Navrhované riešenie

Výsledným riešením je teda integrovaný kolektor, v tomto prípade v nástroji Perun, ktorý pomocou techniky sledovania udalostí s využitím nového aplikačného rozhrania predstaveného v Python 3.12 dokáže profilovať výkonnostné metriky ako je napríklad doba vykonávania funkcií. Navrhovaný profiler prináša novú možnosť profilovania Python programov s využitím existujúcej funkcionality na tvorbu profilov v nástroji Perun. Je dôležité poznamenať, že nástroj Perun podporuje verzie Pythonu 3.9 – 3.11 a preto sa musia v prostredí, kde tento profiler funguje, nachádzať Python 3.12 a aj niektorá z podporovaných verzií Perunom. Pracovný postup kolektoru je vysvetlený na obrázku 7.4.



Obr. 5.1: Ilustrácia pracovného postupu kolektoru. Hlavný modul, `run.py`, inicializuje a koordinuje moduly `collect.py` a `parser.py`. Vo vnútri `collect.py` sa spúšťa cieľový Python program, zachytávajú udalosti, ktoré sú ukladané do dočasného súboru `events.log`. Po skončení vykonávania profilovaného Python programu, `run.py` spúšťa `parser.py`, ktorý spracováva nazbierané dáta v `events.log` a konvertuje ich do unifikovaného JSON formátu. Nakoniec `run.py` odstráni dočasný súbor `events.log` a vytvorí finálny súbor `*.perf` obsahujúci vytvorený profil.

5.3 Implementácia

Táto sekcia sa podrobne venuje implementácii kolektoru a jeho častí, ktorý je integrovaný do nástroja Perun, a poskytuje komplexný pohľad na jeho funkcionalitu a architektúru. Kolektor, implementovaný v jazyku Python, je navrhnutý tak, aby efektívne a presne zachytával dáta vykonávana aplikácií bez potreby externých knižníc. Kľúčovým cieľom bolo dosiahnuť vysokú presnosť dát s minimálnym vplyvom na výkon profilovaných aplikácií. V nasledujúcich odsekoch je popísaná implementácia kritických aspektov kolektora, vrátane jeho integrácie do ekosystému Perun, metód zachytávania a spracovania udalostí, ako aj opatrení, ktoré zabezpečujú jeho efektivitu. Kolektor je implementovaný v podobe troch modulov na riadanie zberu a spracovania dát.

5.3.1 Modul `run.py`

Modul `run.py` predstavuje hlavný modul každého kolektoru v nástroji Perun. Tento skript je štruktúrovaný do troch hlavných fáz:

1. **Before.** V tejto úvodnej fáze kolektor overí, či je v prostredí dostupný Python 3.12, ktorý je nevyhnutný na spustenie modulu `collect.py`. Modul `collect.py` je zodpovedný za zbieranie dát z behu programu.

2. **Collect.** Táto fáza zahŕňa samotné zbieranie dát. Keďže Perun funguje na Python 3.9 až 3.11 a `collect.py` (predstavený v sekcii 5.3.2) vyžaduje Python 3.12+, je potrebné tento skript spustiť v samostatnej inštancii. Na tento účel sa využíva `subprocess`, ktorý umožňuje izolované spustenie skriptu a zabezpečuje správne prostredie na jeho vykonávanie. V tejto fáze `collect.py` zachytí všetky relevantné udalosti a uloží ich do dočasného súboru `events.log`.

3. **After.** Posledná fáza má za úlohu spracovať surové dáta do finálneho formátu, konkrétne do profilu vo formáte JSON, príklad výstupu jednej funkcie je ukázaný vo výpise 5.3. Na spracovanie dát slúži modul `parser.py` (predstavený v sekcii 5.3.3), ktorý transformuje dáta zozbierané v predchádzajúcej fáze.

Modul `run.py` je integrovaný do prostredia príkazového riadku (CLI) pomocou knižnice `click`, čo umožňuje jeho konfiguráciu a spustenie s rôznymi parametrami. Používateľ má možnosť špecifikovať filtre, ktoré určujú, aké súbory alebo moduly by mali byť pri profilovaní ignorované.

5.3.2 Modul `collect.py`

Tento skript predstavuje nástroj na profilovanie Python aplikácií s využitím rozhrania `sys.monitoring` na sledovanie a zachytávanie dôležitých udalostí vyvolaných počas vykonávania programu. Skript je navrhnutý tak, aby bol flexibilný a aby minimálne zasahoval do výkonu profilovaného programu. Táto sekcia popisuje logiku a technológie použité v tomto skripte.

Kľúčové komponenty a princípy

- **Udalosti a ich spracovanie:** Skript využíva `sys.monitoring` na registrovanie callback funkcií pre rôzne udalosti (`PY_START`, `PY_RETURN`, `PY_THROW`, atď.), ktoré reprezentujú štarty a ukončenia funkcií, ako aj iné významné body v behu programu. Každá

zachytená udalosť je spracovaná funkciou `capture_event`, ktorá generuje podrobný záznam o udalosti vrátane časovej pečiatky a identifikátora vlákna, čo umožňuje detailnú analýzu paralelného vykonávania.

- **Efektívne ukladanie dát:** Namiesto okamžitého ukladania každej udalosti do súboru sa udalosti hromadia v pamäti až do dosiahnutia určitej veľkosti (128 KB) a až potom sú dávkovo uložené do súboru `events.log`. Tento prístup zníži počet diskových operácií a zvýši efektivitu celého procesu profilovania.
- **Kontextový manažér[2]:** Kontextový manažér `monitor` je kľúčovou súčasťou skriptu, ktorá zabezpečuje správne zahájenie a ukončenie sledovania udalostí. Kontextový manažér vykonáva nasledujúce úlohy:
  - **Registrácia callback funkcií.** Dynamicky registruje funkcie na spracovanie rôznych typov udalostí, ako sú štart a ukončenie funkcií, vyhodenie výnimiek, a ďalšie udalosti vyvolané počas vykonávania programu.
  - **Správa udalostí.** Po aktivácii všetkých požadovaných udalostí pomocou funkcie `sys.monitoring.set_events`, kontextový manažér `monitor` umožňuje vykonanie sledovaného bloku kódu. Počas vykonávania sú udalosti zachytávané a ukladané do pamäti a následne do dočasného súboru v určitých intervaloch.
  - **Uloženie a uvoľnenie zdrojov.** Pri výstupe z kontextového bloku, zabezpečuje `monitor` uloženie všetkých neuložených udalostí a následne uvoľní pridelené identifikátory nástroja, čím sa zabráni úniku zdrojov alebo zanechaniu sledovania aktívneho v systéme.

```
@contextmanager
def monitor(tool_id: int):
    # Zoznam typov sledovanych udalosti
    event_names = ['PY_START', ... , 'PY_UNWIND']
    events = (sys.monitoring.events.PY_START | ...)

    sys.monitoring.use_tool_id(tool_id, "profile")
    sys.monitoring.set_events(tool_id, events)

    # Registracia callbackov pre kazdu udalost
    for event in event_names:
        register_event_callback(tool_id, event, capture_event)

    try:
        yield
    finally:
        # Ulozenie udalosti a~uvolnenie zdrojov
        if events_to_save:
            save_events_to_file()
        sys.monitoring.free_tool_id(tool_id)
```

Výpis 5.1: Implementácia kontextového manažéra `monitor` pre sledovanie udalostí s využitím API sys.monitoring.

[2] https://book.pythontips.com/en/latest/context_managers.html

V `events.log` sú zaznamenané udalosti zachytené počas vykonávania profilovaného programu. Každý riadok súboru predstavuje jednu udalosť, ktorá obsahuje nasledujúce položky:

1. **Typ udalosti** – označuje, či ide o začiatok alebo koniec funkcie, ...
2. **Identifikátor udalosti** – obsahuje cestu k súboru funkcie, názov funkcie, číslo riadku a identifikátor vlákna, kde udalosť nastala.
3. **Časová pečiatka** – presný čas, kedy udalosť nastala.
4. **Návratová hodnota alebo výnimka** – hodnota vrátená funkciou alebo informácia o vyvolanej výnimke.

```
...
PY_START,test.py:subtract:9:134253467029504,398285.514419068,None
PY_RETURN,test.py:subtract:9:134253467029504,398285.514419068,121
PY_RETURN,test.py:complex_operation:12:134253467029504,398285.514419068,121
PY_RETURN,test.py:run_calculations:32:134253467029504,398285.514419068,None
PY_RETURN,test.py:<module>:1:134253467029504,398285.514419068,None
```

Výpis 5.2: Ukážka záznamov z `events.log`

5.3.3 `parser.py`

Modul `parser.py` je zodpovedný za spracovanie udalostí zachytených skriptom `collect.py` pri vykonávaní Python programov. Trieda `ParseEvents` v tomto module analyzuje a transformuje surové dáta do štruktúrovaného formátu, čo umožňuje ďalšiu analýzu a vizualizáciu výkonu aplikácií. V tejto subsekcii je podrobne opísaný proces spracovania dát, vrátane metód na identifikáciu a agregáciu udalostí, čo zahŕňa sledovanie metrík funkcií, správu zásobníkov volaní a vlákien. Dôraz je kladený na efektívnosť a presnosť pri transformácii surových udalostných dát do analyzovateľnej formy.

Spracovanie a analýza dát

Hlavnou funkciou modulu `parser.py` je metóda `process_data`, ktorá číta udalosti zo súboru `events.log`, spracováva ich a dynamicky menežuje zásobníky volaní pre rôzne vlákna. Táto metóda koordinuje spracovanie udalostí v rámci jednotlivých vlákien a zabezpečuje presné zachytenie a rekonštrukciu sledu vykonávania funkcií.

Spracovanie udalostí zahŕňa párovanie vstupov a výstupov z funkcií, pomocou príslušných metód na spracovanie udalostí: `event_start_or_resume`, `event_return_or_yield`, a `event_exception`. Po spracovaní všetkých udalostí, metóda `get_resources` agreguje a transformuje nazbierané dáta z `function_metrics_out` do štruktúrovaného formátu JSON, pripraveného na ďalšiu analýzu a vizualizáciu.Na výpise 5.3 je ukážka záznamu o vykonaní funkcie. Výsledkom je slovník, ktorý podrobne opisuje výkon a vykonávanie jednotlivých funkcií.

Dáta sú ukladané do slovníka (tzv. *dictionary*), kde každá položka reprezentuje jednu funckiu s nasledujúcou štruktúrou:

- **uid**: Jednoznačný identifikátor funkcie, obsahujúci zdrojový súbor s absolútnou cestou, názov funkcie a číslo riadku.

- **amount**: Exkluzívny čas strávený vo funkcii[3].
- **tid**: Identifikátor vlákna, v ktorom bola funkcia vykonávaná.
- **type**: Typ meranej jednotky, v tomto prípade 'time' pre časové merania.
- **ncalls**: Počet volaní danej funkcie.
- **trace**: Trasa volania funkcií, zaznamenaná ako zoznam volaných funkcií.
- **exceptions**: Zoznam výnimiek vyvolaných počas vykonávania funkcie.

Tento popis dátovej štruktúry umožňuje lepšie pochopenie informácií, ktoré sú k dispozícii na analýzu výkonu aplikácie.

```
{
  "amount": 4.608009476214647e-06,
  "uid": {
    "source": "~/main.py",
    "function": "fibonacci",
    "line": 2
  },
  "tid": "139821382942720",
  "type": "time",
  "ncalls": 1,
  "trace": [
    {
      "source": "~/main.py",
      "function": "<module>",
      "line": 1
    }
  ],
  "exceptions": [
    "ZeroDivisionError('division by zero')"
  ]
}
```

Výpis 5.3: Ukážka záznamu o behu funkcie `fibonacci` uloženej do výsledného profilu.

Metódy na spracovanie udalostí

V module `parser.py` sú implementované špecifické metódy na spracovanie rôznych typov udalostí zachytených počas vykonávania Python programu. Tieto metódy sú volané na základe typu udalosti z metódy `process_data`, ktorá je popísaná vo výpise 5.4.

[3]Čas strávený exkluzívne vo funkcii bez času vnorených funkcií

```
def process_data(self):
    # Nacitanie udalosti z~disku
    with open(filepath, 'r') as file:
        events = file.readlines()

    # Parsovanie eventov a~ulozenie na zasobnik
    for event in events:
        parts = event.strip().split(',')
        event_tuple = (parts[0], parts[1], float(parts[2]), parts[3])
        function_metrics.append(event_tuple)

    # Spracovanie nacitanych udalosti
    for event in function_metrics:
        event_type, ..., exception_var = (event[0], ..., event[3])

        # Ulozenie na zasobnik volani a~zasobnik vlakien
        ...

        # Spracovanie udalosti podla ich typu
        if event_type in ['PY_START', 'PY_RESUME']:
            self.event_start_or_resume(event_key, time, call_stack)
        elif event_type in ['PY_RETURN', 'PY_YIELD']:
            self.event_return_or_yield(time, call_stack)
        elif event_type in ['PY_THROW', 'PY_UNWIND']:
            self.event_exception(time, call_stack, exception_var)
```

Výpis 5.4: Ukážka metódy `process_data`, ktorá slúži na načítanie a spracovanie udalosti z `events.log` súboru.

Spracovanie vstupu do funkcie. Keď je zachytená udalosť typu `PY_START` alebo `PY_RESUME`, je zavolaná metóda `event_start_or_resume`, ktorej implementácia je popísaná vo výpise 5.5. Táto metóda spracováva vstupy do funkcií, vrátane tých, ktoré sú generátormi, a ukladá základné informácie o ich volaní:

- **start_time**: Ukladá čas, kedy bola funkcia aktivovaná.
- **trace**: Zaznamenáva zásobník volaní funkcií pre trasovanie volania.
- **nested_calls_time**: Agreguje čas strávený vnorenými volaniami, čo pomáha pri výpočte exkluzívneho času.
- **exclusive_time**: Vypočíta exkluzívny čas strávený funkciou bez vnorených volaní.
- **exceptions**: Zaznamenáva výnimky, ktoré boli vyvolané počas behu funkcie.

Táto metóda zároveň zaisťuje pridávanie záznamov funkcií do zásobníka volaní, ktorý je nevyhnutný na rekonštrukciu trasy volania funkcie. Súčasne aktualizuje počet volaní jednotlivých funkcií uložených v zásobníku počtu volaní funkcií `function_calls_count`, čo umožňuje detailnejšiu analýzu ich využitia v rámci sledovaného programu.

```
def event_start_or_resume(self, event_key, current_time, call_stack):
    function_name = event_key.split(":")[1]
    # Pocitadlo volania funkcii
    if function_name in self.function_calls_count:
        self.function_calls_count[function_name] += 1
    else:
        self.function_calls_count[function_name] = 1

    call_info = {
        'start_time': current_time,
        'trace': list(call_stack),
        'nested_calls_time': 0,
        'exclusive_time': 0,
        'exceptions': [],
    }

    # Ulozenie na zasobnik vykonavania funkcii
    call_stack.append((event_key, call_info))
```

Výpis 5.5: Implementácia metódy `event_start_or_resume` na spracovanie začiatku alebo pokračovania udalosti.

Spracovanie výstupu z funkcie. Pri zachytení udalostí typu `PY_RETURN` a `PY_YIELD`, ktoré signalizujú ukončenie funkcie alebo jej dočasné pozastavenie (*yield*), je volaná príslušná metóda na ich spracovanie. Rovnako sa postupuje pri udalostiach `PY_THROW`, ktoré indikujú vyvolanie výnimky, a `PY_UNWIND`, oznamujúcich výstup z funkcie počas uvoľňovania zásobníka pri výnimke. Tieto metódy extrahujú informácie o volaní funkcie z vrcholu zásobníka volaní a následne aktualizujú metriky volania pomocou metódy `update_call_info`.

Táto metóda vypočíta celkové trvanie volania na základe časovej pečiatky vytvorenej v čase výstupu z funkcie a času začiatku volania získaného z časovej pečiatky pri zachytení udalosti vstupu do funkcie. Z týchto údajov je určený exkluzívny čas funkcie, ktorý predstavuje dobu trvania funkcie bez započítania času stráveného vo vnorených volaniach.

Ak bola zachytená výnimka, pridáva sa informácia o nej do zoznamu výnimiek príslušného volania funkcie. Tento mechanizmus zabezpečuje, že každé volanie funkcie má komplexný záznam o svojom vykonávaní vrátane chybových stavov. Pri `PY_UNWIND` sa špecificky zaznamenáva proces uvoľnenia zásobníka, čo je kritické pri analýze chýb súvisiacich s výnimkami.

5.4 Známe obmedzenia

Tento kolektor, ako aj mnohé iné nástroje, má svoje obmedzenia, ktoré môžu ovplyvniť presnosť alebo granularitu nazbieraných dát. Nasledujúce body popisujú niektoré z hlavných obmedzení identifikovaných počas vývoja a testovania tohto nástroja.

- **Časové pečiatky.** Pri spracovaní udalostí, kde funkcie sú vykonané veľmi rýchlo za sebou, môže dôjsť k tomu, že časové pečiatky generované funkciou `time.perf_counter`

budú identické. V dôsledku toho výpočet trvania volania môže byť nulový. Ako náhradné riešenie sa používa minimálna hodnota trvania nastavená na 1×10^{-12} sekundy, aby sa predišlo nulovej hodnote trvania.

- **Nové udalosti v `sys.monitoring`.** Počas vývoja tejto práce boli do rozhrania `sys.monitoring` pridané nové udalosti, ktoré umožňujú zachytávanie na úrovni základných blokov kódu. Tieto udalosti by umožnili lepšie určenie granularity a presnejšiu analýzu behu programu, čo by vylepšilo možnosti profilovania oproti súčasnému zameraniu len na funkcie.

- **Optimalizácia správy zásobníkov.** Súčasná implementácia môže byť optimalizovaná z hľadiska správy zásobníkov, ktoré uchovávajú dáta. Efektívnejšia správa zásobníkov by mohla zlepšiť celkovú výkonnosť a redukovať pamäťové nároky nástroja.

- **Prístup k zdrojovému kódu:** Vzhľadom na to, že `sys.monitoring` poskytuje monitoring iba v rámci procesu, v ktorom beží, je nevyhnutné mať prístup k zdrojovému kódu profilovaných aplikácií. Toto obmedzenie znamená, že monitorovanie je možné len pre inštancie interpreta Pythonu, ktoré majú možnosť vykonávať kód monitorovania. Ak nie je možné zasahovať do kódu aplikácie, nie je možné účinne využívať všetky funkcie `sys.monitoring` na detailnú analýzu výkonu.

- **Optimalizácia spracovania udalostí.** Súčasná implementácia spracovania udalostí v module `parser.py` je neoptimalizovaná a jej vykonávanie môže trvať neúmerne dlhý čas, najmä pri analýze veľkých programov. To spôsobuje, že nástroj je takmer nepoužiteľný pre rozsiahle aplikácie. Je nevyhnutné zamerať sa na vývoj efektívnejších metód spracovania, aby sa zvýšila rýchlosť a zlepšila celková použiteľnosť nástroja.

Kapitola 6

Vizualizácia

Táto kapitola sa zaoberá implementáciou modulu na vizualizáciu profilov vo forme `FlameGraph`-u. `FlameGraph` je efektívny spôsob na zobrazovanie počtu volaní funkcií, ich trasy, a času stráveného v jednotlivých funkciách. Implementácia tohto modulu je inšpirovaná skriptom `FlameGraph.pl`[1] od Brendana Gregga, ktorý bol napísaný v jazyku Perl.

Implementácia `FlameGraph` v tejto práci je realizovaná v jazyku Python, čo zabezpečuje lepšiu kompatibilitu a integráciu s nástrojom Perun. Prechod na Python znamená, zbavenie sa závislostí externých nástrojov a umožňuje jednoduchšie modifikácie a prispôsobenie. Táto prispôsobená verzia nie je len prostým prekladom z Perlu do Pythonu, ale zahŕňa dôležité úpravy a rozšírenia, ktoré reflektujú špecifické potreby a požiadavky Python profilera vyvinutého v rámci tejto bakalárskej práce. Integrácia do nástroja Perun navyše uľahčuje dalšie rozšírenia a optimalizáciu, čím sa zlepšuje celková flexibilita a efektívnosť nástroja. Práca obsahuje nielen adaptáciu základnej funkcionality `FlameGraph` skriptu, ale aj jeho rozšírenia o nové funkcie na hlbšiu analýzu a lepšiu interpretáciu profilovacích dát.

6.1 Návrh

Navrhovaný vizualizačný modul je implementovaný ako súčasť ekosystému Perun a slúži na efektívne zobrazovanie a analýzu profilovacích dát v interaktívnom SVG formáte. Tento formát nevyžaduje žiadne externé knižnice, čo eliminuje závislosti a umožňuje neobmedzenú kompatibilitu s rôznymi verziami Pythonu. Vizualizácia využíva HTML značkovanie, čo znamená, že výstup je vytváraný len skladaním reťazcov, čo je rýchle a efektívne.

Pri interakcii s grafom, kliknutie na funkciu zväčší danú časť na podrobnejšie preskúmanie, zatiaľ čo detailné informácie o funkcii sú zobrazené pri prejdení kurzorom nad príslušným segmentom (*hover* efekt). Počet volaní funkcie je vizuálne identifikovateľný na základe farby segmentu, pričom šírka segmentu odráža pomer času stráveného v danej funkcii k celkovému času vykonávania programu a funkcie, z ktorej bola vyvolaná.

Vizualizačný nástroj je navrhnutý s ohľadom na maximálnu integráciu a kompatibilitu s existujúcimi procesmi v Perune, umožňuje rýchlu adaptáciu na nové verzie Pythonu a efektívnu analýzu priamo v prehliadači.

[1] `https://github.com/brendangregg/FlameGraph`

6.2 Implementácia

6.2.1 `flamegraph.py`

Modul `flamegraph.py` slúži ako medzistupeň v procese vizualizácie profilov. Jeho základnou funkcionalitou je transformácia surových profilovacích dát (profilov) do štruktúrovaného zoznamu volaní jednotlivých funkcií pomocou existujúceho modulu `convert.py` nástroja Perun. Táto transformovaná forma je pripravená na ďalšie spracovanie modulom `svg_builder.py`, ktorý z týchto dát vytvára samostatné SVG vizualizácie.

Okrem transformácie dát modul `flamegraph.py` obsahuje aj funkciu na porovnanie dvoch rozličných profilov. Táto funkcia umožňuje identifikovať a zvýrazniť rozdiely medzi dvoma profilmi, čo je užitočné pri analýze výkonnosti a optimalizácii kódu.

6.2.2 `svg_builder.py`

Modul `svg_builder.py` je zodpovedný za generovanie interaktívnych SVG vizualizácií profilovacích dát, pričom jeho implementácia je inšpirovaná nástrojom `flamegraph.pl` od Brendana Gregga. Tento modul je implementovaný ako *singleton*, čo znamená, že existuje len jedna inštancia triedy `SVG`, ktorá zabezpečuje centralizované spracovanie a generovanie grafických výstupov.

- **Singleton Pattern:** Trieda `SVG` využíva dizajnový vzor *singleton* na zabezpečenie, že existuje iba jedna inštancia objektu, ktorá koordinuje všetky operácie súvisiace s generovaním SVG. Je to dôležité na udržanie konzistencie a efektivity spracovania, najmä keď sa manipuluje s veľkým množstvom dát.

- **Transformácia dát do SVG:** Po prijatí predspracovaných dát od `flamegraph.py`, `svg_builder.py` transformuje tieto dáta do SVG formátu. Používa preddefinované šablóny na vykreslenie elementov grafu, kde každá funkcia a jej metriky sú reprezentované graficky.

- **Interaktívne funkcie:** Generované SVG súbory sú vybavené rozšírenou interaktivitou. Okrem klikateľných elementov, ktoré umožňujú užívateľom zväčšiť určité oblasti grafu. Pri prejdení kurzorom nad konkrétnymi segmentami grafu sa automaticky zobrazia detaily o danej funkcii, ako sú časové údaje, prípadne vyvolané výnimky (tzv. *exceptions*). Toto umožňuje užívateľom získať okamžitý prehľad o výkonnostných metrikách bez potreby klikania, čo zjednodušuje analýzu a poskytuje rýchlejší prístup k dátam.

- **Vizuálne rozlíšenie:** Modul aplikuje vizuálne kódovanie založené na analýze výkonnosti rôznych častí kódu. Rôzne farby sú použité na označenie počtu volaní funkcií v pomere k celkovému počtu volaní všetkých funkcií. Farby sa menia v závislosti od počtu volaní jednotlivých funkcií, čo uľahčuje identifikáciu často volaných alebo kritických funkcií. Šírka každého grafického elementu (bloku) v grafe je proporcionálna k pomeru, ktorý zobrazuje čas trvania danej funkcie vzhľadom na celkovú dobu trvania programu. Táto metóda umožňuje užívateľom rýchlo vizuálne identifikovať, ktoré funkcie zaberajú najviac času v rámci celkového vykonávania aplikácie.

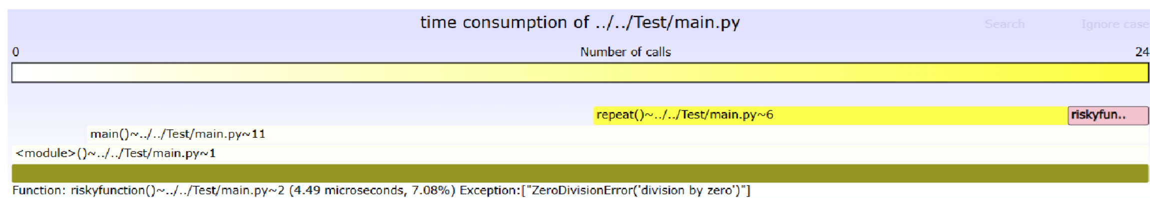
Ukážka výsledného riešenia sa nachádza na obrázku 6.1

6.3 Známe obmedzenia

V rámci implementácie vizualizačného nástroja existujú niektoré známe obmedzenia, ktoré by mohli byť predmetom ďalšieho vývoja a optimalizácie.

- **Možnosť priamej integrácie.** Odstránenie závislosti na moduloch `flamegraph.py` a `convert.py`, ktoré slúžia na spracovanie profilov, ktoré by mohlo byť vykonávané priamo v module `svg_builder.py`. Táto zmena by potenciálne mohla viesť k miernym vylepšeniam v efektívnosti spracovania, znižovaní pamäťových požiadaviek a celkovej rýchlosti generovania SVG.
- **Optimalizácia výkonu.** Hoci aktuálne riešenie je funkčné, existuje priestor na jeho optimalizáciu. Integrácia všetkých potrebných funkcionalít priamo do jedného modulu by mohla zjednodušiť kód a zlepšiť výkon.

Tieto návrhy na vylepšenie by mohli byť adresované v budúcich verziách nástroja, čím by sa nielen zvýšila jeho efektivita, ale aj zjednodušila prípadná modifikácia a rozšírenie funkcionalít.



Obr. 6.1: Na obrázku je zobrazená názorná ukážka vizualizácie skriptu, ktorého implementácia je vo výpise 6.1 , ktorý overuje funkcionalitu daného modulu. Vpravo hore je možné vidieť prvok zafarbený na červeno, čo indikuje, že vo funkcii došlo k vyvolaniu výnimky. Odtieň žltej farby znázorňuje pomer počtu volaní danej funkcie k celkovému počtu volaní všetkých funkcií. Na spodnej časti obrázka sú zobrazené detaily prvku, nad ktorým sa práve nachádza kurzor.

```python
def risky_function():
    return 10/0

def repeat(i):
    print(i)

def main():
    [repeat(i) for i in range(20)]
    try:
        print(risky_function())
    except:
        print(f"Error")

if __name__ == "__main__":
    main()
```

Výpis 6.1: Demonštrácia opakovania funkcie a vyhadodenia a zachytávania výnimky v Python skripte pre názornu ukážku funkcionality modulu `FlameGraph`.

Kapitola 7

Experimenty

Táto kapitola demonštruje overenie funkcionality novovytvoreného nástroja na zber dát o výkonnosti Python programov a ich následnej vizualizácii. Je v nej predstavené praktické využitie tohto nástroja na analýzu Python programov. Kapitola poskytne prehľad o prostredí, v ktorom boli experimenty vykonávané, analyzuje vplyv nástroja na profilovaný program.

7.1 Metodológia

Táto sekcia obsahuje podrobný popis metodológie, vrátane výberu dát, použitých nástrojov a procesu analýzy. Rovnako sa tu nachádza aj stručné predstavenie programov, na ktorých boli experimenty vykonané.

Pre každý experiment prebehlo päť meraní, pričom na metriku ako je doba vykonávania programu alebo doba vykonávania jednotlivých funkcií, bola použitá mediánová hodnota.

Tabuľka nižšie uvádza špecifikáciu prostredia, teda počítača na ktorom boli vykonané experimenty:

Komponent	**Špecifikácia**
Architektúra	x86_64
CPU	AMD Ryzen 7 7700 @ 4.50GHz
RAM	32 GB DDR5 @ 6000 MHz
Operačný systém	Windows 11 – WSL2 Ubuntu 22.04.4 LTS

Tabuľka 7.1: Špecifikácia prostredia pre experimenty

7.2 Experiment 1: Réžia profilovania

Tento experiment je zameraný na kvantitatívne zhodnotenie vplyvu profilovacieho nástroja na výkon programu. Experiment bol vykonaný na trojici benchmarkov z knižnice `PyPerformance`[1], ktoré sú špecificky navrhnuté na testovanie rôznych aspektov výpočtovej náročnosti Python programov. Cieľom je zistiť, ako použitie profilera ovplyvňuje celkový čas vykonávania programu a identifikovať prípadnú réžiu spôsobenú samotným profilovaním.

Pri testovaní boli sledované štyri rôzne konfigurácie:

[1] https://github.com/python/pyperformance

1. **noprof**: Chod programu bez aktivácie profilera, ktorý poskytne základnú líniu (tzv. *baseline*) pre výkon.
2. **all**: Chod programu s kolektorom (modul `collect.py` popísaný v sekcii 5.3.2).
3. **filters**: Chod programu s kolektorom, ktorý používa špecifické filtre na redukciu počtu sledovaných udalostí.
4. **processing**: Chod programu s profilerom vrátane spracovania zachytených udalostí[2] (modul `parser.py` popísaný v sekcii 5.3.3).

Tabuľka 7.2 poskytuje prehľadný súhrn výsledkov z každého benchmarku, zahŕňajúci porovnanie času vykonávania programu v rôznych testovacích konfiguráciách:

Benchmark	noprof	all	filters	processing
bm_nbody	9.99012 sec	10.17219 sec	10.07530 sec	16.82451 sec
bm_json_loads	10.98927 sec	11.08884 sec	11.01884 sec	17.69742 sec
bm_scimark_fft	14.98458 sec	15.11675 sec	15.02738 sec	21.76315 sec

Tabuľka 7.2: Porovnanie času vykonávania benchmarkov s rôznymi nastaveniami profilera. Pri každom behu kolektor zachytil približne 40000 udalostí[2].

Použitý filter V experimente bol použitý filter, ktorý odstránil volania všetkých funkcií pochádzajúcich z externých knižníc, čím sa výrazne zredukoval počet monitorovaných funkcií len na hlavnú funkciu `main`. V dôsledku aplikácie tohto filtru sa zjednodušil proces profilovania a mierne sa znížil celkový čas vykonávania programu s profilovaním. Spracovanie nazbieraných udalostí sa tak zredukovalo na pár milisekúnd, a preto táto konfigurácia nie je zobrazená v tabuľke 7.2.

7.2.1 Vyhodnotenie experimentu 1

V experimente 1 bolo zistené, že vplyv profilovacieho nástroja na výkon programu je pri základnom zbieraní udalostí pomocou kolektoru minimálny. Najvýznamnejšie zvýšenie réžie sa však prejavilo pri konfigurácii **processing**, kde bolo zahrnuté aj spracovanie nazbieraných udalostí. Pri tejto konfigurácii bol počet volaných funkcií v rámci hlavného procesu približne 20 000, čo je výrazne menej ako reálny počet vykonaných funkcií. Tento rozdiel je spôsobený tým, že profilovací nástroj vyvinutý v rámci tejto bakalárskej práce nedokáže zachytávať udalosti z iných procesov, respektíve subprocesov spustených v rámci programu. Tento faktor viedol k menšiemu počtu zachytených udalostí, čo malo za následok nižšiu réžiu. Správne nastavenie a optimalizácia profilovacieho procesu však môže výrazne zmierniť jej dopad. Experiment ukázal, že stupeň zanorenia jednotlivých funkcií a množstvo spracovávaných dát sú kľúčové aspekty, ktoré ovplyvňujú výsledný výkon aplikácie.

7.3 Experiment 2: Profilovanie modulu parser.py

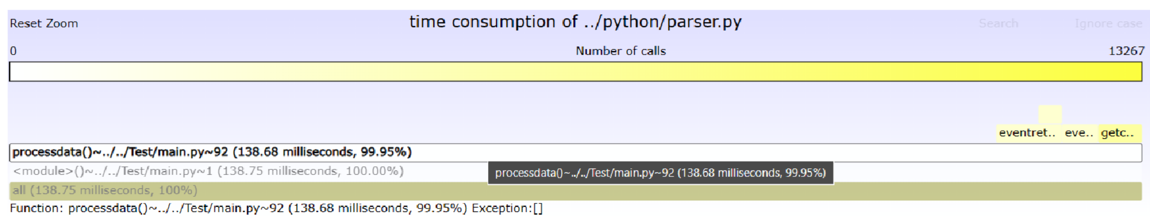
V rámci tohto experimentu som sa zameral na analýzu vplyvu modulu `parser.py`, popísaného v sekcii 5.3.3, na celkový výkon Python profileru vytvorenom v rámci tejto bakalárskej

[2] Profiler nepodporuje *subprocesy*, čiže počet zachytených udalostí je značne limitovaný na približne 20000 volaní funkcií v rámci hlavného procesu.

práce. Modul bol testovaný na základe udalostí získaných kolektorom nad pokusným programom, ktorý zachytil celkom 5306 udalostí, odpovedajúcich 2653 volaniam funkcií. Nad nazbieranými udalosťami bol spustený modul `parser.py`. Profilovanie tohto modulu vygenerovalo 26534 udalostí, odpovedajúcich 13267 volaniam funkcií. Zistený rozdiel v počte udalostí a volaní funkcií ukazuje na rozsiahle spracovanie, ktoré modul vykonáva.

7.3.1 Analýza časovej náročnosti modulu parser.py

Z analýzy výkonu modulu `parser.py` pomocou flamegraphu je jasné, že najväčšiu réžiu predstavuje metóda `processData`. Napriek tomu, že bola volaná iba raz, jej exkluzívny čas trvania tvorí približne 87% celkového času vykonávania modulu. Tento *flamegraph* poskytuje náhľad do časovej náročnosti jednotlivých funkcií a ich vplyvu na celkový výkon.



Obr. 7.1: Flamegraph modulu `parser.py` zobrazujúci časovú náročnosť funkcie `processData`, ktorá dominuje celkovému výkonu modulu.

7.3.2 Vyhodnotenie experimentu 2

Experiment 2 naznačil, že optimalizácia funkcie `processData` v module `parser.py` je kľúčová na zlepšenie výkonu celého profileru. Využitie paralelizmu by mohlo efektívne rozložiť výpočtové zaťaženie a umožniť detailnejšie spracovanie dát bez významnej réžie. Na základe získaných výsledkov bude tento profiler v nadväzujúcej práci optimalizovaný, s cieľom zvýšiť jeho efektívnosť a prispieť k celkovej spokojnosti užívateľov. Tento prístup by mal viesť k značnému zlepšeniu výkonu a lepšej škálovateľnosti profilovacieho nástroja.

7.4 Experiment 3: Výkon a optimalizácie

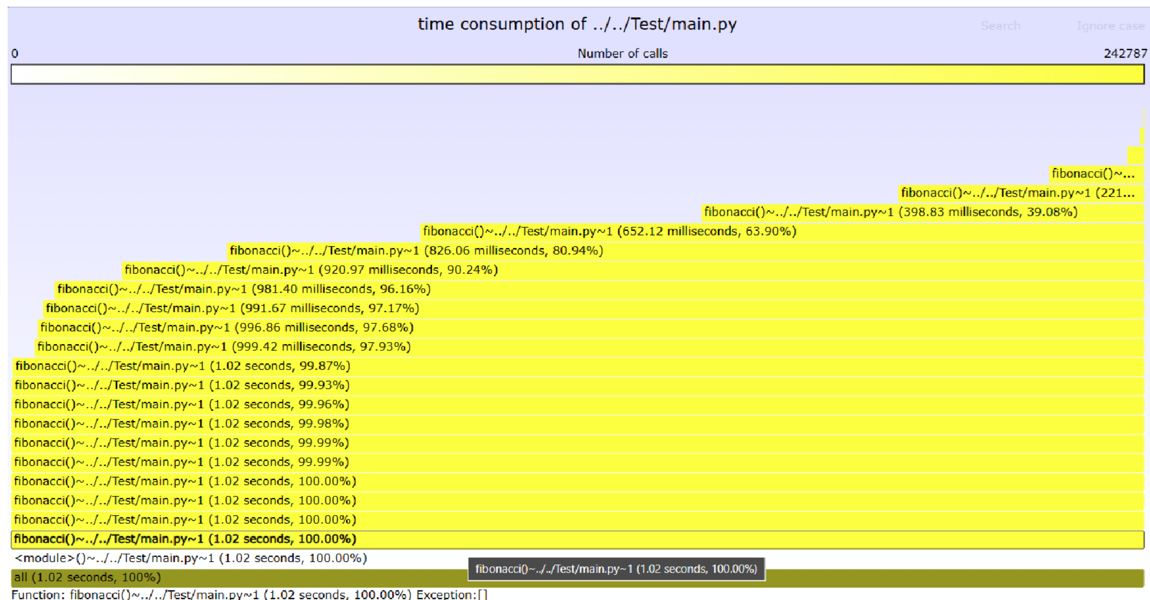
Tento experiment sa zameriava na analýzu troch rôznych implementácií výpočtu Fibonacciho čísel. Porovnáme rekurzívnu, memoizovanú a iteratívnu metódu, aby sme identifikovali, ako každá z nich vplýva na výkon a ako tieto rozdiely dokumentuje náš profiler.

7.4.1 Rekurzívna implementácia

Rekurzívna implementácia je známa svojou jednoduchosťou, ale aj vysokou výpočetnou náročnosťou kvôli mnohokrát zopakovaným volaniam funkcie, ktoré exponenciálne rastú s nárastom hodnoty `n`.

```
def fibonacci(n):
    if n <= 1:
        return n
    else:
        return fibonacci(n-1) + fibonacci(n-2)
```

Výpis 7.1: Rekurzívna implementácia Fibonacciho čísel, ktorá je matematicky priama, ale výpočtovo neefektívna s exponenciálnou časovou zložitosťou $O(2^n)$.



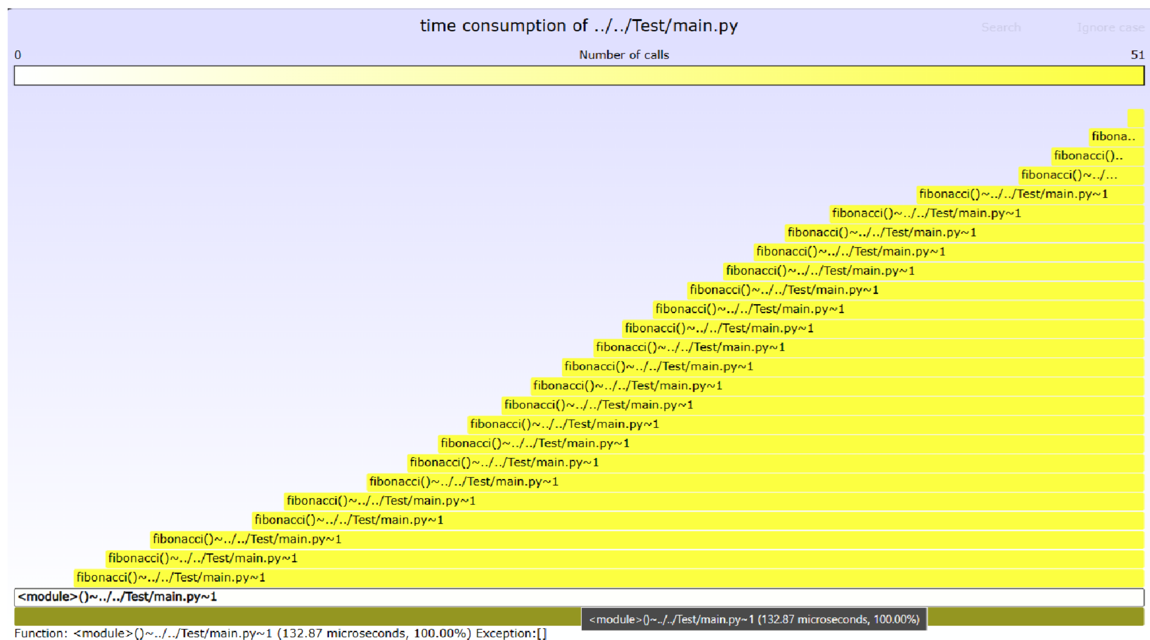
Obr. 7.2: Flamegraph zobrazujúci výkonostnú analýzu rekurzívnej implementácie Fibonacciho funkcie. Jasne žltá farba zvýrazňuje funkciu `fibonacci`, ktorá bola volaná približne 242,787-krát. Tento vysoký počet volaní odráža výpočtovú náročnosť rekurzívneho volania. Vo flamegrafe sú volania funkcie `fibonacci` na rovnakej hĺbke rekurzie vizuálne agregované do širších stĺpcov, čo znázorňuje sčítanie časov pre všetky volania na tej istej úrovni. Tento zjednodušený vizuálny prístup umožňuje rýchlo identifikovať úrovne s najväčším výpočtovým zaťažením.

7.4.2 Memoizovaná implementácia

Memoizácia je technika, ktorá zlepšuje efektivitu rekurzívnych funkcií ukladaním výsledkov volaní funkcie do medzipamäte (tzv. *cache*). Týmto spôsobom sa predchádza opakovanému vykonávaniu výpočtov pre už známe vstupy, čo výrazne znižuje počet volaní funkcie a celkovú výpočtovú náročnosť. Táto technika je obzvlášť užitočná pri problémoch, kde existuje vysoký počet prekrývajúcich sa volaní, ako je to pri rekurzívnej implementácii Fibonacciho čísel.

```python
def fibonacci(n, memo={}):
    if n in memo:
        return memo[n]
    if n <= 1:
        return n
    memo[n] = fibonacci(n-1, memo) + fibonacci(n-2, memo)
    return memo[n]
```

Výpis 7.2: Memoizovaná implementácia Fibonacciho čísel využívajúca memoizáciu pre zlepšenie časovej zložitosti na lineárnu $O(n)$, efektívne znižujúca počet potrebných rekurzívnych volaní.



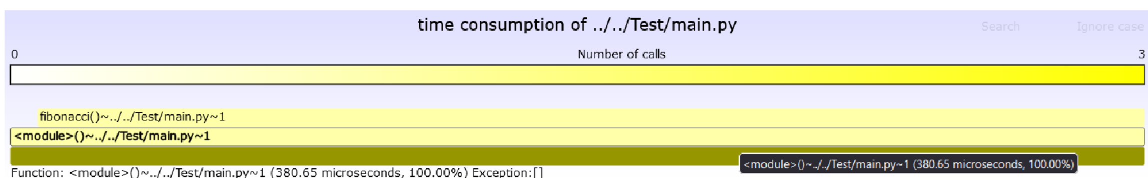
Obr. 7.3: Flamegraph zobrazujúci výkonostnú analýzu memoizovanej implementácie Fibonacciho funkcie. Na rozdiel od rekurzívnej metódy, memoizácia výrazne znižuje počet volaní funkcie, čo je zrejmé z absencie viacerých volaní na rovnakých úrovniach. V dôsledku použitia *cache* mechanizmu sú výsledky predchádzajúcich výpočtov opätovne využité, čo minimalizuje potrebu ďalších volaní a výrazne znižuje výpočtovú náročnosť.

7.4.3 Iteratívna implementácia

Iteratívna metóda na výpočet Fibonacciho čísel odstraňuje potrebu hlbokých rekurzívnych volaní tým, že používa jednoduchý cyklus na postupné vypočítanie hodnôt. Tento prístup je známy svojou vysokou výpočtovou efektivitou a minimálnymi pamäťovými požiadavkami, keďže v každom kroku cyklu sa uchovávajú len hodnoty posledných dvoch čísel.

```
def fibonacci(n):
    a, b = 0, 1
    for _ in range(n):
        a, b = b, a~+ b
    return a
```

Výpis 7.3: Iteratívna implementácia Fibonacciho čísel s lineárnou časovou zložitosťou $O(n)$. Táto metóda je efektívnejšia a jednoduchšia na pochopenie v porovnaní s rekurzívnymi prístupmi, znižuje pamäťové požiadavky a eliminuje riziko pretečenia zásobníka volaní.



Obr. 7.4: Flamegraph zobrazujúci výkonostnú analýzu iteratívnej implementácie Fibonacciho funkcie. Táto metóda odstraňuje potrebu rekurzie a prejavuje sa v flamegraph ako jednoduchá a plochá štruktúra bez hlbokých zanorení. Znížené využitie pamäte a rýchlejšie časy vykonávania sú výsledkom priameho počítania hodnôt bez opakovaných volaní funkcie. Graf ilustruje minimálnu réžiu a optimálnu výpočtovú efektivitu, čo je ideálne pri riešení problémov vyžadujúcich rýchle a efektívne výpočty.

7.4.4 Vyhodnotenie experimentu 3

Experiment porovnával tri rôzne implementácie výpočtu Fibonacciho čísel a s pomocou profilera sme boli schopní kvantitatívne zhodnotiť vplyv každej implementácie na výkon. Profiler nám umožnil identifikovať nielen výkonnostné rozdiely medzi jednotlivými implementáciami, ale aj mieru réžie, ktorú profiler pridáva, výsledky merania sú ukázané v tabulke 7.3. Celková réžia v tejto tabuľke je významne ovplyvnená používaním nástroja Perun, samotný beh kolektoru a parseru v tomto prípade je len zlomok celkového času behu profileru. To znamená, že celková réžia je pri menších, menej výpočtovo náročných programoch veľmi skreslená.

Tabuľka 7.3: Porovnanie výkonnosti implementácií Fibonacciho postupnosti

Metrika / Implementácia	Rekurzívna	Memoizovaná	Iteratívna
Celkový čas behu profileru	1713.02 s	0.03 s	0.03 s
Čas behu s kolektorom	0.966 s	0.000102 s	3.086×10^{-5} s
Čas behu bez kolektora	0.00522 s	4.013×10^{-5} s	2.329×10^{-5} s
Počet vykonaných funkcií	242,787	51	3
Počet zachytených udalostí	485,574	102	6
Režia s kolektorom	$\approx$ 185-krát	$\approx$ 2.5-krát	$\approx$ 1.3-krát
Celková réžia	$\approx$ 330000-krát	$\approx$ 750-krát	$\approx$ 860-krát

Výsledky sú dokumentované pomocou flamegrafov, ktoré poskytujú vizuálny prehľad o vplyve profilovania na výkon jednotlivých implementácií. Tieto grafy jasne ukazujú, ako profiler zasahuje do výkonnosti aplikácií. Sú dôležité aj pri hodnotení vhodnosti rôznych implementačných stratégií pre rôzne výpočtové úlohy.

Celkovo tento experiment potvrdzuje, že hoci profilovanie je extrémne užitočné na identifikáciu a analýzu výkonnostných horúcich miest (tzv. *hotspotov*), môže tiež výrazne ovplyvniť výkon aplikácie, najmä pri veľkom množstve volaných funkcií. Výrazná časť tohto spomalenia je spojená so spracovaním zaznamenaných udalostí, ktoré vyžadujú významné množstvo zdrojov oproti samotnému procesu zbierania udalostí. Táto konkrétna záťaž môže byť kritickým faktorom pri rozhodovaní o konfigurácii a použití profilera v produkčnom prostredí, najmä pri výpočetne náročných úlohách. Tieto zistenia by mali byť zohľadnené pri optimalizácii profileru a jeho použití v rôznych vývojových fázach softvéru.

Kapitola 8

Záver

Cieľom tejto bakalárskej práce bolo navrhnúť a implementovať nástroj na profilovanie Python programov schopný efektívne zbierať a vizualizovať dáta o výkone celého programu, jeho modulov a jednotlivých funkcií. Výsledný nástroj bol úspešne integrovaný do existujúceho nástroja Perun, kde sa ukázal byť schopným zachytávať a interpretovať dáta z rôznych Python programov. Zvláštna pozornosť bola venovaná modulu `parser.py`, ktorý je kľúčovou súčasťou tohto nástroja a ktorého vysoká réžia bola identifikovaná ako hlavná oblasť pre budúce zlepšenia.

Prostredníctvom experimentov vykonaných v kapitole 7, bol analyzovaný vplyv nástroja na výkon profilovaných programov a bolo zistné, že modul `parser.py` vyžaduje značnú optimalizáciu, aby sa zredukovala jeho časová náročnosť a zlepšil celkový výkon nástroja. V nadväzujúcej práci môže byť tento modul rozdelený do viacerých metód, čo by v kombinácii s využitím paralelizmu, malo prispieť k efektívnejšiemu spracovaniu dát, zníženiu réžie a zlepšeniu škálovateľnosti nástroja.

Ďalšími smermi pre nadväzujúce práce je rozšírenie funkcionality kolektora o nové možnosti, ktoré prináša aplikačné rozhranie sys.monitoring. Odstránenie známych obmedzení tohto modulu, popísaných v sekcii 5.4, a vylepšenie logiky modulu `parser.py`, by malo viesť k významnému zlepšeniu efektivity a použiteľnosti tohto nástroja. Tieto kroky by mali prispieť k lepšej škálovateľnosti a výkonnosti profilera, čím sa zvýši jeho konkurencieschopnosť voči existujúcim riešeniam.

Táto práca položila základy pre rozvoj profilovacieho nástroja pre programy napísané v jazyku Python a otvorila mnoho možností pre jeho budúce vylepšenia a rozšírenia v rámci systému Perun.

Literatúra

[1] *Cyclomatic complexity* [online]. [cit. 2024-01-27]. Dostupné z: https://en.wikipedia.org/w/index.php?title=Cyclomatic_complexity&oldid=344289447.

[2] Ball, T. The concept of dynamic analysis. *SIGSOFT Softw. Eng. Notes*. New York, NY, USA: Association for Computing Machinery. oct 1999, zv. 24, č. 6, s. 216–234. DOI: 10.1145/318774.318944. ISSN 0163-5948. Dostupné z: https://doi.org/10.1145/318774.318944.

[3] Berger, E. D., Stern, S. a Pizzorno, J. A. Triangulating Python Performance Issues with Scalene. In: *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. Boston, MA: USENIX Association, 2023, s. 51–64. ISBN 978-1-939133-34-2. Dostupné z: https://www.usenix.org/conference/osdi23/presentation/berger.

[4] Brendan, G. *Systems Performance: Enterprise and the Cloud*. 2. vyd. Pearson, 2020 [cit. 2024-01-21]. ISBN 9780136821694.

[5] Bryant, M. a Kirschner, H. *Building FunctionTrace, a graphical Python profiler* [online]. 2020 [cit. 2024-04-21]. Dostupné z: https://hacks.mozilla.org/2020/05/building-functiontrace-a-graphical-python-profiler/.

[6] Cip, S. *Yappi: Yet Another Python Profiler* [online]. [cit. 2024-04-21]. Dostupné z: https://github.com/sumerc/yappi.

[7] contributors, P. *Pyflame Documentation* [online]. 2024 [cit. 2024-04-21]. Dostupné z: https://pyflame.readthedocs.io/en/latest/index.html.

[8] Feneyrou, D. *Palanteer Documentation* [online]. 2021 [cit. 2024-04-21]. Dostupné z: https://dfeneyrou.github.io/palanteer/index.html.

[9] Fiedor, T. a Pavela, J. *Perun Documentation* [online]. [cit. 2024-02-24]. Verzia 0.21.6. 2023. Dostupné z: https://github.com/Perfexionists/perun/blob/devel/docs/pdf/perun.pdf.

[10] Foundation, P. S. *Time — Time access and conversions* [online]. [cit. 2024-04-21]. Dostupné z: https://docs.python.org/3/library/time.html#time.perf_counter.

[11] Frederickson, B. *Py-Spy: A sampling profiler for Python programs* [online]. 2023 [cit. 2024-04-21]. Dostupné z: https://github.com/benfred/py-spy.

[12] Gao, T. *VizTracer Documentation* [online]. 2020 [cit. 2024-04-21]. Dostupné z: https://viztracer.readthedocs.io/en/stable/.

[13] Hájek, V. *Analýza výkonu programů v jazyce C#*. Brno, 2023. [cit. 2024-02-24]. Bakalárska práca. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedúci práce Pavela, I. J. Dostupné z: https://www.vut.cz/studenti/zav-prace/detail/148641.

[14] Močáry, P. *Performance Analysis of Programs Based on PIN Framework*. Brno: [b.n.], 2022 [cit. 2024-02-24]. Dostupné z: https://www.vut.cz/studenti/zav-prace/detail/145076.

[15] Nielson, F., Nielson, H. a Hankin, C. *Principles of Program Analysis*. 1. vyd. Springer Berlin Heidelberg, 2015 [cit. 2024-01-18]. ISBN 9783662038116. Dostupné z: https://link.springer.com/book/10.1007/978-3-662-03811-6.

[16] Pavela, J. *Efficient Techniques for Program*. Brno, 2020. [cit. 2024-02-24]. Diplomová práca. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedúci práce Mgr. Adam Rogalewicz, P. doc. Dostupné z: https://www.vut.cz/studenti/zav-prace/detail/129252.

[17] Python Software Foundation. *Timeit — Measure execution time of small code snippets* [online]. [cit. 2024-04-21]. Dostupné z: https://docs.python.org/3/library/timeit.html.

[18] Python Software Foundation. *The Python Standard Library: profile - Performance analysis for Python* [online]. 2023 [cit. 2024-04-21]. Dostupné z: https://docs.python.org/3/library/profile.html.

[19] Python Software Foundation. *Sys.monitoring — Execution event monitoring* [online]. 2023 [cit. 2024-04-21]. Dostupné z: https://docs.python.org/3/library/sys.monitoring.html#module-sys.monitoring.

[20] Reiss, S. Event-based performance analysis. In:. 2003 [cit. 2024-01-23]. DOI: 10.1109/WPC.2003.1199191. Dostupné z: https://ieeexplore.ieee.org/document/1199191?arnumber=1199191.

[21] Rickerby, J. a contributors. *Pyinstrument Documentation* [online]. 2021 [cit. 2024-04-21]. Dostupné z: https://pyinstrument.readthedocs.io/en/latest/home.html.

[22] Subramaniam, K. a Thazhuthaveetil, M. Effectiveness of sampling based software profilers. In:. 1994 [cit. 2024-01-23]. DOI: 10.1109/STRQA.1994.526376. Dostupné z: https://ieeexplore.ieee.org/document/526376?arnumber=526376.

[23] Yegulalp, S. 9 fine libraries for profiling Python code. [online]. 2022, [cit. 2024-04-21]. Dostupné z: https://www.infoworld.com/article/3600993/9-fine-libraries-for-profiling-python-code.html.