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 GO

PERFORMANCE ANALYSIS OF GO PROGRAMS

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE **ANDREJ NEŠPOR**
AUTHOR

VEDOUCÍ PRÁCE **Ing. JIŘÍ PAVELA**
SUPERVISOR

BRNO 2024

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

Zadání bakalářské práce

157244

Ústav:	Ústav inteligentních systémů (UITS)
Student:	**Nešpor Andrej**
Program:	Informační technologie
Název:	**Analýza výkonu programů v jazyce Go**
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 Go.
3. Navrhněte a implementujte nástroj, který bude měřit spotřebu alespoň jednoho zdroje v Go 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 Go.
6. Demonstrujte řešení na alespoň jednom netriviálním projektu.

Literatura:

- Oficiální stránky projektu Perun: https://github.com/Perfexionists/perun
- Go Diagnostics: https://go.dev/doc/diagnostics
- 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

Cílem této práce je rozšířit výkonnostní verzovací systém Perun implementací modulu pro profilování programů napsaných v jazyce Go. Tento modul implementuje profilovací nástroj technikou instrumentace volání a návratů funkcí pomocí technologie eBPF. Nástroj dokáže sbírat reálný čas běhu funkcí a jejich trasy volání. Zároveň byl implementován nový způsob vizualizace naměřených dat pomocí tzv. Sankey grafu, což usnadní jejich interpretaci. S vytvořeným profilovacím nástrojem pak provádíme pár experimentů, abychom ověřili jeho funkčnost a demonstrovali jeho použití.

Abstract

The goal of this thesis is to extend the performance versioning system Perun by implementing a module for profiling programs written in the Go language. This module implemets the profiler by instrumenting function calls and returns using eBPF technology. The tool can collect function run times and their traces. We can then interpret the outputted profiles as a Sankey diagram. Additionally we implemented a new way of visualizing the measured data using the so-called Sankey graph, which will help with their interpretation. Using the developed profiler we conduct a few experiments to verify its functionality and demonstrate its use.

Klíčová slova

Go, profilování, výkonnostní testování, Perun, dynamická analýza, eBPF, vizualizace, Sankey

Keywords

Go, profiling, performance testing, Perun, dynamic analysis, eBPF, visualization, Sankey

Citace

NEŠPOR, Andrej. *Analýza výkonu programů v jazyce Go*. Brno, 2024. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Jiří Pavela

Analýza výkonu programů v jazyce Go

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Jiřího Pavely. Další informace mi poskytl Ing. Tomáš Fiedor, Ph.D. Uvedl jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

.....
Andrej Nešpor
7. května 2024

Poděkování

Rád bych tímto poděkoval vedoucímu Ing. Jiřímu Pavelovi a Ing. Tomáši Fiedorovi, Ph.D. za konzultace, rady, připomínky a odbornou pomoc s vývojem nástroje a textem práce.

Obsah

Kapitola 1

Úvod

V dnešní době je testování neodmyslitelnou součástí vývoje softwaru. Poskytuje vývojářům důležité informace o kvalitě a spolehlivosti produktu. Vzhledem k důležitosti testovaní existuje spousta odvětví a technik, které různé potenciální nedostatky testují. Jedná se například o testování funkčnosti, bezpečnosti, ale i o testování výkonu a spotřeby zdrojů programu.

Techniky analýzy výkonu jsou při vývoji programů běžně využívány. Hledání výkonnostních problémů je ale o mnoho těžší než například hledání problémů s funkčností programu. Je to dáno tím, že výkonnostní problémy se mohou objevit až při velmi specifických podmínkách. Aby to nebylo málo, dané podmínky je velice těžké, někdy skoro nemožné, replikovat. Navzdory její obtížnosti dokáže ale analýza výkonu vylepšit celkovou kvalitu vyvíjeného software.

Jednou z technik analýzy výkonu je profilování, které provádí analýzu za běhu sledovaného programu a řadíme jej tedy do skupiny dynamické analýzy. Profilovací nástroje (tzv. *profilery*) dokáží sledovat různé metriky a poskytovat detailní informace o výkonu profilovaného programu. Profilovací nástroje se liší hlavně zaměřením, a to proto, že každý z nich je navržen tak, aby sledoval specifické metriky a chování programů napsaných v různých jazycích.

Dnes je aktivně vyvíjena spousta programovacích jazyků a každým rokem se jejich počet rozšiřuje. Jeden takový jazyk je *Go*, který nás v rámci této bakalářské práce bude zajímat. Jedná se o moderní programovací jazyk vyvíjený společností *Google* od roku 2007. Je zaměřen na efektivitu a jednoduchost použití. Vyznačuje se statickým typováním s odvozováním typů a podporou paralelního programování. Pro analýzu výkonu programů v jazyce *Go* existuje několik nástrojů, všechny ale mají nějaké nedostatky. Navíc žádný z těchto nástrojů nepodporuje automatizaci testování výkonu a správu naměřených dat.

Cílem této bakalářské práce je implementovat profilovací nástroj pro programy v jazyce *Go* měřící dobu trvání jednotlivých funkcí a výsledný *profiler* integrovat do výkonnostního verzovacího systému *Perun*, vyvíjeného výzkumnou skupinou *VeriFIT*. Dále chceme vytvořit vizualizaci, která se hodí k programům napsaným v jazyce *Go* a naměřeným datům. Motivací je rozšířit *Perun* o možnost profilovat i programy v jazyce *Go*.

Struktura práce. Kapitola 2 představuje téma analýzy programů, její typy a způsoby jejího provádění. Uvede také její rozdělení podle granularity měření a příklady metrik, které mohou být za jejího použití získány. Kapitola 3 obsahuje seznámení s výkonnostním verzovacím systémem *Perun*. Jako další následuje v kapitole 4.1 představení technologie *eBPF* a různých způsobů, jak s ní pracovat. Kapitola 4.2 poté pokrývá analýzu již existujících

nástrojů pro profilování programů v jazyce *Go*. Jako další v kapitole 5 analyzujeme požadavky na výsledný profilovací nástroj. Tyto požadavky poté využijeme v kapitole 6, která se zabývá návrhem a implementací profilovacího nástroje. Tato kapitola obsahuje nejen popis implementace, ale i výčet problémů, na které jsme během ní narazili, známá omezení nástroje a popis modulu pro vizualizaci nasbíraných dat. Poslední kapitola 7 demonstruje výsledný profilovací nástroj na pár jednoduchých programech a na jednom středně velikém projektu.

Kapitola 2

Analýza programu

V této kapitole začneme s představením statické a dynamické analýzy spolu s běžně užívanými technikami pro sledování chování běžících systémů a aplikací — *vzorkování, instrumentace a sledování událostí*. Tyto techniky jsou obzvláště užitečné při analýze výkonu díky informacím, které jsme pomocí nich schopni získat, ať jde o využívání prostředků nebo dobu běhu sledovaného systému. Nejdříve vysvětlíme základní terminologii analýzy a pozorování, poté ukážeme vybrané nástroje pro dynamickou instrumentaci, sledovaní událostí a profilování.

2.1 Statická a dynamická analýza

Analýzu programu můžeme kategorizovat do dvou skupin podle toho, kdy k analýze dojde [34].

Statická analýza provádí analýzu zdrojového nebo binárního spustitelného kódu bez jeho vykonání. Statická analýza tedy zahrnuje spoustu standardních technik jako je *analýza toku dat* nebo *typová analýza*. Statickou analýzu využívá spousta nástrojů, zejména nástroje pro kompilaci kódu (tzv. *překladače*), které ji využívají pro kontrolu správnosti kódu nebo analýzu kódu za účelem optimalizace programu.

Dynamická analýza naopak provádí analýzu programu během jeho běhu. Důležitou částí většiny dynamických analýz je *instrumentace*. Dynamickou analýzu využívají nástroje jako jsou *profilery*, výkonnostní analyzátory a vizualizační nástroje. Mezitím co statická analýza zahrnuje všechny možné scénáře spuštění programu, dynamická analýza dokáže analyzovat pouze ty cesty programem, které se opravdu během běhu programu provedou.

2.2 Profilování

Technika profilování spadá do kategorie dynamické analýzy programu. Nástroje, které danou techniku realizují se nazývají *profilery*. Jsou to nástroje, které dokáží sledovat nebo měřit různé metriky profilovaného programu (např. doba běhu funkcí nebo alokace paměti) za jeho běhu. Nasbíraná data poté analyzují a případně i vhodně vizualizují. Profilování je často využíváno vývojáři pro identifikaci výkonnostních chyb, optimalizaci výkonu a odhalení možných problémů např. s pamětí nebo využíváním zdrojů. *Profilery* mohou využívat

různé techniky pro sběr dat, podle kterých jsou klasifikovány. Tato sekce je věnována vysvětlení a přiblížení třem nejčastěji využívaným způsobům sběru dat [26, 35].

2.2.1 Sledování událostí

Nástroj pro sledování událostí (tzv. *Tracer*) funguje na bázi sledování událostí generovaných profilovaným programem. Mezi sledované události můžou patřit systémová volání, volání a návraty z funkcí – ať už v jádru operačního systému nebo v uživatelském prostoru – nebo síťové události jako příchozí a odchozí *pakety*. Typická charakteristika tohoto přístupu je velké množství nasbíraných dat, takže se většinou využívá technik pro jejich následovné zpracování. Nevýhoda tohoto způsobu je, že dokáže získávat data pouze když nastane sledovaná událost.

2.2.2 Vzorkování

Vzorkovací *profiler* sbírá data tak, že sleduje programový zásobník volání ve fixních intervalech – buď v uplynulém čase, frekvenci procesoru, nebo prahové hodnotě čítače událostí. Na rozdíl od sledování událostí dokáže tento přístup výrazně snížit množství výkonnostní režie profilovaného programu. Je to ale většinou na úkor přesnosti a věrohodnosti získaných dat. Nastane-li výkonnostní chyba mezi intervaly sledování, nemá ji *profiler* jak zachytit.

2.2.3 Instrumentace programu

Instrumentace je technika umožňující detailní sledování vybraných lokací aplikace během jejího spuštění. Je toho dosaženo vložením vlastního kódu nebo softwarových háčků do zdrojového kódu programu nebo jeho obrazu v paměti. *Profiler* na základě instrumentačního kódu nasbírá potřebná data. Instrumentace se dále dělí na statickou a dynamickou podle toho, kdy je vlastní kód do programu vložen [34].

Statická instrumentace

Instrumentace je považována za statickou, pokud je instrumentační kód vložen před spuštěním programu. Instrumentace může být provedena buď na zdrojovém nebo na binárním spustitelném kódu. Tento přístup však není dostatečně flexibilní a ne vždy plně automatizovaný pro použití v produkčním prostředí.

Dynamická instrumentace

Jelikož je statická instrumentace v některých situacích limitující a nedostatečná, využívá dynamická instrumentace jiný přístup, který umožňuje přidat instrumentační kód i za běhu programu. A to bez jakékoliv úpravy zdrojového kódu nebo procesu překladu.

2.3 Granularita profilování

Profilování může být prováděno na různých stupních granularity, kde každý stupeň má své výhody a nevýhody. Základní typy granularity jsou tyto: [27].

Granularita instrukcí. Jedná se o nejvyšší stupeň granularity. Znamená měření na základě jednotlivých instrukcí strojového kódu. Je to nejpřesnější granularita profilování, na úkor značného zpomalení. Je možné jí dosáhnout pouze za použití instrumentace programu.

Granularita řádků a základních bloků. Profiluje na základě jednotlivých řádků zdrojového kódu nebo základních bloků[1]. Základní bloky mohou, ale nemusí být více granulární než řádky. Je to přesnější granularita profilování, stále ale na úkor značného zpomalení. Je možné jí dosáhnout pouze za použití instrumentace programu.

Granularita funkcí. Profiluje na bázi celých metod a funkcí. Dokáže díky tomu lokalizovat problém a identifikovat funkci, ve které nastal. Efektem je zrychlení oproti řádkovému stupni granularity. Využívá se u technik sledování událostí a instrumentace programu.

Granularita selektivní. Selektivní granularita je kombinací výše zmíněných stupňů granularity.

2.4 Metriky

Pomocí *profileru* můžeme sbírat a měřit metriky, které využíváme k odhalení různých výkonnostních problémů tak, abychom je mohli opravit. Tato sekce vysvětlí několik základních metrik, které mohou *profilery* měřit [17].

- Počet volání — statistika, kolikrát byly dané funkce volány. Jedná se o základní metriku.
- Čas *Wall clock* — celkový čas nutný pro provedení funkce, včetně času stráveného čekáním na vstupně-výstupní operace, plánovač apod.
- Čas CPU — aktivní čas provádění funkce, nezapočítává čas strávený čekáním.
- Trasa (tzv. *Trace*) — jedná se o cestu k měřené funkci za účelem její identifikace. Metrika je většinou doplňována dalšími metrikami. Měřená funkce může být volána v různých kontextech a pomocí trasy zjistíme, které kontexty mohou být problematické.
- Alokace paměti — pomáhá nám sledovat, jak je alokovaná paměť využívána. Kde v programu je využívána a kdy.
- Automatické uvolnění paměti (tzv. *Garbage Collection* nebo *GC*) — některé programovací jazyky využívají automatické uvolnění paměti, která již není používaná. Metrika sleduje, jak často se *GC* volá a kolik paměti uvolňuje.
- Počet provedení základního bloku nebo instrukce — statistika, kolikrát byly za běhu základní blok nebo instrukce provedeny.
- Exkluzivní čas — celkový čas strávený pouze v této funkci.
- Inkluzivní čas — celkový čas strávený v této funkci, včetně času stráveného ve funkcích volaných touto funkcí.
- Počet výskytů funkce na zásobníku volání — kolikrát se v daném okamžiku nachází funkce na zásobníku volání. Používá se například u vzorkování.

[1] sekvence kódu bez větvení kromě vstupu a výstupu

Kapitola 3

Perun

Cílem bakalářské práce je integrace *profileru* pro programy v jazyce *Go* do výkonnostního verzovacího systému *Perun*. V této kapitole si tedy systém představíme. Kapitola vychází z dokumentace *Perunu* [21] a z prací, které tento systém historicky rozšiřovaly [27, 35, 30, 31].

3.1 O nástroji

Performance **Un**der Control (*Perun*) je výkonnostní verzovací systém s otevřeným zdrojovým kódem, který byl vytvořen výzkumnou skupinou *VeriFIT* na Fakultě informačních technologií Vysokého učení technického v Brně. Je to obálka (tzv. *wrapper*) přes systémy správy verzí (tzv. *Version Control Systems* nebo *VCS*), jako například *Git*, a propojuje verze projektu s korespondujícími výkonnostními profily. Nabízí také sadu nástrojů pro generování, zpracování a interpretaci výkonnostních profilů. Díky přístupu k historii projektu a výkonnostním profilům se *Perun* snaží pravidelně sledovat výkonnostní stav projektu a detekovat jakékoliv změny ve výkonnosti již v moment, kdy se objeví. Snaží se toho dosáhnout pomocí měření a vyhodnocení výkonnostních metrik pokaždé, kdy je zveřejněna nová verze projektu (*commit* nebo *pull-request*), a následného porovnání získaných výsledků s výsledky poslední verze projektu. Na Obrázku 3.1 je ilustrován zamýšlený průběh vývoje projektu za pomoci nástroje *Perun*.

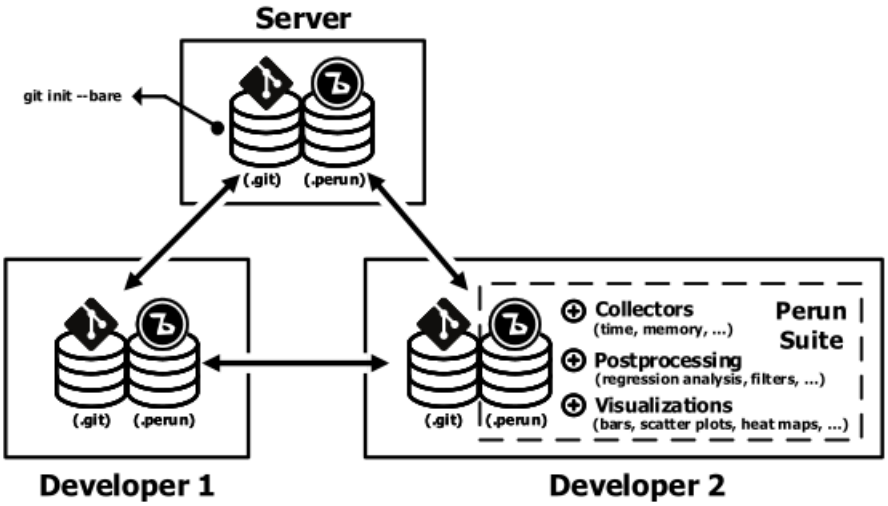
Perun nabízí několik výhod oproti jiným způsobům ukládání a správě výkonnostních profilů, jako například ukládání do verzovacích systémů (*VCS*) nebo databází.

Kontext

Díky tomu, že ukládá výkonnostní profily paralelně ku *VCS* a přiřazuje je specifickým verzím projektu, poskytuje *Perun* kontext pro pomoc s identifikací zdroje výkonnostního problému nebo optimalizacemi.

Automatizace

Perun nabízí konfigurovatelnou automatizaci sběru výkonnostních profilů pomocí háčků (tzv. *hooks*). Tyto háčky mohou být nastaveny na různé akce používaného verzovacího systému (např. Git *pull-request* nebo *commit*). Háčky spustí sekvence příkazů *Perunu* (tzv. *Jobs*) kdykoliv jsou definované akce detekovány. *Jobs* jsou inspirovány systémy kontinuální integrace (tzv. *Continious Integration systems*).



Obrázek 3.1: Ilustrace nástroje *Perun* nad verzovacím systémem *Git* v průběhu vývoje projektu [20].

Rozšiřitelnost

Kvůli široké škále projektů, na které by mohl být *Perun* využit, je systém navržen tak, aby byl jednoduše rozšiřitelný pomocí modulů. Ať už se jedná o moduly obsahující nové formáty dat, vizualizace nebo kolektory dat, tak na sobě nejsou závislé a jsou snadno rozšiřitelné o další komponenty. *Perun* rovněž využívá unifikovaný formát dat založený na formátu *JSON* pro vysokou flexibilitu.

Snadnost použití

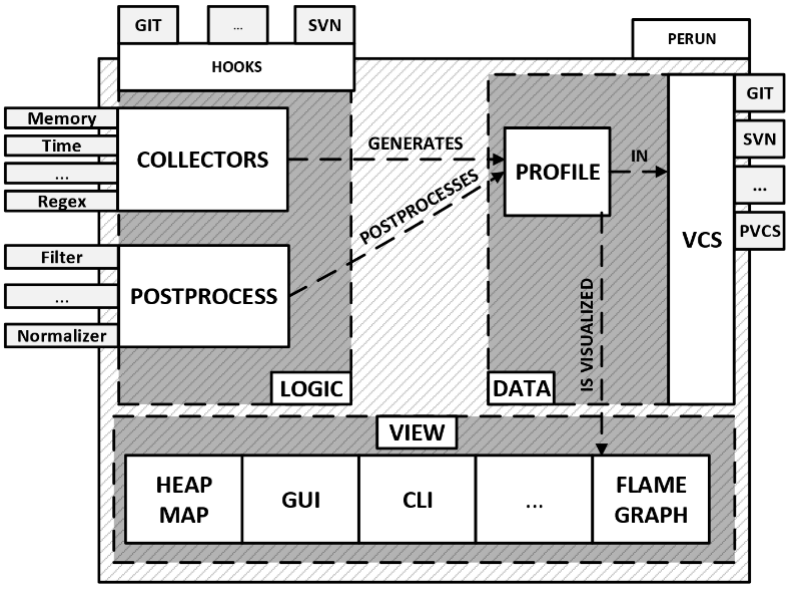
Rozhraní *Perunu* je implementováno tak, aby připomínalo rozhraní nejpoužívanějšího verzovacího systému Git. V současné době nabízí rozhraní přes příkazovou řádku (tzv. *CLI*), grafické uživatelské rozhraní (tzv. *GUI*) už je ale ve vývoji.

3.2 Architektura

Architektura nástroje *Perun* se dá rozdělit na několik komponent — *logic* (příkazy, *jobs*, spouštěče, ukládání), *data* (*VCS* a profily), *view* (vizualizace dat) a *check*. Tato sekce slouží na jejich vysvětlení a popis. Pro lepší představu a pochopení slouží Obrázek 3.2, který zmíněnou architekturu ilustruje.

Data

Tato komponenta obsahuje srdce *Perunu* — manipulaci s profily a *wrappery* nad podporovanými systémy správy verzí (momentálně Git a základní vlastní *VCS*). Profily jsou v unifikovaném formátu založeném na *JSON*, což umožňuje flexibilitu komunikace mezi komponentami a jednoduchou rozšiřitelnost.



Obrázek 3.2: Ilustrace architektury nástroje *Perun* rozdělená do jednotlivých částí (*logic*, data a *view*), integrace *VCS* a toku dat.

Logika

Komponenta logika (*logic*) má na starosti automatizaci, obsluhu vyšší logiky a samotné generování výkonnostních profilů. Její hlavní komponenty jsou rozděleny na:

- **Kolektory** sbírají výkonnostní data. Mohou být implementovány jako *wrappery* existujících profilovacích nástrojů a utilit (např. *time*) nebo to mohou být samostatné profilovací nástroje.
- **Postprocesory** aplikují různé statistické analyzátory na naměřené výkonnostní profily. Jedná se například o zpracování typu *moving average* nebo regresní analýzu [35].

Zobrazení

Komponenta zobrazení (*view*) se stará o interakci s uživatelem. Patří do ní jak CLI rozhraní pro komunikaci s uživatelem, tak i různé vizualizační nástroje pro zobrazení naměřených výkonnostních profilů a jejich statistik. Některé z podporovaných vizualizačních metod jsou:

- *Scatter plot* využívá dvourozměrné matice pro zobrazení dat jako bodů. Je vhodný pro zobrazování doby běhu funkcí, četnosti vykonání základních bloků nebo výsledků analýzy složitosti.
- *Flame Graph* je vizualizací hierarchických dat, sloužící pro zobrazení tras profilovaného programu a rychlou identifikaci nejčastěji navštěvovaných tras [24].

Kontrola

Komponenta kontrola (*check*) implementuje různé metody detekce pro zachycení výkonnostních změn v projektu. Metody mají jako vstup dva libovolné profily, často jeden před-

stavující nejnovější verzi projektu a druhý předchozí verzi projektu. Tyto profily jsou porovnány, což nám poskytne dostatečné informace o stavu nové verze projektu. Na základě typu výkonnostních dat, které profily obsahují, jsou použity různé metody, jako jsou například *Average Amount Threshold* a *Integral Method* [35].

3.3 Architektura kolektorů

Nástroj *Perun* obsahuje spoustu různých kolektorů. Všechny však mají stejnou architekturu, kde je samotný proces profilování rozdělen do čtyř fází:

- *Before* — V této fázi je kolektor inicializován a vyhledá všechna místa pro instrumentaci.
- *Collect* — Během této fáze probíhá samotný sběr výkonnostních dat. Je to jediná povinná fáze kolektoru.
- *After* — Tato fáze slouží ke zpracování a transformaci výkonnostních dat do profilů.
- *Teardown* — Na konci procesu dojde k uvolnění všech zdrojů, které kolektor využíval.

Kapitola 4

Sledování programů v jazyce GO

V této kapitole si představíme existující nástroje pro sledování programů v jazyce *Go*[1]. Začneme představením instrumentačního nástroje *eBPF* v sekci 4.1 a jeho alternativ pro dynamickou instrumentaci v sekci 4.1.4. V sekci 4.2 si poté ukážeme existující *profilery* nebo jiné programy na získání informací pro programy v jazyce *Go*.

4.1 eBPF

BPF (Berkley Packet Filter) byl donedávna považován pouze jako síťový nástroj na filtraci paketů. *BPF* byl ale rozšířen na *eBPF (extended BPF)*, který ho přeměnil na obecně použitelný virtuální stroj uvnitř jádra umožňující vytvářet pokročilé instrumentační nástroje za účelem pozorování programů a jádra. *eBPF* umožňuje spouštění izolovaných programů v privilegovaném kontextu, jako je jádro operačního systému Linux [12]. Používá se k bezpečnému a efektivnímu sledování a monitorování událostí a chování jádra bez potřeby změny zdrojového kódu jádra nebo načítání modulů jádra. *eBPF* program může být zaveden za běhu, není proto potřeba znovu překládat nebo restartovat jakoukoliv službu.

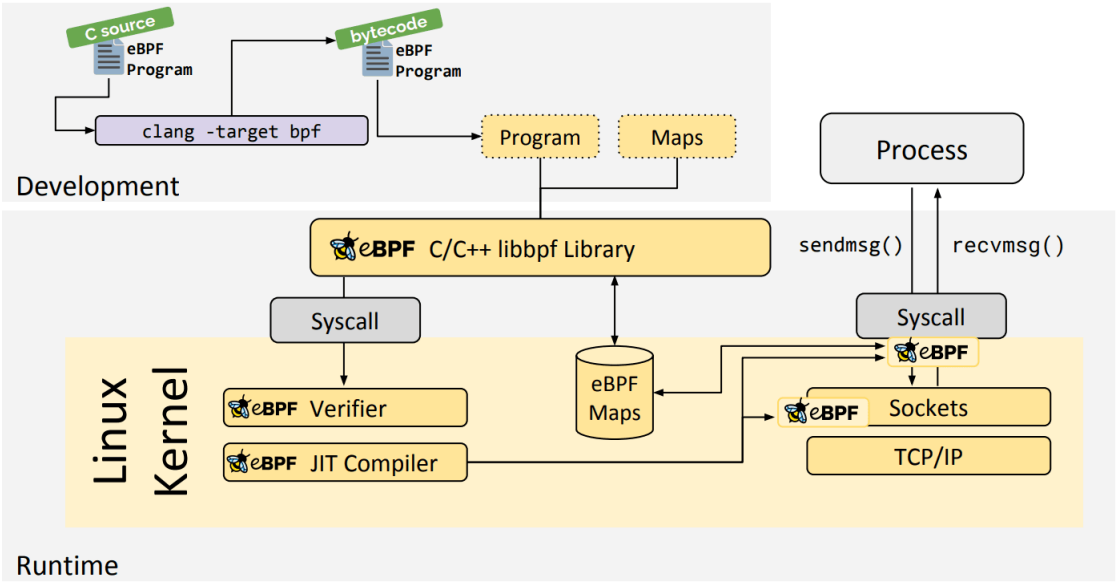
4.1.1 Přehled

eBPF je virtuální stroj v jádře systému, který má k dispozici vlastní pomocné funkce, sadu instrukcí a pravidla. Skládá se z interpretu a *JIT (Just-In-Time)* překladače, který překládá *eBPF bytecode* do nativních systémových instrukcí. Každý *eBPF* program je nejdříve zkontrolován verifikátorem, který analyzuje kód vzhledem k daným pravidlům, aby našel potenciální problémy stability a abychom se vyvarovali pádu jádra (také známo jako *kernel panic*).

Komunikace mezi jádrem a uživatelským prostorem probíhá pomocí *eBPF* map, které mohou mít různé podoby. Například hašovací tabulky, pole nebo *ring-buffer*, který specificky povoluje kontinuální předávání dat z jádra do uživatelského prostoru.

eBPF bytecode může být zaveden ručně systémovým voláním **bpf()** nebo pomocí tzv. *loaderu*. *Loader* se postará o načtení programu, inicializaci, spuštění a předá program interpretu, který ho zkontroluje. Celkově je *eBPF* výkonný a spolehlivý instrumentační nástroj a širokými možnostmi použití.

[1] viz. https://go.dev/



Obrázek 4.1: Ilustrace *eBPF* architektury pro *libbpf* [12].

4.1.2 Vývojové nástroje

Psát *eBPF bytecode* není z mnoha důvodů doporučováno. Existuje tedy několik knihoven nad *eBPF*, pomocí kterých nemusíme *eBPF bytecode* přímo psát a zpříjemňují tak vývoj *eBPF* aplikací. Mezi nejznámější patří *bcc* 4.1.2, *bpftrace* 4.1.2 a *libbpf* 4.1.2. Knihovny poskytují lepší a udržitelnější prostředí pro psaní *eBPF* programů [25].

BCC

BCC (BPF Compiler Collection) je nástrojová sada, která zjednodušuje psaní *eBPF* programů pomocí instrumentace jádra v jazyce C (včetně *obálky* kolem LLVM pro C) a nabízí také rozhraní v jazycích Python a Lua. *BCC* je nejvíce využíváno na psaní *eBPF* programů v jazyce Python [14]. Tento přístup je ale od roku 2020 považován za zastaralý a je doporučováno využívat *libbpf* knihovnu pro C nebo *bpftrace* [23].

bpftrace

bpftrace je vysokoúrovňový jazyk pro *eBPF*, který je dostupný v Linuxových jádrech (od verze 4.x). Využívá *BCC* pro interakci s Linux *eBPF* systémem. Je ideální na psaní jednořádkových příkazů a krátkých skriptů. Inspirací mu byl *awk*, *C* a předchozí instrumentační nástroje *Dtrace* a *SystemTap* [36].

libbpf

libbpf je knihovna založená na jazyce C, která obsahuje modul přijímající zkompilované objektové soubory *eBPF*, které připraví a načte do jádra. Její součástí je i *API* pro interakci s Linux *eBPF* systémem a *API* pro pomocné funkce. Stará se o načítání a ověřování *eBPF* programů, díky čemuž dovoluje vývojářům soustředit se pouze na korektnost programu [5, 33].

Životní cyklus každého *eBPF* programu má 4 fáze, které *libbpf* definuje následovně [32]:

1. Zahajující fáze — načte *eBPF* objektový soubor do paměti a rezervuje paměť pro všechny *eBPF* programy, mapy a globální proměnné. Před načítací fází je stále možné provést různé úpravy, jako například nastavení počátečních hodnot globálních proměnných nebo nastavení typu *eBPF* programů.
2. Načítací fáze — vytvoření *eBPF* map, kontrola programů a jejich načtení do jádra. Programy však nejsou ještě spuštěny.
3. Připojovací fáze — spustí *eBPF* programy jejich připojením na definované háčky.
4. Uklízecí fáze — *eBPF* programy jsou odpojeny a odstraněny z jádra. *eBPF* mapy jsou zrušeny a všechna paměť využívaná programem je uvolněna.

bpftool

bpftool je nástroj pro inspekci a manipulaci eBPF programů a map. *libbpf* využívá nástroj *bpftool* pro generování *skeleton header file* (*.shel.h*) z *eBPF* objektového souboru. Vygenerovaný soubor obsahuje přizpůsobené funkce korespondující k fázi životního cyklu *eBPF* programu, které je spustí. Využívání kódu ve vygenerované *skeleton header file* je doporučený přístup práce s *eBPF* programy [2, 6].

4.1.3 BPF CO-RE

BPF CO-RE (BPF Compile Once-Run Everywhere) je přístup k vytváření *eBPF* programů, který řeší problémy přenositelnosti *eBPF* programů. Tento problém leží v tom, že *eBPF* program je program napsaný uživatelem a je načten do jádra. Tento *eBPF* program poté však nemá kontrolu nad tím, jakým způsobem samotné jádro pracuje. Musí si vystačit s tím, co jim každé nezávisle vyvinuté a přeložené jádro poskytne. Navíc se typy a datové struktury neustále mění. Různé verze jádra mohou mít přeuspořádané položky struktur nebo je mít dokonce přesunuté do struktury jiné. Položky mohou být přejmenovány, přesunuty nebo dokonce odstraněny.

BCC tento problém řeší vložením zdrojového kódu *eBPF* programu v C do programu v uživatelském prostoru v podobě řetězce. Když je program nakonec spuštěn na cílovém stroji, je zdrojový kód znovu přeložen. Tento překlad využívá *Clang/LLVM*, které jsou vložené v *BCC*, a hlavičky jádra. Programy vytvořené pomocí *BCC* tedy pro jejich spuštění vyžadují, aby bylo *BCC* nainstalováno na cílovém stroji. Takto přeložené *eBPF* programy jsou ale kvůli problémům zmíněným výše nestabilní. Hlavičkové soubory jádra dokonce na cílovém stroji nemusí být přístupné, což vede k nemožnosti program spustit.

BPF CO-RE řeší problém s kompatibilitou využíváním vícero komponent spolu s opatrnou integrací a kooperací jejich funkcionalit. Jedná se o tyto komponenty: *BTF (BPF Type Format)*, překladač *Clang* a knihovnou pro načítání *eBPF* programů *libbpf*.

BTF

BTF je formát metadat, který obsahuje ladící informace o *eBPF* v daném jádru [1]. Byl vytvořen jako alternativa k formátu *DWARF*. Umožňuje až stonásobně větší redukci velikosti a přesto zvládá popsat všechny typové informace o programech v jazyce C. Díky jeho kompaktnosti je *BTF* v jádrech standardně uloženo v cestě /SYS/KERNEL/BTF/VMLINUX a může být využito nástrojem *bpftool* pro generaci hlavičkového souboru jazyka C.

Clang

Překladač *Clang* byl pro podporu *BPF CO-RE* rozšířen o pár funkcionalit. Je díky němu možné objevit a vypořádat se s výše zmíněnými problémy vycházející ze změn položek struktur obsažených v jádru. Zachycené změny jsou poté předány *loaderu eBPF* programu, který provede potřebné změny.

libbpf

Všechna předem získaná data (*BTF* jádra a *Clang* změny) bere *libbpf* v potaz. Před načítací fází *eBPF* programu zkontroluje *libbpf* objektový soubor *eBPF* a pokud je podle získaných dat potřeba něco upravit, tak provede potřebné změny pro dané jádro. Tyto kontroly a případné změny se dějí automaticky.

4.1.4 Alternativní řešení

V předchozí sekci jsme si představili instrumentační nástroj *eBPF*, existují ale i jiné alternativní instrumentační nástroje, které si v následující sekci popíšeme.

SystemTap

Jedná se o obecně použitelný instrumentační nástroj pro jádro operačního systému Linux nebo uživatelské programy. *SystemTap* instrumentuje kód načtením vlastního modulu do jádra operačního systému Linux, na rozdíl od *eBPF*, které je virtuální strojem v jádře. Tento přístup znamená, že je potřeba implementovat mimo samotný program i bezpečnostní kontroly, které u *eBPF* potřeba nejsou. Moduly vytváří *SystemTap* automaticky z uživatelem dodaných skriptů. Tyto skripty jsou překládány do jazyka C a načteny do jádra jako moduly jádra [22, 37].

PIN

PIN je nástroj pro dynamickou instrumentaci vyvíjený společností Intel. Je většinou využíván pro instrumentaci uživatelských programů bez potřeby nového překladu sledovaného programu. *PIN* instrumentuje kód až na úrovni instrukcí, funkcí nebo základních bloků pomocí *JIT* překladače. Zajišťuje udržení původního chování instrumentovaného programu během profilování. *PIN* pracuje o úroveň víš nad operačním systémem, dokáže proto instrumentovat pouze kód z uživatelského prostředí. Podporuje také pouze limitovaný počet instrukčních sad — *IA-32*, *x86-64* a *MIC* [30, 7].

4.2 Profilovací nástroje

V této sekci si představíme nejen již existující *profilery* pro programy v jazyce *Go*. Nejdříve si ukážeme oficiální balíčky *pprof* a *trace* v sekcích 4.2.1 a 4.2.2. Poté si ukážeme jiné varianty profilovacích nástrojů.

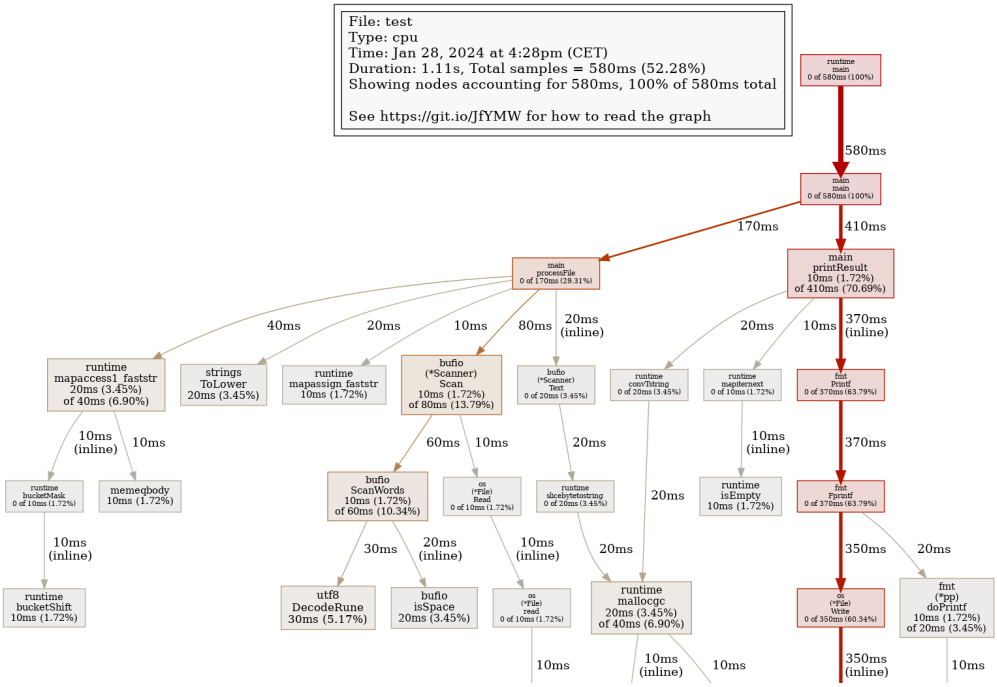
4.2.1 Balíček pprof

Běhové prostředí (tzv. *runtime*) jazyka *Go* poskytuje profilovací data ve formátu, které očekává vizualizační nástroj *pprof* použitý na vizualizaci Obrázků 4.2 a 4.5 [8]. Profilovací

data mohou být získána během testování skrze *go test* nebo koncové body dostupné z *net/http/pprof* balíčku, což vyžaduje úpravu zdrojového kódu a nový překlad programu [3, 9].

Předem definované profily poskytnuté v *runtime/pprof* balíčku jsou:

- *cpu* - Profil *cpu* určuje, kde program tráví čas aktivním využíváním cyklů procesoru. Neměří čas strávený ve spánku nebo čekáním.
- *heap* - Profil haldy (tzv. *heap*) zobrazuje vzorky alokace paměti. Používá se k monitorování aktuálního a historického využití paměti.
- *threadcreate* - Profil vytváření vláken zobrazuje části programu, které vedou k vytváření nových vláken operačního systému.
- *goroutine*[2] - Profil *goroutine* zobrazuje zásobníky aktuálních *goroutin*.
- *block* - Profil blokování ukazuje, kde *goroutiny* blokují čekáním na synchronizační prvky.
- *mutex* - Profil *mutex* zobrazuje obsazení zámků.



Obrázek 4.2: Část vizualizace *cpu* profilu naměřeného balíčkem *runtime/pprof* pomocí *pprof* vizualizačního nástroje. Jedná se o vizualizaci běhu programu jako stromu volání. Jiný typ zobrazení *cpu* profilu je na Obrázku 4.5.

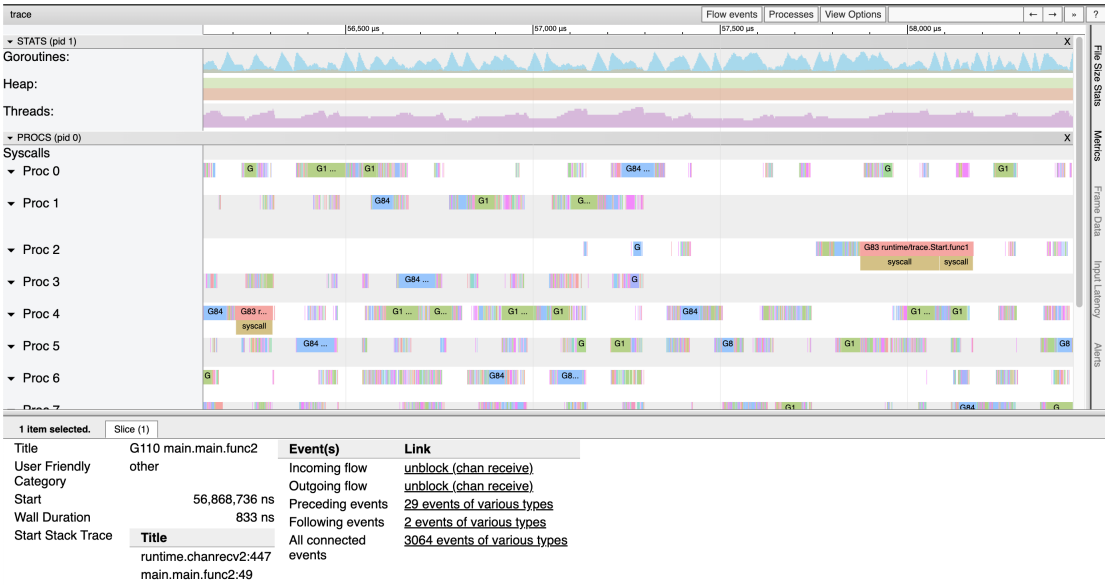
[2]Vlákno spravováno běhovým prostředím jazyka *Go*

4.2.2 Balíček trace

Go poskytuje balíček *trace* jako minimální nástroj pro trasování na úrovní *goroutin* v *Go* a nabízí minimální knihovnu pro instrumentaci s jednoduchým nástrojovým panelem. *Go* také poskytuje trasovač pro vytváření tras událostí běhového prostředí v rámci intervalu (tzv. *execution tracer*) a vizualizaci, která naměřené profily zobrazuje (viz Obrázek 4.3) [3, 10].

Distribuované trasování umožňuje:

- Instrumentovat a analyzovat dobu odezvy aplikací.
- Měřit vzdálené volání procedur (tzv. *RPC*) během požadavku uživatele a najít problémy viditelné pouze v produkčním prostředí.
- Najít výkonnostní problémy a jejich řešení. Mnoho takovýchto úzkých hrdel nelze identifikovat bez informací bez trasy.



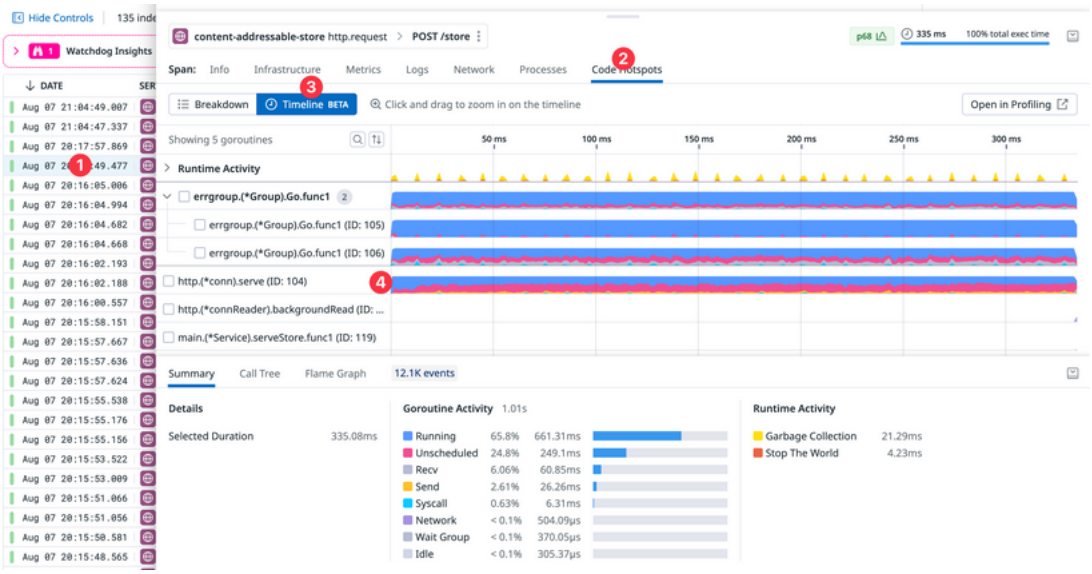
Obrázek 4.3: Vizualizace dat naměřených pomocí balíčku *trace* pomocí jejího vizualizačního nástroje. Zobrazují informace o *goroutinách* a různých procesech, které během měření běžely (např. systémové volání) na časové ose. Zobrazená data se dají přiblížit a zobrazit jejich bližší informace. Obrázek byl přebrán z článku [28].

Použití balíčku *trace* vyžaduje úpravu zdrojového kódu a nový překlad programu. Navíc *Go* nenabízí možnost automatického zachycení volání funkce a vytvoření trasy, uživatel musí manuálně instrumentovat požadované úseky. Vytváření tras má velkou režii a špatnou škálovatelnost, protože trasy mohou být moc velké a nelze je poté zpracovat. Problémy velké režie a špatné škálovatelnosti jsou vyřešeny v nejnovějších verzích *Go* [28].

4.2.3 Datadog

Datadog je nástroj pro monitorování a analýzu, který pomáhá udržovat plynulý chod aplikací a služeb. Nabízí monitorování serverů, databází a mnoha dalších nástrojů v reálném

čase. Poskytuje přehled o výkonu aplikací a umožňuje detekovat problémy předtím, než ovlivní uživatele. *Datadog* sbírá, vyhledává a analyzuje stopy napříč distribuovanými architekturami, poskytující poznatky o výkonu [15]. Příklad vizualizace pomocí tohoto nástroje můžeme vidět na Obrázku 4.4

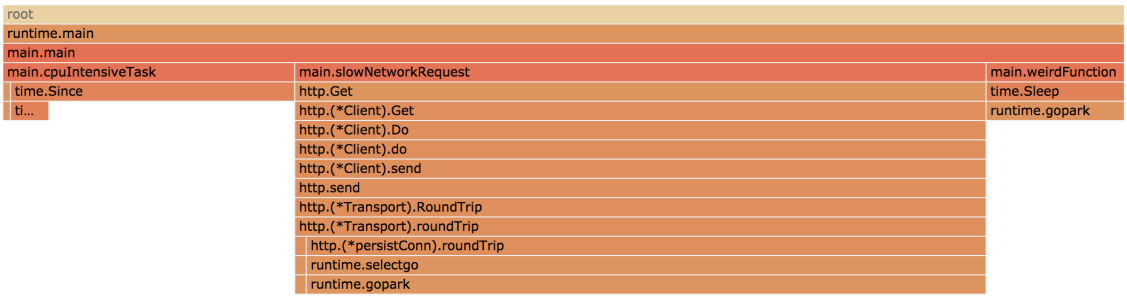


Obrázek 4.4: Ukázka kontinuálního profilovacího nástroje od společnosti *Datadog*. Obrázek obsahuje zobrazení životnosti všech *goroutin* zapojených ve vybraném požadavku (1) na časové ose (2, 3). Ve výchozí úrovni přiblížení jsou data agregovány do barevných schémat (4). Tato funkcionalita je momentálně ve verzi *beta*. Obrázek byl přebrán z článku, který tuto funkcionalitu představuje [19].

Nabízí kontinuální profilování programů v jazyce *Go* skrze jejich trasovací knihovny nebo využití jiných knihoven. Mimo balíčky, které obsahují instrumentaci vybraných knihoven, je možné vytvářet vlastní úseky pro instrumentaci. Vyžaduje *Go* verzi 1.19 nebo pozdější pro profilování a verzi 1.18 nebo pozdější pro trasování a úpravu zdrojového kódu a nový překlad programu [4, 11].

4.2.4 fgprof

Fgprof je vzorkovací *profiler*, který měří čas *wall clock*. Je implementován jako *goroutina* na pozadí, která je aktivována 99x za sekundu a získá list všech *goroutin* bez ohledu na jejich plánovací status. Režie tohoto nástroje se zvětšuje s počtem *goroutin*. Je kompatibilní s vizualizačním nástrojem *pprof* (viz Obrázek 4.5) a podporuje i formát dat pro použití vizualizace *FlameGraph* od Brendana Gregga [24]. Vyžaduje úpravu zdrojového kódu a nový překlad programu. Je doporučeno *fgprof* používat až od verze 1.17 jazyka *Go* [18].



Obrázek 4.5: Vizualizace výsledků naměřených nástrojem *fgprof*. Vizualizováno pomocí vizualizačního nástroje *pprof*. Obrázek byl přebrán z dokumentace nástroje [18].

Kapitola 5

Analýza požadavků

V této kapitole si představíme a popíšeme požadavky na výsledný profilovací nástroj. Tyto požadavky musí být brány v potaz při návrhu a implementaci jednotlivých částí nástroje. Některé požadavky však nemusí být plně splněny nebo se mohou dokonce vylučovat.

5.1 Analýza požadavků

Cílem práce je vytvořit profilovací nástroj aplikovatelný na programy napsané v jazyce *Go* a integrovat nástroj do systému *Perun* (Kapitola 3). Jelikož je *Perun* implementován v programovacím jazyce Python, bude i profilovací nástroj napsán v jazyce Python. Samotné profilování bude využívat technologii *eBPF* (Sekce 4.1) a jeho font-end *libbpf* (Sekce 4.1.2), který je využíván převážně v jazyce C.

Čas *wall clock*. Vzhledem k absenci profilovacích nástrojů měřících čas *wall clock* (viz Sekce 2.4) programů v jazyce *Go* jsme se rozhodli takovýto nástroj implementovat. Pro zajištění co nejmenší režie jsme se rozhodli využít technologii *eBPF*.

Kontext. Bez kontextu o běhu funkcí by naměřená data poskytovala minimální informační hodnotu. Uživatel by totiž nevěděl, odkud byly dané funkce volány. Náš profilovací nástroj tedy musí poskytovat kontext v podobě *Tras* volání funkcí. Tyto trasy bude rekonstruovat při zpracovávání naměřených dat.

Vizualizace. Naměřené profily musíme uživateli umožnit nějak interpretovat. Pro tento účel vytvoříme vizualizační modul obsahující vizualizaci profilu jako typ grafu volání funkcí. Tento nástroj by měl pomoci s porozuměním a interpretaci naměřených profilů.

5.1.1 Funkcionální požadavky

Každý použitelný profilovací nástroj by měl splňovat spoustu kritérií. Vytvořili jsme tedy seznam kritérií pro náš nástroj, které nám pomohou při návrhu a implementaci.

- **Perun integrace** — Profilovací nástroj je plně integrován do systému *Perun* a rozšiřuje jeho funkcionalitu jako nový kolektor (viz Obrázek 3.2).
- **eBPF technologie** — Profilovací nástroj využívá technologii *eBPF* pro implementaci sběru dat.

- **Malá režie** — Instrumentace sledovaného programu by měla způsobit jeho co nejmenší zpomalení.
- **Přesnost dat** — Profilovací nástroj by měl sbírat přesná data.
- **Množství dat** — Množství profilovacích dat by mělo být co nejmenší bez snížení jejich přesnosti.
- **Rychlost zpracování dat** — Čas zpracování nasbíraných dat by měl být co nejnižší pro zajištění příjemné uživatelské zkušenosti.
- **Počet volání a čas funkcí** — Profilovací nástroj bude získávat *wall clock* čas sledovaných funkcí a vést si statistiku o počtu volání.
- **Trasa funkce** — Profilovací nástroj bude nabízet kontext v podobě *Tras* volání funkce.
- **Vizualizace profilu** — Profilovací nástroj bude nabízet způsob vizualizace vygenerovaného profilu pro jeho analýzu.

5.1.2 Nefunkcionální požadavky

Mimo funkcionální požadavky musí každý software splňovat i jisté požadavky pro zajištění kvality a co nejlepší uživatelské zkušenosti.

- **Jednoduchost použití** — Proces použití kolektoru a vizualizace dat by měl být co nejlehčí. Oba nástroje mohou být parametrizovány, ale musí obsahovat rozumné výchozí hodnoty.
- **Minimální počet závislostí** — Kolektor by měl využívat co nejmenší počet povinných závislostí pro jednoduché využití i novým uživatelem.
- **Velikost dat** — Výsledná profilovací data by měla zabírat co nejméně prostoru na disku.
- **Srozumitelnost profilu** — Výstup kolektoru by měl obsahovat srozumitelné informace.
- **Srozumitelnost vizualizace** — Nástroj pro vizualizaci naměřených dat by měl být co nejvíce srozumitelný.
- **Integrita výsledků** — Nástroj pro vizualizaci by měl mít stejný výsledek pro daný profil bez ohledu na systém, na kterém je spouštěn.

Kapitola 6

Návrh a Implementace

Tato kapitola se bude věnovat implementaci profilovacího nástroje *gotrace* pro programy v jazyce *Go*, který využívá technologii *eBPF* popsanou v Kapitole 4.1. Začneme implementací nástroje *gotrace* (Kapitola 6.1), popíšeme jeho jednotlivé části a probereme omezení (Kapitola 6.2) vytvořeného nástroje. Na konci kapitoly představíme modul, který implementuje vizualizaci naměřených profilů.

6.1 Profiler jazyka Go

V této sekci si představíme všechny části nástroje *gotrace*, který měří *wall clock* čas všech nalezených funkcí z vybraných balíčků. Je implementován podle architektury *Perun* kolektorů (viz Sekce 3.3) a skládá se tedy z částí *before*, *collect* a *after*. *Before* zajišťuje nalezení symbolů funkcí ve sledovaném programu v jazyce *Go*, *collect* se stará o samotnou instrumentaci a sběr dat. V části *after* se poté nasbíraná data zpracují a vytvoří se profil ve formátu *JSON*, čímž se zajistí požadavek *Velikost dat*. Úvodní prototyp *gotrace* vycházel z *Perun* kolektoru *ktrace*, který využívá *eBPF* na profilování jádra systému.

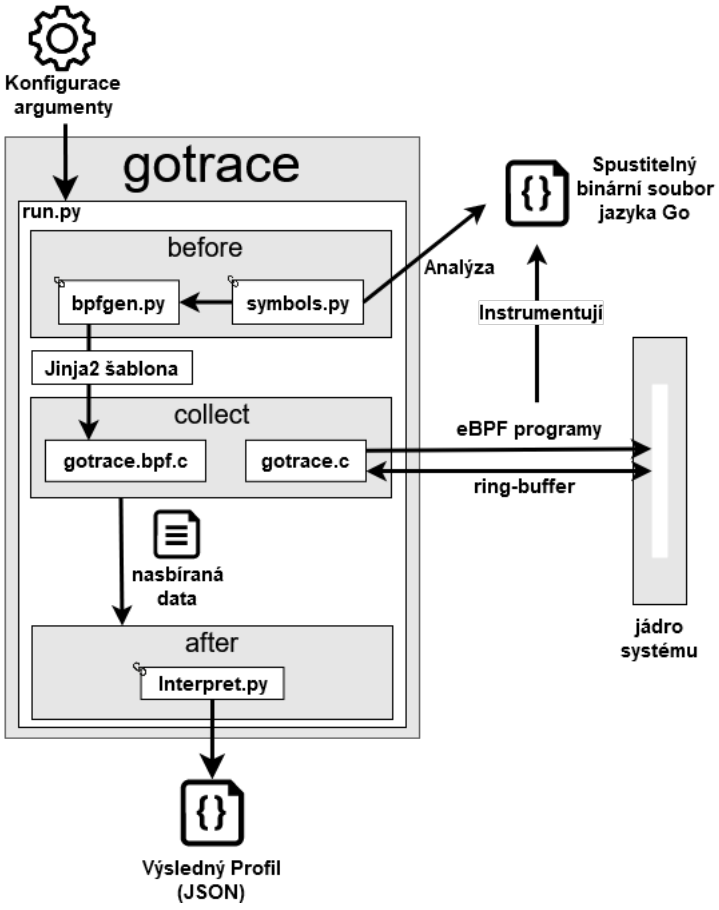
Pro implementaci *gotrace* budeme využívat dva programovací jazyky, a to Python a C. Python byl vybrán jako hlavní jazyk implementace, protože je v něm implementován nástroj *Perun* (viz. Kapitola 3). Tento výběr nám proto pomůže se splněním požadavku *Perun Integrace*. Jazyk C byl zvolen jako další jazyk, a to pro práci s technologií *eBPF*. Pro instrumentaci totiž *gotrace* využívá vývojový nástroj *libbpf* (viz Sekce 4.1.2), který je v jazyce C implementován. Existují sice obálky nad *libbpf* v jazyce Rust a dokonce i v jazyce *Go*, avšak pro dodržení požadavku *Minimální počet závislostí* jsme se rozhodli o využití výchozího jazyka C.

Granularita *gotrace* je na úrovni funkcí, sleduje totiž primárně volání a návraty z funkcí. Dokáže navíc měřit čas strávený modifikací zásobníku volání jazyka *Go* instrumentací specifické instrukce v daných funkcí.

Implementace profilovacího nástroje *gotrace* je rozdělena do více souborů:

- `run.py` implementuje tři hlavní části nástroje – *before*, *collect* a *after*;
- `symbols.py` implementuje vyhledávání funkcí, je využíván částí *before*;
- `bpfgen.py` implementuje generování *eBPF* programů pro každou nalezenou funkci, je využíván částí *before*;

- `bpf_template_go_uprobes.c` obsahuje *jinja2*[1] šablonu *eBPF* programů, je využíván souborem `bpfgen.py`;
- `interpret.py` implementuje zpracovávání naměřených dat a vytváření profilů, je využíván částí *after*;
- `gotrace.c` implementuje uživatelskou část sběru dat a způsob instrumentace;
- `gotrace.h` definuje datové typy a struktury;
- `gotrace.bpf.c` implementuje *eBPF* programy pro každou nalezenou funkci, je generován částí *before*.



Obrázek 6.1: Vizualizace architektury nástroje *gotrace*.

6.1.1 Argumenty příkazové řádky

Nástroj *gotrace* je volán přes modul *collect* systému *Perun* a nabízí přizpůsobení chování přes argumenty. Na správu argumentů příkazové řádky je použitý balíček *click*[2], který je využíván napříč celým systémem *Perun*. Následuje výčet argumentů nástroje *gotrace*:

[1] viz. https://jinja.palletsprojects.com/en/latest/
[2] viz. https://click.palletsprojects.com/en/latest/

- `packages` — Volitelný argument bez přepínače, upřesňuje z jakých balíčků má *gotrace* vyhledávat funkce. Výchozí hodnota je balíček `main`.
- `--bpfring-size (-s)` — Argument určující velikost *ring-bufferu* používaného na komunikaci s *eBPF*. *Ring-buffer* se používá ke kontinuálnímu předávání dat.
- `--save-intermediate-to-csv (-c)` — Přepínač, který uloží zpracovaná data před transformací na profil do formátu *csv*. Výchozí hodnota je `False`.
- `--verbose (-v)` — Přepínač povolující výpis všech funkcí, které byly nalezeny a budou nástrojem *gotrace* profilovány. Vypíše také zachycený standardní výstup profilovaného programu. Tato možnost může snížit čitelnost výstupu systému *Perun*. Výchozí hodnota je `False`.
- `--get-overhead (-o)` — Přepínač, který zapne výpočet režie nástroje *gotrace* na profilovaném programu. Profilovaný program bude spuštěn dvakrát, s a bez nástroje *gotrace*. Je použitelný pouze společně s přepínačem `-v`.

6.1.2 Implementační problémy

Při vytváření *gotrace* jsme narazili na několik problémů použití technologie *eBPF* k instrumentaci programů v jazyce *Go*. V této sekci si problémy představíme, navrhneme řešení, a pokud jich je více, zdůvodníme výběr konkrétního řešení.

Sondy na výstupech funkcí. První problém, na který jsme při implementaci *gotrace* narazili je způsob, jakým *eBPF* vkládá sondy (tzv. *probes*) pro sledování návratů z funkcí [13]. Nejdříve si tento způsob vysvětlíme a poté vysvětlíme, proč se za ním skrývá implementační problém.

Klasický způsob použití sond na funkce z uživatelského prostoru funguje tak, že se vloží uživatelská sonda (tzv. *uprobe*) na adresu začátku funkce. Pokud chceme vložit sondu i na konec funkce (tzv. *uretprobe*), vytvoří se uživatelská sonda, která při její aktivaci modifikuje adresu návratu z funkce na zásobníku volání funkcí. Jsou-li pak sondy aktivovány, vloží se instrukce pozastavení *int3* a provede se instrumentační kód *eBPF* programu, který je s danou sondou asociován [16].

Tento způsob ale nefunguje pro programy v jazyce *Go* – ty mají totiž zásobník dynamické velikosti, který se za běhu programu různě mění (počáteční velikost zásobníku je 2KB). Důsledkem použití zmíněného principu *uretprobe* na programy v jazyce *Go* je jejich zřícení. Funkce se totiž vrací na, díky neočekávané změně na zásobníku volání, neznámou adresu. Klasický způsob použití sond na sledování návratů z funkcí je tedy pro naši potřebu nemožné spolehlivě využívat.

Jako řešení navrhujeme nevyužívat uživatelské sondy, které modifikují adresu návratu z funkce na zásobníku a využívat místo nich normální uživatelské sondy na každém bodu návratu z funkce (instrukce *RET*). Každý tento bod návratu musíme najít a musíme počítat s tím, že bodů muže být více než jeden.

Parelelismus a vlákna. Další problém, na který jsme narazili se týká vysokého využití paralelismu v programech jazyka *Go*. Jestli-že sledovaný program tuto techniku programování využívá, je potom během zpracování nasbíraných dat nemožné spolehlivě rozpoznat dvojice volání a návratu z funkce. Nedokážeme tedy naměřit správné statistiky a nedodržujeme požadavek *Přesnost dat*. Řešení tohoto problému je inspirováno konferencí [29].

Jako řešení tedy navrhujeme využití informace o paralelním procesu navíc. U ostatních programovacích jazyků se pro tento účel používá identifikátor systémového vlákna (tzv. *thread-id*), na kterém daný proces běží. Bohužel, toto řešení nefunguje kvůli způsobu, jakým jazyk *Go* provádí paralelní programování. Využívá totiž *goroutin* místo systémových vláken, jak je běžné v jiných programovacích jazycích. *Goroutiny* jsou vlákna spravované běhovým prostředím jazyka *Go*, a mohou během svého života běžet na vícero systémových vláknech. Může se tedy stát, že paralelní proces začne na jednom systémovém vlákně a skončí na jiném.

Problém tedy zůstává stejný, stále během zpracování dat nezvládneme korektně vytvořit dvojice volání a návratu z funkcí.

Navrhujeme tedy místo identifikátorů systémového vlákna využívat identifikátory *goroutin* (tzv. *go-id*). Získávání tohoto identifikátoru však není tak jednoduché, jako u identifikátoru vlákna a musíme ho získat od běhového prostředí jazyka *Go*. Díky analýze běhového prostředí víme, že je *go-id* obsaženo ve struktuře *g*[3]. Pro získání více informací o této struktuře musíme analyzovat funkci, která ji vytváří – `runtime.setg`. Její podoba v *asembleru* vypadá takto:

```
0000000000046d4c0 <runtime.setg.abi0>:
    mov 0x8(%rsp),%rbx
    mov %rbx,%fs:0xfffffffffffffff8
    ret
```

Výpis 6.1.3: *Asembler* kód funkce <runtime.setg.abi0>.

Z tohoto kódu můžeme vypozorovat, že je struktura *g* obsažená v registru *FS*. K tomuto registru můžeme pomocí *eBPF* přistoupit funkcí `get_current_task()`, která vrátí strukturu obsahující *FS* registr. Z kódu vidíme, na jaké poloze v *FS* registru se struktura *g* nachází, ze které už jenom stačí *go-id* získat.

Opakující se vstupy do funkcí. Při vývoji nástroje jsme také narazili na občasné několikanásobné záznamy volání funkce, které pak přidávají neočekávané záznamy do nasbíraných dat. Museli jsme proto toto chovaní analyzovat a zjistili jsme, že se týká již zmíněného dynamického zásobníku jazyka *Go*. Přesněji toho, jakým způsobem a kdy se zásobník zvětšuje. Za účelem pochopení problému se tedy podíváme na *asembler* kód funkce `main.example` programu v jazyce *Go*, která přijímá jeden argument:

```
000000000067be80 <main.example>:
    cmp 0x10(%r14),%rsp
    jbe 67bed6 <main.example+0x56>
    push %rbp
    mov %rsp,%rbp

    ... function body

    mov %rax,0x8(%rsp)
```

[3] viz. https://go.dev/src/runtime/runtime2.go

```
call 46b800 <runtime.morestack_noctxt.abi0>
mov 0x8(%rsp),%rax
jmp 67be80 <main.example>
```

Výpis 6.1.2: *Asembler* kód příkladové funkce v jazyce *Go*.

Z kódu vidíme, že se na začátku funkce kontroluje velikost zásobníku, aby nedošlo k jeho přetečení. Pokud by k němu mohlo dojít, je zavolána funkce `morestack`, která velikost zásobníku zvětší. Poté je znovu zavolána příkladová funkce a je znovu zkontrolována velikost zásobníku, pokud je velikost zásobníku dostatečná, tak se funkce standardně provede. Můžeme si všimnout řádků před a po volání `morestack`, které slouží pro zachování původních argumentů funkce.

Problém spočívá v tom, že *eBPF* vkládá sondy na začátek funkce, tedy `<main.example>`. Dojde-li ke zvětšení zásobníku, znamená to opakované provedení instrumentačního kódu pro začátek dané funkce. Toto chování potom způsobuje již zmíněné problematické data vstupů funkce.

Jako řešení jsme nejdříve navrhli se s tímto problémem vypořádat během zpracování nasbíraných dat. Tento přístup jsme však zavrhli, protože by bylo nemožné toto chování odlišit od rekurzivního volání funkce.

Další řešení, které jsme navrhli, je instrumentace instrukce volání `morestack` pro každou sledovanou funkci, pokud toto chování obsahuje. Toto řešení jde bohužel proti požadavku *Množství dat*, jsme díky tomuto přístupu ale schopni rozlišit opravdové volání funkce od zvětšení zásobníku. Dává nám také možnost měřit novou metriku pro programy v jazyce *Go*, a to čas strávený zvětšováním zásobníku.

6.1.3 Část *before*

Část *before* zajišťuje vyhledání všech symbolů funkcí zadaného binárního souboru podle argumentu *packages* a dynamické generování *eBPF* programů pomocí *jinja2* šablony. Je implementována ve funkci `before()` souboru `run.py`. Vzhledem ke srozumitelnosti kódu je vyhledávání symbolů a generování rozděleno do vlastních souborů (`symbols.py` a `bpfgen.py`), které jsou poté do hlavního souboru `run.py` importovány.

Vyhledávání funkcí. Vyhledávání funkcí, adres návratových instrukcí a volání `morestack` je implementováno v jazyce Python pomocí balíčku *ELFFile*[4]. Pro hledání funkcí nás zajímá pouze *.symtab* sekce binárního souboru. Projdeme všechny symboly, které značí funkce a pokud jsou z balíčků definovaných v argumentu *packages*, tak si je uložíme. Uložíme si také adresy `morestack` funkcí (dále M_ADDR), aby jsme poté mohli určit, zda jsou volány.

Po vyhledání všech funkcí se musíme podívat na celá těla funkcí, abychom byli schopní najít všechny návraty z funkcí a zvětšování zásobníku. Pro každou funkci si tedy zaznamenáme rozdíl adres od začátku funkce pro každou RET a CALL M_ADDR instrukci. Pokud funkce neobsahuje `morestack`, je místo rozdílu adres od začátku funkce využito hodnoty *None*.

Pokud je aktivní přepínač `--verbose` jsou názvy nalezených funkcí vypsány. Bez ovlivnění přepínače `-v` je poté vypsán i počet nalezených funkcí.

[4] viz. https://pypi.org/project/elffile/

Generování souboru .bpf.c. Generování *eBPF* kódu pro každou funkci, její návraty a `morestack` volání je implementováno pomocí Python balíčku *Jinja2*. Každá funkce je tedy svázaná s minimálně dvěma až třemi *eBPF* programy. Argumenty pro *jinja2* jsou velikost *ring-bufferu*, cesta ke sledovanému binárnímu souboru a Python slovník obsahující všechny potřebné informace o každé funkci.

Nejdříve je vygenerována *eBPF* mapa typu *ring-buffer* a poté jsou generovány *eBPF* programy pro každou funkci podle následující zjednodušené šablony:

```
{% for func_name, (offsets, morestack) in symbols.items() %}
SEC("uprobe//{{ path }}:{{ func_name }}")
    ...
{% for offset in offsets %}
SEC("uprobe//{{ path }}:{{ func_name }}+{{ offset }}")
    ...
{% endfor %}
{% if morestack is not none %}
SEC("uprobe//{{ path }}:{{ func_name }}+{{ morestack }}")
    ...
{% else %}
    ... Nothing
{% endif %}
{% endfor %}
```

Výpis 6.1.3: Kostra *jinja2* šablony použité pro generaci *eBPF* programů pro každou funkci.

Po generování je vygenerovaný soubor ještě přeložen a zkontrolován pro jeho spuštění v *eBPF* prostoru.

6.1.4 Část *collect*

Část *collect* implementuje sběr dat a spouštění nástroje *gotrace* ve funkci `collect()` souboru `run.py`. Tato část je spuštěna až po ukončení části *before*. Nejdříve je uživatel vyzván k manuálnímu spuštění programu *gotrace.c* na cestě `perun/collect/gotrace/bpf_build` příkazem `sudo ./gotrace`. Tento proces totiž vyžaduje práva administrátora. Spuštění *gotrace.c* je kontrolováno každých pět vteřin a jakmile je spuštění uživatelem zaznamenáno, je sledovaný program spuštěn pro profilování. Po jeho dokončení je uživatel vyzván k ukončení nástroje *gotrace.c* zasláním ukončovacího signálu. Ukončení *gotrace.c* je kontrolováno každé dvě vteřiny.

Pokud jsou aktivní přepínače `--get-overhead` a `--verbose` je profilovaný program spuštěn dvakrát spolu s Linux utilitou *time*[5], pomocí které je zjištěn čas běhu programu. Poprvé je spuštěn s nástrojem *gotrace* a podruhé bez něj. Díky zjištěným časům potom můžeme vypočítat režii nástroje *gotrace* na profilovaném programu.

Sběr dat. Sběr dat je implementován v souboru `gotrace.c` a jeho hlavičkovém souboru `gotrace.h`. Po spuštění *gotrace* je nejdříve zvýšen jeho limit paměti a je vytvořena a načtena kostra *libbpf*. Všechny *eBPF* programy jsou poté připojeny na definované háčky

[5] viz https://man7.org/linux/man-pages/man1/time.1.html

pomocí automatického připojení. Následuje vytvoření souboru pro nasbíraná data a *ring-bufferu*, se kterým probíhá kontinuální komunikace pomocí volání `ring_buffer__poll(rb, timeout)`. Data jsou posílána ve formátu definovaném v *gotrace.h*:

```
struct record {
    uint32_t func;
    bool type;
    bool morestack;
    uint64_t goid;
    uint64_t ts;
};
```

Výpis 6.1.4: Struktura určující podobu dat, ve které se posílají data mezi *eBPF* a uživatelským prostorem *gotrace*.

Položka *func* obsahuje ID funkce, pomocí kterého je poté během zpracovávání identifikována. *type* určuje zda se jedná o vstup (0) nebo výstup (1) z funkce, v případě vstupu je využito ještě položky *morestack*, která určuje zda se jedná o záznam volání `morestack` (1) nebo opravdový vstup do funkce (0). Položka *ts* pak obsahuje samotnou časovou značku, pomocí které zjistíme *wall clock* čas funkce.

Profilovací data jsou postupně zapisována do souboru, uložena a předána pro jejich zpracování v části *after*. Jeden záznam uložených dat je v tomto formátu:

```
func_ID;TYPE;MORESTACK;GOID;TimeStamp\n
```

Výpis 6.1.4: Formát jednoho řádku profilovacích dat značící jednu událost.

6.1.5 Část *after*

V části *after* se nasbíraná data zpracovávají a transformují na profily. Část *after* je implementována funkcí `after()` souboru `run.py`. Funkce, které implementují zpracovávání a transformace dat jsou obsaženy v souboru `interpret.py` importovaném v souboru `run.py`.

Data jsou nejdříve zpracována ve funkci `parse_traces()`, kde jsou vedeny zásobníky pro každé využité *go-id*. V těchto zásobnících jsou poté záznamy volání a návratu funkce spárovány a jsou zaznamenány exkluzivní a inkluzivní časy (viz Sekce 2.4) každé funkce. V kontextu těchto zásobníků jsou také vytvářeny trasy každé funkce analýzou aktuálního stavu daného zásobníku. Záznamy `morestack` jsou vyfiltrovány a funkcím, při kterých k volání `morestack` došlo je přidán i čas strávený zvětšování zásobníku jazyka *Go*.

Data mají podobu podle třídy *FuncDataFlat* a jsou aktualizována abstraktní metodou `update()`. Tento přístup povoluje jednoduché přidání jiných typů profilů dat. Po tom, co jsou naměřená data zpracována, mohou být ještě před transformací na profil pomocí argumentu *-c* uložena jako *csv* soubor. Mimo tuto možnost je z dat vytvořen *Perun* profil funkcí.

Výsledný profil mimo profilovacích dat obsahuje i informace o způsobu volání nástroje a systému, který byl ke generování profilu využit.

6.2 Známá omezení

Využívání *eBPF* na programy v jazyce *Go* se ukázalo jako netriviální úkol. Existuje tedy několik známých omezení, které se nám v této práci nepodařilo ošetřit.

Proces překladu. Binární soubory, které *gotrace* zvládne profilovat musí obsahovat symboly. V opačném případě totiž *gotrace* nemůže spolehlivě vyhledat všechny funkce, které má správně sledovat. Navíc musí být binární soubory přeloženy pomocí příkazu `go build -gcflags -l`. Tento způsob překladu zaručí, že překladač nevyužije optimalizační techniku *inlining*.

Profilování rozsáhlých programů. *gotrace* není vhodný na profilování velkých projektů. Vytváří totiž minimálně dva nebo tři *eBPF* programy pro každou sledovanou funkci. U velkých projektů s velkým počtem funkcí se může stát, že systém nedovolí vytvoření tolika *eBPF* programů. Toto omezení by mohlo být možné řešit použitím jednoho *eBPF* programu pro všechny nalezené návraty z funkce.

Ruční spouštění gotrace. Spouštění *eBPF* části *gotrace.c* vyžaduje práva administrátora. Čeká proto *gotrace* na ruční spuštění a ukončení *eBPF* části. Tento přístup není uživatelsky přívětivý.

Čas vytváření eBPF programů. Vzhledem k předchozímu omezení musí *gotrace* vědět, že byla *eBPF* část *gotrace.c* spuštěna. Provádí to kontrolou jeho spuštění každých pět vteřin. Kontroluje ale pouze spuštění *eBPF* části a ne vytvoření a připojení všech *eBPF* programů, což při větších projektech může trvat delší dobu. Spuštění profilovaného programu před připojením všech *eBPF* programů vede ke špatným datům.

Neúplné trasy. Vytváření tras z kontextu volání sice odstraňuje režii jejich získávání ze zásobníku a zasílání z *eBPF* části nástroje. Vytváří ale problém neúplných tras při záznamech s novými *go-id*. Těmto funkcím totiž chybí kontext pro jejich vytvoření.

Operační systém. Technologie *eBPF* je momentálně přístupná pouze na operačním systému Linux. Není tedy možné *gotrace* využívat na strojích s jinými operačními systémy. Microsoft sice pracuje na implementaci *eBPF* pro operační systém Windows[6], ale tento nástroj je zatím nedokončený a nemůžeme tedy zajistit funkčnost *gotrace*.

6.3 Vizualizace

Tato sekce se zaměřuje na popis procesu vytváření vizualizace profilů naměřených pomocí nástroje *gotrace*. Vzhledem k požadavku *Perun integrace* pro nástroj *gotrace* by měla být vizualizace jeho profilů také integrována do nástroje *Perun*. Na její implementaci je tedy využit programovací jazyk Python, stejně jako u nástroje *gotrace*. Integrace do systému *Perun* je dosaženo implementací vizualizace jako *Perun* modulu *view* (viz. Sekce 3.2).

Na vizualizaci profilů nástroje *gotrace* jsme si vybrali tzv. *Sankey* diagram. Jedná se o grafické znázornění složení a časového průběhu stavu určité veličiny (funkcí) v určitém systému (program v jazyce *Go*). Tento typ vizualizace jsme si vybrali kvůli jeho výchozí

[6] viz. https://github.com/microsoft/ebpf-for-windows

podpoře časového průběhu, který můžeme využít pro zobrazení *wall clock* času jednotlivých funkcí. Dokáže také vizualizovat *Trasy* funkcí (viz. Sekce 2.4), přidává tak potřebný kontext pro identifikaci problematických funkcí a splňuje požadavek *Srozumitelnost vizualizace*.

V implementaci vizualizace profilů využíváme oblíbenou a často používanou knihovnu pro zpracovávání dat *Pandas*[7]. Nejdříve pomocí této knihovny transformujeme vizualizovaný profil na datový typ *DataFrame*. Tento *DataFrame* poté transformujeme pro lepší zpracování jednotlivých záznamů funkcí a vytvoříme záznam každé funkce daný třídou *SankeyRecord*.

K vytvoření *Sankey* diagramu musíme nejdříve vytvořit dvojice bodů s časovou hodnotou přechodu, v našem případě funkce a funkce, kterou předchozí funkce volá s *wall clock* časem jako hodnotou přechodu. Pro každou dvojici funkcí vytvoříme dva záznamy, každý s jiným typem času. Jeden záznam obsahuje čas exkluzivní a druhý čas inkluzivní (viz Sekce 2.4). Každý typ času nabízí jiný pohled na běh sledovaného programu a může pomoci k rychlému vyhledání případných problémů (viz Obrázky 6.2 a 6.3).

Díky použití *Sankey* diagramu je také možné vizualizovat *goroutiny* jako ostatní funkce. Běží totiž na novém *go-id* a chybí jim kontext ze zásobníku při zpracování nasbíraných dat pro generování trasy. U *goroutin* se to ale dá řešit pomocí způsobu jejich pojmenování. Například *goroutina* `main.main.func1` je první *goroutina* funkce *main* z balíčku *main*. Je tedy jasné, že je volána z funkce *main* balíčku *main* a je možné ji díky způsobu vytváření *Sankey* diagramu zařadit na své místo ve výsledné vizualizaci.

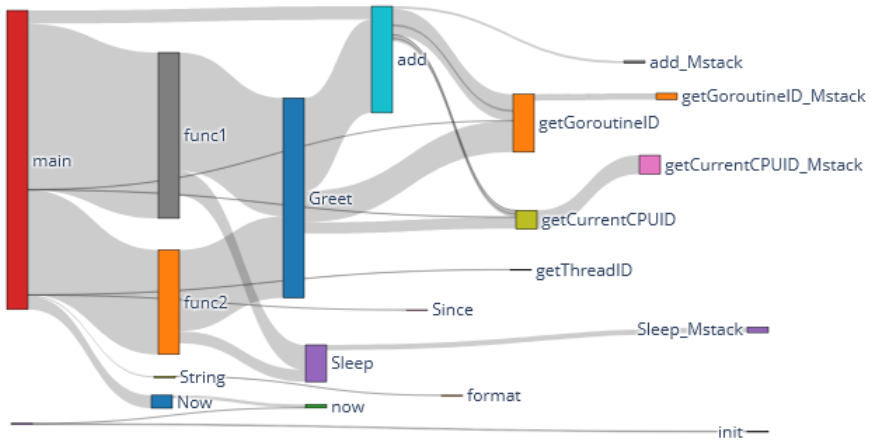
Vizualizační knihovny Důležitým krokem vytváření vizualizace je výběr správné vizualizační knihovny. Jelikož vizualizaci implementujeme v jazyce Python, budeme využívat knihovny, které jsou v tomto jazyce přítomné.

Jako první jsme si vybrali vizualizační knihovnu *Holoviews*[8], která na pozadí využívá knihovny *matplotlib* nebo *bokeh*. Kvůli vyšší podpoře interaktivity jsme si vybrali *Holoviews* v kombinaci s knihovnou *bokeh*. Toto řešení nám umožňovalo s celým grafem pohybovat a upravovat měřítko grafu, nepovolovalo nám však interakce s uzly představující funkce, a celkový vzhled grafu tak působil příliš staticky.

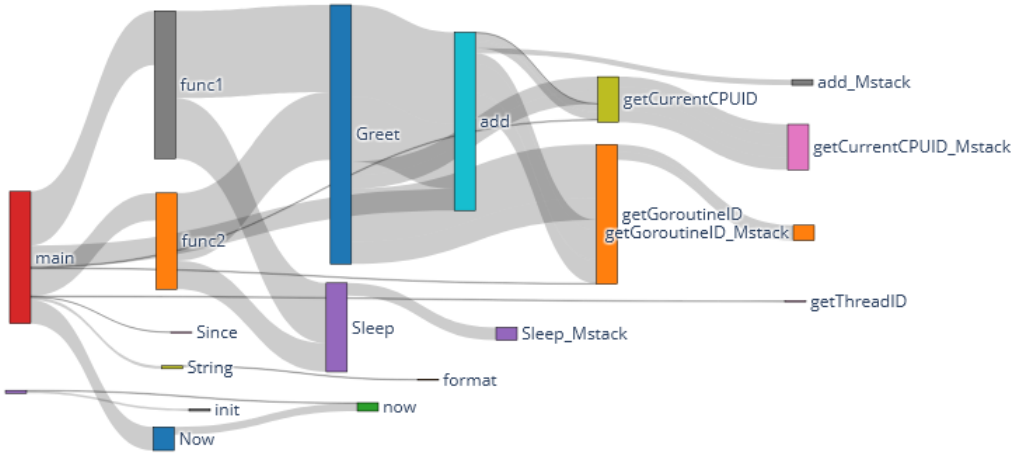
Rozhodli jsme se tedy pro jiné řešení, které nám snad dovolí více interaktivní způsob vizualizace profilu. Tohoto jsme chtěli dosáhnout použitím Python obálky knihovny *Plotly*[9] v kombinaci s knihovnou *Jinja2*. *Sankey* diagram vytvořený touto Python knihovnou, který jsme pomocí *jinja2* vložili do *HTML* souboru, nám dovoloval s jednotlivými uzly funkcí hýbat podle naší potřeby a vypadal celkově lépe. Nedovoluje nám však s celkovým grafem hýbat nebo upravovat měřítko grafu. Pokusili jsme se tedy tyto funkcionality do grafu skrz knihovnu *Plotly* přidat, bohužel tato knihovna tyto funkcionality nenabízí a naše vizualizace je tedy nebude podporovat. Toto může být omezení naší vizualizace při použití na větší projekty s velikým počtem funkcí.

Kvůli způsobu, jakým jazyk *Go* pojmenovává funkce v binárních souborech jsou výsledné vizualizace nepřehledné. Obsahuje-li program např. funkci `add` z balíčku `github.com/u/p` je funkce v binárním souboru pojmenována `github.com/u/p/f.add`. Pokud projekt obsahuje více takovýchto funkcí, tak jsou výsledné vizualizace velmi nepřehledné. Rozhodli jsme se proto zobrazovat pouze názvy funkcí bez jejich původu. Plné názvy funkcí si ale stále můžeme zobrazit najetím na uzel funkce, o které chceme získat více informací.

[7] viz. https://pandas.pydata.org/docs/
[8] viz. https://holoviews.org/
[9] viz. https://plotly.com/



Obrázek 6.2: Vizualizace programu *Greet* v jazyce *Go* s využitím inkluzivního času funkcí.



Obrázek 6.3: Vizualizace programu *Greet* v jazyce *Go* s využitím exkluzivního času funkcí.

Výsledný *HTML* soubor obsahující vizualizaci profilu je uložen na lokaci, ze které byla vizualizace volána, a jeho název je automaticky generovaný. Název a lokaci výstupního souboru je možné upravit pomocí argumentu `-output-file` (`-o`).

Tím, že je modul do systému *Perun* integrován jako `view` modul, tak získává jeho způsob vyhledávání profilu. *Perun* se tedy přidaný profil pokusí vyhledat těmito způsoby:

1. Pokud je profil ve formátu *i@i* (tzv. *index tag*) nebo *i@p* (tzv. *pending tag*), tak je vybrán příslušný profil.
2. Profil je vyhledán v profilech asociovaných s momentální verzí projektu.
3. Profil je vyhledán v adresáři *.perun/jobs*.
4. *Perun* projde celý adresář, ze kterého je volán a každý nalezený profil je předložen uživateli k potvrzení.

Kapitola 7

Experimenty

Tato kapitola se bude věnovat demonstraci použití nástroje *gotrace* z Kapitoly 6 pomocí několika experimentů. Cílem experimentů je ukázat a ověřit schopnosti vytvořeného nástroje. Jako první si v Sekci 7.1 ukážeme způsob provedení experimentů a poté budou následovat jednotlivé experimenty.

7.1 Způsob provedení experimentů

Tato sekce vysvětluje způsob provedení všech experimentů. Poté je popsán systém, na kterém byly experimenty vykonány.

Metodologie. Všechny experimenty byly vykonány s následujícími pravidly:

1. Musí být provedeno minimálně pět měření a první měření musí být zahozeno za účelem stability výsledků.
2. Pro hodnoty časových metrik (např. doba běhu programu) musí být vybrána mediánová hodnota. Čas běhu profilovaného programu je počítán pomocí utility *time* systému Linux.

Specifikace systému. Experimenty byly vykonány na virtuálním stroji s následujícími specifikacemi.

Tabulka 7.1: Specifikace virtuálního stroje.

Operační systém	Linux Ubuntu 22.04.3 LTS
Jádro	6.5.0-28-generic
Architektura	x86_64
Počet jader CPU	4
Frekvence CPU	3400.05 MHz
Velikost paměti RAM	7.7 GiB
Verze jazyka *Go*	1.22.0
Verze nástroje *bpftool*	@ 3e6f814
Verze nástroje *libbpf*	@ 2778cbc

7.2 Experiment 1: Režie

Tato sekce popisuje první experiment, který je zaměřený na zjištění režie *gotrace* při profilování programů. V tabulkách 7.2 a 7.3 je ukázán čas běhu profilovaného programu s *gotrace*, čas běhu profilovaného programu bez *gotrace* a výsledná režie.

Tento experiment byl prováděn s pomocí argumentu `-get-overhead` nástroje *gotrace*. Použití této vlajky znamená spuštění profilovaného programu s využitím *time* utility systému Linux, která měří čas běhu spuštěného příkazu. Je tak provedeno dvakrát: jednou s *gotrace* a jednou bez něj. Z naměřených časů je poté vypočítána režie daného volání *gotrace*. Tento způsob získávání režie je použit, aby byly oba běhy profilovaného programu provedeny v co nejpodobnějších podmínkách.

Použité programy. Experiment byl vykonáván na dvou programech. První program *greet* je jednoduchý program, který jsme během implementace nástroje *gotrace* využívali pro kontrolu funkčnosti. Další program *oto* je ukázková implementace použití knihovny *oto*[1], která přehraje programem definovaný zvuk. *Oto* je nízkoúrovňová knihovna v jazyce *Go* pro přehrávání zvuku.

Program	**Čas s *gotrace* [s]**	**Čas bez *gotrace* [s]**	**Režie**
greet	0.31	0.085	3.65x
oto	5.515	5.065	1.09x

Tabulka 7.2: Tabulka ukazující časy běhu profilovaného programu s a bez nástroje *gotrace*, a výslednou režii.

Výsledky experimentů ověřily naše domněnky o nízké režii profilovaného programu při použití technologie *eBPF*. Mediánová hodnota režie u měření programu *oto* je však ale zvláštně nízká. Po analýze programu jsme byli schopni zjistit, proč tomu tak je. Program totiž napříč více *goroutin* čeká na přehrání zvuku dohromady pět vteřin. Pokud tento čas strávený čekáním z výsledků odstraníme, získáme novou režii, která již odpovídá režii očekávané.

Program	**Čas s *gotrace* [s]**	**Čas bez *gotrace* [s]**	**Režie**
oto	5.515	5.065	1.09x
oto-bez-čekání	0.535	0.06	8.92x

Tabulka 7.3: Tabulka ukazující časy běhu profilovaného programu *oto* s a bez čekání na přehrání zvuku, a výslednou režii.

Při vykonávání experimentu jsme při zvýšení počtu sledovaných funkcí pozorovali zvýšení času, který *gotrace* potřebuje pro vytvoření, připojení a uklizení *eBPF* programů.

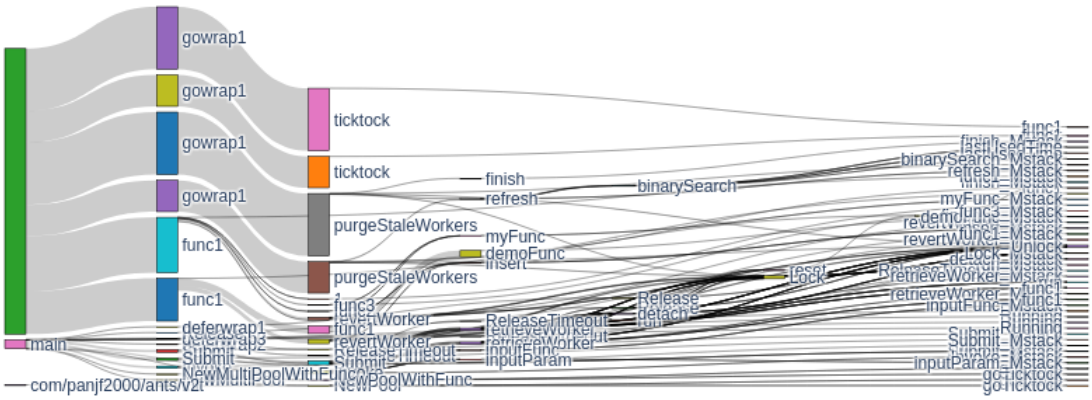
```
perun collect -c './greet' gotrace -v -o main
perun collect -c './oto' gotrace -v -o main github.com/ebitengine/oto/v3
```

Výpis 7.2: Příkazy, které byly využity pro profilování programů a měření režie. Při profilování funkcí z importovaného modulu je potřeba zadat celé jméno modulu.

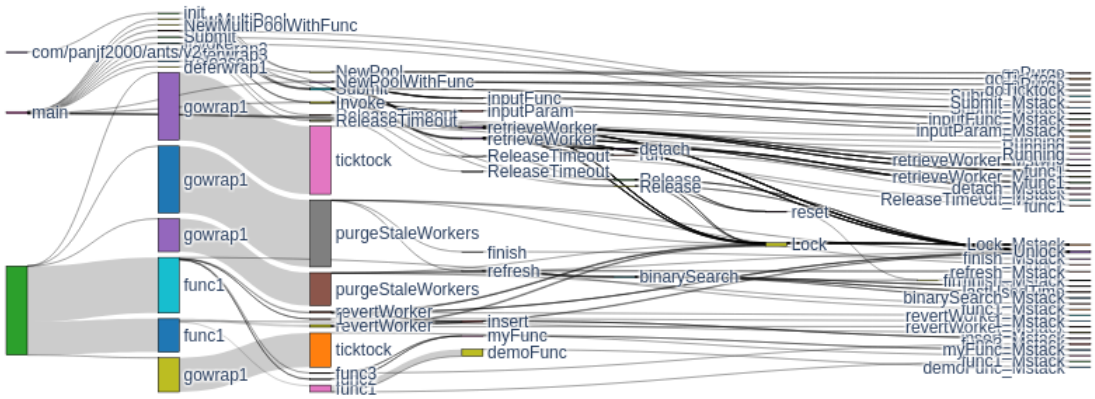
[1] viz https://github.com/ebitengine/oto/tree/main/example

7.3 Experiment 2: knihovna *ants*

Druhý experiment je použití nástroje *gotrace* na větším projektu. Pro tento účel byl vybrán program obsahující ukázkovou implementaci použití oblíbené knihovny *ants*[2] implementující *pool goroutin* s fixní kapacitou, správou a recyklací velkého množství *goroutin*. *Ants* také podporuje přizpůsobení kapacity *poolu goroutin* nejen při jeho vytvoření ale i za běhu programu. V tomto experimentu byly profilovány funkce balíčku `main` a balíčku použité knihovny *ants* `github.com/pandjf200/ants`. Výsledné profily byly poté vizualizovány pomocí *Sankey* diagramu ze Sekce 6.3 a jsou zobrazeny na Obrázcích 7.1 a 7.2.



Obrázek 7.1: Vizualizace programu *ants* v jazyce *Go* implementující ukázku použití balíčku *ants* zobrazující inkluzivní čas funkcí.



Obrázek 7.2: Vizualizace programu *ants* v jazyce *Go* implementující ukázku použití balíčku *ants* zobrazující exkluzivní čas funkcí.

[2] viz https://github.com/panjf2000/ants

Profilování knihovny *ants*, která obsahuje 14 zdrojových souborů a s přepočtem zhruba 2500 řádků kódu, zabralo 66.72 sekund včetně vyhledávání informací o funkcích, čekání na zapnutí a vypnutí `gotrace.c`, profilování, zpracování naměřených dat a vytvoření profilu. Bylo nalezeno 78 funkcí k profilování a bylo vytvořeno 291 *eBPF* programů. Výsledný profil má velikost 1.9 MB a samotné profilování zabralo 42.38 vteřin.

Vizualizace potvrdily, že je možné nástroj *gotrace* použít i na větší projekty, ale potvrdily také jejich omezení skrze chybějící možnost pohybovat celým grafem a měnit měřítko grafu, o kterých jsme mluvili v Sekci 6.3. Taktéž ukázaly, že programy v jazyce *Go* musí zvětšovat zásobník jazyka *Go* poměrně často.

```
perun collect -c './ants' gotrace main github.com/panjf2000/ants
```

Výpis 7.2: Příkaz, kterým byl program profilován. Pro profilování funkcí z importovaného balíčku je potřeba zadat alespoň takovou část jména balíčku aby bylo poznat, o jaký balíček se jedná.

Kapitola 8

Závěr

Cílem této bakalářské práce bylo implementovat modul obsahující profilovací nástroj pro programy napsané v jazyce *Go* a tento modul integrovat do nástroje *Perun* popsaném v Kapitole 3. Výsledný nástroj sbírá nejen původně zamýšlený *wall clock* čas (ze Sekce 2.4), ale během implementace jsme přidali i čas strávený modifikací zásobníku jazyka *Go*. Nasbíraná data poté dokážeme zobrazit implementovaným modulem pro vizualizaci času funkcí a jejich tras pomocí *Sankey* diagramu.

Výslednou implementaci jsme demonstrovali na jednoduchých programech a vybraných knihovnách jazyka *Go*. Výsledky ukázaly nízkou režii implementovaného nástroje díky použití technologie *eBPF*. Ukázaly také, že výsledný nástroj dokáže ve spojení s implementovanou vizualizací zobrazit časově náročnější funkce a pomoci tak odhalit potenciální chyby v programech. Nakonec jsme ukázali, že je možné implementovaný nástroj použít i na větší projekty.

Profilování programů napsaných v jazyce *Go* pomocí technologie *eBPF* je rozsáhlé a poměrně nové téma a tato práce by měla sloužit spíše jako její začátek. Navazující práce se může věnovat řešení známých omezení nástroje, vypsaných v Sekci 6.2, jako např. získávání úplných tras funkcí nebo implementací nástroje pro operační systém Windows. Dále by bylo možné v práci pokračovat vylepšením logiky kolektoru např. v podobě formátu zasílaných dat nebo nástroj rozšířit o nové metriky (automatické uvolnění paměti, argumenty nebo návratové hodnoty z funkcí). Bylo by možné přidat novou vizualizaci pro naměřená data nebo vylepšit implementovanou vizualizaci.

Literatura

[1] *BPF Type Format (BTF)* [online]. The kernel development community [cit. 2024-04-02]. Dostupné z: https://docs.kernel.org/bpf/btf.html.

[2] *Bpftool* [online]. [cit. 2024-04-02]. Dostupné z: https://www.mankier.com/8/bpftool.

[3] *Diagnostics—The Go Programming Language* [online]. [cit. 2024-04-05]. Dostupné z: https://go.dev/doc/diagnostics.

[4] *Enabling the Go Profiler* [online]. [cit. 2024-04-05]. Dostupné z: https://docs.datadoghq.com/profiler/enabling/go/.

[5] *Libbpf* [online]. [cit. 2024-04-02]. Dostupné z: https://github.com/libbpf/libbpf.

[6] *Libbpf Overview* [online]. [cit. 2024-04-02]. Dostupné z: https://libbpf.readthedocs.io/en/latest/libbpf_overview.html.

[7] *Pin—A dynamic Binary Instrumentation Tool* [online]. [cit. 2024-04-03]. Dostupné z: https://www.intel.com/content/www/us/en/developer/articles/tool/pin-a-dynamic-binary-instrumentation-tool.html.

[8] *Pprof* [online]. [cit. 2024-04-12]. Dostupné z: https://github.com/google/pprof.

[9] *Pprof package* [online]. [cit. 2024-04-05]. Dostupné z: https://pkg.go.dev/runtime/pprof.

[10] *Trace package* [online]. [cit. 2024-04-05]. Dostupné z: https://pkg.go.dev/golang.org/x/net/trace.

[11] *Tracing Go Applications* [online]. [cit. 2024-04-05]. Dostupné z: https://docs.datadoghq.com/tracing/trace_collection/automatic_instrumentation/dd_libraries/go/.

[12] *What is eBPF? An Introduction and Deep Dive into the eBPF Technology* [online]. ebpf.io [cit. 2024-04-02]. Dostupné z: https://ebpf.io/what-is-ebpf/.

[13] *Go crash with uretprobe* [online]. Srpen 2013 [cit. 2024-04-26]. Dostupné z: https://github.com/iovisor/bcc/issues/1320.

[14] *BPF Compiler Collection (BCC)* [online]. IO Visor Project, 2022 [cit. 2024-04-02]. Dostupné z: https://github.com/iovisor/bcc.

[15] *What is Datadog? What Does Datadog Do? All Questions Answered* [online]. 4. ledna 2024 [cit. 2024-04-05]. Dostupné z: https://www.finout.io/blog/what-is-datadog.

[16] Arapov, A. *Uretprobes: return probes implementation* [online]. 2013 [cit. 2024-04-19]. Dostupné z: https://lwn.net/Articles/543924/.

[17] Farrell, C. *The Fast Guide to Applicatoin Profiling* [online]. red-gate, duben 2010 [cit. 2024-04-02]. Dostupné z: https://www.red-gate.com/simple-talk/development/dotnet-%20development/the-fast-guide-to-application-profiling/.

[18] Felix, G. *Fgprof—The Full Go Profiler* [online]. [cit. 2024-04-12]. Dostupné z: https://github.com/felixge/fgprof.

[19] Felix, G. *Debug Go Request Latency with Datadog's Profiling Timeline* [online]. 9. srpna 2023 [cit. 2024-04-12]. Dostupné z: https://blog.felixge.de/debug-go-request-latency-with-datadogs-profiling-timeline/.

[20] Fiedor, T. a Pavela, J. *Perun: Lightweight Performance Version System* [online]. 2022 [cit. 2024-04-02]. Dostupné z: https://github.com/Perfexionists/perun.

[21] Fiedor, T. a Pavela, J. *Perun Documentation* [online]. 06. listopadu 2023 [cit. 2024-04-02]. Dostupné z: https://github.com/Perfexionists/perun/blob/devel/docs/pdf/perun.pdf.

[22] Gregg, B. *Choosing a Linux Tracer (2015)* [online]. 2015 [cit. 2024-04-03]. Dostupné z: https://www.brendangregg.com/blog/2015-07-08/choosing-a-linux-tracer.html.

[23] Gregg, B. *BPF binaries: BTF, CO-RE, and the future of BPF perf tools* [online]. 2020 [cit. 2024-04-02]. Dostupné z: https://www.brendangregg.com/blog/2020-11-04/bpf-co-re-btf-libbpf.html.

[24] Gregg, B. *Flame Graphs* [online]. 2020 [cit. 2024-04-04]. Dostupné z: https://www.brendangregg.com/flamegraphs.html.

[25] Gregg, B. *Linux Extended BPF (eBPF) Tracing Tools* [online]. 2021 [cit. 2024-04-02]. Dostupné z: https://www.brendangregg.com/ebpf.html.

[26] Gregg, B. *System Performance: Enterprise and the Clound*. 2. vyd. Addison-Wesley Publishing Company, 2021. ISBN 0-13-682015-8.

[27] Hájek, V. *Analýza výkonu programů v jazyce C#*. Brno, CZ, 2023. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Dostupné z: https://www.vut.cz/studenti/zav-prace/detail/148641.

[28] Knyszek, M. *More powerful Go execution traces* [online]. 14. března 2024 [cit. 2024-04-05]. Dostupné z: https://go.dev/blog/execution-traces-2024.

[29] Liang, Z. Real World Debugging with eBPF. In:. Singapore: USENIX Association, červen 2023.

[30] Močáry, P. *Performance Analysis of Programs Based on PIN Framework*. Brno, CZ, 2022. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Dostupné z: https://www.fit.vut.cz/study/thesis/23847/.

[31] Míchal, O. *Analýza spotřeby zdrojů v programech*. Brno, CZ, 2023. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Dostupné z: https://www.vut.cz/studenti/zav-prace/detail/146746.

[32] Nakryiko, A. *BPF skeleton and BPF app lifecycle* [online]. 2020 [cit. 2024-04-02]. Dostupné z: https://nakryiko.com/posts/bcc-to-libbpf-howto-guide/#bpf-skeleton-and-bpf-app-lifecycle.

[33] Nakryiko, A. *Journey to libbpf 1.0* [online]. 2022 [cit. 2024-04-02]. Dostupné z: https://nakryiko.com/posts/libbpf-v1/.

[34] Nethercote, N. *Dynamic binary analysis and instrumentation.* UCAM-CL-TR-606. University of Cambridge, Computer Laboratory, listopad 2004. Dostupné z: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-606.pdf.

[35] Pavela, J. *Efficient Techniques for Program Performance Analysis.* Brno, CZ, 2020. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Dostupné z: https://www.fit.vut.cz/study/thesis/19092/.

[36] Robertson, A. *Bpftrace* [online]. [cit. 2024-04-02]. Dostupné z: https://github.com/bpftrace/bpftrace.

[37] Rocca, E. *Comparing SystemTap and bpftrace* [online]. 13. dubna 2021 [cit. 2024-04-03]. Dostupné z: https://lwn.net/Articles/852112/.

Příloha A

Obsah přiloženého paměťového média

Přiložené paměťové médium obsahuje následující složky:

1. perun — složka obsahující systém *Perun* včetně vytvořeného modulu profilovacího nástroje a vizualizace.
2. changelog.txt — textový soubor obsahující změny (*patch*).
3. example_programs — složka s programy využitými na experimenty.
4. readme.md — manuál použití nástroje a vizualizace.
5. latex-source — LaTeX zdrojové soubory.
6. xnespo10.pdf — bakalářská práce ve formátu PDF.