VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INTELIGENTNÍCH SYSTÉMŮ
DEPARTMENT OF INTELLIGENT SYSTEMS

KNIHOVNA PRO PROFILOVÁNÍ A VIZUALIZACI SPOTŘEBY PAMĚTI PROGRAMŮ C/C++

LIBRARY FOR PROFILING AND VISUALIZATION OF MEMORY CONSUMPTION OF C/C++ PROGRAMS

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — **RADIM PODOLA**
AUTHOR

VEDOUCÍ PRÁCE — **Ing. TOMÁŠ FIEDOR**
SUPERVISOR

BRNO 2017

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav inteligentních systémů Akademický rok 2016/2017

Zadání bakalářské práce

Řešitel: **Podola Radim**
Obor: Informační technologie
Téma: **Knihovna pro profilování a visualizaci spotřeby paměti programů C/C++**
Library for Profiling and Visualization of Memory Consumption of C/C++ Programs
Kategorie: Algoritmy a datové struktury

Pokyny:

1. Seznamte se s nástroji a přístupy pro profilování spotřeby paměti programů v C/C++ (MemTrack, HeapTrack, MemAnalyze).
2. Seznamte se s modelem paměti programů C/C++ a se způsoby instrumentace alokace paměti.
3. Navrhněte knihovnu pro snadné profilování spotřeby paměti programů v C/C++ a přehlednou visualizaci.
4. Nástroj demonstrujte na sadě alespoň deset netriviálních příkladů.

Literatura:

- Franek, F. Memory as a Programming Concept in C and C++. Cambridge University Press, ISBN-13: 978-0521520430
- Domovské stránky projektu HeapTrack. URL: https://github.com/KDE/heaptrack
- Domovské stránky projektu Massif. URL: http://valgrind.org/docs/manual/ms-manual.html

Pro udělení zápočtu za první semestr je požadováno:

- Body 1 a 2

Podrobné závazné pokyny pro vypracování bakalářské práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva bakalářské práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap (20 až 30% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Fiedor Tomáš, Ing.**, UITS FIT VUT
Datum zadání: 1. listopadu 2016
Datum odevzdání: 17. května 2017

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav inteligentních systémů
612 66 Brno, Božetěchova 2

L.S.

doc. Dr. Ing. Petr Hanáček
vedoucí ústavu

Abstrakt

Chování programu z hlediska výkonu je důležitou, a pro některé třídy programů až kritickou stránkou běhu. Současné přístupy k profilování výkonnosti však nejsou zdaleka dostačující. Cílem této bakalářské práce je rozšířit současný stav profilovacích a vizualizačních řešení novými technikami, které mohou značně usnadnit hledání výkonnostních chyb programů v jazyce C/C++ a jejich následnou interpretaci uživateli. Práce stručně představuje existující nástroje, které se zabývají podobnou problematikou, a následně navrhuje nové řešení pro kolekci profilovacích dat a jejich ilustrativní interpretaci. Výsledná implementace je navíc integrována do platformy Perun — Performance Control System — pro správu výkonnostních profilů. Funkčnost implementace je demonstrována na řadě netriviálních programů.

Abstract

Performance is an important and, for some classes of programs, even critical aspect of user experience. The current approaches to performance profiling are, however, far from being satisfactory. The aim of this bachelor's thesis is to extend the current state-of-the-art of profiling and visualization solutions, with novel techniques which can be used for a more efficient search of performance bugs in C/C++ programs and their subsequent interpretation to the end user. Thesis briefly introduces existing tools dealing with similar problems and then proposes a novel solution to collection of profiling data and their illustrative interpretation. The resulting implementation is, moreover, integrated in the Perun — Performance Control System — platform for profile versioning. The functionality of the implementation is demonstrated on a series of non-trivial programs.

Klíčová slova

profilování, C/C++, vizualizace, operační paměť, výkon, Python, Bokeh, GCC

Keywords

profiling, C/C++, visualization, memory, performance, Python, Bokeh, GCC

Citace

PODOLA, Radim. *Knihovna pro profilování a vizualizaci spotřeby paměti programů C/C++*. Brno, 2017. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Fiedor Tomáš.

Knihovna pro profilování a vizualizaci spotřeby paměti programů C/C++

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Tomáše Fiedora. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Radim Podola
16. května 2017

Poděkování

Rád bych poděkoval panu Ing. Tomáši Fiedorovi za odborné vedení mé práce a poskytnutí cenných zkušeností. Taktéž bych rád poděkoval mé rodinně za podporu při studiu.

Obsah

Kapitola 1

Úvod

Profilování je technika systematického vyhledávání kritických míst v programu, které jsou kandidáty pro optimalizaci. Při profilování správy paměti pak hledáme části kódu, které provádí operace s pamětí. Tyto operace jsou často výpočetně náročné, a proto mají značný dopad na výkonnost celé aplikace. Nevhodný způsob používání těchto operací a strategie správy paměti v aplikaci, může často způsobovat velké výkonnostní potíže a vést až k chybám za běhu. Pro řadu programů jako jsou vestavěné systémy, jádra operačních systémů nebo ovladače zařízení, proto může být minimalizace spotřeby paměti kritickým aspektem. Navíc programy využívající mnoho operační paměti mohou mít problémy se stránkováním na disk.

Profilování správy operační paměti může pomoci aplikaci v řadě případů jako je:

- lokalizace míst v programu, které provádí velké množství paměťových operací,
- detekce alokované paměti, která není používaná (jedná se o dynamicky přidělenou paměť, která není ztracená — stále je referována jiným objektem, ale není využívaná),
- hledání neuvolněné paměti,
- upozornění na dočasné alokace (dynamicky přidělená paměť, která byla ihned po alokaci uvolněna).

Optimalizací aplikace pak může dojít ke značnému zrychlení, k lepší práci s procesorovou mezipamětí a vyhnutí se stránkování na disk.

Pro optimalizaci správy paměti existují postupy a komplexní nástroje zvané profilery. Profilery získávají metadata o správě paměti a usnadňují její bližší analýzu. Analýza je užitečná při lokalizaci části programu způsobující výkonnostní chyby nebo jiné problémy s pamětí. Tyto nástroje často získaná metadata vizualizují, pro lepší pochopení správy paměti. Vhodná vizualizace tak může značně usnadnit celý proces optimalizace.

Stávající řešení profilerů však nelze považovat za ideální. Často totiž poskytují jen výsledky ve formě prvotních alokačních metadat a pro lepší pochopení je nutné vyhledat patřičný externí vizualizér nebo si vizualizér sám vytvořit. Jiná řešení naopak neposkytují dostatečné profilovací informace nebo jsou neúměrně výpočetně náročná.

Cílem této práce je navrhnout knihovnu skládající se z profileru a vizualizéru alokačních metadat pro programy v jazyce C a C++. Je kladen důraz na výpočetně nenáročný způsob profilování, který přitom dosahuje dostačujících výsledků. Cílem je také návrh nových způsobů vizualizací, které by poskytovaly nové možnosti pohledu na správu paměti. Knihovna je navržena a implementována pro lehkou integraci do komplexnějších nástrojů pro optimalizaci výkonu aplikací. Části knihovny jsou na sobě nezávislé, aby se daly využít jako samostatné celky.

Práce je rozdělena do osmi kapitol. Kapitola 2 uvádí potřebný teoretický podklad — základní pojmy a principy týkající se symbolických jazyků s upřesňujícími informacemi v kontextu jazyků C a C++. V následující kapitole 3 je uvedeno několik existujících nástrojů pro profilování paměti a jejich srovnání. V kapitole 4 je představen nástroj zaměřený na správu výkonnostních profilů programu, do nějž je knihovna integrována. V kapitole 5 jsou analyzovány způsoby kolekce alokačních metadat a také popis vlastního řešení profilace programu. V kapitole 6 jsou poté popsány možnosti interpretace alokačních metadat, které vytvořená vizualizační knihovna nabízí. V posledních dvou kapitolách 7 a 8 je shrnut a na sérii příkladů demonstrován výsledek této práce.

Kapitola 2

Paměťový model jazyků C a C++

Tato kapitola se zabývá konceptem paměťového modelu v jazycích C a C++, možnostmi práce s operační pamětí v těchto jazycích a způsoby její dynamické alokace. Budou zde vysvětleny základní pojmy a principy potřebné pro objasnění a porozumění této práce. Tento text vychází z [9].

2.1 Základní pojmy

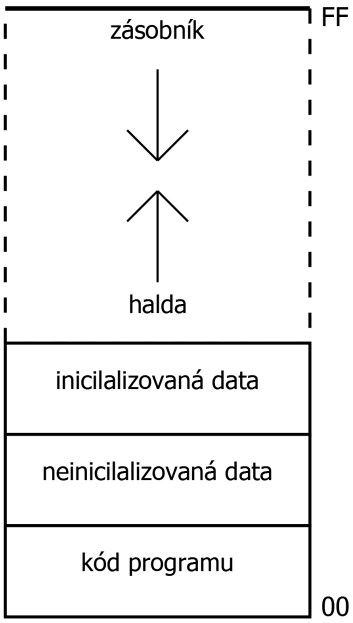
Operační paměť je rychle přístupná paměť používaná pro dočasné uložení kódu spouštěných programů spolu s jejich zpracovávanými daty a výsledky. Procesor vybírá požadovanou buňku operační paměti pomocí adresy pevné délky. Fyzický adresový prostor paměti se jeví jako souvislý prostor paměťových míst určité velikosti. Moderní procesory obvykle používají různé metody překladu adres a virtualizace paměti, takže každý proces může adresovat vlastní logický adresový prostor nebo několik adresových prostorů. Tato práce nezmiňuje operační paměť počítače z fyzického či technického pohledu, ale zaobírá s ní pouze jako s programátorským konceptem a soustředí se na roli jakou hraje v programech. Jazyky C a C++ rozlišují v logickém adresovém prostoru paměti dva typy míst — *zásobník volání* a *haldu*. Logický adresový prostor procesu je znázorněn na obrázku 2.1.

Zásobník volání, neboli též zkráceně jen *zásobník*, je typ místa v paměti procesu, který se vyznačuje LIFO[1] přístupem. Slouží pro uložení lokálních proměnných a předávání parametrů při volání funkcí. Místo se rezervuje před voláním funkce a je automaticky uvolněno ihned po jejím konci. Program má v paměti k dispozici pouze jeden *zásobník*, jehož počáteční pozice je umístěna na konci adresového prostoru. Při vkládání prvků dochází k růstu *zásobníku* směrem dolů.

Halda je prostor v paměti procesu, který je využíván pro dynamickou alokaci místa pro dynamicky vytvořené objekty. Adresa ani velikost alokovaného místa nemusí být předem známá, proto se pro jeho adresaci využívá ukazatelů. Program může alokovat více míst v haldě, a to kdykoliv za běhu. Uvolnění místa neprobíhá automaticky, ale je zcela v režii programátora. Prostor haldy navazuje na staticky alokovaný prostor a roste směrem nahoru. Jednotlivě alokovaná místa nemusí být souvislá, a mohou mezi nimi vznikat nealokované prostory.

Symbolický programovací jazyk používá symboly pro vyjadřování operací a operandů. Všechny moderní programovací jazyky, jako C a C++ jsou symbolické jazyky.

[1] LIFO - Last In First Out je metoda řízení, při které poslední příchozí prvek je obsloužen jako první.



Obrázek 2.1: Schéma logického adresového prostoru procesu.

Alokace paměti označuje rezervaci části operační paměti. Alokaci můžeme dělit na dva způsoby — *statickou* a *dynamickou*. *Statická alokace* rezervuje přesně danou velikost operační paměti před spuštěním programu, její velikost musí být známá v době překladu a nelze ji za běhu měnit. Rezervaci vždy provádí překladač programu, který zná velikost potřebnou pro uložení kódu programu a jeho statických dat. Naproti tomu *dynamická alokace* je prováděna při běhu programu. Jedná se o rezervaci místa, jehož velikost není před překladem programu známá, a může se při každém spuštění programu lišit. Dynamicky alokovat místo lze jak na *zásobníku* tak na *haldě*. Dynamickou alokaci provádí program voláním speciálních funkcí (viz sekce 2.5) nebo logika jazyka (alokace místa na zásobníku pro uložení aktivačního rámce spolu s lokálními proměnnými).

Datovým objektem jsou v textu, ve snaze sjednocení přístupu k jazykům C a C++, souhrnně označeny proměnné, konstanty, datové struktury a pole v obou jazycích a navíc objekty, v kontextu objektově orientovaného jazyka, v jazyce C++.

2.2 Spustitelný soubor symbolického jazyka

Překlad zdrojového kódu do spustitelného souboru se skládá z několik fází. Nejprve je zdrojový soubor přeložen do souboru objektového, který obsahuje strojový kód programu cílové architektury s dodatečnými informacemi o programu, jako je relokační tabulka, adresa zásobníku nebo tabulka mapování symbolů na adresy. Pokud se program skládá z více zdrojových souborů, tak je přeložen do odpovídajícího objektového souboru každý z nich zvlášť. Jednotlivě vytvořené objektové soubory jsou poté slinkovány do jednoho výsledného spustitelného souboru. Linkování má dvě fáze — *relokaci* a samotné *linkování*. Při *relokaci* dochází k překladu interních symbolů v rámci jednoho objektového souboru, a to v každém objektovém souboru zvlášť. Symboly jsou nahrazeny relativními adresami (offsety) k začátku objektového souboru nebo k začátku aktivačního rámce na zásobníku u lokálních datových objektů. Ve fázi *linkování* dochází k překladu externích symbolů, odkazů

do jiného objektového souboru (např. volání funkce z externí knihovny). Při *linkování* se všechny objektové soubory sloučí v jeden spustitelný soubor a upraví se relativní adresy, aby odpovídaly začátku spustitelného souboru.

Vytvořený spustitelný soubor může být poté celý nahrán do operační paměti a proveden procesorem. Nahrání souboru do paměti zprostředkovává operační systém, který překládá relativní adresy programového adresového prostoru na fyzické adresy operační paměti. Není tedy zaručeno, že objekty uložené ihned za sebou v logické paměti programu, budou takto uloženy i v paměti fyzické. Pravidlem ale je, že statické datové objekty budou uloženy ve statické části paměti programu a dynamické objekty v části dynamické. Spuštěný program nazýváme procesem.

2.3 Uložení datových objektů v paměti

Datový objekt je definován svou velikostí a vnitřním obsahem. Uchovává informace, se kterými program pracuje (ukládá je do paměti nebo je z ní čte). Přístup k objektu vyžaduje znalost cesty k němu, tedy jeho *adresy*. *Adresou objektu* je relativní adresa v adresovém prostoru programu, do něhož datový objekt patří.

Není žádoucí, aby v programu existovalo nezměrné množství objektů o různých velikostech, proto každý objekt je určen *datovým typem*. *Datový typ* je definován jazykem nebo uživatelem. *Datový typ* zaručuje, že libovolné objekty stejného *datového typu*, mají stejnou velikost. Velikosti se mohou v rámci různých platforem lišit, avšak v rámci stejné platformy, je zaručena identická velikost. Různá velikost je způsobena rozdílnou architekturou (32-bitová, 64-bitová, ...), zarovnáním v paměti, apod.

Sekce 2.2 definuje datový objekt jako úsek v paměti, kde je reprezentován jako posloupnost bitů. Takovou posloupnost však lze interpretovat různým způsobem (např. jako číslo nebo znak). Aby data v paměti, reprezentující datový objekt byla validní po celou dobu běhu programu, musí být zajištěno, že s nimi bude program po celou dobu pracovat stejným způsobem. Proto datový typ objektu také definuje patřičné *kódování* při čtení i zápisu dat. Aby program mohl pracovat s datovými objekty, potřebuje o nich vědět dvě věci — adresu a datový typ, který udává velikost a *kódování* při manipulaci.

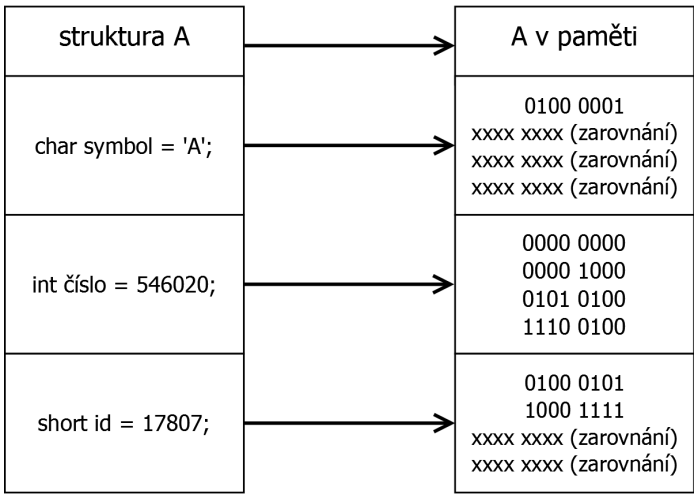
Jelikož fyzická paměť není nikdy prázdná, není ani datový objekt nikdy prázdný, dokonce ani ihned po vytvoření. Při práci s objektem je nutné toto brát na vědomí a před prací datový objekt inicializovat, aby nedocházelo k neočekávanému chování.

U statických objektů jsou informace o atributech datového objektu uloženy v symbolické části spustitelného souboru. Dynamické objekty však nelze ve zdrojovém kódu reprezentovat symboly, a tak je nutné informace o nich udržovat jiným způsobem, pomocí tzv. *ukazatelů* (viz sekce 2.4). Statické datové objekty, na rozdíl od dynamických existují po celou dobu běhu programu. Mohou to být objekty globální nebo lokální, definované pomocí klíčového slova `static`.

2.3.1 Uložení datové struktury

Opravdové uložení datové struktury v paměti může být pro programátora značně nejasné. Při práci se strukturou je potřeba ji vnímat na dvou konceptuálních úrovních zároveň.

- Jako strukturu složenou z komponent,
- a jako souvislý segment v paměti.



Obrázek 2.2: Zarovnání struktury v paměti na blok o velikosti `int`. Schéma převzato z [9].

Součet velikostí jednotlivých komponent struktury nemusí odpovídat skutečné velikosti struktury v paměti.

Při uložení struktury do paměti totiž může docházet (a v jazycích C a C++ dochází) k *zarovnání* některých jejích komponent. *Zarovnání* znamená vložení nevýznamných dat za datový objekt, kvůli vyplnění místa do velikosti bloku paměti, s níž pracuje procesor. Moderní počítače totiž při operacích s pamětí pracují s blokem dat o konstantní velikosti (např. u 32-bitových operačních systémů se jedná o blok dat s velikostí 4 B). Vzhledem ke způsobu, jakým procesor pracuje s pamětí, *zarovnání* nedostatečně velikých dat na velikost bloku výrazně zlepšuje výkon celého systému. Zarovnání struktury je znázorněno na obrázku 2.2. Kvůli práci s bloky je také počátek každého datového objektu v paměti, respektive jeho adresa, n-násobkem velikosti bloku.

Při ukládání struktur je nutné brát zarovnání v potaz, protože může způsobovat rozdílnou velikost stejně definovaných struktur na různých platformách. Pro získání skutečné velikosti struktury lze využít operátoru `sizeof`.

Důležitým prvkem pro paměťově efektivní práci se strukturou je uspořádání komponent v ní. Nevhodně zvolené uspořádání jako například na obrázku 2.2, kdy se střídají komponenty o nedostatečné velikosti s komponenty velikosti bloku, dochází ke zbytečnému zarovnávání.

Speciální strukturou v jazyce C a C++ je *unie*. Její komponenty se v paměti překrývají, proto při změně libovolné komponenty, dochází také ke změně všech ostatních. *Unie* tak uchovává pouze jednu hodnotu, kterou lze různě interpretovat. Interpretace hodnoty závisí na typu komponenty, přes kterou se s ní pracuje. Při práci s komponentou menší velikosti, se tak pracuje jen s částí celé struktury. *Unie* má velikost rovnou velikosti největší komponenty s případným zarovnáním.

2.4 Proměnná ukazatel

Ukazatel je speciální proměnná, kterou definují dva atributy — hodnota a datový typ. Hodnota je počáteční adresou dynamického objektu, který *ukazatel* referuje. Datový typ, na rozdíl od normální proměnné neudává velikost, ani kódování samotného *ukazatele* (toto je

dáno informací, že se jedná o speciální proměnnou *ukazatel*), ale odkazovaného objektu. Může vzniknout situace, kdy na jeden dynamický objekt odkazuje více *ukazatelů* o rozdílných datových typech, v takovém případě se s objektem pracuje, jako by měl datový typ *ukazatele*, přes něhož se k objektu přistoupilo. *Ukazatel* může být definován i speciálním datovým typem `void*`, který reprezentuje čistě adresu bez doménového datového typu. Přistupování k objektu pomocí *ukazatele* se nazývá dereference. V jazyce C a C++ lze dereferencovat objekt pomocí operátoru `*` a získat referenci (adresu) na objekt pomocí operátoru `&`. Referenci lze každému *ukazateli* dynamicky libovolně měnit.

Při práci s *ukazateli* je nutné dávat pozor na neinicializované hodnoty. Práce s neinicializovanými *ukazateli* může vést k přepsání důležitých dat v dynamické paměti, nebo vést až k neoprávněnému přístupu do paměti. Pokud *ukazatel* v části programu neudržuje žádnou validní referenci na objekt, lze mu přiřadit speciální hodnotu, která signalizuje, že *ukazatel* neodkazuje žádný objekt (v jazyce C a C++ je takovou hodnotou `NULL`).

2.4.1 Ukazatel na funkci

Jelikož i funkce programu je uložená v paměti, existuje adresa, na které je začátek této funkce. Stejně jako na počáteční adresu datového objektu, lze odkazovat ukazatelem i na počáteční adresu funkce. Pomocí ukazatele na funkci lze pak funkci zavolat. Ukazatel na funkci musí být takového datového typu, který odpovídá argumentům a návratové hodnotě odkazované funkce.

2.5 Proces dynamické alokace operační paměti

O správu operační paměti se starají správcové. V operačním systému existuje jeden hlavní správce, který spravuje celou fyzickou paměť (přiděluje úseky paměti procesům, provádí překlad logické adresy na fyzickou, atd.), a pak podřízení správci paměti procesu, individuální pro každý běžící proces v systému, kteří spravují logický adresový prostor procesu. Pokud proces žádá o alokaci paměti, žádá správce paměti tohoto procesu. Tento správce má od spuštění procesu k dispozici určitou část paměti, poskytnutou pro požadavky procesu. Pokud proces žádá přidělení paměti a jeho správce nemá k dispozici takové množství, správce zažádá o přidělení dodatečné paměti hlavního správce paměti operačního systému. Systémový správce paměti si vede záznamy přidělené paměti u každého procesu.

Při přidělování paměti lze uplatnit heuristiky, které mohou mít značně kladný dopad na výkon programu, jako například kdy:

- Při žádosti správce paměti procesu o přidělení dodatečné paměti, je přiděleno procesu více paměti, než správce žádal. Bere se v úvahu, že pokud proces vyčerpal svůj přidělený adresový prostor, a již žádá o dodatečnou paměť, může tato situace v blízké době opět nastat. A protože komunikace se systémovým správcem má dopad na celý systém (všechny procesy), je snaha o snížení požadavků k němu.

- Při uvolnění paměti procesem, správce paměti procesu neuvolní tuto paměť zpět systémovému správci ihned, ale ponechá si ji dočasně ve svém adresovém prostoru. Bere se v úvahu, že proces může v budoucnu žádat znovu o tuto paměť. Ponecháním paměti může dojít k vyhnutí se časově náročné komunikaci se systémovým správcem.

Míra uplatňování těchto heuristik se ovšem musí odvíjet od reálné velikosti volné fyzické paměti.

Požadavky a celá komunikace mezi procesem a jeho správcem paměti je realizována prostřednictvím standardních funkcí, které tvoří platformě nezávislé programové rozhraní pro komunikaci se správcem paměti procesu. Nejdůležitější funkce budou dále stručně popsány. Tyto alokátory dynamicky alokují paměť na haldě.

2.5.1 malloc

```
void* malloc(size_t size);
```

Funkce `malloc` alokuje úsek operační paměti o požadované velikosti `size` v bajtech, a v případě úspěchu vrací ukazatel bez doménového typu na jeho začátek. Pro přístup k alokované paměti je nutné přetypování ukazatele. V případě, že požadovaná velikost je rovna nule, tak chování závisí na konkrétní implementaci. Paměť se při alokaci neinicializuje, její obsah je nedefinován. Funkce ve skutečnosti alokuje více paměti než uživatel žádal, což je způsobeno zarovnáním a přidáním pomocných informací o alokaci. V případě selhání alokace, `malloc` vrací hodnotu `NULL`. Selhání může být způsobeno jedním ze tří důvodů:

- není dostatek volného místa v operační paměti,
- žádaná velikost přesahuje povolený limit pro alokaci
- nebo došlo k interní chybě správce paměti procesu.

2.5.2 calloc

```
void* calloc(size_t nelem, size_t elsize);
```

Funkce `calloc` funguje stejně jako `malloc` (viz 2.5.1), s tím rozdílem, že alokovaný úsek operační paměti je vyprázdněn. Požadovaná velikost, na rozdíl od `mallocu`, je dána násobkem parametrů `nelem` a `elsize`, kde parametr `nelem` reprezentuje počet alokovaných prvků, každý o velikosti `elsize` v bajtech.

2.5.3 realloc

```
void* realloc(void* ptr, size_t size);
```

Funkce `realloc` umožňuje upravovat velikost již alokované paměti. Parametr funkce `prt` je ukazatel na již alokovanou paměť, parametr `size` je hodnota nové požadované velikosti a návratovou hodnotou je adresa začátku upravené paměti. `Realloc` garantuje, že data uložená v paměti, zůstanou nezměněna, ale může dojít k přesunu úseku na nové místo (proběhne změna adresy), a v tom případě je starý úsek automaticky uvolněn (nejsou ovšem změněny hodnoty ukazatelů na něj, tzn. ukazatelé se stávají neplatnými). V případě, že parametr `prt` je roven hodnotě `NULL`, tak se funkce chová stejně jako `malloc` (viz 2.5.1). Pokud je hodnota parametru `size` rovna nule, chová se funkce jako `free` (viz 2.5.4). `Realloc` v případě selhání vrací hodnotu `NULL` a alokovaná paměť odkazovaná parametrem `ptr` není změněna, ani uvolněna. Pokud nebyla stará alokovaná paměť alokována jednou z funkcí `malloc`, `calloc` nebo `realloc`, může dojít k neočekávané chybě, nebo až k selhání správce paměti procesu.

2.5.4 free

```
void free(void* ptr);
```

Funkce `free` slouží pro uvolnění alokované paměti odkázané parametrem `ptr`. Paměť je navrácena správci paměti procesu, avšak hodnota všech ukazatelů na ni zůstává nezměněna, tzn. ukazatelé se stávají neplatnými. Pokud je hodnota parametru `ptr` rovna `NULL`, funkce žádnou akci neprovádí. Odkazovaná paměť musí být alokována jednou z funkcí `malloc`, `calloc` nebo `realloc`, v jiném případě může dojít k neočekávané chybě, nebo až k selhání správce paměti procesu.

2.5.5 new, new[]

V jazyce C++ slouží k dynamické alokaci paměti funkce `new`. `New` lze ovšem použít i jako operátor ve výrazu. Pokud je prováděna alokace pro třídní objekt, tak kromě samotné alokace paměti, alokátor také automaticky provádí instanciaci objektu, voláním jeho konstruktoru. Alokátor (jako funkce i jako operátor) může být přetížen programátorskou verzí pro konkrétní specifické potřeby daného objektu, standardně jsou však k dispozici tři verze `new`:

1. `void* operator new(std::size_t size);`
2. `void* operator new(std::size_t size, const std::nothrow_t& nt_value) noexcept;`
3. `void* operator new(std::size_t size, void* ptr) noexcept;`

Verze (1) alokuje místo v paměti o velikosti `size`, v případě selhání je vyvolána výjimka `bad_alloc`. Vrací adresu na počátek alokované paměti.
Verze (2) se chová stejně jako (1), avšak při selhání nevyvolává výjimku, ale vrací ukazatel s hodnotou `NULL`.
Verze (3) nealokuje žádnou paměť, pouze provede instanciaci objektu, na místě odkazovaném parametrem `ptr`. Programátor je odpovědný za správnou alokaci tohoto místa. Alokátor vrací hodnotu parametru `ptr`.

`New[]` je verze alokátoru `new`, která slouží pro alokaci pole objektů. Alokuje místo a provede instanciaci všech objektů (pokud se jedná o třídní objekty).

2.5.6 delete, delete[]

```
void operator delete(void* ptr) noexcept;
```

Pro uvolnění alokované paměti pomocí alokátoru `new`, slouží v C++ funkce `delete`. Stejně jako alokátor, lze i `delete` použít ve výrazu jako operátor. Pokud je uvolňován třídní objekt, `delete` automaticky volá jeho destruktor, a až poté paměť uvolní.

`Delete[]` je verze `delete`, která slouží pro uvolnění pole objektů. Uvolní místo a provede zrušení všech objektů (pokud se jedná o třídní objekty). Pomocí `delete`, resp. `delete[]`, lze uvolnit jen objekt, který byl alokován alokátorem `new`, resp. `new[]`, v jiném případě může dojít k neočekávané chybě nebo až k selhání správce paměti procesu.

2.5.7 Alokace na zásobníku

```
void* alloca(size_t size);
```

Alokace na zásobníku probíhá automaticky při volání funkce v programu, lze však alokovat dodatečné místo pro vlastní použití pomocí funkce `alloca`. Funkce alokuje paměť na zásobníku v zásobníkovém rámci aktuálně prováděné funkce o velikosti dané hodnotou parametru `size` a vrací ukazatel na její začátek ve své návratové hodnotě. Takto alokovanou paměť nelze explicitně uvolnit, ale je uvolněna automaticky při návratu z funkce, ve které alokace proběhla. Pokud alokace způsobí přetečení zásobníku, chování programu je nedefinováno.

Kapitola 3

Existující profilovací nástroje

V této kapitole budou představeny vybrané existující nástroje pro profilování správy operační paměti. Podrobněji pak budou představeny nástroje Massif a Heaptrack, jejichž výhody a nevýhody budou stručně diskutovány.

3.1 Massif

Massif [26] je profilovací nástroj operační paměti, součást profilovací a debugovací sady Valgrind [27] pro platformu Linux. Je vytvořen v programovacím jazyce C a je stále ve vývoji již od roku 2000, aktuálně k dispozici ve verzi 3.12.0. Nové verze opravují pouze chyby a přizpůsobují nástroj změnám podporovaných platforem. Jedná se o volně šiřitelný nástroj pod licencí GNU GPL v. 2[1].

Massif zaznamenává programem dynamicky alokovanou paměť — paměť, kterou si program přímo vyžádal, včetně dodatečného paměťového prostoru vzniklého zarovnáním a uložením administrativních dat o alokaci. Díky záznamu trasy programového zásobníku při každém provedení alokace, poskytuje informace o konkrétních místech v programu, kde dynamická alokace proběhla, jako kombinaci názvu zdrojového souboru, názvu funkce a čísla řádku. Volitelně také dokáže zaznamenávat velikost programového zásobníku (tato funkce je však ve výchozím nastavení vypnuta, protože profilování značně zpomaluje).

Nástroj registruje pouze dynamické alokace provedené standardními alokátory jazyka C a C++ (tzn. `malloc`, `calloc`, `memalign`, `new` a několik dalších podobných funkcí). Alokace pomocí nízkoúrovňových alokátorů operačního systému jako `mmap` či `brk` jsou ignorovány. Proto nezaznamenává alokace paměti při volání systémových alokátorů přímo v programu nebo při alokaci potřebné pro spuštění programu systémem (nahrání programu do operační paměti, alokace paměti pro statické objekty, ...). Toto je důsledkem faktu, že Massif standardně pracuje s operační pamětí jako s logickými paměťovými bloky. Parametrem však lze chování nástroje upravit tak, aby operační paměť profiloval na konceptuálně nižší úrovni jako jsou paměťové stránky, díky čemu bude schopen registrovat i alokace provedené systémovými alokátory.

Pokud dojde během provádění programu k vytvoření nového procesu na základě procesu programu, např. systémovým voláním `fork`, nový proces zdědí veškerá profilovací data.

[1] https://www.gnu.org/licenses/old-licenses/gpl-2.0.html

```
77.83% (265,799B) (heap allocation functions) malloc/new/new[], --alloc-fns, etc.
->40.31% (137,664B) 0x4E6230F: xcalloc (alloc.c:51)
| ->20.03% (68,400B) 0x4E9D4F0: PrependItem (item_lib.c:297)
| | ->20.01% (68,352B) 0x4E62756: PrependAlphaList (alphalist.c:127)
| | | ->20.01% (68,352B) 0x4E757B1: NewClass (env_context.c:422)
| | |   ->19.41% (66,288B) 0x4E7E7D6: ModuleProtocol (evalfunction.c:4168)
| | |   | ->19.41% (66,288B) 0x409EB1: VerifyExec (verify_exec.c:252)
| | |   |   ->19.41% (66,288B) 0x40A409: VerifyExecPromise (verify_exec.c:50)
| | |   |     ->19.41% (66,288B) 0x4073FF: KeepAgentPromise (cf-agent.c:1102)
| | |   |       ->19.41% (66,288B) 0x4E8131B: ExpandPromiseAndDo (expand.c:713)
| | |   |         ->19.41% (66,288B) 0x4E81640: ExpandPromise (expand.c:140)
| | |   |           ->19.41% (66,288B) 0x40864C: ScheduleAgentOperations (cf-agent.c:921)
| | |   |             ->19.41% (66,288B) 0x406B9D: main (cf-agent.c:866)
| | |   |
| | |   ->00.60% (2,064B) in 1+ places, all below ms_print's threshold (01.00%)
| | |
| | ->00.01% (48B) in 1+ places, all below ms_print's threshold (01.00%)
| |
| ->19.19% (65,536B) 0x4E9B882: HashInsertElement (hashes.c:95)
```

Obrázek 3.1: Výstup nástroje `ms_print`.

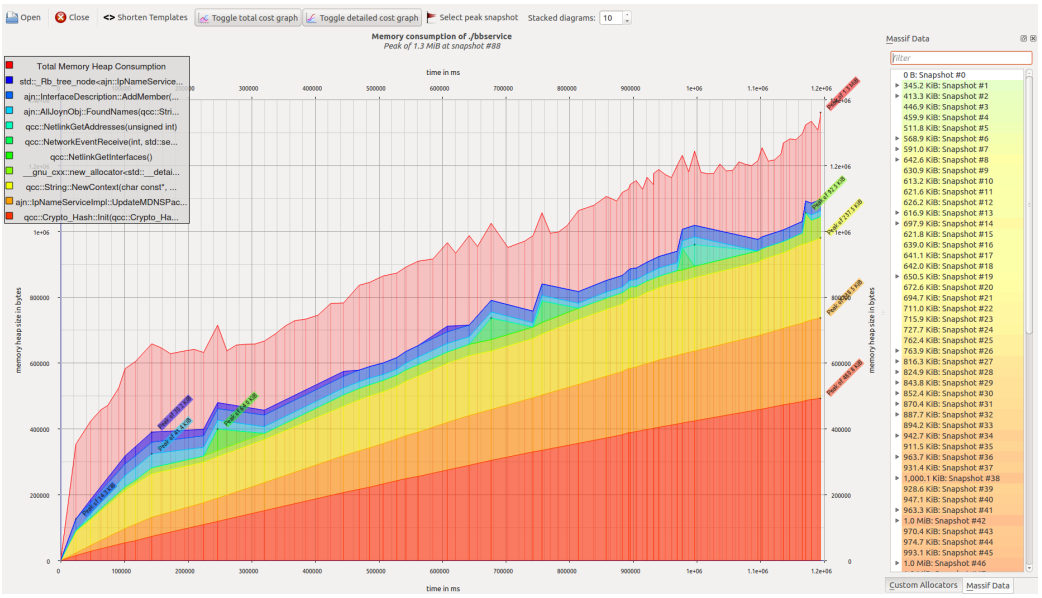
Přístup k alokátorům (systémovým i standardním) a programovému zásobníku umožňuje jádro Valgrindu. Jádro nespouští program přímo fyzickým CPU počítače, ale vlastním abstraktním CPU, které simuluje každou instrukci. Práce přímo s binárním spustitelným souborem umožňuje profilovat program libovolných zdrojových jazyků. Profilován je navíc nejen samotný program, ale i dynamicky linkované knihovny. Pro konkrétnější informace je doporučeno překládat program s povolením debugovacích symbolů. Valgrind poskytuje podporu pro debugovací symboly ve formátu DWARF2/3/4 [6].

Profilovací alokační metadata se ukládají do souboru ve formátu prostého textu. Massif poskytuje post-procesní nástroj `ms_print` pro jejich agregaci a jednodušší interpretaci. Výstupem nástroje `ms_print` je graf znázorňující spotřebu paměti při běhu programu, informace o jednotlivých alokacích a alokační strom, v němž je zachycena trasa funkcí, jak lze vidět na obrázu 3.1. Syntaxe souboru metadat není formálně zdokumentována, je však záměrně lehce čitelná pro člověka i pro počítač, což dovoluje vytvářet další nástroje pro jejich zpracování a vizualizaci. Jedním z takových externích vizualizačních nástrojů je například Massif-Visualizer [31], jehož příklad vizualizace lze vidět na obrázku 3.2. Chybějící vlastní vyzualizační nástroj, stejně jako výkonnostně náročné profilování, je však velkou nevýhodou Massifu. Při profilování rozsáhlých a náročných aplikací se profilování stává časově neúnosným. Výkonnostní dopad profilování lze vidět v tabulce 3.1.

Mezi významné parametry nástroje patří:

- `-pages-as-heap=<yes|no>` — nastavení profilování operační paměti na úroveň stránek (výchozí hodnota: no),
- `-heap-admin=<size>` — specifikace velikosti dodatečného úseku alokované paměti, kde jsou uloženy pomocné alokační informace pro potřeby alokátorů, v bajtech (výchozí hodnota: 8),
- `-alloc-fn=<name>` — specifikace uživatelské alokační funkce jako standardní alokátor,
- `-ignore-fn=<name>` — nastavení ignorování přímých alokací (přímé volání alokátorů) v dané funkci.

Dalšími parametry lze dosáhnout různých modifikací výstupních dat. Kompletní seznam parametrů Massifu lze nalézt na [26].



Obrázek 3.2: Výstup nástroje Massif-Visualizer [31].

3.2 Heaptrack

Heaptrack [30] je profilovací nástroj operační paměti, podporující platformu Linux. Nástroj je vytvořen v programovacím jazyce C++ a jeho vývoj stále pokračuje. Byla již vydána plně funkční verze, která je volně šiřitelná pod licencí GNU LGPL v. 2.1[2]. Heaptrack bohužel není momentálně součástí žádného balíčku v žádné Linuxové distribuci, a pro jeho používání je nutné doinstalovat do systému řadu závislostí (úplný seznam a postup instalace lze nalézt na [30]). Autor uvádí, že tento projekt vznikl z nespokojenosti při používání nástroje Massif.

Heaptrack je konceptuálně rozdělen do dvou částí:

- *kolektor* alokačních metadat — `heaptrack`,
- a grafický *analyzátor* alokačních metadat — `heaptrack_gui`.

Tyto části jsou na sobě nezávislé a mají rozdílné systémové nároky. Lze například zaznamenat metadata kolektorem na jednom systému a na jiném pak metadata analyzovat.

Kolektor je vymezen pro profilování programů napsaných v jazyce C a C++. Zaznamenává všechny dynamické alokace operační paměti provedené standardními alokátory jazyka C (`malloc`, `calloc`, ...). Na programovém zásobníku trasuje funkce, které si alokaci vyžádaly. Přístup k alokačním informacím získává předefinováním standardních alokátorů. Upravené alokátory jsou zkompilovány do dynamické knihovny, kterou se nahradí standardní knihovna se standardními alokátory jazyka C. Nahrazení probíhá pomocí nastavení proměnné prostředí `LD_PRELOAD` [19]. Nastavení zaručí, že upravená knihovna bude vždy linkována s programem, a budou tak volány předefinované alokátory. Upravené alokátory při každé žádosti o alokaci vytrasují informace o žádající funkci na programovém zásobníku. Pro přístup a práci se zásobníkem využívá *kolektor* knihovnu `lubunwind` [20]. Získaná metadata obsahují pouze adresy funkcí v paměti, proto je žádoucí program překládat s povolením

[2] https://www.gnu.org/licenses/old-licenses/lgpl-2.1.html

debugovacích symbolů (*kolektor* podporuje symboly ve formátu DWARF [6]). S debugovacími symboly jsou adresy interpretovatelné v konkrétní místa programu (název funkce, zdrojový soubor, řádek v souboru). Zjištěná metadata jsou pak vypisována na standardní výstup.

Nástroj dovoluje analyzovat již běžící proces identifikovaný pomocí PID[3]. K procesu se připojí pomocí debugeru GDB [12] příkazem `attach`, a poté systémovou funkcí `dlopen` připojí dynamickou knihovnu s upravenými alokátory. Protože při spouštění programu jsou na programovém zásobníku uloženy odkazy na standardní alokátory, přepíše tyto odkazy novými referencemi na předefinované alokátory v připojené knihovně.

Kolektor generuje alokační metadata, která jsou bohužel pro člověka neinterpretovatelná. Pro jejich analýzu je proto nutné využít *analyzátor* s textovým výstupem `heaptrack_print` nebo grafický *analyzátor* `heaptrack_gui`.

`heaptrack_print` poskytuje textový výstup ve formátu ASCII[4]. Obsahem je:

- seřazený seznam lokací v programu dle nejčastějšího volání alokačních funkcí,
- seřazený seznam funkcí dle největšího podílu na celkové alokované velikosti operační paměti
- a seřazený seznam lokací dle nejčetnějších dočasných alokací (alokace paměti, které byly ihned uvolněny).

Výstup obsahuje prvních 10 položek každého seznamu.

`heaptrack_gui` je grafické uživatelské prostředí *analyzátoru*. Ve formě tabulek prezentuje výsledky stejné jako `heaptrack_print`, a navíc nabízí řadu možností jejich vizualizace jako:

- zobrazení tras funkcí, které prováděly alokace paměti, ve stromové reprezentaci,
- Flame graf [15]
- nebo graf alokace paměti v průběhu času provádění programu.

Analyzátor nabízí možnost konverze formátu alokačních metadat, které generuje *kolektor* do formátu výstupních metadat nástroje Massif. Pro jejich grafickou interpretaci, pak lze použít vizualizéry, které tento formát podporují, například Massif-Visualizer [31]. Z Massif-Visualizeru vychází i nástroj `heaptrack_gui`, neboť jeho autorem je právě autor Heaptracku.

[3] *Process IDentifier* — číslo, pod kterým je v jádře operačního systému jednoznačně evidován proces.
[4] *American Standard Code for Information Interchange*

	bez profilování	Heatrack	Massif
počet instrukcí	156 034 336	5 403 905 664	9 692 798 770
strojový čas [s]	0.08497	1.00318	2.53458
počet výpadků stránek	910	106 589	8 856
spotřeba paměti [MB]	26.1	58.5	264.9

Tabulka 3.1: Výkonnostní srovnání profilování.

3.2.1 Srovnání s nástrojem Massif:

□ Výhody:

- Menší paměťová a výpočetní náročnost profilování. Dáno způsobem profilování, Massif spouští programy pomocí vlastního abstraktního CPU, serializuje také více-vláknové aplikace jako jedno-vláknové, a tím vytěžuje systémové CPU po celou dobu profilování.
- Více dostupných alokačních metadat. Massif před výstupem část metadat agreguje, a tím se ztrácí část informací o alokacích.
- Možnost profilování již běžícího programu.

□ Nevýhody:

- Nemožnost profilovat operační paměť na nižší konceptuální úrovni stránek. Heaptrack není schopný zaznamenat alokace provedené uživatelskými či systémovými alokátory.
- Nemožnost zaznamenávat dynamické alokace na programovém zásobníku.

V tabulce 3.1 je uvedeno výkonnostní srovnání při profilování nástrojem Massif, Heaptrack a spuštěním aplikace bez profilování. Je zřejmé, že profilování běh aplikace značně zpomaluje a využívá větší množství systémových zdrojů. Zajímavým srovnáním je však výkonnostní dopad obou profilerů. Z hodnot srovnání lze vyvodit, že profilování nástrojem Massif je asi 2.5krát pomalejší než nástrojem Heaptrack a také provádí skoro 2krát více instrukcí. Massif také vykazuje znatelně větší spotřebu paměti, ačkoliv s ní lépe pracuje.

3.3 Další profilovací nástroje

MTuner [24] je C/C++ paměťový profilovací nástroj pro platformu Windows, PS4 a PS3. Při profilování uchovává celou historii paměťových operací s časovou značkou. Zaznamenaná metadata jsou uložena do souboru, nad kterým lze provádět sofistikované dotazy, filtrování a další analytické metody. Pomocí poskytnutého SDK[5] uživatel může spravovat celý proces profilování a přizpůsobit ho na míru vlastním potřebám.

PERF [22] slouží k profilování a zaznamenávání nejrůznějších událostí v operačním systému Linux. Skládá se z několika částí, kdy každá profiluje jiný typ událostí. Část pro profilování paměti je označena `perf-mem`. `Perf-mem` zaznamenává každý typ operace s operační pamětí pro specifikovaný systémový příkaz nebo aplikaci a zobrazuje uživateli frekvenci provedených operací. PERF je součástí linuxového jádra, což mu dovoluje pracovat na velmi nízké úrovni, a je tak například vhodný pro profilování mezipamětí.

[5] *Software Development Kit*

Intel® VTune™ Amplifier [17] je profilovací nástroj od společnosti Intel®. Jedná se o komerční proprietární nástroj pro komplexní profilování výkonu aplikací pro platformy Windows a Linux. Pro pokročilejší analýzu je vyžadována Intel® architektura procesoru, protože nástroj dokáže využívat hardwarovou jednotku pro monitorování výkonu PMU[6]. Použití PMU vykazuje minimální výpočetní náročnost profilování. Nástroj umožňuje profilovat i dynamicky generovaný kód JIT[7] překladačem nebo analyzovat dopad datových struktur na výkon aplikace. Ovládání je možné prostřednictvím grafického uživatelského prostředí nebo příkazové řádky. Součástí je analyzátor profilovacích metadat, který umožňuje i jejich řazení, filtrování a vizualizaci.

[6] *Performance Monitoring Unit*

[7] *Just In Time* je označení pro speciální metodu překladu využívající částečné přeložení programu do mezikódu pro urychlení jeho běhu.

Kapitola 4

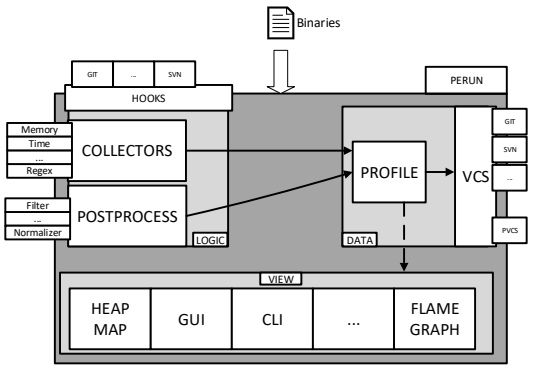
Perun: Performance Control System

Tato kapitola popisuje nástroj Perun [8], v jehož rámci jsou profilovací a vizualizační moduly z této práce realizovány. Je zde popsána architektura nástroje, tak podíl této práce a integraci do něj. Autorem nástroje je T. Fiedor, vedoucí této práce. Autor této práce spolupracoval na návrhu architektury nástroje — zejména na modulech a integraci kolektorů, vizualizací, a jejich rozhraní v podobě unifikovaného profilovacího formátu.

Perun je odlehčený opensource nástroj zaměřený na správu výkonnostních profilů programu. Je založený na principech systémů pro správu verzí. Nástroj umožňuje uložení výkonnostních profilů pro každou dílčí verzi projektu, kdy každý profil je svázán s konkrétním kontextem fáze vývoje — co bylo v kódu změněno, kdy to bylo změněno a kdo změnu provedl. Profily lze generovat manuálně nebo automatizovaně — Perun se dokáže integrovat do používaného verzovacího systému pomocí tzv. *hooks* aby zajistil, že pro každou novou verzi projektu bude vytvořen výkonnostní profil (např. při každém commitu ve verzovacím systému GIT automatizovaně vytvoří profil). Inspirován systémy průběžné integrace umožňuje jednoduše vytvářet matici úkolů (např. kolekci dat pomocí konkrétních kolektorů a následné generování profilů), která je prováděna pro každou verzi projektu. Nástroj takto dokáže uchovávat historii projektu z hlediska výkonu a umožňuje pohled na výkonnostní změny v průběhu vývoje a času.

4.1 Architektura

Perun je rozdělen do tří hlavních částí: *pohledu*, *dat* a *logiky*. *Datová* část obsahuje základní jednotku Perunu — profil a wrapper nad verzovacímy systémy. *Logika* má na starosti manipulaci a automatizaci vytváření profilů. Skládá se ze sady kolektorů pro generování profilů, sérií postprocesorů pro jejich transformace, přístupů k verzovacím systémům a jiným externím jednotkám pro dosažení automatizace. *Pohled* je nezávislý modul, který obsahuje vizualizační metody generovaných profilů a wrappery pro grafické a textové rozhraní. Znázornění architektury je zobrazeno na obrázku 4.1.



Obrázek 4.1: Znázornění architektury nástroje Perun. [8]

4.2 Proces profilace

Profilovací data jsou generována sadou podporovaných kolektorů (jako je například kolektor dat paměťových operací — výsledek této práce, kolektor dat složitosti, nebo kolektor řízený regulárními výrazy) a mohou být následně rozšířeny a transformovány sérií postprocesorových sad (jako jsou například filtry, normalizéry, atd.).

Profil je po výstupu z kolektorů komprimován, uložen do adresářové struktury nástroje a přiřazen k aktuálnímu kontextu projektu. Uložené profily poté mohou být vizualizovány řadou dostupných vizualizačních metod jako je mapa haldy — výsledek této práce, korelační diagramy, aj.

4.3 Integrace

Perun nabízí pro integraci jednotlivých modulů jednoduché API. Každý modul musí být umístěn do vlastního Python balíčku pojmenovaného podle jména konkrétního modulu. V tomto balíčku musí také být skript `{jméno modulu}.py`, přes který probíhá komunikace mezi modulem a logikou Perunu. Rozhraní nástroje vynucuje uvnitř skriptu definice následujících tří funkcí:

1. `before(**kwargs)` — funkce spustí inicializační fázi kolektoru, modul má v této fázi prostor pro vytvoření a inicializaci potřebných prostředků pro následnou kolekci dat,
2. `collect(**kwargs)/postprocess(**kwargs)` — funkce spustí fázi kolekce nebo postprocesových operací dat; modul v této fázi profiluje spustitelný soubor a sbírá metadata, popřípadě provádí postprocesové operace nad již získanými daty,
3. `after(**kwargs)` — funkce spustí fázi po úspěšné kolekci dat; modul má v této fázi prostor pro dodatečnou manipulaci s nasbíranými daty, např. transformaci do jednotného formátu profilu.

Fáze 1 a 3 nejsou povinné. Každá z těchto funkcí má při svém volání očekávat data obsahující hlavičku profilu a dodatečné informace pro spuštění kolekce dat ve formě pojmenovaných volitelných parametrů. Každá funkce také musí vracet sadu tří hodnot:

- **návratový kód** — decimální číslo reprezentující stav proběhlé fáze (0 = v pořádku, jiné = chyba),
- **návratovou zprávu** — text poskytující dodatečné informace o proběhlé fázi (využíván zejména pro chybové hlášení),
- **upravené parametry volání** — parametry volání funkce s dodatečně přidanými hodnotami získanými v dané fázi potřebnými pro následující fáze. Formou úpravy parametru profilu jsou také vrácena získaná profilovací data.

Kapitola 5

Sběr alokačních metadat

V této kapitole budou analyzovány způsoby pro získání informací o správě dynamické paměti programem. V sekci 5.1 jsou uvedeny způsoby přístupu k dynamickým alokacím, v sekci 5.2 pak možnosti získání dodatečných informací o dynamicky alokované paměti, tzv. alokačních metadat. Na konci kapitoly v sekci 5.3 je popsána integrace vytvořeného kolektoru alokačních metadat do nástroje Perun.

5.1 Přístup k alokacím

Pro zaznamenání každé alokace provedené v programu je nutné mít přímý přístup k žádosti o dynamickou alokaci a to přímo za běhu dané aplikace. Takto je poté možné zaznamenat každou alokaci a základní metadata o ní. Možnosti přístupu jsou zmíněny v podsekcích 5.1.1, 5.1.2 a 5.1.3.

Kromě záznamu alokačních metadat je však nutné také zajistit správnou funkcionalitu dané alokace, tj. správně dynamicky alokovat požadovanou paměť. Toto lze řešit dvěma způsoby:

- Vlastní implementací alokačních funkcí, které se funkcionálně shodují se standardními. Tyto funkce žádají o přidělení paměti přímo systémového správce operační paměti prostřednictvím nízkoúrovňových systémových alokátorů jako `sbrk` nebo `mmap`.
- Delegováním samotné dynamické alokace na původní standardní alokátor.

5.1.1 Malloc Hooks

Knihovna GNU C [14] poskytuje možnost změnit chování standardních alokačních funkcí za pomoci tzv. *Memory Allocation Hooks* [16]. Jedná se ve skutečnosti o ukazatele na příslušné alokační funkce. Nastavením ukazatele na vlastní funkci lze docílit toho, že při každém volání standardní alokační funkce, bude namísto toho zavolána funkce odkazována odpovídajícím ukazatelem.

Tyto ukazatele jsou však rozšířením GNU C knihovny a tedy závislé na jejím využití v programu. Navíc jejich použití není bezpečné ve vícevláknových aplikacích a oficiálně jsou proto označeny jako zastaralé.

5.1.2 Vlastní alokační funkce

Další možností je definovat vlastní alokační funkce s rozdílným prototypem od standardních funkcí. Je však nutné s těmito funkcemi pracovat již ve zdrojovém kódu, od čehož se odvíjí problém, že tyto funkce nebudou automaticky využívat i jiné části programu, které nejsou napsány s tímto záměrem, jako jsou další zdrojové soubory nebo dynamické knihovny.

5.1.3 Předefinování standardních alokačních funkcí

Ve finální verzi kolektoru jsou alokační metadata získávána předefinováním standardních alokačních funkcí. Implementace těchto funkcí jsou nahrazeny za implementace vlastní a upravené pro potřeby profilování. Tento přístup nevyžaduje změnu zdrojového kódu ani specifický překlad aplikace. Díky tomu, že implementace alokačních funkcí se nachází v dynamické knihovně, a ve spustitelném souboru jsou umístěny pouze reference na ně, lze nahradit jejich implementace i v již přeložených spustitelných aplikacích a to dokonce za běhu dané aplikace. Toto řešení tak není ani závislé na použitém překladači, jedinou podmínkou je využití shodného API[1] i ABI[2] knihovny.

Problémem může být donucení profilované aplikace k používání této nestandardní dynamické knihovny s upravenou implementací. Kolektor tohoto na platformě Linux dociluje nastavením systémové proměnné `LD_PRELOAD` (podobně jako nástroj Heaptrack viz 3.2). Nastavením této proměnné na adresu upravené knihovny je zaručeno, že tato knihovna bude načtena před jakoukoliv jinou knihovnou, a to včetně standardních knihoven.

Také delegace alokace na standardní alokátory je problematická. Kvůli předefinování, nelze volat alokátory přímo (volány by byly předefinované funkce). Avšak pomocí funkce `dlsym()`[2] lze získat reference na implementace standardních alokátorů, na které již lze alokace delegovat. Upravená implementace funkce `malloc` včetně delegace na originální implementaci je uvedena v ukázce 5.1.

Zdrojový kód 5.1: Upravená implementace funkce malloc.

```
1  void *malloc(size_t size){
2      //static pointer variable holds a reference to the real
           malloc
3      static void *(*real_malloc)(size_t) = NULL;
4      //getting a reference to the real malloc in case it is not
           set yet
5      if(!real_malloc){
6          real_malloc = dlsym(RTLD_NEXT, "malloc");
7          init_log_file();
8      }
9      //allocation request delegation to the real malloc
10     void *ptr = real_malloc(size);
11     //backtrace and allocation info record
12     if(!profiling && ptr != NULL)
13         ad_log("malloc", size, ptr);
14
15     return ptr;
16 }
```

[1] *Application Programming Interface*
[2] *Application Binary Interface*

5.2 Získání dodatečných alokačních metadat

V redefinovaných alokačních funkcích lze získat pouze taková metadata, s nimiž alokační funkce přímo pracuje. Ze sekce 2.5 vyplývá, že se jedná o velikost požadované paměti a adresu přidělené paměti. Pro profilování je však vhodné mít o alokaci dodatečné informace jako je název funkce, která alokaci provedla, a přesnější lokaci této alokace. Pro komplexnější pohled na správu paměti může být užitečné lokalizovat celou trasu funkcí, která k alokaci vedla.

Možnosti získání adres alokací jsou blíže popsány v podsekcích 5.2.1, 5.2.2 a 5.2.3. Takto získané adresy jsou však ve formě hexadecimálních relativních adres v programu, které jsou těžce interpretovatelné. Je tedy nutné tyto adresy přeložit na informace, se kterými programátor při vývoji běžně pracuje, jako je *jméno* funkce ve zdrojovém souboru. Ve spustitelném souboru jsou informace, které mapují adresy na *jména* funkcí. Existuje řada nástrojů, které jsou schopny tyto informace zpracovat a poskytují přímý překlad, jako je například GNU[13] nástroj `addr2line`[3]. Pro získání upřesňujících informací jako je mapování adres na zdrojové soubory a řádky v nich, kde k alokaci došlo, je na většině systémech nutné provést překlad programu s povolením debugovacích symbolů.

V jazyce C++ může docházet k problému se získanými názvy funkcí, například protože v něm lze funkce přetěžovat a mít více jmenných prostorů, a překladač tak názvy mapuje na jednoznačné identifikátory. Tento proces je zvaný jako *name mangling* [28]. Existuje řada možností, jak provést inverzní mapování na původní názvy nebo alespoň názvy v čitelné podobě. Jako příklad lze uvést GNU nástroj `c++filt`[4] nebo funkci `abi::__cxa_demangle()`[5] knihovny `libstdc++`.

Je nutné brát na vědomí, že řada optimalizací prováděných překladačem může vést k získání nevalidní trasy (pro potřeby profilování je doporučeno překládat program bez optimalizací). Kromě optimalizací může k nepřesné trase vést také použití *inline* funkcí [29], jelikož pro ty nejsou vytvářeny rámce na zásobníku.

Ve finální implementaci kolektoru jsou získaná alokační metadata upravována a překládána až po jejich sběru při analýze. K překladu relativních adres je využit nástroj `addr2line` a ke zpětnému mapování jmen nástroj `c++filt`.

5.2.1 Vestavěné GCC funkce

Překladač GCC [10] poskytuje řadu vestavěných funkcí pro získání informací o volající funkci [11]. `__builtin_return_address(LEVEL)` je funkce, která vrací adresu aktuální funkce nebo jedné z volajících na zásobníku volajících funkcí. Argument `LEVEL` udává pořadové číslo rámce na zásobníku směrem od vrcholu. Volání funkce s hodnotou argumentu 0, vrací adresu aktuální funkce, s hodnotou 1 adresu volající funkce, atd.

Tato funkce však není pro potřeby profilování dostatečná, jelikož není k dispozici informace o počtu rámců a volání s většími hodnotami argumentu je pak potencionálně nebezpečné a nelze tak získat celou trasu alokace. Dostupnost a použitelnost je závislá na platformě a překladači.

[3] https://sourceware.org/binutils/docs-2.20/binutils/addr2line.html
[4] https://sourceware.org/binutils/docs-2.20/binutils/c_002b_002bfilt.html
[5] https://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.3/a01696.html

5.2.2 backtrace()

GNU C knihovna poskytuje funkci `backtrace()` [1], která přes parametr vrací celou trasu zásobníku volajících funkcí v okamžiku jejího zavolání v programu. Trasa je ve formě pole, kde každý prvek odpovídá návratové adrese funkce odpovídající rámci zásobníku.

Doplňující funkcí `backtrace_symbols()` lze poté získat upřesňující údaje jako jsou symbolická jména funkcí (pokud jsou určitelná) a relativní adresy volání k počáteční adrese funkce.

Tyto funkce jsou rozšířením GNU C knihovny stejně jako 5.1.1, narozdíl od nich však jsou bezpečné při použití ve vícevláknových aplikacích. Možnost získání symbolických jmen adres může vyžadovat speciální nastavení procesu linkování a pro *statické* funkce nelze získat vůbec.

5.2.3 Libunwind

Dodatečná alokační metadata se ve finální verzi kolektoru získávají prostřednictvím knihovny *libunwind* [20]. Ta nabízí aplikační rozhraní k přístupu k programovému zásobníku, z nějž lze analyzovat trasu volaných funkcí. Knihovna byla zvolena, protože poskytuje jedno z nejmodernějších řešení analýzy zásobníku, je rozšířená a platformě nezávislá. Je také více flexibilní než například `backtrace()` (viz 5.2.2), protože dokáže získat hodnoty všech programových registrů v každém rámci, a může tak být použita i pro jiný profilovací záměr. Nevýhodou je pak přidání závislosti do finální verze kolektoru. Princip operací se zásobníkem a získání trasy je zmíněn v podsekci 5.2.4.

Během procesu profilování si kolektor při každém záznamu metadat vytvoří pomocí funkce `unw_getcontext()` obraz programu, tj. uloží si hodnoty programového zásobníku a programových registrů, na jehož základě inicializuje speciální kurzor. Pomocí funkce `unw_step()` poté postupně posouvá kurzorem na předchozí rámce a prochází tak celý zásobník od vrcholu až k začátku programu (prvnímu zásobníkovému rámci). V kurzorem odkazovaném rámci se získává hodnota IP[6] registru, ve kterém jsou uložené potřebné informace pro získání trasy. Pro identifikaci IP registru konkrétní počítačové architektury knihovna poskytuje makro `UNW_REG_IP`. Část vlastní implementace analýzy zásobníku s využitím knihovny je zobrazen v ukázce 5.2.

Knihovna poskytuje nástroje pro analýzu lokálního (od totožného procesu) tak i vzdáleného (od jiného procesu) zásobníku. Pokud bude program použit jen k lokální analýze, mělo by na začátku zdrojového souboru být definováno makro `UNW_LOCAL_ONLY`, které zajistí použití mnohem rychlejší optimalizované verze nástroje na rozdíl od obecnější, vytvořené i pro vzdálenou analýzu. V případě lokální analýzy je ve vícevláknových aplikacích použití knihovny bezpečné, ve vzdálené nikoliv.

Kromě čtení registrů rámce, knihovna také dokáže zprostředkovat zápis do nich nebo spuštění programu od zvoleného rámce, což může značně usnadnit implementaci například zpracování výjimek nebo globálních skoků. Poskytuje také prostředky pro analýzu dynamicky generovaného kódu, například pomocí JIT překladače.

[6] *Instruction Pointer* je speciální registr v procesoru, který adresuje aktuálně prováděnou instrukci strojového kódu v operační paměti.

Zdrojový kód 5.2: Implementace získání trasy.

```
1  void backtrace(FILE *log, unsigned skip){
2      unw_cursor_t cursor;
3      unw_context_t context;
4      unw_word_t ip, offset;
5      int ret = 0;
6      //creating a snapshot of the process state
7      unw_getcontext(&context);
8      //initializing the Libunwind cursor for the local unwinding
9      unw_init_local(&cursor, &context);
10     //unwinding the stack
11     while(unw_step(&cursor) > 0){
12         char symbol[SYMBOL_LEN];
13
14         if(skip > 0){
15             skip--;
16             continue;
17         }
18         //obtaining value of the IP register for the current
               stack frame
19         unw_get_reg(&cursor, UNW_REG_IP, &ip) != 0);
20         unw_get_proc_name(&cursor, symbol, SYMBOL_LEN, &offset);
21         //backtrace record
22         fprintf(log, "%s 0x%lx\n", symbol, ip);
23     }
24 }
```

5.2.4 Průchod programovým zásobníkem [25]

Součástí volání každé funkce je uložení a obnovení registrů (tj. stavu procesu) volající funkce. Hodnoty registrů jsou na většině architektur uloženy v odpovídajících rámcích na programovém zásobníku. Rámce jsou uloženy postupně za sebou, tak jako se událo volání funkcí v průběhu programu. Vzestupným průchodem celého programového zásobníku lze získat kompletní trasu volaných funkcí od začátku programu až po bod, kdy byl zásobník analyzován.

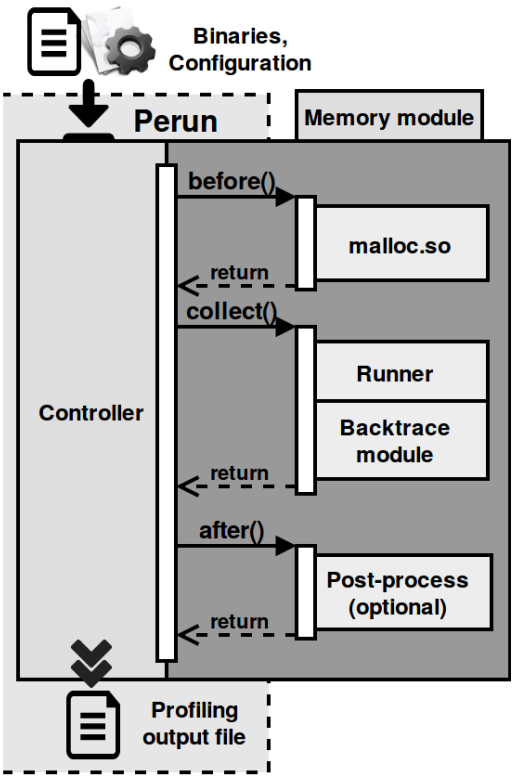
Trasa je tedy vlastně zásobník, kde v každém rámci jsou uloženy hodnoty registrů. Přečtením IP registru každého rámce se získá přehled o řetězci volaných funkcí vedoucí od hlavní funkce programu až k funkci, která provedla paměťovou operaci. Hodnota IP registru je adresou návratového místa volané funkce, respektive relativní adresou instrukce následující po návratu z volané funkce.

Při kompilaci však dochází k výrazným optimalizacím kódu, což značně znesnadňuje rekonstrukci trasy. Proto překladače ukládají potřebné informace pro rekonstrukci do sekce `.debug_frame` spustitelného souboru. Tyto informace pomáhají debuggerům zjistit, jak mají procházet jednotlivými rámci zásobníku, tj. jak obnovit hodnoty programových registrů předchozího rámce v jakékoliv instrukci aktuálního rámce. Generování nebo nahrání této sekce do operační paměti však není zaručeno. Při vývoji x86-64 ABI a nových jazykových konstrukcí (jako jsou výjimky, automatická správa paměti, atd.) se rozhodlo, že k tomuto potřebné informace budou ukládány v jiné sekci, konkrétněji v `.eh_frame`. Např. *libunwind* 5.2.3 získává informace z obou těchto sekcí spustitelného souboru.

5.3 Kolektor v rámci nástroje Perun

V současné době je kolektor úspěšně integrován do nástroje Perun. Nástroj automatizovaně komunikuje a řídí profilaci prostřednictvím svého API, popsaného v kapitole 4, postupným voláním tří funkcí:

1. `before` — v této funkci probíhá kompilace dynamické knihovny s upravenými implementacemi standardních alokačních funkcí,
2. `collect` — v této funkci probíhá profilace programu; program je spuštěn a alokační metadata jsou postupně ukládána do dočasného souboru s názvem `MemoryLog`,
3. `after` — v této fázi profilace již skončila a alokační metadata z dočasného souboru jsou transformována do formátu jednotného profilu; profil v této fázi prochází také dodatečnými úpravami jako je překlad adres a filtrování nežádoucích záznamů; filtrační pravidla lze specifikovat prostřednictvím nástroje Perun (ve výchozím stavu jsou odstraňovány záznamy samotných alokačních funkcí a záznamy, u kterých se nepodařilo získat úplné alokační informace jako např. jméno funkce).



Obrázek 5.1: Znázornění integrace kolektoru do nástroje Perun. Toto je upravená verze schématu vycházející z [21].

Výsledný profil je předán návratovou hodnotou třetí funkce zpět do režie Perunu. Integrace modulu paměťového kolektoru je znázorněna na obrázku 5.1.

Kapitola 6

Analýza a interpretace profilovacích dat

Analýza dat je proces, ve kterém jsou data transformována (pomocí jejich redukování, filtrování, reorganizování či agregování) do podoby, která usnadňuje navazující práci s nimi. Účelem je získání užitečných informací pro podporu rozhodování nebo objevení nových možností. Interpretace je pak způsob výkladu, pochopení nebo objasnění těchto dat. Cílem je porozumění jinak hůře srozumitelným informacím, která jsou analyzována a transformována např. do podoby tabulek, korelačních koeficientů nebo specifických formátů.

6.1 Analýza profilovacích dat

V samotném průběhu profilování (fáze kolekce dat) kolektor pouze ukládá veškeré profilovací data do souboru bez komplexnějšího formátu či zmíněných úprav (viz sekce 5.2). Až později (avšak stále ve fázi kolekce dat) jsou tato data upravována a transformována do generického profilu (popis profilu lze nalézt v sekci 6.1.1). Tento přístup výrazně snižuje negativní dopad na výkon profilování (mírní se tak i nepřesnost časových razítek), umožňuje analýzu pomocí jiných nástrojů a jinými přístupy bez nutnosti opakovat profilaci nebo vytvářet odlišné výsledné formáty profilu bez nutnosti zasahovat do jádra kolektoru.

Kolekci a následnou analýzu lze parametrizovat nastavením vzorkovací frekvence sbíraných dat na základě časového razítka a filtračních pravidel pro odstranění nežádoucích záznamů. Filtrací lze odstranit jak celé záznamy o alokacích, tak i části tras. Z tras lze odstranit záznamy o standardních alokačních funkcích nebo záznamy specifikované zdrojovým souborem nebo názvem funkce. Při implicitním chování jsou odstraněny části tras, které tvoří standardní alokátory a neanalyzované funkce. Neanalyzovanou funkcí je myšlena funkce, u které se sice podařilo získat jméno, ale dodatečné informace nikoliv. Tyto funkce jsou totiž často součástí externích dynamických knihoven a získání informací o nich vyžaduje implementačně náročnější přístup. Avšak ve většině případů při profilování není žádoucí tyto funkce zahrnovat, protože jejich úprava ani není umožněna, proto je vynechání těchto funkcí ze záznamů bráno jako korektní chování.

Celý proces analýzy tedy probíhá až po dokončení samotného sběru profilovacích dat, je však součástí kolektoru. Pro implementaci analýzy byl zvolen jazyk Python [23].

6.1.1 Profil

Výsledná profilovací data jsou uložena ve formátu inspirovaném formátem JSON [18]. Formát *profilu* je navržen v jednotné syntaxi, která umožňuje využití pro více typů profilování. Jednotný profil dat zlehčuje implementaci interpretačních technik zcela nezávislých na konkrétních datech a činí tak vytvořené knihovny flexibilní a znovupoužitelné. Lze také naopak dodatečně upravovat nebo nahrazovat existující kolektory bez nutnosti přizpůsobovat je interpretaci.

Profil kromě samotných dat obsahuje také obecné informace o nastavení a průběhu profilování. Nastavení parametrů kolektoru a specifikace profilovaného programu je uložena v sekcích `header` a `collector`. Na základě těchto kritérií proběhne kolekce dat a profil je doplněn o výsledná profilovací data. Data jsou ukládána v časových intervalech ve formě obrazů procesu, kdy jednotlivé obrazy jsou uloženy v sekci `snapshots`. Zároveň je vytvořen globální obraz, který je uložen v sekci `global` (tato sekce je také využívána pro uložení dat profilování bez dělení do obrazů a odpovídá kumulativním datům). Úplnou specifikaci formátu lze nalézt v dokumentaci nástroje Perun [8] a ukázku lze vidět v 6.1.

Zdrojový kód 6.1: Ukázka formátu profilu

```
Profile = {
  "header": {
    "type": "memory",
    "cmd": "./prog",
    "params": "-g -w -c",
    "workload": "load.in",
    "units": {"memory": "B"}
  },
  "collector": {"name": "memory", "params": "-s 0.001",
                "result": {"status": 0, "status-msg": ""}
               },
  "postprocessors": [{'name': 'filter', 'params': '<30'}],
  "snapshots": [
    {"resources": [
        {"subtype": "malloc",
          "trace": [{"line": 0, "source": "unreachable", "function": "malloc"},
                    {"line": 45, "source": "/home/dev/test.c", "function": "main"}
                   ],
          "uid": {"line": 45, "source": "/home/user/dev/test.c", "function": "main"},
          "amount": 4,
          "type": "memory",
          "address": 13374016
        }
      ],
      "time": "0.001"
    }],
  "global": [{
        "resources": [{"subtype": "valloc",
            "trace": [{"line": 0, "source": "unreachable", "function": "valloc"},
                      {"line": 45, "source": "/home/dev/test.c", "function": "foo2"}
                     ],
            "uid": {"line": 45, "source": "/home/user/dev/test.c", "function": "main"},
            "amount": 512,
            "type": "memory",
            "address": 13374189}
      ],
      "time": "0.0038759"
    }
  ]
}
```

6.2 Interpretace analyzovaných dat

Tato práce se zaměřuje na textovou a vizuální interpretaci. První část výsledného řešení se zaměřuje na práci s konzolovým výstupem ve formě samostatného modulu `memory_print`, který pracuje nad zmíněným unifikovaným *profilem* (viz sekce 6.1.1) a poskytuje výstup jak ve formě textu, tak i sofistikovanější vizualizaci (popis možností nástroje se nachází v podsekci 6.2.1).

Pro různé interpretace je však často nezbytné *profil* převést do jiné podoby. Například pro mapu haldy nebo FLOW graf (viz podsekce 6.2.3) je potřebné upravit data do reprezentace stavu dynamické paměti v každém jednotlivém obraze procesu tak, aby se alokace v průběhu akumulovaly (záznamy neuvolněné alokace v určitém obraze byly přeneseny do obrazu následujícího). Profil byl dále rozšířen o statistiky o každém obraze kvůli ulehčení implementace vizualizace. Problémem převodu je však vysoká redundance úseků dat vedoucí k velké velikosti vytvořených profilů. Výsledný profil je tak redukován pomocí identifikace větších částí profilu, které jsou převedeny na symbolické odkazy do tabulky dat na konci profilu.

6.2.1 Vizualizace v konzoli

Modul `memory_print` je navržen pro jednoduchou správu interpretací výsledků profilování v textové konzoli. Poskytuje základní funkce agregace a filtrování a dva další vizualizační prostředky. Mezi možnosti interpretace profilovacích dat patří:

- **LIST** — seznam alokací (včetně trasy) v časové ose. Parametry `-f(--from=k)` a `-t(--to=k)` lze volit časový úsek,
- **TOP** — seznam největších alokací (včetně trasy). Parametrem `-t(--top=k)` lze omezit počet zobrazených záznamů ze seznamu,
- **MOST** — seznam funkcí, kde byly alokace nejčetnější. Parametrem `-t(--top=k)` lze udat počet zobrazených záznamů ze seznamu,
- **SUM** — výčet funkcí s největší celkovou hodnotou alokované paměti. Parametrem `-t(--top=k)` lze omezit počet zobrazených záznamů z výčtu,
- **FUNC** — výčet alokací vybrané funkce (včetně trasy). Parametr `--all` zobrazí i inkluzivní alokace, tzn. takové, kde funkce nealokovala paměť přímo,
- **HEAP** — interaktivní vizualizace dynamické paměti (více popsána v podsekci 6.2.3),
- **FLOW** — interaktivní vizuální graf spotřeby paměti (více popsán v podsekci 6.2.3).

Volání modulu s parametrem `-h(--help)` vyvolá nápovědu k ovládání.

6.2.2 Textová interpretace

Textová interpretace je složena z řady agregačních funkcí. Výstup každé funkce je formátován pro snadné zobrazení v textové konzoli a snadné pochopení člověkem. Příklad výstupu dvou z řady funkcí lze vidět níže:

```
$ python3 memory_print.py --mode=most -t 2 memory.perf
#1 parsing_packet: 11x in /home/user/dev/client.cpp

#2 server_deamon: 4x in /home/user/dev/server.cpp
$ python3 memory_print.py --mode=top -t 2 memory.perf
#1 malloc: 1095B at 13380000
by
   allocate()  in  /home/user/dev/allocator.c:89
      main()  in  /home/user/dev/allocator.c:45
```

```
#2 valloc: 625B at 13378050
by
   prepare_struct()  in  /home/user/dev/allocator.c:128
      allocate()  in  /home/user/dev/allocator.c:89
         main()  in  /home/user/dev/allocator.c:45
```

Výstup pochází z naměřených dat programu pro jednoduchý přenos souborů. První textová interpretace je v módu `MOST`. Ve výstupu lze vidět, že nejčetněji alokovala paměť funkce `parsing_packet` ve zdrojovém souboru `/home/user/dev/client.cpp`, a to 11krát. Druhá je v módu `TOP`. Zde lze vidět, že největší alokací procesu byla alokace provedena standardním alokátorem `malloc` o velikosti 1095B na adrese 13 380 000. Pod ní je zobrazena trasa vedoucí k této alokaci. Obě funkce zpracovávají profilovací data z profilu `memory.perf` a vypisují pouze první dva záznamy ze seznamů (dáno voláním s parametrem `-t 2`).

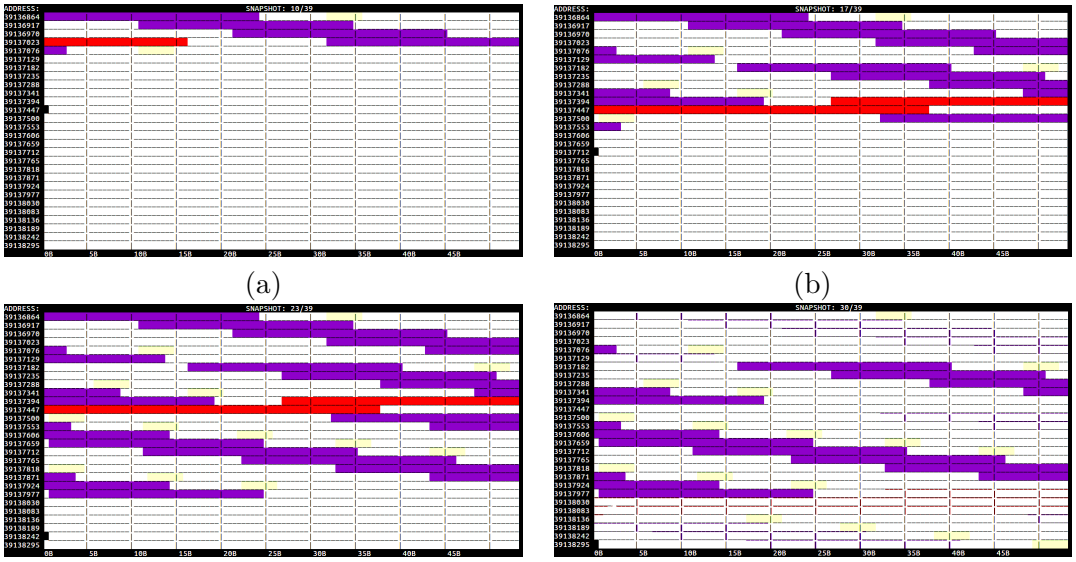
6.2.3 Vizuální interpretace

Druhým implementovaným způsobem interpretace je grafická vizualizace profilovacích dat. Vizualizace dokáží vytvořit srozumitelnější pohled na data a vést ke snadnějšímu pochopení chování programu, přesněji v kontextu práce jak program pracuje a jak využívá systémové zdroje. Vizualizace profilovacích dat jsou blízké informačním *dashboardům* [7]. Problémem při jejich vytváření je návrh, který ke splnění svého účelu musí zobrazit správný rozsah informací na malém zobrazovacím prostoru způsobem pro jasnou a rychlou interakci s daty. Toto vyžaduje využití vlivu vizuálního vnímaní a schopnosti okamžitého zpracování velkého množství informací.

Tato práce implementuje čtyři grafické vizualizace. Dvě z nich jsou navrženy pro konzolový výstup, což může být značně výhodné při profilování programu na vzdáleném serveru bez možnosti grafického uživatelského prostředí. Tyto vizualizace jsou vytvořeny s využitím konzolově nezávislé knihovny *NCURSES* (verze pro jazyk Python [5]), která poskytuje API pro vykreslování a ovládání textové konzole. Zbývající vizualizace jsou ve formě interaktivních grafů pomocí HTML[1] + JS[2], vytvořené pomocí knihovny *Bokeh* [3]. *Bokeh* je knihovna pro jazyk Python, která vytváří interaktivní vizualizace cílené pro zobrazování v moderních internetových prohlížečích.

Mapa haldy První z navržených konzolových vizualizací je mapa haldy. Ta přibližuje pohled na spotřebu a manipulaci s dynamickou pamětí za dobu běhu programu. Tato vizualizace zobrazuje dynamické úseky operační paměti počítače přidělené procesu v podobě mapy, kde jednotlivé paměťové úseky jsou v této mapě znázorněny barevnými poli. Velikost pole aproximačně odpovídá velikosti přidělené paměti a jeho umístění v adrese paměti. Pole jsou barevně odlišeny podle alokačních míst v programu, kdy odpovídající alokace mají totožnou barvu. O každém paměťovém úseku si lze také zobrazit dodatečné profilovací informace. Ve vizualizaci si lze prohlédnout více snímků map, kdy každá z nich odpovídá danému časovým úseku běhu programu. Postupným přehráním všech map, tak lze sledovat celý průběh paměti programu. K automatizovanému přehrání slouží pro usnadnění funkce *Animace*, která všechny mapy postupně přehraje (náznak průběhu lze postupně vidět na snímcích *(a) až (d)* obrázku 6.1).

[1] *HyperText Markup Language* je značkovací jazyk používaný pro tvorbu webových stránek.
[2] *JavaScript* je multiplatformní objektově orientovaný skriptovací jazyk.



(a) (b) (c) (d)

Obrázek 6.1: Na snímku *(a)* jsou viditelné alokace několika paměťových míst, patřících dvěma alokačním místům v programu, na následujícím snímku *(b)* je vidět uvolnění menšího paměťového prostoru a zároveň alokace většího prostoru, patřící prvnímu (červenému) místu programu. Na snímku *(c)* pouze druhé (fialové) místo programu alokovalo další paměťové úseky. Na posledním snímku *(d)* první místo úplně a druhé místo částečně uvolnilo alokované úseky.

Mapa haldy může pomoci například při:

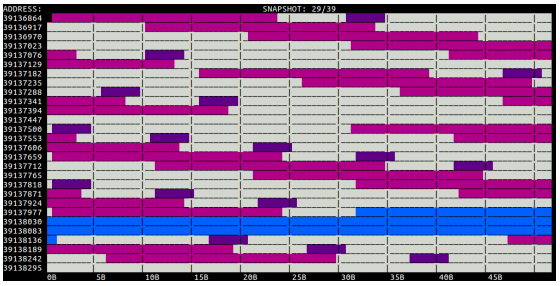
- analýze fragmentace alokovaných paměťových úseků,
- zobrazení rozložení alokovaných dat v datových strukturách různých datových typů
- nebo vidět funkčnost standardních alokačních funkcí, popřípadě při úpravě kolektoru otestovat funkčnost vlastních alokačních funkcí, což může být užitečné například při vývoji vestavěných systémů.

Vizualizace je demonstrována na uložení dat v rozdílných ADT[3] jazyka C++ na obrázku 6.2. Modře jsou data uložená v datovém typu `std::list`[4], zeleně v `std::vector`[5]. Mapa dokazuje, že druhý zmiňovaný typ ukládá data v paměti kontinuálně, a je tudíž často jeho použití o mnoho výhodnější, neboť paměťové operace s daty mohou být méně časově náročné.

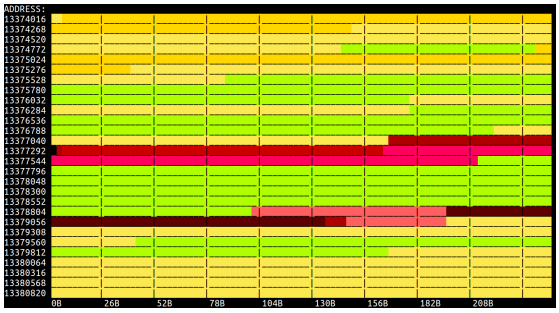
Součástí vizualizace je i vykreslení tzv. *heat* mapy. V *heat* mapě je barevně znázorněna četnost alokací na jednotlivých adresách paměti během celého procesu. S vyšším počtem alokací se mění intenzita barvy daných paměťových úseků (žádná alokace = zelená, menší počet alokací = odstíny žluté, větší počet alokací = odstíny červené), tmavší odstíny tak znázorňují vyšší počet alokací. U každého paměťového úseku si lze zobrazit počet provedených alokací a přesnou adresu. Ukázku lze vidět na obrázku 6.3.

[3] Abstraktní datový typ
[4] http://www.cplusplus.com/reference/list/list/list/
[5] http://www.cplusplus.com/reference/vector/vector/

Heat mapa může upozornit na úseky paměti, které jsou neúměrně často alokovány a uvolňovány. Takové úseky pak může být výhodnější alokovat omezeně a spíše si uchovávat reference na ně a používat je několikrát bez nutnosti uvolnění (ušetří se tak opakované paměťové operace uvolnění i alokace).

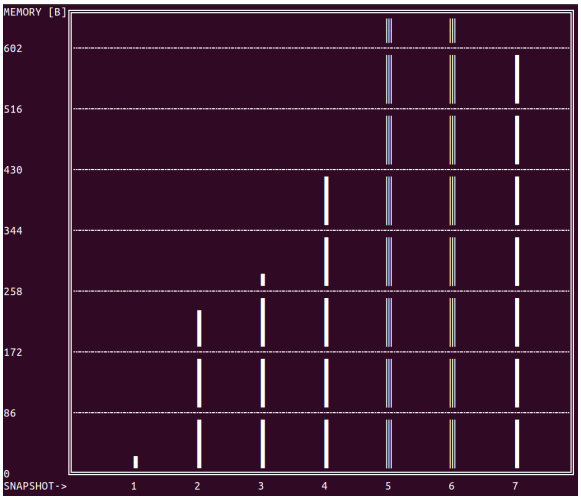


Obrázek 6.2: Zobrazení uložení dat vybraných standardních struktur pomocí mapy haldy.



Obrázek 6.3: Zobrazení četnosti alokací na jednotlivých adresách paměti procesu pomocí *heat* mapy.

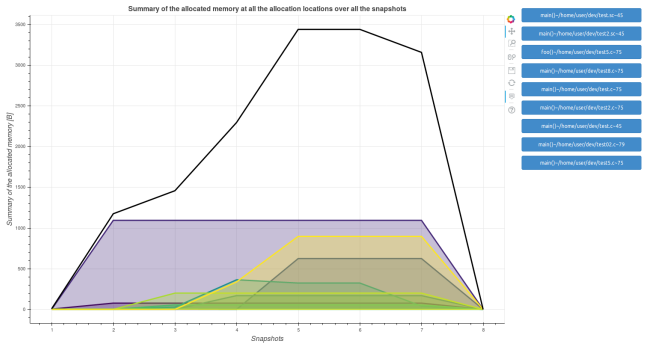
FLOW graf celkové spotřeby Druhou z konzolových vizualizací je interaktivní FLOW graf, který zobrazuje celkovou velikost alokované paměti pro každý obraz v časové linii běhu programu. Základní pohled obsahuje všechny obrazy procesu na jedno okno. Pokud je obrazů větší množství než lze vykreslit, část z nich se sloučí do sebe a hodnota velikosti alokací je zprůměrována. Pro detailní pohled na všechny obrazy procesu je k dispozici interaktivní mód. V tomto módu se zobrazí grafy obrazů v rozmezí velikosti okna, neprovádí se žádné sloučení či aproximace. Rozmezím lze posunovat v časové linii dopředu a zpět, postupným posunem tak lze zobrazit graf libovolného množství obrazů procesu.



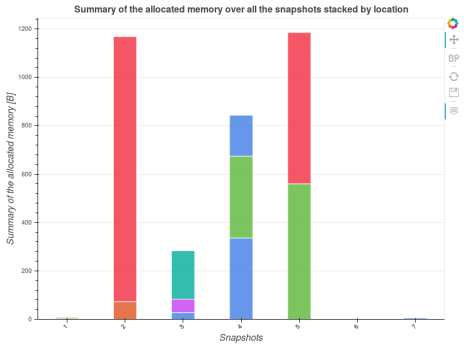
Obrázek 6.4: Znázornění architektury nástroje Perun [8].

Sloupcové grafy První z vizualizací pomocí knihovny Bokeh jsou sloupcové grafy. V nich si lze zobrazit agregovaná alokační metadata. Aktuálně je při vizualizaci vytvořeno šest vybraných sloupcových grafů, které zobrazují zajímavé agregační kombinace metadat, jako je například počet paměťových operací v jednotlivých lokacích programu sdružené podle úseků procesu, ve kterých byly provedeny nebo suma alokované paměti ve všech úsecích procesu sdružená podle standardních alokačních funkcí, které alokaci provedly. Ukázku sloupcového grafu lze vidět na obrázku 6.6.

Plošný graf Poslední vytvořenou vizualizací je plošný graf celkové alokované paměti v každém zachyceném obraze procesu. Suma alokované paměti je v něm vyznačena jak pro každou unikátní lokaci v programu, která některou z alokací provedla, tak i pro součet všech lokací. Graf je interaktivní, pomocí tlačítek lze jednotlivé lokace z grafu zobrazovat a skrývat, najetím kurzoru se zobrazí informace o konkrétní bodě grafu a celý graf lze přibližovat, posouvat či uložit si jeho obrázek. Lze si takto snadno zjistit spotřebu paměti v daný okamžik procesu pro jednotlivé alokační lokace programu, což může upozornit na podezřelou manipulaci s pamětí. Ukázku grafu lze vidět na obrázku 6.5 nebo v příloze D.



Obrázek 6.5: Plošný graf sum alokované paměti pro jednotlivé úseky procesu v každé dílčí alokační lokaci.



Obrázek 6.6: Sloupcový graf sum alokované paměti pro jednotlivé úseky procesu v každé dílčí alokační lokaci.

6.2.4 Možnosti využití existujících vizualizací

Kromě vytváření vlastních nebo jen reimplementací již existujících vizualizací se nabízí možnost využít, již vytvořené vizualizační implementace. Jedna taková existující vizualizace, která je zajímavá pro zobrazení alokačních metadat, je tzv. Flame graf [15]. Tato vizualizace nabízí rychlou a přesnou identifikaci tras vedoucích k sumám provedených alokací paměti. Osa x zobrazuje sumu alokované paměti, osa y ukazuje trasu jednotlivých alokací. Hodnoty alokované paměti pro alokace se shodnými trasami jsou sloučeny do jednoho rámce, jehož šířka poté prezentuje míru účasti na alokacích paměti v celém procesu. Vizualizace produkuje graf ve formátu SVG[6] a nabízí interaktivitu ve formě vyhledávání funkcí v trasách a zaměření konkrétní trasy.

Pro možnost využití této vizualizace byl vytvořen konvertor využívaného profilu (viz podsekce 6.1.1) na formát, který využívá Flame graf vizualizace převzata z [15].

[6] *Scalable Vector Graphics* je značkovací jazyk a formát souboru, který popisuje dvojrozměrnou vektorovou grafiku pomocí XML.

Kapitola 7

Experimentální vyhodnocení

V této kapitole je demonstrována funkcionalita vypracovaného řešení této práce na deseti netriviálních příkladech. Prvních šest experimentů (viz sekce 7.1) se zaměřuje na ověření základní funkcionality profilovací knihovny — schopností profilovat program a generovat výsledný profil — a vizualizační knihovny — schopností interpretovat výsledný profil. Další tři experimenty se zabývají paměťovou analýzou datových struktur (viz sekce 7.2) a poslední experiment demonstruje vliv použití vlastního alokátoru na spotřebu paměti (viz sekce 7.3).

V odevzdaném archivu se nachází Python skript `experiments.py`, který simuluje základní logiku nástroje a umožňuje tak zreprodukovat demonstraci funkcionality řešení spuštěním jednotlivých experimentů. Manuál pro spuštění je přiložen v příloze B. Zmíněný skript provádí pouze profilaci testových programů, pro ověření funkcionality konzolových interpretací profilovacích dat lze využít modul `memory_print`. Tento modul se ovládá pomocí skriptu `memory_print.py` a jeho možnosti jsou popsány v podsekci 6.2.1 nebo nápovědě modulu. K vytváření interaktivních grafů pomocí knihovny Bokeh je k dispozici skript `graphs.py`, jehož ovládání je popsáno v nápovědě skriptu nebo v `README` souboru v odevzdaném archivu.

7.1 Demonstrace základní funkcionality

Pro ověření základní funkcionality bylo zavedeno šest netriviálních projektů v jazyce C/C++. Každý z nich dokazuje schopnost profilovací knihovny zaznamenat alokační metadata a převést je do výsledného jednotného profilu. Profil je poté možné interpretovat všemi dostupnými způsoby vizualizační knihovny.

Experiment 1 V prvním testovacím programu v jazyce C++ je prováděn velký počet iterací. Iterační cykly se nachází ve více funkcích a v každé iteraci je vytvořena prázdná třída. Vytváření tříd probíhá dynamicky pomocí funkce `new` a ihned po vytvoření jsou následně alokované objekty uvolněny.

Experiment 2 Ve druhém testovacím programu v jazyce C je rekurzivně voláno několik funkcí pro výpočet faktoriálu nebo Fibonacciho posloupnosti. Na základě zbývající rekurzivně volané funkci lze demonstrovat přidělování paměti v jednotlivých voláních na mapě haldy (viz obrázky C.1 a C.2), kdy alokované úseky postupně přibývají a po dosažení ukončující podmínky se zase postupně uvolňují ve směru rekurzivního volání.

Experiment 3 Ve třetím testovacím programu je ověřena správná alokační funkcionalita. Program postupně alokuje paměť pomocí všech standardních alokačních funkcí a ověřuje jejich korektní chování splněním vložených podmínek pomocí makra `assert`. Takto je ověřen přístup delegování alokačních požadavků na standardní implementace alokačních funkcí, kterou vytvořená profilovací knihovna využívá. Dále je v programu několikrát alokována paměť pomocí rozdílných alokátorů, jejichž podíl na celkové alokaci paměti procesem lze zobrazit na agregovaných sloupcových grafech (viz obrázky v příloze D).

Experiment 4 V tomto testu je profilován program v jazyce C, který zpracovává grafický obrázek a z jeho dat dekóduje zprávu. Na testu je také poukázáno, že profilovací knihovna je schopna rozeznat alokace paměti v různých zdrojových souborech.

Experiment 5 V tomto testu je profilován program v jazyce C, který pracuje nad protokolem HTTP 1.0. Program slouží pro stažení zadané HTTP stránky. V interpretaci profilu tohoto programu lze vidět četné alokace na stejných adresách (heat mapa na obrázku 7.1) a dalo by se tak uvažovat nad optimalizací opakovaných alokací těchto míst. Taková optimalizace by u webové aplikace mohla být velice žádoucí.

Obrázek 7.1: Heat zobrazení mapy haldy příkladu 5.

Experiment 6 V posledním testu základní funkcionality je profilován program v jazyce C, který na začátku alokuje několik paměťových míst pomocí funkce `malloc` a poté zvětšuje nebo zmenšuje jejich velikosti pomocí funkce `realloc`. Na mapě haldy lze poté zobrazit rozmístění alokované paměti v jednotlivých úsecích programu a demonstrovat tak chování a relokační strategii funkce `realloc`.

7.2 Srovnání datových struktur z hlediska paměti

Vytvořenou interpretací mapy haldy lze dobře vizualizovat rozložení uložených hodnot různých datových struktur a analyzovat tak jejich alokační strategie. Výběr vhodné datové struktury pro konkrétní úlohu programu je důležitým aspektem výkonnostního dopadu na celý tento proces. Navržená mapa haldy může výrazně pomoci při rozhodování a přispět tak k optimalizaci finálního produktu. V následujících třech odstavcích jsou uvedeny analýzy datových struktur `std::list`, `std::vector` a `skip list`.

Experiment 7 V tomto testovém programu v jazyce C++ byly prováděny operace s datovou strukturou `std::list`. V programu byl na začátku vytvořen list a v průběhu do něho byly dynamicky přidávány prvky. Mezi jednotlivými operacemi se strukturou byly prováděny i jiné alokace paměti. Na obrázku C.3 je vidět způsob uložení všech prvků struktury v paměti, barevně odlišených podle různých alokačních lokací.

Experiment 8 Tento testový program v jazyce C++ je stejný jako testový program 7, avšak byly v něm prováděny operace s datovou strukturou `std::vector`. V programu byl vytvořen vektor a v průběhu do něho byly dynamicky přidávány prvky. Na obrázku C.4 je vidět kontinuální uložení celé struktury v paměti, barevně se liší různé alokační lokace.

Experiment 9 Tento testový program v jazyce C++ je stejný jako dva předešlé testové programy, ale byly v něm prováděny operace s datovou strukturou typu `skip list`, jejíž implementace vychází z práce [21]. Paměťové nároky této struktury lze vidět v grafu na obrázku E.1.

Srovnáním těchto vybraných struktur pomocí mapy haldy lze vyvodit řadu závěrů, které mohou pomoci při výběru vhodné datové struktury. Jednotlivé prvky datové struktury `std::list` jsou fragmentované a proto úlohy například s častými operacemi typu prohledávání mohou mít výraznou časovou režii. Prvky datové struktury `std::vector` jsou uloženy vždy kontinuálně, a proto časté operace typu prohledávání tak výraznou časovou režii jako u struktury `std::list` mít nemusí, naopak ale operace typu vkládání mohou mít značně vyšší režii kvůli nutnosti realokace celé struktury v případě nedostatku místa v okolí pro nový prvek. Datová struktura typu `skip list` je založena na podobném principu struktury `std::list` a její prvky jsou tedy také fragmentovány, avšak kvůli odlišnému konceptu propojení prvků není prohledávání této struktury tak časově náročné. Naopak však její paměťová režie je vyšší než u obou předchozích, jak lze ověřit na agregační funkci `sum`.

7.3 Vliv použití vlastního alokátoru

Standardní alokační strategie nemusí být vhodné pro některé typy programů a je tak žádoucí vytvoření vlastní alokační funkce přizpůsobené konkrétním požadavkům. Nejčastěji tyto uživatelské alokační funkce alokují velký úsek paměti standardní alokační funkcí na začátku programu a z tohoto úseku poté přidělují paměť již vlastní alokační strategií. Volba vhodné počáteční velikosti paměti, a zda alokovat paměť dostatečně velkou pro celý proces najednou nebo v kterých úsecích procesu její velikost měnit podle potřeby je otevřeným problémem. Tyto otázky může pomoci vyřešit profilování programu nejprve se standardními alokačními funkcemi, a na základě výsledků zvolit vhodné řešení. Výsledky profilování také mohou značně pomoci s určením vhodné alokační strategie (např. sloupcové grafy četností paměťových operací nebo plošné grafy spotřeby paměti v průběhu procesu).

Experiment 10 Poslední test profiluje program s vlastní alokační funkcí využívající tzv. *pool* strategii. Tento alokátor povoluje jen alokace pevné délky a se zarovnáním, takže operace typu alokace a uvolnění jsou velice rychlé. V programu je nejprve provedena řada alokací standardním alokátorem a poté vlastní alokační funkcí s *pool* strategií. Zpracováním profilu programu agregační funkcí `sum` lze zjistit přibližnou potřebnou velikost paměti k práci, a na základě této hodnoty pak alokovat velikost pro vlastní alokátor. Implementace *pool* alokátoru byla převzata z [4].

7.4 Shrnutí experimentů

Tabulka 7.1 zobrazuje shrnující data profilování deseti provedených experimentů. Sloupec *doba běhu* udává čas běhu experimentu bez provedeného profilování, tj. běžné spuštění procesu. Sloupec *doba běhu profilování* udává čas běhu experimentu v rámci profilovacího modulu. *Doba fáze postprocessu* udává čas potřebný pro zpracování nasbíraných dat a transformování jich do profilu (zahrnuje i časově náročné systémové volání pro překlad adres). Sloupec *suma alokací* udává celkovou velikost procesem požadované paměti a *počet alokací* udává součet všech žádostí o alokaci paměti. U obou posledních sloupců jsou hodnoty vypočítávány z vygenerovaného profilu a hodnoty tak nemusí odpovídat skutečnosti kvůli možným odstraněným záznamům filtrací (např. nelokalizovatelné alokace).

Číslo experimentu	Doba běhu [ms]	Doba běhu profilování [ms]	Doba fáze postprocessu [s]	Suma alokací [B]	Počet alokací
1	2.662	1 228.935	86.158	2 100	2 100
2	1.639	547.004	45.459	50 624	249
3	1.599	200.837	14.729	3 332	432
4	4.256	8.113	0.181	248 368	4
5	994.155	1 010.988	2.968	35 946	67
6	1.889	400.738	30.471	425 880	904
7	2.682	48.043	2.557	4 440	48
8	24.761	26.525	0.673	3972	18
9	25.296	38.161	1.561	4608	46
10	2.837	1 061.749	75.221	136 864	2 002

Tabulka 7.1: Výkonnostní srovnání profilování experimentů.

Hodnoty dob běhu byly naměřeny funkcí `time` z Python modulu `time`, celková velikost alokací a počet alokací byl získán pomocí agregačního textového výstupu funkcí `sum` a `most` modulu `memory_print`. Všechny testy byly úspěšně provedeny na stroji s architekturou x86_64-linux-gnu. Profilovací knihovna a testové programy byly přeloženy překladačem GCC ve verzi 5.4.0.

Kapitola 8

Závěr

V této práci byl popsán koncept paměťového modelu jazyků C a C++ pro pochopení důsledku strategie dynamické alokace na výkon programů napsaných v těchto jazycích. Jsou zde popsány možnosti a průběh dynamických paměťových operací těchto jazyků, způsoby uložení datových objektů v dynamické paměti a práce s nimi. Rovněž byly v práci představeny existující nástroje zabývající se profilováním správy dynamické paměti. Dva z nich byly podrobněji analyzovány — způsob jakým profilují programy, možnosti, které nabízí, jejich výhody i nevýhody a popis práce s nimi.

Cílem práce bylo vytvoření nového způsobu profilování programů, který bude řešit nedostatky stávajících řešení. Výstupem práce je vytvoření profilovacího a vizualizačního modulu pro profilování programů napsaných v jazycích C a C++. Jádrem profilovacího modulu je kolektor alokačních metadat, který sbírá alokační metadata, a poté je transformuje do výkonnostního profilu. Vizualizační modul poskytuje sadu interpretací profilu, jako jsou agregované textové výstupy či vizualizace správy paměti. Na konci práce jsou popsány příklady využití profilace při analýze správy paměti datových struktur nebo vlastních alokačních strategií.

Práce je realizována v rámci nástroje Perun, který lze využít pro komplexní správu výkonu programu. Vytvořené moduly byly úspěšně integrovány do upstreamu. Koncept vytvořeného výkonnostního profilu byl úspěšně využit i v bakalářské práci zabývající se časovou náročností operací dynamických struktur [21]. Tato práce také úspěšně provádí regresní analýzu alokačních metadat z vytvořeného profilovacího modulu.

Vývojáři na základě této práce mohou vytvořit vlastní profilovací nebo vizualizační moduly podle vlastních potřeb a preferencí, například profilování jiného typu dat nebo profilování programů v jiných jazycích. Dokonce s dodržením konceptu jednotného profilu mohou využít vytvořené moduly této práce.

Literatura

[1] backtrace(3) - Linux manual page. [online], navštíveno 4.3.2017.
URL http://man7.org/linux/man-pages/man3/backtrace.3.html

[2] dlsym(3) - Linux man page. [online], navštíveno 5.3.2017.
URL https://linux.die.net/man/3/dlsym

[3] Bokeh Development Team: *Bokeh: Python library for interactive visualization*. 2014, navštíveno 20.4.2017.
URL http://www.bokeh.pydata.org

[4] Costa, T.: C++: Custom memory allocation. 2013.
URL https://www.gamedev.net/resources/_/technical/general-programming/c-custom-memory-allocation-r3010

[5] *Curses Programming with Python — Python 3.6.1 documentation*. Navštíveno 20.4.2017.
URL https://docs.python.org/3/howto/curses.html#curses-howto

[6] DWARF Standards Committee: Dwarf Home. [online], navštíveno 22.1.2017.
URL http://dwarfstd.org/Home.php

[7] Fewk, S.: *Information Dashboard Design*. O'Reilly, 2006, ISBN 0596100167.

[8] Fiedor, T.: Perun: Lightweight Performance Control System. [online], navštíveno 20.4.2017.
URL https://github.com/tfiedor/perun

[9] Franěk, F.: *Memory as a Programming Concept in C and C++*. Cambridge University Press, 2004, ISBN 9780511838620.

[10] GCC, the GNU Compiler Collection. [online], navštíveno 4.3.2017.
URL https://gcc.gnu.org/

[11] Using the GNU Compiler Collection (GCC): Return Address. [online], navštíveno 4.3.2017.
URL https://gcc.gnu.org/onlinedocs/gcc/Return-Address.html

[12] GDB: The GNU Project Debugger. [online], navštíveno 23.1.2017.
URL https://www.sourceware.org/gdb/

[13] The GNU Operating System and the Free Software Movement. [online], 2017, navštíveno 5.3.2017.
URL https://www.gnu.org/

[14] The GNU C Library (glibc). [online], navštíveno 4.3.2017.
URL https://www.gnu.org/software/libc/

[15] Gregg, B.: Flame Graphs. [online], navštíveno 25.1.2017.
URL http://www.brendangregg.com/flamegraphs.html

[16] malloc_hook(3) - Linux manual page. [online], navštíveno 4.3.2017.
URL http://man7.org/linux/man-pages/man3/malloc_hook.3.html

[17] Intel Corporation: Intel® VTune™ Amplifier | Intel® Software. [online], navštíveno 2.2.2017.
URL https://software.intel.com/en-us/intel-vtune-amplifier-xe

[18] JSON. [online], navštíveno 20.4.2017.
URL http://www.json.org/json-cz.html

[19] ld.so(8) - Linux manual page. [online], navštíveno 25.1.2017.
URL http://man7.org/linux/man-pages/man8/ld.so.8.html

[20] The libunwind project. [online], navštíveno 25.1.2017.
URL http://www.nongnu.org/libunwind/

[21] Pavela, J.: *Knihovna pro profilování datových struktur programů C/C++*. Bakalářská práce, Vysoké učení technické v Brně, Fakulta informačních technologií, Brno, 2017, Vedoucí práce Fiedor Tomáš.

[22] Perf Wiki. [online], navštíveno 28.1.2017.
URL https://perf.wiki.kernel.org/index.php/Main_Page

[23] Python Core Team: *Python: A dynamic, open source programming language*. Python Software Foundation, 2015.
URL https://www.python.org/

[24] Tosic, M.: MTuner - C/C++ memory profiler. [online], 2016, navštíveno 28.1.2017.
URL http://mtuner.net/index.html

[25] Mark J. Wielaard » Blog Archive » Stack unwinding. [online], navštíveno 17.4.2017.
URL https://gnu.wildebeest.org/blog/mjw/2007/08/23/stack-unwinding/

[26] Valgrind™ Developers: Valgrind. [online], navštíveno 21.1.2017.
URL http://valgrind.org/docs/manual/ms-manual.html

[27] Valgrind™ Developers: Valgrind. [online], navštíveno 22.1.2017.
URL http://valgrind.org/

[28] Wikipedia: Name mangling — Wikipedia, The Free Encyclopedia. [online], 2016, navštíveno 5.3.2017.
URL https://en.wikipedia.org/w/index.php?title=Name_mangling&oldid=7550198074

[29] Wikipedia: Inline function — Wikipedia, The Free Encyclopedia. [online], 2017, navštíveno 5.3.2017.
URL https://en.wikipedia.org/w/index.php?title=Inline_function&oldid=759827534

[30] Wolff, M.: Heaptrack. [online], navštíveno 25.1.2017.
URL https://github.com/KDE/heaptrack

[31] Wolff, M.: Massif Visualizer. [online], navštíveno 22.1.2017.
URL https:
//www.linux-apps.com/content/show.php/Massif+Visualizer?content=122409

Přílohy

Příloha A

Obsah paměťového média

Obsahem přiloženého CD jsou adresáře s následujícím obsahem:

- `collect/` — složka se zdrojovými soubory profilovacího modulu,
- `view/` — složka se zdrojovými soubory vizualizačního modulu,
- `experiments/` — složka s jednotlivými testy,
- `latex/` — složka se zdrojovými soubory textové práce,
- `experiments.py` — testovací Python skript simulující logiku nástroje Perun sloužící pro reprodukci demonstrativních testů,
- `memory_print.py` — spouštěcí skript modulu `memory_print` pro konzolovou interpretaci profilu,
- `graphs.py` — skript pro ovládání generování interaktivních grafů pomocí knihovny Bokeh,
- `LICENSE` — licenční soubor pro užití práce,
- `README` — textový soubor popisující spuštění jednotlivých modulů práce.

Příloha B

Manuál k ovládání modulů

Pro vytvořené moduly jsou v odevzdaném archivu poskytnuty ovládací skripty v jazyce Python, které se spouští interpretem `python` nebo `python3`.

B.1 Profilovací modul

Profilovací modul se ovládá skriptem `experiments.py`, který simuluje logiku nástroje Perun a poskytuje tak jednoduché API pro komunikaci s ním. Skript umožňuje přeložení jednotlivých testů a profilovací knihovny a spuštění jednotlivých testů pro reprodukování demonstrace funkcionality modulu. Skript lze spustit s následujícími možnými parametry:

- `-b(--build) [-e(--experiment=)k]` — provede překlad experimentu `k`. Pokud nebyl experiment specifikován, provede se překlad všech experimentů.
- `-c(--clean) [-e(--experiment=)k]` — provede vyčištění experimentu `k` v případě dat z minulého spuštění. Pokud nebyl experiment specifikován, provede se vyčištění všech experimentů.
- `-r(--run) [-e(--experiment=)k]` — provede se profilování experimentu `k`. Pokud nebyl experiment specifikován, provede se profilování všech experimentů.

Číslo experimentu `k` musí být v rozsahu počtu experimentů, tj. 1-10. Po provedení profilování jsou vygenerované profily přesunuty do složky s příslušným experimentem. Příklad spuštění skriptu:

```
$ python3 experiments.py --build -e 5
```
(B.1)

```
$ python3 experiments.py --experiment=5
```
(B.2)

B.2 Modul memory_print

Modul pro konzolovou interpretaci se ovládá skriptem `memory_print.py`. Jednotlivé možností volání modulu jsou popsány v podsekci 6.2.1. Příklad spuštění skriptu:

```
$ python3 memory_print.py --mode=func --all profile.perf
```
(B.3)

```
$ python3 memory_print.py --mode=top -t 3 profile.perf
```
(B.4)

```
$ python3 memory_print.py -m heap profile.perf
```
(B.5)

B.3 Modul pro tvorbu interaktivních grafů

Modul tvořící interaktivní grafy pomocí knihovny Bokeh se ovládá skriptem `graphs.py`. Skript lze spustit s následujícími možnými parametry:

- `[--bars] [--flow] [--flame]` — vytvoří všechny zvolené grafy pro zadaný profil. Pokud nebyl specifikován žádný graf, vytvoří se všechny dostupné grafy.

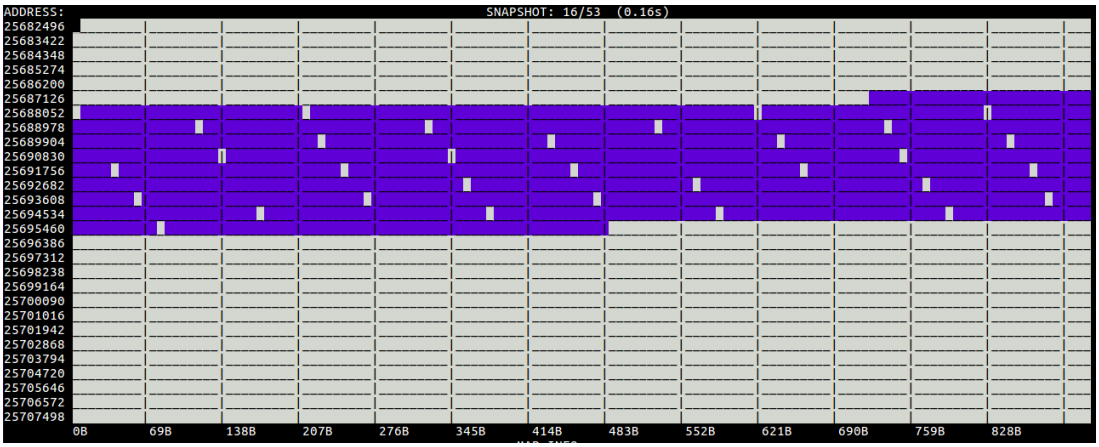
Po vygenerování jsou grafy přesunuty do složky se zvoleným profilem. Název souboru s grafem odpovídá názvu typu grafu (např. `bars.html` nebo `flame.svg`). Příklad spuštění skriptu:

```
$ python3 graphs.py --bars profile.perf
```
(B.6)

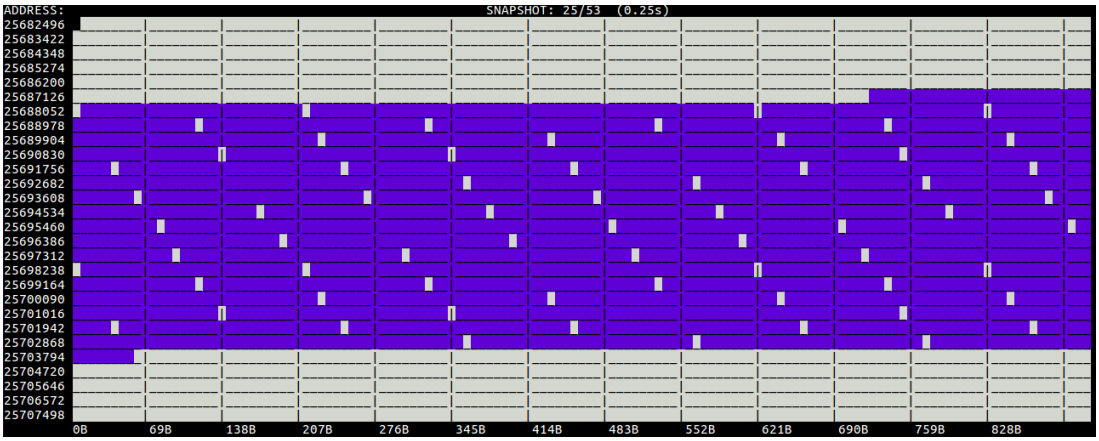
```
$ python3 graphs.py --flame --flow profile.perf
```
(B.7)

Příloha C

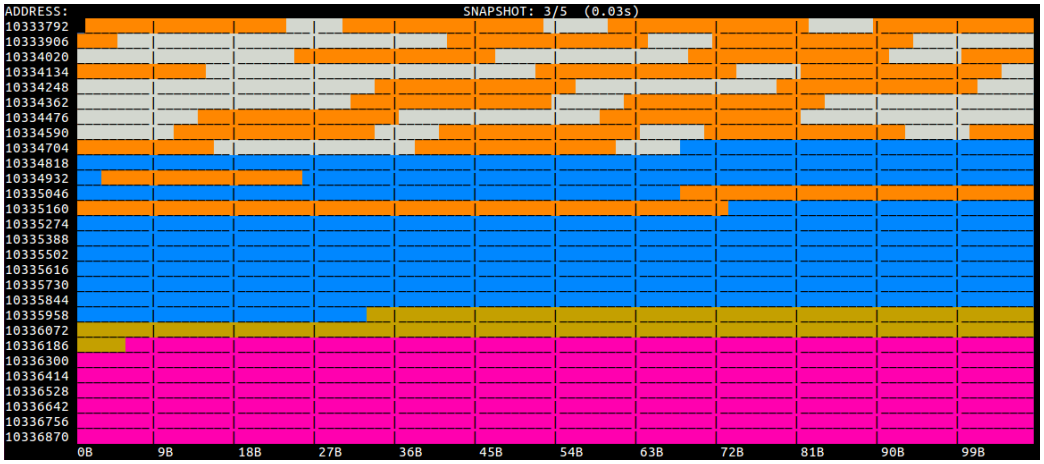
Mapy haldy



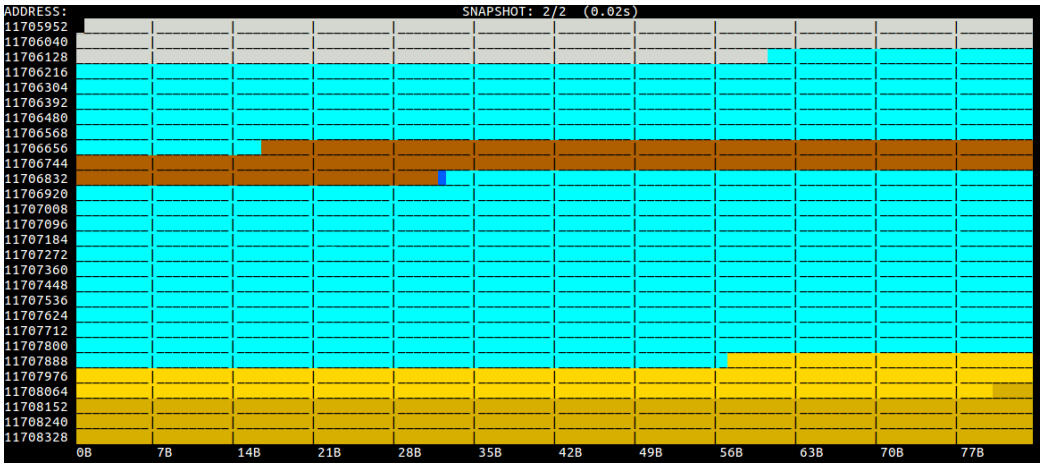
Obrázek C.1: Mapa haldy testu 2 na počátku rekurzivního volání.



Obrázek C.2: Mapa haldy testu 2 před ukončující podmínkou rekurze.



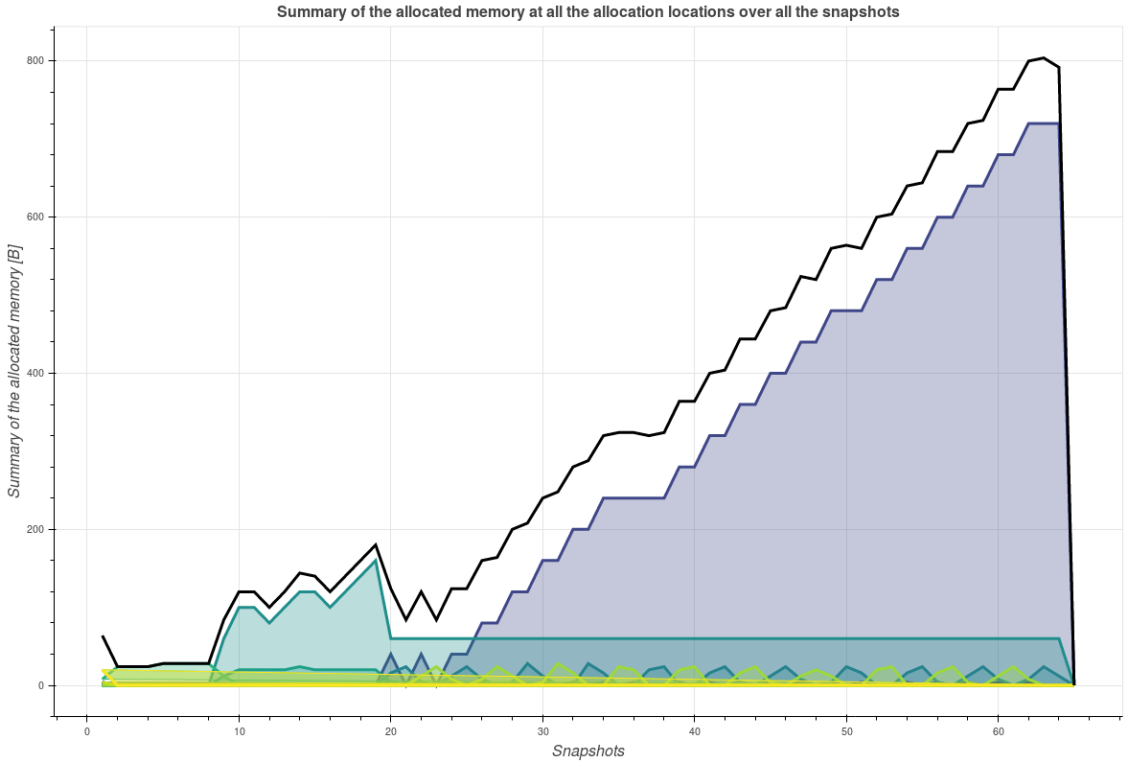
Obrázek C.3: Rozložení prvků datové struktury `std::list` v paměti. Prvky struktury jsou označeny oranžovou barvou.



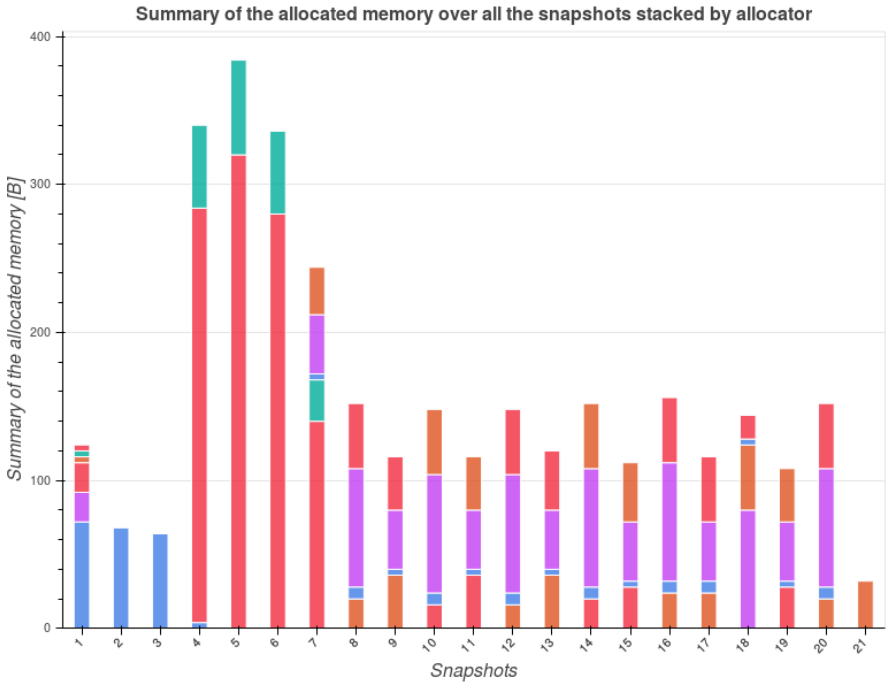
Obrázek C.4: Rozložení prvků datové struktury `std::vector` v paměti. Prvky struktury jsou označeny hnědou barvou.

Příloha D

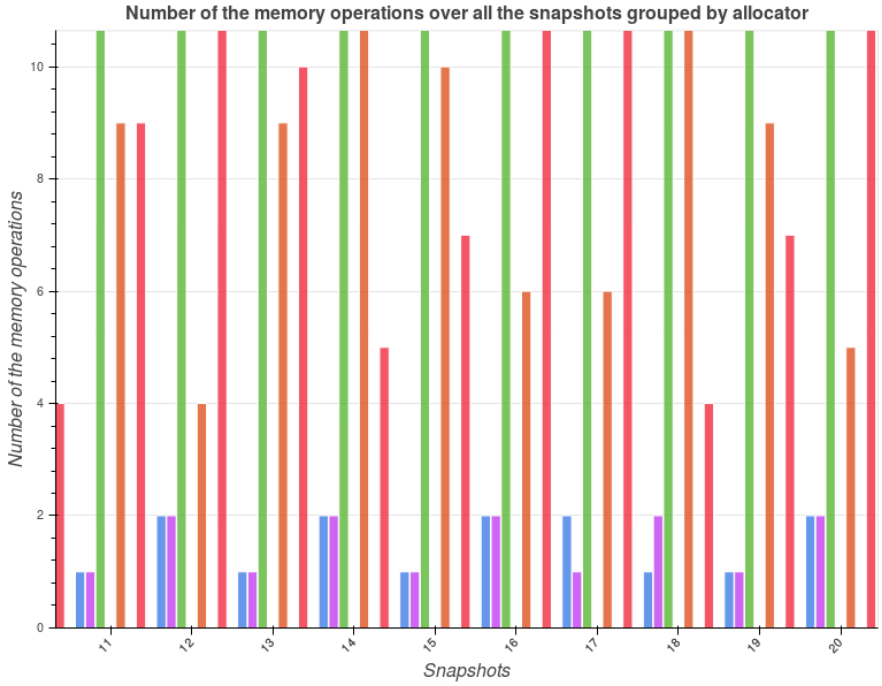
Interaktivní grafy



Obrázek D.1: Plošný graf celkové spotřeby paměti.



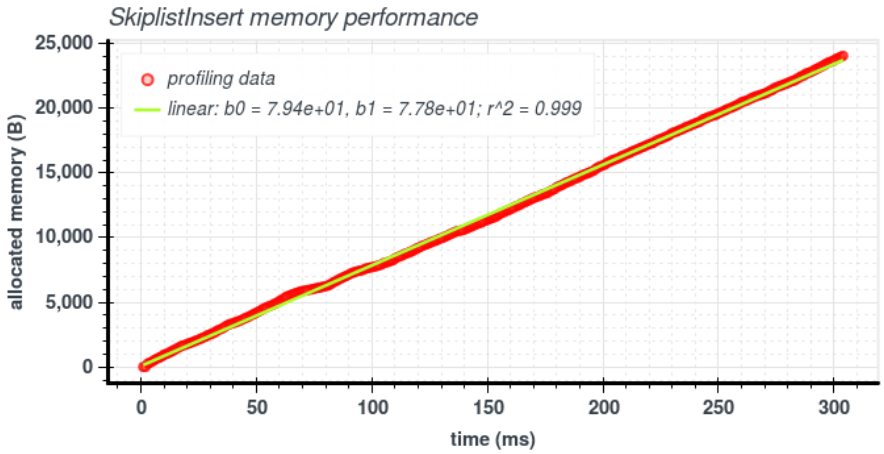
Obrázek D.2: Sloupcový graf sum alokované paměti agregovaný podle alokační funkce, která alokaci provedla.



Obrázek D.3: Sloupcový graf počtu paměťových operací agregovaný podle alokační funkce, která operaci provedla.

Příloha E

Graf regresní analýzy



Obrázek E.1: Regresní analýza paměťových dat vygenerovaných vytvořeným kolektorem alokačních metadat vizualizovaná pomocí výstupu práce [21].