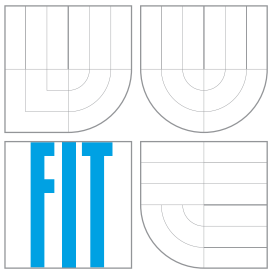
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

# NÁVRH A IMPLEMENTACE PROFILERU PRO APLIKAČNĚ SPECIFICKÉ PROCESORY

DESIGN AND IMPLEMENTATION OF A PROFILER FOR ASIPS

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

PAVEL RICHTARIK

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. MARCELA ZACHARIÁŠOVÁ, Ph.D.

BRNO 2016

# Abstrakt

Hlavným cieľom tejto práce je analyzovať možnosti profilovania aplikačne špecifických procesorov, preskúmať bežne používané profilovacie techniky a využiť získané informácie pri návrhu a implementácii nového profilovacieho nástroja použiteľného pri vývoji a optimalizácii procesorov. Táto bakalárska práca prezentuje požiadavky na nový profiler a popisuje jeho hlavné časti z pohľadu procesu návrhu a implementácie.



# Abstract

The major objective of this work is to analyse possibilities of profiling application specific instruction-set processors, to explore some common profiling techniques and to use the collected information to design and implement a new profiling tool suitable for utilization in the processors development and optimization. This bachelor thesis presents requirements on the new profiler and describes its key parts from the design and the implementation perspective.



# Kľúčové slová

profilovanie, ASIP, optimalizácia, simulácia

# Keywords

profiling, ASIP, optimization, simulation


# Citácia

RICHTARIK, Pavel. *Návrh a implementace profileru pro aplikačně specifické procesory.* Brno, 2016. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Zachariášová Marcela.

# Návrh a implementace profileru pro aplikačně specifické procesory

## Prehlásenie

Prehlasujem, že som túto bakalársku prácu vypracoval samostatne pod vedením pani Ing. Marcely Zachariášovej, Ph.D. Ďalšie informácie mi poskytol pán Ing. Zdeněk Přikryl, Ph.D. Uviedol som všetky literárne pramene a publikácie, z ktorých som čerpal.

.....
Pavel Richtarik
17. mája 2016

## Poďakovanie

Rád by som poďakoval pani Ing. Marcele Zachariášovej za jej vedenie a pomoc pri písaní tejto práce. Tiež by som chcel poďakovať pánovi Ing. Zdeňkovi Přikrylovi, Ph.D. za jeho odbornú pomoc pri návrhu a implementácii popisovaného profileru.

# Obsah

# Kapitola 1

# Úvod

Hlavnou úlohou aplikačne špecifických procesorov je urýchlenie náročných a často vykonávaných výpočtov v systéme, pričom pri ich návrhu je snaha o čo najväčšiu minimalizáciu rozmerov a spotreby výsledného produktu. Na rozdiel od procesorov pre všeobecné použitie (*general-purpose processors*) sú aplikačne špecifické procesory určené pre konkrétnu triedu aplikácií, vďaka čomu je možné efektívnejšie využiť možnosti hardvéru a dosiahnuť lepšie výsledky pri nižšej frekvencii a tým pádom i spotrebe [9].

Keďže kľúčovou vlastnosťou aplikačne špecifických procesorov je efektívnosť, nutnosťou je možnosť merať a vyhodnocovať dosiahnuté výsledky. Získané poznatky tvoria spätnú väzbu a sú fundamentálnym predpokladom pre ďalšiu optimalizáciu.

Základným prostriedkom pri získavaní informácií o správaní architektúry (z hľadiska softvéru aj hardvéru) je profilovanie. Ide o dynamickú analýzu, teda sledovanie systému v čase, počas jeho fungovania. Nástroj, ktorý profilovanie umožňuje sa nazýva profiler [7].

Cieľom tejto práce bola analýza možností profilovania aplikačne špecifických procesorov a vytvorenie nového profilera pre firmu Codasip Ltd., ktorý by bol integrovateľný do existujúceho vývojového prostredia. Náhrada starého profilera bola súčasťou viacerých veľkých zmien v architektúre sady nástrojov firmy Codasip. Zároveň bolo potrebné rozšíriť jeho funkcionalitu o sledovanie viacerých štatistík a ich lepšiu prezentáciu používateľovi.

Dve kapitoly na začiatku práce predstavujú teoretický úvod do problematiky aplikačne špecifických procesorov a profilovania vo všeobecnosti. Druhá kapitola zhrňuje vlastnosti aplikačne špecifických procesorov a popisuje základné fázy ich návrhu. Tretia kapitola uvádza a porovnáva 3 najčastejšie spôsoby profilovania a vyčleňuje dva samostatné procesy - zber dát o behu programu a ich následné spracovanie.

Štvrtá kapitola je venovaná existujúcim profilovacím nástrojom (konkrétne *OProfile* a *gprof*), z ktorých sa vychádzalo pri návrhu nového profileru a demonštruje na nich praktické využitie rôznych prístupov. Taktiež uvádza ukážky ich výstupov, ktoré čiastočne poslúžili ako podklad pre výstup nového profileru.

Piatu kapitolu tvorí súhrn požadovaných kľúčových vlastností nového profileru a popis návrhu jeho realizácie a riešenia konkrétnych problémov. V rámci popisu architektúry je uvedený náčrt použitých dátových štruktúr a algoritmov. Časť kapitoly venovaná analýze zásobníka pri spracovaní profilovacích výsledkov obsahuje rovnice popisujúce vzťahy medzi volanými funkciami ktoré boli v algoritme použité.

V kapitole 6 sú uvedené ukážky výstupov výsledného profilera. Uvedená je ukážka hlavného grafického HTML výstupu a pre porovnanie je zahrnutá aj časť výstupu v textovom formáte. Okrem toho je ukázaný aj spôsob zobrazenia anotovaného zdrojového kódu. Časť kapitoly je venovaná krátkej demonštrácii použitia profileru na teoretickom príklade.

Keďže profilovanie procesora sa deje v rámci simulácie, je dôležité poznať veľkosť réžie profileru a mieru spomalenia simulácie v dôsledku jeho použitia. Výkonnostné testovanie prebehlo na troch vybraných procesorových modeloch a výsledky meraní sú zhrnuté v siedmej kapitole. Poznatky o vplyve na rýchlosť simulácie boli využité počas vývoja a boli smerodajné pri nasadzovaní rôznych optimalizácií.

Posledná kapitola prináša niekoľko návrhov na vylepšenia a možné rozšírenia v budúcnosti.

# Kapitola 2

# Aplikačne špecifické procesory

Aplikačne špecifický procesor (*ASIP - Application Specific Instruction-set Processor*) je procesor zameraný na konkrétne špecifické použitie. Inštrukčná sada takého procesoru je špeciálne navrhnutá na urýchlenie najpoužívanejších funkcií a jeho architektúra sa snaží o implementáciu týchto inštrukcií s čo najnižšími nákladmi na hardware [9].

Aplikačne špecifické procesory predstavujú kompromis medzi zákazníckymi integrovanými obvodmi (*ASIC - Application Specific Integrated Circuit*), ktorých vývoj je drahý, a ktorých použitie je veľmi úzko špecifikované a procesormi pre všeobecné použitie (*General Purpose Processors*), ktoré často nespĺňajú konkrétne požiadavky na výkon alebo rozmery.

Obrázok 2.1 ukazuje flexibilitu použitia a výkon jednotlivých vstavaných technológií. Najväčšiu flexibilitu, tj. najširšie možnosti použitia poskytujú procesory pre všeobecné použitie (*General purpose microprocessor*), ich efektivita je však najnižšia. Naproti tomu, najvyššiu efektivitu vykazujú aplikačne špecifické integrované obvody s nemennými hardvérovými dátovými cestami (*Hardwired datapath*), ich použitie je však jednoúčelové. Aplikačne špecifické procesory (ASIP) sa nachádzajú v strede grafu a kombinujú výhody oboch technológií.

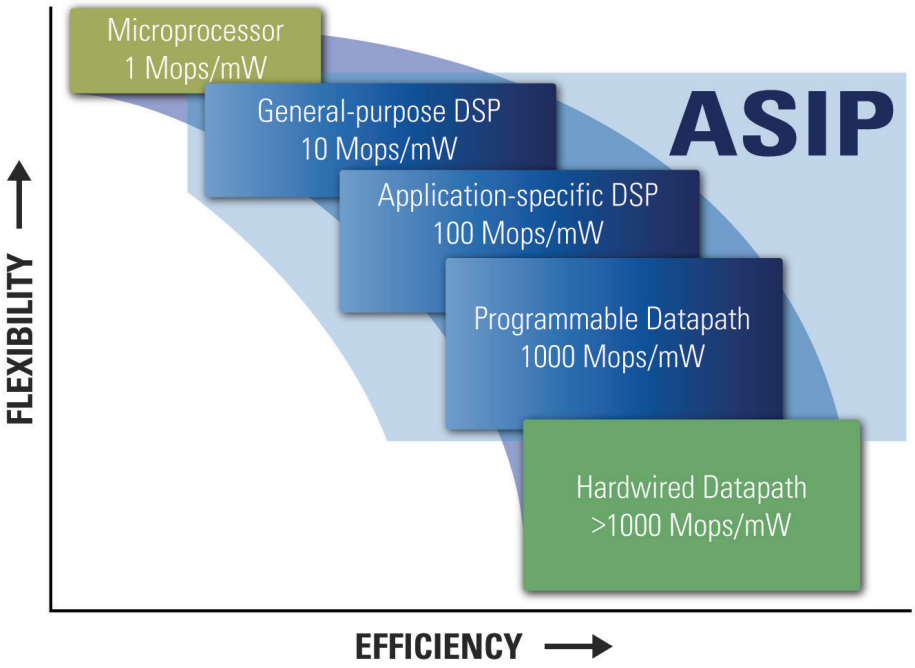
Obmedzením procesora na určitú triedu aplikácií vzniká priestor pre väčšiu optimalizáciu oproti všeobecne použiteľným procesorom. Pri tom však stále ostáva možnosť modifikácie softvéru pre konkrétnu úlohu, čo zvyšuje použiteľnosť procesora, vďaka čomu klesajú počiatočné náklady na vývoj (*NRE - Non-Reccuring Engineering*) pripadajúce na jeden kus [6].

Medzi typické úlohy pri návrhu aplikačne špecifických procesorov patrí výber typu architektúry procesora (*RISC, CISC, VLIW...*), návrh efektívnej inštrukčnej sady podporujúcej optimálnu mieru paralelizácie alebo návrh zreťazenej linky procesora s konkrétnymi stupňami (*pipeline*). Pri hotovom prototype modelu prichádza na rad testovanie, vyhodnocovanie správania procesora a optimalizácia na základe získaných dát.

### Typy procesorov podľa inštrukčnej sady

Výber typu inštrukčnej sady a jej návrh je jedným zo základných krokov pri návrhu procesora. Odvíjajú sa od neho možnosti zreťazeného spracovania inštrukcií alebo štruktúra dekodérov. Výber správneho typu inštrukčnej sady je kľúčový aj z hľadiska rozmerov, príkonu a výslednej ceny produktu.

Podľa zložitosti inštrukcií je možné rozdeliť procesory na RISC (*Reduced Instruction-set Processor*) a CISC (*Complex Instruction-set Processor*) [10]. CISC procesory sa vyznačujú komplexnou inštrukčnou sadou s množstvom rôznych inštrukcií, ktoré dokážu pra-




Obr. 2.1: Flexibilita a výkon vstavaných technológií [2]

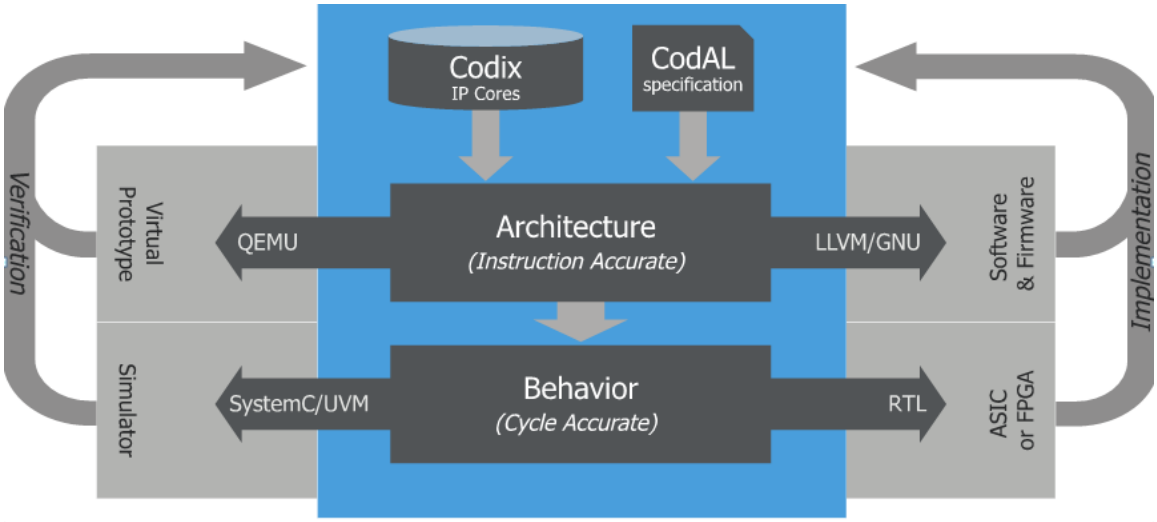
covať priamo s pamäťovými operandmi. Príkladom môže byť architektúra x86 od firmy Intel bežne využívaná v osobných počítačoch. Zatiaľ čo takáto architektúra je použiteľná v procesoroch pre všeobecné použitie, v špecifických aplikáciách a vstavaných systémoch sa uplatňujú skôr procesory s redukovanou inštrukčnou sadou (RISC). Tie ponúkajú naopak malú paletu inštrukcií, ktoré v zásade pracujú s registrovými operandmi. V rámci snahy o minimalizáciu pamäťových operácií obsahujú väčšinou veľké množstvo registrov. Dôležité je, že pri procesoroch typu RISC typicky platí, že v každom takte sa dokončí jedna inštrukcia, čo poskytuje širšie možnosti pre optimalizáciu zreťazeného spracovania.

Osobitnú kategóriu tvoria procesory typu VLIW (*Very Long Instruction Word*). Tieto procesory poskytujú možnosti paralelizácie programu na úrovni inštrukcií (*Instruction Level Paralelism*) - inštrukčné slovo obsahuje niekoľko zakódovaných inštrukcií, ktoré sa vykonajú naraz. Zhlukovanie inštrukcií do paralelne vykonateľných skupín má na starosti prekladač - počas vykonávania programu teda nevzniká na strane procesora žiadna réžia v súvislosti s časovaním. Tieto procesory sa využívajú pri spracovaní obrazu alebo počítačovom videní, kde má paralelizmus veľké využitie [4].

### Proces návrhu aplikačne špecifických procesorov

Na návrh aplikačne špecifických procesorov slúžia špecializované vývojové nástroje, napríklad *ASIP Designer* od spoločnosti Synopsys alebo *Codasip Studio* od firmy Codasip Ltd. Tieto nástroje poskytujú okrem prostredia pre tvorbu a ladenie softvéru tiež prostriedky pre efektívny návrh inštrukčnej sady a architektúry procesora. Podľa vytvoreného popisu architektúry v špecializovanom ADL[1] jazyku sú schopné vygenerovať podrobný RTL popis hardvéru na úrovni registrov (*Register - transfer level*) v HDL[2] jazyku, na základe ktorého

[1] *Architecture Description Language* - jazyk pre popis architektúry, napr. *LISA, nML, CodAL...*
[2] *Hardware Description Language* - jazyk pre popis hardvéru, napr. *VHDL, Verilog, SystemC...*




Obr. 2.2: Proces návrhu ASIP [3]

môže prebehnúť syntéza výsledného procesora [3].

Po vytvorení inštrukčnej sady a popisu procesora prebieha automatizované generovanie softvérových nástrojov špecifických pre nový procesor (prekladač, debugger, simulátor, profiler). Tie predstavujú spojenie medzi špecificky navrhnutým hardvérom a inštrukciami a štandardným programovacím jazykom, v ktorom je písaný softvér. Vygenerovaný simulátor umožňuje spustenie preloženého programu a sledovanie správania modelu procesora.

Codasip Studio použité v tejto práci umožňuje modelovať procesor na dvoch úrovniach - na úrovni inštrukcií (*IA - instruction accurate model*) a na úrovni cyklov (*CA - cycle accurate model*). V prvom prípade sa jedná o jednoduchší a menej podrobný popis procesora, ktorý zachytáva sémantiku jednotlivých inštrukcií, zatiaľčo popis na úrovni cyklov je podrobnejší, komplikovanejší a umožňuje detailnejšie simulovať procesy v hardvéri.

Pred výrobou a nasadením navrhnutého modelu prebieha verifikácia a validácia výsledkov. Proces verifikácie typicky zahŕňa porovnávanie abstraktného modelu (konceptu) s jeho konkrétnou realizáciou. V prípade vývojového cyklu v spoločnosti Codasip ide o porovnanie správania referenčného IA modelu, ktorý je jednoduchý na pochopenie, s podrobným CA modelom, respektíve s automaticky vygenerovaným RTL modelom v HDL jazyku. Keďže formálna verifikácia reálneho modelu procesora by bola extrémne náročná alebo nemožná, v praxi sa využíva funkčná verifikácia (*Functional Verification*), ktorej úlohou je zaručiť, že sa systém (procesor) správa podľa špecifikácie, a to za každých okolností. Najčastejšie používaný je simulačný prístup (*Simulation-Based Verification*), kedy sa na dvoch modeloch spustí rovnaký program a porovnajú sa výstupy simulácií, pričom je snaha, aby testovací program pokryl (a teda otestoval) čo najväčšiu množinu inštrukcií, a čo najväčší objem popisného kódu.

Proces návrhu aplikačne špecifických procesorov je znázornený na obrázku 2.2. Na začiatku procesu je výber jadra (*Codix IP Core*) a jeho modifikácia pre konkrétne účely - špecifikácia v jazyku CodAL. Na základe tohto popisu architektúry je vytvorený IA model, IA simulátor a prekladač. Po doplnení implementácie je možné vygenerovať CA model a CA simulátor, ktorého výsledky sú konfrontované s výsledkami referenčného IA simulátora. Z CA modelu je automaticky vygenerovaný RTL popis výsledného hardvéru, ktorý môže byť syntetizovaný.

# Kapitola 3

# Profilovanie

Pojem profilovanie (*profiling*) označuje v softvérovom inžinierstve proces dynamickej analýzy programu, čo zahŕňa zber dát o behu programu a ich následnú analýzu typicky za účelom optimalizácie. Výstupom profilovania je súbor štatistík, na základe ktorých je možné identifikovať kritické miesta v programe (najčastejšie vykonávané bloky kódu, najviac využívaná oblasť pamäte...), závislosť doby behu programu od vstupných parametrov alebo postupnosť a hierarchiu volaní jednotlivých funkcií [14].

V prípade profilovania aplikačne špecifických procesorov sú sledované aj ďalšie parametre súvisiace s architektúrou, napríklad efektívnosť vyrovnávacej pamäte alebo často vykonávané sekvencie inštrukcií. Na základe udalostí zreťazenej linky procesora je ďalej možné usudzovať vznik hazardov (*pipeline stall* - zdržanie zreťazenej linky) alebo vyhodnocovať úspešnosť prediktora skokov (*pipeline clear* - vyprázdnenie zreťazenej linky z dôvodu chybnej predikcie a rozpracovania nasledujúcej inštrukcie).

Podľa pravidla 90-10, 90% času behu programu tvorí vykonávanie len 10% kódu. Analýzou dát získaných profilovaním je možné identifikovať práve tie oblasti, ktorých úprava môže znamenať významné zvýšenie výkonu alebo zníženie spotreby. Preto je profilovanie základnou súčasťou a predpokladom optimalizácie či už softvéru alebo architektúry [14].

Pri návrhu profilera je dôležitý výber sledovaných hodnôt a spôsob ich agregácie a zobrazenia v zrozumiteľnej podobe. Samotná implementácia profilera musí byť dostatočne efektívna, aby dokázala výsledky poskytnúť v rozumnom čase.

Pri profilovaní klasických aplikácií v rámci operačného systému počítača nemá profiler kontrolu nad behom programu. Na sledovanie programu vyžaduje podporu systému (pri vzorkovaní) alebo prekladača (pri inštrumentácii). Profilovanie v rámci simulátora je špecifické v tom, že profiler je tesne zviazaný s profilovaným systémom - je priamo súčasťou simulátora - a jeho možnosti sú teda oveľa väčšie, vyžaduje však úpravy jadra simulátora.

## 3.1 Inštrumentácia

Profilovanie programov môže zahŕňať vkladanie špeciálnych profilovacích inštrukcií na úrovni strojového kódu, prípadne vkladanie volaní špecializovaných funkcií na úrovni zdrojového kódu, ktoré explicitne zaznamenávajú vybrané dáta počas behu programu. Nevýhodou tohto prístupu je skreslenie výsledkov spôsobené réžiou pridaného kódu, ktorý môže ovplyvňovať celý beh programu. Inštrumentácia môže mať tiež nezanedbateľný vplyv na celkovú dobu trvania programu, preto nemusí byť použiteľná v prípade časovo náročných programov [12].

## 3.2 Štatistické profilovanie

Štatistické profilovanie využíva vzorkovanie. Do programu nie je pridaný žiadny profilovací kód, ktorý by ovplyvňoval výsledky, profilovanie je spustené v samostatnom vlákne a v pravidelných intervaloch na základe systémových prerušení je zaznamenávaný kontext programu - hodnota čítača inštrukcií (*program counter*) a prípadne stav zásobníka (*call stack*). Výsledky štatistického profilovania sú aproximáciou reálneho správania programu, pričom vzniknutá chyba závisí od použitej vzorkovacej frekvencie. Rýchlosť programu je však ovplyvňovaná menej ako pri inštrumentácii [12].

Táto metóda typicky vyžaduje podporu operačného systému a nástroje využívajúce tento prístup sú väčšinou závislé na konkrétnej platforme.

## 3.3 Profilovanie v rámci simulátora

Pri interpretovaní programu v simulátore (resp. v interprete) je možné zbierať dáta o behu bez akejkoľvek modifikácie samotného programu. V rámci simulácie (interpretácie) je možné ľubovoľne nastaviť frekvenciu vzorkovania a aj to, ktoré dáta sa budú zaznamenávať. Výhodou je nezávislosť na platforme a plná kontrola nad behom programu. Problémom však môže byť rýchlosť, preto je kladený veľký dôraz na efektívnosť simulátora [12].

## 3.4 Spracovanie zozbieraných dát

Spracovanie zozbieraných dát je spravidla oddelený proces, ktorý kvôli zvýšenej časovej náročnosti prebieha až po ukončení behu programu. Pre analýzu využitia inštrukcií stačí analyzovať hodnoty čítača inštrukcií. Pre analýzu na úrovni zdrojového kódu je však potrebné opätovne namapovať program do pamäti a na základe adries z čítača v kombinácii s ladiacimi informáciami identifikovať bloky zdrojového kódu.

Okrem toho je možné za použitia snímkov zásobníka zrekonštruovať postupnosť volaní jednotlivých funkcií, pričom pri použití vzorkovania musí algoritmus dokázať pracovať s neúplným záznamom - vynorenie z / vnorenie do viacerých funkcií naraz. V tomto prípade sú výstupné informácie o volaniach len orientačné, keďže niektoré volania sa môžu úplne stratiť, alebo môže dôjsť k ich zlúčeniu.

Profilovacie nástroje štandardne poskytujú dva pohľady na volané funkcie - graf volaní (*call graph*) a plochý profil (*flat profile*). Zatiaľ čo plochý profil obsahuje iba celkový počet volaní a celkový čas strávený v jednotlivých funkciách, graf volaní poskytuje detailnejšie informácie o hierarchii volaní - pre každú funkciu obsahuje zoznam funkcií, z ktorých bola zavolaná a ktoré volala, koľko krát sa uskutočnili a ako dlho trvali jednotlivé volania. Ako príklad oboch pohľadov je na obrázkoch 4.4 a 4.5 v kapitole 4 uvedený výstup programu *gprof*.

# Kapitola 4

# Existujúce nástroje

Spomedzi existujúcich profilovacích nástrojov medzi najznámejšie patria programy *OProfile* a *gprof*, ktoré slúžia na analýzu bežných programov prekladaných do strojového kódu. Obidva poskytujú informácie na úrovni inštrukcií aj zdrojového kódu, avšak využívajú odlišné prístupy.

## 4.1 OProfile

OProfile je open source projekt, ktorý zahŕňa štatistický profiler pre Linuxové systémy, schopný profilovať akýkoľvek bežiaci kód s nízkou réžiou.[1]

### 4.1.1 Operf

Program *operf* tvorí jadro profileru OProfile. Na základe konfigurácie v podobe vstupných parametrov je schopný zbierať dáta o behu programu v rôznych režimoch - dokáže reagovať na rôzne systémové alebo hardvérové prerušenia, sledovať systémové počítadlá (*performance counters*) a ukladať hodnotu inštrukčneho čítača v reakcii na špecifikovanú udalosť.

Profilovaný program je spustený ako samostatný proces, ktorého stav je paralelne monitorovaný programom *operf*, vďaka čomu je vplyv profilovania na beh programu minimálny.

Jednotlivé vzorky sú ukladané do vyrovnávacej pamäte prislúchajúcej konkrétnemu jadru procesora. Okrem hodnôt inštrukčného čítača môže byť zaznamenávaný aj obsah zásobníka - adresy volaných funkcií. Dáta z jednotlivých jadier procesora sú priebežne premiestňované do jednotnej vyrovnávacej pamäte spolu s identifikátorom príslušného jadra a nakoniec sú zapisované do súborov na disk.

Názvy súborov spolu s cestami majú komplikovanú schému a obsahujú v sebe okrem iného názov profilovaného programu, knižnice, identifikátor udalosti, procesu a vlákna.[2]

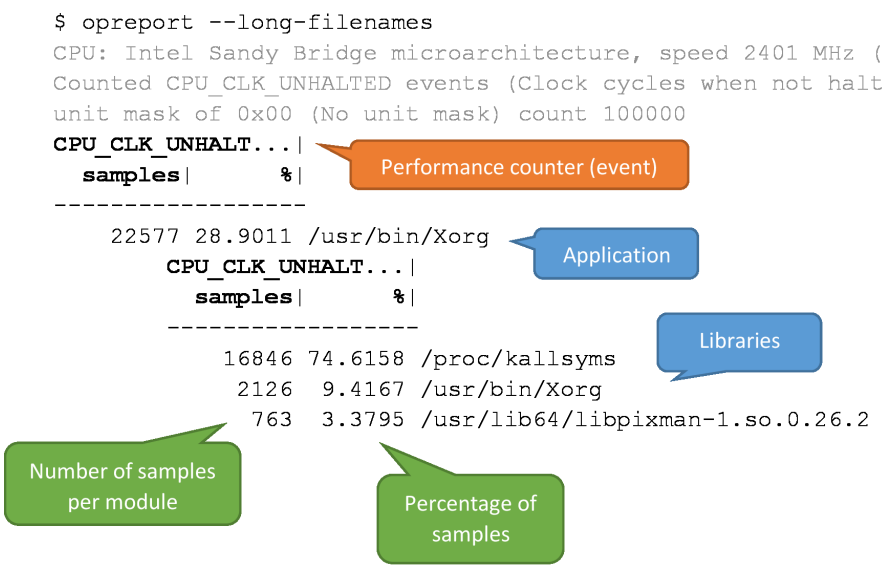
Komplikovaná súborová štruktúra síce umožňuje jednoduchšie triedenie a výber profilovacích súborov v priebehu spracovania (*post-processing*), avšak môže byť prekážkou v prípade niektorých operačných systémov, ktoré limitujú dĺžku cesty k súboru (napr. Windows).

### 4.1.2 Opreport

Spracovanie profilovacích záznamov, ktoré generuje *operf*, a ich zobrazenie v zrozumiteľnom formáte má na starosti program *opreport*.

[1] http://oprofile.sourceforge.net/about/

[2] http://oprofile.sourceforge.net/doc/internals/sample-file-generation.html




Obr. 4.1: Ukážka výstupu programu *opreport* (bez rozlíšenia symbolov)

Pomocou vstupných parametrov je možné zvoliť, ktoré dáta majú byť spracované. Vďaka komplikovanej súborovej štruktúre profilovacích záznamov môže prebehnúť selekcia relevantných súborov už na základe názvov, čím sa môže v prípade profilovania rozsiahlych komplexných programov celý proces zjednodušiť.[3]

Dôležitou súčasťou výstupu je graf volaní (*call graph*), ktorý poskytuje informácie o postupnosti a hierarchii volaní jednotlivých funkcií na úrovni zdrojového kódu. Okrem toho obsahuje údaje o tom, koľko percent času behu programu bolo strávených v určitej funkcii. Tento údaj je však len štatistický (kvôli vzokovaniu) a vychádza z počtu vzoriek, v ktorých bolo volanie danej funkcie zaznamenané.

Okrem štandardného textového výpisu do tabuliek je možné generovať aj výstup vo formáte XML, ktorý odstraňuje niektoré nedostatky klasického dvojrozmerného výstupu.[4]

Na obrázku 4.1 je zobrazená ukážka výstupu programu *opreport* bez rozlíšenia symbolov. Obsahuje názov sledovaného hardvérového počítadla, počet a percento vzoriek, v ktorých bola zaznamenaná činnosť daného programu (*Xorg*) a rozdelenie vzoriek medzi jeho jednotlivé súčasti - samotný program a knižnice.

Obrázok 4.2 zobrazuje výstup programu *opreport* s rozlíšením symbolov. Počet a percento vzoriek teda udáva vzhľadom ku konkrétnej funkcii v rámci programu.

### 4.1.3 Opannotate

Nástroj *opannotate* umožnuje kombinovať profilovacie informácie so zdrojovým kódom programu, resp. s kódom symbolických inštrukcií (*assembly language*). V tomto prípade je nutné, aby preložený program obsahoval ladiace informácie. Z profilovacích dát je extrahovaný počet prístupov na jednotlivé adresy v programe, ktoré sú potom premietnuté na riadky v zdrojovom kóde. Výstup obsahuje pri každom riadku zdrojového kódu počet vzoriek, v ktorých bol zaznamenaný a percentuálne vyjadrenie tejto hodnoty (viď Obr. 4.3).[5]

[3] http://oprofile.sourceforge.net/doc/internals/sample-file-generation.html
[4] http://oprofile.sourceforge.net/doc/opreport.html#opreport-xml
[5] http://oprofile.sourceforge.net/doc/opannotate.html

```
$ opreport -l -p /lib/modules/`uname -r` `which operf` 2>/dev/null | more
CPU: Intel Sandy Bridge microarchitecture, speed 2401 MHz (estimated)
Counted CPU_CLK_UNHALTED events (Clock cycles when not halted) with a
unit mask of 0x00 (No unit mask) count 100000
samples  %        image name               symbol name
860      7.4607   kallsyms                 avtab_search_node
474      4.1121   operf                    OP_perf_utils::op_write_event(
event_union*, unsigned long long)
461      3.9993   kallsyms                 avc_has_perm_noaudit
455      3.9473   libstdc++.so.6.0.13      /usr/lib64/libstdc++.so.6.0.13
412      3.5742   libc-2.12.so             _IO_vfscanf
369      3.2012   kallsyms                 __d_lookup
350      3.0363   kallsyms                 sidtab_context_to_sid
```

Obr. 4.2: Ukážka výstupu programu *opreport* (s rozlíšením symbolov)

```
                :static uint64_t pop_buffer_value(struct transient * trans)
11510  1.9661 :{ /* pop_buffer_value total:  89901 15.3566 */
                :    uint64_t val;
                :
10227  1.7469 :    if (!trans->remaining) {
                :       fprintf(stderr, "BUG: popping empty buffer !\n");
                :       exit(EXIT_FAILURE);
                :    }
                :
                :    val = get_buffer_value(trans->buffer, 0);
 2281  0.3896 :    trans->remaining--;
 2296  0.3922 :    trans->buffer += kernel_pointer_size;
                :    return val;
10454  1.7857 :}
```

Obr. 4.3: Ukážka výstupu programu *opannotate*

## 4.2 Gprof

Pôvodná verzia programu *gprof* vznikla na kalifornskej univerzite v Berkeley pre Berkeley Linux (BSD). Neskôr, v roku 1988, vytvoril Jay Fenlan v rámci projektu GNU (*GNU Binutils*) novú verziu.[6]

### 4.2.1 Preklad programu

Profiler *gprof* funguje na princípe inštrumentácie s využitím vzorkovania [5].

Kvôli inštrumentácii vyžaduje podporu prekladača - pri preklade je modifikovaný proces volania funkcií a do programu sú vložené volania monitorovacích procedúr profilera. Do GNU prekladača *gcc* je priamo integrovaný ako prepínač *-pg*, ktorý zabezpečí nahradenie štandardného počiatočného súboru *crt0.c* upraveným súborom *gcrt0.c*, upraví prológy funkcií a pripojí profilovaciu verziu štandardnej knižnice jazyka C (*libc_p.a*).[7]

Sledovanie prístupov na adresy v kóde je implementované periodickým sledovaním hodnoty inštrukčného čítača na základe systémového prerušenia. Vo väčšine UNIX systémov je na tento účel využité volanie systémovej funkcie *profil* na začiatku programu, ktorá zabezpečí zaznamenávanie počtu prístupov na jednotlivé adresy do zaregistrovaného poľa na systémovej úrovni. V prípade systémov, kde funkcia *profil* nie je dostupná, je použitý časovač (volanie *setitimer*), avšak tento prístup vyžaduje podstatne vyššiu réžiu a jeho výsledky vykazujú menšiu presnosť.[8]

### 4.2.2 Beh programu

Po preložení programu pre profilovanie (s prepínačom *-pg*) je potrebné program spustiť s požadovanými argumentmi. Výstup a správanie programu nebudú ovplyvnené, avšak doba behu programu sa môže predĺžiť - to je spôsobené réžiou profilovacieho kódu vloženého do programu.

Vzorkovacia frekvencia záleží od použitého systému, ale typicky sa hodnota inštrukčného čítača zaznamená približne 100 krát za sekundu času behu programu. Keďže interval je pevne daný vzhľadom k čistému času programu, profilovací výstup neodhalí dobu čakania programu (napr. v dôsledku IO operácii alebo prepínania úloh), a teda výsledná doba programu uvedená v profilovacom výstupe môže byť značne kratšia ako reálna doba od začiatku behu programu po jeho ukončenie. Na druhej strane, výhodou tohto prístupu je eliminovanie vplyvu ostatných súčasne bežiacich procesov na výsledky profilovania.

Po skončení programu sa profilovacie dáta zapíšu do súboru *gmon.out* v aktuálnom priečinku programu.[9]

### 4.2.3 Spracovanie profilovacích dát

Na spracovanie profilovacích dát získaných počas behu programu slúži program *gprof*. Formát profilovacích dát je podstatne jednoduchší oproti výstupu profileru *OProfile* - jedná sa len o 1 súbor, ktorý obsahuje hlavičku a 3 časti: histogram prístupov na adresy (*Histogram Records*), záznam volaní jednotlivých funkcií (*Call-Graph Records*) a záznamy o počte vykonaní základných blokov (*Basic-Block Execution Count Records*).

[6] http://www.gnu.org/bulletins/bull5.html
[7] https://sourceware.org/binutils/docs/gprof/Compiling.html#Compiling)
[8] https://sourceware.org/binutils/docs/gprof/Implementation.html#Implementation
[9] https://sourceware.org/binutils/docs/gprof/Implementation.html#Implementation

```
Each sample counts as 0.01 seconds.
%     cumulative   self              self     total
time     seconds   seconds    calls  ms/call  ms/call  name
33.34        0.02     0.02     7208     0.00     0.00  open
16.67        0.03     0.01      244     0.04     0.12  offtime
16.67        0.04     0.01        8     1.25     1.25  memccpy
16.67        0.05     0.01        7     1.43     1.43  write
16.67        0.06     0.01                             mcount
0.00        0.06     0.00      236     0.00     0.00  tzset
```

Obr. 4.4: Ukážka výstupu programu *gprof* - *flat profile*

```
index % time    self  children    called     name
<spontaneous>
[1]    100.0    0.00    0.05                 start [1]
0.00    0.05       1/1           main [2]
0.00    0.00       1/2           on_exit [28]
0.00    0.00       1/1           exit [59]
-----------------------------------------------
0.00    0.05       1/1           start [1]
[2]    100.0    0.00    0.05       1         main [2]
0.00    0.05       1/1           report [3]
-----------------------------------------------
0.00    0.05       1/1           main [2]
[3]    100.0    0.00    0.05       1         report [3]
0.00    0.03       8/8           timelocal [6]
0.00    0.01       1/1           print [9]
```

Obr. 4.5: Ukážka výstupu programu *gprof* - *call graph*

Pri analýze grafu volaní sa postupuje od nasledovníkov k predchodcom (od listov ku koreňu stromu), pričom v prípade rekurzívnych volaní, ktoré v grafe predstavujú cykly, sa najprv pomocou Tarjanovho algoritmu vyčlenia silne súvislé komponenty (*strongly-connected components*) - podgrafy, v ktorých z každého uzla existuje sled do všetkých ostatných uzlov v oboch smeroch - ku ktorým sa pristupuje ako k samostatným uzlom. Čas strávený v jednotlivých funkciách sa distribuuje v rámci týchto komponentov osobitne [5].

Na obrázku 4.4 je zobrazený plochý profil volaných funkcií, ktorý obsahuje súhrnné štatistiky pre každú funkciu. V prvom stĺpci je uvedené percentuálne vyjadrenie času stráveného v danej funkcii počas behu programu. Nasleduje kumulatívny čas (*cumulative seconds*), tj. počet sekúnd strávených vo funkcii a všetkých funkciách z nej volaných. Tretí stĺpec pre každú funkciu uvádza čistý čas (*self seconds*), do ktorého sa nezapočítavajú volané funkcie. Nasleduje počet volaní a počet milisekúnd pripadajúcich na jedno volanie.

Obrázok 4.5 zobrazuje komplikovanejší výstup - graf volaní. Pre každú funkciu je vytvorený záznam, pričom jednotlivé záznamy sú oddelené horizontálnymi čiarami a sú očíslované. Každý záznam obsahuje riadok s funkciou, ktorej je venovaný, nazývaný primárny riadok (obsahuje číslo v stĺpci index). Nad ním sú uvedené funkcie ktoré danú funkciu volali, pričom ku každej je uvedený počet volaní a čas strávený v primárnej funkcii keď bola volaná danou funkciou. Pod primárnym riadkom sa nachádzajú volané funkcie. Stĺpec *called* uvádza, koľko krát bola daná funkcia zavolaná primárnou funkciou a koľko krát dokopy v celom programe.

# Kapitola 5

# Návrh profileru pre Codasip Studio

Hlavným motívom tejto práce je vytvorenie profilovacieho nástroja pre firmu Codasip Ltd., ktorý bude použiteľný pri analýze aplikácií pre aplikačne špecifické procesory. Cieľom profilovania teda nie je len analýza programu, ale aj samotného procesora - efektívnosti jeho návrhu vzhľadom k cielenému použitiu (triede aplikácií).

## 5.1 Požadované vlastnosti

- Z hľadiska sledovaných metrík je potrebné analyzovať program na úrovni zdrojového kódu, inštrukcií a kódu jazyka CodAL (tzn. popisu samotného procesora).
- Kým analýza zdrojového kódu dokáže odhaliť chyby v návrhu programu a identifikovať oblasti kódu, ktorých optimalizácia môže byť výhodná, sledovaním využitia jednotlivých inštrukcií je možné vyhodnotiť účelnosť a efektívnosť navrhnutej inštrukčnej sady. Pridanie ďalších monitorovaných parametrov, ako sú prístupy do pamäti, využitie registrov alebo vznik hazardov pri zreťazenom vykonávaní inštrukcií umožňuje účinné profilovanie architektúry procesora, čo zvyšuje užitočnosť profilera pri návrhu aplikačne špecifických procesorov, pri ktorom je optimálne spojenie softvéru aj hardvéru kľúčové.
- S ohľadom na existujúce nástroje od firmy Codasip Ltd. musí byť profiler integrovateľný do vývojového prostredia Codasip Studio IDE[1]. Zároveň je však žiadúce aby bol podobne ako ostatné nástroje zahrnuté v Codasip Studiu použiteľný samostatne, bez nutnosti spúšťania IDE, teda ako program spustiteľný z príkazového riadku (*command line utility*).
- Profilované programy sú spúšťané v simulátore, ktorý musí byť vysoko optimalizovaný na výkon. Je preto nevyhnutné, aby všetky operácie súvisiace s profilovaním boli minimalizované a implementované čo najefektívnejšie tak, aby réžia profileru bola čo najmenšia.
- Hlavný výstup profileru, ktorý bude primárne zobrazovaný v rámci vývojového prostredia Codasip Studio, by mal byť tvorený interaktívnymi grafickými prvkami, ako sú tabuľky a grafy, na ktoré je možné kliknúť a zobraziť podrobnejšie detaily o danej metrike. Toto grafické rozhranie by však malo byť nezávislé od Codasip Studia, tak

[1] *Integrated Development Environment* - Integrované vývojové prostredie

aby sa dal výstup zobraziť aj bez neho. Okrem toho musí byť dostupný textový výstup použiteľný v prípade spustenia cez príkazový riadok. Navyše je požadovaný výstup kompatibilný s projektom *OProfile*, tak aby ho bolo možné spracovať programom *opannotate*.

- Jednou z funkcií profileru by mala byť anotácia zdrojového kódu podobne ako je to v prípade programu *opannotate* - tzn. doplnenie súborov obsahujúcich zdrojový kód o informácie o počte vykonaní jednotlivých riadkov.
- Codasip Studio umožňuje vytvárať a simulovať multiprocesorové platformy (*multi-core*), s čím musí nový profiler počítať.

## 5.2 Návrh

Pri návrhu profilera je potrebné, podobne ako to bolo pri popísaných existujúcich profilovacích nástrojoch, odlíšiť dva hlavné procesy, a to zber dát počas behu programu a spracovanie zozbieraných dát po skončení programu (*post-processing*). Rozlíšenie je nutné najmä kvôli vysokým nárokom na rýchlosť simulácie a potenciálne vysokej časovej náročnosti analýzy profilovacích dát.

### 5.2.1 Požiadavky na zber dát

Zber dát je najkritickejšou fázou profilovania. Zatiaľ čo pri profilovaní bežných aplikácií vykonávaných procesorom počítača je potrebná buď podpora prekladača za účelom inštrumentácie alebo podpora periodického sledovania stavu programu v systéme, pri profilovaní programu v rámci simulátora je funkcionalita potrebná na zber dát o behu programu priamo súčasťou simulátora. Pri vykonaní inštrukcie je automaticky spustený proces zaznamenania stavu programu. Keďže sa jedná o vysoko frekventovanú operáciu, jej implementácia musí byť čo najefektívnejšia a optimalizovaná na čo najnižšej úrovni.

- Dôležitý je výber sledovaných hodnôt. Principiálne stačí na rekonštrukciu priebehu programu hodnota inštrukčného čítača v každom hodinovom cykle. V praxi však tento prístup z dôvodu nárastu objemu dát nie je vhodný a vo valnej väčšine prípadov nie je takýto podrobný záznam ani potrebný. Vhodným spôsobom minimalizácie dát je v tomto prípade agregácia prístupov na rovnakú adresu - teda vytvorenie histogramu.
- Pri agregovaní počtu prístupov počas behu programu nie je možné zapisovať získané hodnoty priebežne na výstup (na disk) - nevyhnutnosťou je existencia dátovej štruktúry slúžiacej ako vyrovnávacia pamäť pre uchovávanie aktuálnych hodnôt počítadiel, ku ktorým musí umožňovať rýchly prístup na základe adresy (asociatívna mapa).
- Vo väčšine prípadov nie sú absolútne presné profilovacie výsledky nutnosťou. Prioritou je okamžitá dostupnosť a pohodlná manipulácia s výsledkami, čo vytvára tlak na rýchlosť simulácie a rozumný objem výstupných dát aj za cenu akceptovateľných nepresností. To vedie k zavedeniu možnosti vzorkovania. Ideálne je užívateľovi ponechaná možnosť nastaviť si vzorkovaciu frekvenciu podľa potreby, pričom možnosťou je aj vzorkovanie v každom takte.
- Za účelom analýzy a optimalizácie programu na úrovni zdrojového kódu je výhodné poznať hierarchiu, počet a trvanie volaní funkcií. To nie je problém, ak je k dispozícii

zaznamenaná hodnota inštrukčného čítača v každom cykle, avšak pri použití agregácie prístupov podľa adresy sa informácia o časovom priebehu programu stráca. Je teda potrebné uchovávať obsah zásobníka, čo zahŕňa jeho rozvinutie a identifikáciu adries funkcií v simulátore. Vzhľadom k veľkosti výstupných dát je vhodné aj tu použiť vzorkovanie, to však vyžaduje pokročilejšiu analýzu pri rekonštrukcii hierarchie volaní (*call-stacku*) počas fázy spracovania výsledkov (*post-processing*) z dôvodu chýbajúcich záznamov, s čím treba počítať.

- Keďže jednotlivé záznamy (snímky) stavu zásobníka sú navzájom nezávislé (okrem poradia), je možné ich na rozdiel od agregovaných hodnôt ukladať na výstup (disk) priebežne, čo znamená určitú úsporu operačnej pamäti.

- Pri optimalizácii inštrukčnej sady procesora sú potrebné štatistiky o využití jednotlivých inštrukcií - jedná sa o počet dekódovaní konkrétnej inštrukcie oproti všetkým dekódovaným inštrukciám. Z toho dôvodu je nutné ukladať hodnoty na vstupe dekodérov (vo všeobecnosti môže dekodérov existovať viacero).

- Dôležitá je tiež identifikácia často vykonávaných sekvencií inštrukcií - v rámci optimalizácie je potom možné nahradiť sekvenciu *RISC* inštrukcií komplexnou *CISC* inštrukciou. Dĺžka sekvencií môže byť rôzna, ale keďže každá sekvencia musí byť zachytávaná počas behu programu, je treba zvážiť potrebnú réžiu a jej dopad na rýchlosť a pamäťové nároky simulácie. Ideálne je voľba dĺžky sledovaných sekvencií ponechaná užívateľovi.

- K ďalším parametrom dôležitým z hľadiska analýzy procesora patrí prístup k zdrojom (napr. registrom), a k jednotlivým oblastiam pamäte, konkrétne počet zápisov a čítaní. Podobne môže byť užitočné sledovať udalosti v rámci konkrétnych štádií zreťazenej linky procesora - zdržanie zreťazenej linky (*pipeline stall*) alebo jej vyprázdnenie (*pipeline clear*) - alebo úspešnosť vyrovnávacej pamäte (*cache hit rate*). Výpadky vyrovnávacej pamäte (*cache miss*) - stav keď sa požadované dáta nenachádzaju vo vyrovnávacej pamäti a musia byť načítané z operačnej pamäte - majú totiž výrazný vplyv na efektívnosť aplikačne špecifického procesora [8].

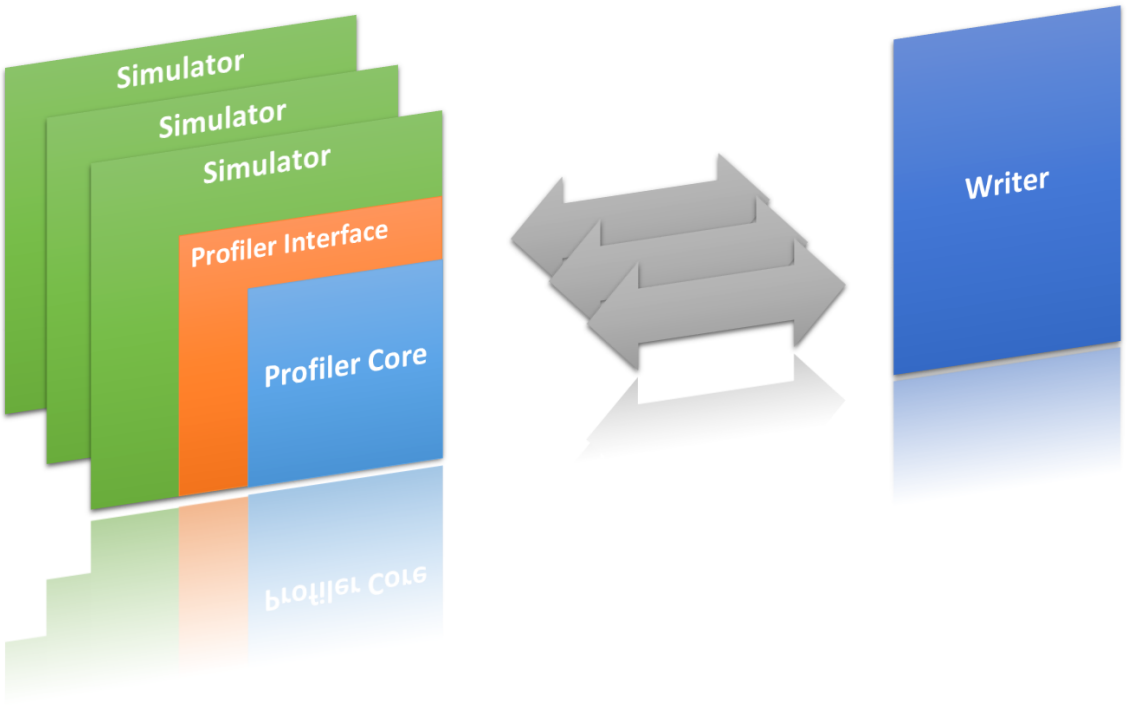
- Všetky tieto hodnoty sú uchovávané formou histogramu - pri inkrementácii hodnoty je potrebné nájsť konkrétne počítadlo podľa určitého kľúča, čo znovu vyžaduje použitie dátovej štruktúry optimalizovanej na rýchle vyhľadávanie.

- V prípade multiprocesorových simulácií musí profiler dokázať priebežne spracúvať dáta z rôznych modelov. Zároveň musí korektne reagovať na udalosti štart, stop a reset (konkrétneho procesora alebo celej platformy).

### 5.2.2 Interný dátový formát

V závislosti od komplexnosti programu a dĺžky simulácie je potrebné počítať s nárastom objemu zozbieraných dát a čo najviac minimalizovať veľkosť profilovacieho záznamu. Riešením je vytvorenie vlastného binárneho formátu s pevne danou štruktúrou.

Nevýhodou binárneho súboru je náročnejšie odhaľovanie chýb. Preto je nevyhnutné prácu so súborom centralizovať tak, aby prípadná zmena formátu vyžadovala čo najmenej zásahov do kódu (ideálne len dve - v zapisujúcom a čítajúcom module). Nutnosťou je tiež používanie jednotných dátových typov a spôsobu ich uloženia (*Big Endian* vs. *Little Endian*) nezávisle na platforme.




Obr. 5.1: Architetúra zberača dát (*Runtime profiler*)

### 5.2.3 Architektúra zberača

Zberač dát (*Runtime Profiler*), ktorý je súčasťou simulátora, je tvorený generickým jadrom (*Profiler Core*) implementovaným formou C++ knižnice a rozhraním (*Profiler Interface*), ktoré sa generuje spolu so simulátorom, je špecifické pre konkrétny procesor a odráža napríklad rôzne bitové šírky zberníc a dekodérov.

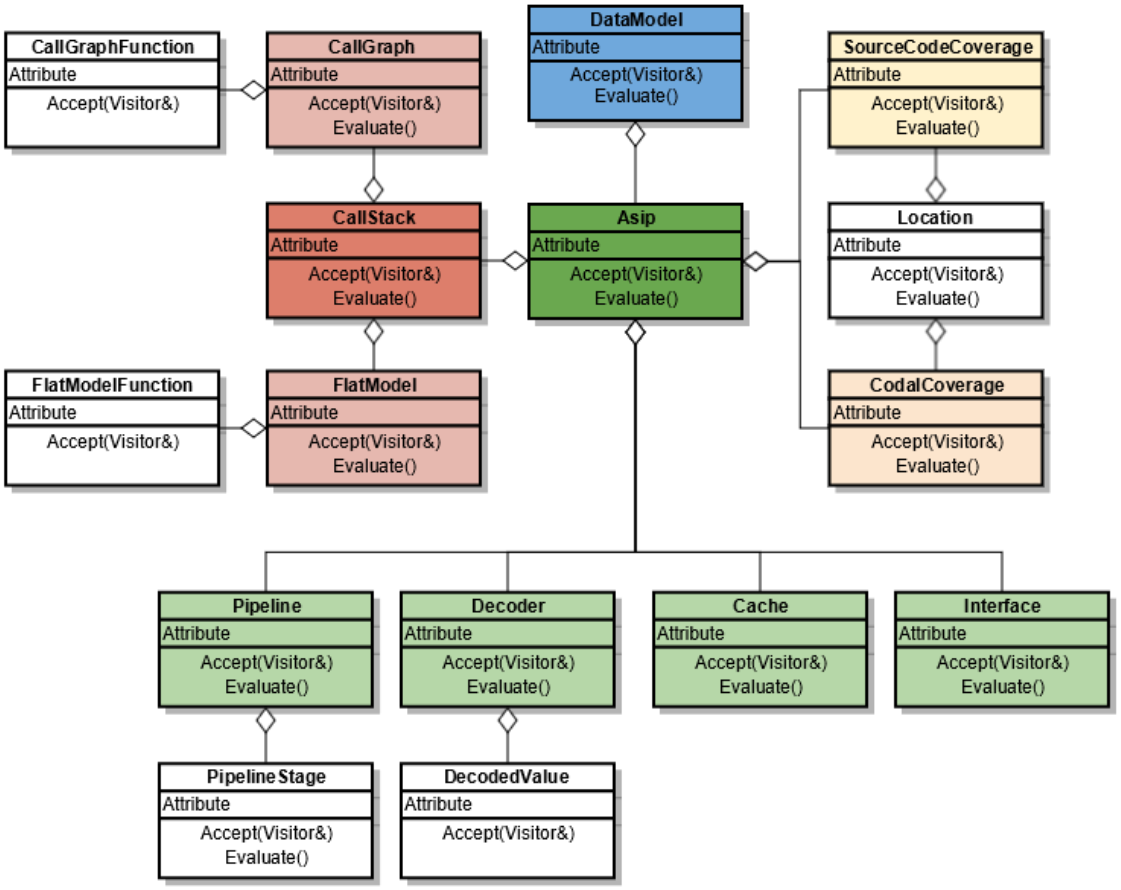
Pri multiprocesorovej simulácii existuje viacero simulátorov tvoriacich hierarchickú stromovú štruktúru. V prípade profilera nebol dôvod túto štruktúru kopírovať, postačujúci bol plochý model. Preto, hoci existuje viacero zberačov (pre jednotlivé procesory), všetky operujú na rovnakej úrovni nezávisle od seba. Centrálnym prvkom je zapisovač (*Writer*), ktorý priebežne prijíma dáta od jednotlivých zberačov a riadi formát a synchronizáciu výstupu. Architektúra je znázornená na obrázku 5.1.

### 5.2.4 Spracovanie profilovacích dát

Spracovanie profilovacích dát (*post-processing*) zahŕňa načítanie profilovacieho záznamu, uloženie profilovacích dát do vhodných dátových štruktúr, vykonanie potrebných dodatočných výpočtov a transformáciu a zobrazenie výsledkov vo formáte zrozumiteľnom pre užívateľa. Keďže profiler má podporovať niekoľko druhov výstupov, je logické oddeliť prezentačnú vrstvu od datového modelu. Podobne načítanie vstupu a naplnenie modelu je samostatnou činnosťou, ktorá priamo nesúvisí s analýzou.

#### Dátový model

Analyzované dáta tvoria určitú hierarchiu, ktorú zachytáva datový model. Jednotlivé hodnoty je vhodné logicky usporiadať podobne ako je to v prípade komponentov, ktoré popisujú

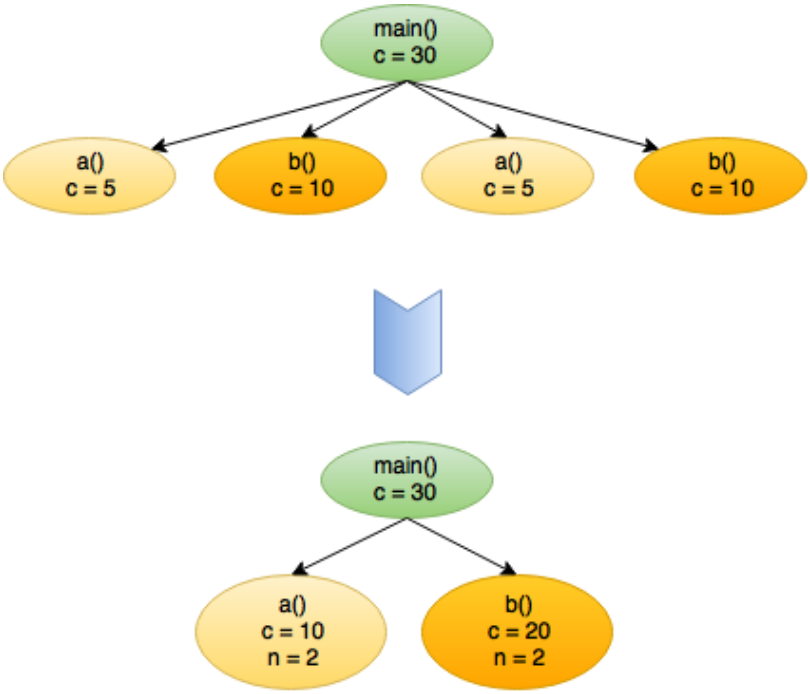



Obr. 5.2: Zjednodušený dátový model

- napríklad dekódované inštrukcie patria jednotlivým dekodérom, ktoré existujú v rámci konkrétneho procesora (CPU). Takáto dátová štruktúra potom umožnuje intuitívnejšiu manipuláciu a celý proces sprehľadňuje. (Obr. 5.2)

Jednotlivé entity dátového modelu sú ucelené jednotky, ktoré okrem uchovávania hodnôt obsahujú aj zapuzdrené evaluačné algoritmy (napríklad výpočet percent, vyčlenenie nepoužitých inštrukcií alebo roztriedenie hodnôt do definovaných intervalov pre účely vytvorenia histogramu), pričom každá entita sa stará o vyhodnotenie seba a svojich potomkov. Z globálneho pohľadu stačí spustiť vyhodnotenie celého modelu a správa o tom sa spropaguje do všetkých jeho častí, čím sa zvyšuje miera rozšíriteľnosti modelu (*extensibility*).

Špecifickú časť tvorí model hierarchie volaných funkcií, nad ktorým existujú tak ako v prípade profilerov *OProfile* a *gprof* dva pohľady - plochý profil a graf volaní - popísané v podkapitole 3.4. Základnou vnútornou reprezentáciou tohto modelu je stromová štruktúra (*call tree*), kde uzly stromu reprezentujú jednotlivé volania. Keďže oba pohľady prezentované užívateľovi nerozlišujú samostatné volania ale agregujú trvania volaní podľa názvu funkcie, je možné už v rámci interného modelu zlúčiť viacnásobné volania tej istej funkcie so spoločným predkom do jedného uzla a uchovávať len informáciu o počte volaní a súčte trvaní (Obr. 5.3). Prechodom tohto stromu je potom možné pre konkrétnu funkciu spočítať celkový čas trvania a počet jej volaní, pričom sú k dispozícii aj informácie o tom odkiaľ bola funkcia volaná a ktoré funkcie volala (pre účely grafu volaní).




Obr. 5.3: Interný strom volaní - zlúčenie volaní

### Analýza inštrukčnej sady

Po načítaní profilovacích dát sú profileru dostupné štatistiky k hodnotám dekódovaným počas behu programu. Tie je potrebné znovu spätne dekódovať a oddeliť inštrukcie a operandy, tak aby bolo možné agregovať štatistiky podľa inštrukcie nezávisle od hodnôt operandov. Vyčlenenie operandov je tiež potrebné pri analýze sekvencií inštrukcií, konkrétne pri identifikácii dátových závislostí (tokov). Toto dekódovanie má na starosti nástroj *disassembler* špecifický pre konkrétny ASIP.

### Analýza zdrojového kódu

Zdrojový kód je možné analyzovať z hľadiska volania funkcií (*call graph*), alebo z hľadiska jednotlivých riadkov zdrojového kódu.

Názvy funkcií (symboly) môžu byť za účelom rekonštrukcie grafu volaní získané z binárneho súboru aplikácie, kde sú uložené spolu s adresou v kóde vo formáte *ELF*.[2] Na rekonštrukciu postupnosti volaní postačujú adresy funkcií - názvy symbolov sú podstatné len z hľadiska výstupu.

Informácie o mapovaní adries kódu, ktoré má k dispozícii profiler, na riadky v zdrojových súboroch sú dostupné v binárnom súbore aplikácie vo formáte *DWARF*.[3] Skombinovaním týchto údajov je možné ohodnotiť každý riadok zdrojového kódu, a vytvoriť podobný výstup ako generuje program *opannotate*, kde riadok kódu obsahuje priamo informáciu o tom, v koľkých vzorkách bol zachytený. Navyše, použitím disassemblera je možné na základe rozsahu adries pre daný riadok získať úsek kódu symbolických inštrukcií (*assembly language*), čo po priradení štatistík k jednotlivým inštrukciám ponúka detailnejší pohľad na beh

[2] http://www.tachyonsoft.com/elf.pdf
[3] http://www.tachyonsoft.com/dwarf.pdf

programu a vytvára priestor pre hlbšiu analýzu.

### Analýza volaných funkcií

Vstupom pre analýzu volaných funkcií je záznam zásobníka procesora v čase vo forme snímkov stavu v okamihoch vzorkovania (viď Tab. 5.1). Pri postupnom prechode snímkov sa pripočítava uplynutý časový interval (vzorkovacia perióda) ku všetkým uzlom reprezentujúcim aktuálne rozpracované volania na zásobníku, pričom konkrétny uzol je identifikovaný postupnosťou funkcií v snímku - cestou od koreňa stromu. Ak sa v rámci snímku objaví volanie, pre ktoré neexistuje uzol, vytvorí sa nová vetva stromu. Výsledkom pripočítavania vzorkovacej periódy je pre každé volanie (uzol) tzv. kumulatívny počet cyklov volania (*cumulative clock cycles*), teda celková doba od zavolania funkcie až po návrat z nej. Čistý čas $s(F)$ strávený vo volaní $F$ (*self clock cycles*) je možné jednoducho vypočítať podľa vzorca 5.1, kde $G_i$ sú všetky volania vykonané v rámci volania $F$ a $c(X)$ je kumulatívny počet cyklov volania $X$.

$$s(F) = c(F) - \sum_{i=1}^{n} c(G_i) \tag{5.1}$$

V prípade, keď je použité vzorkovanie s periódou väčšou ako 1 sa môže stať, že v jednom snímku sa objaví $n$ nových volaní. Ak by sa ku každému pripočítala rovnaká hodnota $\Delta t$, tak napr. v príklade z Tab. 5.1 by funkcia $a$ mala rovnaký počet kumulatívnych cyklov ako $b$, z čoho by vyplývalo, že čistý čas $s(a)$ je rovný 0. Z toho dôvodu je použitá heuristika 5.2, kde $C(H_i)$ je počet kumulatívnych cyklov nového volania $H_i$ v aktuálnom snímku a $\Delta t$ je počet cyklov od posledného snímku (vzorkovacia perióda), pričom $i\epsilon < 1, n >$.

$$C(H_i) = \Delta t - (\Delta t/n) * (i - 1) \tag{5.2}$$

Po spracovaní všetkých snímkov sa prechodom stromu uzly volaní roztriedia podľa funkcie, ktorej prislúchajú a pri vyhodnocovaní plochého modelu sa jednoducho počty strávených cyklov a volaní sčítajú. Pri vyhodnocovaní grafu volaní sa postupuje rovnako, no pri každom uzle sa naviac evidujú volajúce a volané funkcie, teda rodič a potomkovia daného uzla, pre ktorých sa taktiež spočítavajú strávené cykly a počty volaní.

Komplikáciu tvoria rekurzívne volania, ktoré môžu byť priame ($a \rightarrow a$) alebo nepriame ($a \rightarrow b \rightarrow ... \rightarrow a$). Detekcia rekurzie prebieha už pri spracovaní snímkov zásobníka - akonáhle sa na zásobníku vyskytne tá istá funkcia viac ako raz, uzol odpovedajúci prvému volaniu tejto funkcie je označený ako vstup do rekurzie (*Recursion Entry*) a každé opakované volanie (akejkoľvek funkcie) je označené ako rekurzívne (*Recursive Call*). Pri agregácii je potom celkový kumulatívny počet cyklov funkcie $c(f)$ vypočítaný podľa vzorca 5.3, kde $F_i'$ sú rekurzívne (opakované) a $F_i$ nerekurzívne (prvotné) volania $f$.

$$c(f) = \sum_{i=1}^{n} c(F_i) - \sum_{i=1}^{m} c(F_i') \tag{5.3}$$

Postup vyhodnocovania volaných funkcií sa do značnej miery líši od postupu použitom v profileri *gprof*, čo je dané najmä rôznym formátom vstupných dát. Profiler *gprof* si strom volaní zostavuje už počas behu programu a pri spracovaní profilovacích dát len agreguje volania rovnakých funkcií, avšak keďže neanalyzuje stav zásobníka, nemá k dispozícii informáciu o aktuálne rozpracovaných volaniach, a preto je detekcia rekurzie komplikovanejšia.

| clock | snapshot | nodes |
|---|---|---|
| 10 | main | main(10) |
| 20 | main | main(20) |
| 30 | main a b | main(30), a(10), b(5) |
| 40 | main a b | main(40), a(20), b(15) |
| 50 | main a b c | main(50), a(30), b(25), c(10) |
| 60 | main | main(60), a(40), |

Tabuľka 5.1: Záznam zásobníka

Profiler musí zostaviť kompletný graf závislostí medzi funkciami, kde uzol reprezentuje funkciu a hrana volanie, a potom musí prebehnúť topologické radenie - analýza silne súvislých komponentov, teda slučiek, ktoré predsavujú rekurzívne závislosti. Naproti tomu, navrhovaný profiler má informácie o stave zásobníka k dispozícii a samotná detekcia rekurzie je teda triviálna.

### 5.2.5 Výstup

Profiler by mal podporovať minimálne 3 výstupné formáty - textový výstup, interaktívny HTML výstup a binárny výstup kompatibilný s formátom využitým v rámci open source projektu *OProfile*, konkrétne s výstupom programu *operf*.

Textový výstup musí obsahovať všetky základné štatistické hodnoty, organizované primárne do tabuliek. V prípade anotácie zdrojového kódu je výstup tvorený kópiami zdrojových súborov, kde každý riadok obsahuje v úvode zakomentovaný počet vykonaní a za riadkom nasleduje príslušný úsek kódu v jazyku symbolických inštrukcií tiež vo forme komentára.

Hlavným výstupom, ktorý je integrovaný do vývojového prostredia Codasip Studio, je kolekcia HTML stránok. Tento formát poskytuje možnosť interaktívnosti (vďaka hypertextovým odkazom a JavaScriptu) a väčšej názornosti a prehľadnosti zobrazovaných dát. Výsledky profilovania je možné logicky rozčleniť na viacero samostatných obrazoviek (podstránok), medzi ktorými sa dá ľahko navigovať. Použitím grafov a farebným odlíšením jednotlivých oblastí sa zvyšuje schopnosť užívateľa rýchlo sa zorientovať, zatiaľ čo zobrazovanie detailov po kliknutí na určitú oblasť pomáha užívateľovi sústrediť sa na vybranú časť výsledkov.

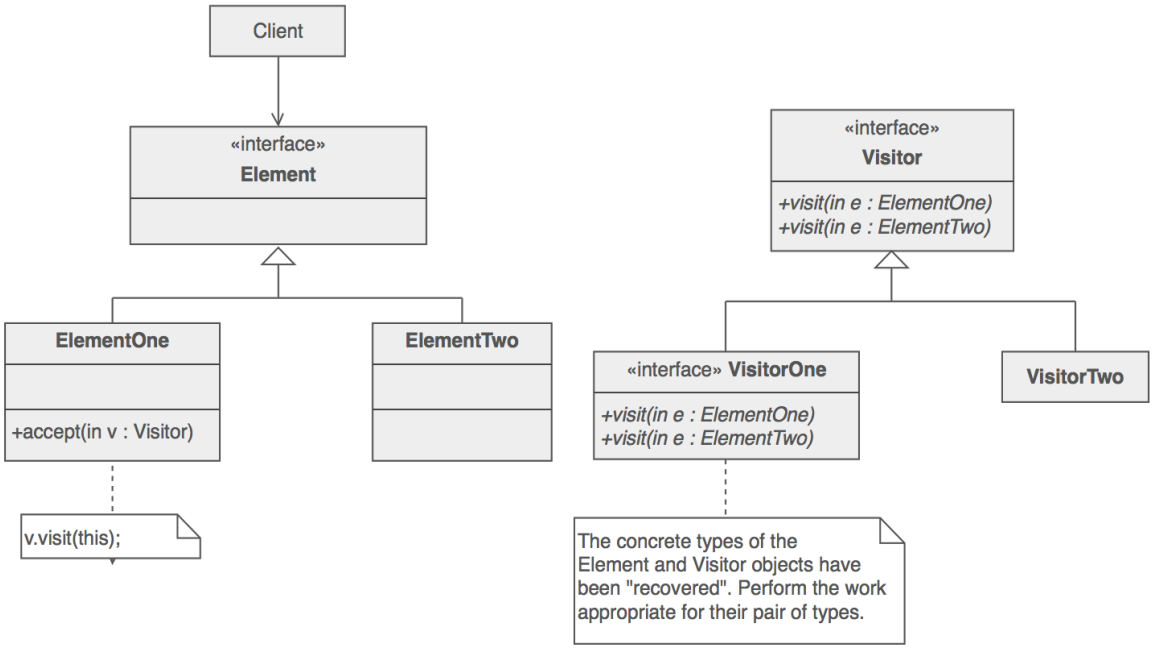
Kvôli rozšírenosti open source sady nástrojov *OProfile* je vyžadovaná podpora výstupu, s ktorým dokážu tieto nástroje pracovať. Štruktúru formátu je možné získať rozborom zdrojového kódu programu *operf*, ktorý ho vytvára, alebo programu *opreport*, ktorý ho spracúva.

#### Návrhový vzor *Visitor*

Hierarchická štruktúra datového modelu, nad ktorým je potrebné vykonať prechod, pričom konkrétne operácie nad jednotlivými uzlami sú pri rôznych typoch výstupu rôzne, bola hlavným predpokladom k zvoleniu architektúry *Visitor*. [11]

Ide o behaviorálny návrhový vzor, ktorý rieši problém vykonávania rôznych nezávislých operácií nad heterogénnou zloženou dátovou štruktúrou. Jeho schéma je uvedená na obrázku 5.4.

Každá operácia je reprezentovaná konkrétnou triedou odvodenou od bázovej abstraktnej




Obr. 5.4: Návrhový vzor *Visitor* [13]

triedy *Visitor* a obsahujúcou metódu *visit*, ktorá je preťažená pre rôzne typy elementov. Konkrétna implementácia operácie je teda vybraná automaticky na základe typu.

Zjavnou výhodou je oddelenie aplikačnej logiky od dát. V prípade potreby doplnenia novej operácie stačí implementovať novú triedu odvodenú od triedy *Visitor*. Takáto operácia je potom aplikovateľná na každý element implementujúci rozhranie *Element* obsahujúce metódu *accept*.

Pri aplikovaní operácie (invokácii metódy *accept*) teda dôjde k rozpoznaniu konkrétneho typu objektu *visitor* skrze polymorfické volanie *visit* a zároveň sa vyberie správna preťažená verzia tejto metódy na základe typu navštevovaného elementu. Tento koncept sa nazýva aj *double dispatch* [1].

V rámci implementácie nového profilera predstavuje operácia modelovaná konceptom *Visitor* spôsob výstupu. Konkrétne boli implementované štyri triedy zodpovedné za textový a html výstup, výstup pre *OProfile* a interný binárny výstup pre opätovné spracovanie (použitý pri kombinovaní profilovacích výsledkov z viacerých simulácií).

Okrem toho bol pri návrhu využitý fakt, že prechod dátovým modelom je pri generovaní výstupov rovnaký. Preto bol vyňatý do bázovej triedy *BaseVisitor* - tá zaručí zavolanie polymorfnej metódy *visit* nad daným prvkom aj jeho komponentmi. Vďaka tomu sa triedy reprezentujúce konkrétne typy výstupov stali de facto deklaratívnymi popismi toho, akým spôsobom je ktorá entita prezentovaná, bez ohľadu na spôsob alebo poradie uloženia entít v rámci modelu.

# Kapitola 6

# Výsledný profiler

Na obrázku 6.1 je zobrazená ukážka výstupu nového profilera integrovaného vo vývojovom prostredí Codasip Studio. V rámci pokrytia inštrukčnej sady obsahuje výstup 5 najčastejšie vykonávaných inštrukcií pričom pre každú uvádza počet dekódovaní a percentuálne zastúpenie. Pre porovnanie je na obrázku 6.2 uvedený aj textový formát výstupu zobraziteľný v termináli.

Obrázok 6.3 ukazuje príklad anotovaného zdrojového kódu pri zapnutej možnosti anotovania jazyka symbolických inštrukcií (*asembleru*). Ku každému riadku zdrojového kódu sú pridané informácie o tom, na ktorú adresu v pamäti sa premietol a koľko krát boli inštrukcie, na ktoré bol preložený, v rámci vzorkovania inštrukčného čítača zaznamenané (aj s percentuálnym zastúpením). Pod riadkom potom nasleduje jeho preklad do asembleru a štatistiky k jednotlivým inštrukciám.

Všetky doplnené anotácie sú uvedené v komentároch. Výsledný zdrojový kód je teda validným programom v jazyku C a je možné ho opätovne preložiť.

Okrem profilovacích informácií o počte vykonaní jednotlivých riadkov je tento výstup pre programátora užitočný aj na preskúmanie spôsobu prekladu jednotlivých riadkov zdrojového kódu do jazyka inštrukcií. Môže to byť výhodné pri optimalizovaní programu na nízkej úrovni.

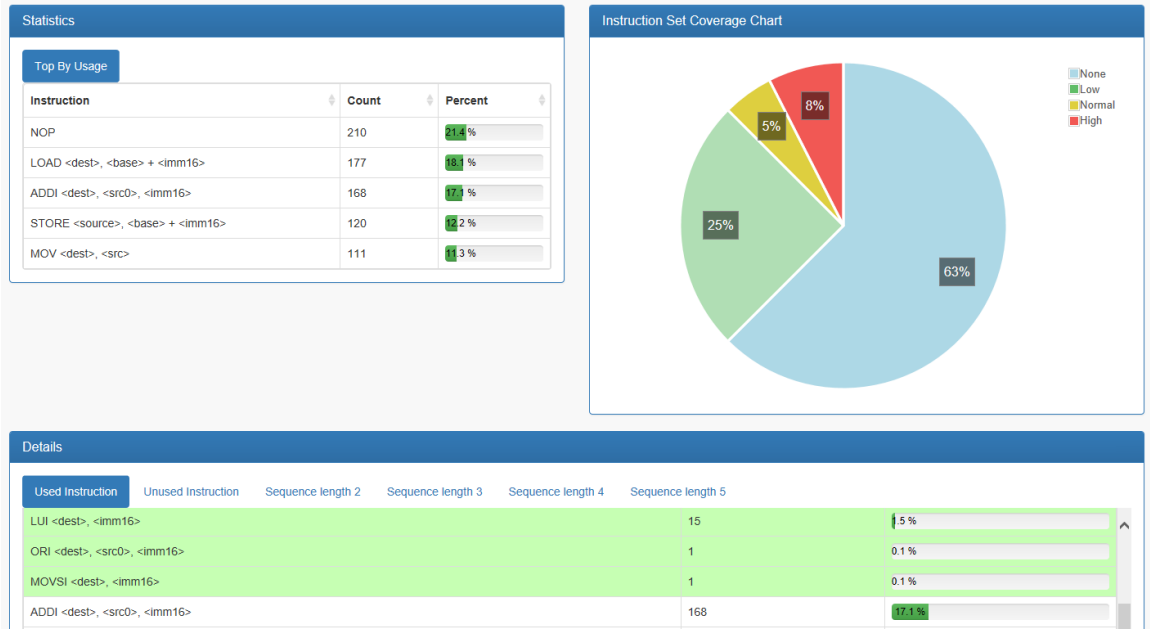
## 6.1 Príklad praktického použitia profilera

Ako modelový príklad použitia profilera poslúži aplikácia *Bitcount*, ktorá počíta nastavené bity (s hodnotu 1) v daných bitových poliach. Konkrétne je zadané pole 256 32-bitových hodnôt. Nad každým prvkom poľa je zavolaná funkcia *bitcount*, ktorá vracia počet jednotkových bitov.

Algoritmus spočítania jednotkových bitov je založený na paralelnom sčítaní bitov po skupinách a využíva konštantný počet operácií - niekoľko binárnych súčinov, posuvov, sčítaní a priradení.

V profilovacom výstupe vidieť, že najviac času trávi program práve vo funkcii *bitcount* - to naznačuje, že by mohla byť vhodným kandidátom na optimalizáciu. Vďaka tomu, že ide o relatívne krátky úsek kódu obsahujúci jednoduché aritmetické operácie, ktorý sa vykonáva často je vhodné preskúmať alternatívu nahradenia tejto funkcie špecializovanou inštrukciou.

Obrázky 6.4 a 6.5 ukazujú pohľad na dĺžku behu volaných funkcií *main* a *bitcount* pred a po definovaní špecializovanej inštrukcie *BITCNT*. Celkový počet cyklov strávených vo funkcii *bitcount* sa znížil z 16384 na 4352 a počet cyklov celého programu sa tým zmen-




Obr. 6.1: Výsledky profilovania vo formáte HTML

```
Used instructions
-----------------

+====================================+==========+=========+========+
| Instruction                        | Executed | Percent |  Usage |
+====================================+==========+=========+========+
| NOP                                |       42 |   16.9% |   High |
| ADD rf_gpr , rf_gpr , rf_gpr       |       10 |    4.0% |    Low |
| HALT                               |        1 |    0.4% |    Low |
| LUI rf_gpr , <imm>                 |        2 |    0.8% |    Low |
```

Obr. 6.2: Výsledky profilovania v textovom formáte

```
38                                  int f1()
39  /* 0x0b0 211 23.950% */         {
40  /*
41     0x0b0   61   6.924%          LOAD R2 , R1 + -12
42     0x0b4   60   6.810%          LOAD R3 , R1 + -8
43     0x0b8   30   3.405%          NOP
44     0x0bc   30   3.405%          ADD R2 , R2 , R3
45     0x0c0   30   3.405%          STORE R2 , R1 + -16
46  */
47  /* 0x0c4   90 10.216% */             return f0() + 1;
48  /*
49     0x0c4   30   3.405%               LOAD R2 , R1 + -12
50     0x0c8   30   3.405%               NOP
51     0x0cc   30   3.405%               STORE R2 , R1 + -8
52  */
53  /* 0x0d0   90 10.216% */         }
```

Obr. 6.3: Ukážka anotovaného zdrojového kódu

| Function | Address | Cumulative Percent | Cumulative Clock Cycles | Self Clock Cycles | Count |
|---|---|---|---|---|---|
| main | 0x170 | 100 | 22573 | 6189 | 1 |
| bit_count | 0x70 | 72.6 | 16384 | 16384 | 256 |

Obr. 6.4: Analýza volaných funkcií v programe *Bitcount*

| Function | Address | Cumulative Percent | Cumulative Clock Cycles | Self Clock Cycles | Count |
|---|---|---|---|---|---|
| main | 0xb4 | 99.9 | 10541 | 6189 | 1 |
| bit_count | 0x70 | 41.3 | 4352 | 4352 | 256 |

Obr. 6.5: Analýza volaných funkcií v programe *Bitcount* po optimalizácii

šil na polovicu. Je teda vidieť, že špecializované inštrukcie majú z hľadiska optimalizácie procesora veľký význam.

Ďalším krokom pri optimalizácii by mohlo byť odstránenie nepoužívaných inštrukcií - vhodným kandidátom je napríklad inštrukcia *MUL*. Nikde v programe sa totiž násobenie nepoužíva, a navyše by odstránením hardvérovej násobičky došlo k zníženiu výsledného počtu tranzistorov, čo by sa odrazilo na cene výsledného procesora.

Prezentovaný príklad bol zjednodušený a mal slúžiť len na ľahkú demonštráciu potenciálu, ktorý profiler pri návrhu procesora prináša - ukazuje oblasti, ktoré môžu byť z hľadiska efektívnej realizácie kľúčové. V praxi by boli navrhované optimalizácie konfrontované napríklad s vygenerovaným RTL modelom (dopad špecializovaných inštrukcií na počet tranzistorov) alebo s požiadavkami na rozšíriteľnosť (nepoužité inštrukcie môžu byť v novšej verzii softvéru potrebné).

# Kapitola 7

# Testovanie

Súčasťou vývoja nového profilera bolo jeho priebežné testovanie. Ako základná testovacia platforma bolo zvolené procesorové jadro *Codix uRISC* vytvorené spoločnosťou *Codasip* a dodávané spolu s vývojovým prostredím *Codasip Studio* ako demonštračný projekt.

Otestovať bolo potrebné obidve časti profilera - zberač dát, ktorý je súčasťou simulátora, a analyzátor existujúci ako samostatný program - ako aj ich vzájomnú komunikáciu prostredníctvom interného binárneho profilovacieho záznamu. Prvotné testovanie bolo teda zamerané na overenie schopnosti zapisovať a čítať definovaný binárny formát.

Po implementovaní textovej formy výstupu bolo možné ručne kontrolovať výsledky - napríklad počet a hierarchiu volaní funkcií alebo počet vykonaní jednotlivých riadkov kódu (resp. konkrétnych inštrukcií) v prípade anotovaného zdrojového kódu. Pri testovaní grafu volaní funkcií bol ten istý testovací program preložený pre ASIP aj pre procesor počítača a výstup nového profilera bol potom porovnávaný s výstupom profilera *gprof*.

HTML výstup mohol byť následne porovnávaný s už overeným textovým výstupom. Navyše bolo potrebné manuálne skontrolovať interaktívne prvky - tlačidlá, zvýrazňovanie vybraných oblastí, zoraďovanie a filtrovanie dát v tabuľkách a pod.

Správnosť výstupu kompatibilného s projektom *OProfile* bola overená spustením programu *opannotate* nad týmto výstupom a následným porovnaním výsledného anotovaného zdrojového kódu s textovým anotovaným výstupom nového profileru. Pri tom bolo nutné pre program *opannotate* špecifikovať okrem preloženého programu a zdrojových kódov aj disassembler špecifický pre daný ASIP.

V rámci automatizácie testovania validity profilovacích výsledkov boli vytvorené skripty v jazyku bash a Python, ktoré okrem porovnávania výstupov s profilermi *OProfile* a *gprof* kontrolujú aj definované závislosti medzi jednotlivými hodnotami (napríklad súčty cyklov pri grafe volaní fukncií).

## 7.1 Výkonnostné testy

Z hľadiska výkonnosti bol sledovaný vplyv zberu profilovacích dát na dĺžku trvania simulácie. Použité boli modely *Codix uRISC* (Obr. 7.1), *Codix Helium* (Obr. 7.2) a *Codix Titanium* (Obr. 7.3), pričom pre každé jadro bol vygenerovaný simulátor z IA aj CA modelu bez profilera a s profilerom.[1]

Vo vygenerovaných simulátoroch boli spustené 3 testovacie aplikácie - *Bitcount* (počítanie bitov), *SHA* (výpočet hash hodnoty algoritmom SHA) a *Fibonacci* (výpočet členov

[1] https://www.codasip.com/products/cores/

Fibonacciho postupnosti) - a to so zapnutým aj vypnutým sledovaním zásobníka (*callstacku*).

Pre každú kombináciu parametrov (model procesora - aplikácia - úroveň profilovania) bolo spustených 40 simulácií a z dĺžok ich trvania boli vytvorené grafy (Obr. 7.1, 7.2 a 7.3), na ktorých je vidieť spomalenie simulácie spôsobené rôznou úrovňou (detailom) profilovania. Teda napríklad pre procesor *Codix uRISC* (Obr. 7.1) trvala simulácia CA modelu s aplikáciou *Bitcount* pri nízko úrovňovom profilovaní 2,1 krát dlhšie ako bez použitia profilera (2,1 násobné spomalenie). Pre vysoko úrovňovom profilovaní bolo spomalenie až 5,1 násobné.

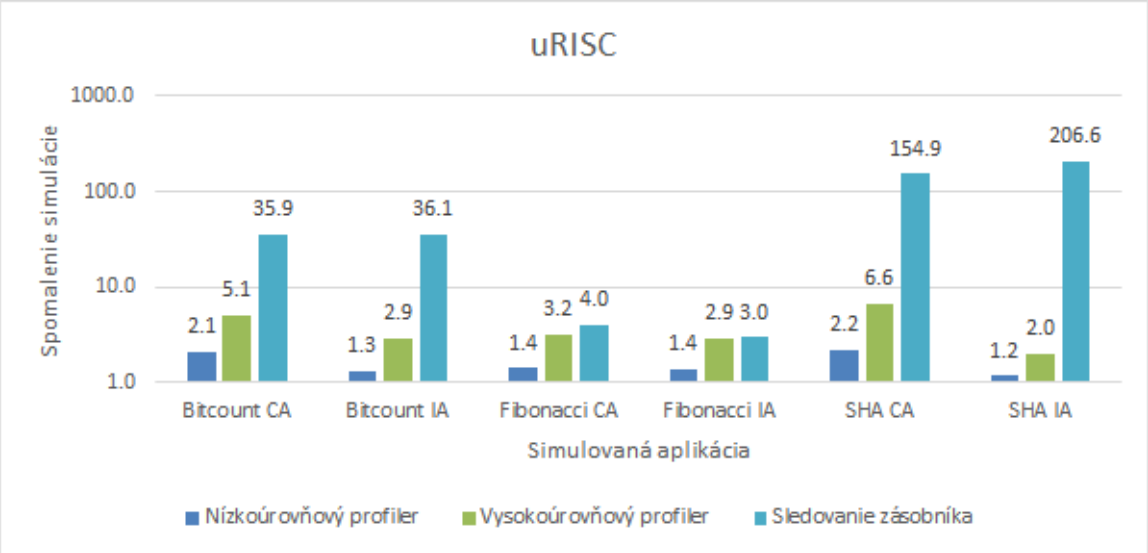
Nízko úrovňový profiler je využívaný pri interaktívnom ladení a slúži na priebežné počítanie zápisov a čítaní pre jednotlivé zdroje procesora. Pri normálnom behu simulácie neprodukuje žiadny profilovací záznam a nesleduje žiadne ďalšie štatistiky, preto je jeho vplyv na rýchlosť simulácie najnižší.

Vysoko úrovňový profiler sleduje všetky štatistiky popisované v práci okrem sledovania zásobníka. Jeho réžia je značne vyššia ako pri nízko úrovňovom profilovaní, avšak napriek množstvu sledovaných hodnôt je to vďaka optimalizovanému kódu vo výsledku len dvoj až trojnásobné spomalenie oproti nízko úrovňovému profileru.

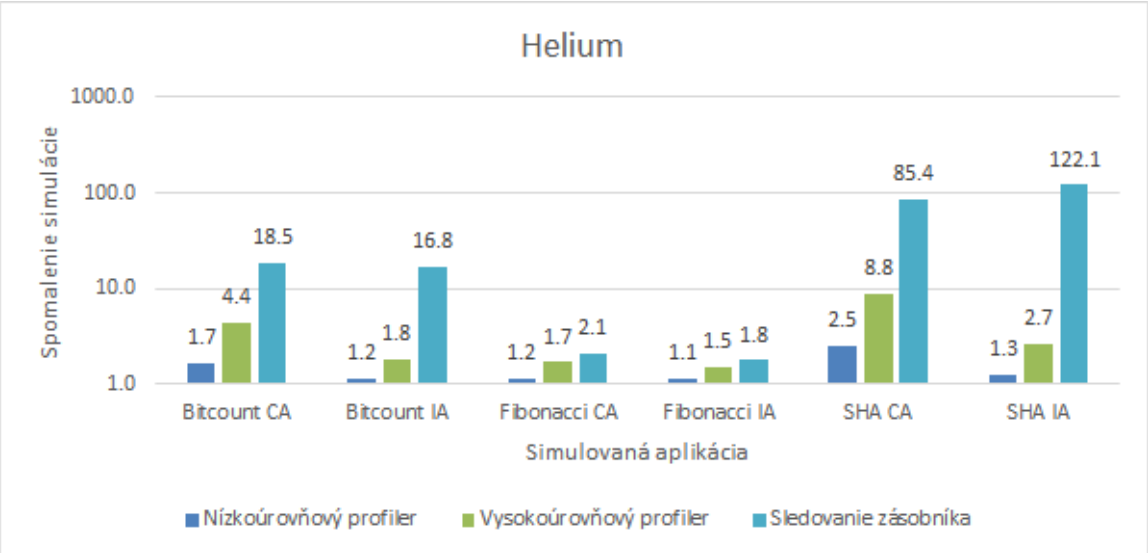
Na uvedených grafoch je vidieť drastický vplyv sledovania zásobníka na rýchlosť simulácie. To je spôsobené nutnosťou rozvinutia zásobníka v každom takte (resp. v každom momente vytvárania vzorky). Táto operácia je závislá od aktuálnej hĺbky zásobníka (počtu aktuálne rozpracovaných funckií), čo sa prejavilo na menšom spomalení v prípade aplikácie *Fibonacci* obsahujúcej jedinú funkciu - *main*.

Vo všeobecnosti trvá simulácia CA modelu dlhšie ako je to v prípade IA modelu. To je dané väčšou komplexnosťou a detailnosťou CA modelu, keďže sa narozdiel od IA uvažuje reálny počet taktov pre prístupy do pamäti, dekódovanie a vykonanie každej inštrukcie. S tým súvisí aj vyššia réžia profileru - je nutné spracovať viac udalostí, napríklad pozastavenie či vyprázdnenie zretazenej linky alebo úspešnosť vyrovnávacích pamätí. Taktiež sa sleduje pokrytie väčšieho objemu popisného kódu v jazyku CodAL. Preto je pre tú istú aplikáciu spomalenie spôsobené profilerom vyššie pre CA model a to na nízkej aj vysokej úrovni profilovania. Najvýraznejšie je to v prípade komplexného modelu *Titanium* (7.3) - pri simulácii aplikácie SHA na CA modeli s vysoko úrovňovým profilerom bolo spomalenie až 11 násobne vyššie oproti IA modelu.

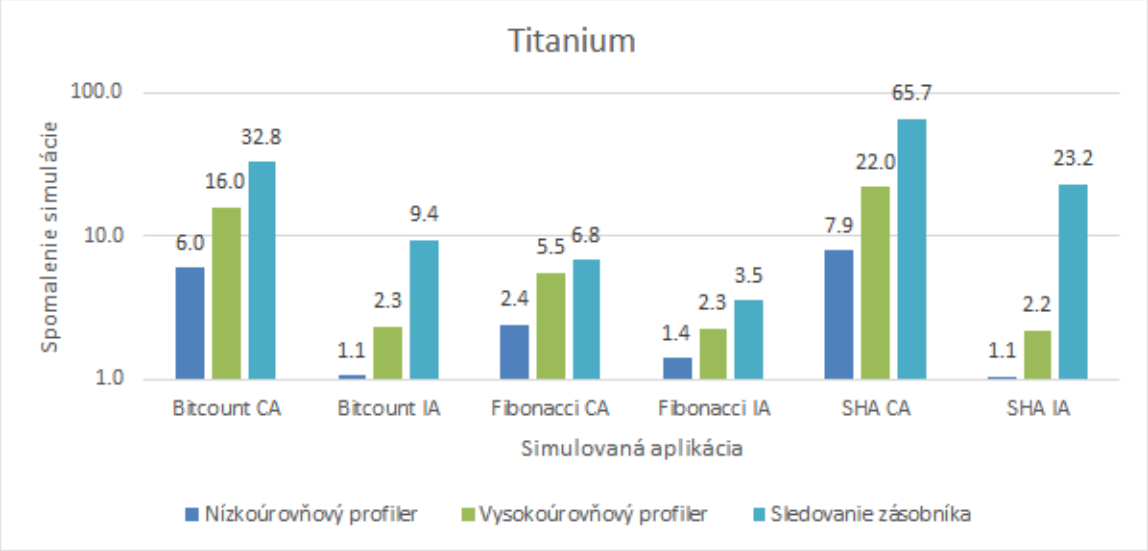
Priemerná chyba merania bola určená ako aritmetický priemer smerodajných odchýlok vypočítaných pre jednotlivé kombinácie parametrov. Jej hodnota bola približne 5%, pričom najvýraznejšia bola v prípade testovacieho programu *Fibonacci*, ktorý bol najkratší.




Obr. 7.1: Spomalenie simulácie vplyvom profilera - uRISC




Obr. 7.2: Spomalenie simulácie vplyvom profilera - Helium




Obr. 7.3: Spomalenie simulácie vplyvom profilera - Titanium

# Kapitola 8

# Návrhy na zlepšenie

### Analýza zásobníka

Počas testovania výsledného profilera sa zistilo výrazné spomalenie simulácie pri zapnutom sledovaní zásobníka (*callstacku*). Časovo najnáročnejšiu činnosť predstavuje analýza a rozvinutie rámcov jednotlivých volaní v momente vytvárania vzorky. Preto je v budúcnosti potrebné preskúmať možnosti optimalizácie, napríklad pamätanie si stavu zásobníka po dobu kým sa nezmení, čím by sa eliminovalo opakované rozvíjanie adries počas jedného volania.

Inou alternatívou by bolo zavedenie inšrumentačného profilovania, ako je to v prípade profileru *gprof*. To by však mohlo znamenať značné skreslenie výsledných dát - pridané inštrukcie by totiž ovplyvňovali napríklad počet cyklov, štatistiky dekodérov alebo aj pokrytie jazyka CodAL. Nevýhody by teda mohli prevýšiť potenciálny zisk na stane rýchlosti.

### Paralelizácia simulácie

Centralizovaná architektúra zberača dát o behu simulácie (5.2.3) bola navrhovaná hlavne s ohľadom na možnosť jednoduchej úpravy interného formátu profilovacieho záznamu. Jej použitie má však aj ďalšiu veľkú výhodu - vďaka existencii centrálneho prvku, ktorý riadi generovanie výstupu a synchronizáciu všetkých zberačov je možné v budúcnosti simulovať multiprocesorové platformy paralelne (v samostatných vláknach resp. procesoch). Úpravy profilera by boli minimálne - synchronizácia procesov by bola potrebná na jedinom mieste.

### Sekvencie inštrukcií

Z pohľadu optimalizácie inštrukčnej sady je výhodné sledovať často vykonávané sekvencie inštrukcií. Ich zaznamenávanie umožňuje už aktuálna verzia profileru. V budúcnosti sa však ponúka možnosť analyzovať v rámci sekvencie aj dátové závislosti medzi jednotlivými inštrukciami - ak 3 inštrukcie idúce za sebou pracujú s tým istým registrom, naznačuje to, že by mohlo byť výhodné nahradiť ich jednou komplexnou inštrukciou.

Keďže užívateľ má možnosť si navrhnúť vlastnú podobu jazyka symbolických inštrukcií, je problematické rozlíšiť operátory a operandy. Preto by bolo nutné zaviesť spôsob predávania šablón inštrukcií profileru, tak aby operandy (registre) dokázal identifikovať.

### C++ 11

V ďalšej verzii Codasip Studia sa počíta s použitím štandardu C++ 11. Prechod na túto verziu by mohol mať priaznivý vplyv na rýchlosť profilera, keďže prináša napríklad optimalizovanú asociatívnu dátovú štruktúru *unordered_map* alebo nový typ konštruktoru - *move constructor*.

# Kapitola 9

# Záver

Táto práca bola venovaná profilovaniu aplikačne špecifických procesorov. Hlavnou úlohou bolo navrhnúť a implementovať nový profiler pre firmu Codasip Ltd. pracujúci v rámci simulátora mikroprocesoru, ktorý by dokázal analyzovať program aj samotný procesor.

Pri návrhu profilera boli využité všeobecné teoretické poznatky z oblasti profilovania, pričom spôsob zobrazenia výsledných štatistík vychádzal čiastočne z existujúcich nástrojov, ktoré boli popísané v kapitole 4. Oproti softvérovým profilerom bola doplnená analýza využitia inštrukčnej sady, zdrojov procesora a pokrytia kódu behaviorálneho popisu v jazyku CodAL.

V praxi je nový profiler využiteľný okrem analýzy softvéru aj na analýzu samotného hardvéru procesora. To je užitočné nie len z hľadiska optimalizácie výsledného produktu - informácie o pokrytí kódu alebo o dekódovaných inštrukciách môžu byť využívané aj pri automatizovanej funkčnej verifikácii, kedy dochádza k simuláciám náhodne generovaných testovacích programov a je potrebné sledovať ich efektívnosť a účinnosť pri overovaní správnosti správania procesora.

Pri simulovaní programov na navrhnutom procesore je rýchlosť simulácie kritická. Preto bol kladený veľký dôraz na efektivitu všetkých operácií súvisiacich s profilovaním, konkrétne so zberom profilovacích dát počas trvania simulácie, keďže tieto operácie so sebou prinášajú zvýšenú réžiu v každom takte procesora. V kapitole 7.1 boli zhrnuté výsledky výkonnostných meraní odrážajúce vplyv profilera na rýchlosť simulácie.

V poslednej kapitole bolo spomenutých niekoľko návrhov na zlepšenie v budúcnosti. Pravdepodobne najvýraznejšou a najnaliehavejšou výzvou ostáva vyriešiť enormnú záťaž spojenú so sledovaním zásobníka, predstavujúcu v niektorých prípadoch až 200 násobné spomalenie simulácie. Je potrebné preskúmať možnosti optimalizácie tohto procesu napríklad elimináciou opakovaného rozvíjania tých istých adries počas jedného volania. To je už však nad rámec rozsahu tejto práce.

Pravdou ostáva, že profiler je použiteľný aj bez tejto optimalizácie, nakoľko je používateľovi ponechaná možnosť voľby vzorkovacej frekvencie. Stále je teda možné za cenu štatistických nepresností profilovať aj dlhé a komplikované programy.

Výsledkom tejto práce je funkčný profilovací nástroj, ktorý bol úspešne integrovaný do vývojovej platformy *Codasip Studio*. Dnes je súčasťou tohto komerčného produktu a okrem toho je využívaný aj pre interné potreby spoločnosti, pri vývoji vlastných procesorových jadier.

# Literatúra


[1] BETTINI, L.; CAPECCHI, S.; VENNERI, B. : Double dispatch in C++. *Software: Practice and Experience*, roč. 36, č. 6, 2006: s 581–613, ISSN 1097-024X, 10.1002/spe.709. Dostupné z: <http://dx.doi.org/10.1002/spe.709>

[2] BURSKY, D. : New Processor Core Options Try Some ARM Wrestling. 2013, [Online; navštívené 13.5.2016]. Dostupné z: <http://www.chipdesignmag.com/bursky/?p=113>

[3] Codasip : Codasip Studio. 2015, [Online; navštívené 10.3.2016]. Dostupné z: <http://www.codasip.com>

[4] FISHER, J. A. *Embedded computing: a VLIW approach to architecture, compilers and tools*. Boston: Morgan Kaufmann Publishers, 2005.

[5] GRAHAM, S. L.; KESSLER, P. B.; MCKUSICK, M. K. : gprof: a Call Graph Execution Profiler. 1982.

[6] IMAI, M.; TAKEUCHI, Y.; SAKANUSHI, K.; aj. : Advantage and Possibility of Application-domain Specific Instruction-set Processor (ASIP). *Information and Media Technologies*, roč. 5, č. 4, 2010: s 1064–1081, 10.11185/imt.5.1064.

[7] KARURI, K. *Application analysis tools for ASIP design application profiling and instruction-set customization*. New York: Springer, 2011. ISBN 978-1-4419-8255-1.

[8] KULKARNI, C.; GHEZ, C.; MIRANDA, M.; aj. : Cache Conscious Data Layout Organization for Conflict Miss Reduction in Embedded Multimedia Applications. *IEEE Trans. Comput.*, roč. 54, č. 1, Leden 2005: s 76–81, ISSN 0018-9340, 10.1109/TC.2005.2. Dostupné z: <http://dx.doi.org/10.1109/TC.2005.2>

[9] LIU, D. ASIP (Application Specific Instruction-set Processors) design. Oct 2009. 10.1109/ASICON.2009.5351271.

[10] MASOOD, F. : RISC and CISC. *CoRR*, roč. abs/1101.5364, 2011. Dostupné z: <http://arxiv.org/abs/1101.5364>

[11] OLIVIERA, B. C.; WANG, M.; GIBBONS, J. : The Visitor Pattern As a Reusable, Generic, Type-safe Component. *SIGPLAN Not.*, roč. 43, č. 10, Říjen 2008: s 439–456, ISSN 0362-1340, 10.1145/1449955.1449799. Dostupné z: <http://doi.acm.org/10.1145/1449955.1449799>

[12] PATEL, R.; RAJAWAT, A. : A Survey of Embedded Software Profiling Methodologies. *CoRR*, roč. abs/1312.2949, 2013. Dostupné z: <http://arxiv.org/abs/1312.2949>

[13] SHVETS, A. : Visitor Design Pattern. 2006, [Online; navštívené 14.4.2016]. Dostupné z: <https://sourcemaking.com/design_patterns/visitor>

[14] SKOGLUND, B. *Code profiling as a design tool for application specific instruction sets* 2007. 74 s. Vedoucí práce Dake Liu.