

Západočeská univerzita v Plzni
Fakulta aplikovaných věd
Katedra informatiky a výpočetní techniky

Bakalářská práce

Vizualizace dat profilovacích nástrojů

Místo této strany bude
zadání práce.

Prohlášení

Prohlašuji, že jsem bakalářskou práci vypracoval samostatně a výhradně s použitím citovaných pramenů.

V Plzni dne 16. dubna 2016

Martin Úbl

Abstract

The main goal of this paper is to analyze available profiling tools, their output formats, visualization methods of their collected data and to design and implement portable modular tool for such data visualization.

In first chapters, the problem of profiling and collecting performance data is investigated. Next part focuses on analysis of commonly used profiling tools and their output formats, common visualization methods and already available visualization tools.

The second part contains design of modular tool, which would be able to load, analyze and visualize profiling data independently of what profiler was used and which operating system the user runs. The last part then focuses on tool realization, contains implementation details and user documentation. Outputs are validated against outputs from already existing tools.

Abstrakt

Hlavním cílem této práce je analýza dostupných profilovacích nástrojů, jejich výstupních formátů, způsobů vizualizace jimi nasbíraných dat, návrh přenositelného modulárního nástroje pro jejich vizualizaci a jeho realizace.

V prvních kapitolách je rozebrána problematika profilingu a způsobů sběru dat. Následuje analýza používaných profilovacích nástrojů a formátu jejich výstupu, dále způsobů vizualizace profilovacích dat a nástrojů pro vizualizaci, které jsou již dostupné.

V druhé části je obsažen návrh modulárního nástroje, který bude schopen nezávisle na operačním systému a použitém profileru načíst odpovídajícím modulem profilová data, vnitřně je analyzovat a pomocí výstupního modulu poskytnout jejich vizualizaci. Poslední část se pak zabývá realizací nástroje, obsahuje programátorskou a uživatelskou dokumentaci výsledného programu. Výstupy jsou dále ověřeny oproti výstupům již existujících nástrojů.

Poděkování

Rád bych touto cestou poděkoval Ing. Jindřichu Skupovi za odborné vedení a cenné rady v průběhu této práce. Dále patří poděkování panu Jiřímu Jabůrkovi za pomoc při získávání praktických zkušeností v oblasti zkoumané problematiky.

Obsah

1	Úvod	9
2	Profiling	10
2.1	Výkon	10
2.2	Bottleneck	10
2.2.1	Typické řešení	11
2.3	Kritické optimalizace	12
2.4	Metody sběru dat	15
2.4.1	Vzorkování	15
2.4.2	Instrumentace	16
2.4.3	Interpretace	16
2.4.4	Událostní profiling	16
3	Analýza dostupných nástrojů	17
3.1	perf	17
3.2	gprof	18
3.3	OProfile	21
3.4	Callgrind a Cachegrind	22
3.5	DTrace	24
3.6	Visual Studio Performance Profiling	25
3.7	Very Sleepy	26
3.8	RotateRight Zoom	27
3.9	Shrnutí	28
4	Analýza vizualizačních technik	30
4.1	Flat view	30
4.2	Hierarchical view	31
4.3	Graph view	31
4.4	Object-method view	33
4.5	Flame graph	33
4.5.1	Detailed graph	34
4.6	Heat maps	35
4.7	Shrnutí	37

5	Dostupné nástroje pro vizualizaci	38
5.1	Visual Studio Performance Profiling	38
5.2	KCachegrind	38
5.3	KProf	39
5.4	GProf2dot	39
5.5	RotateRight Zoom	39
5.6	Shrnutí	40
6	Návrh nástroje	41
6.1	Základ systému	41
6.1.1	Jádro	41
6.1.2	Analyzér	42
6.2	Vstupní moduly	43
6.3	Výstupní moduly	43
6.4	Technologie	43
6.5	Licence	44
6.6	Shrnutí	44
7	Implementace nástroje	45
7.1	Základ systému	45
7.1.1	Jádro	45
7.1.2	Analyzér	46
7.2	Vstupní moduly	49
7.2.1	Vstupní modul - gprof	50
7.2.2	Vstupní modul - perf	52
7.3	Výstupní moduly	55
7.3.1	Výstupní modul - webová stránka	56
8	Uživatelská dokumentace	64
8.1	Získání	64
8.2	Instalace	65
8.3	Spuštění	66
9	Ověření výsledků	68
9.1	Jednoduchá aplikace a gprof	68
9.2	Rozsáhlá aplikace a perf	70
10	Závěr	80
	Literatura	81
	Seznam zkratk	82

1 Úvod

Výkon programů, tedy schopnost zpracovat co největší množství dat nebo provést co nejvíce výpočtů za jednotku času, byl vždy velmi důležitým kritériem použitelnosti kteréhokoliv software. Při psaní programového kódu je tedy nutné dbát na vhodnou volbu algoritmu, na způsob zápisu a v krajních případech i na to, jak kompilátor přeloží zapsaný programový kód do binární podoby, případně jak interpret uvnitř odvodí posloupnost operací nutnou k jeho vykonání. Takových informací je ale příliš na to, aby bylo v lidských silách bez pomoci strojové analýzy vyvinout optimální kód - tedy takový, který má prokazatelně největší výkon bez ztráty přesnosti a správnosti výsledku. Proto existují nástroje zvané „profilery“, které analyzují běh programu, výkon jednotlivých částí kódu, případně staticky i kód samotný, aby mohly posléze vygenerovat zprávu o tom, kde program tráví nejvíce času a kde je tedy vhodné provést optimalizaci. Stejně jako profilery nasbíraná data je důležitý způsob jejich prezentování vývojáři, aby mohl optimalizovat tu část programu, která je skutečně zodpovědná za pomalý běh.

Cílem této práce je analyzovat dostupné profilery, jejich výstupní formáty, způsoby, jakým prezentují nasbíraná data uživateli a existující způsoby vizualizace těchto dat obecně, a na základě této analýzy navrhnout a implementovat modulární nástroj, který dovede vizualizovat výstupy různých profilerů v jednotné formě. Modularita bude spočívat v oddělené implementaci jádra aplikace, modulů pro načítání dat z profilerů a modulů výstupních. Dalším požadavkem je přenositelnost mezi běžnými platformami.

Navržený nástroj tedy nebude obstarávat sběr dat, pouze jejich zpracování a vizualizaci standardními způsoby.

2 Profiling

Profiling, někdy nazývaný *výkonnostní analýza*, je způsob vyhledávání míst v programu, která výrazně snižují výkon celé aplikace, popř. celého systému. Takové místo se nazývá *bottleneck* a jeho identifikace nemusí být snadná, stejně jako pozdější řešení. K identifikaci výkonnostních problémů slouží nástroje zvané *profilery*, které se starají o sběr informací z běhu programu a někdy i ze statické analýzy kódu. Tato data pak předají vizualizačnímu nástroji, ať už ve formě souboru nebo datového proudu, a ten je poskytne v lidmi interpretovatelné formě vývojáři.

2.1 Výkon

Bavíme-li se o analýze výkonu programů, je nutné stanovit, co vlastně sledujeme, a co očekáváme, že nám profilery poskytnou. Hlavní veličinou, kterou budeme sledovat, je výkon. V tomto kontextu lze použít fyzikální definici, tedy že výkon je práce vykonaná za jednotku času. U času se můžeme zabírat pouze jednotkou, tedy za jaký časový úsek budeme práci sledovat - to je ale velmi specifické pro každý případ, a proto se tím nebudeme v obecné rovině zabírat. Zbývá pouze definovat práci. Tu můžeme v této terminologii popsat například jako počet vykonaných instrukcí, což je exaktní měřítko z hlediska hardware, ale nemusí mít dostatečnou vypovídací hodnotu o skutečné efektivní práci. Proto práci definujeme spíše jako počet vykonaných operací, kde operaci můžeme abstrahovat například na funkční volání. Z toho vyplývá i to, že je nutné důsledně členit programový kód do funkcí (metod, objektů), aby bylo možné tuto metriku vůbec použít.

2.2 Bottleneck

Pojmem *bottleneck* se obecně označuje kterýkoliv element (modul, komponenta, funkce), který způsobuje zpomalení celku (aplikace, stroje). Bottleneck můžeme najít na různých místech, a to nejen v programovém kódu - může například jít o zpoždění při komunikaci přes síť, při zápisu nebo čtení z disku. To jsou takzvané hardwarové bottlenecky, a mají pouze nepřímý

vliv na výkon programu samotného. Lze je řešit buď výměnou součástky, nebo jinou fyzickou změnou (výměna kabelu, změna topologie sítě, atd.).

Hlavním bodem zájmu bude ale bottleneck softwarový, tedy ten, co lze optimalizovat pouze změnami v programovém kódu. Nutno dodat, že optimalizace softwarového bottlenecku nemusí nutně znamenat jeho odstranění. Pokud jde například o zápis velkého objemu dat na pevný disk, je samozřejmé, že tato činnost bude trvat delší dobu, a bez optimalizace na úrovni hardware (např. výměna rotačního disku za SSD) se nelze zbavit ani zpoždění při provádění programu. Ve spoustě případů, pokud dokončení zápisu nemusí blokovat běh programu, se ale lze vyhnout čekání, a to například vhodnou paralelizací.

Doba provádění programu bude vždy záviset na hardware. Důležité je proto oddělit problém softwarový od problému hardwarového. Výkonnostní optimalizaci softwarového rázu lze definovat jako takový zásah do programového kódu, který zkrátí dobu provádění daného úseku kódu beze změny jeho výstupu. Použitím abstrakce z předchozí sekce jde o snížení počtu operací, případně o výměnu za méně náročné operace.

Dále se budeme zabývat pouze bottlenecky softwarovými.

2.2.1 Typické řešení

Velmi často vzniká chyba v použití nevhodného algoritmu, a to buď obecně, nebo v závislosti na situaci. Školním případem by byly řadící algoritmy - primitivním způsobem, jak řadit pole čísel, je například *bubble sort*. Jsou ale obecně známy mnohem efektivnější algoritmy řazení pro jakoukoliv množinu čísel. Proto je volba *bubble sortu* obecně nevhodná z hlediska výkonu. Nevhodná volba v závislosti na situaci často spočívá v opomenutí nějakého faktu, který se situací souvisí - například nějaká definovatelná pravidelnost ve vstupních datech. V řadících algoritmech by byla dobrým příkladem seřazená posloupnost na vstupu. Pokud bychom mohli ve většině případů předpokládat seřazenou nebo téměř seřazenou posloupnost, pak není vhodné implementovat algoritmus, který přidává velkou režii v podobě přeskupování již seřazené podposloupnosti (např. *heap sort*), ale je lepší využít takový, který lépe pracuje s téměř seřazenou posloupností (např. *insertion sort* a jeho varianty).

Další typickou příčinou neoptimálního běhu je nevhodné využití datových

úložišť' obecně. Zpravidla je vhodné mít jako prostředníka mezi pomalým a okamžitým úložištěm nějakou vyrovnávací paměť, takzvanou *cache*. Využitím takovéto paměti odpadá nutnost žádat pomalé úložiště pro každý blok, který nás zajímá. Místo toho lze do cache nahrát mnohem větší množství dat, která jsou například často využívána, nebo v blízkém okolí aktuálně žádaných dat. Tím je možné minimalizovat přístupovou dobu pro případ opakovaného nebo sekvenčního čtení.

Speciálním případem nepřímě viditelného datového úložiště je cache procesoru, jejíž využití je přímo nutností pro rozumný běh kteréhokoli programu. Čtení a zápis zde totiž probíhá v několikanásobně kratším čase, než ve standardní operační paměti. To, co se v cache procesoru uchovává je možné v rámci programu ovlivnit například tak, že je respektován princip *lokality* - pokud jsou data uchována v paměti za sebou, a je k nim i tak přistupováno, nevzniká v CPU cache tolik cache-miss¹ a provádění programu je podstatně rychlejší.

2.3 Kritické optimalizace

V krajních případech, kdy máme jistotu, že nelze použít lepší algoritmus a nelze lépe optimalizovat přístupy k datovým úložištím, ale stále potřebujeme zvednout výkon, se naskytá možnost přistoupit k tzv. *kritickým optimalizacím*. Ty spočívají hlavně ve snížení náročnosti na úrovni instrukcí. Velkou část optimalizací na úrovni instrukcí ale dělá sám kompilátor, jelikož se často jedná o typické případy.

Dobrým příkladem kompilátorem obtížně proveditelné optimalizace je minimalizace počtu skoků, které jsou vykonány. Procesory, které implementují pipelining², musí totiž při provedení skoku zahodit doposud předzpracované instrukce a začít se zpracováním dalších na novém místě programu. Jelikož ve veškerých moderních programovacích jazycích dochází ke skoku pouze v rámci podmínek a cyklů, budou právě tyto struktury hlavním bodem zájmu při optimalizaci počtu skoků. Co se podmínek týče, lze využít pravděpodobnostní přístup - pokud je pravděpodobné, že podmínka bude splněna ve většině případů, je vhodné zajistit, aby se při splnění podmínky neprováděl skok.

¹skutečnost, kdy nebyla nalezena položka v cache procesoru, takže musí být vyzvednuta z pomalejšího úložiště

²způsob paralelního zpracování instrukcí ve vzájemně se nepřekrývajících fázích

Příklad optimalizace skoku lze vidět v následujících úryvcích kódu. V kódu 2.1 je implementována funkce, která určitým způsobem transformuje vstupní parametr. Uprostřed funkce je transformace podmíněná, která testuje, zdali se nejedná o krajní případ - funkce `is_corner_case()` bude vracet ve většině případů hodnotu 0, tedy podmínka splněna nebude. V pseudoassembly kódu 2.2 lze vidět, že se při nesplnění podmínky provede skok na návěští pod blokem příkazů, co by se provedly při platnosti podmínky. Jelikož podmínka bude splněna ve velmi malém množství případů, bude se naopak ve velkém množství případů provádět skok.

Výpis kódu 2.1: Funkce v jazyce C znázorňující nutnost skoku při splnění podmínky

```
int testFunction(int value)
{
    value *= 2;

    if (is_corner_case(value))
        value += 10;

    value *= 2;

    return value;
}
```

Výpis kódu 2.2: Pseudoassembly verze kódu 2.1

```
testFunction(VALUE):
    mul    2, VALUE
    call   is_corner_case(VALUE)
    cmp    %eax, 0
    beq    .LBL1
    add    10, VALUE
.LBL1:
    mul    2, VALUE
    ret
```

Přepis kódu do pravděpodobnostně optimální podoby by obecně záležel na konkrétní situaci. V tomto případě lze provést invertování podmínky a návrat z funkčního volání o něco dříve při pozitivním scénáři. Jak lze vidět z kódu 2.3, funkce je o něco delší. Za cenu delšího kódu byl získán takový sled instrukcí, ve kterém je výrazně menší pravděpodobnost skoku.

Výpis kódu 2.3: Funkce v jazyce C znázorňující nutnost skoku při splnění podmínky

```
int testFunction(int value)
{
    value *= 2;

    if (!is_corner_case(value))
    {
        value += 2;
        return value;
    }

    value += 10;
    value *= 2;

    return value;
}
```

Výpis kódu 2.4: Pseudoassembly verze kódu 2.3

```
testFunction(VALUE):
    mul    2, VALUE
    call   is_corner_case(VALUE)
    cmp    %eax, 0
    bne    .LBL1
    add    2, VALUE
    ret
.LBL1:
    add    10, VALUE
    mul    2, VALUE
    ret
```

Pokud je možné pravdivost podmínky alespoň s určitou pravděpodobností předpovědět, je vhodné kód přeuspořádat tak, aby byly minimalizovány skokové instrukce, tedy nutnost resetovat pipelining. Tento případ je totiž jedním z mála, který kompilátor nedovede s jistotou optimalizovat - v momentě kompilace totiž nelze strojově určit, zdali je některý scénář splnění podmínky více pravděpodobný. Některé kompilátory kombinované s vývojovým prostředím obsahují možnost tzv. *profile-guided optimizations*, které právě s pravděpodobností skoků pracují, a výsledný kód na základě nasbíraných dat optimalizují. Pro potřeby této práce je ale nutné znát hlavně fakt, že existují profily, které četnosti scénářů splnění podmínky počítají a předávají je skrze vizualizační nástroj programátorovi.

Některé kompilátory, například *gcc*, obsahují tzv. *branch prediction built-in* funkci `__builtin_expect`, prostřednictvím které lze předat kompilátoru informaci o pravděpodobnosti splnění podmínky. Není tedy nutné kód přeskupovat ručně.

2.4 Metody sběru dat

Způsobů, jakými jsou získávána výkonnostní data, může být více. Mezi hlavními a reálně používanými jsou ale pouze čtyři, které budou popsány v následujících podkapitolách.

2.4.1 Vzorkování

Principem vzorkování se rozumí opakované snímkování stavu provádění programu. Každý snímek obsahuje data spojená s funkčním voláním a současným zanořením, hodnotu programového čítače a volitelně i další údaje, jako hodnotu ostatních registrů, informaci o vlákne, které kód vykonává, a další, spíše specifické údaje pro daný profiler.

Vzorky mohou být pořizovány různými způsoby. Starším přístupem je periodické vzorkování čistě na základě času, konkrétně pomocí hardwarového přerušení časovače (IRQ 0). To v první řadě dovolovalo použít vzorkovací mechanismus pro profiling. Nevýhodou byla ale poměrně obtížně definovatelná frekvence, jelikož bylo třeba zvolit přiměřenou granularitu vzorků tak, aby stále poskytovala dostatečně prokazatelná data.

Podstatné zlepšení přinesl koncept hardwarových výkonnostních čítačů (dále jen HPC), což jsou speciální registry procesoru, jejichž jediným účelem je uchovávat počet vybraných hardwarových událostí. Takovou událostí může být například provedení instrukce, výskyt cache-miss, nutnost pozastavit pipelining kvůli zamezení datovému hazardu (tzv. stalling), a další. Hardwarové výkonnostní čítače jsou ale vždy závislé na konkrétním modelu, popř. řadě procesorů, a ne vždy je k dispozici stejná sada. Problém rozdílnosti těchto sad čítačů řeší modul obecně označovaný jako *perf events* (konkrétně u jádra GNU/Linux označováno jako *Linux Kernel Performance Events Subsystem*) v jádře OS, který poskytuje jednotné rozhraní k používání HPC a přidává další události související například s jádrem OS[4].

Přístup s HPC používá hardwarové přerušení NMI. To je generováno při každém přetečení některého z čítačů, jehož hodnotu sledujeme[4]. NMI je specifické v tom, že je vyvoláváno i tehdy, když jsou přerušení momentálně zakázána. Taková situace nastává například přímo při zpracování některých tzv. blokujících přerušení. Je tedy možné provádět profiling i jádra samotného.

2.4.2 Instrumentace

Dalším přístupem je takzvaná *instrumentace*. Ta spočívá ve vložení speciálních profilovacích funkčních volání, která se starají o inkrementaci odpovídajících čítačů volaných a volajících funkcí, a zaznamenání aktuální pozice vykonávání programu na základně hodnoty programového čítače.

Tato volání mohou být buď integrována v čase kompilace přímo kompilátorem (pokud to podporuje), ručně v programovém kódu, nebo dokonce do již zkompilovaného binárního spustitelného souboru.

2.4.3 Interpretace

Podstatně rozdílným přístupem je pojetí zkompilovaného strojového kódu jako intermediate kód³, který je interpretován nad vlastním virtuálním strojem[6]. Tento přístup dovoluje obalit kteroukoliv instrukci jakýmkoliv vlastním kódem, ovšem za cenu výrazného zpomalení běhu.

2.4.4 Událostní profiling

Příbuzným přístupem instrumentaci je událostní profiling. Potenciál tohoto způsobu získávání dat byl využit až v oblasti interpretovaných jazyků, kdy je přímo ve virtuálním stroji, který námi psaný program interpretuje, přítomno sledování určitých událostí, jako je například funkční volání, alokace paměti pro objekt daného typu a další.

³meziformát určený pro zefektivnění interpretace, lze např. uvést *bytecode* používaný v rámci Java Virtual Machine, nebo *CIL* z prostředí .NET

3 Analýza dostupných nástrojů

Pro všechny běžné platformy, tedy Windows, Linux a MacOS, existuje poměrně rozsáhlý sortiment profilerů. V této kapitole budou stručně popsány ty, které se řadí mezi nejpoužívanější.

3.1 perf

Perf je systém pro měření výkonu na operačních systémech založených na jádře GNU/Linux verze 2.6 a vyšší. Využívá metodu vzorkování, a to na základě hardwarových výkonnostních čítačů.

Nutnou podmínkou je tedy podpora v CPU pro výkonnostní čítače, které nás zajímají. Tato podpora byla zaváděna již od modelů procesoru Intel Pentium[1] a stala se standardním prvkem pro všechny následující modely nejen firmy Intel. Všechny dnešní procesory architektury x86, x86-64 a dalších tedy tuto podmínku jistě splňují. Dále je nutné provozovat operační systém s implementovanou podporou HPC.

Sběr dat

Samotný profiling je prováděn pomocí zaznamenávání těchto hardwarově vyvolávaných událostí přes buffer v jádře, a to buď do souboru (`perf record`) nebo přímo na výstup konzole (např. `perf top`).

V případě použití `perf record` jsou zachycené události zapisovány do souboru s implicitním názvem `perf.data` (lze změnit parametrem). Jedná se o binární soubor obsahující všechny pořízené vzorky. Z takového souboru lze posléze extrahovat různé pohledy - `perf report` spustí textové rozhraní pro navigaci v rámci seznamu funkcí řazených podle četnosti výskytu zkoumané události, `perf annotate` sestaví disassembly, ke kterému v případě, že byly do binárního souboru zakompilovány debugovací symboly, připojí i napařovaný zdrojový kód, `perf diff` pro sestavení rozdílu mezi dvěma reporty,

a další.

Ve výstupním souboru je obsažena hlavička a tři sekce. První sekcí je sekce s atributy, obsahující metainformace o vzorcích, druhou sekce se seznamem sledovaných událostí, a třetí sekce se samotnými vzorky nasbíraných za běhu[3].

Výhody a nevýhody

Velkou výhodou nástroje *perf* je právě ve využívání hardwarové podpory pro zjišťování výkonu, a interfacing s jádrem, které řeší rozdílnost řešení a sady čítačů v CPU samotném. Zkoumaný proces pak není nijak výrazně zpomalen oproti jeho normálnímu běhu, jelikož není nijak měněn instrukční tok programu samotného. Další výhodou je bezesporu sortiment veličin, které je možné zkoumat. Kromě „běžného“ zkoumání tráveného času prováděním specifických úseků kódu lze pozorovat i možné příčiny toho, proč je naměřený čas tak dlouhý.

Nevýhoda může být skryta ve výše zmíněném vzorkování. Jelikož je pořízen vzorek v rámci diskretních časových úseků, může se stát, že nějaký výkyv může uniknout. Vzhledem k tomu, že takové výkyvy většinou trvají mnohem déle, než je perioda vzorkování, není pravděpodobné, že bychom nepozorovali důležitý úsek z hlediska výkonu. Problémy vzorkování v oblasti *perf_events* lze připodobnit k problémům vzorkování kdekoli jinde - například vzorkování průběhu matematické funkce s velkými a rychlými výkyvy v průběhu nebo vzorkování průběhu audio signálu při převodu do digitální formy. Další nevýhodou je nutnost mít CPU, který podporuje hardwarové výkonnostní čítače, ale ty jsou v dnešní době standardem ve všech moderních procesorech.

3.2 gprof

Gprof je nástroj rozdělený na dvě části - část v kompilátoru a část pro interpretaci dat. V podstatě zaujímá přesně opačné postavení oproti nástroji *perf*. Namísto neinvazivního pozorování procesu na úrovni jádra OS a snímkování událostí z již dostupných zdrojů (HPC) je použita metoda instrumentace. Ta je metodou invazivní, tedy přímo mění instrukční tok při vstupu do bloku instrukcí náležícího každé funkci. Z toho plyne značné zpomalení běhu pro-

gramu.

Jelikož *gprof* nevyužívá přímo žádnou hardwarovou podporu, jsou prerekvizity čistě softwarové. Pro možnost profilovat tímto nástrojem je nutné mít nainstalovaný kompilátor, který dovede vložit potřebnou posloupnost instrukcí do každé z funkcí. Příkladem takového kompilátoru je *gcc*. Na většině linuxových distribucí je zároveň třeba doinstalovat balík *binutils*, kde je obsažen samotný nástroj pro interpretaci výstupu generovaného vloženými instrukcemi.

Dále je nutné kompilátor instruovat, aby profilovací volání do programu zakompiloval, a to zpravidla pomocí nějakého přepínače. Nástroj *gcc* tato volání integruje při kompilaci s přepínačem `-pg`.

Sběr dat

Jak již bylo zmíněno, v čase kompilace je do instrukčních bloků funkcí zapravena část kódu, která se stará o zaznamenání volání funkce a o měření času stráveného uvnitř funkce v jednom volání. To v případě nástroje *gprof* zaručují dvě funkce - `mcount()`, která zaznamenává volanou a volající funkci, a `profil()`, což je systémové volání pro zjištění hodnoty programového čítače a jeho zaznamenání do tabulky v paměti[7]. Toto systémové volání ale nemusí být v jádře implementované, a proto se jeho absence dá do jisté míry substituovat pomocí signálů zasílaných procesu.

Funkce `mcount()` je zodpovědná za evidenci počtu volání každé funkce. Díky zaznamenání volané i volající funkce v podobě programových čítačů je posléze možné zobrazit celý strom volání, případně vymezit větev, která je z hlediska výkonu kritická.

Funkce `profil()` nemusí být v daném OS dostupná. V případě, že dostupná je, je typicky volána v určitém časovém intervalu. Každé volání zjistí hodnotu programového čítače a inkrementuje hodnotu počítadla na odpovídající adrese v paměti.

Pokud funkce `profil()` dostupná není, je využito služeb časovačů v jádře OS k zasílání signálů zkoumanému procesu, kam je dodatečně zakompilována i funkce, která tento signál obstarává. Výsledek zpracování tohoto signálu je v podstatě identický s výsledkem volání funkce `profil()`, jen v podstatně delším čase a s možným zpožděním kvůli režii přidané na generování a ob-

starávání signálu.

Po spuštění aplikace zkompilevané se zapravením výše uvedených funkcí, je generován soubor s implicitním názvem `gmon.out` (popř. `jmenoprogramu.out`), který obsahuje veškerá počítadla zaznamenaná po čas běhu. K jejich interpretaci je možné použít příkaz `gprof`.

Tento soubor je binární, a kromě hlavičky může obsahovat až tři typy záznamů. Prvním záznamem je histogramový, který je zaznamenán voláním `profil()`, druhý je takzvaný *call-graph record*, obsahující informaci o funkčním volání zaznamenaným pomocí `mcount()`. Třetí záznam, tzv. *basic-block record*, může nahradit záznam histogramový, a to v případě, že byl použit *line-by-line* mód, tedy speciální režim získávání dat, který spojuje vzorkovací volání ne s funkcemi, ale s jednotlivými řádky zdrojového kódu. Tato funkcionality ale již v novějších verzích nástroje *gprof* není obsažena[2].

Histogramový záznam se vztahuje na určitý rozsah adres, v rámci nichž je ještě výsledná uložená struktura dělená po krocích zahrnujících například rozsah 64 adres. Tento rozsah má společný čítač nazývaný *bin*. Nástroj pak zaznamenává pomocí funkce `profil()` na jakých adresách se provádění programu vyskytovalo a provede „zaokrouhlení“ na nejbližší *bin*, jemuž zvýší jeho čítač. Při analýze je poté provedeno rovnoměrné rozdělení času mezi sousední symboly tak, jak zasahují do příslušných *binů*.

Záznam o funkčním volání, tedy *call-graph record* je zaznamenáván pro každou unikátní dvojici adres z volající a volané funkce, v paměti je mu navyšován čítač funkcí `mcount()`, kolikrát bylo takové volání provedeno a do výsledného souboru je zapsán pouze počet, kolikrát bylo přesně takové volání provedeno.

Výhody a nevýhody

Podstatnou výhodou je prakticky žádná přímá závislost na hardware. Jedinou nutnou závislostí zůstává podpora v kompilátoru, který je pro daný systém dostupný. Dalším pozitivem je, že jsou zaznamenány veškerá volání funkcí (krom těch, které jsou kompilátorem inlinovány¹ v rámci optimalizací).

Nevýhod je ale podstatně více. První znatelnou nevýhodou je zpomalení

¹nahrazení volání funkce přímo blokem příkazů, který funkce vykonává

běhu programu, což může být fatální pro velké množství případů. Tím odpadá možnost použít *gprof* pro vysokozátěžové systémy, kdy se výkonnostní problém objeví až při dosažení určité meze zátěže. Další podstatnou nevýhodou je nutnost mít program zkompilovaný se zapravenými funkcemi, tedy nelze profiling nijak „vypnout“ bez provedení nové kompilace.

3.3 OProfile

Sada nástrojů OProfile je poměrně starým způsobem profilování na systémech založených na GNU/Linux, ale dodnes je udržována a adaptována na nové technologie. Před verzí jádra Linux 2.6 již bylo v rámci OProfile možné využívat podporu hardwarových čítačů, a to použitím specifického driveru, který je v podstatě velmi podobný současné implementaci v rámci Linux Kernel Performance Events subsystému (zmíněného v kapitole 2.4.1). Dále bylo nutné mít zavedený daemon, obstarávající vzorkování hodnot výkonnostních čítačů, který je opět podobný již zmíněnému systému *perf events*.

S implementací subsystému pro hardwarové čítače a integrací *perf events* do samotného jádra OS odpadla nutnost udržování vlastního vývoje jejich substitucí, a bylo možné se soustředit přímo na vývoj samotného profileru.

Nástroj OProfile je nutné stáhnout v podobě zdrojových souborů buď jako archiv, nebo pomocí verzovacího systému *git*. Stažený zdrojový kód je třeba přeložit, je tedy nutné mít nainstalovanou sadu kompilátorů *gcc* a pro potřeby přeložení ještě přítomné závislosti. Další požadavky se mohou lišit podle verze a distribuce operačního systému, jejich přítomnost je kontrolována standardním *configure* skriptem.

Sběr dat

Jedná se o vzorkovací profiler, podobně jako nástroj *perf*. V současné době oba tyto profily dokonce využívají stejný modul v jádře OS.

Výstupem je ale celý adresář *oprofile_data*, kde jsou uloženy profilovací vzorky (podadresář *samples*), logovací výstupy a další věci související s profilingovým sezením. Soubory se vzorky jsou binárního typu a obsahují pouze páry offset:počet, kde offset je programový čítač, a počet je množství vzorků,

které bylo na tomto offsetu zaznamenáno. Každý soubor se vzorky může obsahovat vzorky jiného druhu - v závislosti na veličině, kterou sledujeme. To je rozlišeno podle struktury podadresářů a názvu souboru samotného.

Výstup lze interpretovat buď příkazem **opreport** pro výstup standardní, nebo příkazem **opgprof**, jehož výstup je přizpůsoben, aby vypadal shodně s výstupem nástroje *gprof*.

Výhody a nevýhody

Většinu výhod má tento nástroj shodnou s výhodami nástroje *perf*. Oproti němu ale disponuje možností formátovat výstup do stejné podoby jako nástroj *gprof*. Co se podporovaných funkcí týče, dá se říci, že nástroje *OProfile* a *perf* jsou zhruba na stejné úrovni a liší se pouze drobnostmi.

Nevýhody jsou opět poměrně stejné, jen nástroj *OProfile* není obsažen mezi standardními balíčky většiny distribucí a musí se dodatečně překládat a instalovat ručně.

3.4 Callgrind a Cachegrind

Tyto dva nástroje jsou přítomny v jednom velkém balíku vývojářských laďících nástrojů *valgrind* pro systémy unixového typu. Pod jednou sekcí jsou uvedeny proto, že funkcionalita nástroje *cachegrind*, tedy profileru zaměřeným na efektivní využití CPU cache, je v této době ve velké míře již obsažena i v nástroji *callgrind*, který se primárně staral pouze o statistiku volání funkcí a generování stromů volání.

Prvním kritériem pro provozování těchto nástrojů je samozřejmě operační systém - tím může být kterýkoliv z podporovaných OS unixového typu, tedy veškeré distribuce GNU/Linux na většině používaných architektur, Solaris, Android a Darwin (od verze 10.9 i MacOS X)[8]. Dále je třeba mít nainstalovaný odpovídající balíček *valgrind*, který obsahuje všechny přidružené nástroje, mezi kterými se objevuje právě i *callgrind* a *cachegrind*.

Sběr dat

Narozdíl od profilerů z předchozích kapitol, *callgrind* a *cachegrind* (dále pouze souhrnně *callgrind*) používají trochu nezvyklý způsob získávání výkonnostních dat - interpretací. To dovoluje každou instrukci obalit takřka libovolným kódem, který se v případě nástroje *callgrind* stará o počítání funkčních volání, cache-miss nad simulovanou cache, počítání přístupů do paměti a dalších událostí. Skutečnost, že je kód nejprve interpretován, bohužel běh programu značně zpomaluje.

Výsledkem je soubor `callgrind.out.PID`, kde PID je nahrazeno identifikátorem zkoumaného procesu. Jedná se o soubor, jehož obsahem je textová reprezentace záznamů existujících funkcí a všech funkcí z nich volaných včetně časů (zde označených jako „cena“, jelikož vzhledem ke zpomalení generovaném interpretací nelze porovnávat samotné časy) a pozic, ze kterých byly volány. V hlavičce souboru se navíc nachází výčet všech zkoumaných veličin, které jsou v souboru obsaženy[9].

Výstup lze interpretovat použitím příkazu `callgrind_annotate`, který při připojení cesty ke zdrojovým kódům generuje výstup ve formě kódu, kde je ke každé řádce připsán počet výskytů zkoumané události.

Výhody a nevýhody

Výhodou z hlediska funkcionality je rozhodně způsob, jakým nástroj sbírá data. Je zaručeno, že pokud existuje nějaké slabé místo, a je nástrojem správně interpretováno, tak nezmeškáme jeho průchod, jako tomu mohlo být u nástrojů provádějících vzorkování. Dále je velkou výhodou podpora na poměrně široké škále operačních systémů.

Velkou nevýhodou je ale zpomalení, které je vlivem interpretování kódu generováno. Původně strojový kód totiž nikdy není spuštěn přímo, ale přes další vrstvu, která umožňuje sběr dat. Další nevýhodou je, že je nutné interně určitý sortiment hardwarové funkcionality simulovat, aby došlo k detekci událostí v těchto místech. Typicky jde například o výskyt cache-miss, kdy musí *callgrind* simulovat cache na úrovni programového kódu. Pokud bychom se mohli spoléhat na implementaci s identickým fungováním, pak by o nevýhodu nešlo. Existuje ale velké množství rozdílných architektur, a prakticky každá může danou funkcionalitu implementovat jinak. Proto se

musíme spokojit s tím, že je tato simulace sice velmi blízko očekávanému modelu, ale nejde o její identickou implementaci.

3.5 DTrace

DTrace je framework určený obecně pro trasování a sledování veškerých akcí, které samotné jádro nebo zkoumaný program vykonává. Je dostupný pro OS Solaris, MacOS X, FreeBSD a jim příbuzné systémy.

Sběr dat

Veškerý sběr dat je uskutečněn definicí filtrů a událostí v programovacím jazyce *D*, což je speciální jazyk vyvinutý pro použití v rámci frameworku *DTrace*. Jsme schopni sledovat události jako jsou syscalls, čtení nebo zápis na disk, případně využívat již zmíněných hardwarových výkonnostních čítačů. Pro potřeby profilingu můžeme dále například snímkovat zásobník specifického procesu na určité frekvenci příkazem

```
dtrace -n 'profile-99 /pid == 189 && arg1/ { [ustack()] =  
    count(); '
```

Výstupem je pouze textová reprezentace zaznamenaných hodnot na konzoli nebo do souboru.

Výhody a nevýhody

Výhodou je bezesporu to, že je *DTrace* velmi obecným a dynamickým nástrojem, který oplývá vlastním programovacím jazykem. Můžeme si tak snáze přizpůsobit průběh trasování specifickým potřebám bez nutnosti použití externích nástrojů. Také je možné sledovat širokou škálu událostí, systémových veličin a používat hardwarovou podporu sledování výkonu.

Nevýhodou je hlavně složitost. Bez znalosti používaného jazyka nelze využít jeho potenciál v plné míře. Spousty uživatelů *DTrace* se proto soustředí pouze na používání skriptů, které již byly v minulosti napsány a používány za specifickým účelem.

3.6 Visual Studio Performance Profiling

V rámci vývojového prostředí Microsoft Visual Studio je od verze 2010 přítomen i nástroj pro sledování výkonu - Visual Studio Performance Profiling (dále jen *VSPerf*). Principiálně nabízí možnost měřit výkon využitím hardwarových výkonnostních čítačů a snímkováním zásobníku (mód „CPU Sampling“, ekvivalentní k nástroji *perf*), dále pomocí zakompilovaného kódu a počítačů zkoumat určité veličiny zblízka (mód „Instrumentation“, principem ekvivalentní k nástroji *gprof*), pokud je aplikace psána pro .NET Framework, pak poskytuje možnost sledovat práci s pamětí, identifikovat místa s největší četností požadavků o alokaci a sledovat funkci garbage collectoru. Také je přítomen mód pro detekování synchronizačních problémů, které vyúsťují k podstatnému snížení výkonu (příliš dlouhá kritická sekce, apod.).

Pro používání těchto nástrojů je nutné mít nainstalován OS MS Windows ve verzi XP nebo novější. Dále je třeba stáhnout a nainstalovat Microsoft Visual Studio alespoň ve verzi 2010.

Sběr dat

Metody sběru dat byly popsány již v kapitolách 3.1 (CPU sampling) a 3.2 (Instrumentation). Metoda použitá ve sledování správy paměti v .NET aplikacích se principiálně blíží nástrojům sady *valgrind*, jen s tím rozdílem, že se v tomto případě jedná o interpretovaný intermediate kód i bez použití dalšího nástroje.

Výstupem je soubor se sesbíranými daty, určený k prohlížení v rámci nástroje Visual Studio. Ten dovede vizualizovat interaktivní strom volání, vyhodnotit tzv. „Hot Path“, tedy výkonnostně kritickou větev stromu volání, vyhodnotit funkce, které vykonáváním vlastního kódu (tedy bez volání ostatních funkcí) zabíraly největší procento času a další, spíše doplňkové funkce. Formát výstupního souboru má uzavřenou specifikaci. Možné je ale provést export zformátovaných dat do formátů CSV nebo XML, které obsahují potřebné údaje pro zrekonstruování pohledů. Tento export je ale ztrátový, a tak neumožňuje na základě obsažených dat vytvořit pohledy jiného charakteru.

Výhody a nevýhody

Mezi výhody patří zejména integrace do jednoho z nejpoužívanějších IDE na OS MS Windows. Tím, že je implementace obstarána přímo na úrovni tohoto prostředí, je zaručena maximální kompatibilita, a lze přímo využívat funkcí prostředí. Další výhodou je možnost využít různé přístupy ke sběru dat v závislosti na tom, co zkoumáme, a to beze změny způsobu vizualizace.

Nevýhodou je nemožnost použít profilovací nástroj bez nutnosti mít nainstalované IDE. Je sice možné spouštět profiling z příkazové řádky, ale stále je potřeba mít k dispozici zbytek vývojového prostředí. To i v minimální nutné instalaci zabírá několik stovek megabajtů až jednotek gigabajtů v závislosti na verzi a edici.

3.7 Very Sleepy

Very Sleepy je jednoduchý profiler pro OS Windows, který vzniknul jako klon staršího, dnes již nevyvíjeného profileru Sleepy. Jedná se o neinvazivní profiler, který podobně jako ostatní profily z této kategorie pouze sleduje v pravidelných intervalech zásobník zkoumaného procesu (popř. analogicky zásobníky všech vláken daného procesu) a na základě generovaného stromu volání tvoří statistiku.

Pro používání je nutné mít nainstalován OS Windows verze XP a vyšší. Poté stačí mít pouze stažený profiler *Very Sleepy*, jehož balíček obsahuje vše potřebné.

Sběr dat

Jak bylo uvedeno, jedná se o neinvazivní profiler, který ve velmi krátkých pravidelných intervalech vzorkuje zásobník, extrahuje údaje o funkčním volání a inkrementuje čítače u funkcí, které se v tomto vzorku nachází. Zároveň na základě programového čítače identifikuje i instrukci, u které také provede inkrementaci čítače. Metrikou je zde tedy „počet výskytů“, který by měl být snadno předveditelný na čas, který je v dané funkci tráven.

Výstupem je primárně pouze vizualizace v podobě přehledu volaných funkcí

a procenty času tráveného jejich prováděním. Celé profilovací sezení je také možné uložit do souboru binárního typu, případně exportovat do formátu CSV.

Výhody a nevýhody

Výhodou je jistě to, že jde o velmi malý nástroj. Není třeba žádných dodatečných knihoven, ani velkých balíků závislostí.

Oproti zkoumaným profilerům má ale značně více nevýhod. Jelikož jde opět o profiler, který provádí vzorkování, objevuje se zde znovu možnost, že nějakou skutečnost přehlédne. Také má implementován pouze základní pohled, jehož obsahem jsou pouze procenta tráveného času v jednotlivých funkcích. Mimo to obsahuje poměrně velké množství nedodělaných, nedoladěných funkcí, což má za příčinu jistou nestabilitu.

Pro profilování menších aplikací za účelem získat orientační přehled o možných slabých místech se ale hodit může.

3.8 RotateRight Zoom

Tato sada nástrojů je určena pro operační systémy založené na GNU/Linux a MacOS X. Profiler funguje na již popsaném principu vzorkování, obdobně jako třeba nástroje *perf* nebo *OProfile*. Má ale několik vybraných odlišností v doplňových funkcích.

Od ostatních výše uvedených nástrojů se liší hlavně tím, že nejde pouze o profiler, ale celou sadu nástrojů, obsahující například profilovací server, který dovoluje přes síť vzdáleně zpřístupnit rozhraní k profilování. Není tedy nutné mít pro pokročilou vizualizaci výsledků nainstalované grafické prostředí, což je výhodou zejména při profilování na vzdálených serverech, kde by bylo jinak grafické prostředí zbytečné. Také obsahuje statický analyzátor kódu, takže je schopen do jisté míry i bez profilování určit potenciální slabá místa.

3.9 Shrnutí

Analyzované nástroje implementují tři různé způsoby sběru výkonnostních dat (shrnutí v tabulce 3.1). Prvním je vzorkování, tedy neinvazivní snímkování zásobníku a registrů, které sice výrazně nezpomaluje běh zkoumaného programu, ale nemusí poskytovat stoprocentně přesná data. Druhým je instrumentace, čili zakompilování diagnostických volání přímo do programu, což zpomaluje běh podstatně více, ale poskytuje to často velmi přesný přehled o provádění každého funkčního volání - u těchto nástrojů musíme operace konkretizovat na volání funkcí (metod). Třetím způsobem je interpretování spustitelného souboru „virtuálním strojem“, kde můžeme zkoumat prakticky libovolnou veličinu, ale běh programu je zpomalen za produkčně únosnou mez.

Výrazně se tedy liší situace, kdy je který nástroj vhodné použít. Nemusí však jít pouze o rozdílné druhy software - můžeme zkoumat stejný software, jen z jiného úhlu pohledu. Z tohoto důvodu je třeba, aby bylo možné nasbírané informace vizualizovat ideálně v jednotné formě. To by mimo jiné dovolilo vývojáři porovnat výsledky nad normalizovaným pohledem, a tedy by mohlo vést ke spolehlivějšímu odhalení bottlenecku. V případě všech headless² systémů může být problémem i to, že pokud požadujeme pokročilejší vizualizační techniky, je nutné mít nainstalované grafické prostředí, což není na většině serverů obvyklé. Mimo to lze sice vygenerovat nějaké statické výstupy (např. ve formátu SVG), ale ty se často liší od různých nástrojů - tedy buď úplně chybí, nebo nemají jednotnou formu.

Dalším zřejmým problémem je dostupnost napříč různými operačními systémy (znázorněno v tabulce 3.2). Tento problém sice nelze efektivně vyřešit, ale je možné poskytnout takovou vizualizaci dat, která na operačním systému závislá nebude.

Těmito problémy se budou zabírat následující kapitoly - bude navržen přenositelný nástroj, který bude schopen načíst výstupní formáty profilerů, analyzovat je, a pomocí známých vizualizačních technik poskytnout ucelený pohled, jehož dostupnost nebude podmíněna platformou. Pro potřeby této práce bude nástroj analyzovat pouze výstupy nejpoužívanějších nástrojů pod GNU/Linux, tedy nástroje *perf* a *gprof*. Bude ale navržen tak, aby bylo možné kdykoliv rozšířit podporu i pro jiný formát pouze připojením modulu.

²systémy bez nainstalovaného grafického prostředí, často pouze se vzdáleným terminálovým přístupem

	vzorkování	instrumentace	interpretace
perf	✓	✓	✗
gprof	✗	✓	✗
OProfile	✓	✗	✗
Callgrind	✗	✗	✓
DTrace	✓	✗	✗
VSPerf	✓	✓	✗
Very Sleepy	✓	✗	✗
RotateRight Zoom	✓	✗	✗

Tabulka 3.1: Tabulka způsobu sběru dat

	Windows	GNU/Linux	MacOS
perf	✗	✓	✗
gprof	✗	✓	✓
OProfile	✗	✓	✗
Callgrind	✗	✓	✓
DTrace	✗	✗	✓
VSPerf	✓	✗	✗
Very Sleepy	✓	✗	✗
RotateRight Zoom	✗	✓	✓

Tabulka 3.2: Tabulka podpory profilovacích nástrojů na nejpoužívanějších platformách

4 Analýza vizualizačních technik

Shromážděná data profilovacími nástroji je nutné převést do formy, které bude vývojář rozumět, a bude na základě ní schopný provést odpovídající změny v kódu. Jelikož jsme použili abstrakci operace na funkční volání, budou zejména konkrétní funkce subjektem zkoumání.

Před analyzováním jednotlivých pohledů je třeba nadefinovat dva pojmy - *inkluzivní čas* a *exkluzivní čas*. Čas trávený ve funkci je totiž nutné rozlišit na část, která byla strávena pouze v rámci instrukčního toku dané funkce, a část, která byla strávena prováděním funkcí z této funkce volaných. Lze snadno vyvodit, že *exkluzivní čas* udává tu část, která byla trávena pouze prováděním zkoumané funkce (exkludovali jsme čas volaných funkcí), a *inkluzivní čas* udává celkový čas včetně času stráveném ve volaných funkcích.

4.1 Flat view

Flat view, jinak nazývaný i „function view“, je prostý seznam funkcí s četností jejich volání, inkluzivním a exkluzivním časem (popř. vzorky). Jedná se o nejjednodušší pohled, který lze vygenerovat, a často je dostačující pro detekci možných bottlenecků. Postrádá ale kontext. Například nelze vydedukovat poměr počtu volání a náročnosti funkce. Nelze tedy přímo zjistit, zdali je třeba optimalizovat počet volání, nebo tělo funkce.

Z tohoto pohledu tedy lze vydedukovat pouze základní problémy a hodí se pouze pro jednoduché programy bez většího množství funkcí. V takových případech lze často kontext vypustit, jelikož je zřejmé, odkud je která funkce volaná.

Function Name	Inclusive Samples	Exclusive Samples ▼	Inclusive Samples %	Exclusive Samples %
[ntdll.dll]	106	106	45,89	45,89
[MSVCR120.dll]	204	73	88,31	31,60
[kernel32.dll]	48	15	20,78	6,49
[KERNELBASE.dll]	10	10	4,33	4,33
[MSVCP120.dll]	99	9	42,86	3,90
rpn_evaluate_stack	119	5	51,52	2,16
recalculate_fitness	122	3	52,81	1,30
perform_mutation	2	2	0,87	0,87
rpn_apply_operator	40	2	17,32	0,87
rpn_pop_two_values	27	2	11,69	0,87
get_fittest	1	1	0,43	0,43
print_population	100	1	43,29	0,43
return_fittest	1	1	0,43	0,43
std::operator<<<std::char_t	64	1	27,71	0,43
_tmainCRTStartup	229	0	99,13	0,00
main	229	0	99,13	0,00
mainCRTStartup	2	0	0,87	0,00
perform_crossover	40	0	17,32	0,00
select_random_chromozom	37	0	16,02	0,00
std::endl<char,std::char_trai	2	0	0,87	0,00
std::vector<chromosome *,s	2	0	0,87	0,00
Unknown	2	0	0,87	0,00

Obrázek 4.1: Flat view generovaný nástrojem VSPerf

4.2 Hierarchical view

Hierarchický pohled (také *call tree*, *strom volání*) přidává kontext do předchozího případu. Konkrétně jde o dedukování, která funkce volala kterou, tedy je možné snáze identifikovat možný výkonnostní problém, pokud není přímo ve volané funkci, ale v kontextu okolo.

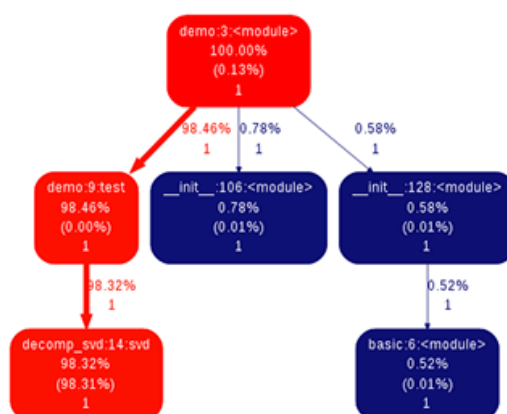
Tento pohled výrazně rozšiřuje množinu identifikovatelných problémů. Příkladem problému, který odhalí tento pohled navíc oproti *flat view*, je například ten, kdy máme poměrně náročnou funkci, která je volaná z mnoha míst v celém programu. Pokud je ale někde chybně volána častěji, než je nezbytně nutné, bude velké procento volání (popř. vzorků, tráveného času) zvyrazňovat místo problému.

4.3 Graph view

Grafový pohled je grafickým vylepšením pohledu hierarchického. Výhodou je možnost vidět všechny vazby naráz, bez duplikace jednotlivých funkcí v několika větvích stromu.

Function Name	Inclusive ...	Exclusive...	Inclusive Samples %	Exclusive Samples %	Module N...
GA_1_equation_solution.exe	231	0	100,00	0,00	
└─ tmainCRTStartup	229	0	99,13	0,00	GA_1_equatio
└─ main	229	0	99,13	0,00	GA_1_equatio
└─ print_population	100	1	43,29	0,43	GA_1_equatio
└─ recalculate_fitness	85	3	36,80	1,30	GA_1_equatio
└─ rpn_evaluate_stack	82	4	35,50	1,73	GA_1_equatio
└─ [MSVCR120.dll]	50	2	21,65	0,87	MSVCR120.dll
└─ rpn_apply_operator	28	1	12,12	0,43	GA_1_equatio
└─ rpn_pop_two_values	17	2	7,36	0,87	GA_1_equatio
└─ [MSVCR120.dll]	10	10	4,33	4,33	MSVCR120.dll
└─ perform_crossover	40	0	17,32	0,00	GA_1_equatio
└─ select_random_chromosome	37	0	16,02	0,00	GA_1_equatio
└─ recalculate_fitness	37	0	16,02	0,00	GA_1_equatio
└─ rpn_evaluate_stack	37	1	16,02	0,43	GA_1_equatio
└─ [MSVCR120.dll]	24	2	10,39	0,87	MSVCR120.dll
└─ rpn_apply_operator	12	1	5,19	0,43	GA_1_equatio
└─ std::vector<chromosome>,std::al	2	0	0,87	0,00	GA_1_equatio
└─ [MSVCR120.dll]	1	0	0,43	0,00	MSVCR120.dll
└─ perform_mutation	2	2	0,87	0,87	GA_1_equatio
└─ get_fittest	1	1	0,43	0,43	GA_1_equatio
└─ return_fittest	1	1	0,43	0,43	GA_1_equatio
└─ mainCRTStartup	2	0	0,87	0,00	GA_1_equatio

Obrázek 4.2: Hierarchical view generovaný nástrojem VSPerf

Obrázek 4.3: Graph view s vyznačenou *hot path*, Zdroj: <http://claudiovz.github.io/scipy-lecture-notes-ES/advanced/optimizing/index.html>

V hierarchickém pohledu totiž mohla nastat situace, kdy se v celém stromu jedna funkce vyskytuje na více místech, a to tehdy, pokud je volána z několika jiných funkcí. V grafovém pohledu jsou všechny duplikáty sloučeny do jednoho uzlu a z uzlů představujících volající funkce jsou do něj vedeny hrany.

4.4 Object-method view

V podstatě se jedná o *flat view*, ovšem v modifikaci pro programy psané v objektově orientovaném jazyce. Položky seznamu (zde metody), jsou sdruženy do skupin podle náležitosti třídě. Pokud je program správně dekomponován, lze pomocí tohoto pohledu identifikovat třídu, jejíž metody představují časově nejnáročnější operace.

Function/Method	Count	Total (s)	%	Self (s)	Total ms
CProfileInfo					
CProfileViewItem					
KAboutData					
KProfTopLevel					
KProfWidget					
KProfWidget(QWidget *, char const *)	1	0.02	0	0	45.94
applySettings(void)	2	0.02	0	0	7.18
fillFlatProfileList(void)	1	0.02	0	0	1.89
fillHierProfileList(void)	1	0.02	0	0	1.89
fillHierarchy(CProfileViewItem *, CProfileInfo *, QArray<CProfileInfo *> &, int &)	69	0.02	0	0	0
fillObjHierarchy(CProfileViewItem *, QString *)	19	0.02	0	0	0
fillObjProfileList(void)	1	0.02	0	0	1.96
loadSettings(void)	1	0.02	0	0	4.31
locateProfileEntry(QString const &)	309	0.02	50	0.01	32.36
openResultsFile(void)	255	0.02	0	0	0.05
parseProfile(QString &)	1	0.02	0	0	19216.1
prepareProfileView(KListView *, bool)	3	0.02	0	0	10.05
processCallGraphBlock(QVector<KProfWidget::SCallGraphEntry> const &)	71	0.02	0	0	140.85
staticMetaObject(void)	287	0.02	0	0	1.28
QArray<CProfileInfo *>					

Obrázek 4.4: Object-method view, Zdroj: <http://kprof.sourceforge.net/>

Výhoda tohoto pohledu může spočívat například v použití pro týmově vyvíjený produkt, kde každou komponentu měl na starosti jiný vývojář. Lze tedy snadno identifikovat osobu zodpovědnou za pomalý běh. Další vlastnosti tento pohled přebírá od *flat view*.

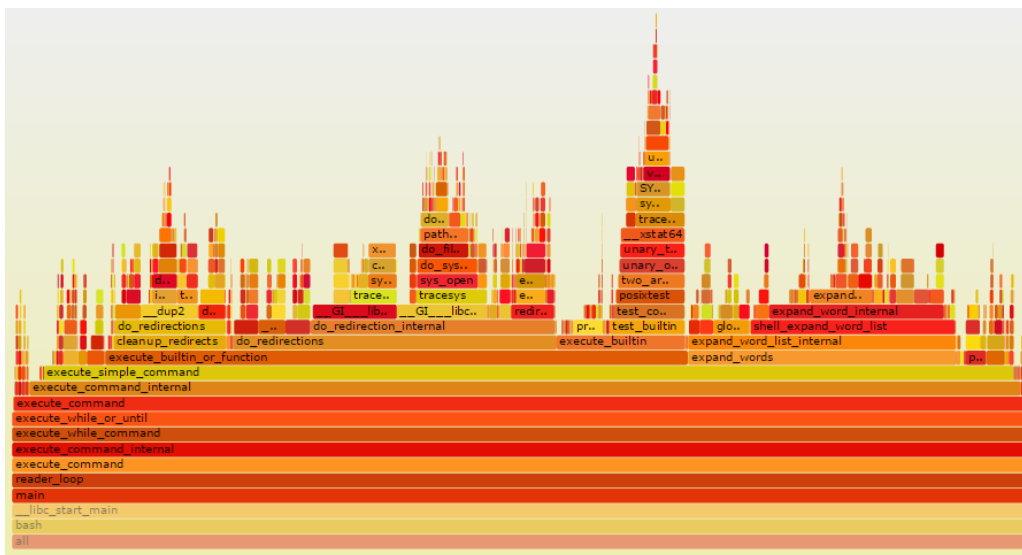
4.5 Flame graph

Tento pohled lze snadno generovat z pohledu hierarchického. Ve své podstatě jde pouze o expandovaný hierarchický pohled, který je uspořádán odspoda (kořen) nahoru (listy). Značnou výhodou je škálování jednotlivých položek podle procentuálních hodnot zkoumané veličiny - tráveného času (zde inkuzivního). Kořen zaujímá 100% šířky grafu, a veškeré volané funkce jsou

o úroveň výše, škálované relativně vůči celkovému exkluzivnímu času volající funkce a jsou vyneseny tak, aby zaujímaly přesně takovou procentuální šířku.

Tento druh vizualizace lze považovat za přínosný zejména z toho důvodu, že znázorňuje poměrný strávený čas vůči všem ostatním volaným funkcím. Tyto poměry nemusí být z hierarchického pohledu vidět. Také tento pohled významně redukuje množství dat a efektivně eliminuje nevýznamné položky - ty mají ve výsledném grafu minimální šířku.

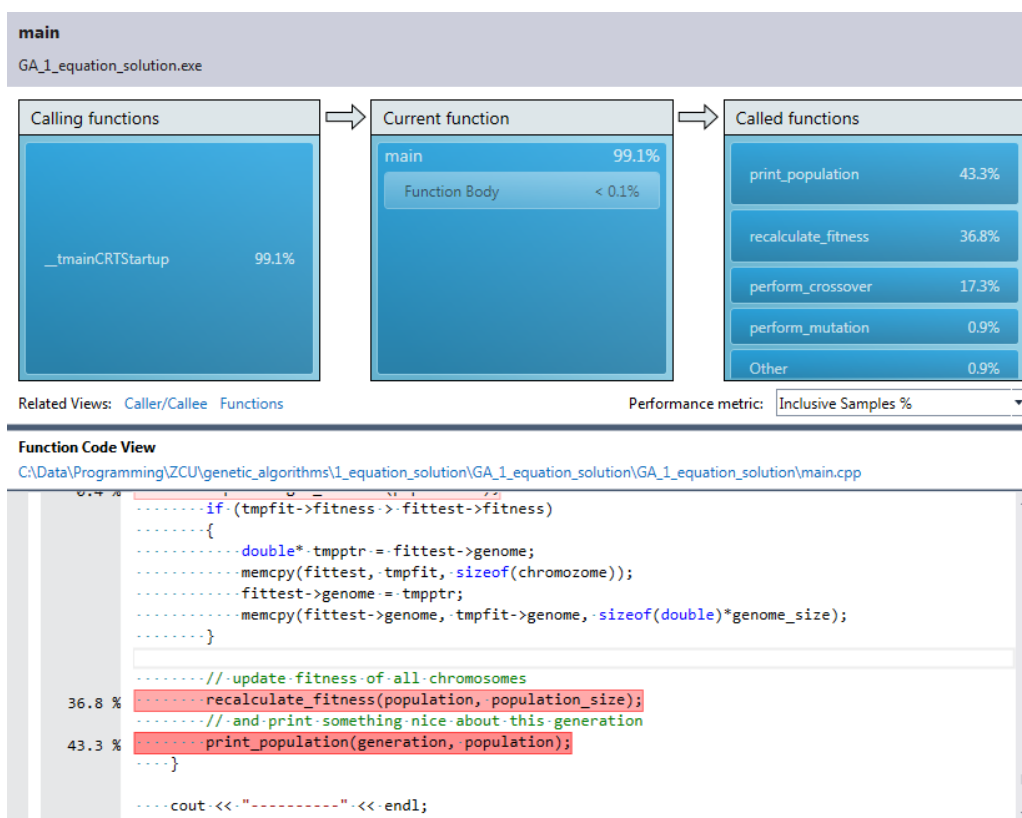
Takový graf ale vzhledem ke své obsáhlosti musí být buď dostatečně velký, nebo interaktivní. Interaktivitu si můžeme představit například možností rozkliknout funkci, která je pro nás zajímavá - poté se změní poměry šířek, vybraná funkce tvoří výchozí bod se 100% šířkou a všechny ostatní jí volané funkce škálují podle ní.



Obrázek 4.5: Flame graph, Zdroj: <http://www.brendangregg.com/FlameGraphs/cpuflamegraphs.html>

4.5.1 Detailed graph

Variací *flame graph* pohledu je tzv. detailní pohled. Ten pouze zužuje celkový pohled na výběr třech po sobě jdoucích úrovní. U takového pohledu je tedy nutná jistá interaktivita, která spočívá v možnosti procházet jednotlivé úrovně klikáním. Dále může být připojen výpis programového kódu se zvýrazněnými částmi, které jsou z hlediska výkonu významné.

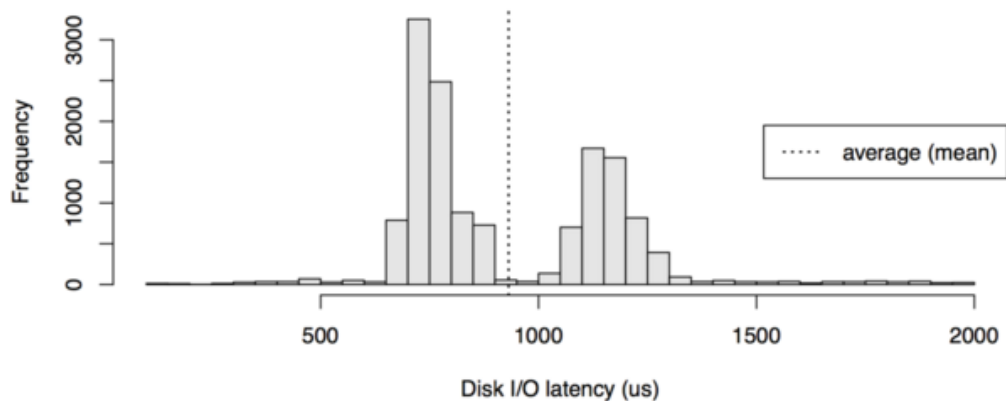


Obrázek 4.6: Detailed view generovaný nástrojem VSPerf

4.6 Heat maps

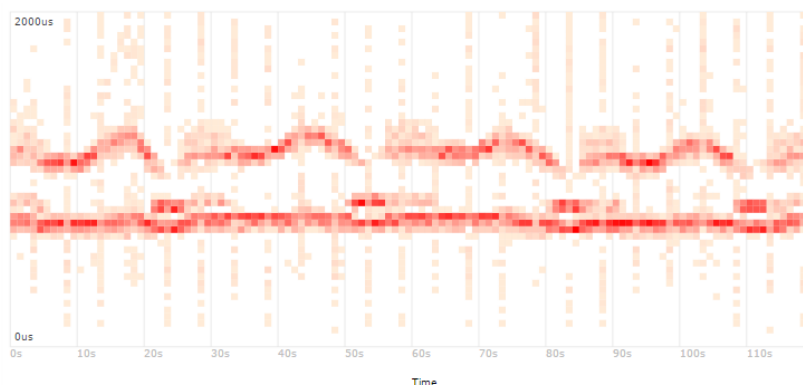
Heat mapy nejsou v současné době příliš rozšířenou technikou vizualizace profilových dat, ale je možné je pro určité body zájmu aplikovat. Oproti předchozím způsobům dovozuje zobrazovat do určité míry i trojrozměrná data. Vodorovná osa znázorňuje tok času, svislá osa zkoumané body zájmu, v případě profilu jde o funkce (metody). Osa času je často diskretizovaná na časové úseky větší, než je snímkovací perioda, jelikož je nutné za tento interval nasbírat takový počet výskytů zkoumaného jevu (programový čítač ukazuje na instrukci z instrukčního toku funkce), aby bylo možné rozlišit významnost.

Na obrázku 4.8 je vidět heat mapa odezvy pevného disku, což je momentálně jeden z hlavních případů použití této techniky. Vodorovná časová osa je rozdělena na diskrétní úseky po 1 vteřině a v rámci tohoto časového úseku bylo provedeno několik stovek čtení. Po uplynulé vteřině byl z časů odezvy vytvořen histogram četností (obrázek 4.7), výšky sloupců byly převedeny na



Obrázek 4.7: Histogram četností dob odezvy pevného disku při čtení a zápisu, Zdroj: <http://www.brendangregg.com/HeatMaps/latency.html>

intenzitu barvy, a do výsledného grafu byl takto obarvený sloupec vložen ve vertikálním směru.



Obrázek 4.8: Heat map, Zdroj: <http://www.brendangregg.com/HeatMaps/latency.html>

Výhodou takového grafu může být zejména viditelnost tendence. Na základě takto viditelných dat je možné posléze odhadovat, jak se bude program chovat při násobně větší zátěži. Určité aplikace totiž nedovolují provádět profilování při vysoké zátěži a to zejména kvůli zpomalení, které profilery způsobují. Příkladem mohou být veškeré služby, u kterých je žádoucí minimální odezva - kromě realtime systémů lze uvést například herní servery pro masově hrané hry, kde může vyšší odezva znamenat zhoršení herního zážitku.

4.7 Shrnutí

Způsobů vizualizace existuje poměrně velké množství, z velké části jde ale o modifikaci nebo kombinaci způsobů výše uvedených. Každý pohled zvyrazňuje trochu jinou informaci, přičemž záleží vždy na konkrétním případě užití, jak je interpretována. Proto je důležité mít k dispozici co možná nejširší škálu různých pohledů.

Také je vhodné řešit, jakým způsobem, tedy pomocí jakého „média“ bude výsledek předán vývojáři. Velká část nástrojů poskytuje výsledky v textové formě, čili snadno zobrazitelné kteroukoliv konzolí. Některé nástroje obsahují vlastní GUI, které je ovšem často závislé na platformě nebo grafickém prostředí. Dále malá skupina nástrojů poskytuje možnost exportovat výsledek do nějakého známého obrazového formátu - typicky PNG nebo SVG.

V rámci této práce bude realizován výstup do pohledů *flat view*, *hierarchical view*, *flame graph*, a budou prozkoumány možnosti *heat map* pohledu pro potřeby profilingu. Dále bude zahrnuta vizualizace do formy webové prezentace, tedy za použití technologií HTML, CSS a Javascript. Zároveň bude dodržena podobná modularita jako u vstupů, tedy bude možné kdykoliv přidat další „médiu“ jen přidáním modulu.

5 Dostupné nástroje pro vizualizaci

Vizualizačních nástrojů existuje také poměrně velké množství. Kritériem výběru by byl určitě podporovaný profiler, jehož data lze nástrojem zpracovávat, dále určitě závislost na platformě, podporované vizualizační techniky a případně další kritéria, jako je například interaktivita.

5.1 Visual Studio Performance Profiling

VSPerf je kombinovaným řešením, tedy obsahuje i vizualizaci nasbíraných dat. Jedná se o uzavřené řešení, podporován je pouze formát samotného profileru z této sady, z čehož opět vyplývá i závislost na prostředí MS Visual Studio a OS MS Windows. Data lze zobrazit pomocí *flat view*, *hierarchical view*, *detailed graph* a dalších, nepříliš významných technik.

Nástroj poskytuje pouze GUI se všemi pohledy, tedy nelze generovat žádná obrazová data, ani jiný přenositelný formát výstupu.

5.2 KCachegrind

Pro nástroje Callgrind a OProfile lze použít vizualizační prostředí KCachegrind. Jedná se o grafické prostředí, které ke svému běhu potřebuje OS GNU/Linux a grafické prostředí KDE, a je schopné vizualizovat data v pohledech *flat view*, *hierarchical view*, *graph view* a něčem, co připomíná *flame graph*.

Kromě GUI umožňuje KCachegrind navíc exportovat grafy do vektorových formátů.

5.3 KProf

Tento nástroj je velmi podobný nástroji předchozímu, jen dovede zpracovávat data nástroje *gprof* a jemu podobných. Také se jedná o grafické prostředí vyžadující OS GNU/Linux a grafické prostředí KDE. Implementuje pohledy *flat view*, *hierarchical view*, *graph view* a *object-method view*. Jedná se o podstatně jednodušší prostředí, než kterékoliv výše uvedné, ale pohledy nejsou ochuzeny o nic podstatného.

Tento nástroj poskytuje pouze GUI, opět tedy není přítomna žádná možnost, jak zpracovaná data přenést na jinou platformu.

5.4 GProf2dot

Poměrně cenným nástrojem v oblasti vizualizace profilovacích dat je *gprof2dot*. Původně dokázal zpracovat pouze výstupy nástroje *gprof*, postupně byla podpora rozšířena na širokou škálu profilerů zahrnujících *perf*, *OProfile*, *callgrind*, i například *Very Sleepy*. Jelikož se jedná o skript psaný v jazyce Python, jeho přenositelnost je podmíněná existencí interpretu pro danou platformu, a také několika málo závislostmi. Oficiálně jsou podporovány OS MS Windows a GNU/Linux. Výstupem je vždy pouze *graph view*, který lze poměrně ve velké míře přizpůsobovat.

Nástroj sice neposkytuje GUI, zato výstupem je obrázek v rastrovém nebo vektorovém formátu.

5.5 RotateRight Zoom

Podobně jako VSProf je i RotateRight Zoom kombinovaným uzavřeným řešením. Dokáže tedy zpracovávat pouze data nashromážděná vlastním profilerem. Oproti předchozím nástrojům má tu výhodu, že lze vizualizovat data na kterékoliv z nejpoužívanějších platforem. Ačkoliv není podporován profilování na OS MS Windows, existuje pro něj grafické rozhraní, které dovoluje zobrazovat data z profilingu probíhajícího na vzdáleném serveru. Kromě *flat view* a *hierarchical view* implementuje i pohled *timeline*, tedy časové osy.

Nástroj obsahuje GUI a zároveň je možné exportovat data do textového formátu.

5.6 Shrnutí

Dostupné nástroje často buď neposkytují přenositelnost výstupu, neobsahují potřebné pohledy, nebo nepodporují širší škálu profilerů.

Obsahem následující kapitoly bude návrh takového nástroje, který poskytne vysokou přenositelnost výstupu, velký výběr pohledů na profilovací data a zároveň zaručenou podporu pro velké množství profilerů, resp. jejich výstupních formátů. V rámci této práce bude realizováno jádro takového nástroje, a implementace pouze vybraných vstupních a výstupních modulů.

6 Návrh nástroje

Účelem nástroje bude načítat profilovací data z výstupních souborů známých profilerů, analyzovat je a následně vizualizovat pomocí vybraného výstupního modulu. Jedním z hlavních požadavků je i modularita nástroje, která bude spočívat ve členění na tři hlavní skupiny modulů, jak je znázorněno na obrázku 6.1. První částí je pevný základ, tedy samotná spustitelná část. Ta bude přejímat požadavky od uživatele, vybírat vstupní modul, analyzovat data a předávat je výstupnímu modulu. Druhou skupinou jsou vstupní moduly - pro každý formát bude existovat právě jeden vstupní modul. Tyto moduly se budou starat o načtení dat, transformaci do jednotné formy a předání zpět do jádra nástroje. Poslední skupinou jsou výstupní moduly, kterým jsou předána analyzovaná data opět v jednotné formě. Tyto moduly na základě vstupních parametrů od uživatele poskytnou požadovaný výstup.

Aplikace bude pouze konzolová. Tím je odbourána závislost na jakémkoliv grafickém prostředí.

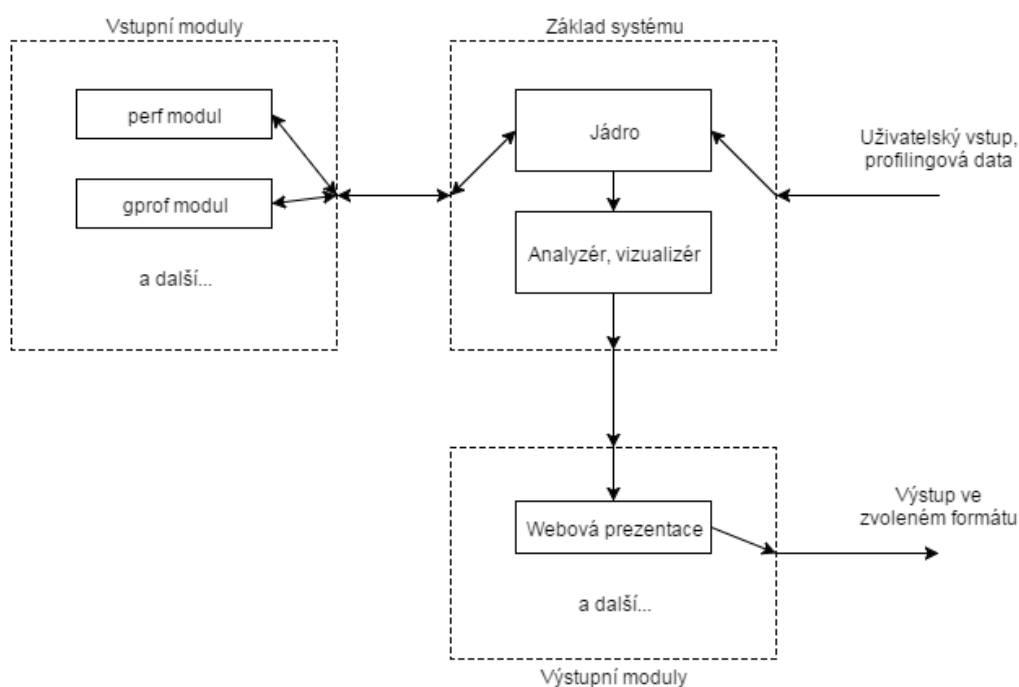
6.1 Základ systému

Tato skupina modulů bude představovat spustitelný program. Moduly základu budou tedy pevně svázány po čase kompilace, tedy nebudou fyzicky oddělitelné. To z důvodu, že je nutné mezi nimi zajistit maximální kompatibilitu, kterou lze do jisté míry zaručit při sestavení.

6.1.1 Jádro

Hlavním úkolem jádra bude obstarávat propojení všech modulů, které budou zapojeny do procesu zpracování vstupů, analýzy a následně formátování výstupu. Samotné jádro bude mít znalost pouze rozhraní se vstupními moduly a s analyzérem, kterému předá zpracovaná data ze vstupních modulů.

Jádro bude představovat i vstupní bod programu, tedy bude přejímat parametry příkazové řádky a na základě nich vybírat příslušné moduly a para-



Obrázek 6.1: Architektura nástroje, rozdělení do třech hlavních skupin modulů

metry vizualizace. Také se bude starat o logovací výstupy, které bude možné tisknout buď do konzole, nebo přesměřovat do souboru.

6.1.2 Analyzář

Tento modul bude na vstupu očekávat dostupná profilovací data v normalizovaném formátu, která bude analyzovat a tvořit z nich datové struktury pro jednotlivé pohledy. Bude tedy zodpovědný za první část sestavení pohledů. Druhá část bude implementována až v samotném výstupním modulu, který pouze přečte naformátované datové struktury a vytvoří z nich konkrétní pohled.

Analyzář nesmí předpokládat existenci veškerých dat, jelikož ne každý profiler poskytuje stejnou škálu informací. Od toho se bude odvíjet i množina pohledů, které je schopen předzpracovat a předat výstupnímu modulu k vizualizaci.

6.2 Vstupní moduly

Vstupní moduly jsou takové knihovny, které odděleně od sebe implementují jednotlivé formáty výstupních souborů profilerů. Jejich úkolem je na základě vstupního souboru (popř. adresáře) načíst data nashromážděná profilerem, normalizovat je do jednotného formátu a předat je jádru.

Veškeré vstupní moduly budou zrealizovány jako dynamické knihovny, na jejichž existenci nesmí spuštění programu záviset, vyjma případu, kdy je vyžadováno zpracování formátu, jehož zpracování tento modul implementuje.

6.3 Výstupní moduly

Tyto moduly budou mít na starost implementaci specifických způsobů vizualizace předzpracovaných dat, tedy konkrétní technologii. Budou přejímat normalizovaná data dostupných způsobů vizualizace profilových dat od analyzáru. Nesmí ale předpokládat existenci dat pro všechny pohledy, a to ze stejného důvodu, jako analyzáru - profiler nemusel potřebná data poskytnout.

Výstupní moduly budou stejně jako vstupní realizovány jako nezávislé dynamické knihovny.

6.4 Technologie

Nástroj bude realizován jako přenositelný. Jeho zdrojový kód bude psán v jazyce C++, a to proto, že je snadno zkompileovatelný na většině platform bez nutnosti instalovat nestandardní závislosti, a jeho kompilátor je zpravidla součástí výbavy každého serverového systému. Také se jedná o objektově orientovaný jazyk, takže bude možné psát čitelný a snadno rozšiřitelný kód, který bude strukturou odpovídat výše uvedenému návrhu.

Pro generování souborů sestavovacího nástroje bude použit nástroj CMake, jelikož je schopen pracovat právě s určitou modularitou a také je možné ho provozovat na všech běžně rozšířených platformách. Jeho výstup je pak možné přizpůsobit například nástroji *make* nebo lze generovat sadu projektů pro MS Visual Studio, a to bez nutnosti měnit cokoliv v konfiguračních či

zdrojových souborech. Také dovede detekovat absenci některých závislostí a vyhledávat přesné cesty k nástrojům a knihovnám na daném systému.

Pro správu zdrojových kódů bude použit systém správy verzí *git*.

Použití dalších technologií je závislé na implementačních detailech. V rámci této práce bude například použita základní škála frontendových webových technologií pro vizualizaci profilových dat formou webové prezentace. Jde o technologie HTML, CSS a Javascript. Webová prezentace je dobrým příkladem snadno interpretovatelné formy vizualizace, která je podporována napříč širokou škálou platforem od standardních počítačů až po mobilní zařízení, a to bez rozdílu operačního systému.

6.5 Licence

Nástroj bude vyvíjen jako svobodný software pod licencí GNU GPLv3 [5]. Důvodem je možnost otevřít vývoj široké veřejnosti. Takový nástroj ani není vhodné uzavírat co se vývoje týče, jelikož existuje příliš velké množství profilerů, vizualizačních technik a možných technologií pro vizualizaci. Zapojením široké vývojářské veřejnosti může dojít k podstatnému zlepšení kvality a obsáhlosti nástroje.

6.6 Shrnutí

Byl navržen modulární nástroj, který bude řešit problémy uvedené v předchozích kapitolách. Jedná se zejména o přenositelnost výstupu nástroje a sjednocení vizualizace pro různé profily. Dále jde o přenositelnost samotného programu, jeho modularitu, která úzce souvisí s rozšiřitelností a v neposlední řadě i o otevření zdrojových kódů veřejnosti.

Tento nástroj byl v rámci bakalářské práce implementován. Následující kapitoly se budou zabývat popisem implementace a dalšími detaily spojenými s vývojem.

7 Implementace nástroje

Cílem této kapitoly je popsat a vysvětlit důležité části v implementaci navrženého nástroje. Schéma systému bylo dodrženo a plně koresponduje s návrhem, proto bude tato kapitola členěna obdobně, jako předchozí. Zároveň budou popsány konkrétní moduly, které byly v rámci této práce vytvořeny, a v případě výstupního modulu i realizované vizualizační techniky.

Nástroj nese název Profiler-Independent Visual Output (zkr. PIVO).

7.1 Základ systému

Základem systému je jeden spustitelný program, který se vnitřně skládá ze dvou komponent - jádra a analyzáru. Moduly jsou k němu připojeny jako dynamické knihovny.

7.1.1 Jádro

Hlavní třída, která představuje tuto část systému je třída **Application**. Ta je jedináčkem, a je jí předáno řízení hned na začátku provádění programu. Nejprve je provedena inicializace v metodě **Init**, která zahrnuje načtení parametrů příkazové řádky, nastavení logování, prvotní validaci vstupních parametrů a následné načtení vstupního a výstupního modulu. Poté je metodou **Run** zahájeno samotné provádění hlavního toku v podobě volání metod vstupního modulu, analyzáru a modulu výstupního. Voláním metod vstupního modulu je postupně plněna struktura **NormalizedData**, která již obsahuje normalizovaná data pro vizualizaci. V těch může analyzáru provést určité změny, případně doplnit některá pole v rámci vlastních výpočtů. Tato struktura je poté předána výstupnímu modulu k vizualizaci.

Samotné jádro neobsahuje žádnou pokročilou logiku spojenou s konkrétním problémem. Stará se pouze o postupné kontaktování ostatních částí programu a modulů.

V rámci jádra je realizován i systém logování, který je dostupný prostřednic-

tvím třídy `Log`, případně v modulech pomocí injektované funkce `LogFunc`. Existuje 5 úrovní logování, konkrétně jde o úrovně `ERROR`, `WARNING`, `INFO`, `VERBOSE`, `DEBUG`. Zvolená úroveň logování zahrnuje také všechny závažnější úrovně. Výchozí úroveň logování je úroveň `INFO`, která obsahuje pouze nejnutnější informace o průběhu provádění programu.

7.1.2 Analyzér

Tato část systému se stará výhradně o post-processing dat, která jsou poskytnuta vstupním modulem. V závislosti na podporovaných funkcích vstupního a výstupního modelu je provedena analýza, která v současné době zahrnuje výpočet procentuální hodnoty exkluzivního času (resp. počtu vzorků), možnost na základě flat view a call graph dat spočítat inkluzivní čas (resp. počet vzorků; pokud modul neohlásí, že se o tuto funkcionalitu stará sám), a dále předzpracovat flat view tak, aby byl v počátku seřazen podle exkluzivního času (resp. počtu vzorků) a počtu volání.

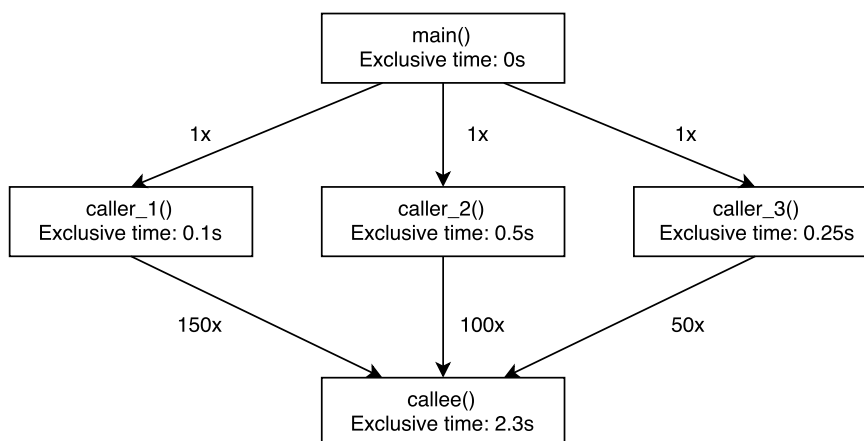
Analyzér pracuje pouze s takovými daty, jejichž typ vstupní modul ohlásil naplněním příslušného příznaku ve struktuře `IMF_SET`. Také bere v potaz schopnosti výstupního modulu, který naplněním struktury `OMF_SET` ohlásí typ dat, jež je schopen vizualizovat. Tyto struktury dovolují jak sjednotit možné drobné odchylky v chápání dat, tak možnost omezit se pouze na analýzu takových dat, které půjde předat uživateli.

Výpočet inkluzivního času

Inkluzivní čas lze vypočítat pouze za předpokladu, že má algoritmus k dispozici jak hodnoty exkluzivního času, tak údaje o kontextu funkčního volání. Za předpokladu, že máme k dispozici plnou hierarchii volání, lze inkluzivní čas vypočítat pouhým postupným přičítáním vzorků do všech úrovní hierarchie. To se děje v případě analýzy formátu *perf* a řeší to sám vstupní modul při získávání stromu volání.

Pokud není k dispozici plná hierarchie volání, ale pouze dvojice adres, z nichž jedna se nachází v těle volající a druhá v těle volané funkce, je třeba zvolit přístup jiný. Taková data jsou pro změnu typická pro profilování pomocí instrumentace, kde je zástupcem například nástroj *gprof*. Jak bylo v kapitole 3.2 uvedeno, pro každou dvojici adres funkčního volání je zaznamenán *call-*

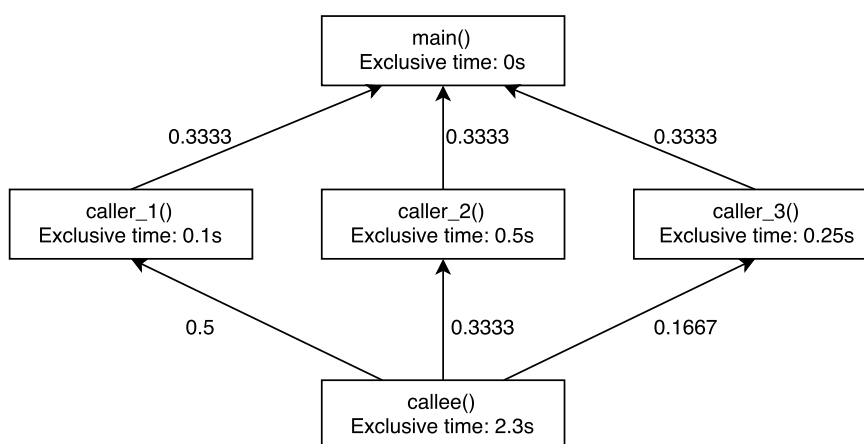
graph record, který krom samotných adres obsahuje ještě počet, kolikrát bylo takové funkční volání provedeno. Z těchto dat je nutné zrekonstruovat graf volání a poté na základě počtu volání a exkluzivního času volané funkce poměrovně vyhodnotit inkluzivní čas.



Obrázek 7.1: Zjednodušený výřez grafu volání

Mějme například graf volání znázorněný na obrázku 7.1. Každý uzel má ve výchozím stavu známý exkluzivní čas, a každá hrana mezi uzly, představující funkční volání, je ohodnocena počtem volání, které bylo mezi touto dvojicí uzlů uskutečněno. Nejdříve je pro každý uzel (funkci, symbol) vyhodnocen celkový počet volání. Následně je zrekonstruován invertovaný graf volání znázorněný na obrázku 7.2, jehož hrany jsou ohodnoceny poměrem počtu volání znázorněným touto hranou a celkového počtu volání cílového (nyní zdrojového) uzlu. Na příkladu lze vidět, že funkce `callee()` má celkový počet volání roven 300, z čehož funkce `caller_1()` byla zdrojem celkem 150 krát. Z toho vyplývá, že hrana mezi `callee()` a `caller_1()` bude ohodnocena číslem $\frac{150}{300}$, tedy 0.5.

V posledním kroku je pro každý uzel spuštěn algoritmus DFS, který postupně přiděluje redukovaný exkluzivní čas volané funkce všem v hierarchii nad ní. Opět lze předvést na funkci `callee()`, kde celkový exkluzivní čas je roven 2.3s. Ten je algoritmem propagován do uzlu `caller_1()` jako $2.3 * 0.5$, tedy 1.15s. Algoritmus pokračuje zanořením do funkce `main()`, kde je tento časový příspěvek redukován koeficientem hrany mezi `caller_1()` a `main()` jako $1.15 * 0.3333$, tedy (po zaokrouhlení) 0.3833s. Takto je expandován každý uzel grafu, kde počáteční hodnotou je vždy exkluzivní čas daného uzlu. Výsledný inkluzivní čas uzlu je pak součtem všech takto propagovaných příspěvků od volaných funkcí a vlastního exkluzivního času.



Obrázek 7.2: Invertovaný a ohodnocený graf volání

Zde může působit problémy rekurze, jelikož vytváří v grafu kružnici. Algoritmus je sice opatřen bezpečnostním mechanismem pro detekci kružnice v podobě značení vrcholů jako navštívených, ale v takovém případě se poté nemusí chovat korektně výpočet inkluzivního času, jelikož je i přesto redukován každou z hran v rekurzivním volání a v určitých případech není výsledkem správné číslo. Přímá rekurze problémem není, jelikož bezpečnostní mechanismus by nedovolil expandovat sebe sama, a tedy nějak znehodnotit výpočet. Problémy působí hlavně rekurze nepřímá, a to v případě, že nemá jednotný „vstupní bod“. Typickým příkladem je parsování výrazu rekurzivním sestupem, kde mohou být tvořeny různé kružnice v grafu volání, a to často s různými vstupními body. V takovém případě se musí inkluzivní čas v rámci kružnice propagovat neredukovaný, aby bylo dosaženo přesnějšího výsledku.

Pro tyto případy byl implementován algoritmus detekce silně souvislých komponent, konkrétně Tarjanův algoritmus. Ten je pouze modifikovanou variantou algoritmu DFS, která všem uzlům přidělí unikátní identifikátor komponenty, do které patří - na počátku je tedy počítáno s tím, že každý uzel patří do své vlastní silně souvislé komponenty a algoritmus bude pouze komponenty spojovat. Při procházení grafu do hloubky se uzly značí jako navštívené, a pokud algoritmus narazí na již navštívený uzel, propaguje zpětně identifikátor jeho komponenty až do úplného vynoření do onoho uzlu. Tím zařadí celou cestu do jedné silně souvislé komponenty. V případě vícenásobné kružnice, tedy takového podgrafu, který jako podgraf obsahuje více kružnic s neprázdným průnikem množin vrcholů, je za silně souvislou komponentu považováno sjednocení všech vrcholově nedisjunktních podgrafů - kružnic.

7.2 Vstupní moduly

Vstupní modul musí být kompilován jako dynamická knihovna, která implementuje funkce `CreateInputModule()` a `RegisterLogger()`. Funkce `CreateInputModule()` vytváří novou instanci třídy vstupního modulu, která vždy dědí od abstraktní třídy `InputModule` a implementuje její čistě virtuální metody. Funkce `RegisterLogger()` přejímá jako parametr ukazatel na funkci, která představuje logovací funkci v jádře.

Jádro přenechává veškerou logiku získávání dat vstupnímu modulu, a proto se každý vstupní modul stará sám i o validaci vstupů, případně obsahu vstupních souborů. Typicky je sice na vstupu právě jeden soubor, ale obecně to tak neplatí - například nástroj *OProfile* generuje celou adresářovou strukturu.

Data jsou předávána jádru pomocí implementovaných virtuálních metod abstraktní třídy `InputModule`, konkrétně se jedná o metody:

- `ReportName()` - ohlášení názvu vstupního modulu
- `ReportVersion()` - ohlášení verze vstupního modulu
- `ReportFeatures()` - ohlášení featur modulu předáním struktury `IMF_SET`
- `LoadFile()` - načtení vstupní cesty
- `GetFunctionTable()` - získání tabulky funkcí
- `GetClassTable()` - získání tabulky tříd (zatím není podporováno žádným z modulů)
- `GetFlatProfileData()` - získání základních profilových dat (flat view)
- `GetCallGraphMap()` - získání call graph struktury v podobě dvojic indexů funkcí z tabulky funkcí a počtem volání
- `GetCallTreeMap()` - získání call tree struktury v podobě seznamu sousednosti

Typicky je v metodě `LoadFile()` načten celý vstupní soubor (resp. adresářová struktura) a jsou vyextrahována potřebná data. Následně jsou pomocí `Get`-metod pouze předány jádru.

Co se týká jednotlivých featur modulu, jsou definovány v enumerátoru `InputModuleFeatures` takto:

- `IMF_FLAT_PROFILE` - podpora flat view a přidružených featur (exkluzivní čas, apod.)
- `IMF_CALL_GRAPH` - podpora grafu volání - modul je schopen získat údaje o funkčním volání
- `IMF_INCLUSIVE_TIME` - modul sám počítá inkluzivní čas
- `IMF_CALL_TREE` - podpora plného stromu volání - modul je schopen získat údaje o úplné hierarchii volání
- `IMF_USE_SECONDS` - modul používá jednotky času (vteřiny), tedy ne počet vzorků

Krom ohlašování podporovaných funkcionalit tedy může předávaná struktura featur obsahovat i nějaký přepínač, jakým je například `IMF_USE_SECONDS`. Ten se stará o to, že jako profilovací jednotku nebude používat bezrozměrné jednotky, které znamenají počet „přistižení“ funkce v hierarchii volání, ale jednotky času - zde konkrétně sekundy. Na výpočty ale tato změna nemá vliv, pouze na výstupní formátování a popisky.

Vstupní moduly také musí počítat s mapováním adres na názvy funkcí, a to za pomoci tzv. symbolů. Symboly jsou ve své podstatě pouze adresy označené názvem. Rozsah symbolu, tedy spodní a horní ohrazení např. funkce, je dán rozdílem adres v seřazené posloupnosti symbolů. Implementované moduly využívají nástroje `nm`, který je na linuxových distribucích součástí balíku `binutils`.

7.2.1 Vstupní modul - gprof

Načítání výstupního souboru nástroje *gprof* probíhá ve standardních etapách. Jako první je načtena hlavička souboru s profilovacími daty, je validována „magic“ značka¹ a podporovaná verze. Modul podporuje verzi formátu 1, jelikož se jedná o přibližně 10 let používaný ustálený formát. Předcházela

¹krátký identifikátor na začátku souboru, slouží k prvotnímu ověření správnosti očekávaného formátu

verze 0, která obsahuje pouze některé informace navíc v *basic block* záznamech.

Po validaci metadat následuje samotné čtení záznamů. Každý záznam je na začátku opatřen tzv. *tagem*, který určuje, o jaký záznam se jedná. Poté je další čtení rozřazeno dle přečteného tagu.

Histogramové záznamy

Z histogramového záznamu je přečteno rozmezí adres vymezených začátkem a koncem, počet jednotlivých *binů* a další údaje potřebné později. Histogramové záznamy by se neměly překrývat částečně - buď bude existovat více záznamů pro identický rozsah adres, nebo se záznamy nebudou překrývat vůbec. Po přečtení je tedy validováno překrývání, rozlišení - tedy jestli se nezměnil počet *binů*, jestli se nezměnila jednotka apod. Následuje rozdělení naměřených hodnot do interní kopie *binů* (pro případ překrývání) a tím je načítání u konce.

Po načtení jsou záznamy zpracovány do podoby flat profile tabulky. Nejdříve je provedeno předzpracování rozdělením hodnot v *binech* jednotlivým symbolům, a to rovnoměrně v závislosti na tom, jak zasahují do rozsahu jím vymezeného. Mějme například symbol A na adrese² 100 a symbol B na adrese 200. Analyzovaný *bin* zahrnuje rozsah adres 150 – 225 a obsahuje čítač o hodnotě 10. Symbol A proto zasahuje rozsahem 50 a symbol B rozsahem 25. Celkový rozsah je 75, proto první funkci případně hodnota $10 * \frac{50}{75} = 6.6667$ a druhé funkci hodnota $10 * \frac{25}{75} = 3.3333$. Díky způsobu jakým jsou údaje ukládány lze provést přímý převod na vteřiny pomocí profilovacího rozlišení z hlavičky histogramového záznamu, a to pouhým vynásobením. Postupným zpracováním všech *binů* je tedy získána celková hodnota exkluzivního času stráveného v dané funkci.

Call-graph záznamy

Tyto záznamy sestávají pouze z údajů uvedených v kapitole 3.2, tedy z adresy v těle volající funkce, adresy na začátku funkce volané a počtu těchto volání. Pokud je stejná funkce volána stejnou funkcí, jen z jiného místa,

²adresy jsou uváděny v decimálním formátu a pro názornost jsou zvoleny nízké hodnoty, v praxi tomu tak téměř nikdy není

je vytvořen jiný call-graph záznam - *gprof* při sběru dat nebere v potaz symboly, ale pouze adresy, které jsou zde vyhodnoceny jako odlišné.

Záznamy jsou uloženy a v následujícím kroku analyzovány právě vyhodnocením symbolů, kterým patří. Tyto hodnoty jsou sečteny a je z nich vytvořen podobný pohled, který již ale místo adres obsahuje indexy v tabulce funkcí a finální počet volání.

Jak bylo zmíněno v kapitole 7.1.2, tyto záznamy jsou stěžejním prvkem pro pozdější vyhodnocení inkluzivního času. Modul ale sám takovou funkcionalitu neobstrává a přenechává ji jádru.

Basic-block záznamy

Zpracování tohoto typu záznamů nebylo implementováno a vzhledem k odstranění podpory v novějších verzích nástroje *gprof* byla implementována pouze zpětná kompatibilita ve prospěch zbylé funkcionality v podobě přeskočení přesného počtu bajtů, který tento blok zaujímal.

7.2.2 Vstupní modul - perf

Modul pro čtení formátu nástroje *perf* je podstatně složitější kvůli komplikovanějším strukturám v tomto formátu uloženým. Vyplývá to z nutnosti obsáhnout širokou škálu druhů dat, které lze touto metodou a tímto nástrojem shromažďovat.

V rámci této práce byla implementována pouze základní škála těchto struktur pro potřeby běžné výkonnostní analýzy uživatelských procesů. Ta zahrnuje načtení hlavičky, kde se nacházejí potřebné offsety jednotlivých sekcí, následuje načtení sekce s atributy, podle které jsou později načítána jednotlivá pole datové sekce, načtení sekce s typy, která momentálně nemá využití, a nakonec načtení sekce s daty.

Načítání dat

V sekci s atributy je nyní využito pouze pole `sample_type`, jelikož se jedná o bitfield, kde jsou zapnuté bity podle toho, která data *perf* za dobu běhu

sbíral. Nejpodstatnější je pro potřeby současné podoby nástroje bit odpovídající hodnotě enumerátoru `PERF_SAMPLE_IP` a `PERF_SAMPLE_CALLCHAIN`. Hodnota `PERF_SAMPLE_IP` oznamuje, že je ve vzorcích obsažená hodnota instruction pointeru, tedy aktuální adresy prováděné instrukce. Hodnota `PERF_SAMPLE_CALLCHAIN` pak oznamuje, že k jednotlivým vzorkům je připojen i údaj o řetězu volání, tedy ve své podstatě snímek call stacku. To otevírá možnost práce s plnou hierarchií volání.

Sekce s typy není v současné podobě nijak využita, je implementován pouze úryvek jejího načítání.

Nejdůležitější částí je ale sekce se samotnými nasbíranými událostmi - sekce datová. Ta obsahuje posloupnost po sobě jdoucích záznamů, které mají různé typy. Nejdříve je vždy přečtena hlavička tohoto záznamu, která obsahuje jak typ, tak velikost dat k přečtení. V současné podobě je důležitá jen omezená množina záznamů, konkrétně jde o typy označené hodnotou enumerátoru `PERF_RECORD_MMAP`, `PERF_RECORD_MMAP2` a `PERF_RECORD_SAMPLE`. Události typu `mmap` jsou důležité kvůli například vyhledávání symbolů. V podstatě jde o záznam, který přímo propojuje určitý rozsah adres ve virtuálním paměťovém prostoru procesu s konkrétním souborem, respektive obecně zdrojem. V případě události `PERF_RECORD_MMAP` jde často o mapování kernelových modulů a jiných, často obtížně ovlivnitelných zdrojů. V případě `PERF_RECORD_MMAP2` jde o často o mapování nějaké knihovny, typicky *libc* nebo jiných, které jsou spojené se samotným vývojem profilované aplikace. Událost `PERF_RECORD_SAMPLE` pak představuje samotný vzorek pořízený na základě zkoumané veličiny. Ten je složen z informací, jejichž sada je zaznamenána v poli `sample_type`. Jak bylo uvedeno, důležitými body jsou krom samotného vzorkovaného instruction pointeru i snímek call stacku. Tyto informace jsou v rámci načítání extrahovány.

V nástroji lze vidět implementaci podpory i pro záznamy typu `PERF_RECORD_COMM`, `PERF_RECORD_FORK` a `PERF_RECORD_EXIT`. Ty ale v současnosti nejsou nijak využívány vzhledem k tomu, že pro profilování jediného procesu nejsou podstatné. Jak již názvy napovídají, jde o záznamy spojené s vytvářením a zanikáním procesů.

V poslední fázi načítání jsou přečteny záznamy typu `mmap`, a je provedeno načtení debuggovacích symbolů z takto mapovaných knihoven pomocí nástroje `nm`. Nejdříve je proveden pokus o načtení ze standardního umístění knihoven ve verzích pro debugging a pokud odpovídající knihovna není nalezena, je načtena z původního umístění (to je uvedeno v příslušných `mmap` záznamech

načtených v předchozím kroku) a je proveden pokus o získání symbolů z ní. Ten ale nemusí být úspěšný - symboly knihovna ve standardním umístění obsahuje pouze tehdy, jsou-li do ní cíleně zapraveny. Takové knihovny lze na linuxových systémech ve standardních balíčkovacích systémech najít se suffixem *-dev*.

Také jsou načteny kernelové symboly ze souboru `/proc/kallsyms`, pokud na to má současný uživatel práva. Standardně totiž práva nemá a tento soubor je zpřístupněn pouze uživateli `root`.

Předzpracování

Důležitým znakem tohoto modulu je fakt, že naměřená data nejsou nijak převáděna do časových jednotek. Vzhledem k tomu, že dochází ke vzorkování v poměrně vysokých frekvencích za sběru různého objemu dat, je vyvinut pouze *best effort*³ pro dodržení rozestupu mezi pořizováním vzorků, ale nelze to předpokládat.

Nejdříve jsou filtrovány načtené symboly podle toho, jsou-li skutečně v nasbíraných vzorcích obsaženy. Tato operace urychlí vyhledávání a minimalizuje objem dat předávaných jádru a následně výstupnímu modulu. Nutno dodat, že jsou v nasbíraných datech obsaženy jako hodnota instruction pointeru i adresy, které jsou mimo mapovaný rozsah. Takovým hodnotám je vytvořen nový symbol v rozsahu 100 adres před a za navzorkovanou adresu a je označený jako nenalezený. Pokud se hodnota nachází v mapovaném rozsahu, ale není pro ni nalezen platný symbol, je opět vytvořen nový symbol, taktéž označený jako nenalezený, ale do jména je přidáno jméno knihovny, z níž se nepodařilo získat potřebné symboly.

Jako další krok následuje zpracování flat view, což spočívá v pouhém nasčítání výskytů dané funkce v nasnímaných datech. V této proceduře je počítán jak exkluzivní počet vzorků, kdy je vzorek přičten do vlastního pole pro exkluzivní počet vzorků pouze poslednímu článku callchainu, tak inkluzivní, kdy je vzorek přičten do pole pro inkluzivní počet vzorků všem článkům callchainu.

Následuje vyhodnocení call grafu, který je také sestaven z callchainů, a to čistě nasčítáním výskytu dvojic indexů po sobě jdoucích symbolů v rámci

³výraz standardně používaný pro označení snahy maximalizovat efektivitu a spolehlivost, která ale nemusí být vždy úspěšná

jednoho řetězu. Podpora call grafu pro nástroj *perf* není obvykle součástí ostatních nástrojů, které dovolují vizualizovat data jím nasbíraná. Důvodem je často průkaznější identifikace bottlenecku pomocí stromu volání. K prvotní identifikaci možného problému je ale tento pohled poměrně důležitý, a to právě díky možnosti zobrazit *hot path*, tedy posloupnost funkcí, které svým voláním tvoří největší výkonnostní problém.

Posledním krokem je právě vyhodnocení call tree, respektive úplné ohodnocené hierarchie volání. V podstatě jde o „sečtení“ všech callchainů, a to do složitější struktury, zde konkrétně základní implementace stromu. Ten je sestavován tak, že pro každý callchain je započato prohledávání mezi kořenovými uzly, které zároveň uzly vytváří, pokud ještě neexistují. Při procházení je opět stejným způsobem přičítán exkluzivní a inkluzivní počet vzorků.

7.3 Výstupní moduly

Konceptem se výstupní moduly od vstupních příliš neliší. Také musí být kompilovány do podoby dynamické knihovny a implementovat funkce `CreateOutputModule()` a `RegisterLogger()`. Analogicky k modulu vstupnímu i modul výstupní funkcí `CreateOutputModule()` tvoří instanci třídy, která dědí od abstraktní třídy `OutputModule` a implementuje její čistě virtuální metody.

Vzhledem k velmi omezené množině požadavků na výstupní modul, které mohou být jádrem vyvolány, není toto rozhraní tak složité. Funkce obecné, tedy `ReportName()`, `ReportVersion()` a `ReportFeatures()` zůstávají stejné, a jedinou metodou, která je využívána v rámci vizualizace je metoda `VisualizeData()`. Ta přejímá jako parametr strukturu `NormalizedData` generovanou v předchozích krocích běhu programu.

Výstupní modul má samozřejmě oddělenou množinu featur od modulu vstupního. Tyto featury jsou definovány v enumerátoru `OutputModuleFeatures`, a v současné době se jedná o tyto hodnoty:

- `OMF_FLAT_PROFILE` - podpora flat view a práce s takovými hodnotami
- `OMF_CALL_GRAPH` - podpora práce s daty grafu volání
- `OMF_CALL_TREE` - podpora práce s úplnou hierarchií volání

Tyto featury jsou poměrně obecné a oznamují pouze charakter dat, se kterými je výstupní modul schopen pracovat. Například oznámením featury `OMF_CALL_TREE` pouze výstupní modul dal vědět, že je schopen pracovat s úplnou hierarchií volání. Z té se ale dá generovat více pohledů, než je jen obyčejný strom volání - například i flame graph.

7.3.1 Výstupní modul - webová stránka

Jediným výstupním modulem v současné podobě je modul, který podporuje generování webového hlášení. Je implementován šablonovací systém s podporou jednoduchých značek pro plnění proměnnými daty. Princip fungování tohoto modulu spočívá pouze v načtení šablon, rozparsování, naplnění daty ze vstupu a uložení do výstupního adresáře. Žádná pokročilejší logika se zde nenachází, ta je obsažena až v generované šabloně v podobě funkcí psaných v jazyce Javascript. Ty se starají o generování dynamického a interaktivního výstupu, zpřístupnění filtrů a nějaké dodatečné výpočty, obarvování uzlů a další funkcionalitu.

Šablonovací systém

Základem celého modulu je systém práce se šablonami. Ty jsou uloženy v podsložce `HtmlTemplates`. Modul nejdříve načte soubor `template.html`, a v něm může být specifikováno, jaké další soubory chceme načíst a vložit, případně pouze zkopírovat z výchozí složky do výstupní.

V šablonách lze použít značky pro takové akce. Značka je vždy započata sekvencí znaků `<#` a ukončena `#>`. Jedná se o tyto značky:

- `<#INCLUDE soubor#>` - načtení, rozparsování a zpracování dalšího souboru šablony
- `<#COPYFILE soubor#>` - zkopíruje soubor do cílového umístění
- `<#BEGIN název_bloku#>` - označení začátku bloku, vždy znamená opakování pro každý záznam daného bloku
- `<#END název_bloku#>` - označení konce bloku
- `<#VALUE název_pole#>` - v rámci bloku je tato značka nahrazena hodnotou pole; hodnota je escapována podle HTML pravidel

- `<#JSVALUE název_pole#>` - v rámci bloku je tato značka nahrazena hodnotou pole; hodnota je escapována podle Javascript pravidel
- `<#RAWVALUE název_pole#>` - v rámci bloku je tato značka nahrazena hodnotou pole; hodnota není escapována

Typy bloků a podporované názvy polí v rámci nich:

- SUMMARY - souhrnné informace o profilovém sezení
 - KEY - název hodnoty, titulek
 - CONTENT - hodnota
- FLAT_VIEW_ROWS - jednotlivé řádky flat view pohledu
 - PCT_TIME - procentuální exkluzivní čas (počet vzorků)
 - PCT_INCLUSIVE_TIME - procentuální inkluzivní čas (počet vzorků)
 - TOTAL_TIME - absolutní exkluzivní čas (počet vzorků)
 - TOTAL_INCLUSIVE_TIME - absolutní inkluzivní čas (počet vzorků)
 - CALL_COUNT - počet volání
 - FUNCTION_NAME - název funkce
 - FUNCTION_TYPE - typ funkce
- CALL_GRAPH_DATA - data grafu volání
 - CALLER / CALLEE_ID - ID volající/volané funkce
 - CALLER / CALLEE_NAME - název volající/volané funkce
 - CALLER / CALLEE_FUNCTION_TYPE - type volající/volané funkce
 - CALLER / CALLEE_TOTAL_CALL_COUNT - celkový počet volání volající/volané funkce
 - CALLER / CALLEE_FLAT_TIME_PCT - procentuální exkluzivní čas volající/volané funkce
 - CALLER / CALLEE_FLAT_TIME - absolutní exkluzivní čas volající/volané funkce
 - CALLER / CALLEE_FLAT_INCLUSIVE_TIME_PCT - procentuální inkluzivní čas volající/volané funkce
 - CALLER / CALLEE_FLAT_INCLUSIVE_TIME - absolutní inkluzivní čas volající/volané funkce

- `CALL_COUNT` - počet volání mezi aktuální dvojicí funkcí
- `CALL_TREE_DATA` - data stromu volání
 - `ID_CHAIN` - posloupnost ID funkcí ve stromu volání (právě jedna cesta od kořene stromu až k listu)
 - `TIME_CHAIN` - posloupnost inkluzivních časů
 - `TIME_PCT_CHAIN` - posloupnost procentuálních inkluzivních časů
 - `SAMPLE_COUNT_CHAIN` - posloupnost počtů vzorků (v případě že modul podporuje jak jednotky časové, tak jednotky počtu vzorků)

Jediná netypicky generovaná data jsou obsažena v rámci stromu volání. Při generování je spuštěn z každého kořene algoritmus DFS. Jakmile je po čas procházení dosaženo listu, je zpětně vygenerována posloupnost identifikátorů funkcí, času, procentuálního času a počtu vzorků, které jsou propsány do šablony v podobě čárkou oddělených hodnot. Počítá se s tím, že v šabloně je obslužný skript, který dovede naparsovat takovou posloupnost a zrekonstruovat strom volání včetně potřebné interaktivity.

Připravená šablona

V rámci této práce byla zpracována i šablona, v rámci níž je podporována základní škála pohledů. Jedná se o šablonu pro webovou stránku, která je psaná v jazycích HTML a Javascript a nastylována pomocí CSS.

Přehled souborů:

- `template.html` - hlavní soubor šablony
- `header.template.html` - záhlaví stránky; zde jsou například značky pro nakopírování souborů podpůrných knihoven
- `footer.template.html` - zápatí stránky
- `functions.js` - hlavní logika vizualizace, zpracování dat vygenerovaných výstupním modulem, předání knihovnám, generování pohledů
- `style.css` - soubor s hlavními styly stránky
- `jquery.js` - knihovna jQuery

- `jquery.tablesort.min.js` - knihovna jQuery Tablesort, použitá pro řazení flat view
- `vis.min.js` - knihovna vis.js použitá pro generování call graph pohledu
- `vis.min.css` - soubor se styly knihovny vis.js

V šabloně je implementována podpora základního přehledu, kde jsou vypsaný jádrem a moduly sestavená metadata ve formátu klíč-hodnota. Tento pohled lze vidět na obrázku 7.3.

PIVO - Report	
BASIC INFO	FLAT PROFILE
CALL GRAPH	CALL TREE
FLAME GRAPH	
PROFILING SUMMARY	
Analyzed function count	374
Binary file	../../GA_EquationSolving/a.out
Input module name	perf input module
Input module version	0.1-dev
Input path	perf.data
Output module name	HTML File Output module
Output module version	0.1-dev
Total execution time	106.14s

Obrázek 7.3: Základní přehled generovaný nástrojem PIVO

Dalším pohledem je flat view, který sestává z tabulky, v níž každý řádek představuje jednu funkci. V rámci řádku tabulky jsou vypsané exkluzivní a inkluzivní časy (počty vzorků) a počet volání. Také je vypsáno jméno funkce, a ve vrstvě pod ním lze vidět obdélník, jehož šířka je odvozena od exkluzivního času a barva podle lokace v rámci analyzované aplikace. Pokud pochází daná funkce z původní aplikace ze sekce `.text`⁴, je podbarvena zeleně. Pokud pochází z jiného zdroje, je podbarvena červeně. Výjimku tvoří funkce z jádra operačního systému, které jsou podbarveny modře. Takové rozlišení dovoluje odlišit funkce, u kterých je možné optimalizovat jejich samotný kód (typicky funkce ze zkoumaného programu) od funkcí, kde lze optimalizovat

⁴obvykle read-only sekce binárního souboru, ve které je uložen samotný kód programu

pouze způsob a počet volání (typicky knihovní funkce). Celou tabulku lze řadit kliknutím na záhlaví sloupce, a to jak vzestupně, tak sestupně. Také je možné definovat filtrovací kritéria. Náhled lze vidět na obrázku 7.4

PIVO - Report

BASIC INFO FLAT PROFILE CALL GRAPH CALL TREE FLAME GRAPH

Filter:
Inclusive time [%] > 1
ADD FILTER

Time [%]	Time [s]	Inclusive time [%]	Inclusive time [s]	Call count	Function name
10.86	11.52	56.80	15.01	77176	rpn_evaluate_stack(c_stack*, double*)
3.20	3.39	2.85	0.75	8967	rpn_pop_two_values(c_stack*, double*, double*)
2.69	2.85	84.83	22.41	89089	fitness_function(chromosome*)
2.40	2.54	13.63	3.60	18097	rpn_apply_operator(operator_type, c_stack*)
0.48	0.51	18.96	5.01	20114	perform_crossover(chromosome**, int)
0.43	0.46	90.75	23.97	90483	recalculate_fitness(chromosome**, int)
0.32	0.34	2.20	0.58	2705	__GI_vfprintf
0.29	0.31	1.68	0.44	2415	perform_mutation(chromosome**)
0.22	0.23	16.64	4.40	17072	select_random_chromosome(chromosome**, int, chromosome*)
0.05	0.06	1.79	0.47	1444	libstdc++.so.6.0.19:0x00007fbb084f2540
0.03	0.04	99.86	26.38	98881	main
0.02	0.02	3.08	0.81	2741	__vsnprintf
0.00	0.00	3.13	0.83	2872	libstdc++.so.6.0.19:0x00007fbb084fc5d0
0.00	0.00	3.28	0.87	2507	libstdc++.so.6.0.19:0x00007fbb084fc5d0
0.00	0.00	3.13	0.83	0	libstdc++.so.6.0.19:0x00007fbb084fc5d0
0.00	0.00	100.00	26.42	0	__libc_start_main
0.00	0.00	2.73	0.72	0	libstdc++.so.6.0.19:0x00007fbb084f2540

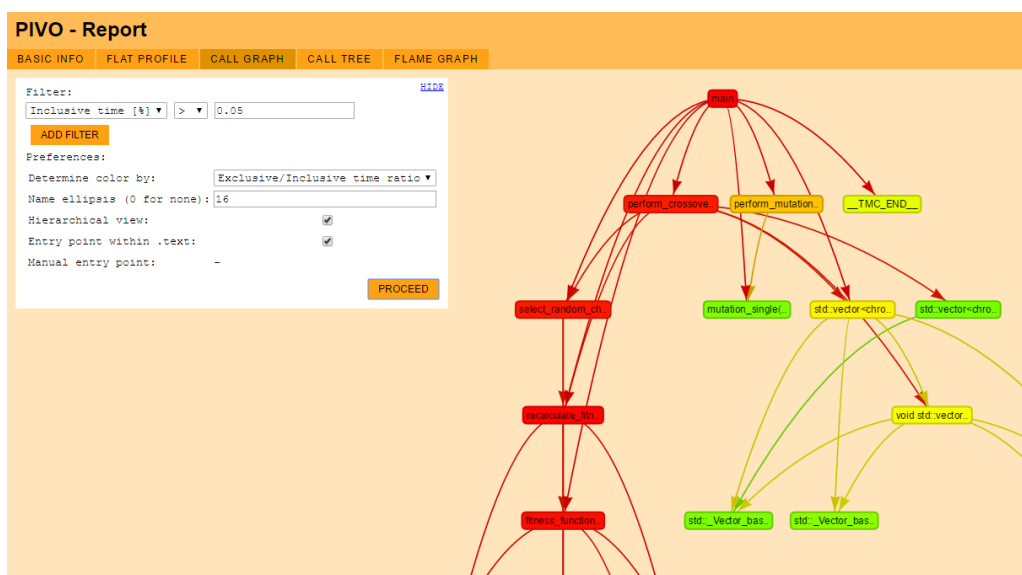
Obrázek 7.4: Flat view generovaný nástrojem PIVO

Šablona dále podporuje call graph pohled, kde využívá funkcionalitu z knihovny `vis.js`. Ten má dva různé způsoby, jakými je uspořádán. Prvním je způsob hierarchický, kdy je každému uzlu předpočítána úroveň a knihovna se snaží vybalancovat takový graf pomocí vestavěné simulace fyziky do čitelné podoby. Druhým způsobem je graf bez generované organizace, který sice postrádá hierarchii, ale často dovede bez ztráty přehlednosti zobrazit podstatě větší množství uzlů. Zde je opět výsledné uspořádání balancováno pomocí vestavěné simulace fyziky, zde pro změnu v obou rozměrech. Ukázku hierarchického pohledu lze vidět na výřezu na obrázku 7.5.

Graf je také interaktivní. Tažením myši lze uzly přesouvat, po najetí kurzorem myši na uzel jsou zobrazeny detailní informace o funkci, kterou daný uzel představuje. Také je možné dvojklikem na uzel expandovat graf pouze z vybraného uzlu, čímž se dá efektivně redukovat celý výstup, a zaměřit se pouze na jeho část. V neposlední řadě lze definovat filtrovací kritéria, zvolit

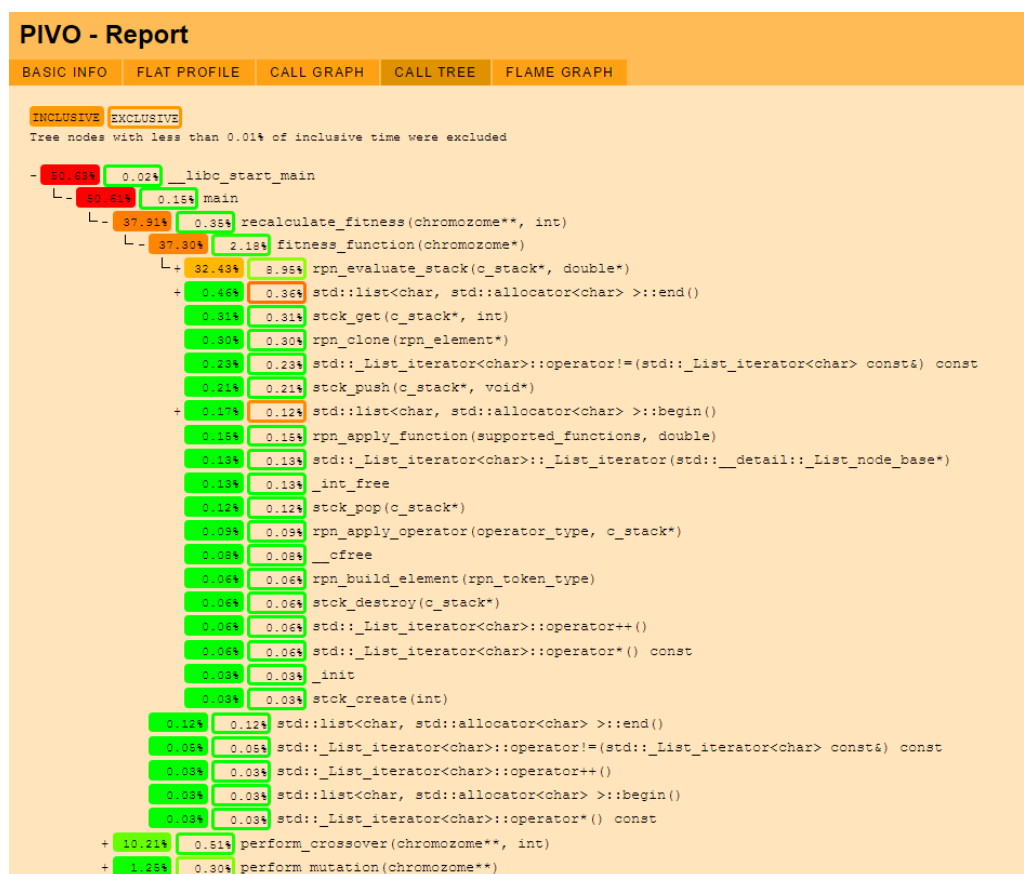
způsob, podle kterého se budou uzly grafu obarvovat, omezit vstupní bod pouze na funkce ze sekce `.text`, nebo zvolit maximální délku názvu symbolu ve výstupu.

Co se týká barvení, uzly lze obarvit standardně podle exkluzivního nebo inkluzivního času (počtu vzorků), počtu volání, nebo podle poměru inkluzivního a exkluzivního času (popř. obráceným poměrem). Barvení pomocí poměru exkluzivního ku inkluzivního času dopomáhá odhalit hot path, obrácený poměr pak zvýrazní uzly, ve kterých je poměrově tráveno exkluzivně minimum času, ale tvoří vstupní bod do nějaké náročnější komponenty. Barvená škála je lineárně interpolována od červené (největší hodnoty) přes žlutou až po zelenou (nejnižší hodnoty), a vždy je dynamicky vyhodnocena podle viditelných dat.



Obrázek 7.5: Výřez pohledu call graph generovaný nástrojem PIVO

Dalším podporovaným pohledem je pohled hierarchický v podobě stromu volání. V základním pohledu jsou vidět pouze kořeny jednotlivých stromů, kterých může být více vzhledem k tomu, že po dobu provádění programu se může v rámci jednoho procesu vykonávat více vláken, případně jsou například nástrojem *perf* navíc vzorkovány ještě další kontexty. Tyto kořeny lze kliknutím expandovat a vypsat tak všechny potomky v rámci daného podstromu. V rámci řádky, která opět představuje jedno zanoření v hierarchii volání, lze vidět na levé straně dva obdélníky. Plný obdélník obsahuje procentuálně inkluzivní čas (počet vzorků) a obdélník pouze s rámem čas exkluzivní. Barvení je opět přizpůsobeno viditelné škále.



Obrázek 7.6: Call tree generovaný nástrojem PIVO

Posledním v současnosti podporovaným pohledem je flame graph. Ten je sestavený za pomoci stejných dat jako pohled předchozí, tedy strom volání. Tento pohled lze opět rozklikávat a měnit tak výchozí bod, ze kterého bude expandována hierarchie volání. Barvy jsou v tomto pohledu voleny náhodně, jelikož není třeba žádné další rozlišení. Inkluzivní čas je znázorněn šířkou sloupce, exkluzivní čas tím, co „přebývá“ oproti další úrovni hierarchie. Po rozklikání lze resetovat pohled volbou **Reset view**.

8 Uživatelská dokumentace

V této kapitole bude popsán postup stažení, instalace a používání vytvořeného nástroje.

8.1 Získání

Software je dostupný buď z online zdrojů nebo z přiloženého CD, a to v podobě zdrojových souborů s připravenými skripty pro generování sestavovacích konfigurací.

Pro přeložení jádra tohoto nástroje je třeba mít tyto prerekvizity:

- Sestavovací nástroj dle platformy (*gcc + make, MS Visual Studio, ..*)
- Nástroj *CMake*

Pro vstupní moduly formátů *gprof* a *perf* je navíc potřeba software překládat na OS GNU/Linux, a mít nainstalovaný balíček **binutils**. Webový výstupní modul nemá žádné dodatečné závislosti.

Online zdroje

Pro získání nástroje z online zdrojů je nutné buď mít nainstalovaný verzovací nástroj *git* a naklonovat jím příslušný vzdálený repozitář, nebo lze stáhnout zdrojové soubory zabalené v archivu díky službě portálu *github.com*

Přehled repozitářů:

- <https://github.com/ProjectPIVO/pivo-core> - jádro nástroje - je potřeba v každém případě
- <https://github.com/ProjectPIVO/pivo-input-gprof> - *gprof* vstupní modul

- <https://github.com/ProjectPIVO/pivo-input-perf> - *perf* vstupní modul
- <https://github.com/ProjectPIVO/pivo-output-html> - webový výstupní modul
- <https://github.com/ProjectPIVO/pivo-input-dummy> - kostra implementace vstupního modulu pro případ tvorby nového

Príslušnou adresu buď použijeme jako zdroj pro nástroj *git*, nebo jako URL, kde lze po navigování webovým prohlížečem najít tlačítko s popiskem „Download ZIP“, kterým je spuštěna archivace a stažení.

Nejdříve provedeme vybraným způsobem stažení jádra nástroje, přesuneme se do podadresáře **modules**, kam můžeme vkládat adresáře obsahující jednotlivé moduly nástroje. Nástroj *CMake* je pak dovede zaregistrovat pomocí skriptu, který je v repozitáři přibalen.

Příložené CD

Při sestavení a instalaci nástroje z příloženého CD je vhodné nejdříve přepokopírovat obsah příslušného adresáře někam na místní disk. V připravené adresářové struktuře jsou připraveny všechny moduly nástroje, není tedy nutné nic dodatečně stahovat nebo přesouvat.

8.2 Instalace

V připravené adresářové struktuře je nyní potřeba vygenerovat konfigurační soubory pro sestavení, a to nástrojem *CMake*. V případě konzolové verze stačí zadat **cmake** . v kořenovém adresáři adresářové struktury nástroje - je použit výchozí kompilátor pro danou platformu, pokud je přítomen. Běžně je k dispozici i grafické rozhraní pro nástroj *CMake*, kde stačí zadat vstupní a výstupní cestu, zvolit kompilátor, kterým se chystáme projekt sestavit, a tlačítkem „Configure“ a „Generate“ spustit generování.

Následně je možné projekt sestavit zvoleným nástrojem.

Výstupem je složka **bin**, která obsahuje jak přeložené jádro, kterým se nástroj spouští, tak všechny vybrané moduly v podobě dynamických knihoven,

a také všechny přidružené soubory, které moduly vyžadují. V případě webového výstupního modulu to je složka se šablonami s názvem `HtmlTemplates`.

Příklad posloupnosti příkazů instalace na GNU/Linux

```
# Naklonovani vzdaleneho repozitare jadra
git clone https://github.com/ProjectPIVO/pivo-core
# Presun do slozky s moduly
cd pivo-core/modules
# Naklonovani vseh dostupnych modulu
git clone https://github.com/ProjectPIVO/pivo-input-gprof
git clone https://github.com/ProjectPIVO/pivo-input-perf
git clone https://github.com/ProjectPIVO/pivo-output-html
# Presun zpet do korenoveho adresare
cd ..
# Spusteni CMake a vygenerovani Makefile
cmake .
# Sestaveni nastroje
make
```

8.3 Spuštění

Nástroj je standardně spouštěn přes aplikaci vzniklou kompilací jádra, která nese název `pivo-core`. Veškerá parametrizace probíhá pomocí argumentů příkazové řádky, není zde tedy žádný konfigurační soubor nebo jiný způsob.

Parametry, které lze předat aplikaci jsou tyto:

- `-im <název>` - (povinný) specifikace vstupního modulu (`gprof`, `perf`, `..`)
- `-om <název>` - (povinný) specifikace výstupního modulu (`html`, `..`)
- `-i <cesta>` - (povinný) specifikace vstupního souboru nebo adresáře - typicky výstup profileru (`gmon.out`, `perf.data`, `..`)
- `-o <cesta>` - cesta pro generování výstupu; ve výchozím nastavení se používá současný pracovní adresář
- `-b <cesta>` - cesta ke spustitelnému souboru, který byl v rámci profilingového sezení analyzován

- `-ll <číslo>` - logovací úroveň (0 = vypnuto, 1 = **ERROR**, 2 = **WARNING**, 3 = **INFO**, 4 = **VERBOSE**, 5 = **DEBUG**); ve výchozím nastavení je použita úroveň 2 = **WARNING**
- `-lf <cesta>` - soubor, do kterého se budou ukládat logy; ve výchozím nastavení vypnuto
- `-s` - kompletně vypíná veškerý výstup na konzoli (**stdout**, **stderr**)

Příkaz pro zpracování výstupu nástroje *gprof* z analýzy programu `a.out` za použití webového výstupního modulu, kdy specifikujeme i parametr pro detailnější log navíc ukládaný do souboru, může například vypadat takto:

```
./pivo-core -im gprof -om html -i gmon.out -b a.out -ll 4 -lf  
pivo.log
```

Jelikož jsme nspecifikovali výstupní složku, budou vygenerované soubory webové stránky uloženy do současného pracovního adresáře.

9 Ověření výsledků

V této kapitole budou prezentovány výstupy realizovaného nástroje na základě výkonnostní analýzy dvou existujících projektů a ve vhodných případech budou porovnány s výstupy již existujících nástrojů. Takto analyzované výstupy by měly korespondovat minimálně v hodnotách časů, respektive počtů vzorků. Také bude provedena krátká diskuse nad výsledky.

9.1 Jednoduchá aplikace a gprof

Pro potřeby porovnání výsledků je prvním vybraným programem pro výkonnostní analýzu a následnou vizualizaci program jednodušší - v tomto případě se jedná o semestrální práci z předmětu KIV/PRO. V rámci ní byl implementován genetický algoritmus pro nalezení alespoň jednoho vektoru řešení rovnice o N neznámých. Program tedy sestává z parsování matematického výrazu algoritmem shunting yard do podoby RPN¹ zásobníku, opakovaného vyhodnocení takto zpracovaného zásobníku, a pak z operací souvisejících s genetickými algoritmy, jako je například křížení a mutace.

Zdrojové kódy jsou k dispozici na adrese:

https://github.com/MartinUbl/GA_EquationSolving

Výstup flat view nástroje původního (obr. 9.1) a nástroje *PIVO* (obr. 9.2) se formou moc neliší - jde o tabulku funkcí s časy, procenty a počtem volání. Na první pohled lze však vidět, že původní *gprof report* výstup neobsahuje inkluzivní časy, a také neobsahuje žádnou podpůrnou vizualizaci v podobě podbarvení řádků, jako výstup nástroje *PIVO*. Oproti tomu původní výstup obsahuje pole „*us/call*“, které uvádí průměrný počet mikrosekund na jedno volání, a také kumulativní čas. Jedná se ale spíše o podpůrné hodnoty, jejichž význam je do velké míry nahrazen kombinací exkluzivního a inkluzivního času.

¹Reverse Polish notation, také nazývaná postfixová notace, je tvar matematického výrazu, který lze snadno strojově vyhodnotit

% time	cumulative seconds	self seconds	calls	self us/call	total us/call	name
23.89	0.16	0.16	1280784	0.12	0.40	rpn_evaluate_stack(c_stack*, double*)
14.93	0.26	0.10	15369430	0.01	0.01	stck_push(c_stack*, void*)
11.95	0.34	0.08	5123136	0.02	0.02	rpn_pop_two_values(c_stack*, double*, double*)
5.97	0.38	0.04	15369419	0.00	0.00	stck_pop(c_stack*)
5.97	0.42	0.04	3842352	0.01	0.01	rpn_apply_function(supported_functions, double)
4.48	0.45	0.03	15369479	0.00	0.00	stck_get(c_stack*, int)
4.48	0.48	0.03	1280784	0.02	0.46	fitness_function(chromosome*)
2.99	0.50	0.02	6403920	0.00	0.00	rpn_clone(rpn_element*)
2.99	0.52	0.02	5123151	0.00	0.00	rpn_build_element(rpn_token_type)
2.99	0.54	0.02	3842359	0.01	0.01	std::_List_iterator<char>::_List_iterator(std::_detail::_List_node_base*)
2.99	0.56	0.02	2561572	0.01	0.01	std::list<char, std::allocator<char> >::end()
1.49	0.57	0.01	5123136	0.00	0.02	rpn_apply_operator(operator_type, c_stack*)
1.49	0.58	0.01	2561571	0.00	0.00	std::_List_iterator<char>::operator!=(std::_List_iterator<char> const&) const
1.49	0.59	0.01	2126398	0.00	0.00	frand()
1.49	0.60	0.01	1280786	0.01	0.01	stck_create(int)
1.49	0.61	0.01	100165	0.10	0.10	mutation_single(chromosome*)
1.49	0.62	0.01	26133	0.38	16.72	select_random_chromosome(chromosome**, int, chromosome*)
1.49	0.63	0.01	22150	0.45	0.45	_gnu_cxx::new_allocator<chromosome*>::deallocate(chromosome**, unsigned long)
1.49	0.64	0.01	10000	1.00	1.00	return_fittest(chromosome**, chromosome*)
1.49	0.65	0.01	10000	1.00	2.52	perform_mutation(chromosome**)
1.49	0.66	0.01	10000	1.00	1.00	print_population(int, chromosome**)
1.49	0.67	0.01	10000	1.00	46.17	perform_crossover(chromosome**, int)
0.00	0.67	0.00	1280787	0.00	0.01	std::list<char, std::allocator<char> >::begin()

Obrázek 9.1: Flat view (výřez) nástroje *gprof report* z běhu genetického algoritmu

Time [%]	Time [s]	Inclusive time [%]	Inclusive time [s]	Call count	Function name
23.88	0.16	76.12	0.51	1280784	rpn_evaluate_stack(c_stack*, double*)
14.93	0.10	14.93	0.10	15369430	stck_push(c_stack*, void*)
11.94	0.08	15.92	0.11	5123136	rpn_pop_two_values(c_stack*, double*, double*)
5.97	0.04	5.97	0.04	15369419	stck_pop(c_stack*)
5.97	0.04	5.97	0.04	3842352	rpn_apply_function(supported_functions, double)
4.48	0.03	4.48	0.03	15369479	stck_get(c_stack*, int)
4.48	0.03	83.08	0.56	1280784	fitness_function(chromosome*)
2.99	0.02	2.99	0.02	6403920	rpn_clone(rpn_element*)
2.99	0.02	2.99	0.02	5123151	rpn_build_element(rpn_token_type)
2.99	0.02	2.99	0.02	3842359	std::_List_iterator<char>::_List_iterator(std::_detail::_List_node_base*)
2.99	0.02	4.98	0.03	2561572	std::list<char, std::allocator<char> >::end()
1.49	0.01	17.41	0.12	5123136	rpn_apply_operator(operator_type, c_stack*)
1.49	0.01	1.49	0.01	2561571	std::_List_iterator<char>::operator!=(std::_List_iterator<char> const&) const
1.49	0.01	1.49	0.01	2126398	frand()
1.49	0.01	1.49	0.01	1280786	stck_create(int)
1.49	0.01	1.56	0.01	100165	mutation_single(chromosome*)
1.49	0.01	61.60	0.41	26133	select_random_chromosome(chromosome**, int, chromosome*)
1.49	0.01	1.49	0.01	22150	_gnu_cxx::new_allocator<chromosome*>::deallocate(chromosome**, unsigned long)
1.49	0.01	64.82	0.43	10000	perform_crossover(chromosome**, int)
1.49	0.01	3.76	0.03	10000	perform_mutation(chromosome**)
1.49	0.01	1.49	0.01	10000	print_population(int, chromosome**)
1.49	0.01	1.49	0.01	10000	return_fittest(chromosome**, chromosome*)

Obrázek 9.2: Flat view (výřez) nástroje *PIVO* z běhu genetického algoritmu

Na pohledu call-graph lze snadno demonstrovat výhodu interaktivity. Obrázek 9.3 obsahuje call-graph vygenerovaný běžně používaným nástrojem *gprof2dot*, který musel být ale ručně oříznut v grafickém editoru, aby se vešel na stránku alespoň otočený. Za podpory interaktivity je oproti tomu možné spousty informací skrýt a ponechat viditelné jen ty nejdůležitější - konkrétní časy mohou být vidět až po najetí myši, stejně jako celá jména funkcí. Dále je hlavně v rozsáhlejších grafech velmi vhodné mít možnost filtrování a případně i změny stylu obarvování. Na obrázku 9.4 lze vidět pohled generovaný nástrojem *PIVO* odpovídající původnímu z nástroje *gprof2dot*. Na obrázku 9.5 lze pak vidět obarvený graf podle poměru exkluzivního a inkluzivního času, což dovoluje vizuálně oddělit *hot path* od „méně podstatné“ části grafu.

Jednoznačný výkonnostní problém obsahuje funkce `rpn_evaluate_stack()`, což lze poznat jak z velkého procenta exkluzivního času (flat view), tak z call-graph pohledu, kde tímto uzlem ostře končí *hot path*. Dále by bylo vhodné zvážit optimalizaci v oblasti práce se zásobníkem, jelikož všechny funkce mají poměrně vysoké procento exkluzivního času, což lze vidět ve flat view. Navíc funkce `rpn_pop_two_values()` má příliš velké procento exkluzivního času na to, co skutečně provádí - to by samo o sobě napovídalo, že funkce pravděpodobně špatně využívá cache nebo obsahuje jinou neoptimální strukturu. Nicméně vzhledem k časům ostatních zásobníkových funkcí se dá usoudit, že problém bude v samotné implementaci zásobníku nebo jeho nesprávném využívání ve funkci vyhodnocující RPN zásobník.

Vzhledem ke znalosti kódu lze problém se zásobníkem svést na špatnou lokalitu paměti, která vede k velkému počtu cache-miss při provádění operací nad zásobníkem. Jednotlivé prvky vkládané do zásobníku totiž velmi pravděpodobně leží daleko od sebe vzhledem ke způsobu, jakým je zásobník sestavován. Funkce pro vyhodnocení RPN zásobníku je neefektivně implementovaná hlavně kvůli skutečnosti, že provádí příliš zbytečných alokací a dealokací paměti.

9.2 Rozsáhlá aplikace a perf

Druhým programem je projekt poměrně rozsáhlý. Jedná se o emulátor serverové strany MMORPG² hry World of Warcraft. Taková aplikace musí v

²massively multiplayer online role-playing game, online hra na hrdiny

reálném čase komunikovat po síti se stovkami až tisíci klienty, vyhodnocovat akce spojené s herní aktivitou, přeposílat vyhodnocené akce okolí hráčů, starat se o „umělou inteligenci“ veškerých postav ve hře, které jsou ovládány serverem a spousty dalších věcí. V momentě, kdy je připojeno více než cca 50 hráčů by již nebylo možné v žádném případě použít nástroj *gprof*, jelikož generuje takovou zátěž v podobě instrumentovaných volání navíc, že se hra stává v podstatě nehratelnou. Proto je pro takto zatížený systém vhodné použít profiler *perf*.

Zdrojové kódy jsou k dispozici na adrese:

<https://github.com/TrinityCoreArchive/TrinityCoreOld/>

V této podkapitole budou prezentovány výstupy nasbírané z emulátoru běžícího v rámci projektu iCe Online (<http://ice-wow.eu>), kde je provozována obdoba výše uvedeného emulátoru včetně potřebného zatížení.

Lze porozovat hned několik typických věcí, ve kterých se výstup mírně liší u nástroje *gprof* a *perf*. V první řadě jde samozřejmě o schopnost pracovat s plnou hierarchií volání, tedy možnost generovat strom volání a flame graph. Dalším poměrně významným bodem je přítomnost „falešných“ hran v call-graph pohledu (obrázek 9.8). Ty vznikají pravděpodobně kvůli optimalizacím, které se dějí, ač byl zkoumaný program kompilován s přepínačem `-fno-omit-frame-pointer`³. Například metoda `Player::Update()` je volána výhradně z metody `Map::Update()`, ale přesto existují hrany z jiných metod. Tyto hrany mají často podstatně menší počet vzorků, ale nelze je jednoduše identifikovat, a tedy odstranit. Nástroj by musel umět analyzovat i zdrojový kód, a to v současné době není podporováno.

Dále si lze všimnout na stejném pohledu i toho, že jsou přítomny dvě oddělené komponenty grafu. To je důsledkem filtrování, jelikož je nastaven filtr na inkluzivní počet vzorků, kdy je zobrazen pouze ten uzel, který má procentuální počet inkluzivních vzorků větší než 15%. Metoda `Unit::CastSpell()` je volána z mnoha míst v celé hierarchii volání, takže její inkluzivní počet vzorků byl rozložen mezi všechny volající funkce, což mělo za důsledek to, že ve výsledku žádná z nich nevyhovovala filtru.

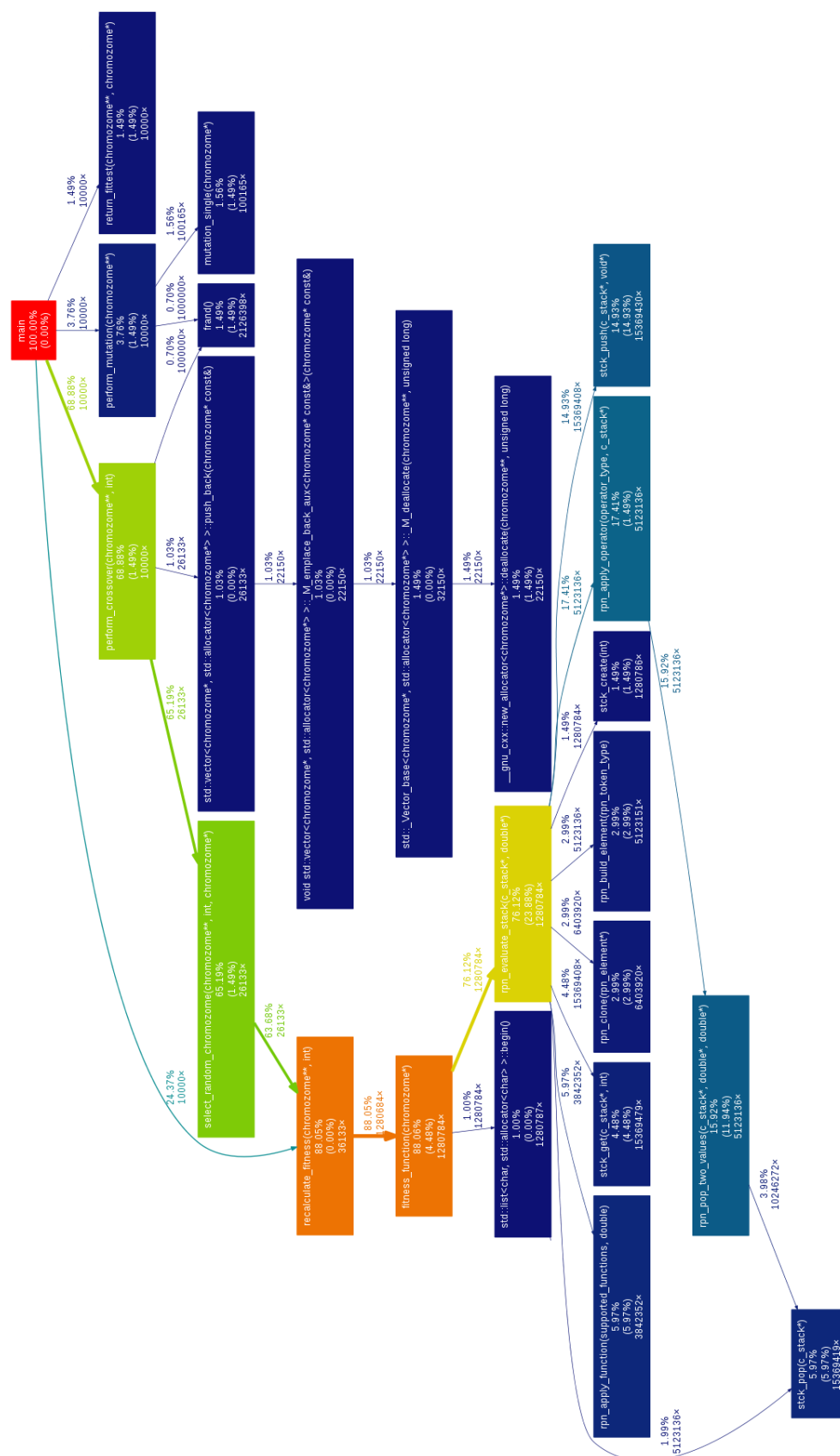
Na zbylých pohledech lze pozorovat vcelku očekávané skutečnosti. Flat view (obrázek 9.6) má nyní trochu jinou distribuci časů, ale to jen díky tomu, že byl analyzován neměřitelně větší projekt. Nejvíce exkluzivních vzorků se nachází v metodě `Map::Visit()`, což je poměrně očekávaná skutečnost

³přepínač kompilátoru *gcc* používaný pro zamezení optimalizace call stacku

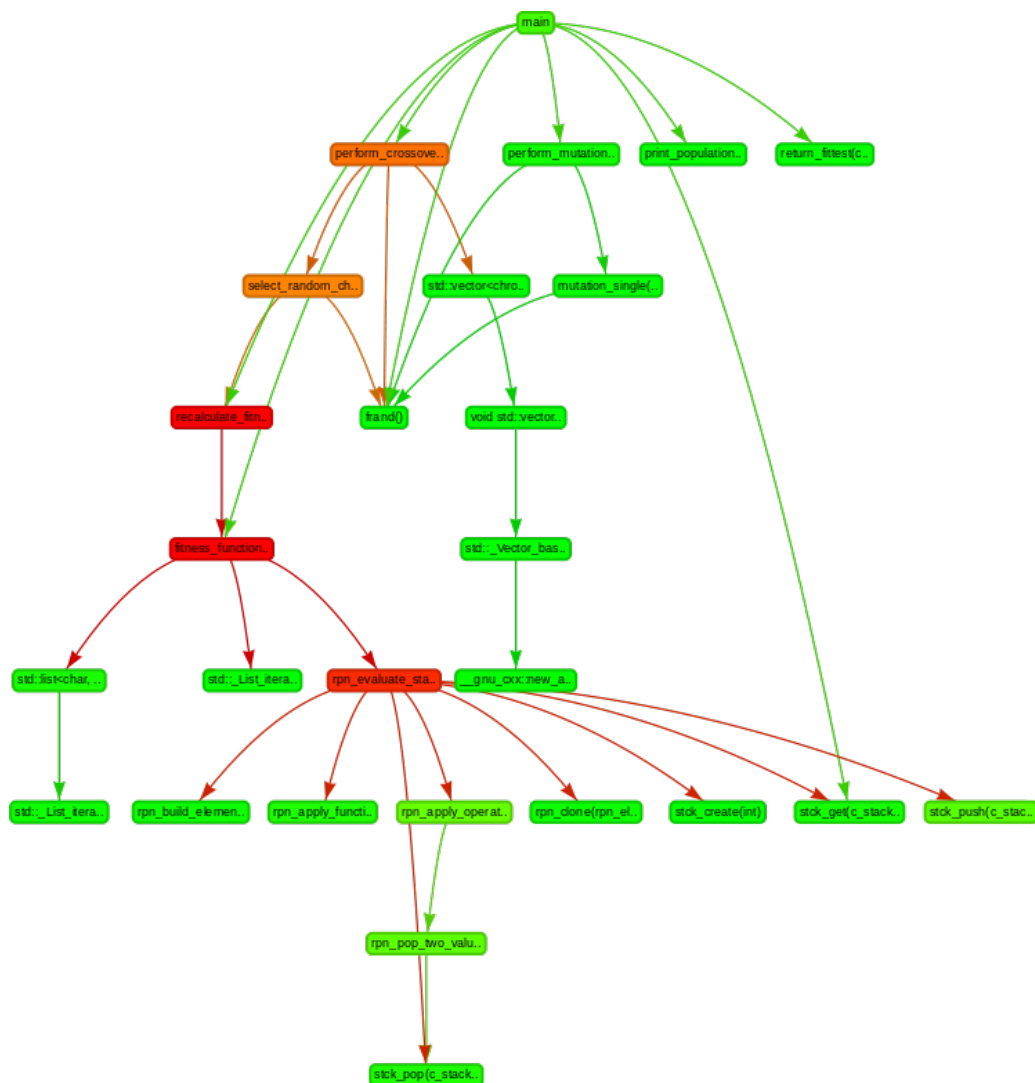
vzhledem k tomu, že tato metoda se stará o průchod jednotlivých polí mapy a zavolání příslušných updaterů. Zároveň v ní lze očekávat největší počet výskytů cache-miss, jelikož její implementace sestává z podstaty věci z nelineárního přístupu k paměti.

Nejvíce průkazný je zde pohled zachovávající hierarchii, zde call tree nebo flame graph. Jelikož se jedná o srovnatelné pohledy, jen trochu jinak postavené, lze pozorovat podobné skutečnosti. Výhodou call tree je možnost identifikovat funkci s velkým počtem exkluzivních vzorků v poměru k inkluzivním. To poměrně spolehlivě identifikuje výkonnostní problém. Z obrázku 9.7 lze vidět, že takový problém leží v metodě `Aura::GetCaster()`. Jde ale pouze o 0.25%, a proto tento problém není tak velký. Jelikož je výstup v podobě stromu volání velmi rozsáhlý, obtížně se v něm vyhledává takový problém bez znalosti kódu. Vhodnější je pak flame graph, který lze vidět na obrázku 9.9. Pro porovnání na obrázku 9.10 lze vidět výstup nástroje *FlameGraph*, který až na odlišné uspořádání a seskupení nenalezených symbolů vypadá velmi podobně.

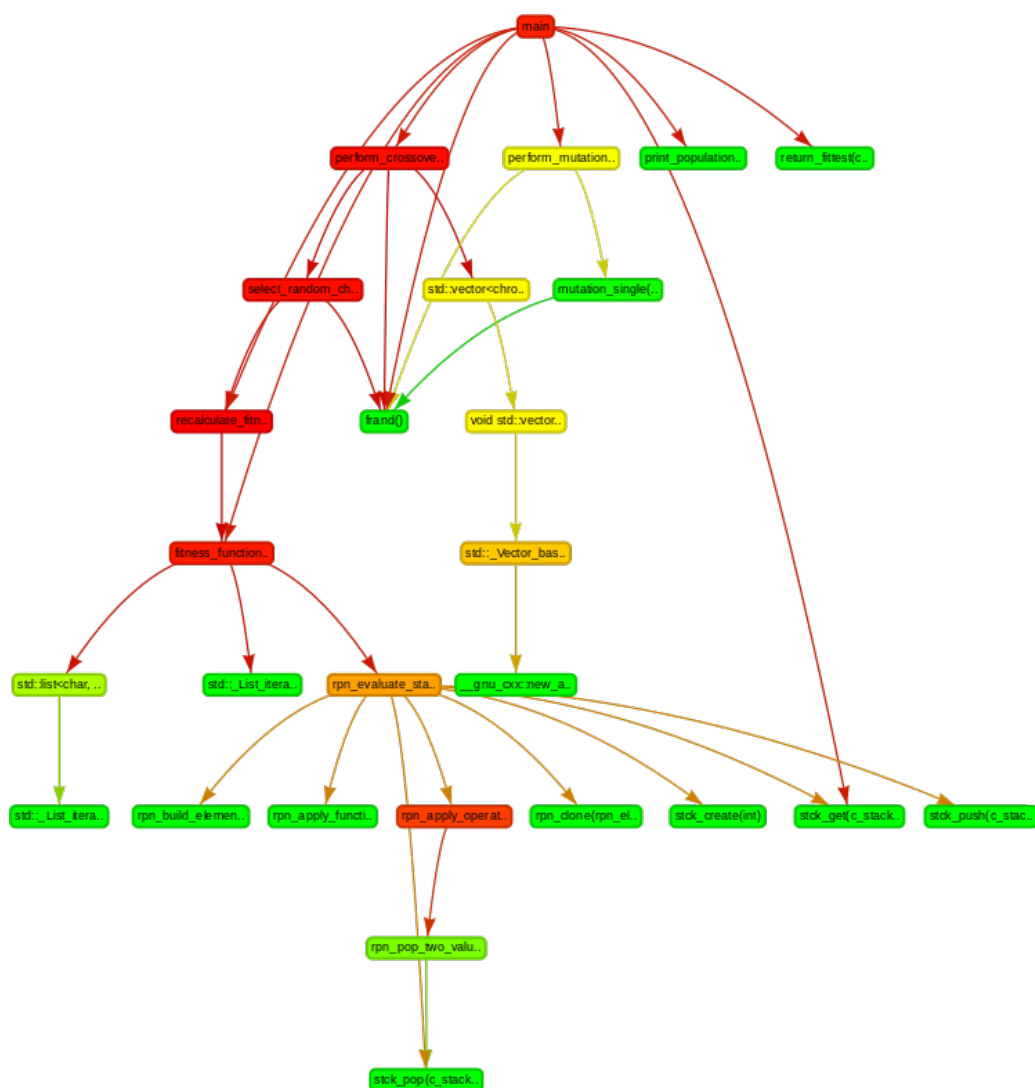
V tomto pohledu lze poměrně spolehlivě identifikovat nejnáročnější metody v celém programu, kterými jsou jistě všechny updatovací metody. Díky interaktivitě je možné proklikat do jednotlivých metod a zjistit, co činí výkonnostní potíže. Na grafu je mj. vidět i vysoký sloupec sestávající ze stále stejných symbolů. To je zvláštní případ metod z knihovny *ACE*, kde nejdříve dojde k poměrně rozsáhlému rekurzivnímu volání a až na konci celé rekurze se nachází nějaký výkonný kód.



Obrázek 9.3: Call-graph zpracovaný nástrojem *gprof2dot* z běhu genetického algoritmu



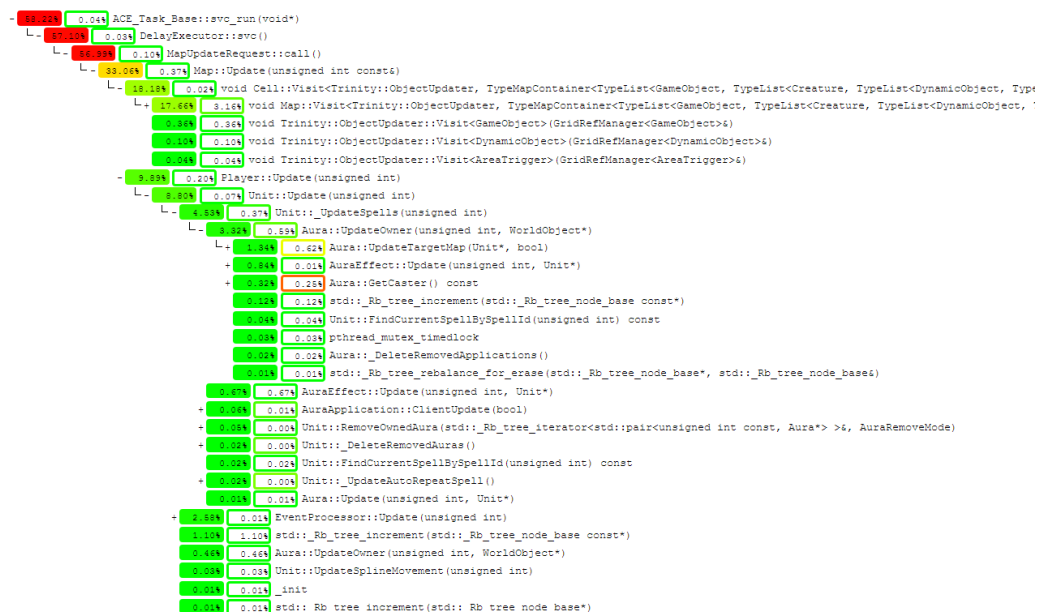
Obrázek 9.4: Call-graph zpracovaný nástrojem *PIVO* z běhu genetického algoritmu, uzly jsou obarveny podle inkluzivního času



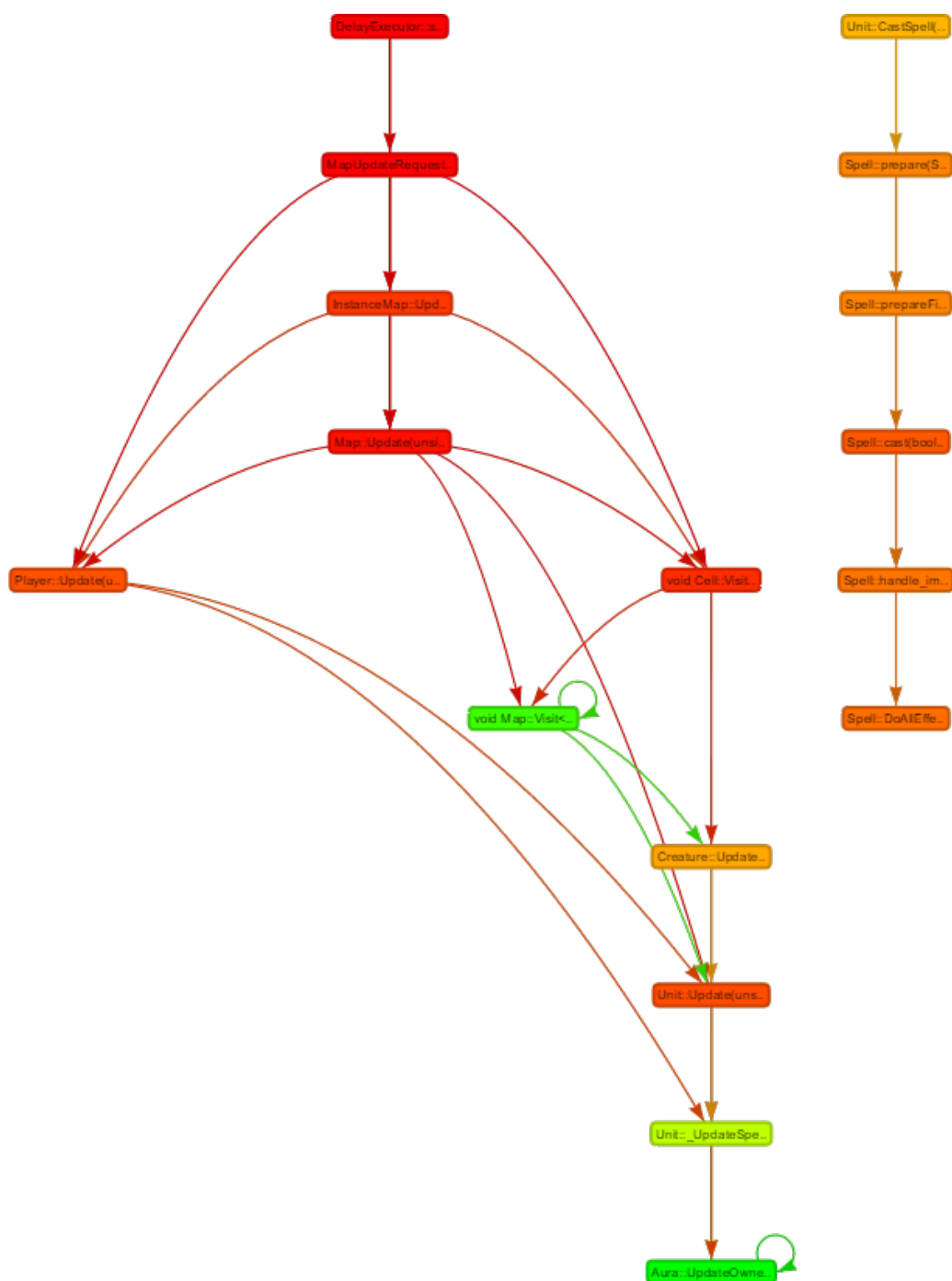
Obrázek 9.5: Call-graph zpracovaný nástrojem *PIVO* z běhu genetického algoritmu, uzly jsou obarveny podle poměru exkluzivního a inkluzivního času; lze snadno pozorovat *hot path*

Samples [%]	Samples []	Inclusive samples [%]	Inclusive samples []	Call count	Function name
3.94	9833	37.58	54671	65031	void Map::Visit<Trinity::ObjectUpdater, TypeMapContainer<TypeList<GameObj
2.09	5218	18.25	26542	32322	ACE_OS::gettimeofday()
1.80	4502	18.43	26815	31618	Aura::UpdateOwner(unsigned int, WorldObject*)
1.47	3669	10.25	14915	18752	AchievementMgr::UpdateAchievementCriteria(AchievementCriteriaTypes, unsig
1.31	3274	23.57	34285	37905	Unit::UpdateSpells(unsigned int)
1.28	3209	10.94	15915	19275	AuraEffect::Update(unsigned int, Unit*)
1.26	3150	1.45	2105	5317	Trinity::CreatureRelocationNotifier::Visit(GridRefManager<Creature>&) [cl
1.12	2798	2.09	3041	5886	void Trinity::ObjectUpdater::Visit<GameObject>(GridRefManager<GameObject>
1.05	2618	3.50	5096	7813	Aura::UpdateTargetMap(Unit*, bool)
0.94	2358	35.02	50950	53980	Creature::Update(unsigned int)
0.79	1965	7.82	11377	13594	Map::ProcessRelocationNotifies(unsigned int const&)
0.68	1711	54.56	79363	81927	Unit::Update(unsigned int)
0.67	1675	1.06	1540	3421	free
0.64	1602	12.27	17856	19599	Unit::ProcDamageAndSpellFor(bool, Unit*, unsigned int, unsigned int, Weap
0.62	1545	1.58	2292	3883	Trinity::MessageDistDeliverer::Visit(GridRefManager<Player>&)
0.57	1429	4.40	6396	7753	ACE_Dev_Poll_Reactor::Token_Guard::acquire_quietly(ACE_Time_Value*)
0.54	1356	6.78	9860	11306	MotionMaster::UpdateMotion(unsigned int)
0.48	1203	1.55	2250	3477	void BIH::intersectPoint<VMAP::LocationInfoCallback>(GSD::Vector3 const&
0.42	1050	1.35	1968	3049	void BIH::intersectRay<VMAP::MapRayCallback>(GSD::Ray const&, VMAP::MapRa
0.40	990	95.76	139299	141808	Map::Update(unsigned int const&)
0.37	912	1.74	2532	3485	Player::SendMessageToSet(WorldPacket*, bool)
0.36	889	1.37	1987	2904	void BIH::intersectPoint<VMAP::AreaInfoCallback>(GSD::Vector3 const&, VMA

Obrázek 9.6: Flat view (výřez) z běhu emulátoru MMORPG herního serveru



Obrázek 9.7: Call tree (výřez) z běhu emulátoru MMORPG herního serveru



Obrázek 9.8: Call-graph z běhu emulátoru MMORPG herního serveru; výstup je velmi redukován za pomoci filtru

10 Závěr

Cílem této práce bylo analyzovat dostupné profilovací nástroje, způsoby vizualizace dat jimi nasbíraných, navrhnout a implementovat takový nástroj, který bude umět načíst formáty různých profilerů, zpracovat je a vytvořit pohled, který nebude závislý na platformě.

TODO: DO

Literatura

- [1] BANDYOPADHYAY, S. *A Study on Performance Monitoring Counters in x86-Architecture* [online]. Indian Statistical Institute, 2004. [cit. 20.11.2015]. Dostupné z: <http://www.cise.ufl.edu/~sb3/files/pmc.pdf>.
- [2] FENLASON, J. – STALLMAN, R. *GNU gprof - the GNU profiler* [online]. GNU, 1998. [cit. 10.12.2015]. Dostupné z: ftp://ftp.gnu.org/old-gnu/Manuals/gprof-2.9.1/html_chapter/gprof_9.html.
- [3] FÄSSLER, U. – NOWAK, A. *Perf file format* [online]. Cern openlab, 2011. [cit. 25.11.2015]. Dostupné z: https://openlab-mu-internal.web.cern.ch/openlab-mu-internal/03_Documents/3_Technical_Documents/Technical_Reports/2011/Urs_Fassler_report.pdf.
- [4] GOURIOU, E. – MOSELEY, T. – BRUIJN, W. *Tutorial - Perf Wiki* [online]. Perf Wiki, 2011. [cit. 8.11.2015]. Dostupné z: <https://perf.wiki.kernel.org/index.php/Tutorial>.
- [5] *GNU General Public License* [online]. Free Software Foundation, 2007. Dostupné z: <http://www.gnu.org/licenses/gpl-3.0.txt>.
- [6] MARKWARDT, U. – LIEBER, M. *Cache Profiling with Callgrind* [online]. Technische Universität Dresden, 2009. [cit. 6.12.2015]. Dostupné z: http://wwwpub.zih.tu-dresden.de/~mlieber/practical_performance/04_callgrind.pdf.
- [7] *Profiling a Program: Where Does It Spend Its Time?* [online]. Free Software Foundation, Inc., 2014. [cit. 6.12.2015]. Dostupné z: <https://sourceware.org/binutils/docs/gprof/Implementation.html>.
- [8] *Valgrind: Supported platforms* [online]. Valgrind Developers, 2015. [cit. 6.12.2015]. Dostupné z: <http://valgrind.org/info/platforms.html>.
- [9] *Valgrind: Callgrind Format Specification* [online]. Valgrind Developers, 2015. [cit. 6.12.2015]. Dostupné z: <http://valgrind.org/docs/manual/cl-format.html>.

Seznam zkratek

IRQ	Interrupt Request - vnější hardwarové přerušení
HPC	Hardware Performance Counters - hardwarové výkonnostní čítače
SSD	Solid-state drive - disková jednotka založená nejčastěji na nevolatilní flash paměti
NMI	Non-maskable interrupt - nemaskovatelné přerušení
OS	Operační systém
CPU	Central processing unit - hlavní výpočetní jednotka počítače
PC	Program counter - instrukční čítač (často registr CPU) v rámci programu, obsahuje offset instrukce k provedení
IP	Instruction pointer - jiný název pro program counter (PC), často spojovaný navíc s nějakým segmentovým registrem, typicky kódovým (CS:IP)
CSV	Comma-separated values - formát souboru s buňkami oddělenými specifickým znakem (čárka, středník, aj.)
HTML	HyperText Markup Language - značkovací jazyk používaný pro webové stránky
CSS	Cascading Style Sheets - kaskádové styly používané pro webové stránky
PIVO	Profiler-Independent Visual Output - název nástroje vyvíjeného v rámci této práce