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 C#

PERFORMANCE ANALYSIS OF C# PROGRAMS

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE **VOJTĚCH HÁJEK**
AUTHOR

VEDOUCÍ PRÁCE **Ing. JIŘÍ PAVELA**
SUPERVISOR

BRNO 2023

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

Zadání bakalářské práce

148641

Ústav:	Ústav inteligentních systémů (UITS)
Student:	**Hájek Vojtěch**
Program:	Informační technologie
Specializace:	Informační technologie
Název:	**Analýza výkonu programů v jazyce C#**
Kategorie:	Analýza a testování softwaru
Akademický rok:	2022/23

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 C#.
3. Navrhněte a implementujte v rámci nástroje Perun modul, který bude měřit spotřebu alespoň jednoho zdroje v C# programech. Dále prozkoumejte možnosti asociace naměřené spotřeby vzhledem ke konkrétním prvkům programu (např. k funkcím, základním blokům nebo řádkům kódu).
4. Navrhněte a implementujte vhodnou vizualizaci pro interpretaci dat naměřených vytvořeným modulem (např. sunburst graph nebo flame graph).
5. Demonstrujte řešení na alespoň jednom netriviálním projektu.

Literatura:

- Oficiální stránky projektu Perun: https://github.com/tfiedor/perun
- Osipov, A. (2019). Selecting C# Profiling Tools. Bachelor's Thesis.
- 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:	1.11.2022
Termín pro odevzdání:	10.5.2023
Datum schválení:	3.11.2022

Abstrakt

Cílem této práce je rozšířit výkonnostní verzovací systém – Perun implementací modulu pro profilování programů napsaných v jazyce C#. Toto rozšíření implementuje profiler s technikou sledování událostí, které jsou získávány pomocí .NET runtime profilovacího aplikačního rozhraní. Profiler dokáže sbírat metriky o sledování funkcí a spotřebě paměti. Naměřené profily potom dokáže dále interpretovat do grafů jako je korelační diagram nebo mapa stromu volání.

Abstract

The goal of this thesis is to extend the Performance Version System – Perun by implementing a module for profiling programs written in C# language. This extension implements a tracing profiler with the use of .NET runtime profiling aplication interface. Profiler can collect metrics about trace functions and memory consumption. Measured profiles can then be interpreted into graphs like a scatter plot or a treemap.

Klíčová slova

C#, .NET, profilování, výkonnostní testování, Perun, dynamická analýza

Keywords

C#, .NET, profiling, performance testing, Perun, dynamic analysis

Citace

HÁJEK, Vojtěch. *Analýza výkonu programů v jazyce C#*. Brno, 2023. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Jiří Pavela

Analýza výkonu programů v jazyce C#

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.

.....
Vojtěch Hájek
8. května 2023

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

Testování je v dnešní době již nedílnou součástí softwarového vývoje. Slouží nám k ověření funkčnosti programu tak, aby se výsledný produkt dostal k zákazníkovi v co nejlepší možné podobě. Zpravidla se zaměřuje na testování funkčnosti (program dělá co má), bezpečnosti (program není zranitelný), ale v poslední době i na často přehlíženou výkonnost a spotřebu zdrojů programu (program je efektivní).

Analýza výkonu programů se postupně stává klíčovou součástí vývoje, protože nedostatečná výkonnost může často vést na značné problémy, jako jsou nadměrné využití procesoru, úniky paměti, nebo dokonce i výpadky serverů a systémů. Vše může mít pak za následek ztrátu zákazníků a nebo i finanční újmu. Čím jsou ale programy rozsáhlejší a komplexnější, tím je i více složitější tyto výkonnostní hrozby najít a opravit. Tomuto procesu se zpravidla věnuje tzv. profilování: získávání informací o běhu programu.

Profilování je pouze jedna z forem dynamické analýzy programu, tedy analýzy vykonávané za běhu programu. K této analýze nám slouží profilovací nástroje (tzv. "*profilery*"), které dokáží sledovat a zaznamenávat potřebné metriky v závislosti na profilovaném programu. Naměřená data pak dále dokáží interpretovat a vývojáři díky nim dokáží např. analyzovat výkonnostní hrozby, potenciály pro optimalizaci nebo požadavky na stroje. *Profilery* se odlišují především podle zaměření na konkrétní programovací jazyk, a to z důvodu limitací *frameworků*, které slouží pro instrumentaci programů.

Jedním z populárních programovacích jazyků je jazyk C#: jazyk, který je moderní, objektově orientovaný a typově bezpečný. Umožňuje vytvářet mnoho typů bezpečných a robustních aplikací. Vychází z rodiny céčkových jazyků, tedy jazyka C a C++. Pro analýzu těchto programů existuje řada profilovacích nástrojů, ale většina je dostupná pouze v komerční placené licenci. Současně, ne všechny podporují analýzu nebo interpretaci dat pro programy na Unixových systémech. Jejich použití v rozsáhlejších řešeních pro správu výkonu aplikací, jako jsou nástroje Perun nebo PerfRepo, je tedy nemožné.

Tato bakalářská práce se věnuje právě tomuto problému analýzy výkonu C# programů. Cílem je vytvoření modulu, který bude rozšiřovat systém pro verzování výkonnostních profilů Perun, který je vyvíjen skupinou VeriFIT na fakultě Informačních technologií VUT, o profilování programů napsaných v jazyce C#. Motivací je rozšířit Perun, který je multiplatformní, o budoucí možnosti profilovat programy vytvořené pro systém Windows, ale i rozšířit současnou podporu programů pro systém Linux.

Následující kapitoly se budou věnovat těmto tématům. Představení analýzy programu, profilování, jeho způsoby a měřené metriky v kapitole 2. Dále seznámení se s verzovacím systémem výkonnostních profilů Perun v kapitole 3. Poté analýza existujících řešeních

pro profilování C# programů v kapitole 4. Kapitola, která se věnuje .NET platformě, jeho *runtime* architektuře a jeho profilovacímu aplikačnímu rozhraní je popsána v kapitole 5. Další kapitoly se věnují implementaci a návrhu modulu. Sběr profilovacích dat je popsán v kapitole 6 a vizualizace v kapitole 7. Poslední kapitolou jsou experimenty vykonané nad tímto modulem v kapitole 8.

Kapitola 2

Analýza programu

Tato kapitola je věnována popisu způsobu analýzy programu s důrazem na analýzu výkonnosti. Je v ní definováno profilování, jeho způsoby měření, granularita a základní metriky, které profilováním můžeme získat. Na závěr jsou uvedeny nejčastěji používané techniky pro profilování C# programů.

2.1 Statická a dynamická analýza

Analýza programu se dá kategorizovat do dvou skupin na základě toho, jak analýza programu probíhá [31].

Statická analýza. Statická analýza je analýza zdrojového nebo binárního spustitelného kódu programu, která probíhá bez spuštění programu. Statickou analýzu využívají především nástroje pro kompilaci kódu, které díky ní provádí částečnou analýzu korektnosti v programu, např. typová kontrola, nebo analýzu pro optimalizaci, tedy např. různé transformace kódu vylepšující výkonnost programu. Pro analýzu výkonu programu existují přístupy, které odvozují z kódu složitost programu [10] nebo tzv. WCET (*Worst case Eexecution Time*) [5]. Současně existují přístupy, které hledají vybrané výkonnostní chyby (např. neefektivní iterace). Nevýhodou je, že současné přístupy neškálují dostatečně a tedy je nelze použít pro pokročilé projekty.

Dynamická analýza. Dynamická analýza analyzuje program za jeho běhu programu. Dynamickou analýzu využívají nástroje, jako jsou *profilery*, vizualizéry a výkonnostní analyzátory. Zatímco statická analýza umožňuje zahrnovat veškeré scénáře běhu programem, tak dynamická může analyzovat pouze ty cesty programem, které byly provedeny v rámci jeho běhu. Nevýhodou dynamické analýzy je, že vyžaduje vhodné vstupy (tzv. "*workloads*"), které nemusí vést k anomálii nebo problému s výkonem.

2.2 Profilování

Techniku profilování (a tzv. "*profilery*": nástroje, které ji realizují) řadíme do oblasti dynamické analýzy. *Profiler* je nástroj, který dokáže sledovat vybrané metriky programu (např. doba běhu funkcí) za jeho běhu. Nasbíraná data pak následně dokáže analyzovat případně i vizualizovat. *Profilery* slouží především vývojářům k detekci a dodatečné lokalizaci výkonnostních hrozeb v jejich programu. *Profilery* dělíme na několik různých druhů podle

toho, jakým způsobem získávají naměřená data. Tato sekce je věnována třem základním způsobům, jakými *profiler* může získávat data [12].

2.2.1 Vzorkování (Sampling)

Vzorkovací *profiler* sbírá data tak, že sleduje instrukce programu v daných intervalech (zpravidla časových nebo diskrétních, tzn. sledují každé n-té volání), např. s využitím přerušení v operačním systému. Tato metoda, oproti například instrumentaci programu, má výhodu takovou, že nezasahuje do zdrojového kódu, ale pouze ho sleduje (pasivní přístup). Vzorkování má minimální vliv na výkon programu. Jeho značnou nevýhodou je ale skutečnost, že vzorkovací *profilery* nejsou zdaleka tak přesné, jako ostatní přístupy a nemusí nasbírat věrohodná data o výkonu programu. Pokud se totiž mezi intervaly sledování vyskytne výkonnostní problém, *profiler* ho nemá jak zachytit. Dalším problémem může být, že některé menší funkce volané vícekrát v daném intervalu si může *profiler* vyhodnotit jako výkonnostní hrozbu. Vzorkování je využíváno, například v nástroji *DotTrace* (viz. sekce 4.1), nebo *IntelVtune* (viz. sekce 4.5).

2.2.2 Instrumentace programu

Instrumentační *profiler* funguje na základě vložení vlastního kódu do profilovaného programu, tedy tzv. instrumentací. *Profiler* na základě instrumentovaného kódu nasbírá potřebná data. Jsou dva způsoby, jak instrumentovat profilovaný program: první je instrumentaci provést přímo nad zdrojovým kódem před spuštěním programu (tzv. "*source-code modifying profiler*"), druhý způsob pracuje v *runtime*[1], tím že instrumentuje binární spustitelný kód (tzv. "*binary profiler*"). Zpravidla se instrumentují volání funkcí, alokace paměti nebo uživatelem vybráné bloky kódu.

Díky instrumentaci programu nasbírá *profiler* daleko přesnější data. Ovšem nevýhodou je přidaný kód navíc do programu, který může program značně zpomalit, tudíž měření výkonu nemusí být tak přesné. Instrumentace je využívána například v nástroji *Visual Studio profiler* viz. 4.2.

2.2.3 Sledování událostí (tzv. "*Tracing*")

Funguje na principu sledování generovaných událostí programu. Události mohou vznikat například v *runtime* programu, nebo jako hardware notifikace. Tyto *profilery* je možné použít pouze pro programy, které mají v jejich *runtime* exekuci[2] profilovací aplikační rozhraní (například Java, .NET, nebo Python). Toto aplikační rozhraní dokáže posílat události *profileru*. Mezi tyto události může patřit spouštění funkcí, CPU využití, automatické uvolnění paměti, a další. Podobně jako vzorkování nezasahuje do zdrojového kódu profilovaného programu. Jeho nevýhoda spočívá v tom, že dokáže sledovat metriky pouze na základě událostí.

2.3 Granularita profilování

Profilování může být prováděno na různých stupních granularity. Vyšší granularita je často na úkor rychlosti profilování. Mezi základní typy granularity profilování patří [13]:

[1] runtime = virtuální prostředí běhu programu
[2] exekuce = provedení

- Řádková – nejpřesnější granularita profilování. Dokáže měřit a analyzovat na základě jednotlivých řádků zdrojového kódu. Přesnost profilování je na úkor značného zpomalení. Tato granularita se dá uskutečnit pouze pomocí instrumentačního *profileru*.
- Funkcionální – profiluje na základě celých metod. Dokáže díky tomu lokalizovat problémy pro jednotlivé funkce a nezpomalit tolik proces profilování. Je to jediná granularita, která se dá zachytit vzorkováním. Využívá se ale i u techniky sledování událostí nebo profilování pomocí instrumentace.
- Selektivní – je kombinací granularity funkcionální a řádkové.

2.4 Výkonnostní metriky

Pomocí *profileru* můžeme měřit metriky, které nám mohou pomoct odhalit příčiny výkonnostních problémů. Zde je souhrn základních metrik, které mohou být využity i pro analýzu C# programů [8].

- Počet volání – udává statistiku, kolikrát byly dané funkce volány. Jedná se o nejjednodušší metriku, kterou může analýza výkonnosti podporovat. Pokud je totiž některá netriviální funkce volána častokrát, může to znamenat výkonnostní problém.
- Čas *Wall clock* – celkový čas nutný pro provedení funkce, včetně času stráveného čekáním. Pomáhá určit, které části programu se dají urychlit např. zavedením asynchronního přístupu.
- Čas CPU - celkový čas provedení funkce, kromě času stráveného čekáním.
- Úzká hrdla využití zdrojů - vysoké zatížení některého ze zdrojů, např. procesoru, místa na disku, grafické karty, sdílených vláken.
- Alokace paměti – metrika, která nám pomáhá sledovat, jak je využívaná paměť alokována. Pokud by se s pamětí nepracovalo správně, mohlo by to způsobit značné problémy, např. únik paměti, tedy spotřebování veškeré dostupné paměti, někdy až do pádu programu.
- Automatické uvolnění paměti (tzv. "*Garbage Collection)*"– u moderních programovacích jazyků je využíváno automatické uvolnění paměti. Její sledování nám pomůže odhalit, zda s pamětí pracuje podle očekávání.

Další metriky můžou být například vytížení sítě, práce s vyrovnávací pamětí, spotřeba energie a další.

2.5 Způsoby profilování C# programů

Profilování C# programů se ve většině stávajících *profilerů* řeší následujícími způsoby. V kapitole 4, jsou dále uvedeny vybrané existující *profilery* pro jazyk C#.

2.5.1 Instrumentace programu

Pomocí knihoven (např. *prometheus-net*[1]) je možné C# programy instrumentovat pro potřeby profilování. Buď se zde může instrumentovat samotný C# zdrojový kód, nebo lze instrumentovat přímo binární spustitelný soubor. Další možností pro instrumentaci je tzv. Microsoft Intermidiate Language, což je neutrální jazyk pro podporované .NET programovací jazyky, jako jsou F#, nebo Visual Basic.

Tento způsob umožňuje především detailní a přesné naměřená data, ovšem neumožňuje nám například získávat údaje o automatickém uvolněním paměti. Další nepřesnost pak plyne z použití instrumentace samotné, tedy z přidání kódu navíc [26].

2.5.2 Event Tracing for Windows (ETW)

Event Tracing for Windows je nástroj pro operační systém Windows, který umožňuje zaznamenávat události na úrovni jádra. Jako *profiler* ho využívá především *PerfView* blíže popsán v sekci 4.3. Díky tomuto nástroji dokáže poskytnout detailní informace o událostech, které se v aplikaci dějí [16].

Jeho nevýhoda spočívá především v nutnosti používat operační systém Windows. Pro jiné operační systémy by bylo nutné využít jiný nástroj, jako např. LLTng, který využívá modul *PerfCollect* (součást nástroje *PerfView*).

2.5.3 CLR Profiling API

Většina implementací *profilerů* pro C# programy využívá zabudovaného aplikačního rozhraní uvnitř *Common Language Runtime* (CLR), popsaný v sekci 5.3. CLR lze totiž nakonfigurovat tak, aby zasílal upozornění konkrétním knihovnám DLL *profileru* o generovaných událostech. Pomocí tohoto mechanismu lze shromažďovat klíčové informace o provádění funkcí, CPU, paměti a o automatickém uvolnění paměti (více v sekci 5.3). Vzhledem k rozšiřitelnosti platformy .NET pro operační systémy Linux a MacOS, by tento přístup umožnil vývoj multiplatformního *profileru* [24].

[1] Prometheus-net instrumentační knihovna: https://github.com/prometheus-net/prometheus-net

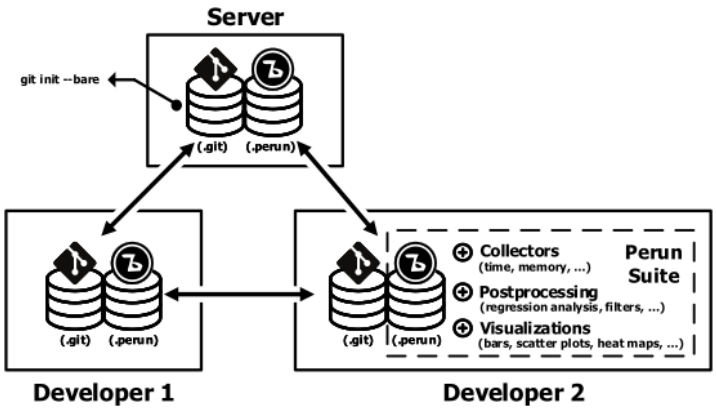
Kapitola 3

Perun

V této kapitole je stručně představen výkonnostní verzovací systém Perun. První část je věnována přehledu tohoto nástroje. Druhá část pojednává o architektuře nástroje a o tom, jak nástroj funguje. Kapitola vychází zejména z dokumentace Perunu [33] a dále z prací [32, 29], které tento nástroj postupně rozšiřovaly.

3.1 O nástroji

Perun je otevřený výkonnostní verzovací systém. Byl vytvořen výzkumnou skupinou VeriFIT na Fakultě informačních technologií na Vysokém učení technickém. Nástroj ve svém jádru propojuje verze projektu s korespondujícími výkonnostními profily s využitím verzovacího systému (např. Git). Perun následně umožňuje automatickou detekci výkonnostních rozdílů v nových verzích projektu, čímž se dají včas detekovat výkonnostní problémy. Perun je určen především pro využití jedním vývojářem (nebo malým týmem) jako kompletní řešení automatizace, ukládání a interpretace analýzy výkonu projektu, nebo může být využit pro velké projekty a týmy jako dedikované úložiště. Na obrázku 3.1 je ilustrován průběh vývoje projektu s nástrojem Perun.



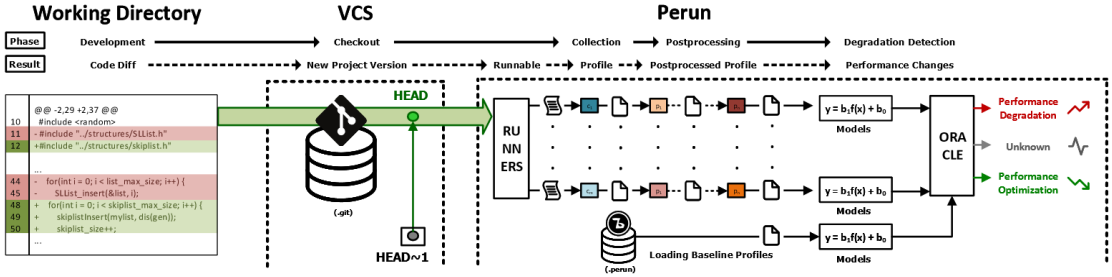
Obrázek 3.1: Ilustrace nástroje Perun nad verzovacím systémem Git v průběhu vývoje projektu [34].

Součástí nástroje je snadná **automatizace** procesu. Pomocí automatizace lze nastavit tzv. háčky (”*hooks*”) v používaném verzovacím systému (např. při použití Git - "pull-request", nebo "commit"), podle kterých Perun dokáže poznat publikaci nové verze projektu a poté sám provede nové měření výkonu projektu a porovná výkonnostní profily s předcházejícími verzemi.

Perun ukládá výkonnostní profily do souborového systému. Současně Perun poskytuje **kontext** historie výkonnosti projektu, který uživateli pomůže najít zdroj výkonnostního problému nebo pomůže s optimalizací programu s využitím změn zdrojového kódu z minulých verzí projektu.

Další výhodou nástroje Perun je jeho snadná **rozšiřitelnost**, ať už novými formáty dat, kolektorů, nebo vizualizací. Samotné moduly jsou totiž nezávislé a snadno rozšiřitelné o další komponenty. K tomu pomáhá i unifikovaný formát dat v podobě JSON. Jednoduchost použití je poté docílena podobností příkazů se známým systémem Git, který je široce rozšířený mezi vývojáři. Současná verze nabízí rozhraní přes příkazovou řádku, přičemž interaktivní uživatelské rozhraní je ve vývoji.

Perun je tedy vhodný nástroj pro automatizaci výkonnostních regresních testů, zpracovávání již existujících profilů, nebo pro efektivní interpretaci nasbíraných výsledků.

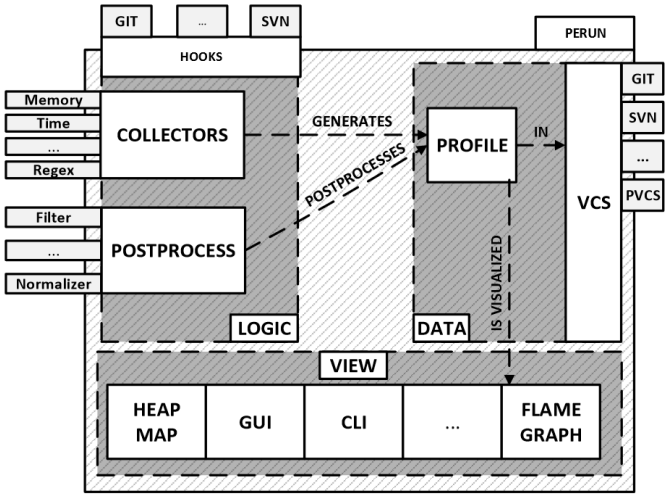


Obrázek 3.2: Ilustrace obvyklého pracovního postupu nástroje Perun. Perun při každé nové verzi naměří nové výkonnostní profily a provede porovnání s předchozími profily. Perun následně provede vyhodnocení optimalizace nebo degradace výkonu projektu [34].

3.2 Architektura

Architektura Perunu se dá rozdělit do několika hlavních komponent: logika (příkazy, procesy, spouštěče, úložiště, zpracování), data (profily), pohled (vizualizace dat) a kontrola. Dále se skládá ze spousty menších komponent, jako jsou šablony, obálky verzovacích systémů a další. Tato sekce je zaměřena na popis jednotlivých hlavních komponent. K lepšímu znázornění architektury slouží obrázek 3.3, který vystihuje architekturu se zmíněnými komponenty.

Data. Tato komponenta poskytuje rozhraní pro správu výkonnostních profilů, kterou využívají všechny ostatní komponenty. Jedná se tedy o základní komponentu nástroje. Dále se stará o podporované obálky (tzv. ”*wrappers*”) nad verzovacím systémem. Data jsou unifikovaná do formátu JSON notace, pro zajištění snadné komunikace mezi dalšími komponentami.



Obrázek 3.3: Architektura nástroje Perun [34].

Logic. Stará se především o automatizaci, vyšší logiku nad správou a generováním výkonnostních profilů. Komponenta zapouzdřuje řadu sběračů dat a zpracování dat, které jsou zodpovědné za měření a následné zpracování naměřených profilů. Příklady hlavních kolektorů:

- *Memory* je sběrač, který zaznamenává paměťovou alokaci a celkové využití paměti pro C/C++ programy. Sběr dat je zajištěn pomocí `libunwind` knihovny a vlastních `libmalloc` knihoven.
- *Trace* měří délku běhu funkcí a uživatelsky definovaných bloků zdrojového kódu. Architektura sběrače umožňuje uživateli vybrat si z řady instrumentačních frameworků (tzv. "*engines*") pro kolekci výkonnostních dat.
- *Time* je sběrač pro sbírání času libovolných příkazů z profilovaného programu. Implementován je jako jednoduché zapouzdření nad časovou utilitou `time`.

Dále Perun obsahuje tyto hlavní zpracovače dat:

- *Normalizer* je používán pro škálování nasbíraných dat do intervalu (0,1). Hlavní záměr tohoto zpracovače je umožnit porovnávání z různých vstupů (tzv. "*workload*") nebo parametrů, které mohou mít různou škálovanost zdrojů.
- *Regression analysis* se snaží modelovat data různými regresními funkcemi a rozpoznat nejlepší model s respektem ke kvadratické chybě. Vyžaduje datovou sadu se závislými a nezávislými proměnnými, kdy výsledkem je model nezávislé proměnné (např. doba běhu na základě hodnot závislých proměnných – velikost parametrů funkce).

View. Tato komponenta je zodpovědná za vstupní/výstupní interakci s uživatelem. Pro lepší interpretaci dat používá několik vizualizačních technik. Příklady podporovaných vizualizačních metod jsou:

- *Bars* zobrazuje profily ve formě vertikálních pruhů, které mohou být buď seskupeny vedle sebe, nebo poskládaný na sobě.

- *Scatter* používá dvoudimenzionální mříž pro zobrazení dat jako bodů. Je vhodný pro výstup sledování nebo složité výsledky.
- *Heap Map* je vhodná pro vizualizaci paměťové spotřeby. Provádí vizualizace s mapou paměťových adres s lokacemi individuálních adres, jejich alokovanými objekty, nebo jak často je adresa využívána.

Check. Obsahuje několik detekčních metod, které se starají o reportování případných změn ve výkonu projektu. Metoda má na vstupu dva profily, kde jeden představuje novou verzi projektu a druhý stabilní verzi projektu (tzn. předchozí verzi). Ty jsou podle některé ze zabudovaných metod vyhodnoceny o optimalizaci, nebo degradaci výkonu. Příkladem porovnávacích metod jsou *Average Amount Threshold* (metoda založena na porovnávání průměrů), *Integral Method* (metoda založena na porovnávání integrálů pod modely), nebo *Local Statistics* (metoda založena na porovnávání souboru vybraných statistik).

Kapitola 4

Analýza existujících nástrojů

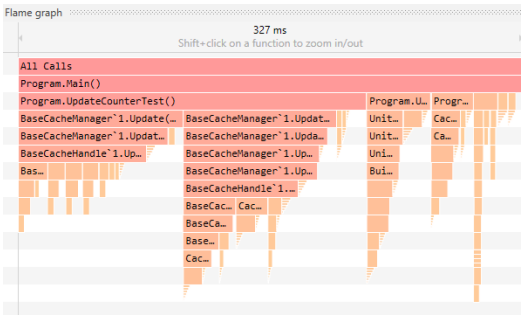
V této kapitole jsou představeny existující nástroje pro profilování C# programů. U *profilerů* je především uvedeno, co nabízí, jaká je jejich dostupnost a jak jsou na tom s podporou pro operační systém Linux.

4.1 JetBrains

Profilovací nástroje od společnosti JetBrains patří k nejpoužívanějším *profilerům*. JetBrains rozděluje profilování na dva různé nástroje. *DotTrace* je nástroj pro měření výkonnosti programu a *DotMemory* se zaměřuje na měření spotřeby paměti v programu. Oba nástroje jsou komerční a nezveřejňují zdrojový kód.

DotMemory i *DotTrace* jsou převážně nástroje pro operační systém Windows, pro profilování na jiných operačních systémech nabízí JetBrains řešení v podobě nástrojů příkazové řádky. Tyto nástroje zaznamenávají data pomocí techniky profilování. Nicméně analýzu dat je možné provádět pouze na operačním systému Windows.

DotTrace. Jedná se o výkonnostní *profiler*, který nabízí širokou nabídku podpory profilování .NET aplikací, od klasických aplikací, až po napojení se na vzdálené aplikace běžící na jiných počítačích. Dále poskytuje širokou nabídku možností profilování, jako jsou módy *sampling* (vzorkování zásobníku funkcí, podle časových intervalů), *tracing* (sledování událostí od CLR profilovacího API), *timeline* (využití událostí od ETW technologie), nebo *line-by-line* (instrumentace na řádkové úrovni granularity). Pro svou vizualizaci využívá především grafy volání funkcí, či tzv. "*flame graph*" ilustrovaný obrázkem 4.1. Vizualizace je doplněna ukázkami funkcí přímo ze zdrojového kódu [14].



Obrázek 4.1: Obrázek grafu *Flame graph* v nástroji *DotTrace*.

DotMemory. Jedná se o *profiler* pro měření spotřeby paměti. Stejně jako *DotTrace* podporuje jak lokální, tak vzdálené profilování. Navíc k tomu přidává možnost nahrání

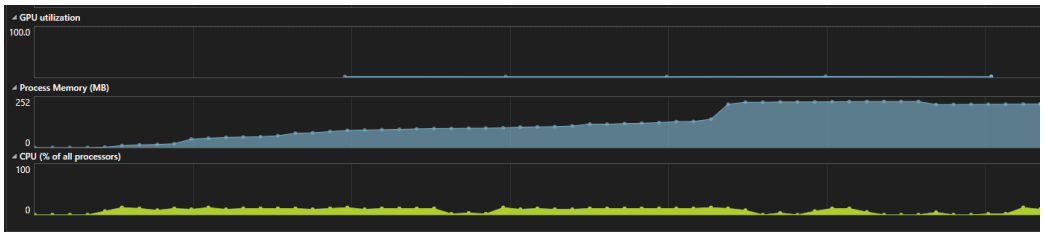
snímků paměti (tzv. "*memory dumps*") pořízené jiným způsobem, např. pomocí nástroje správce úloh. Pro vizualizaci používá například tzv. "*Sunburst diagram*", ilustrovaný obrázkem 4.2. *DotMemory* nabízí také funkce pro automatizaci profilování, které se mohou využít např. v tzv. "*Continous Integration*" procesu [15].

Jeho nevýhoda je především v analýze, která je pouze na základě pořízených snímků paměti. Není tedy možné detailně analyzovat průběh alokace paměti v programu. Tyto snímky se dají pořídit buď manuálně, nebo automaticky po každém automatickém uvolnění paměti. Nástroj *DotMemory* je tedy spíše vhodný pro velké projekty, kde by detailní informace o paměti v čase zabrali hodně paměti na disku.

Obrázek 4.2: Obrázek grafu *Sunburst diagram* v nástroji *DotMemory*, pro metriku velikosti alokace objektů v dané funkci programu.

4.2 Visual Studio profiler

Visual Studio profiler je komerční nástroj zabudovaný ve vývojovém prostředí *Visual Studio*. Nabízí řadu nástrojů, jako samostatné profilování výsledné verze (tzv. "*release build*") i profilování při ladění programu. Na začátku si uživatel může sám vybrat, jaké vlastnosti programu chce profilovat. Jsou zde například módy pro měření využití procesoru, paměti, grafické karty, nebo sledování asynchronních operací. Obrázek 4.3 ilustruje graf pro profilování celkové spotřeby paměti, GPU utilizaci a využití CPU.



Obrázek 4.3: Obrázek grafu v nástroji *Visual Studio profiler*. Profilování je pro metriky celkové spotřeby paměti, GPU utilizaci a využití CPU.

Visual studio profiler je postaven na CLR profilovacím API, ale podporuje i nastavení sledování událostí pomocí mechanismu *Event Tracing for Windows*, vzorkování a profilování pomocí instrumentací programu. *Visual studio* je dostupný pouze na operačním systému Windows [20].

4.3 PerfView

PerfView je otevřený *profiler*, vyvíjený společností Microsoft. Jako jeden z mála je založen na mechanismu *Event Tracing for Windows*, přes který dokáže profilovat. Od ostatních *profilerů* se liší především svou širokou nabídkou nástrojů, které se správnou konfigurací dokáží poskytnout opravdu detailně naměřená výkonnostní data.

Poskytuje i variantu pro operační systém Linux. Pomocí modulu *PerfCollect* lze na Linuxu profilovat program a nasbírat potřebná data. Tento modul se dá spustit pomocí příkazové řádky. Ovšem výsledná data se dají analyzovat pouze na operačním systému Windows. Tento postup je uveden jako oficiální doporučení od Microsoftu pro profilování na operačním systému Linux [23].

4.4 ANTS performance a memory profiler

ANTS performance a memory profiler jsou vyvíjeny společností RedGate. Podporují pouze operační systém Windows, pro jiné neposkytují žádné řešení. Tento *profiler* má komerční licenci a nenabízí žádnou studentskou licenci. Nabízí pouze 14-ti denní zkušební licenci pro firmy, které si o ní zažádají. Za těchto okolností nebylo možné tento nástroj vyzkoušet a je zde uváděn pouze jako populární komerční řešení [6].

4.5 Intel VTune

Profiler od společnosti Intel, který dokáže profilovat aplikace na Windows, MacOS či Linuxu. Pracuje na principu vzorkování, tedy sbírá data na základě událostí, které generuje procesor a další komponenty. Jak bylo již popsáno v sekci 2.2.1, vzorkovací technika často neudává přesná a spolehlivá data. Je ovšem nutné zmínit, že Intel VTune funguje pouze na hardwaru od společnosti Intel [11].

4.6 Mono profiler

Mono bylo využíváno především pro podporu běhu programů .NET na operačních systémech Linux a MacOS. Je to podporovaný open-source projekt Microsoftu, který vyvíjí společnost Xamarin s pomocí široké komunity nadšených vývojářů. Mono je kompatibilní do verze .NET 4.7 a C# 7. V rámci implementace Mono platformy se dá využít i *Mono profiler*, který je zabudovaný uvnitř platformy.

Mono profiler využívá generované události přes *Mono runtime*. Mezi tyto události patří notifikace o vstupu a výstupu funkcí, *Just In Time* kompilace, automatické uvolnění paměti a řada dalších. *Profiler* slouží pouze jako *logovací* nástroj a je spouštěn z příkazové řádky. Je kompatibilní napříč operačními systémy. Jeho výhodou jsou především nastavitelné režimy pro profilování, např. omezení sledování funkcí do určité hloubky zásobníku, nebo omezení vzorkování na určitý interval. Nevýhoda je především v analýze dat, kdy nástroj nenabízí žádnou interpretaci v podobě grafů, pouze souhrné výsledky profilování [27].

4.7 Zhodnocení

Z analýzy existujících nástrojů pro profilování C# programů vyplývá velký nedostatek podpory pro operační systémy Linux, nebo MacOS a je vidět nedostatek v podporovaných otevřených řešeních.

	Linux/MacOS		**Licence**	**Typ profileru**
	Sběr dat	Analýza		
JetBrains	Ano	Ne	komerční	vzorkování instrumentace sledování událostí
Visual Studio	Ano	Ne	komerční	vzorkování instrumentace sledování událostí
PerfView	Ano	Ne	otevřený	sledování událostí
ANTS	Ano	Ne	komerční	vzorkování instrumentace sledování událostí
Intel VTune	Ano	Ano	komerční	vzorkování
Mono profiler	Ano	Ano	otevřený	sledování událostí

Tabulka 4.1: Srovnání existujících nástrojů pro profilování.

Kapitola 5

.NET architektura pro profilování

Pro jazyk C# existuje řada implementací kompilátorů jazyka, jako například .NET platforma, Mono, Elements, nebo DotGNU. Jelikož většina C# aplikací je tvořena pod .NET implementací, budeme tuto práci cílit pouze na ni, a to konkrétně od verze .NET Core 1.0, jelikož starší verze .NET Framework běželi pouze na platformě Windows.

.NET je otevřená vývojářská platforma vytvořená společností Microsoft, která slouží pro vývoj velkého množství různých typů aplikací. Podporuje řadu programovacích jazyků: zejména C#, ale také třeba F#, nebo Visual Basic. Verze .NET Core je multiplatformní a tedy kromě operačního systému Windows je nyní možné vyvíjet aplikace i na operačních systémech Linux nebo MacOS.

V této kapitole popíšeme jak funguje *runtime* .NET programů a jak jej využít pro profilování C# programů. Následně představíme COM rozhraní a profilovací aplikační rozhraní.

5.1 Common Language Runtime (CLR)

Tato sekce je založena na Microsoft dokumentaci [18][28]. Každý program má řadu závislostí na prostředí, ve kterém běží. Může se jednat o potřebné knihovny, architekturu počítače, nebo podporované programovací jazyky. Splnění těchto závislostí typicky (alespoň částečně) zajišťují různé verze, nebo varianty kompilátorů. Ty mohou být uzpůsobeny pro různé architektury, operační systémy, nebo programovací jazyky. Tento způsob je využíván u programů napsaných v kompilovaných jazycích, jako např. C nebo C++. Jiným přístupem, který toto splnění závislostí dokáže řešit napříč různými prostředími, je tzv. virtuální prostředí běhu programu (*runtime*). Ten mimo podpory různých architektur či operačních systémů dokáže zajistit například i automatické uvolnění paměti. Virtuální prostředí běhu programu je také součástí platformy .NET, kde má název *Common Language Runtime* (zkráceně CLR). Microsoft CLR definuje následně [28]:

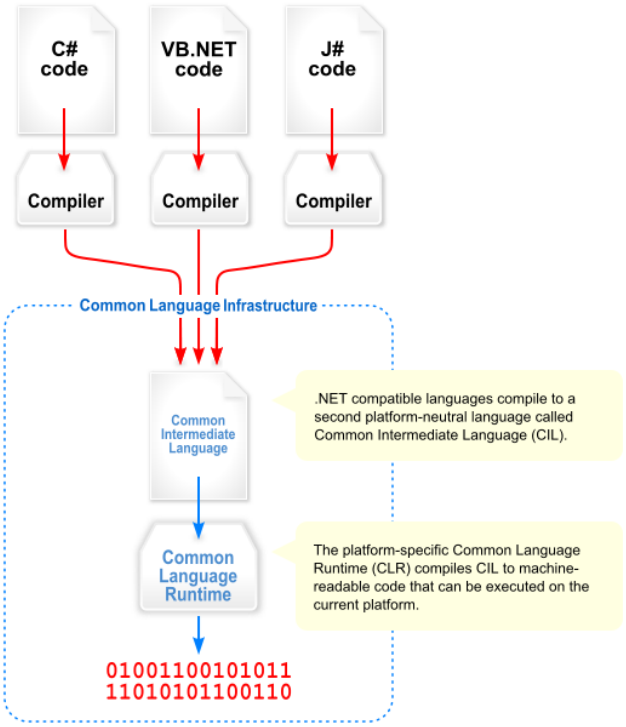
„*Common Language Runtime (CLR) je kompletní, vysokoúrovňové virtuální prostředí navržené pro podporu široké řady programovacích jazyků a součinnosti nad nimi.*“

Hlavní záměr CLR je především zjednodušení programování. Toho se snaží docílit zjednodušením programovacích jazyků, jednoduchostí knihoven, paměťovou a typovou bezpečností, automatickou správou paměti, nebo podporou vysokoúrovňových jazykových funkcí.

Předností CLR oproti jiným virtuálním systémům běhu programu je především podpora více programovacích jazyků. To programátorům umožní psát programy ve více různých

.NET spravovaných jazycích, a tím mohou vznikat vysoce komplexní a robustní programy. CLR totiž sám o sobě nepracuje s jednotlivými spravovanými jazyky (C#, F# ..), ale s mezikódem *Microsoft Intermidiate Language* (MSIL), do kterého se všechny spravované jazyky nejdříve zkompilují, viz obrázek 5.1. MSIL je sada instrukcí, která je nezávislá na procesoru a dá se efektivně převést do nativního kódu a to pomocí CLR.

V některých zdrojích bývá *Microsoft Intermidiate Language* (MSIL) uváděn jako *Common Intermidiate Language* (CIL), jedná se ovšem o stejný jazyk. V této práci budeme tento mezikód označovat vždy jako MSIL z důvodu konzistence.



Obrázek 5.1: Spravované jazyky v .NET jsou nejdříve kompilovány do *Microsoft Intermediate Language* jazyka a poté v CLR do nativního kódu [3].

CLR může provést kompilaci MSIL do nativního kódu dvěma způsoby, a to buď pomocí kompilace *Just-In-Time* (JIT), nebo pomocí kompilace *ReadyToRun* (R2R). JIT kompilace se provádí v době spuštění aplikace, kde je její obsah načten a proveden. To může často zapříčinit zpomalení vykonávaného programu, avšak ve většině případů se jedná o zpomalení zanedbatelné. Rychlejší způsob kompilování do nativního kódu je pomocí kompilace R2R, který jazyk MSIL předběžně zkompiluje pro jeho rychlejší spuštění.

5.1.1 Hlavní koncepty CLR

CLR se vyznačuje hned několika unikátními koncepty, které jej odlišují od ostatních *runtime* exekucí. V této části jsou některé z nich popsány [28].

Spravovaný kód (tzv. "*managed code*"). Je takový kód, který je spravován uvnitř CLR. Nespravovaný kód (tzv. "*unmaneged code*"), označuje kód mimo správu CLR, např. kód na úrovni operačního systému. Jelikož programy často využívají i kód operačního sys-

tému, dochází zde k přechodům mezi spravovaným a nespravovaným kódem. Spravovaný kód je ve formátu již uvedeného jazyka MSIL. Spravovaný kód je velkou výhodou CLR, jelikož díky němu má značnou kontrolu nad prováděním programu – často i na úrovni instrukcí. Tuto kontrolu pak CLR dokáže využít pro další své funkce, jako je například automatické uvolnění paměti.

Správa paměti a typová bezpečnost. Jedna z hlavních předností CLR je paměťová a typová bezpečnost. Ta je zajištěna především díky automatické správě paměti, která zaručuje, že se program může dostat pouze k alokované paměti, a také že tato paměť bude následně uvolněna. Typové bezpečnosti je dosaženo tím, že pro každý alokovaný objekt je přiřazen typ. Tento typ má pro sebe označené jednotlivé operace, které s ním program může provádět. Verifikace typové bezpečnosti v CLR, založené na statické analýze jazyka MSIL, probíhá ještě před spuštěním programu [28]. Tato analýza ověřuje, jestli je většina operací nad objekty opravdu typově bezpečná. To zajišťuje nejen typovou bezpečnost, ale také dodatečné záruky pro jednotlivé typy.

Spravovaná vlákna (tzv. "*Managed Threads*"). CLR plně podporuje více-vláknové programy ve spravovaném kódu. Vytváří pro ně tzv. spravovaná vlákna, která jsou zapouzdřena nad vlákny operačního systému.

Sestavení (tzv. "*Assembly*"). Sestavení je kolekce typů a prostředků, které jsou vytvořeny tak, aby vzájemně spolupracovaly a tvořily logickou jednotkou funkčnosti. Sestavení v .NET má podobu spustitelných souborů (*.exe*) nebo dynamických linkovaných knihoven (*.dll*). Sestavení poskytují CLR informace o implementaci typů. Při spuštění programu tedy CLR ví jaké typy jsou k dispozici a jak je správně použít [22].

5.1.2 Automatická správa paměti

Automatická alokace a uvolnění paměti je významnou funkcionalitou CLR. Díky ní se programátor nemusí zabývat manuální správou paměti, jako je tomu například v programech C++. Tato funkcionalita ulehčuje práci vývojářům, ale také pomáhá zamezit řadě běžných chyb při nesprávné práci s pamětí v programech. Princip automatické správy paměti v CLR spočívá ve sledování alokovaných objektů a občasné detekci, zda jsou objekty stále v programu dosažitelné. Pokud nejsou, paměť pro tyto objekty se uvolní.

Automatická správa paměti, kromě odstranění potřeby práce s manuální uvolňováním paměti, zajišťuje také efektivní alokování paměti na spravované haldě, zpětné získávání paměti po uvolnění některých z objektů a také zajišťuje bezpečnou práci s pamětí [28].

Architektura automatické správy paměti se skládá ze dvou komponent, tedy alokátoru a kolektoru:

Alokace paměti. Při inicializaci nového procesu je vytvořena souvislá oblast paměti. Tato rezervovaná paměť má v CLR název spravovaná halda (tzv. "*managed heap*"). Spravovaná halda se dále dělí na malou haldu objektů (tzv. "*small object heap*") a velkou haldu objektů (tzv. "*large object heap*"). Velká halda je vyhrazena pro objekty vetší než 85 000 bajtů, všechny zbylé objekty jsou alokovány na malé haldě. Všechny reference na objekty jsou alokovány ve spravované haldě, což umožňuje automatické správě paměti sledovat všechny reference na objekty. Když aplikace vytvoří referenci na typ, je pro ni přidělena výchozí adresa a při další alokaci se objektu přidělí adresa za ní následující.

Alokace objektů na spravované haldě je rychlejší než na nespravované, protože CLR alokuje paměť pro objekt pomocí přidělení hodnoty ukazateli.

Uvolňování paměti. Uvolnění paměti programu probíhá v tzv. "*garbage collection*". U každého objektu dojde k detekci, zda je alokovaný objekt dále využíván programem, pokud ne, uvolní přidělené paměťové místo pro něj vyhrazené. Detekce je založena na tzv. kořenových referencích (statické pole, lokální proměnné, CPU registry ...), tedy zda pro daný objekt existuje kořenová reference.

Automatické uvolňování paměti může být buď kompaktní nebo nekompaktní formou. Kompaktní uvolňování znamená, že neuvolněné objekty v paměti jsou přesunuty tak, aby adresy těchto objektů byly seřazené a nebyly mezi nimi žádné mezery po uvolněných objektech. Tento proces vede ke kompaktnímu uspořádání paměti. Nekompaktní uvolňování znamená, že adresa neuvolněných objektů zůstává stejná, i když jsou některé objekty uvolněny. Nekompaktní metoda se používá tehdy, když se uvolní pouze malé množství objektů.

Automatické uvolnění paměti je spuštěno za následujících podmínek:

- Systém má malé množství fyzické paměti, na což může upozornit operační systém.
- Paměť, která je alokována na spravované haldě, převyšuje přijatelnou hranici. Hranice se v průběhu vykonávání programu postupně může lišit.
- Automatické uvolnění paměti může být vyvoláno funkcí `GC.Collect()`. Ve většině případů není nutné tuto funkci explicitně volat, využívá se tak spíše pro speciální případy (např. optimalizace) a testování.

Automatické uvolnění paměti je zaměřeno primárně na uvolnění krátkodobých objektů. Pro zrychlení tohoto procesu bylo v CLR zavedeno rozdělení do tzv. generací. Generace je zásobník referencí na alokované objekty. Generace jsou rozděleny podle toho, jaké reference na objekty drží. Díky generacím se dokáží oddělit krátkodobé objekty od dlouhodobých a automatické uvolnění paměti tak může probíhat daleko efektivněji. Automatické uvolnění paměti totiž nemusí procházet všechny alokované objekty, ale pouze ty ve vybraných generacích. Nově alokované objekty jsou ukládány do nulté generace.

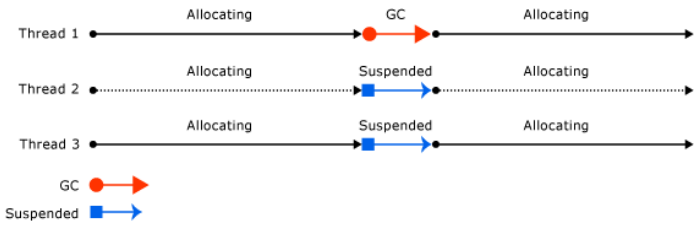
Automatické uvolnění paměti si zvolí generaci a z referencí vyhodnotí, které objekty jsou stále dosažitelné, pro nultou a první generaci se neuvolněné objekty přesunou na následující generaci [21]. Jednotlivé generace jsou detailněji popsány níže.

Generace 0. Nejmladší generace určená převážně pro krátkodobé objekty (např. dočasná proměnná). Tato generace je nejčastěji volána automatickým uvolněním pamětí. Při zaplnění generace se vykoná automatické uvolnění paměti. Objekty, které přežily uvolnění paměti, se přesunou na generaci 1.

Generace 1. Slouží jako vyrovnávací paměť mezi nultou a druhou generací. Obsahuje krátkodobé i dlouhodobé objekty. Funguje na stejném principu jako nultá generace, tedy po vykonání uvolnění se neuvolněné objekty přesunou na druhou generaci.

Generace 2. Obsahuje dlouhodobé objekty programu. Při vykonávání uvolnění již nemůže přesunout žijící objekty na další generaci, musí tedy postupně vykonat uvolňování objektů a poté seřadit ty, co přežily uvolnění. Tento proces je ze všech generací nejpomalejší, ovšem dělá se nejméně často. Tato generace slouží také pro uvolnění objektů z velké haldy objektů.

Při spuštění automatického uvolnění paměti ve vícevláknových programech je určeno jedno vlákno, které provádí automatické uvolnění a všechna ostatní vlákna jsou pozastavena. Obrázek 5.2 tento postup ilustruje.



Obrázek 5.2: Ilustrace automatického uvolnění paměti ve vícevláknových programech [21].

Nespravované zdroje (tzv. "*unmanaged resources*") jsou jediné zdroje paměti, při kterých se nemůžeme spolehnout na automatickou správu paměti. Většinou se jedná o typy, které zapouzdřují některé zdroje operačního systému. Je proto doporučeno, implementovat pro nespravované zdroje metodu `Dispose()`, která provede úklid objektu v případě, že jej programátor již nepotřebuje.

5.2 Component Object Model (COM)

Component Object Model (COM) je binární rozhraní pro softwarové komponenty vyvíjené společností Microsoft. COM se využívá pro interní komunikaci mezi více procesy tak, že tyto procesy navenek působí jako jeden proces. COM je nezávislý na jazyku implementace a jedinou podmínkou pro jeho fungování je nutnost, aby jazyk uměl vytvářet strukturu ukazatelů a uměl volat funkce přes ukazatele (ať už implicitně nebo explicitně).

COM objekt se vytváří jako instance třídy, která v sobě obsahuje pouze virtuální členy funkcí. Tyto funkce se implementují jako zděděné funkce rozhraní, které jej přepisují a mají vlastní implementaci. Při implementaci tohoto rozhraní musíme zahrnout implementaci všech funkcí, i těch, které nechceme implementovat. Pro tyto funkce se používá implementace ve formě jednoduchého návratového příkazu [19].

COM třídu je poté nutné identifikovat klíčem CLSID (tzv. "*Class ID*"). CLSID je řetězová reprezentace klíče, díky které se na COM rozhraní dokážeme připojit. Řetězová reprezentace se skládá ze 32 souvislých číslic v šestnáctkové soustavě. Pro vygenerování nového CLSID můžeme použít funkci `CoCreateGuid` [17].

IUnknown rozhraní. Jedná se o rozhraní, ze kterého všechny COM rozhraní dědí. Obsahuje základní operace COM pro polymorfismus a správu životnosti její instance. Poskytuje tři základní funkce – `QueryInterface`, `AddRef` a `Realease`.

`QueryInterface` je funkce, která provádí polymorfismus pro COM. Zjišťuje, zda v běhu programu COM podporuje požadované rozhraní. Tuto informaci poskytuje v ukazateli `ppvObject`. Pokud je ukazatel vrácen s hodnotou `NULL`, znamená to, že se požadované rozhraní nepodařilo načíst.

Životnost instance COM rozhraní je manipulována metodami `AddRef` a `Realease`. `AddRef` funkce přičítá počet referencí na COM objekt a `Realease` odečítá počet referencí na COM objekt. Pokud se počet referencí dostane na hodnotu nula, dojde k uvolnění instance COM objektu, protože ji už nemá kdo používat [25].

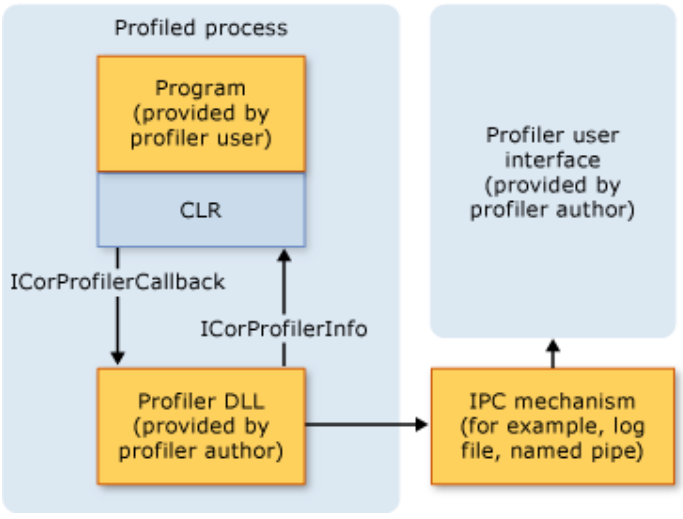
Klient/Server model. COM klient může být jakýkoliv kód, nebo objekt, který získá ukazatel na COM server. Server je potom rozhraní, které klienta dokáže obsloužit tím, že poskytne implementaci požadovaného rozhraní od klienta. Klient se dotazuje serveru a spojením mezi nimi se vytvoří instance COM objektu, přes kterou probíhá komunikace.

Existují dva hlavní typy serverů, a to buď v procesové, nebo mimo procesové komunikaci. V procesové jsou servery implementovány jako dynamicky linkovaná knihovna (DLL) a v mimo procesové jako spustitelný soubour (EXE).

5.3 CLR Profilovací aplikační rozhraní (Profiling API)

CLR umožňuje nahlédnout do své *runtime* exekuce, a to pomocí zabudovaného profilovacího rozhraní. Toto aplikační rozhraní se dá využít pro profilování .NET programů. Jedná se o COM rozhraní, které dokáže CLR komunikovat přes model klient/server, kde CLR je v pozici volajícího klienta a *profiler* v pozici COM serveru.

Profilovací aplikační rozhraní je implementováno autorem *profileru* jako COM server, který musí být ve společné procesové komunikaci, tedy v dynamicky linkované knihovně (DLL). Jiné typy COM serverů nejsou podporovány. Uvnitř DLL *profileru* je potom implementováno rozhraní `ICoreProfilerCallback`. Toto rozhraní slouží pro komunikaci CLR do DLL *profileru*, konkrétně pro zasílání generovaných událostí, které se staly uvnitř CLR. CLR profilovací rozhraní nabízí i druhé rozhraní s názvem `ICoreProfilerInfo`, které slouží pro komunikaci z DLL *profileru* do CLR pro poskytnutí informace o stavu profilovaného programu. Obě tyto rozhraní jsou již delší dobu součástí CLR a s postupným přidáváním nové funkcionality došlo ke vzniku nových verzí rozhraní, kdy každá nová verze dědí z verze předchozí. V době psaní této práce se můžeme setkat s `ICoreProfilerCallback10` a `ICoreProfilerInfo13`. Obrázek 5.3 ilustruje propojení mezi DLL *profilerem*, CLR a programem, který bude profilován [24].



Obrázek 5.3: Schématikcá ilustrace *profileru* DLL spolupracujícího s profilovaným programem a CLR [24].

Profilovací aplikační rozhraní nabízí rozšířenou sadu možných funkcí, kterými se dají sledovat různé události generované CLR. A tak i navzdory názvu „*profilovací aplikační roz-*

hraní“ je možné jej využít i pro jiné účely než profilování (např. ladění, či tvorbu různých diagnostických nástrojů). Dále také nabízí možnost vlastní instrumentace kódu spravovaného jazyka MSIL.

Pomocí CLR profilovacího aplikačního rozhraní můžeme zachytit tyto události:

- Spuštění a ukončení *runtime* exekuce.
- Vytvoření a ukončení aplikační domény.
- Načítání sestavení.
- Načítání modulů.
- *Just-in-time* (JIT) kompilace a *code-pitching*.
- Načítání tříd.
- Vytvoření a destrukce vláken.
- Vstupy a výstupy funkcí.
- Výjimky.
- Přechody mezi spravovanou a nespravovanou exekucí kódu.
- Přechody mezi odlišným *runtime* kontextem.
- Informace o pozastavení *runtime* exekuce.
- Informace o *runtime* paměťové haldě a automatickém uvolnění paměti.

Spuštění profilování probíhá na straně CLR, kdy pomocí nastavených proměnných prostředí se CLR dokáže připojit k DLL *profileru* a komunikace mezi nimi může začít. Tyto proměnné prostředí jsou:

- `CORECLR_ENABLE_PROFILING` – nastavení pro povolení nebo zakázání profilování, nenulová hodnota znamená povolení.
- `CORECLR_PROFILER` – jednoznačné CLSID pro připojení se k DLL *profileru*.
- `CORECLR_PROFILER_PATH` – cesta k DLL *profileru*.

Tyto proměnné jsou kompatibilní pouze s .NET Core platformou a pro starší verze .NET se liší.

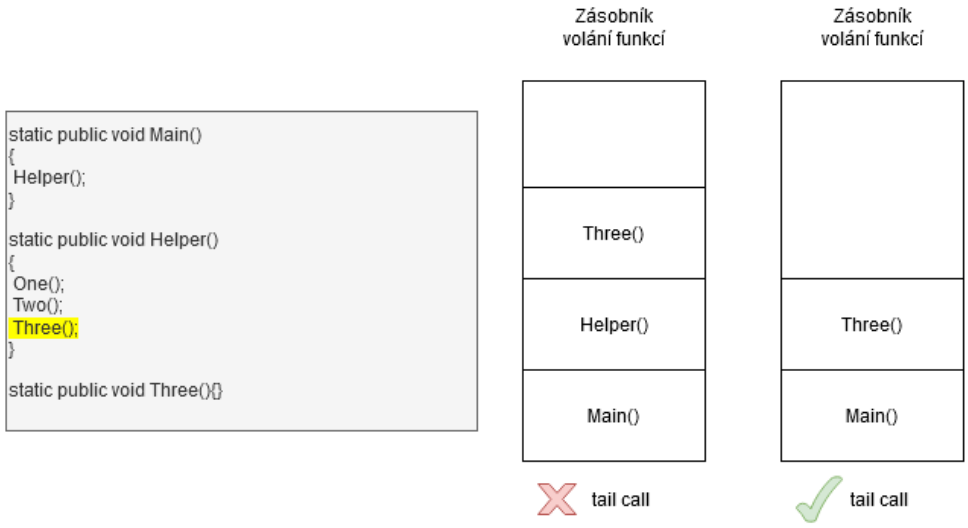
Identifikační klíče. Přes rozhraní `ICoreProfilerCallback` dostáváme různé notifikace od CLR, ty jsou většinou provázeny identifikačním číslem (ID). Tato čísla jsou unikátními klíči pro přístup k informacím o třídách, vláknech, funkcích, nebo modulech. V rámci implementace profileru se s těmito ID musí pracovat s opatrností – pokud by bylo třeba přistupovat k již neexistujícímu ID, mohlo by dojít k chybě a pádu programu. V implementaci je proto nutné pracovat s životností a stabilitou těchto ID. Ve většině případů je životnost určena událostmi pro načtení a zahození z CLR (např. `ClassLoadFinished` a `ClassUnloadFinished`), mezi těmito voláními tedy můžeme k ID přistupovat.

5.3.1 Zásobník volání funkcí

Zásobník volání funkcí, je zásobník který si uchovává informace o právě volaných funkcích. Každá funkce nebo metoda, která se volá v CLR, se přidá na vrchol zásobníku volání. Vrchní položka na zásobníku reprezentuje aktuálně volanou funkci nebo metodu. CLR profilovací API nabízí dvě možnosti získávání informace o zásobníku volání funkcí.

Snímek zásobníku. První možností je získání zásobníku funkcí pomocí pořízení snímku (tzv. "*snapshot*"), který zachycuje stav zásobníku programu v konkrétním čase. Snímek je možné získat pomocí rozhraní `ICoreProfilerInfo`. Tento způsob se využívá tehdy, pokud chceme znát zásobník volání pouze v některých momentech, např. u vyhození výjimky v programu. Pro sledování zásobníku po celou dobu běhu programu není vhodný, jelikož jeho opakované volání je časově náročná operace, která by značně proces profilování zpomalila.

Stínový zásobník. Je zásobník, který po celou dobu programu kopíruje chování skutečného zásobníku funkcí. Využívá k tomu profilovací funkce `FunctionEnter`, `FunctionLeave` a `FunctionTailCall`. Pokud tedy *profiler* sleduje informace o vstupech a výstupech funkcí, může si v paměti udržovat přehled o zásobníku funkcí. Je vhodnější pro sledování zásobníku funkcí po celou dobu běhu programu.



Obrázek 5.4: Ilustrace použití CLR optimalizace *tail call*. Program se nachází ve funkci `Three()`, ale zásobník volání funkcí se liší v závislosti na *tail call* optimalizaci.

`FunctionTailCall` je profilovací funkce pro sledování tzv. "*tail call*" optimalizace, která je součástí CLR. CLR ji využívá pro ušetření místa na zásobníku. V některých případech, kdy funkce končí voláním další funkce, dojde k odstranění aktuálně prováděné funkce ze zásobníku. Z pohledu pozorovatele by se tedy mohlo zdát, že volání další funkce probíhá přímo z těla funkce, která volala aktuálně prováděnou funkci. Obrázek 5.4 s příkladem kódu tento princip vysvětluje [7].

Kapitola 6

Sběr profilovacích dat

Prvním krokem při implementaci *profileru* je zajištění sběru profilovacích dat. V naší práci budeme tento modul nazývat kolektor. Kolektor využívá CLR profilovací aplikační rozhraní, přes které se na CLR připojí a dokáže díky němu sledovat události generované C# (i .NET) programem. V této kapitole jsou popsány jednotlivé požadavky pro tento kolektor, dále jeho návrh a způsob implementace.

6.1 Analýza požadavků

V této sekci jsou popsány hlavní požadavky na výsledný kolektor. Ne všechny tyto požadavky je však možné splnit zároveň, jelikož se některé požadavky mohou navzájem vylučovat.

- **Množství dat**. Výsledné množství profilovacích dat by mělo být co nejmenší, ale zároveň dostatečné pro zajištění vysoké přesnosti.
- **Přesnost dat**. Kolektor by měl sbírat přesná data.
- **Malá režie**. Kolektor by měl způsobit co nejmenší zpomalení profilovaného programu.
- **Počet volání a čas funkcí**. Kolektorem by mělo být možné sledovat počet volání jednotlivých funkcí, včetně jejich času. Tyto metriky jsou popsány blíže v sekci 2.4.
- **Graf volání funkcí**. Při sběru informací o funkcích programu by kolektor měl mít informaci o pořadí volání funkcí a o rodičovské funkci (funkce, ze které byla funkce volána). Z výsledných profilů by mělo být možné sestavit graf volání funkcí.
- **Spotřeba paměti**. Kolektor by měl znát čas a velikost alokací objektu v programu. Dále by si měl uchovávat informace o provedených automatických uvolňováních paměti.
- **Rozšířitelnost pro jiné operační systémy**. Kolektor by měl být spustitelný alespoň v operačních systémech Windows a Linux. Pro splnění tohoto požadavku v jeho minimální podobě postačí, když budou návrh a implementace koncipovány tak, aby bylo možné zajistit plnou multiplatformost v budoucnu.

6.2 Návrh

Tato sekce se zaměřuje na průběh návrhu kolektoru. Jsou zde popsány jednotlivé diskutované kroky v průběhu řešení spolu s jejich výhodami, nevýhodami a problémy.

6.2.1 Způsob profilovaní

Díky CLR profilovacímu API je možné implementovat všechny tři způsoby profilovaní popsané v sekci 2.2. Pro splnění požadavků na přesnost dat a nízkou režii byl vybrán způsob sledování událostí (tzv. "*tracing*"). Tento způsob je možné implementovat v rámci CLR profilovacího API, pomocí rozhraní `ICoreProfilerCallback` a `ICoreProfilerInfo`, popsaných v sekci 5.3.

Výhody. Technikou sledování událostí je možné získat vysoce přesná data, na rozdíl od profilování vzorkováním. Navíc pouhým sledováním událostí z *runtime* exekuce získáváme data s menší režií, než například u instrumentačního *profileru*, kde dochází ke vkládání dodatečného kódu do sledovaného programu. Další výhodou sledování událostí je, že můžeme sbírat data o automatickém uvolnění paměti, což není možné zajistit pomocí jiných dostupných technik.

Nevýhody. Na úrovni sledování událostí nemůžeme sledovat metriky s nižší granularitou než volání funkcí. Nižší granularity by bylo možné dosáhnout pomocí CLR profilovacího API v kombinaci s dodatečnou instrumentací v rámci JIT kompilátoru. Instrumentační *profiler* by mohl být součástí navazující práce.

Problémy. Při sledování všech notifikací od CLR jsme zjistili, že množství dat, které CLR vygeneruje, je příliš velké. Především pak u notifikací o volání a ukončení funkcí. Tento problém se částečně vyřešil omezením načítaných sestavení, u kterých funkce sledujeme. Omezení se vztahuje buď na vynechání všech systémových sestavení, nebo je možné si nastavit parametr, které sestavení je žádoucí sledovat. Porovnání časové a paměťové režie návrhů s a bez tohoto omezení je v tabulce 6.1.

	Velikost dat	CPU čas	*Wall* čas	Počet funkcí
Profil bez omezení sestavení	199 MB	3.828 s	7.201 s	2 626
Profil s omezením sestavení	**95 MB**	2.426 s	2.711 s	**476**

Tabulka 6.1: Tabulka srovnání výsledků návrhů s a bez omezení sledovaných sestavení. Omezení se vztahuje pouze na systémová sestavení. Profilovaným programem je `CacheManagerExample`, popsán v sekci experimenty 8.1.

6.2.2 Způsob ukládání nasbíraných dat

Způsob ukládání nasbíraných dat z profilování se ukázal jako klíčový pro splnění požadavků ze sekce 6.1. Jednotlivá řešení se liší na základě metrik, které profilování zachycuje.

Sledování funkcí. V prvním návrhu kolektoru bylo zvoleno řešení vypisovat generované události hned při jejich zavolání do externího souboru. To se ukázalo jako nevhodný způsob hned z několika důvodů:

1. Profilovaný program se značně zpomalil.
2. Kolektor generoval velký objem dat.
3. Data vyžadovala dodatečné zpracování.

Z těchto důvodů jsme ve výsledném řešení zvolili jiný návrh. Ten si díky své zabudované logice uchovává nasbíraná data v paměti kolektoru a při skončení profilování vypíše zpracovaná data do souboru. Tato data sestávají z informací o počtu volání funkcí, jejich doby běhu v CPU a *Wall* čase, nebo informací o pořadí volání funkcí. Díky tomu můžeme následně sestavit graf volání funkcí. Tato logika je blíže vysvětlena v sekci 6.3.3.

Sledování spotřeby paměti. Sledování spotřeby paměti také využívá paměť kolektoru. V kolektoru jsou sledovány jednotlivé události o alokaci objektů v profilovaném programu, tato data se ukládají do paměti a na konci profilování se vypíší do externího souboru. Data uchovávají informace o čase, velikosti a typu alokace objektu. V kolektoru je dále zabudovaná logika pro zjištění životnosti jednotlivých objektů – po automatickém uvolněním paměti se kolektor podívá, které alokované objekty toto automatické uvolnění přežily. Objekty, které nepřežily jsou v kolektoru označeny jako uvolněné. Díky tomu získáme informaci o životnosti objektů. Implementace spotřeby paměti je blíže vysvětlena v sekci 6.3.4.

Výhody. Hlavní výhodou využití paměti kolektoru je především výrazné snížení režie kolektoru a získání již zpracovaných dat. Rozdíl doby běhu profilovaného programu pro jednotlivé varianty návrhu je přibližně 85% v celkovém čase. Srovnání měření je ilustrováno tabulkou 6.2.

	Celkový CPU čas	Celkový *Wall* čas
Profil bez využití paměti kolektoru	17.453 s	21.302 s
Profil s využitím paměti kolektoru	**2.426 s**	**2.711 s**

Tabulka 6.2: Tabulka srovnání časové režie návrhu s a bez využití paměti kolektoru. Profilovaným programem je `CacheManagerExample`, popsán v sekci experimenty 8.1.

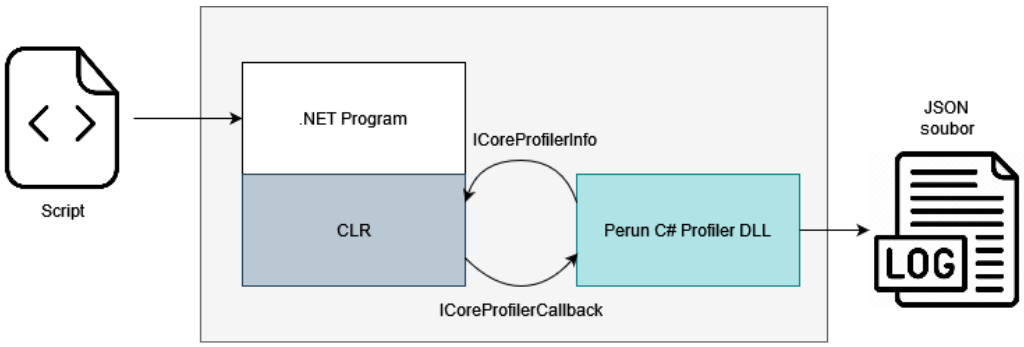
Nevýhody. Kolektor je omezen dostupnou pamětí, kterou může v průběhu profilování využít. Použití kolektoru tedy není doporučeno pro rozsáhlé projekty, u kterých by se mohlo stát, že dostupná paměť může být vyčerpána. Vyřešení tohoto problému by mohlo být součástí navazující práce, kde by bylo možné například využít sdílené systémové roury, kterými by kolektor odesílal data druhému programu, určenému pro zpracování těchto dat.

Problémy. Při návrhu logiky pro zjištění životnosti objektů v paměti bylo zjištěno, že tato operace značně zvýšila režii profilování. Kolektor se totiž dotazuje každého objektu, který si uložil v paměti, zda automatické uvolnění přežil. Zvýšená režie také znemožnila možnost sledování času automatického uvolnění paměti, protože je touto logikou výrazně zpomalena. Tento problém je vyřešen přidáním parametru `PROFILER_ENABLE_DEALLOC` do nastavení kolektoru, který zjišťování životnosti aktivuje pouze v případě potřeby.

6.2.3 Navrhované řešení

Navrhované řešení je tedy kolektor, který pomocí techniky sledování událostí s využitím CLR profilovacího aplikačního rozhraní dokáže profilovat výkonnostní metriky jako je např.

doba běhu funkcí, alokace a uvolnění objektů v paměti, informace o automatickém uvolnění paměti, nebo celkové doby běhu programu. Pracovní postup kolektoru je vysvětlen pomocí ilustrace 6.1.



Obrázek 6.1: Ilustrace pracovního postupu kolektoru. Skript nastaví proměnné prostředí pro povolení profilování a nastavení módu profilování podle parametrů uživatele. Dále skript spustí .NET program, který načte CLR. CLR se pak díky proměnným prostředí připojí a začne komunikovat s kolektorem. Výsledný profil je uložen do JSON souboru.

Přes vstupní skript si uživatel může definovat několik parametrů nastavení kolektoru.

- `PROFILER_MODE` – nastavení režimu, ve kterém bude profilování probíhat.
  - Kombinace spotřeby paměti se sledováním funkcí (díky spojení informace o alokaci a právě běžící funkci víme, v jaké funkci alokace proběhla).
  - Sledování spotřeby paměti (CPU a *Wall* čas alokace objektů, informace o automatickém uvolnění paměti, včetně cílové generace, velikost alokovaných objektů a názvy typů objektů).
  - Sledování doby běhu funkcí (CPU a *Wall* čas volání a návratu z funkce, název funkce a pořadí, ve kterém byly funkce volány).
- `PROFILER_PATH` – adresář pro uchovávání výsledných profilů.
- `PROFILER_ENABLE_ASSEMBLIES` – omezení sledování funkcí na povolené sestavení, které chceme profilovat.
- `PROFILER_ENABLE_DEALLOC` – aktivace zjišťování uvolněné paměti.

Vstupním skriptem se také nastaví proměnné prostředí popsané v sekci 5.3. Díky nim CLR naváže spojení s kolektorem. Kolektor poté využívá zabudovanou logiku pro sběr a agregaci dat a data následně vypisuje do souboru s notací JSON.

6.3 Implementace

V této sekci jsou podrobně popsány jednotlivé části implementace kolektoru.

6.3.1 Implementační jazyk

Kolektor musí implementovat COM rozhraní, abychom zajistili mezi procesovou komunikaci mezi CLR a kolektorem. COM rozhraní může být implementováno v různých jazycích tak, jak je to popsáno v sekci 5.2. Tyto jazyky ovšem nesmějí být součástí .NET spravovaných jazyků. Spravované jazyky totiž samy běží pod exekucí CLR a to by znamenalo, že by kolektor profiloval i sám sebe. CLR by totiž načetl kolektor, který by se sám připojil taky na CLR [9].

Jazyk C# s Native AOT technologií. Původní návrh implementace počítal s využitím nové technologie `Native AOT` (*Ahead of Time*) kompilace, která byla přidána do verze .NET 7. To by umožnilo implementovat samotný kolektor v jazyce C#. Tato technologie slouží k předkompilaci programu do nativního kódu ještě před začátkem spuštění programu, díky čemuž je možné se vyhnout zmíněným problémům profilováním kolektoru samotného. Při testování prototypu jsme ovšem odhalili problémy s nedostatečnou podporou pro operační systém Linux, kde kolektor skončil chybou CLR. Dále by implementace zahrnovala značné množství práce navíc kvůli zapouzdření a abstrakci nad COM rozhraním.

Jazyk C++. Jazyk C++ se ukázal jako vhodnější pro implementaci, jelikož je v něm implementována většina COM rozhraní. Dále je většina dostupné dokumentace i existujících implementací *profilerů* využívajících CLR profilovací API v tomto jazyce. Jeho nevýhoda spočívá v rozdílné podpoře knihoven pro různé operační systémy, nebo komplexnější logika pro manuální správu paměti. Ve výsledku však byl jazyk C++ zvolen jako vhodnější.

6.3.2 Načtení kolektoru do procesu CLR

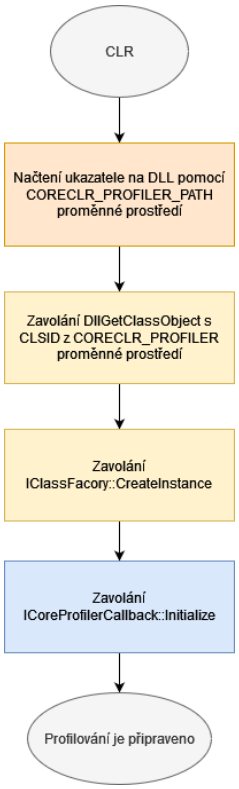
První částí kolektoru je implementace COM serveru v podobě DLL souboru, tak jak je popsáno v sekcích 5.2 a 5.3. Tato část byla inspirována a z části převzata z přednášky Pavla Yosifoviche[1] a jeho implementace prototypu *profileru*[2]. Program je nejprve načten do `DllMain()` funkce. Ta zajišťuje, že se DLL načte do stejného procesu v jakém běží CLR, a že při ukončení procesu dojde k jeho uvolnění. Po načtení DLL si CLR načte CSLID klíč, přes který se připojí na třídu `CoreProfilerFactory`, která implementuje rozhraní `IClassFactory`. Třída slouží především jako mezikrok k vytvoření instance COM ve funkci `CreateInstance`. Ve funkci se vytvoří instance třídy `CoreProfiler`, což je hlavní třída našeho kolektoru. Třída dědí z rozhraní `ICoreProfilerCallback8`, kterou používá CLR pro volání funkcí profilovacího aplikačního rozhraní, popsaného v sekci 5.3. Ilustrace procesu napojení se na CLR a vytvoření instance COM je na obrázku 6.2.

Ve třídě `CoreProfiler` je poté implementace tří hlavních funkcí COM serveru vycházející z `IUnknown` rozhraní. Funkce `AddRef` a `Realase` slouží pro počítání počtu referencí na COM objekt, přičemž tento počet je uložen v proměnné třídy `_refCount`. Třetí funkce `QueryInterface` slouží ke kontrole, zda bylo načteno požadované rozhraní `ICoreProfiler-Callback8`. Kolektor poté načte funkci `Initialize`, kterou spustí proces profilování.

[1] Pavel Yosifovich — Writing a .NET Core cross platform *profiler* in an hour https://www.youtube.com/watch?v=TqS4OEWn6hQ&list=PLWWfkA8puZwypfHDokt5B6CSByKy_ZEvv&index=10

[2] Git DotNext Moscow 2019 https://github.com/zodiacon/DotNextMoscow2019

Metoda Initialize. Jedná se o metodu, která inicializuje profilování. Zde dojde k vytvoření instance rozhraní `ICorProfilerInfo8`, popsané v sekci 5.3. Díky tomuto rozhraní dokáže nyní kolektor komunikovat s CLR. Na to, abychom od CLR přes rozhraní `ICoreProfilerCallback8` dostávali notifikace na události, je potřeba nastavit masku definující, které události nás zajímají. Masku lze nastavit pomocí funkce `SetEventMask`. V rámci navrhovaného řešení popsaného v sekci 6.2.3 podporuje kolektor 3 módy nastavení. Zvolený mód je zjištěn čtením nastavené proměnné prostředí `PROFILER_MODE`, a na základě zvolené možnosti je vybrána maska.



Obrázek 6.2: Ilustrace procesu napojení se na CLR a vytvoření instance COM objektu pro profilování.

6.3.3 Sledování funkcí

Sledování funkcí je v *profileru* implementováno pomocí stínového zásobníku volání funkcí, který je popsán v sekci 5.3.1. Funkce pro sledovaní volání a návratu z funkcí ovšem nejsou dostupné v samotném rozhraní `ICoreProfilerCallback8`, ale jsou pojaté jako globální statické funkce. Tyto funkce musí mít v *profileru* vlastní implementaci, a to v *assembler* kódu. Důvodem je závislost implementace na jednotlivých architekturách počítače. To by v CLR vyžadovalo zbytečnou logiku navíc, která by jej zpomalila a implementaci tedy přenechává kolektoru.

V rámci implementace jsou tyto funkce implementovány s tím, že jejich prolog a epilog je napsán v *assembler* kódu. V těle těchto funkcí je volání funkce jazyka C++, ve kterých tyto události můžeme zpracovat. Implementace prologu a epilogu těchto globálních funkcí byla převzata z ukázkového prototypu `Profiling API Samples for CoreCLR`[1].

Při sledování funkcí je hlavním problémem velký počet volání některých funkcí, což způsobuje zpomalení programu a zvýšení výsledného množství dat. Proto byly v kolektoru implementovány optimalizace, které mají za úkol snížit vliv tohoto problému. Prvním z nich je implementace funkce `FunctionIdMapper` v rámci rozhraní `ICoreProfilerInfo8`. Funkce slouží jako vyrovnávací paměť informaci o funkcích. Ve funkci se dotážeme na potřebné údaje (např. název funkce) a data se uloží do třídy `FunctionInfo`, která je následně přidána do mapy `m_functionMap`. Při následném volání funkce `FunctionEnter` se kolektor nemusí znovu dotazovat na údaje o funkci a místo toho zjistí potřebné údaje z třídy `FunctionInfo`, kterou získá z vyrovnávací paměti pomocí ID klíče funkce. Tato optimalizace cílí zejména na ušetření času při opakovaném volání funkcí.

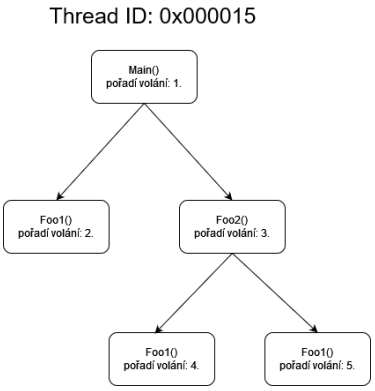
Další optimalizací kolektoru je zavedení logiky pro ukládání naměřených dat nejdříve do paměti kolektoru podle návrhu v sekci 6.2.2. Pro implementaci této logiky byla vytvořena třída `FunctionClass` a mapa `m_activeFunctionInThread`. V mapě jsou uložena jednotlivá spravovaná vlákna a s nimi ukazatel na třídu `FunctionClass` reprezentující právě aktivní funkci v tomto vlákně. Ve třídě `FunctionClass` je potom ukazatel na předchozí funkci a také

[1]Github Profiling API Samples for CoreCLR https://github.com/microsoft/clr-samples

seznam ukazatelů na funkce, které byly volány z této funkce. Tato logika, spolu s uchováváním si informací o pořadí funkcí, umožňuje kolektoru neustále sledovat zásobník volání funkcí. Z výsledných profilů je poté možné sestavit si graf volání funkcí, ilustrovaný obrázkem 6.3.

Poslední optimalizací je filtrování funkcí podle sestavení, tak jak bylo popsáno v sekci 6.2.3. Tato optimalizace je implementována s využitím funkce `FunctionIdMapper`, která jako návratovou hodnotu vrací `ULONG` hodnotu, která bude dál v profilování jediným identifikátorem pro funkce se stejným názvem a parametry. V této funkci zjistíme, z jakého sestavení funkce pochází, a pokud zjistíme, že se nejedná o funkci ze sledovaného sestavení, je vrácena nulová hodnota jako identifikátor funkce. Funkce s nulovým identifikátorem budou v kolektoru dále ignorovány.

Následuje popis jednotlivých profilovacích metod určených ke sledování funkcí profilovaného programu.



Obrázek 6.3: Ilustrace grafu volání.

FunctionEnter. Volána při volání funkce profilovaného programu. Nejdříve z mapy `m_functionMap` získá ukazatel na třídu `FunctionInfo`, ve které jsou informace o funkci. Dále se dotáže na to, ve kterém spravovaném vlákně se nachází. Podle vlákna tak může zjistit z mapy `m_activeFunctionInThread`, která funkce jej volala (rodičovská funkce). Tato funkce se přidá do objektu rodičovské třídy jako volaná funkce a vytvoří se třída `FunctionClass` pro volání funkce, do této třídy se uloží informace o časových značkách a přiřadí se k ní ukazatel na třídu `FunctionInfo`. V mapě `m_activeFunctionInThread` se poté změní ukazatel na třídu právě aktivní funkce.

FunctionLeave. Volána při návratu z funkce profilovaného programu. Funkce zjistí právě aktivní funkci programu a ukončí ji. Do `m_activeFunctionInThread` poté uloží objekt rodičovské funkce.

FunctionTailCall. Volána pokud CLR uplatní optimalizaci *tail call*, která je popsána v sekci 5.3.1. V kolektoru by bylo velice obtížné tuto logiku implementovat bezpečným způsobem. Může se totiž stát, že *tail call* se uplatní v rámci pomocné nativní funkce, pro kterou nebyl zaznamenán vstup v podobě volání `FunctionEnter`. To by způsobilo rozbití logiky sledování funkcí. `FunctionTailCall` je tedy v kolektoru implementován stejným způsobem jako volání funkce `FunctionLeave`. Funkcionalita je zde uvedena pro úplnost, pokud by naměřené výsledky neodpovídaly zdrojovému kódu profilovaného programu.

6.3.4 Spotřeba paměti

Pro sledování alokací objektů v programu slouží volání funkce `ObjectAllocated`. Pro každý alokovaný objekt je vytvořena instance třídy `ObjectClass` a následně uložena do mapy `m_objectsAlloc` s klíčem identifikátoru objektu. Díky tomu si kolektor uchovává tyto objekty stejně jako profilovaný program. Při nastavení módu pro sledování funkcí i spotřeby paměti, se jednotlivé alokace dokáží přiřadit k funkcím programu. Ovšem při nastavení na

omezení sestavení mohou být tyto alokace přiřazeny do funkcí, ve kterých ve skutečnosti nevznikly (např. alokace vznikne v systémové funkci, kterou nesledujeme).

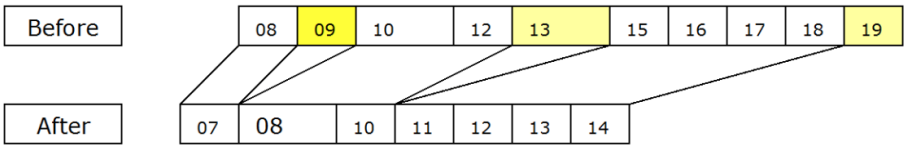
Další metrikou při sledování paměti je v kolektoru sledování automatického uvolnění paměti. To můžeme sledovat díky funkcím `GarbageCollectionStarted` a `GarbageCollectionFinished`. Z funkce `GarbageCollectionStarted` můžeme i vyčíst, jaká generace automatického uvolnění paměti je zrovna zkoumána. Automatické uvolnění paměti a její generace jsou vysvětleny v sekci 5.1.2.

Uvolňování paměti

Sledování uvolnění paměti je v kolektoru implementováno pomocí dvou funkcí, konkrétně `MovedReferences` a `SurvivingReferences`. Funkce jsou volány v závisloti na tom, jaké automatické uvolnění paměti se vykonalo.

SurvivingReferences. Funkce, která je volána po nekompatkním automatickém uvolnění paměti, popsaném v sekci 5.1.2. Funkce nám neposkytuje přesné informace o tom, které objekty přežily uvolnění, ale dává nám intervaly adres, mezi kterými se nachází objekty, které nebyly uvolněny. Kolektor tedy iteruje sledované objekty, uložené v mapě `m_objectsAlloc` a pokud se jeho identifikátor nachází v tomto rozmezí, můžeme jej na konci automatického uvolnění paměti označit za stále alokovaný. Zbylé objekty vyhodnotíme jako uvolněné a přesuneme je do seznamu `m_objectsDeAlloc`.

MovedReferences. Funkce, která je volána po kompaktním automatickém uvolnění paměti, popsaném v sekci 5.1.2. Funguje na podobném principu jako `SurvivingReferences`, pouze s tím rozdílem, že mění adresu pro alokovaný objekt. Adresa objektu odpovídá jejímu identifikátoru. Každá třída objektu si tedy tuto hodnotu musí změnit. Obrázek 6.4 popisuje tento proces změn adres objektů, které zůstaly alokované i po automatickém uvolnění paměti.



Obrázek 6.4: Ilustrace kompaktního automatického uvolnění paměti [24].

6.3.5 Hlavní koncepty kolektoru

V této sekci je popsána implementace funkcí a konceptů, které jsou využity napříč různými módy sběru dat.

Shutdown. Metoda volaná na konci procesu profilování. V této metodě všechna data serializujeme do JSON formátu a vypíšeme je do souboru stanoveným v proměnné prostředí `PROFILER_PATH`. Po skončení výpisu by se měla veškerá paměť kolektoru uvolnit a skončit proces profilování.

Kritické sekce kolektoru. Jelikož kolektor je vícevláknová meziprocesová aplikace, je zapotřebí synchronizovat kritické sekce částí kolektoru. Příkladem kritické sekce je výpis dat

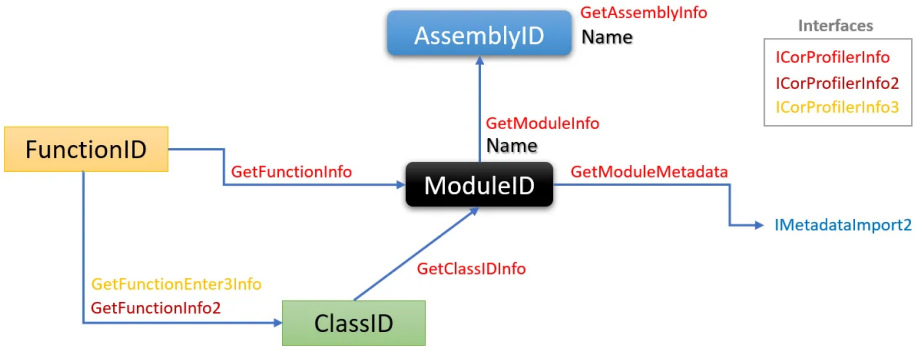
do souboru, kdy při přístupu více vláken k jednomu souboru mohl znamenat chybu a pád programu. Pro předejití těchto chyb byla v implementaci přidána třída `Mutex` a `Autolock`. Kritické sekce jsou poté zamknuty funkcí `locker()`.

Sledování vláken. .NET podporuje vícevláknové programy (viz sekce 5.1.1) a je proto důležité, aby k tomu byl uzpůsoben kolektor. Při profilování si tedy kolektor u každé metriky uchovává i informaci, v jakém spravovaném vlákně k události došlo. K tomu se v kolektoru využívá funkce `GetCurrentThreadID`, která je součástí rozhraní `ICoreProfilerInfo`.

Názvy funkcí a tříd. Pro získání názvů funkcí, tříd, modulů, nebo sestavení, slouží rozhraní `ICoreProfilerInfo`. To umožní nahlédnout do metadat CLR, které si tyto údaje uchovávají ve svých tabulkách. Obrázek 6.5 ilustruje, kterými funkcemi se lze dostat k jakým informacím. Tyto funkce jsou pak volány například z funkcí `GetName()` nebo `GetTypeName()`.

Čas CPU a Wall. Tato metrika se sbírá pro všechna profilovaná data. Kolektor ji získává z funkcí `GetCpuTime()` a `GetWallTime()`, které jsou součástí třídy `OS`. Obě funkce mají rozdílnou implementaci pro operační systém Windows a Linux.

Logování. *Logování* dat kolektoru je zajištěno pomocí třídy `Logger`. Ta si při inicializaci vytvoří JSON soubor s názvem ve formátu *PerunCSharpProfiler_YYYY-MM-DD-HH-MM-SS* do cesty získané z parametru *PROFILER_PATH* popsaném v sekci 6.2.3. Pro serializaci do JSON formátu není využívaná žádá knihovna, jelikož ta by kolektor značně zpomalila.



Obrázek 6.5: Jednotlivé funkce pro zjištění názvu funkcí, tříd, modulů a sestaveních CLR přes rozhraní `ICoreProfilerInfo` [30].

6.3.6 Podpora operačního systému Linux

Tato část byla inspirována a z části převzata z přednášky Pavla Yosifoviche[1] a jeho implementace prototypu *profileru*[2]. Pro splnění požadavku rozšiřitelnosti na operační systém Linux, uvedeného v sekci 6.1, je v kolektoru odlišná implementace pro jednotlivé operační systémy. Ta je dosažena pomocí direktiv #if, #elif, #else a #endif podporovaných v C++ (např. #ifdef _WINDOWS pro operační systém Windows). Díky direktivám si kolektor

[1] Pavel Yosifovich — Writing a .NET Core cross platform *profiler* in an hour https://www.youtube.com/watch?v=TqS4OEWn6hQ&list=PLWWfkA8puZwypfHDokt5B6CSByKy_ZEvv&index=10

[2] Git DotNext Moscow 2019 https://github.com/zodiacon/DotNextMoscow2019

v některých částech funkcí či třídách dokáže načíst odlišné knihovny a funkce pro jiné operační systémy.

Některé knihovny a rozhraní potřebné pro profilování jsou ovšem podporovány pouze na operačním systému Windows (např. COM rozhraní je čistě technologie na operačním systému Windows). Tento problém řeší v implementaci `Platform Adaptation Layer` (PAL). PAL je abstraktní vrstva mezi CLR a operačním systémem, která poskytuje kolekce rozhraní založených na Win32.

6.3.7 Známá omezení

Profilování C# programů je technicky složité a rozsáhle téma. V této práci nebylo tedy možné pokrýt celou problematiku. Zde proto uvádíme aktuálně známé omezení a nedostatky kolektoru.

Profilování rozsáhlých programů. Kolektor v tuto chvíli není vhodný pro profilování velkých projektů. Na základě návrhu si totiž data ukládá do paměti kolektoru a až na konci profilování dojde k jejich uložení do perzistentní paměti. Při profilování velkých projektů se tedy může stát, že kolektoru dojde dostupná paměť.

Podpora operačního systému Linux. Ačkoliv implementace by měla fungovat i pro operační systém Linux, nebyla v rámci této práce řádně otestována kompatibilita a nemůžeme tedy v tuto chvíli jasně říci, že by kolektor fungoval i na operačním systému Linux. Doplnění chybějící podpory je možné řešit v rámci navazující práce.

Časové značky CPU. CPU čas se v kolektoru pro operační systém Windows, získává pomocí funkce `GetProccessTime()`, která nezískává přesná data o CPU čase, ale pouze agregované hodnoty. Je tedy vhodné v tuto chvíli analyzovat profily v přesném *Wall* čase. Řešení tohoto problému by mohlo být získávání CPU času podle vláken programu.

Sledování spotřeby paměti. První omezení je v sledování uvolněných objektů po automatickém uvolnění paměti, kdy v rámci této práce nebyla ověřena korektnost této funkcionality, a proto je v tuto chvíli vhodné tento mód ponechat neaktivní. Druhým omezením je, že pro některé alokované objekty není načten jejich příslušný typ.

Kapitola 7

Vizualizace

V této kapitole je popsán návrh a implementace modulu sloužící pro interpretaci naměřených profilů – vizualizér. Ten slouží ke zpracování a interpretaci dat. Vizualizace jsou primárně určeny k manuální analýze naměřených dat pomocí uživatele. První část se věnuje návrhu a výběru vhodné interpretace profilů. Druhá část je zaměřena na implementaci tohoto modulu s popisem využitých knihoven.

7.1 Návrh

Vizualizace jsou vytvořeny v modulu, který má na vstupu nastavitelné parametry. Program nejdříve načte naměřený profil z kolektoru. Tento profil se v modulu nejdříve zpracuje a následně vizualizuje pomocí grafů. Jako návrh vizualizací byly vybrány dva grafy, prvním je tzv. "*Treemap*", který vizualizuje graf zanoření volání funkcí a druhý je bodový graf, který interpretuje primárně spotřebu paměti pomocí bodů, přičemž velikost těchto bodů je závislá na velikosti alokace.

Bodový graf (tzv. "*Scatter plot*"). Je graf, který zobrazuje v kartézských souřadnicích hodnoty dvou proměnných. Data jsou poté znázorněna jako množina bodů, kdy svislá osa určuje hodnotu jedné proměnné a vodorovná té druhé. Pro vizualizaci byla vybrána metrika alokování objektů v čase, kdy na ose y je n nejčastějších typů objektů a na ose x jsou jednotlivé alokace těchto typů v čase. Jednotlivé body pak odpovídají velikosti alokace v bajtech.

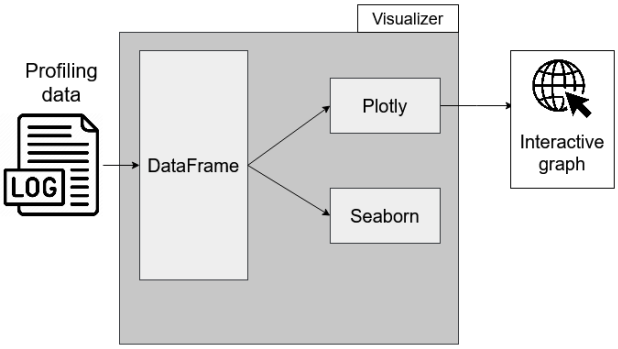
Treemap. Je graf zobrazení hierarchických dat pomocí vnořených obrazců, obvykle obdélníků. Graf zde slouží pro zobrazení metriky sledování funkcí, kde podle jednotlivých zanoření lze vidět jejich pořadí, ve kterém byly postupně volány. Každý obdélník tedy reprezentuje jednu volanou funkci. Při náhledu je zobrazeno více informací, jako jsou celková doba funkce, nebo její třída.

Graf je možné také rozšířit o metriku alokace objektů, kde jsou jednotlivé alokace asociovány s konkrétní volanou funkcí reprezentovanou jako obdélník. Tyto obdélníky jsou pak velikostně a barevně rozlišeny, podle velikosti alokací vykonaných v této funkci. To v analýze pomůže rychle odhalit, ve které funkci se alokuje nejvíce paměti.

7.2 Implementace

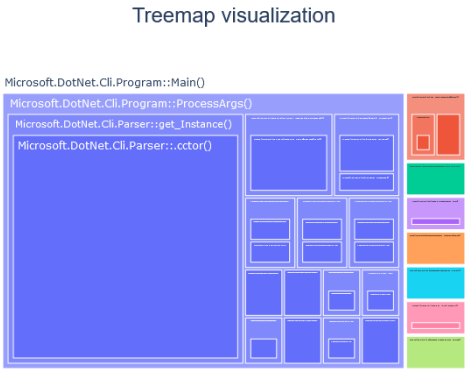
Modul je implementován v jazyce Python, dle zvyklostí a požadavků nástroje Perun. Pro jazyk Python navíc existuje široká škála knihoven pro vizualizace a jedná se tedy o jazyk vhodný k implementaci a prototypování různých způsobů vizualizace dat. Postup práce vizualizéru je ilustrován na obrázku 7.1.

Pro zpracování dat byla využita knihovna `Pandas`[1] s její datovou strukturou `Dataframe`. Nejdříve si modul pomocí JSON rozšíření načte profil, který poté vloží do datové struktury `Dataframe`. Pomocí této struktury lze dále tato data jednoduše zpracovávat a současně vizualizační knihovny jí podporují na vstupu.

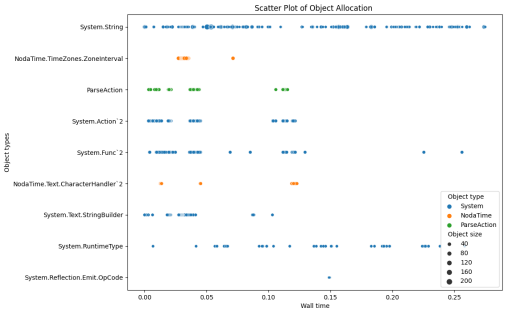


Obrázek 7.1: Ilustrace pracovního postupu vizualizačního modulu. Profil je nejdříve načten do `Dataframe` struktury a pomocí vybraných knihoven je vytvořena vizualizace. `Plotly` pro graf nabízí interaktivní prostředí ve webovém prohlížeči.

Seaborn. Pro vizualizaci bodového grafu byla použita knihovna `Seaborn`[1]. `Seaborn` je rozšířená knihovna založena na `matplotlib`. V nástroji Perun je pro řadu zavedených grafů již využívána a byla proto vhodným výběrem. Pro implementaci grafu je zde využita funkce `scatterplot()`. Pro lepší orientaci v grafu bylo implementováno její barevné rozlišení, podle typů sestavení. Implementovaný graf je ilustrován na obrázku 7.3.



Obrázek 7.2: Graf `Treemap` pro zobrazení volání funkcí.



Obrázek 7.3: Bodový graf pro zobrazení alokací objektů daných typů ve *Wall* čase.

Plotly. Pro implementaci druhého grafu `treemap` byla vybrána knihovna `Plotly`[2]. Ta je vhodná díky přímé podpoře tohoto typu grafu ve funkci `treemap()`, která dokáže data

[1]Pandas knihovna https://pandas.pydata.org/docs/index.html
[1]Seaborn knihovna https://seaborn.pydata.org/
[2]Plotly knihovna https://plotly.com/

vizualizovat jako interaktivní graf. Graf nabízí interaktivitu v podobě libovolného zanoření do stromové mapy. Současně se dá pořídit snímek obrázku z jakékoliv části grafu. Graf se zobrazí ve webovém prohlížeči a je tedy vhodný pro případné rozšíření nástroje Perun do webového rozhraní. Graf se dá dále využít i pro jiné metriky v nástroji Perun. Implementovaný graf je ilustrován na obrázku 7.2.

V rámci testování implementace se ukázalo, že načítání interaktivního grafu `Treemap` je pro velké profily paměťově značně náročné a graf je nepřehledný. Proto bylo implementováno nastavitelné uživatelské omezení, do jaké hloubky stromové mapy volání funkcí sledovat. Toto nastavení především zlepšilo vizualizaci a načítání grafu.

Kapitola 8

Experimenty

Tato kapitola demonstruje ověření funkcionality vytvořeného nástroje a představení jeho praktického využití pro analýzu C# programů. Nejdříve je popsáno jakým způsobem a v jakém prostředí byly experimenty prováděny, následně jsou popsány jednotlivé programy, se kterými byly experimenty provedeny. Je zde popsán vliv nástroje na profilovaný program. Nakonec jsou zdokumentovány jednotlivé experimenty s jejich vizualizacemi a závěry.

8.1 Způsob provedení experimentů

V této sekci je popsána metodologie pro všechny experimenty. Dále je zde popsáno prostředí, na kterém byly experimenty vykonány a krátce představeny programy, se kterými byly experimenty provedeny.

Metodologie experimentů. Každý experiment byl vykonán následujícím způsobem: bylo provedeno alespoň pět měření, z toho první dvě byly zahozeny (pro stabilitu výsledků). Následně pro metriky jako je doba běhu programu, nebo doba běhu funkcí, byla ze zbývajících profilů vybrána mediánová hodnota.

Specifikace počítače. Experimenty byly vykonány v prostředí počítače v následující specifikaci.

Model	`Lenovo Legion 5-15IMH05H`
Architektura	`x86_64`
CPU	`Intel(R) Core(TM) i5-10300H CPU @ 2.50GHz`
RAM	`16GB DDR4 SODIMM @ 3 200MHz`
OS	`Windows 10`

Vybrané testovací projekty. Pro vykonání experimentů byly vybrány tři projekty. Projekty jsou zde krátce popsány se vstupy, které se využily v experimentech. U každého popisu vstupu je také krátký přehled o jeho velikosti.

CacheManager	`Popis`	`CacheManager` je otevřená knihovna pro .NET, která poskytuje jednoduché a výkonné řešení pro práci s vyrovnávací pamětí. Podporuje různé techniky tzv. "*cachování*", jako je paměťové *cachování*, *cachování* do souborů, nebo vzdálené *cachování* [2]. Pro experimenty byl vybrán jako reprezentace středně velké otevřené knihovny.
	`Vstup`	Program určený pro experimenty byl převzat z ukázkové implementace použití knihovny[1].
	`Velikost`	Střední (390 funkcí, 2 vlákna, 500 000 volání funkcí, knihovna má ≈10 000 LoC).
	`Označení`	`CM`
	`Verze`	`.NET 6.0`
IronPython	`Popis`	`IronPython3` je otevřená implementace jazyka Python 3, která běží na platformě .NET. Umožňuje vývojářům psát kód v jazyce Python 3 a používat ho v prostředí .NET. Vývojáři tedy mohou využívat výhody Pythonu a platformy .NET zároveň [4]. Do experimentů byl vybrán jako reprezentace rozsáhlého projektu.
	`Vstup`	Program pro experimenty, je jednoduchá ukázka využití `IronPython3` v jayzce C# [2].
	`Velikost`	Středně velká (1 milión volání funkcí, 170 tisíc alokací objektu, projekt má ≈170 000 LoC)
	`Označení`	`IP`
	`Verze`	`.NET 6.0`
AvalonStudio	`Popis`	`AvalonStudio` je otevřené vývojové prostředí pro vývoj .NET aplikací [1]. Program byl do experimentů vybrán pro ukázání vysokého zatížení nástroje.
	`Vstup`	Spuštění a následné vypnutí vývojového prostředí.
	`Velikost`	Velká (6.5 miliónu volání funkcí, projekt má ≈190 000 LoC)
	`Označení`	`AS`
	`Verze`	`.NET 5.0`

Módy nastavení. *Profiler* podporuje nastavení několika módů. Jednotlivé experimenty se především liší použitím těchto módů. Zde jsou módy popsány s jejich označením, který je v popisu experimentů využíváno:

`noprof`	Spuštění programu bez profilování.
`all_trace`	Mód pro sledování alokace paměti v asociaci s funkcemi programu.
`func_trace`	Mód pro sledování pořadí a doby funkcí programu.
`mem_alloc`	Mód pro sledování alokací paměti programu.
`mem_dealloc`	Mód pro sledování alokací a uvolnění paměti.

8.2 Experiment 1: Vliv nástroje na program

V této sekci je popsán experiment zaměřený na vliv profilování pomocí vytvořeného *profileru* na profilovaný program. V první tabulce 8.1 je ukázáno časové zatížení programu. Sloupce jsou popsány dle módů popsaných v sekci 8.1.

	noprof	all_trace	func_trace	mem_alloc	mem_dealloc
CM	144.1 ms	2 744.1 ms	2 386.9 ms	410.8 ms	410.8 ms
IP	723.9 ms	7 267.1 ms	6 149.7 ms	1 822.1 ms	10 900 ms
AS	3 234.8 ms	32 120.1 ms	28 580 ms	8 272.2 ms	83 980.1 ms

Tabulka 8.1: Tabulka srovnání časové zátěže profilování. Čas je měřen jako *Wall* čas.

V tabulce 8.2 je ukázka srovnání maximálního využití paměti nástroje v průběhu profilování (označena jako Mem_{peak}) a výsledná velikost naměřeného profilu (označena jako Mem_{prof}).

	all_trace		func_trace		mem_alloc	
	Mem_{peak}	Mem_{prof}	Mem_{peak}	Mem_{prof}	Mem_{peak}	Mem_{prof}
CM	271 MB	109 MB	255 MB	102 MB	22 MB	9.24 MB
IP	588.6 MB	235 MB	516 MB	207 MB	102 MB	28.5 MB
AS	2 591.4 MB	1 110 MB	2 396 MB	1 006 MB	325.6 MB	80.9 MB

Tabulka 8.2: Tabulka využití paměti a výsledné velikosti profilů, podle nastavení módu profilování.

Výsledky experimentu především odhalily, že na profilování má především vliv nastavený mód. Nejnižší režie byla naměřena u módu `mem_alloc`, kde zpomalení programu bylo naměřeno minimální. Naopak nejvyšší naměřená režie byla při aktivaci zjišťování uvolněných objektů (mód `mem_dealloc`). Z tohoto důvodu je v tuto chvíli vhodné spustit tyto režimy odděleně a analyzovat metriky zvlášť. Vyřešení tohoto problému by mohlo být součástí navazující práce, která by jej mohla vyřešit např. posíláním dat přes systémové roury do samostatného procesu, který by je zpracoval.

Z výsledků zatížení paměťové náročnosti profilování je vidět především problém pro rozsáhlé projekty. Profily jsou velké kvůli uchovávání informací o volání jednotlivých funkcí, které jsou stěžejní pro sestavení grafu volání funkcí a spojení této metriky i s alokací objektů. Tyto metriky jsou u většiny konkurenčních nástrojů zobrazeny pouze jako agregované hodnoty pro funkce, nebo typy.

8.3 Experiment 2: Sledování funkcí

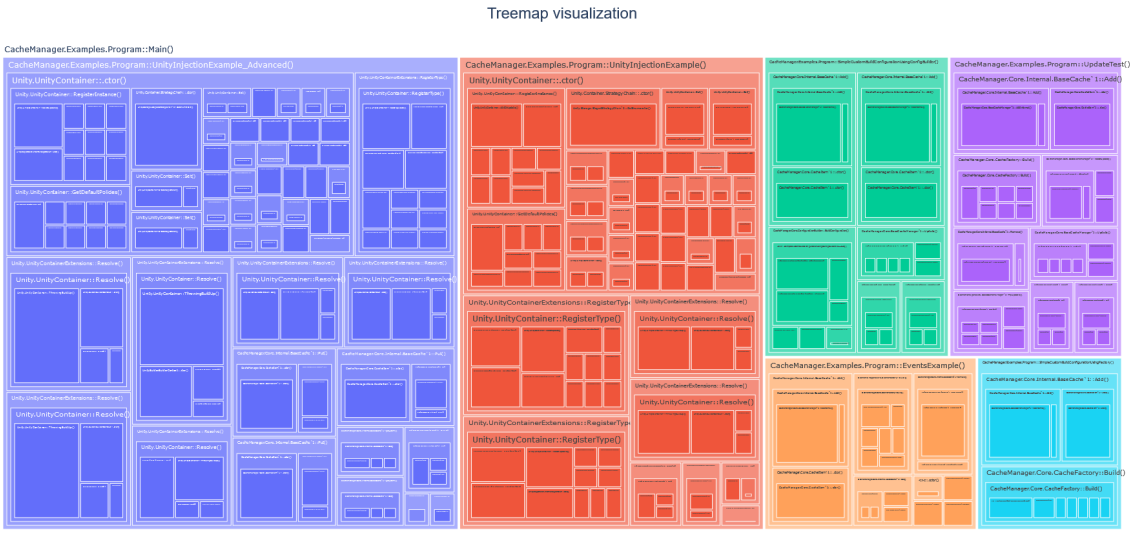
Cílem experimentu bylo ověřit, zda nástroj sbírá data o sledování funkcí korektně. To znamená, zda odpovídají metriky počtu volání funkcí, celkové doby strávené v jednotlivých funkcích a správnému pořadí volání funkcí. Pro tento experiment byl vybrán program `CM` (*CacheManagerExample*) a profilování bylo spuštěno v módu `func_trace` (sledování funkcí). Výsledky měření jsou ukázány v tabulkách 8.3 a 8.4. Obrázek 8.3 potom ilustruje graf volání funkcí.

	Třída	Funkce	Počet Volání
1.	CacheManager.Core.Utility.Guard	NotNull	86 654
2.	CacheManager.Core.Utility.Guard	NotNullOrEmpty	74 225
3.	CacheManager.Core.Utility.Guard	NotNullOrWhiteSpace	74 223

Tabulka 8.3: Tabulka nejčastěji volaných funkcí.

	Třída	Funkce	Celkový čas
1.	CacheManager.Example.Program	Main	2 416.96 ms
2.	CacheManager.Example.Program	UpdateTest	2 078.02 ms
3.	CacheManager.Core.BaseCacheManager'1	UpdateInternal	2 064.83 ms

Tabulka 8.4: Tabulka celkové doby funkcí programu ve *Wall* čase.



Obrázek 8.1: Graf *Treemap* pro zobrazení grafu volání funkcí. Zvětšená verze tohoto obrázku je v příloze A.3.

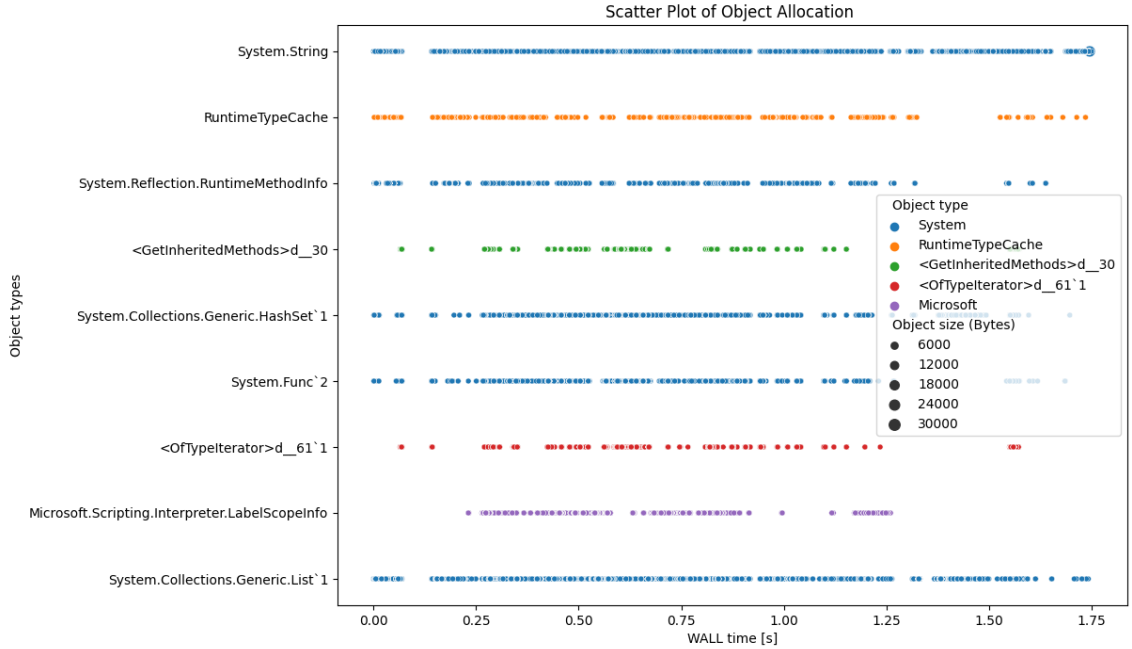
Výsledky všech naměřených metrik byly srovnatelné s konkurenčním nástrojem *DotTrace*. Odlišné hodnoty byly naměřeny u metriky počtu volání funkcí, kdy se pro vybrané funkce počet volání lišil. Pro tyto funkce byly ovšem po opakovaném profilování ukázány jiné hodnoty v obou *profilerech*. Důvodem může být například paralelizace vláken, či jiné faktory běhu programu. Odlišné výsledky pak můžou být ovlivněny jinou technikou profilování, nebo zvýšenou režií u našeho *profileru*.

8.4 Experiment 3: Sledování alokace objektů

Cílem tohoto experimentu bylo demonstrování sledování paměti programu. Pro experiment byl vybrán program `IP` (*IronPythonExample*) s módem `mem_alloc` (sledování alokací a automatické uvolnění paměti). Pro interpretaci alokací objektů ve *Wall* čase je zde ilustrován bodový graf na obrázku 8.2. Na grafu je možné vidět deset nejčastěji volaných typů v průběhu běhu programu.

Pořadí GC	Gen 0	Gen 1	Gen 2	Čas
1.	Ano	Ne	Ne	22.86 ms
2.	Ano	Ano	Ne	72.22 ms

Tabulka 8.5: Tabulka informací o automatickém uvolnění paměti.



Obrázek 8.2: Bodový graf pro zobrazení jednotlivých alokací v čase, podle jejich typu. Zvětšená verze tohoto obrázku je v příloze A.1

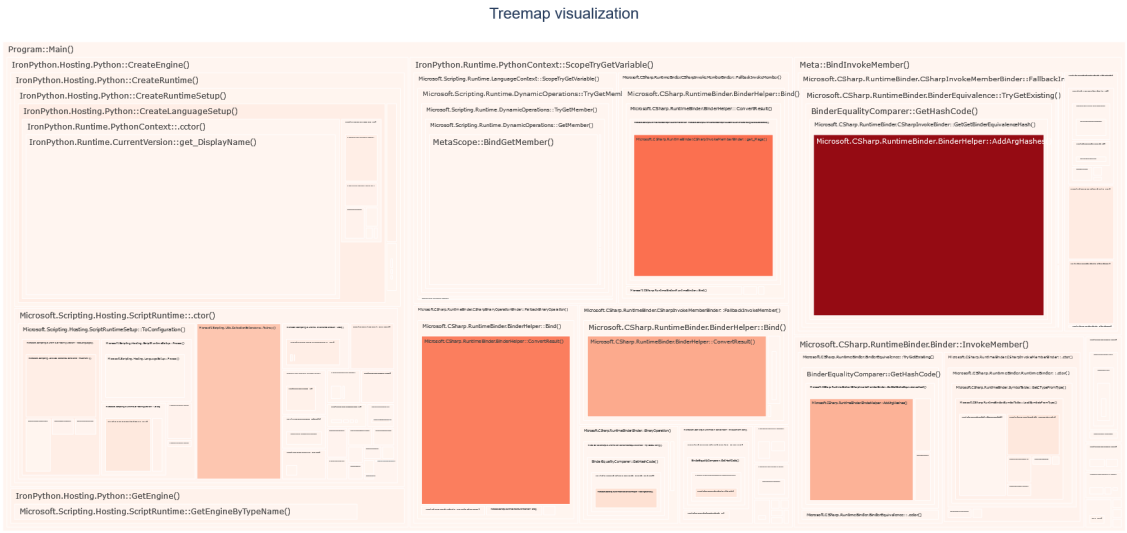
Další nasbíraná metrika pro tento experiment je informace o automatickém uvolněním paměti, která je popsána v tabulce 8.5. V tabulce je uvedeno, které generace se v automatickém uvolnění odstavily a kolik času uvolnění trvalo.

Výsledky experimentu byly porovnány s dalším komerčním *profilerem DotMemory*. Ovšem kvůli neznámým typům některých objektů, popsáno v sekci 6.3.7, nebylo možné porovnat všechny objekty. Porovnány byly tedy pouze objekty se známými typy, u kterých se podařilo naměřit srovnatelná data. Nasbírané informace o automatickém uvolnění, včetně jejich generací, odpovídaly předpokladu.

8.5 Experiment 4: Sledování alokace objektů v závislosti na funkcích

Cílem posledního experimentu bylo demonstrovat funkčnost asociace jednotlivých alokací k funkcím programu. Pro tento experiment byl vybrán program `AS` (*IronPythonExample*). Profilování proběhlo s parametry nastavení režimu `all_trace` (sledování funkcí i paměti). Interpretace těchto dat je následně zobrazena pomocí grafu `Treemap` na obrázku 8.3.

Výsledek experimentu se nepodařilo srovnat s některým jiným profilerem. Většina profilerů totiž sledování spotřeby paměti udává pouze jako agregované hodnoty podle typů, nebo funkcí. Graf stromového volání asociovaný s alokací paměti se podařilo získat pouze z nástroje `Visual studio`. Ovšem tyto asociace jsou spojeny i se systémovými funkcemi, které jsou v našem nástroji omezeny. Můžeme tedy pouze vycházet z ověřené funkcionality



Obrázek 8.3: Graf Treemap pro zobrazení jednotlivých alokací asociovaný na funkce programu. Graf je vizualizován s omezením do hloubky sedm. Zvětšená verze tohoto obrázku je v příloze A.3.

z předcházejících experimentů. Bližší ověření korektnosti této funkcionality kolektoru může být náplní navazující práce.

Kapitola 9

Závěr

Cílem této práce bylo navrhnout a implementovat modul sloužící k profilování programů napsaných v jazyce C#. Výsledný nástroj sbírá data o sledování funkcí a spotřebě paměti v C# programech, resp. ve všech .NET programech. Tato data následně dokáže interpretovat pomocí grafů.

Výsledná implementace byla demonstrována na vybraných známých knihovnách jazyka C#. Výsledky byly porovnány s existujícími komerčními *profilery*, kdy se ve většině případů podařilo naměřit srovnatelná data. Nástroj především vyniká asociací alokace objektů ke konkrétním volání funkcí, kdy lze ze stromového grafu přesně zjistit, v jakém volání funkce byla nejvyšší alokace objektů. Většina konkurenčních nástrojů tuto metriku dokáže zobrazit pouze jako agregované hodnoty pro funkce, nebo typy. Dále je nástroj schopný získávat data o celkové době funkcí, alokací jednotlivých objektů, nebo informací o vykonání automatického uvolnění paměti.

Profilování C# programů je rozsáhlé téma a tato práce by měla sloužit spíše jako její začátek. Navazující práce se bude věnovat integraci tohoto modulu do nástroje Perun. Další možností je zaměření se na opravu známých omezeních, ze sekce 6.3.7, jako např. rozšíření a otestování nástroje na operačním systému Linux, nebo otestování a případné změnění logiky pro zjištění životnosti objektů v paměti. Dalšími směry pro navazující práce, by mohlo být vylepšení logiky kolektoru, v podobě např. posílání dat přes sdílené roury, jiný způsob filtrování funkcí, nebo rozšíření nástroje o další metriky (systém výjimek, argumenty a návratové hodnoty z funkcí, rodičovské reference na objekty a další), či vizualizace. Závěrem by bylo možné vyzkoušet využití jiného způsobu profilování jako je instrumentace, nebo vzorkování, které se dají také implementovat pomocí CLR profilovacího aplikačního rozhraní.

Literatura

[1] *AvalonStudio* [online]. [cit. 2023-4-25]. Dostupné z: https://github.com/VitalElement/AvalonStudio.

[2] *CacheManager* [online]. [cit. 2023-4-25]. Dostupné z: https://github.com/MichaCo/CacheManager.

[3] *Introduction to the Common Language Runtime (CLR)* [online]. [cit. 2023-1-12]. Dostupné z: https://en.wikipedia.org/wiki/.NET.

[4] *IronPython 3* [online]. [cit. 2023-4-25]. Dostupné z: https://github.com/IronLanguages/ironpython3.

[5] *Worst-case execution time* [online]. [cit. 2023-3-12]. Dostupné z: https://en.wikipedia.org/wiki/Worst-case_execution_time.

[6] ANTS. *ANTS Performance Profiler* [online]. [cit. 2023-3-15]. Dostupné z: https://www.red-gate.com/products/dotnet-development/ants-performance-profiler/.

[7] Broman, D. *Enter, Leave, Tailcall Hooks Part 2: Tall tales of tail calls.* 2007. [cit. 2023-4-1]. Dostupné z: https://learn.microsoft.com/en-us/archive/blogs/davbr/enter-leave-tailcall-hooks-part-2-tall-tales-of-tail-calls.

[8] Farrell, C. The Fast Guide to Application Profiling. April 2010. Dostupné z: https://www.red-gate.com/simple-talk/development/dotnet-development/the-fast-guide-to-application-profiling/.

[9] Gosse, K. *Writing a .NET profiler in C# - Part 1.* August 2022. [cit. 2023-4-1]. Dostupné z: https://minidump.net/writing-a-net-profiler-in-c-part-1-d3978aae9b12.

[10] Hamza, H., Nafaa, A. a Sahraoui, H. Static Analysis and Code Complexity Metrics as Early Indicators of Software Defects. In: IEEE. *Proceedings of the 2018 IEEE International Conference on Software Quality, Reliability and Security Companion.* 2018.

[11] Intel. *Intel VTune Profiler* [online]. [cit. 2023-1-10]. Dostupné z: https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html.

[12] Iyengar, P. *All You Need to Know About Code Profiling Tools and How to Choose One.* December 2021. [cit. 2023-1-5]. Dostupné z: https://www.headspin.io/blog/all-you-need-to-know-about-code-profiling-tools-and-how-to-choose-one.

[13] Jan, V. *Performance Profiling for .NET Platform*. 2008. Diplomová práce. České vysoké učení technické, Fakulta elektrotechnická, Praha. Dostupné z: https://dspace.cvut.cz/handle/10467/8738.

[14] JetBrains. *DotTrace Features* [online]. [cit. 2023-1-10]. Dostupné z: https://www.jetbrains.com/profiler/features/.

[15] JetBrains. *DotMemory Features* [online]. Poslední revize 2023. [cit. 2023-1-5]. Dostupné z: https://www.jetbrains.com/dotmemory/features/.

[16] Microsoft. *About Event Tracing* [online]. [cit. 2023-4-20]. Dostupné z: https://learn.microsoft.com/en-us/windows/win32/etw/about-event-tracing.

[17] Microsoft. *CLSID Struct* [online]. [[cit. 2023-4-20]. Dostupné z: https://learn.microsoft.com/en-us/dotnet/api/microsoft.aspnet.snapin.clsid?view=netframework-3.5&viewFallbackFrom=netframework-4.8.

[18] Microsoft. *Common Language Runtime (CLR) overview* [online]. [cit. 2023-4-5]. Dostupné z: https://learn.microsoft.com/en-us/dotnet/standard/clr.

[19] Microsoft. *Component Object Model (COM)* [online]. [cit. 2023-4-15]. Dostupné z: https://learn.microsoft.com/en-us/windows/win32/com/component-object-model--com--portal.

[20] Microsoft. *First look at profiling tools (C#, Visual Basic, C++, F#)* [online]. [cit. 2023-1-6]. Dostupné z: https://learn.microsoft.com/en-us/visualstudio/profiling/profiling-feature-tour?view=vs-2022.

[21] Microsoft. *Fundamentals of garbage collection* [online]. [cit. 2023-4-21]. Dostupné z: https://learn.microsoft.com/en-us/dotnet/standard/garbage-collection/fundamentals.

[22] Microsoft. *.NET Assemblies* [online]. [cit. 2023-4-8]. Dostupné z: https://learn.microsoft.com/cs-cz/dotnet/standard/assembly/.

[23] Microsoft. *PerfView Overview* [online]. [cit. 2023-2-10]. Dostupné z: https://github.com/microsoft/perfview.

[24] Microsoft. *Profiling Overview*. [cit. 2023-3-10]. Dostupné z: https://learn.microsoft.com/en-us/dotnet/framework/unmanaged-api/profiling/profiling-overview.

[25] Microsoft. *IUnknown Interface (Unknwn.h)* [https://docs.microsoft.com/en-us/windows/win32/api/unknwn/nn-unknwn-iunknown]. 2021. [cit. 2023-4-1].

[26] Miroslav, C. *Profiler jazyka C#*. 2008. Bakalářská práce. Univerzita Karlova, Matematicko-fyzikální fakulta, Praha. Dostupné z: https://dspace.cuni.cz/handle/20.500.11956/18541.

[27] Mono. *Mono Profiler* [online]. [cit. 2023-1-2]. Dostupné z: https://www.mono-project.com/docs/debug+profile/profile/profiler/.

[28] Morrison, V. *Introduction to the Common Language Runtime (CLR)* [online]. 2007. [cit. 2023-4-5]. Dostupné z: https://github.com/dotnet/coreclr/blob/master/Documentation/botr/intro-to-clr.md.

[29] 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/.

[30] Nasarre, C. *Dealing with Modules, Assemblies and Types with CLR Profiling APIs*. September 2021. [cit. 2023-4-1]. Dostupné z: https://chnasarre.medium.com/dealing-with-modules-assemblies-and-types-with-clr-profiling-apis-a7522a5abaa9.

[31] Nethercote, N. a Seward, J. Valgrind: A Program Supervision Framework. In: *Proceedings of the 3rd International Conference on Compiler Construction*. 2004. Dostupné z: https://valgrind.org/docs/phd2004.pdf.

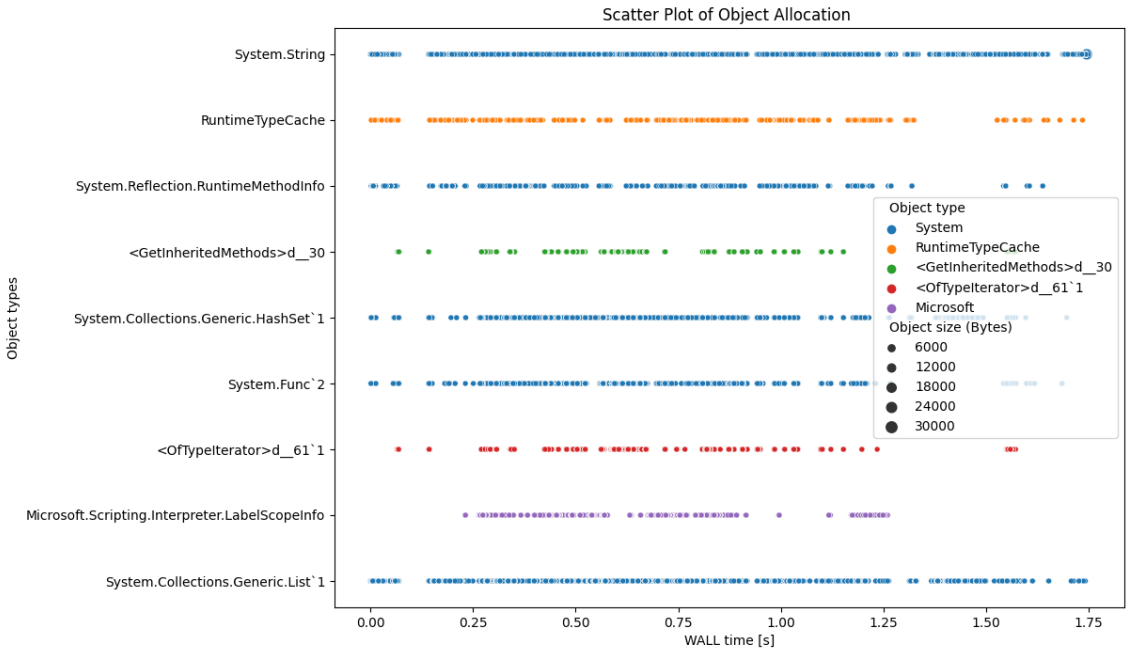
[32] Pavela, J. *Efficient Techniques for Program Performance Analysis*. Brno, CZ, 2020. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Dostupné z: https://www.fit.vut.cz/study/thesis/19092/.

[33] Tomas Fiedor, J. P. *Perun Documentation*. [cit. 2023-4-1]. Dostupné z: https://github.com/tfiedor/perun/blob/master/docs/pdf/perun.pdf.

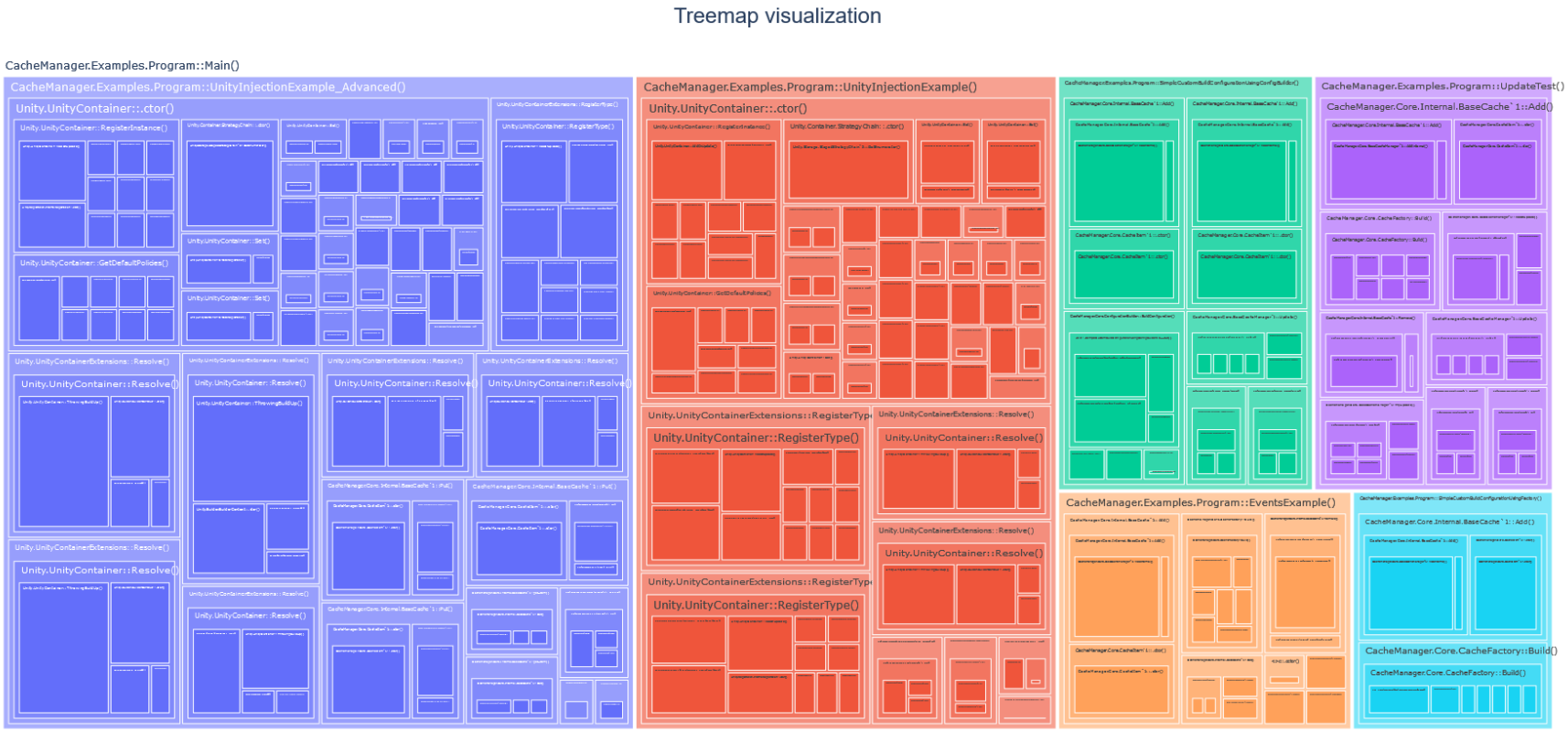
[34] Tomáš, F. *Perun* [online, Github Repository]. [cit. 2022-12-10]. Dostupné z: https://github.com/tfiedor/perun.

Příloha A

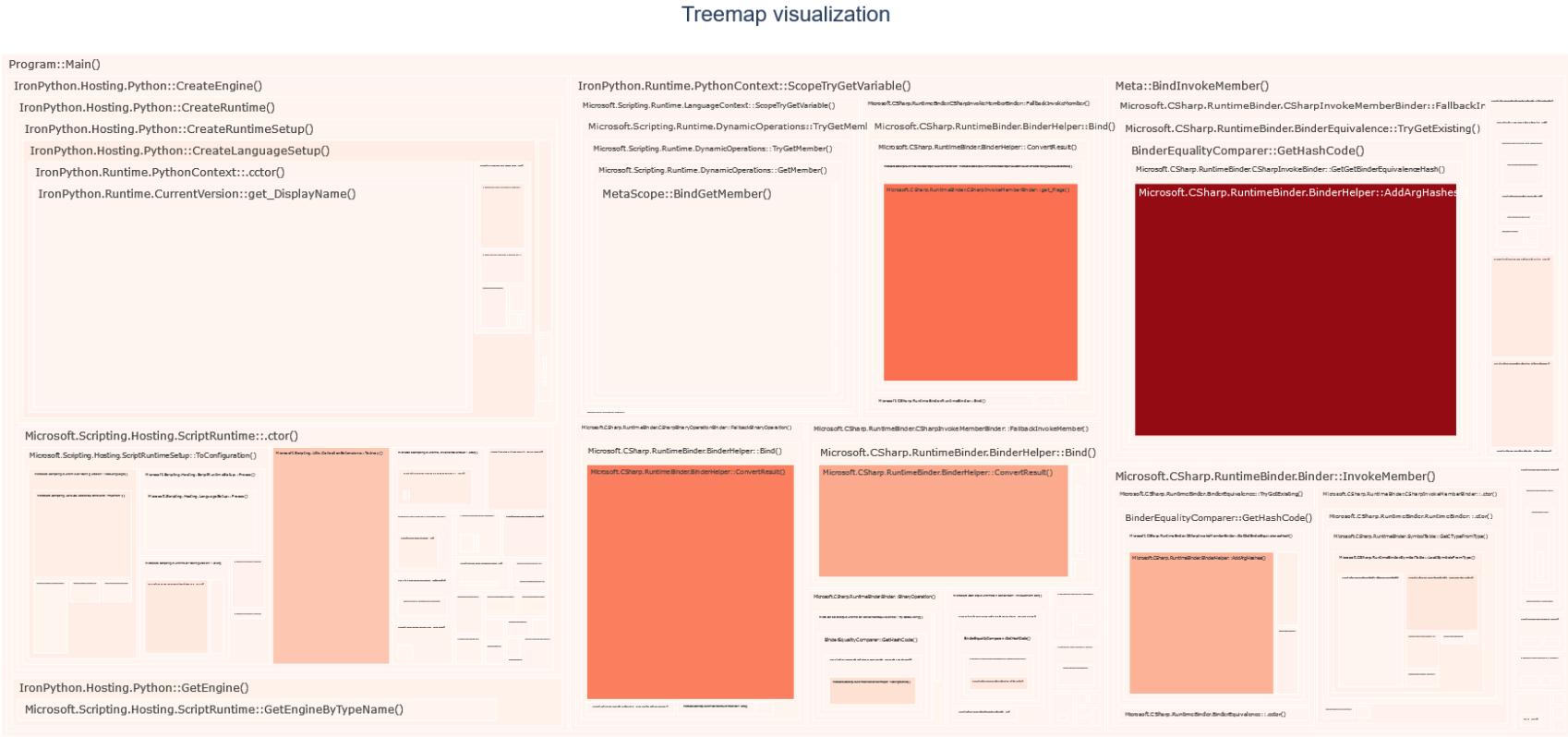
Grafické výstupy experimentálního ověření



Obrázek A.1: Bodový graf pro zobrazení jednotlivých alokací deseti největších typů v programu, zobrazení je v čase *Wall*.



Obrázek A.2: Graf `Treemap` pro zobrazení grafu volání funkcí.



Obrázek A.3: Graf `Treemap` pro zobrazení grafu volání funkcí s velikostí podle alokací objektů. Graf je vizualizován s omezením do hloubky 7.

Příloha B

Obsah paměťového média

Přiložené paměťové médium obsahuje následující složky:

1. `PerunCSharpProfiler` – složka implementace vytvořeného modulu.
2. `ExamplePrograms` – složka pro programy využité pro experimenty.
3. `README.md` – manuál.
4. `thesis-source` – LaTeX zdrojové soubory.
5. `xhajek51.pdf` – bakalářská práce ve formátu PDF.