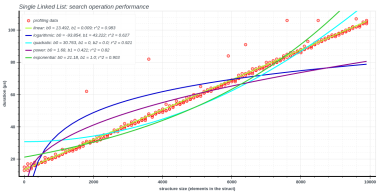
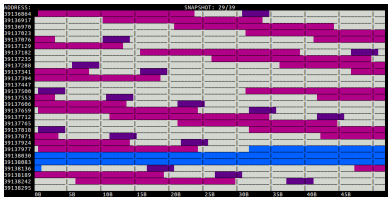
BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Excel@FIT 2017
http://excel.fit.vutbr.cz

Profilace bez komplikace: Profilovací knihovny pro programy v C/C++

Radim Podola*, Jiří Pavela**





Abstrakt

Chování programu z hlediska výkonu je důležitou, a pro některé třídy programů až kritickou stránkou běhu. V této práci se zaměřujeme na hledání výkonnostních chyb programu, jejich názornou interpretaci uživateli a nové efektní způsoby vizualizace spotřeby zdrojů. Práce se zaměřuje na dvě oblasti — správu operační paměti a časovou složitost operací nad datovými strukturami. Pro každý z problémů je vytvořena vlastní profilovací knihovna, která získává profilovací informace o výkonnostní stránce programu, a poté je interpretuje uživateli. Knihovny jsou navrženy jako snadno modifikovatelné a integrovatelné do komplexnějšího profilovacího nástroje, a zcela navzájem nezávislé. Při vizualizaci výsledků hledáme nové způsoby jejich interpretace pro názornější pohled na výkonnostní stránku programu, které mohou uživateli značně usnadnit nalezení výkonnostních chyb. Popisujeme zvolení principů kolekce dat vedoucí k minimální výpočetní režii profilování, a zároveň poskytující dostatečné množství užitečných informací.

Klíčová slova: Profilování — Operační paměť — C/C++ — Datové Struktury — Vizualizace — Lineární Regrese

Přiložené materiály: Git repozitář Radim Podola — Git repozitář Jiří Pavela

*xpodol06@stud.fit.vutbr.cz, *Faculty of Information Technology, Brno University of Technology*
**xpavel32@stud.fit.vutbr.cz, *Faculty of Information Technology, Brno University of Technology*

1. Úvod

Pojem *bug* vnímá velká část programátorů a vývojářů pouze jako chybu v kódu, která způsobí nekorektní výsledky výpočtu. Jako bug však můžeme označit i úsek kódu, jehož provedení zabere výrazně více času, nebo spotřebuje více paměti než je očekávané. Tyto chyby označujeme jako tzv. *performance bugs*[1], přičemž jejich přítomnost ve zdrojovém kódu může rapidně snížit kvalitu aplikace a uživatelův dojem z ní. V případě bezpečnostně kritických systémů mohou mít tyto chyby přímo nedozírné následky. Je proto důležité umět tyto chyby spolehlivě odhalit a lokalizovat.

Tento úkol se však v praxi ukazuje jako obtížně řešitelný. Zatímco chyby ve výpočtech se často projeví špatným výsledkem nebo havárií systému, výkonnostní chyby mohou po většinu doby zůstávat skryté. Problém nekorektní (neefektivní) manipulace s pamětí nebo nevhodného návrhu (případně implementace) algoritmu nemusí být na první pohled patrný, ale jeho kumulativní dopad může způsobit komplikace. Tyto chyby nemohou být automaticky odstraněny optimalizacemi při překladu [1], což přenáší zodpovědnost za jejich nalezení a opravení na tvůrce softwaru.

Identifikace a lokalizace výkonnostních chyb manuálním způsobem klade na vývojáře vysoké nároky především z hlediska hlubší znalosti zdrojového kódu,

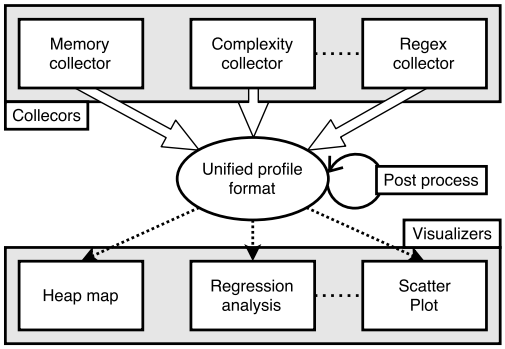
programovacího jazyka, operačního systémů a jeho jednotlivých vnějších vrstev. U tohoto druhu chyb mohou hrát roli rozličné faktory jako například adresování nebo lokalita odkazů a s ní související účinnost *cache* paměti.

Cílem naší práce je umožnit uživateli analyzovat software a získané informace přehledně vizualizovat za použití nových, námi navržených technik. Díky těmto informacím bude uživatel schopen nalézt a lokalizovat případné výkonnostní chyby, a tedy lépe navrhnout a fundovaněji zvolit vhodné datové struktury pro řešený problém. Výsledné knihovny jsou navíc součástí komplexnějšího celku, v jehož rámci bude možné provádět např. verzování výsledných analýz. Nástroje nabízejí ve svých oblastech rychlejší (za cenu možného snížení přesnosti) alternativu ke stávajícímu řešení a mají velký potenciál pro další vývoj.

Existující nástroje. V současnosti existuje několik nástrojů, které mohou být využity pro profilování kódu a hledání výkonnostních chyb. Velmi rozšířeným je *Valgrind's Tool Suite* [2], který poskytuje sadu profilovacích nástrojů, mimo jiné například *Memcheck* (analýza problémů spojených se správou paměti), *Massif* (profilování dynamické paměti), *Cachegrind* (profilovací nástroj pro cache paměť) nebo *Callgrind* (umožňující profilování programu z hlediska času). Tyto nástroje jsou velmi robustním řešením a ve většině případů produkují spolehlivé výsledky. Přesné výsledky jsou však za cenu výrazného prodloužení doby běhu programu ([2] uvádí zpomalení běhu až *20–100x* v případě Cachegrindu). Tato omezení mohou být pro celou řadu programů nepřípustná. Rovněž grafická vizualizace není oficiální součástí balíku nástrojů a vizualizaci tak lze pro některé nástroje provádět s využitím dodatečných aplikací.

Naše řešení. Výsledkem naší práce jsou nové nástroje pro analýzu správy operační paměti a odhad výkonnosti datových struktur nebo algoritmů. Řešení sestává ze dvou částí — sběru profilovacích dat (sekce 2) v jednotném universálním formátu a z nových přístupů pro vizualizaci spotřeby zdrojů (jako je například interaktivní mapa haldy) a výkonu datových struktur (pomocí automatizované regresní analýzy a přibližného odvození složitosti z naměřených dat). Práce je navíc realizována v rámci nového open source projektu *Perun* [3] — odlehčené platformy pro výkonnostní verzování programů — vyvíjeného vedoucím práce.

Konceptuálně se knihovny skládají ze tří klíčových prvků — *kolektorů dat*, jednotného profilovacího formátu a *vizualizačních modulů*. Kolektory zajišťují sběr vybraných profilovacích dat ze zkoumaného programu



Obrázek 1. Schematické znázornění konceptu profilovacích knihoven a jejich jednotlivých částí — kolektorů dat, jednotného formátu profilovacích dat, post-processing fází a vizualizérů.

a jejich uložení v jednotném formátu. Profilovací data je možné následně dále zpracovávat pomocí tzv. *post-processing modulů* a výsledky tak například vizualizovat nebo nad nimi provádět regresní analýzu. Jednotný formát pro uložení profilovacích dat je stěžejní částí knihoven umožňující bezproblémovou (zejména z hlediska nekompatibilních formátů dat) rozšiřitelnost o další *kolektory* nebo *vizualizéry*. Obrázek 1 schematicky znázorňuje koncept profilovacích knihoven.

2. Sběr profilovacích dat

Přední částí naší knihovny je sběr profilovacích dat, realizovaný pomocí *kolektorů*, které jsou implementačně založeny na odlehčené instrumentaci analyzovaných programů. Návrh a implementaci kolektorů provází rovněž snaha o minimalizaci závislostí na externích knihovnách a jednoduchost použití.

Produkovaná profilovací data jsou uložena v jednotném profilovacím formátu založeném na *JSON* notaci. Profil kromě dat, obsahuje také obecné informace o průběhu profilování. Detailnější popis viz [3].

V rámci řešení vznikly dva kolektory — kolektor dat dynamických paměťových operací spolu s kolektorem časových a rozměrových dat datových struktur — které byly integrovány v rámci nástroje Perun.

2.1 Kolektor dat dynamických operací

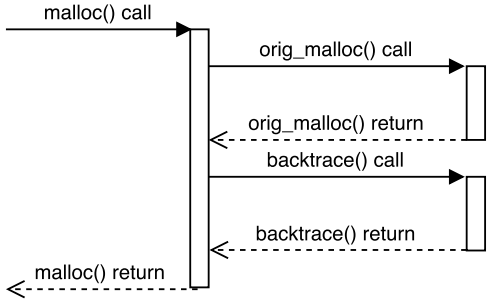
Kolektor sbírá informace o dynamických operacích s operační pamětí, které jsou potřebné pro analýzu a následné vybudování detailnějšího pohledu na správu paměti procesem. Tyto operace jsou prováděny za běhu aplikace a nelze je tudíž analyzovat staticky.

Principem dynamického záznamu operací v naší knihovně je předefinování standardních alokačních funkcí (jako je malloc, calloc, free, ...). Přepsáním jejich implementace docílíme získání profilovacích dat bez potřeby měnit zdrojový kód aplikace nebo aplikaci specificky překládat. Kolektor má k dispozici

dynamickou knihovnu s vhodně upravenými implementacemi těchto funkcí pro potřeby profilování. Díky tomu, že implementace těchto funkcí se do kódu aplikace nahrávají až za běhu, neboť v době překladu jsou v kódu pouze reference na ně, lze nahradit jejich implementace i v již přeložených aplikacích, a to dokonce i za běhu dané aplikace. Tento princip není závislý na použitém překladači, jedinou podmínkou je shodné API[1] i ABI[2] knihovny.

Pro užitečnější interpretaci záznamů kolektor získává další informace jako je název funkce, která alokaci provedla, trasu volaných funkcí, která k alokaci vedla a odpovídající časové razítko operace v průběhu programu.

Princip průběhu alokace při profilování je znázorněn na obrázku 2. V upravené alokační funkci je nejprve delegována požadovaná operace na původní alokační funkci (volání funkce `orig_malloc()`), poté jsou zaznamenány základní profilovací informace a nakonec je zaznamenána trasa volaných funkcí (volání funkce `backtrace()`).



Obrázek 2. Znázornění proudu programu při volání upravené alokační funkce `malloc()`

Trasa je získána pomocí knihovny *libunwind* [4]. Ta poskytuje aplikační rozhraní pro přístup k programovému zásobníku, kde v každém rámci jsou uloženy hodnoty registrů počítače (reprezentace stavu programu). Přečtením IP[3] 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 relativní adresou instrukce kódu spustitelného souboru a není tudíž interpretovatelná přímo. Adresy se proto dodatečně překládají do čitelnější podoby ve formě jmen funkcí ve zdrojovém souboru, názvů zdrojových souborů a čísel řádků v něm, kde k volání funkce došlo. Pro získání těchto informací je však na většině systémech nutné přeložit program s *debugovacími symboly*.

[1] Application Programming Interface

[2] Application Binary Interface

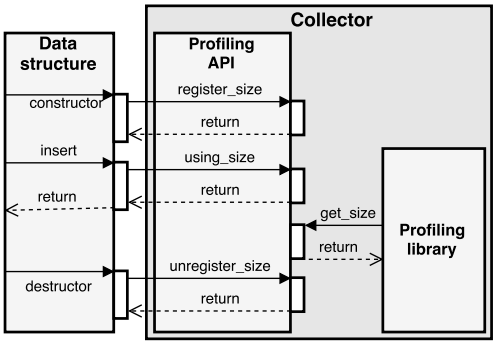
[3] *Instruction Pointer* je speciální registr v procesoru, který adresuje aktuálně prováděnou instrukci strojového kódu v operační paměti

V samotném průběhu profilování kolektor pouze ukládá veškeré profilovací data do souboru bez úprav a zmíněných překladů. Až později jsou tato data upravována a převáděna do profilu. Převod lze parametrizovat nastavením vzorkovací frekvence sbíraných dat na základě časového razítka a filtračních pravidel pro odstranění nežádoucích záznamů. Tento přístup výrazně snižuje negativní dopad na výkon profilování (mírní tak i nepřesnost časových razítek), umožňuje využít k překladu jiných metod, parametrizovat profilovací data několika způsoby bez nutnosti opakovat profilaci nebo vytvářet odlišné výsledné formáty profilu bez nutnosti zasahovat do jádra kolektoru.

2.2 Kolektor dat složitostí

Teoretické odvození časové složitosti algoritmu probíhá formou analýzy jeho zdrojového kódu. Doba běhu algoritmu je charakterizována funkcí velikosti jeho vstupních dat. Při tomto způsobu odvození se pracuje s parametry času a velikosti dat na abstraktní úrovni, kde je čas reprezentován počtem provedených primitivních operací (kroků) a velikost struktury je reprezentována jako počet prvků n [5]. Pro empirické odvození složitosti je však potřeba získat skutečné hodnoty těchto parametrů.

Algoritmy jsou v našem případě jednotlivé operace nad datovými strukturami, které jsou ve zdrojovém kódu zapsány formou funkcí nebo metod. GCC (GNU Compiler Collection) poskytuje možnost instrumentovat funkce pomocí vkládání (tzv. *injection*) speciálních funkcí do vstupního a výstupního bodu instrumentované funkce [6]. V rámci instrumentačních funkcí jsou získávána časová razítka, pomocí nichž je možné odvodit dobu běhu funkce.



Obrázek 3. Znázornění konstrukce nové struktury s registrací, volání operace nad strukturou s notifikací a destrukce struktury se zrušením registrace. Registrace struktury informuje knihovnu o profilované struktuře a pomocí notifikací dochází ke zjišťování dat o velikosti dané struktury

Pro každou naměřenou hodnotu doby trvání je potřeba mít i údaj o velikosti struktury, nad kterou oper-

ace pracovala, abychom byli schopni odvodit závislost času na množství dat. Ke zjištění velikosti struktury slouží navržené profilovací API, které je součástí knihovny. Základem profilovacího API jsou funkce pro registraci, notifikaci a zrušení registrace datové struktury. Princip těchto funkcí je ilustrován obrázkem 3.

Profilovací API je navrženo s ohledem na snadnou rozšiřitelnost např. o klasickou analýzu amortizované složitosti [7] nebo monitorování spotřeby dalších zdrojů (kam spadá např. práce se soubory). Díky tomu bude API využitelné napříč množstvím profilovacích nástrojů, případně dostupné přímo uživateli.

3. Regresní analýza datových struktur

Regresní analýza je statistický proces, kterým se snažíme odhadnout vztah mezi (obecně) několika proměnnými. V případě, že takový vztah existuje, tak jsme následně schopni odvodit hodnoty *závislých proměnných* na základně znalosti hodnot *nezávislých proměnných* v daném vztahu, který vyjadřujeme *regresní funkcí*. V případě odvození složitosti algoritmů nad datovými strukturami provádíme regresní analýzu nad dvěma proměnnými — *nezávislou* velikostí struktury a *závislou* dobou běhu algoritmu — s cílem nalézt model, který co nejpřesněji vystihuje vztah těchto proměnných.

Problém je řešitelný pomocí *metody nejmenších čtverců*, jejíž princip spočívá v nalezení parametrů modelové funkce $f(x,\beta)$ (kde β je vektor parametrů) takových, aby součet čtverců chyb R_i nabýval minima. Chyba R_i je definována jako rozdíl naměřené hodnoty y_i a hodnoty regresního modelu $\widehat{y_i}$ v daném bodě.

$$S = \sum_{i=1}^{n} R_i^2 \qquad \widehat{y_i} = f(x_i,\beta) \qquad R_i = y_i - \widehat{y_i} \tag{1}$$

Různé regresní modely se však liší ve vzorci, pomocí kterého tuto optimalizační úlohu řeší. Naším cílem bylo nalézt obecný vzorec, který by byl aplikovatelný na co nejširší spektrum regresních modelů, aniž by došlo k příliš velké ztrátě přesnosti. Rovnice 2 a 3 reprezentují obecný vzorec pro výpočet dvou koeficientů $\hat{\beta}_0$ a $\hat{\beta}_1$, kde $f(x)$ je obecná funkce jedné proměnné. Zajímá nás především *Big-O*[4] složitost a z toho důvodu se zaměřujeme pouze na dva koeficienty pro dosažení dostačujících výsledků.

$$\hat{\beta}_0 = \frac{\sum y_i - \hat{\beta}_1 \sum f(x_i)}{n} \tag{2}$$

$$\hat{\beta}_1 = \frac{n\sum f(x_i)y_i - \sum f(x_i)\sum y_i}{n\sum (f(x_i))^2 - (\sum f(x_i))^2} \tag{3}$$

[4] Složitost algoritmu v nejhorším případě

Obecný vzorec však u některých modelů neprodukuje dostatečně přesné výsledky a proto jsou tyto modely počítány pomocí speciálních vzorců pro daný model (např. mocninný nebo exponenciální model). Zpřesňování obecného vzorce je předmětem dalšího vývoje.

Při odhadu složitosti algoritmu je nutné zjistit, který regresní model nejlépe odpovídá naměřeným datům. Vhodnost použitého modelu na proložení dat je vyjádřena pomocí tzv. *coefficient of determination* [8], značeném jako r^2 a vyjádřeném pomocí vztahu 4. Čím víc se koeficient r^2 blíží hodnotě 1, tím přesněji vyjadřuje proložený model závislost mezi body.

$$r^2 = 1 - \frac{\sum (y_i - \hat{y}_i)^2}{\sum (y_i - \bar{y})^2} \tag{4}$$

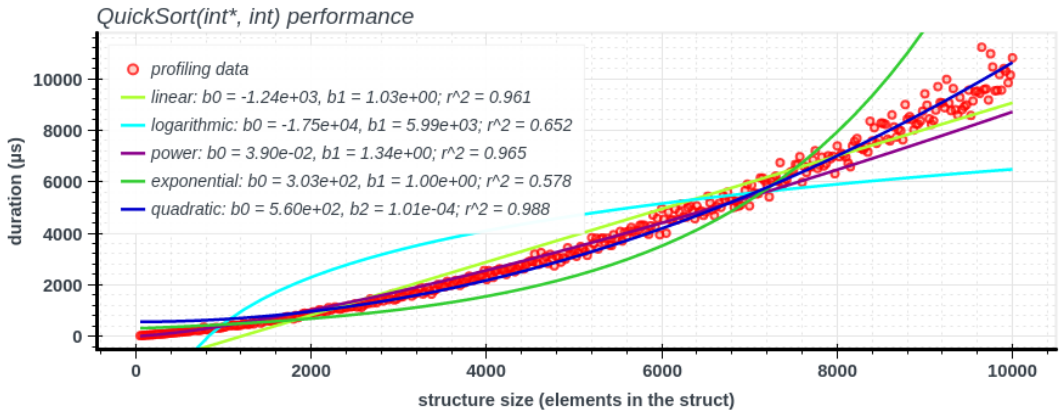
Nástroj implementuje několik technik pro nalezení nejvhodnějšího modelu. Jednou z technik je kompletní výpočet, kdy jsou použity všechny naměřené body pro výpočet všech regresních modelů zajímavých z hlediska složitostí (lineární, logaritmický, kvadratický atd.). Následně je model s nejvyšší hodnotou koeficientu r^2 vybrán jako nejvhodnější reprezentace složitosti daného algoritmu. Výstup této techniky je ilustrována obrázkem 4.

Iterativní technika lineární regrese. Technika kompletního výpočtu však může být výpočetně náročná při velkém počtu dat a zkoumaných modelů. Tento problém adresuje další technika, zvaná iterativní. Iterativní technika se snaží zpřesňovat ten regresní model, který se jeví jako nejslibnější. Výpočet je prováděn po menších celcích (určité % všech naměřených bodů), které jsou voleny náhodně. Celky jsou následně prokládány modely a pouze výpočet modelu, který má nejvyšší hodnotu koeficientu r^2 je rozvíjen o další celky. Výhodou je využití mezivýpočtů z předchozích celků, čímž je dosažena vyšší rychlost výpočtů. Nevýhodou popsané techniky je možné nalezení lokálního extrému, a tedy nemusí být nutně nalezen nejpřesnější model.

4. Efektivní vizualizace profilovací dat

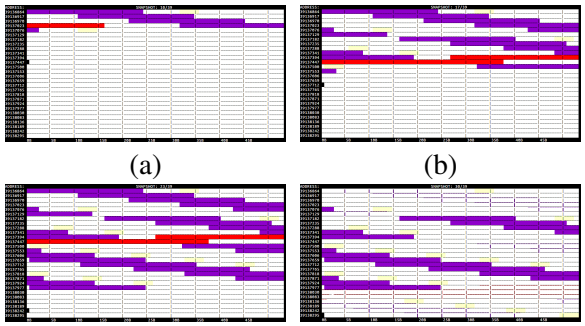
Další součástí našich profilovacích knihoven je interpretace profilovacích dat pomocí vhodných vizualizací. Naše vizualizace dat jsou blízké informačním *dashboardům* [9]. Problémem při jejich vytváření je návrh, který ke splnění svého účelu musí zobrazit správný rozsah informací na malém zobrazovacím prostoru způsobem pro jasnou a rychlou interakci s daty.

V našich knihovnách se snažíme o generický návrh některých z vizualizačních metod tak, aby bylo možné jejich použití i pro jiné profilovací směry (data).



Obrázek 4. Technika kompletního výpočtu nad algoritmem *Quicksort*. Vstup je na pomezí nejhoršího a průměrného případu díky šumu v podobě náhodných prvků v poli. Kvadratický model dosahuje nejlepší hodnoty r^2, čímž je potvrzen teoretický předpoklad, že algoritmus nabývá složitosti n^2 pro pole s opačně seřazenými prvky navzdory vloženému šumu. Odhad složitosti lze provádět obecně pro algoritmy pracující nad proměnným počtem dat a nejen nad operacemi datových struktur.

Jednotný formát profilovacích dat pak zlehčuje jejich implementaci a není tudíž náročné vytvořit si vlastní vizualizaci získaných dat podle specifických potřeb bez nutnosti zasahovat do ostatních částí knihovny.



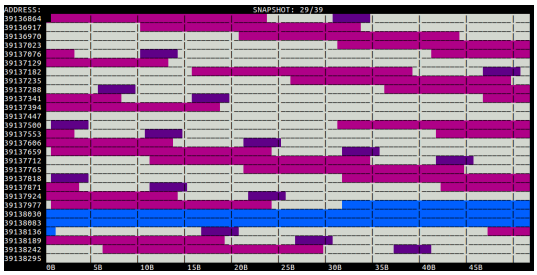
Obrázek 5. Na snímku *(a)* jsou viditelné alokace několika paměťových míst, patřících dvěma alokačním místům v programu, na následujícím snímku *(b)* je vidět uvolnění menšího paměťového prostoru a zároveň alokace většího prostoru, patřící prvnímu (červenému) místu programu. Na snímku *(c)* pouze druhé (fialové) místo programu alokovalo další paměťové úseky. Na posledním snímku *(d)* první místo úplně a druhé místo částečně uvolnilo alokované úseky.

Mapa haldy Jednou ze zatím hotových vizualizací je interaktivní mapa haldy. Ta přibližuje pohled na spotřebu a manipulaci s dynamickou pamětí za dobu běhu programu. Tato vizualizace zobrazuje dynamický úsek operační paměti počítače přidělený procesu v podobě mapy, kde jednotlivé paměťové úseky jsou v této mapě znázorněny barevnými poli. 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 časovému snímku běhu programu. Postupným přehráním všech map (ručně nebo využitím možnosti *Animace*) tak lze sledovat celý průběh paměti programu (náznak průběhu lze postupně vidět na snímcích *(a)* až *(d)* obrázku 5).

Vizualizace by měly vést ke snadnějšímu pochopení běhu programu, přesněji jak program pracuje a jak využívá systémové zdroje. V mapě haldy lze:

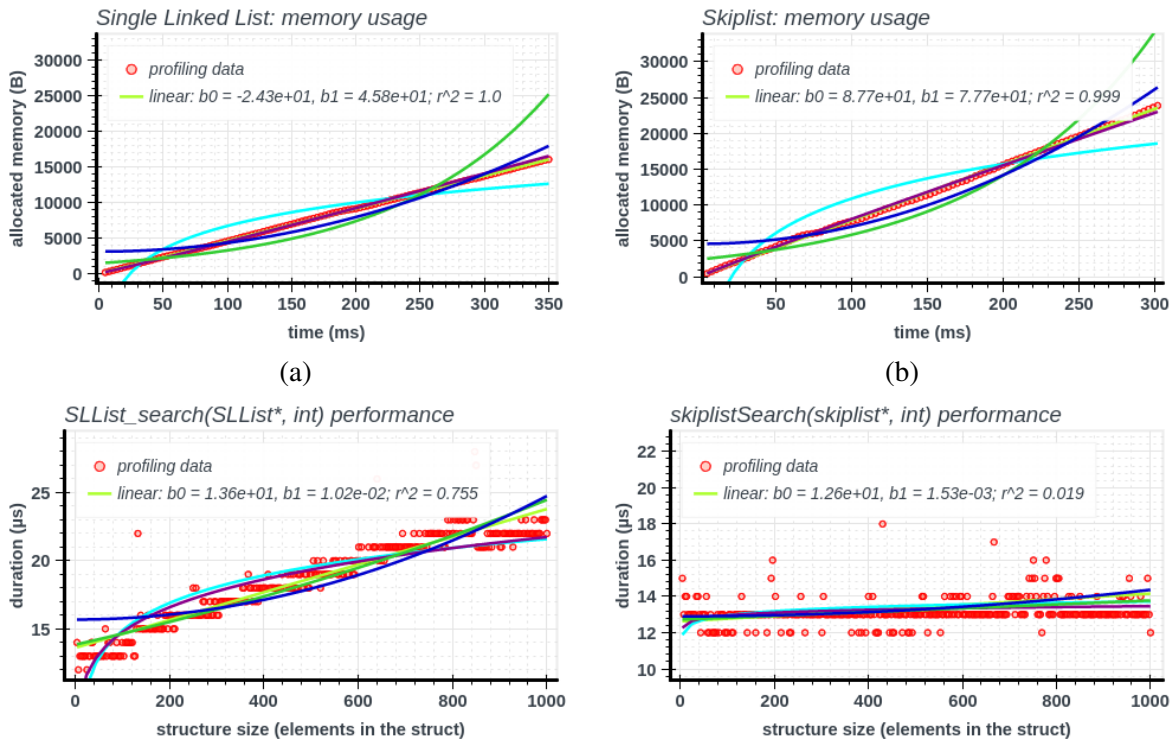
- Analyzovat fragmentaci alokovaných paměťových úseků,
- demonstrovat uložení jednotlivých dat v datové struktuře různých datových typů,
- nebo vidět funkčnost standardních alokačních funkcí, popřípadě při úpravě kolektoru otestovat funkčnost vlastních alokačních funkcí, což může být užitečné například při vývoji vestavěných systémů.

Konkrétně obrázek 6 znázorňuje rozdíl uložení dat v rozdílných ADT[5] jazyka C++. Purpurově jsou znázorněna data uložená v datovém typu `std::list`[6], modře v `std::vector`[7]. Mapa dokazuje, že druhý zmiňovaný typ ukládá data v paměti kontinuálně a je



Obrázek 6. Uložení dat v rozdílných ADT.

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



Obrázek 7. Jak seznam (a), tak *skiplist* (b) mají pro 1 000 vložených prvků lineární paměťové nároky, avšak *skiplist* spotřeboval o polovinu paměti více. Vyhledání prvku v nejhorším případě má u seznamu ale lineární časovou složitost (c), oproti téměř konstantní složitosti v případě *skiplistu* (d).

tudíž jeho použití často o mnoho výhodnější, neboť paměťové operace s daty jsou méně časově náročné.

Obrázek 7 demonstruje spolupráci obou profilovacích knihoven na srovnání paměťových a časových nároků jednosměrně vázaného seznamu se *skiplistem*.

5. Perun: Performance Control System

Hlavním autorem nástroje je T. Fiedor [3] a oba autoři práce spolupracovali na návrhu jeho architektury — zejména na modulech kolektorů, vizualizací a jejich rozhraní v podobě unifikovaného profilovacího formátu.

Perun je odlehčený otevřený nástroj zaměřený na správu výkonu programu. Je založený na principech systémů pro správu verzí, který sleduje změny kódu a udržuje obrazy jednotlivých verzí projektů. Klasické verzovací systémy lze pochopitelně rovněž použít pro uložení výkonnostních profilů programu pro každou dílčí verzi projektu v čase — uživatel je pak ovšem nucen provádět profilaci a anotaci manuálně, jinak ztrácí přesný přehled o historii výkonu. Perun řeší tato omezení přidáním:

1. **Kontextu** — každý profil je svázán s konkrétní verzí programu, což do daného profilu přidává chybějící kontext — co bylo v kódu změněno, kdy to bylo změněno a kdo změnu provedl.
2. **Automatizace** — Perun se dokáže zachytit nad verzovací systém aby zajistil, že pro každou novou verzi projektu bude vytvořen výkonnostní 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.
3. **Pozice v čase** — Perun uchovává historii projektu a umožňuje pohled na výkonnostní změny v průběhu vývoje a času.

Životní cyklus profilu. Profil je po výstupu z kolektorů komprimován, uložen do adresářové struktury Perunu a přiřazen k aktuálnímu kontextu projektu (např. u verzovacího systému *git* je kontext projektu vyjádřen konkrétním commitem).

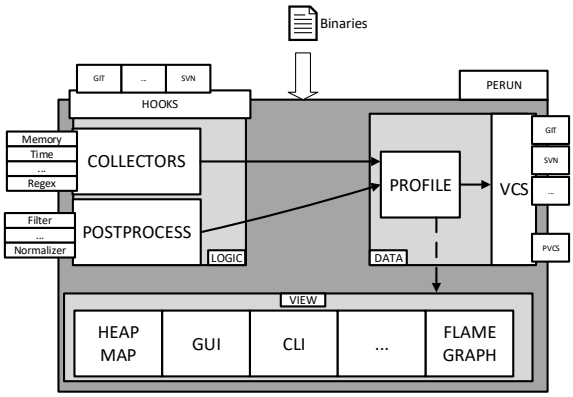
Profily jsou generovány sadou podporovaných kolektorů (jako je například kolektor dat paměťových operací 2.1, kolektor dat složitosti 2.2, nebo kolektor řízený regulárními výrazy) a mohou být následně rozšířeny a transformovány sérií postprocesorových sad (jako jsou například filtry, normalizéry, atd.).

Tyto profily poté mohou být vizualizovány celou řadou vizualizačních metod jako je Flame graf, mapa haldy, korelační diagramy, aj.

Architektura. Perun je rozdělen do tří hlavních částí: pohledu, dat a logiky. Datová část obsahuje základní jednotku Perunu — profil a obálku verzovacích systémů. Logika má na starost automatizaci, manipulaci a vytvá-

ření profilů. Skládá se ze sady kolektorů pro generování profilů, sérií postprocesorů pro jejich transformaci, přístupů k verzovacím systémům a jiným externím jednotkám pro dosažení automatizace. Pohled je nezávislý modul, který obsahuje vizualizační metody a obálky pro grafické a textové rozhraní.

Perun je aktuálně ve vývoji a zatím nebyla vydána stabilní verze. Nástroj je dostupný online [3].



Obrázek 8. Architektura nástroje Perun [3].

6. Závěr

Článek představuje společnou práci na generování profilů jednotného formátu pro shromážděné časové a prostorové data. Kromě toho jsme realizovali vizualizační metody pro odlišný pohled na profilovací data — mapu haldy pro lepší obrázek o užívané paměti, korelační diagramy zdrojů odpovídajících struktur a regresní analýzy naměřených dat.

Ukázali jsme, jak snadno lze použít vygenerované profily a vizualizace dat pro analýzu výkonu struktur a algoritmů — jako je například vizualizace spotřeby paměti *vektorů* a *seznamů* nebo výpočetní náročnost algoritmu *Quicksort* nad opačně seřazeným polem s vloženým šumem v podobě náhodných prvků. Navíc jsme ukázali možnost propojení výsledků obou našich prací provedením regresní analýzy spotřebované paměti analyzovaným programem.

Profilovací informace mohou značně usnadnit objevení problémů s výkonem programu. Získání základních profilovacích informací není náročné, avšak samotná data nejsou nijak užitečná a musí se vhodně interpretovat. Návrh je důležitou stránkou užitečné interpretace a měl by přímo poukazovat na možné problémy.

Ostatní vývojáři mohou díky našemu návrhu profilovacích knihoven a popisu práce s nimi vytvořit nové nebo upravit stávající části podle vlastních potřeb a preferencí. Například profilovaní jiných problematických částí programů nebo profilovaní programů napsaných v jiných jazycích. Další pokračování práce zahrnuje dopracování návrhů, odladění chyb a další integraci do nástroje Perun popsaného v kapitole 5.

Poděkování

Rádi bychom poděkovali vedoucímu našich prací, Ing. Tomáši Fiedorovi za cenné rady a připomínky při tvorbě nástrojů a psaní technických zpráv. Rovněž bychom chtěli poděkovat Mgr. Bc. Haně Pluháčkové za odbornou konzultaci k problematice regresní analýzy. Práce je realizována za podpory firmy RedHat v rámci společného výzkumu s VUT FIT v Red Hat Labu.

Literatura

[1] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. *Understanding and detecting real-world performance bugs. Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 77 – 88, 2012.

[2] *Valgrind's Tool Suite*. http://valgrind.org/info/tools.html. [Online; navštíveno 24.3.2017].

[3] Tomáš Fiedor. *Perun: Lightweight Performance Control System*. https://github.com/tfiedor/perun. [Online; navštíveno 26.3.2017].

[4] *The libunwind project*. http://www.nongnu.org/libunwind/. [Online; navštíveno 24.3.2017].

[5] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, 1990.

[6] *Options for Code Generation Conventions*. https://gcc.gnu.org/onlinedocs/gcc-4.6.0/gcc/Code-Gen-Options.html. [Online; navštíveno 24.3.2017].

[7] Robert Endre Tarjan. *Amortized Computational Complexity*. *SIAM Journal on Algebraic Discrete Methods*, 6(2):306–318, 1985.

[8] Jay L. Devore. *Probability and Statistics for Engineering and the Sciences*. Cengage Learning, 8 edition, 2011.

[9] Stephen Few. *Information Dashboard Design*. O'Reilly, 2006.