

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 2. 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 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 last chapter 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.

## **Abstrakt**

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

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 poslední kapitole 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.

## 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

<b>1</b>	<b>Úvod</b>	<b>7</b>
<b>2</b>	<b>Profiling</b>	<b>8</b>
2.1	Výkon . . . . .	8
2.2	Bottleneck . . . . .	8
2.2.1	Typické řešení . . . . .	9
2.3	Kritické optimalizace . . . . .	10
2.4	Metody sběru dat . . . . .	13
2.4.1	Vzorkování . . . . .	13
2.4.2	Instrumentace . . . . .	14
2.4.3	Interpretace . . . . .	14
2.4.4	Událostní profiling . . . . .	14
<b>3</b>	<b>Analýza dostupných nástrojů</b>	<b>15</b>
3.1	perf . . . . .	15
3.2	gprof . . . . .	16
3.3	OProfile . . . . .	18
3.4	Callgrind a Cachegrind . . . . .	20
3.5	DTrace . . . . .	21
3.6	Visual Studio Performance Profiling . . . . .	22
3.7	Very Sleepy . . . . .	24
3.8	RotateRight Zoom . . . . .	25
3.9	Shrnutí . . . . .	25
<b>4</b>	<b>Analýza vizualizačních technik</b>	<b>28</b>
4.1	Flat view . . . . .	28
4.2	Hierarchical view . . . . .	29
4.3	Graph view . . . . .	29
4.4	Object-method view . . . . .	31
4.5	Flame graph . . . . .	31
4.5.1	Detailed graph . . . . .	32
4.6	Heat maps . . . . .	33
4.7	Shrnutí . . . . .	35

---

<b>5</b>	<b>Dostupné nástroje pro vizualizaci</b>	<b>36</b>
5.1	Visual Studio Performance Profiling . . . . .	36
5.2	KCachegrind . . . . .	36
5.3	KProf . . . . .	37
5.4	GProf2dot . . . . .	37
5.5	RotateRight Zoom . . . . .	37
5.6	Shrnutí . . . . .	38
<b>6</b>	<b>Návrh nástroje</b>	<b>39</b>
6.1	Základ systému . . . . .	39
6.1.1	Jádro . . . . .	39
6.1.2	Analyzér . . . . .	40
6.2	Vstupní moduly . . . . .	41
6.3	Výstupní moduly . . . . .	41
6.4	Technologie . . . . .	41
6.5	Licence . . . . .	42
6.6	Shrnutí . . . . .	42
<b>7</b>	<b>Závěr</b>	<b>43</b>
	<b>Literatura</b>	<b>44</b>
	<b>Seznam zkratk</b>	<b>45</b>

# 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 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-miss<sup>1</sup> 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 naskýtá 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 pipelining<sup>2</sup>, 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.

---

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

<sup>2</sup>způsob 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[5].

Přístup s HPC používá hardwarové přerušení NMI. To je generováno při každém přetečení některého z čítačů, jehož hodnotu sledujeme[5]. 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 zkompilevaného binárního spustitelného souboru.

### 2.4.3 Interpretace

Podstatně rozdílným přístupem je pojetí zkompilevaného strojového kódu jako intermediate kód<sup>3</sup>, který je interpretován nad vlastním virtuálním strojem[7]. 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.

---

<sup>3</sup>meziformát 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[4].

## 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 diskrétních 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[3]. 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].

## 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 inlinovány<sup>1</sup> v rámci optimalizací).

Nevýhod je ale podstatně více. První znatelnou nevýhodou je zpomalení 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 zkompileovaný 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

<sup>1</sup>nahrazení volání funkce přímo blokem příkazů, který funkce vykonává

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 výkonostních 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 profilovým 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 ná-

stroj *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 využívat 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 vyústí 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 převeditelný 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 balíčků 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 přehledně. 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 profilování 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

Analýzované 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 headless<sup>2</sup> 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 zabývat 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.

---

<sup>2</sup>systémy 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_traits<char>::basic_string_view<char, std::char_traits<char>, std::allocator<char>>>	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_traits<char>, std::allocator<char>>>	2	0	0,87	0,00
std::vector<chromosome *, std::allocator<chromosome *>>	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.

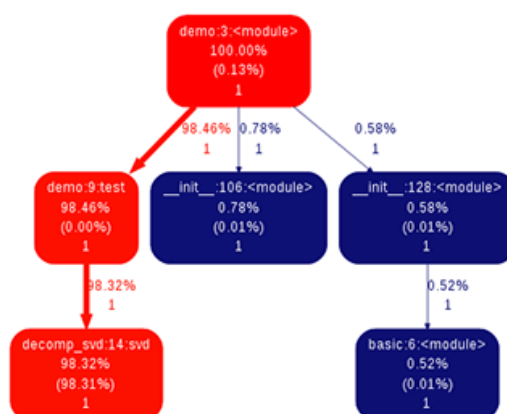
Tento pohled výrazně rozšiřuje množinu identifikovatelných problémů. Příkladem problému, který odhalí tento pohled navíc oproti *flat view*, je například ten, kdy máme poměrně náročnou funkci, která je volaná z mnoha míst v celém programu. Pokud je ale někde chybně volána častěji, než je nezbytně nutné, 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

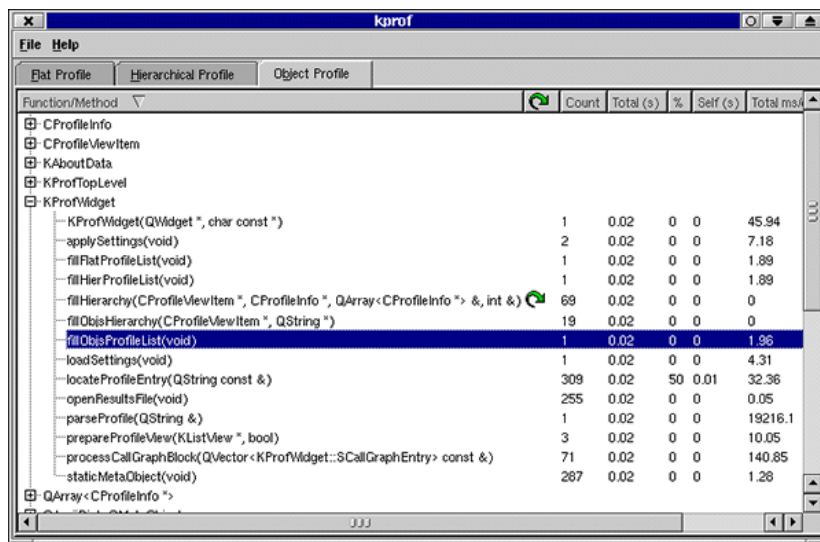
Obrázek 4.2: Hierarchical view generovaný nástrojem VSPerf

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
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
<b>fillObjProfileList(void)</b>	<b>1</b>	<b>0.02</b>	<b>0</b>	<b>0</b>	<b>1.96</b>
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

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

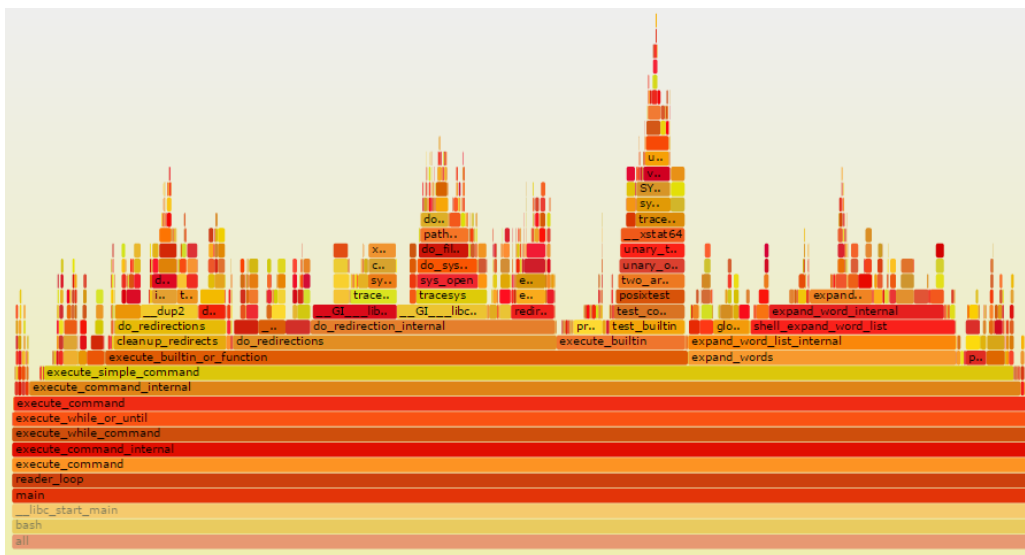
Tento pohled lze snadno generovat z pohledu hierarchického. Ve své podstatě jde pouze o expandovaný hierarchický pohled, který je uspořádán odspoda (kořen) nahoru (listy). Značnou výhodou je škálování jednotlivých položek podle procentuálních hodnot zkoumané veličiny - tráveného času (zde inkuzivního). 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.

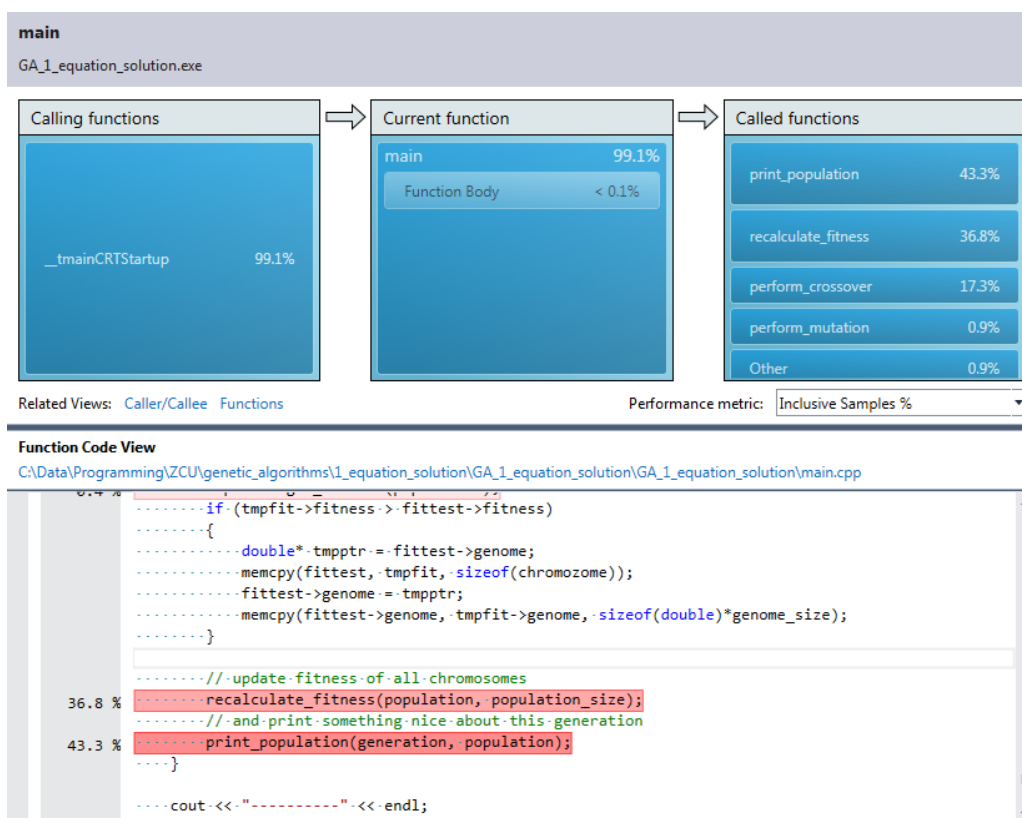
Takový graf ale vzhledem ke své obsáhlosti musí být buď dostatečně velký, nebo interaktivní. Interaktivitu si můžeme představit například možností rozkliknout funkci, která je pro nás zajímavá - poté se změní poměry šířek, vybraná funkce tvoří výchozí bod se 100% šířkou a všechny ostatní jí volané funkce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

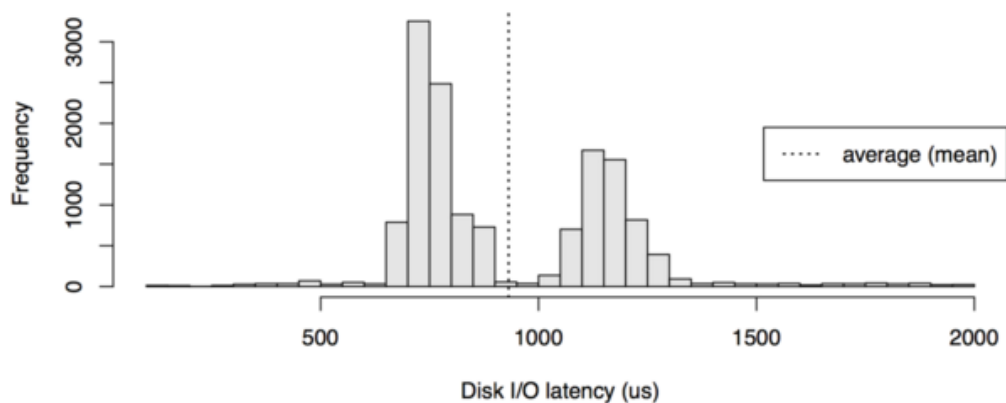


Obrázek 4.6: Detailed view generovaný nástrojem VSPerf

## 4.6 Heat maps

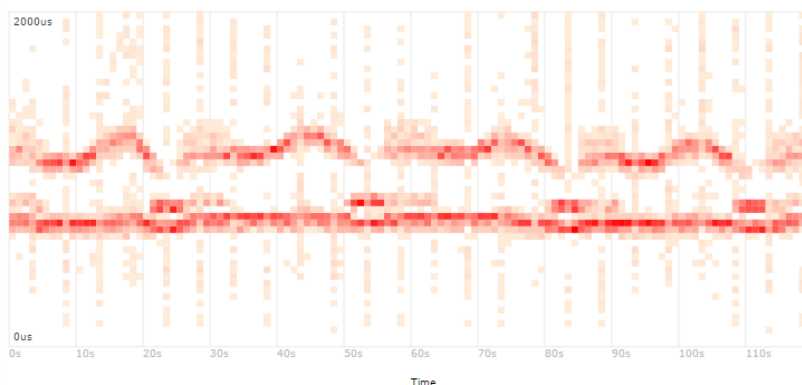
Heat mapy nejsou v současné době příliš rozšířenou technikou vizualizace profilových dat, ale je možné je pro určité body zájmu aplikovat. Oproti předchozím způsobům dovozuje zobrazovat do určité míry i trojrozměrná data. Vodorovná osa znázorňuje tok času, svislá osa zkoumané body zájmu, v případě profilingu 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 platform. 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

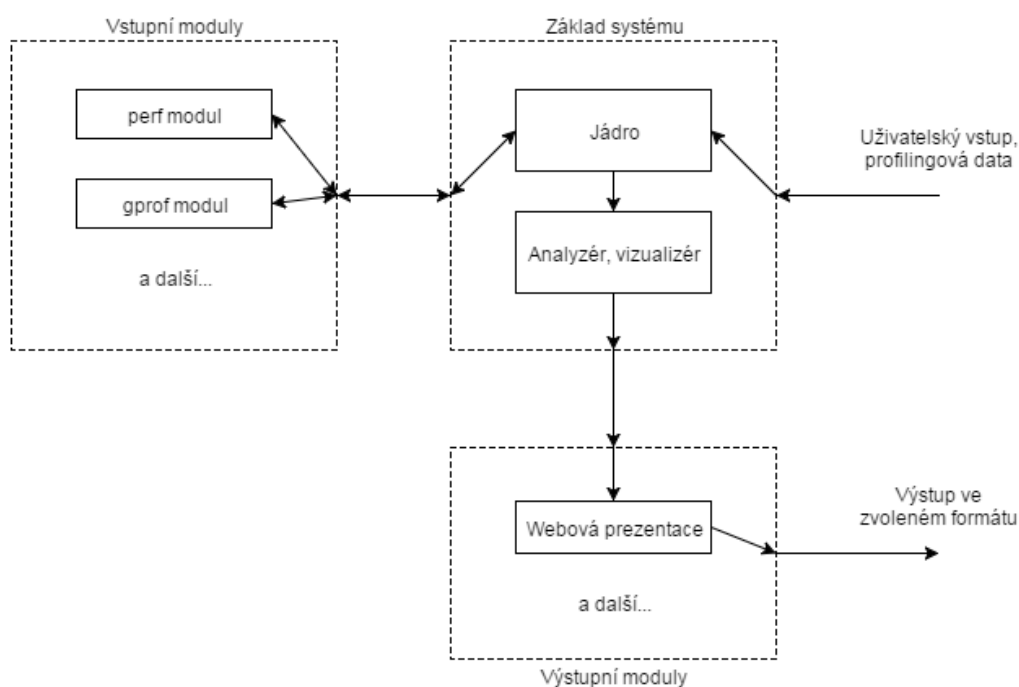
Tato skupina modulů bude představovat spustitelný program. Moduly základu budou tedy pevně svázány po čase kompilace, tedy nebudou fyzicky oddělitelné. To z důvodu, že je nutné mezi nimi zajistit maximální kompatibilitu, kterou lze do jisté míry zaručit při sestavení.

#### 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 Analyzér

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.

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

Použití dalších technologií je závislé na implementačních detailech. V rámci navazující bakalářské práce bude například použita základní škála frontových 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 platform 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 [6]. 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.

Vývoj tohoto nástroje se předpokládá v rámci bakalářské práce.

## 7 Závěr

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

Problematika profilingu je poměrně rozsáhlá, a to díky tomu, že výkonnostní analýzu bylo žádoucí provádět již od počátků moderních systémů. Za tu dobu bylo vyvinuto velké množství různých profilerů a byly ustáleny standardní pohledy z profilovacích dat generované. Analýza obsahuje nástroje, které se řadí mezi obecně nejpoužívanější. Z nich byly vybrány dva, které budou v rámci bakalářské práce implementovány do výsledného nástroje. Také je obsažena analýza nejčastěji používaných vizualizačních technik, ze kterých byly vybrány tři pro implementaci, a jedna k analýze možností pro vizualizaci takových dat.

V poslední části byl navržen nástroj, který bude realizován v rámci bakalářské práce. Výsledkem bude modulární přenositelná aplikace, která bude implementovat vstupní moduly pro načtení výstupních formátů nástrojů *perf* a *gprof*, a výstupní modul pro vizualizaci pomocí standardních webových technologií.

Osobně pro mě práce byla přínosná hlavně díky znalostem, které jsem v průběhu psaní získal. Poskytla mi přehled o existujících profilech, způsobech získávání dat a také o vizualizačních technikách. V minulosti jsem se s profilinkem setkal pouze v omezené míře, a tak tato práce zahrnovala i velké množství studia.

# Literatura

- [1] BANDYOPADHYAY, S. *A Study on Performance Monitoring Counters in x86-Architecture*. Indian Statistical Institute, 2004. Dostupné z: <http://www.cise.ufl.edu/~sb3/files/pmc.pdf>.
- [2] FENLASON, J. – STALLMAN, R. *GNU gprof - the GNU profiler*. GNU, 1998. Dostupné z: [ftp://ftp.gnu.org/old-gnu/Manuals/gprof-2.9.1/html\\_chapter/gprof\\_9.html](ftp://ftp.gnu.org/old-gnu/Manuals/gprof-2.9.1/html_chapter/gprof_9.html).
- [3] *Profiling a Program: Where Does It Spend Its Time?* Free Software Foundation, Inc., 2014. Dostupné z: <https://sourceware.org/binutils/docs/gprof/Implementation.html>.
- [4] FÄSSLER, U. – NOWAK, A. *Perf file format*. Cern openlab, 2011. Dostupné z: [https://openlab-mu-internal.web.cern.ch/openlab-mu-internal/03\\_Documents/3\\_Technical\\_Documents/Technical\\_Reports/2011/Urs\\_Fassler\\_report.pdf](https://openlab-mu-internal.web.cern.ch/openlab-mu-internal/03_Documents/3_Technical_Documents/Technical_Reports/2011/Urs_Fassler_report.pdf).
- [5] GOURIOU, E. – MOSELEY, T. – BRUIJN, W. *Tutorial - Perf Wiki*. Perf Wiki, 2011. Dostupné z: <https://perf.wiki.kernel.org/index.php/Tutorial>.
- [6] GPL. GNU General Public License, 2007. Dostupné z: <http://www.gnu.org/licenses/gpl-3.0.txt>.
- [7] MARKWARDT, U. – LIEBER, M. *Cache Profiling with Callgrind*. Technische Universität Dresden, 2009. Dostupné z: [http://wwwpub.zih.tu-dresden.de/~mlieber/practical\\_performance/04\\_callgrind.pdf](http://wwwpub.zih.tu-dresden.de/~mlieber/practical_performance/04_callgrind.pdf).
- [8] *Valgrind: Supported platforms*. Valgrind Developers, 2015. Dostupné z: <http://valgrind.org/info/platforms.html>.
- [9] *Valgrind: Callgrind Format Specification*. Valgrind Developers, 2015. Dostupné z: <http://valgrind.org/docs/manual/cl-format.html>.

# Seznam zkratek

<b>IRQ</b>	Interrupt Request - vnější hardwarové přerušení
<b>HPC</b>	Hardware Performance Counters - hardwarové výkonnostní čítače
<b>SSD</b>	Solid-state drive - disková jednotka založená nejčastěji na nevolatilní flash paměti
<b>NMI</b>	Non-maskable interrupt - nemaskovatelné přerušení
<b>OS</b>	Operační systém
<b>CPU</b>	Central processing unit - hlavní výpočetní jednotka počítače
<b>PC</b>	Program counter - instrukční čítač v rámci programu, obsahuje offset instrukce k provedení
<b>CSV</b>	Comma-separated values - formát souboru s buňkami oddělenými specifickým znakem (čárka, středník, aj.)
<b>HTML</b>	HyperText Markup Language - značkovací jazyk používaný pro webové stránky
<b>CSS</b>	Cascading Style Sheets - kaskádové styly používané pro webové stránky