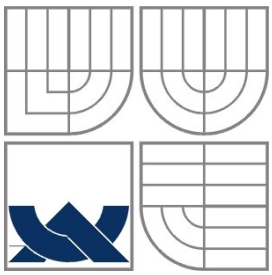
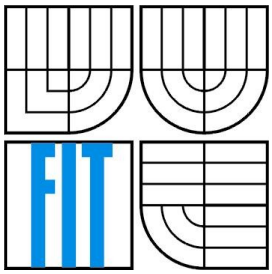


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



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

PROFILOVACÍ NÁSTROJ PRO GENERICKÉ SIMULÁTORY MIKROPROCESORŮ

PROFILING TOOL FOR GENERIC MICROPROCESSOR SIMULATORS

BAKALÁŘSKÁ PRÁCE
BACHELOR´S THESIS

AUTOR PRÁCE
AUTHOR
MILAN WILCZÁK

VEDOUCÍ PRÁCE
SUPERVISOR
ING. KAREL MASAŘÍK

BRNO 2007

Abstrakt

Tato práce popisuje návrh a implementaci profilovacího nástroje pro nově vytvářené procesory. K popisu procesoru je použit specializovaný jazyk ISAC, který popisuje zdroje procesoru, instrukční sadu a chování procesoru. Výsledný profiler je rozšířením existujícího simulátoru a lze ho použít pro optimalizaci procesoru i jeho aplikací. Základem je generování událostí simulátorem a jejich následné zpracování do statistik.

Klíčová slova

Profiler, mikroprocesor, Lissom, ISAC, vestavěný systém, simulace.

Abstract

This thesis presents design and implementation of profiling tool for newly created microprocessors. For microprocessor description specialized language ISAC is used which describes processor resources, instruction set and behavior. Final profiler is an extension to existing simulator and can be used for optimizing of both processor and its applications. Principle of the profiler is generating events by simulator and processing these events into statistics.

Keywords

Profiler, microprocessor, Lissom, ISAC, embedded system, simulation.

Citace

Milan Wilczák: Profilovací nástroj pro generické simulátory mikroprocesorů, bakalářská práce, Brno, FIT VUT v Brně, 2008

Profilovací nástroj pro generické simulátory mikroprocesorů

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením Ing. Karla Masaříka. Další informace mi poskytli Ing. Karel Masařík, Ing. Zdeněk Přikryl. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Milan Wilczák
12. 5. 2008

Obsah

1 Úvod

V dnešním světě se s mikroprocesory setkáváme na každém kroku a často si to ani neuvědomujeme. Většinou si pod mikroprocesorem představíme tu "věc" v počítači, která se stará o chod počítače a jejich výrobci se honí za vyššími frekvencemi a množstvím jader, nebo procesor v grafické kartě, který velkou rychlostí vyrábí 3D obrázky. Existují ale i mikroprocesory specializované, které nejsou tolik vidět. Obsahují je například chytré mobilní telefony, DVD přehrávače nebo různé automatické výrobní stroje.

Procesory se od sebe liší v různých směrech. Prvním z rozdílů je například instrukční sada, která se tradičně dělí na CISC a RISC. Dále to mohou být velikosti vyrovnávacích pamětí, zpracovávaný počet bitů nebo počet registrů. Kromě toho se liší mírou paralelismu zpracovávání instrukcí. Nejjednodušší procesory zpracovávají najednou pouze jednu instrukci, jiné používají pipeline, další to řeší dlouhým instrukčním slovem (VLIW).

U procesorů (a to nejen těch specializovaných) se hledí kromě výrobních nákladů a spotřeby na výkonnost v aplikacích, pro které jsou primárně určené. Zde nastupují profilery. Jedná se o nástroje, s jejichž pomocí lze určit slabá nebo často používaná místa a pak se na ně zaměřit při optimalizacích.

Cílem této práce je vytvořit nástroj pro analýzu výkonnosti procesorů, jejichž struktura a chování bude popsána jazykem ISAC, a aplikací běžících na těchto procesorech. Měl by se orientovat nejen tradičně na analýzu programů běžících na procesorech, ale i na procesory samotné, protože v těchto případech je možné zvýšit efektivitu aplikace vylepšením procesoru. Mezi možné statistiky patří například jak často se jaké zdroje procesoru používají, jaká část instrukční sady se používá a jaká se používá nejvíce.

Úkolem první kapitoly je seznámení s existujícími profilery (RealView profiler, GProf a OProfile). Druhá kapitola ukazuje současný stav simulátoru procesorů z projektu Lissom zapsaných jazykem ISAC, do kterého se bude profiler implementovat. Třetí kapitola představuje základní principy profileru. Čtvrtá kapitola se zaměřuje na popis implementace jednotlivých částí profileru.

2 Existující profilery

2.1 Principy profilerů

2.1.1 Co jsou profilery

Profilery jsou nástroje pro usnadnění optimalizace výkonu aplikací. Jejich úkolem je během vykonávání programu sbírat vhodné statistiky, z nichž lze pak vyčíst, které části kódu se často volají nebo se vykonávají významnou dobu. Tato místa jsou pak předmětem optimalizací. K získávání těchto statistik se používá instrumentace kódu, služby operačního systému pro zachytávání událostí (hooks), různé hardwarové prostředky, například přerušení nebo čítače výkonu (performance counter) a jiné metody (budou popsány dále).

Výstupem z profileru pak může být plochý profil (flat profile) nebo profil grafu volání (call-graph profile). Plochý profil ukazuje, kolikrát se jaká funkce programu zavolala a jak dlouho trvala všechna její vykonání celkem. Profil grafu volání tyto statistiky počítá i podle toho, odkud byla funkce zavolána a které funkce volala.

2.1.2 Metody sbírání informací

Jednou z metod sbírání statistik je sledování událostí v programu. Tuto metodu lze použít u aplikací běžících ve virtuálním prostředí, například .NET nebo Java. Díky tomu, že jednotlivé příkazy programu jsou spouštěny běhovým prostředím, je možné přesně sledovat, jaké funkce nebo metody se kdy volají a jaké proměnné se používají.

Druhou metodou je vzorkování. V pravidelných časových intervalech se zjišťuje hodnota programového čítače. Výhodou této metody je její rychlost, protože nezatěžuje procesor při každé instrukci, ale jen při výskytu časové události, takže program běží přibližně stejně rychle, jako by profilován nebyl. Nevýhodou této metody je její nepřesnost. Protože mezi jednotlivými vzorky se mohlo dít prakticky cokoliv, jsou výsledky pouhým statistickým odhadem. Tuto metodu používá například gprof nebo oprofile.

Třetí z používaných metod je instrumentace. Jedná se o úpravy kódu programu, často již ve zkompilované podobě. Instrumentace má podobnou sílu jako sledování událostí, ale protože se jedná o zásahy do kódu, můžou nastat nepřesnosti ve sledování výkonu nebo se mohou objevit chyby, které v původním programu nebyly nebo se naopak mohou některé chyby ztratit (heisenbug).

Metod instrumentace je více. Patří mezi ně instrumentace překladačem (používá ji gprof), úprava hotového binárního spustitelného souboru, instrumentace za běhu programu, spuštění programu pod hypervizorem nebo instrukčním simulátorem a manuální instrumentace zdrojového kódu (pro případ kdy profiler není po ruce nebo nedává dostatečně podrobné výsledky).

2.2 RealView Profiler

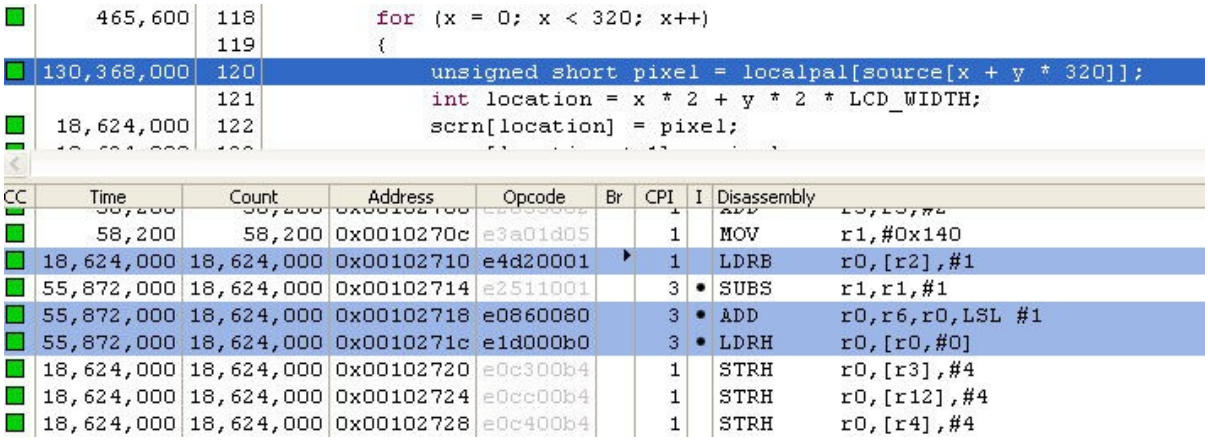
RealView Profiler je produkt specializovaný na profilování aplikací pro procesory ARM. Aplikaci je možné spustit buď ve virtuálním prostředí v simulátoru nebo přímo v procesoru ARM připojeného speciálním rozhraním, kdy lze profilovat aplikaci přímo na zařízení, pro které je aplikace určena.

Profiler se vyznačuje tím, že nepotřebuje ke své činnosti instrumentaci kódu aplikace. Jedná se o komerční produkt a v době psaní této práce neposkytoval školní licence, proto byly možnosti jeho studia silně omezeny pouze na volně dostupné materiály.

RealView Profiler umožňuje dlouhodobé sledování výkonu aplikace, sleduje pokrytí příkazů a větvení, počty a dobu vykonávání funkcí i jednotlivých instrukcí a umožňuje zobrazení těchto statistik v grafu volání. Díky podrobné znalosti procesorů ARM dokáže určit, jak dlouho by instrukce mohly ideálně trvat a informuje o datových konfliktech a čekání v pipeline.

Profiler používá ladící informace, například pro přiřazení příkazů C k instrukcím v assembleru nebo pro určení hranic funkcí a větví. Je možné zobrazit jak původní zdrojový kód, tak i zkompilovaný program a to zároveň se synchronizovaným označováním, tj. když se vybere řádek v C, označí se relevantní kód v assembleru.

Další ze schopností je výpis pěti nejnáročnějších funkcí podle času vykonání, zpoždění a přístupů do paměti.



Ilustrace 1: Profilovaný kód v RealView Profiler

2.3 GProf

2.3.1 Principy funkce

GProf je open-source program, který spolupracuje s překladačem. Tento profiler používá instrumentaci při překladu. Pro profilování programu je nutné přeložit ho a slinkovat pomocí GCC s parametrem -pg. Tím se řekne kompilátoru a linkeru, aby generoval do výsledného kódu volání funkcí z knihovny profileru, kterou pak linker přilinkuje a navíc se použije alternativní zaváděcí modul (tj. ten co volá funkci main).

Sběr profilovacích informací pak proběhne tak, že se program běžným způsobem spustí a po jeho úspěšném dokončení (tj. návratem z funkce main nebo voláním funkce exit) se vytvoří soubor gmon.dat se statistikami. Tyto statistiky se pak zpracují a zobrazí pomocí programu gprof.

Doba běhu je určována statisticky podle vzorkování programového čítače a informace tohoto typu jsou zatíženy statistickou chybou. Pro spolehlivější výsledky je dobré spustit profilování několikrát. Výsledkové soubory gmon.dat lze pomocí programu gprof sesumovat. Ostatní statistiky jsou určovány pomocí volání speciálních funkcí profileru a jsou tedy přesné.

2.3.2 Poskytované výsledky

Statistiky jsou implicitně sbírány pro jednotlivé funkce, ale při překladu je možné profiler nastavit tak, aby se statistiky daly zobrazit pro základní bloky případně i pro řádky programu.

Pomocí gprof je možné zobrazit plochý profil nebo profil grafu volání pro funkce nebo pro jednotlivé řádky programu a dále je možné zobrazit zdrojový kód obohacený o získané statistiky.

Plochý profil zobrazuje pro každou funkci její název a čas, po který funkce běžela, jednak včetně funkcí, které zavolala, a jednak bez jejich zahrnutí, tj. čas, po který běžel kód pouze této funkce. Dále je zobrazeno procento tohoto času z celkového času programu, počet volání funkce a průměrné časy jednoho vykonání funkce.

```
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
```

Graf volání je rozdělen do částí podle zobrazované funkce. U každé funkce je vidět její jméno, procento času stráveného ve funkci (včetně zavolaných) z celkového času programu, celkový čas strávený ve funkci včetně volaných funkcí a bez nich a nakonec ještě počet volání funkce. U každé funkce jsou zobrazeny tyto údaje ještě pro funkce z této volané a funkce, odkud je tato funkce volána. Časy a počty volání u těchto podřízených funkcí pak jsou v relaci k hlavní funkci, tj. zobrazuje se počet volání podřízené funkce z hlavní funkce apod.

```
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       1/1           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]
```

Při zobrazení statistik s rozlišením na úrovni řádků programu je výstup velmi podobný těm základním. Nejsou však zobrazeny statistiky pro všechny řádky, ale jen ty, do kterých se "trefilo" vzorkování programového čítače. O který řádek se jedná, je zobrazeno tak, že se za jméno funkce vloží do závorek název souboru a číslo řádku.

Posledním zobrazením z gprof je tzv. anotace zdrojového kódu, tj. výpis zdrojového kódu obohacený o získané statistiky. Na začátek každého vykonaného řádku programu je vložen počet vykonání tohoto řádku.

2.4 OProfile

2.4.1 Principy

OProfile je open-source aplikace, jejíž základem je modul linuxového jádra. Na rozdíl od jiných profilerů tento profiler nesleduje jen vybraný program, ale sleduje všechny procesy v systému. Jeho

principem je vzorkování programového čítače při různých uživatelem definovaných událostech a výsledkem jsou počty "zásahů" do jednotlivých adres v programu. Tyto události se generují na základě hardwarových čítačů výkonu (performance counters).

Profiler se ovládá pomocí příkazu opcontrol. Tento program slouží k inicializaci profileru, ke spuštění a ukončení profilování, k nastavení hloubky grafu volání a k nastavení událostí, které se budou používat pro vzorkování.

Pro každou sledovanou událost je třeba nastavit její typ a počet těchto událostí mezi vzorky. Menší vadou na kráse je, že pro logicky stejnou událost jsou často pro různé procesory různé názvy událostí. Mezi sledovatelné události patří například tik na sběrnici, tik RTC hodin, výpadek stránky, L2 cache miss, L2 cache hit nebo TLB miss.

2.4.2 Poskytované výsledky

K zobrazení výsledků se pak používá několik utilit, ale nejčastěji to jsou opreport a opannotate. V parametrech těchto příkazů se specifikuje filtr, které programy a události nás zajímají.

Program opreport zobrazuje výsledky v tabulce. Výsledky se spočítají buď podle programu, nebo podle názvu funkce. Při výpisu podle programu se kromě cesty k programu zobrazí počet vzorků a jejich procento z celkového počtu vzorků.

```
Counted CPU_CLK_UNHALTED events (clocks processor is not halted) with a unit mask of
0x00 (No unit mask) count 23150
   905898 59.7415 /usr/lib/gcc-lib/i386-redhat-linux/3.2/cc1plus
   214320 14.1338 /boot/2.6.0/vmlinux
   103450  6.8222 /lib/i686/libc-2.3.2.so
    60160  3.9674 /usr/local/bin/madplay
```

Při výpisu statistik podle symbolů se zobrazí název funkce, virtuální adresa jejího začátku, název programu, do kterého funkce patří, počet zásahů do této funkce a procento z celkového počtu.

```
Counted CPU_CLK_UNHALTED events (clocks processor is not halted) with a unit
mask of 0x00 (No unit mask) count 23150
vma      samples  %          image name               symbol name
0804be10 14971    28.1993    oprofiled                odb_insert
0804afdc 7144     13.4564    oprofiled                pop_buffer_value
c01daea0 6113     11.5144    vmlinux                  __copy_to_user_ll
0804b060 2816      5.3042    oprofiled                opd_put_sample
```

Program opannotate slouží k obohacení zdrojového kódu o výpis statistik. Pro jeho správnou funkci je třeba mu říct, kde zdrojové kódy najde. Navíc je možné kromě řádků kódu zobrazit i výpis v assembleru, který je pak mezi řádky kódu vložen. Před každý řádek, který má alespoň jeden vzorek se vloží počet těchto zásahů a procento z celkového počtu.

```
              :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 :}
```

3 Simulátor Lissom

3.1 Projekt Lissom

Projekt Lissom je zaměřen na výzkum jazyka pro popis mikroprocesorů a následné použití těchto znalostí k vytvoření pokročilé sady nástrojů, které by zefektivnily vývoj a v některých ohledech jej i zautomatizovaly.

Projekt je založen na několika starších projektech použitých pro vývoj mikroprocesorů v minulosti. Jejich použití ale bylo obtížné kvůli objevům nových mikroprocesorových architektur. Cílem projektu je vytvoření a implementace jazyka pro popis architektur mikroprocesorů. Jazyk se jmenuje ISAC, jeho autorem je Tomáš Hruška a vychází z podobného jazyka LISA. Pro dobrou použitelnost jazyka je potřeba vytvořit vývojové prostředí, které poskytuje nástroje pro vývoj programového vybavení a zároveň i hardwaru procesoru. Díky současné práci na hardware i software (hardware-software co-design) se celkový čas vývoje zmenší a zkrátí se i vývojový cyklus.

Projekt se skládá z několika částí. Práce v projektu je soustředěna na návrh konstrukcí popisného jazyka a na implementaci nástrojů pro vývoj software (kompilátor, assembler, disassembler, linker a simulátor).

3.2 Jazyk ISAC

Během vývoje software pro mikroprocesor jsou nutné některé podpůrné nástroje. Vývoj těchto podpůrných nástrojů je časově náročný a ve všech fázích vyžaduje spolupráci expertů. Obvykle je více týmů a každý z těchto týmů dělá samostatně jeden z nástrojů. Tento způsob odděleného vývoje ztěžuje pozdější integraci těchto nástrojů. Je jasné, že vývoj mikroprocesorů probíhá v mnoha iteracích. V každém případě je nutné upravit vytvořené nástroje, čímž se do nich zavlečou další chyby. Použití smíšeného jazyka pro popis architektur řeší problém špatné integrace nástrojů. Do této kategorie patří i jazyk ISAC.

Jazyk ISAC popisuje architekturu ve dvou oblastech. První je popis zdrojů mikroprocesoru. Mezi tyto zdroje se řadí registry (mezi nimi i je programový čítač), cache, paměti a navíc speciální zdroj pro mapování adresového prostoru na tyto zdroje. Druhá oblast popisuje, jaké operace v procesoru probíhají. Tyto operace se mohou sdružovat do skupin, mohou aktivovat jiné operace a můžou sloužit jako vzor pro dekódování instrukcí.

3.2.1 Popis zdrojů

Zdroje procesoru se definují v sekci RESOURCE. Registry se definují pomocí REGISTER a je možné u nich určit jejich bitovou šířku a dále specifikovat, zda se nejedná o nějaký speciální registr, jako například programový čítač. Základní paměť se definuje pomocí RAM, které se kromě datové šířky určí její velikost. Navíc lze určit i velikost bloku a informace o účelu (čtení, zápis, vykonání). Podporu pro zřetězené zpracování nabízí (kromě aktivačních sekcí) PIPELINE. U ní se pro každý stav určí jeho název a seznam registrů, které jsou v tomto stavu nastavovány. Adresový prostor je reprezentován zdrojem MEMORY_MAP. Ten je rozdělen do rozsahů a každému je přidělen zdroj, do kterého se daný rozsah mapuje.

```
RESOURCE
{
        PC REGISTER    bit[8] pc;
        REGISTER       bit[8] ax;
        ...
        RAM bit[8] data_mem   { SIZE (255); FLAGS (R, W); };
        MEMORY_MAP defaultmap
        {
                RANGE(0, 254)->program_mem [(7..0)];
                RANGE(255, 509)->data_mem [(7..0)];
        };
}
```

GROUP se používá k přiřazení operací ke skupinám. Používá se zejména pro dekódování instrukcí, kdy uvedení jména skupiny místo jména operace znamená výběr jedné z operací v této skupině. Výběr proběhne na základě sekce CODING u operací skupiny. Navíc jsou skupiny použity při podmíněném dekódování, kdy první dekódovaná skupina ve SWITCHi určuje, jaká větev se použije.

OPERATION symbolizuje operace. Pokud je operací myšleno část instrukce jsou jejím základem sekce ASSEMBLER (symbolický zápis instrukce) a CODING (zakódování instrukce do jednotlivých bitů). Operace mohou při svém provádění (tím se myslí i to, že jsou právě dekódovány) vykonat nějaké akce. Tyto akce jsou specifikování v sekcích BEHAVIOR, která obsahuje příkazy v jazyce C, nebo ACTIVATION, která může (i podmíněně pomocí IF a SWITCH) s volitelným zpožděním aktivovat jiné operace. Pokud je použita pipeline, je toto zpoždění (při neuvedení nulové) zvýšeno o zpoždění dané řetězovým zpracováním. Operace může být základem pro dekódování instrukcí pomocí sekce CODINGROOT.

```
OPERATION add
{
        INSTANCE ireg ALIAS { i };
        INSTANCE led1 ALIAS { led1 };
        ASSEMBLER { "add" i };
        CODING { 0b0 i };
        ACTIVATION{ led1; };
        BEHAVIOR{ printf("add[%d],",i); };
}
```

Sekce CODINGROOT obsahuje operace (resp. skupiny operací), které se použijí jako kořeny dekódovacího stromu. Sekce může obsahovat příkazy IF a SWITCH, a tím je možné dekódovat instrukce podmíněně. U každého kořene je kromě názvu skupiny uvedena adresa instrukce, která slouží především debuggeru pro účely breakpointů.

```
GROUP  GR1 = GR1A, GR1B, GR1C;
OPERATION decode
{
  INSTANCE GR1 ALIAS {gr1};
  INSTANCE OP1 ALIAS {op1};
  INSTANCE OP2 ALIAS {op2};
  CODINGROOT {
        SWITCH(gr1(defaultmap[pc]))
        {
                CASE GR1A: { op1(defaultmap[pc]); }
                CASE GR1B: { op2(defaultmap[pc]); }
        }
  };
}
```

3.3 Součásti simulátoru

Součásti simulátoru lze rozdělit do dvou skupin. První skupina jsou knihovny společné pro všechny simulátory, druhá skupina je generovaná pro každý model zvlášť nástrojem gentool. Ten kromě generování simulátoru generuje i další nástroje - assembler a disassembler.

3.3.1 Společné knihovny

Do první skupiny patří spouštěcí modul, modul debuggeru a knihovny zdrojů.

Spouštěcí modul je vstupním bodem simulátoru a přebírá parametry z příkazové řádky. Jeho cílem je inicializovat simulátor, zavolat modul debuggeru pro nahrání programu do paměti, spuštění vlastní simulace a nakonec předání výsledků.

Modul debuggeru má na starost podporu ladění programu. Umožňuje nastavení breakpointů a nahrání obrazu programu do paměti. Knihovna zdrojů má na starost definici chování zdrojů, řeší to pomocí tříd, případně pomocí maker u rychlé simulace.

3.3.2 Generovaná část simulátoru

Druhá skupina zahrnuje definici zdrojů procesoru, dekodér instrukcí a časovací jádro.

Definice zdrojů procesoru jsou řešeny tak, že pro každý zdroj procesoru se vygeneruje kód vytvářející globální proměnnou zvoleného typu (registr, paměť, atd.). Navíc se vygenerují ukazatele na implicitní programový čítač a implicitní mapu paměti. Ty jsou informacemi především pro debugger. Zvláštními zdroji jsou pipeline a mapa paměti, u kterých se generuje celá implementace. Mapa paměti se totiž skládá z částí ohraničených spodním a horním indexem a zdrojem, na který se tato část mapuje. Podobně pipeline má několik stupňů, jejichž počet se liší model od modelu.

Základem dekodérů instrukcí je párový automat, který se generuje pro každou kořenovou skupinu (případně operaci) pro dekódování. Ke každému přechodu je možné nastavit sémantické akce. Pro simulátor jsou to většinou kód sekce BEHAVIOR u dekódované operace a nastavení výběru z kořenové skupiny.

Jádrem simulátoru je časovací jádro. Odtud se (i nepřímo) volají všechny ostatní části simulátoru s výjimkou spouštěče. Základem je automat, u kterého je výběr nového stavu ovlivněn sémantickými akcemi. Pro každý stav existuje několik pravidel zadaných výchozím stavem, cílovým stavem a sémantickými akcemi. Sémantické akce mají svůj typ a hodnotu. Jedna ze sémantických akcí může mít úlohu podmínky uplatnění pravidla. Pro každý výchozí stav existuje právě jedno pravidlo bez této podmínky. Pro simulátor je běžnou sémantickou akcí spuštění sekce BEHAVIOR u operace zadané sémantickou akcí.

3.4 Vývojové prostředí

Vývojové prostředí představuje prezentační vrstvu projektu. Základem je obecné vývojové prostředí Eclipse a funkcionalitu pro projekt Lissom zajišťuje plugin Eclissom. Prostředí Eclipse je natolik univerzální, že jej zvolily i komerční produkty, jako například RealView Profiler.

Mezi vývojovým prostředím a nástroji pro vývoj existuje vrstva middleware. Jejím smyslem je co nejvíce zjednodušit plugin Eclissom tím, že se z něj přesune veškerá logika do této vrstvy. Použitím middleware se navíc zabezpečí, že budou vždy využity aktuální verze nástrojů. Komunikace

mezi middleware a pluginem se navíc stará o přenos potřebných souborů, protože samotné nástroje jsou vytvářeny bez síťové podpory (výjimkou je modul ladění v simulátoru). Vývojové prostředí pak umožňuje prostřednictvím střední vrstvy překlad modelu ISAC do interního formátu v XML, který je pak výchozím souborem pro generátor nástrojů, jako jsou assembler, disassembler a simulátor.

Simulace probíhá tak, že se po zkompilování a slinkování programu a vytvoření simulátoru vývojové prostředí přepne do režimu ladění, kde se spustí simulátor, počká se na jeho ukončení a poté se přenesou výsledky do IDE.

4 Principy profileru

4.1 Přehled principů

Profiler je založen na generování a zpracování událostí během zpracovávání instrukcí. Každá instance instrukce si s sebou nese svoji identifikaci po celou dobu svého vykonávání. V tomto případě se instancí instrukce myslí konkrétní jedno zpracování instrukce, tj. například v cyklu má na stejné adrese stejný kód je považován za různé instance. Smyslem je, aby profiler správně počítal statistiky při paralelním zpracování.

4.1.1 Instrukce a události

Jádrem získávání statistik jsou události. Tyto události symbolizují zahájení nebo dokončení instrukce, pokračování v přerušené instrukci nebo přístup ke zdrojům. Jako události jsou brány i dodatečné informace o instrukci, které ovšem nemusely být známy při jejím zahájení. Jedná se především o nastavení názvu instrukce a adresy umístění v paměti.

Aby je vývojářům procesoru nevnucovalo, jak přesně se instrukce musí zapsat, aby ji profiler bral jako instrukci, nejsou instrukce brány jako operace se sekcemi CODING, ASSEMBLER a BEHAVIOR, případně ACTIVATION a název by byl určen buď podle názvu operace, nebo podle začátku v sekci ASSEMBLER. Tento přístup by fungoval dostatečně dobře jen u jednoduchých modelů. Jako jméno instrukce bude považován průchod dekódovacím stromem.

Největším problémem je uchovávání identifikace instrukce. U jednoduchých modelů se zpracováním jedné instrukce najednou, případně jednoduchých procesoru s pipeline nebo VLIW by se dalo ID uchovávat v proměnných s tím, že u pipeline by se ID posunovalo k dalším fázím. Toto ale není možné aplikovat u procesorů, kde jedna operace může aktivovat jinou operaci (podmíněně a i do budoucna). Identifikace instrukce bude proto přiřazována instancím operací při jejich naplánování a při jejich následném spuštění bude identifikace instrukce k dispozici.

4.1.2 Realizace profileru

Profiler je integrován do simulátoru a proto je i podobným způsobem rozdělen do dvou částí. První část se bude specifická pro model a bude se generovat. Cílem je na vhodná místa simulátoru doplnit volání událostí z knihovny profileru. Druhá část je obecná, společná pro všechny modely. Jedná se o knihovnu profileru, která implementuje reakce na události a z těchto událostí počítá statistiky, které pak předá jako výsledek profilování.

Výsledky své práce nástroj poskytuje ve formě XML, která je pak odeslána prostřednictvím middleware do vývojového prostředí, které je jen zobrazí uživateli. Cílem je, aby XML dokument obsahoval všechny zobrazitelné údaje a vývojové prostředí neobsahovalo pokud možno žádnou logiku, nejlépe ani obyčejné sumace.

4.2 Pojmenování instrukcí

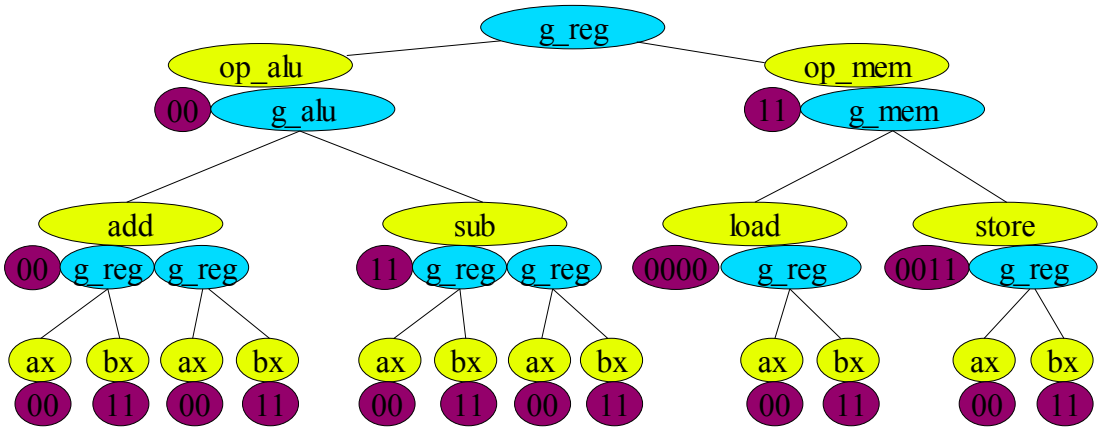
Pojmenování instrukcí je založeno na průchodu dekódovacím stromem při jejich dekódování. Výsledkem průchodu stromem je řetězec, který obsahuje názvy skupin a operací, skrz které se prošlo.

Při použití podmíněného dekódování nebo jiné složité sekce CODINGROOT jsou tyto průchody v nezměněné podobě vloženy do výsledného komplexního pojmenování. Průchod stromem je ve stylu pre-order, potomci jsou vloženi do závorek.

Při dekódování skupiny se ve výsledku objeví jediná členská operace. Při dekódování operace s instancemi jiných operací nebo skupin je možné, aby operace měla více potomků. V takovém případě jsou řetězce podstromů řazeny přímo za sebe, nejsou nijak speciálně odděleny. Aby mechanismus správně fungoval i při výskytu více listů vedle sebe, platí, že všechny listy mají za sebou závorky, které ohraničují prázdnou množinu potomků.

Pro vytvoření dekódovacího řetězce nejdříve z kompletního dekódovacího stromu vytvoříme úplně stejný strom s jediným rozdílem, že nový strom bude obsahovat jen operace a skupiny operací, kterými se při dekódování prošlo. Tento nový strom se pak serializuje za pomoci závorek.

Následující ilustrace ukazuje vytváření stromu pro řetězec z dekódovacího stromu pro instrukci zapsanou kódem 00000011. Použité uzly jsou zvýrazněny. Fialové bubliny znázorňují skupiny operací, modré bubliny operace. Červené bubliny obsahují kódování části instrukce a v novém stromu pak nejsou.



Ilustrace 2: Dekódovací strom

Ze zvýrazněné části stromu se vytvoří nový strom, který pak lze zapsat řetězcem instr_set(op_alu(g_alu(add(g_reg(ax())g_reg(bx()))))). Závorky ukazují sestup ve vytvořeném stromu, případně návrat do rodičovského uzlu. Pro názornost uvádím ještě několik dalších kódování instrukcí a zápisy jejich dekódovacích řetězců.

- 00000011 = instr_set(op_alu(g_alu(add(g_reg(ax())g_reg(bx()))))))
- 00111100 = instr_set(op_alu(g_alu(sub(g_reg(bx())g_reg(ax()))))))
- 11001111 = instr_set(op_mem(g_mem(store(bx()))))
- 11000000 = instr_set(op_mem(g_mem(load(ax()))))

Toto byl popis generování dekódovacího řetězce elementu dekódovací sekce. Kořenem celého dekódování je název operace obsahující dekódovací sekci. Dekódovací sekce ale nemusí obsahovat jen jeden element, může jich být za sebou více, v takovém případě se za sebe řadí i dekódovací řetězce těchto elementů. Například pro tuto CODINGROOT sekci:

```
OPERATION decode
{
    CODINGROOT
    {
        op1(memory[pc]);
        op2(memory[pc+1]);
        op3(memory[pc+2]);
    }
}
```

bude dekódovací řetězec vypadat: decode(<op1><op2><op3>), kde <op1> až <op3> se nahradí příslušnými dekódovacími řetězci odpovídajících skupin (nebo operací) op1 až op3. Stojí za pozornost, že jednotlivé řetězce lze od sebe poznat podle závorek, kořenem každého z podřetězců jsou názvy jejich kořenových skupin (příp. operací).

Speciálním elementem je příkaz SWITCH, pomocí kterého se tvoří podmíněná dekódovací sekce. Příkaz SWITCH pak vytváří další strukturu, která obsahuje identifikaci SWITCH (pomocí čísla), jeho podmínkový element, název splněného případu CASE a podřetězce dané splněným případem. Například takovýto zápis:

```
OPERATION decode
{
    CODINGROOT
    {
        SWITCH(op1[pc])
        {
            CASE mem:     op2(pc+1)
            CASE alu:     op3(pc+1)
            CASE control: op4(pc+1)
        }
    }
}
```

při splnění případu mem vygeneruje řetězec (zápisy <op1> a <op2> se nahradí podřetězci podle elementů): decode(switch_0(<op1>cases(mem(<op2>)))). Protože struktura SWITCH je sama elementem, lze do sebe větvení vnořovat.

4.3 Uchovávání identifikátoru instrukce

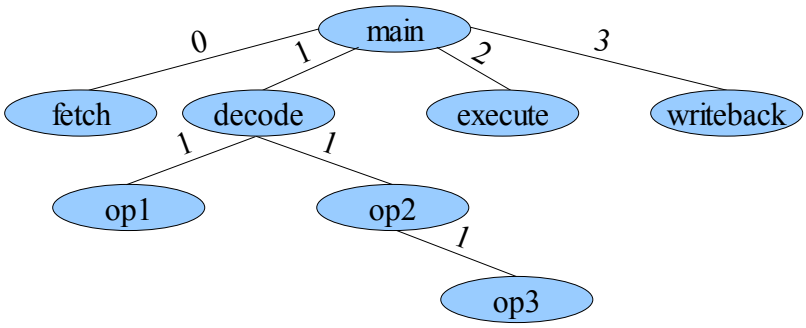
Pro správné profilování je třeba přiřadit události ke správným instrukcím. Instrukce může být rozdělena do více fází a to pomocí pipeline nebo pomocí aktivování operací jinými operacemi. Pro pipeline v modelech ISAC platí, že pipeline jen zvyšuje zpoždění aktivovaných operací.

Zpracování všech instrukcí začíná bez výjimky operací main. Při jejím startu se určí ID zahajované instrukce. Při naplánování další operace se tomuto naplánování předá i aktuální ID. Tímto postupným předáváním dojde k tomu, že celý strom dané instrukce bude mít společné ID, a protože se ID generuje při každém zahájení instrukce, je toto ID jednoznačným identifikátorem vykonávané instrukce.

Plánování aktivování operací tvoří strom. Teoreticky se může stát, že jednu operaci plánuje více operací, kvůli tomu se ve skutečnosti neplánují operace, ale jejich instance a tím se zajistí opět jednoznačné přidělování ID. Pro jednoduchost ale budu nadále používat pojmenování "operace". U plánování operací je možné nastavit zpoždění. Pokud se neuvede, předpokládá se nulové zpoždění. Zpoždění je navíc zvýšeno o zpoždění dané pipeline. I v případě, že máme jednoduchou pipeline, je nutné (většinou v operaci main) naplánovat operace představující jednotlivé fáze pipeline.

Pokud se veškerá plánování operací včetně celkového zpoždění (tím je myšleno explicitní zpoždění + zpoždění pipeline) zakreslí do grafu, vznikne hranově ohodnocený strom, kde ohodnocení

hrany je zpožděním. Pro jednoduchý model s pipeline a několika aktivacemi může graf časového modelu vypadat takto:



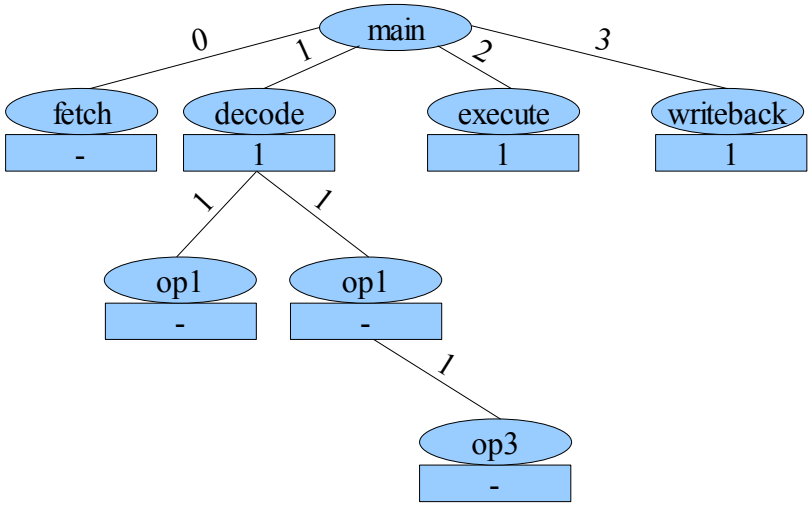
Ilustrace 3: Graf časového modelu

Plánování operací mohou být podmíněná. Z hlediska předávání ID to ale na věci nic nemění. Pokud se operace nenaplánuje, není důvod jí předávat ani ID. V tomto časovém modelu budou podmíněné aktivace operací op1 a op2.

Předávání ID operace je založeno na tom, že se každému uzlu časového stromu přiřadí fronta identifikátorů instrukcí. Při vykonávání operace se pak z fronty této operace použije první ID a toto se ze začátku fronty odebere. Do fronty se ID přidávají při naplánování příslušné operace.

Při inicializaci simulátoru (a profileru) jsou fronty u operací prázdné. Na začátku každého cyklu se zpracovává operace main. V každém kroku zpracovává jedna operace, obvykle v pořadí, v jakém byly naplánovány v rámci cyklu. Při zpracování se nejdříve z fronty aktuální operace odebere první ID a to se nastaví jako aktuální. Výjimkou z tohoto pravidla je operace main, která frontu nemá a aktuální ID si vždy při svém startu generuje nové. Poté se vykoná kód sekce BEHAVIOR. Na závěr se naplánují operace ze sekce ACTIVATION, některé podmíněně. Do fronty každé jiné naplánované operace se přidá aktuální ID. Do front operací, které naplánovány nakonec nejsou kvůli nesplněné podmínce, se nic nepřidává.

V našem příkladě jsou v prvním cyklu provedeny operace main a fetch, která byla operací main naplánována bez zpoždění. Graf časového modelu rozšířený o stav front jednotlivých operací na konci cyklu ukazuje ilustrace:



Ilustrace 4: Graf časového modelu se stavy front na konci prvního cyklu

Při dalších krocích a cyklech se dál postupně vykonávají a plánují operace. Průběh simulace jednotlivých kroků ukazuje následující tabulka. První sloupce ukazují, jaká operace se vykonává a s jakým ID, zbylé sloupce zobrazují obsahy front pro jednotlivé operace na konci daného kroku. Pro demonstraci podmíněného aktivování je pro první instrukci aktivována operace op2, pro druhou instrukci op1.

Aktuální		**Fronty u operací**						
operace	**ID**	**fetch**	**decode**	**execute**	**writeback**	**op1**	**op2**	**op3**
Cyklus 2								
main	2	2	1,2	1,2	1,2	-	-	-
decode	1	2	2	1,2	1,2	1	1	-
fetch	2	-	2	1,2	1,2	-	1	-
Cyklus 3								
main	3	3	2,3	1,2,3	1,2,3	-	1	-
execute	1	3	2,3	2,3	1,2,3	-	1	-
decode	2	3	3	2,3	1,2,3	2	1	-
op2	1	3	3	2,3	1,2,3	2	-	1
fetch	3	-	3	2,3	1,2,3	2	-	1

Tabulka 1: Vývoj obsahu front identifikátorů při simulaci

Časový model se při realizaci nakonec převede do automatu, kde kompletní stav všech aktivací je určen stavem automatu. Protože při zahrnutí i ID instrukcí by byl automat příliš složitý, jsou ID spravována samostatně a dynamicky se přidávají a ubírají. Samotný automat pak pouze určí, se kterou operací se právě pracuje a jaké operace se v aktuálním kroku plánují.

4.4 Události profileru

Profiler je založený na generování a zpracování událostí, které nastanou při simulaci, přičemž pro správnou funkci je třeba s nimi pracovat v pořadí, v jakém jsou generovány. Zpracování událostí ale nemusí nastat hned po vygenerování události, takže by teoreticky bylo možné nejdříve provést simulaci a generované události zaznamenávat a po dokončení simulace tyto události zpracovat a poskytnout výsledky.

Události lze rozdělit do dvou skupin. První skupina se stará o plánování instrukcí. Události z této skupiny jsou generovány v modulu časovacího jádra simulátoru. Druhou skupinu tvoří informace o vykonávané instrukci jako například přístup ke zdrojům nebo o jakou instrukci se jedná. Tato skupina událostí je generována z různých míst, například název instrukce je určen v dekodéru instrukcí, události využití zdrojů jsou generovány přímo zdroji, když se k nim přistoupí.

4.4.1 Plánování instrukcí

První událostí této kategorie a zároveň první událostí cyklu je zahájení instrukce. Nastává při spuštění operace main, ve které každá instrukce začíná. Při výskytu této události se vygeneruje nové ID, nastaví se jako aktuální. Tato událost zároveň znamená, že byl zahájen nový cyklus.

Druhá událost, spuštění naplánované operace, znamená pokračování ve vykonávání instrukce. Z fronty operace se první prvek a nastaví se jako aktuální ID. Na druhé straně je naplánování instrukce. Výsledkem této události je přidání aktuálního ID do fronty naplánované instrukce.

Jako poslední událost by se dalo brát dokončení instrukce. Protože zjišťování, která operace může být tou poslední, by bylo příliš náročné, je jednodušší použít počítání referencí na ID instrukce. Instrukce končí, pokud její ID není naplánované a operace s ID odpovídajícím této instrukci dokončila své zpracování. Při přidání ID do fronty se počet referencí zvýší, při odebrání z fronty a okamžitém nastavení jako aktuální ID se počet nemění, při nastavení aktuálního ID z tohoto ID se počet referencí snižuje.

4.4.2 Informace o instrukci

Tyto události pracují s instrukcí danou aktuálním ID a jejich cílem je shromažďování informací o vykonávané instrukci. Mezi tyto události patří přístup ke zdroji, zjištění adresy a názvu instrukce a zrušení efektu instrukce při spekulativním vykonávání.

Událost přístup ke zdroji generují samy zdroje. Adresa a název instrukce se nastavují částečně v dekodéru instrukcí (element CODINGROOT sekce) a částečně v časovacím jádru (adresa a podmíněná CODINGROOT sekce). Na základě těchto událostí se pak generují statistiky pro instrukce. Jedinou výjimkou je začátek instrukce, která je použita pro počet vykonání instrukce.

5 Implementace profileru

Implementace profiler vypadá tak, že jsem do modulů existujícího simulátoru doplnil potřebný zdrojový kód.

5.1 Generování dekódování

Dekódování se skládá ze dvou součástí. První součást tvoří dekódovací automaty. Druhou z nich je v generátoru časového jádra a stará se o generování kódu pro volání dekódovacích párových automatů.

Jejich cílem je předaný vstup dekódovat a vygenerovat část názvu instrukce. Jsou generovány pro každý kořenový prvek dekódovacího elementu typu operace nebo skupina. Jedním z jejich výsledků je identifikace prvku, který byl vybrán v nejvýše postavené skupině, kterou je obvykle kořen elementu.

Generování dekódování v časovém jádru je řešeno prakticky ve dvou funkcích, substcr_element a substcr_control. První z nich má za úkol generovat kód volající párový automat, druhá se stará o zpracování příkazu SWITCH v CODINGROOT sekci. Obě funkce se navzájem volají, protože SWITCH je vlastně speciálním druhem elementu. Kód generovaný těmito funkcemi je uložen pod generovaným názvem elementu nebo řídícího prvku a teprve při generování časovacího automatu je tento kód do automatu vložen podobně jako sekce BEHAVIOR u plánovaných operací nebo přímo kód přímých příkazů pro profiler.

Tvorba názvu instrukce je možná pouze ve funkci substcr_element nebo přímým příkazem vloženým do časovacího automatu. Funkce proto používají prefix a suffix, který daný element uvozuje nebo zakončuje a je vkládán do kódu pomocí substcr_element. V případě příkazu SWITCH se ovšem může stát, že se nevybere žádná větev. V takovém případě je nutné suffix doplnit jiným způsobem, a to vložením příkazu do časovacího automatu. O toto vložení se postará funkce substcr_suffix. Tvorba názvu není možná přímo v subscr_control, protože tato struktura v časovacím automatu není zastoupena a nebylo by tedy kam kód vložit

Kód generovaný pomocí substcr_element obsahuje kód pro debugger a volání párového automatu, který vygeneruje řetězec obsahující příslušnou část názvu instrukce. Kód pro profiler je až na konci vygenerovaného kódu, který nejdříve vloží případný prefix, poté řetězec vygenerovaný párovým automatem a nakonec případný sufix. Kromě toho navíc nastaví adresu instrukce.

Funkce substcr_control především nastavuje prefixy a suffixy. U rozhodovacího elementu SWITCHe je prefixem "switch_0(", kde 0 se nahradí pořadím SWITCHe v CODINGROOT sekci. Protože je to zaručeně první prefix v elementu, přidá se před něj prefix předaný v parametru funkce. Suffixem rozhodovacího elementu je "cases(". Hned poté následuje doplnění sémantické akce do časovacího automatu pomocí substcr_suffix, aby se zaručilo zakončení řetězce v případě, že se ve SWITCH nic nevybere. Tento sufix "))" je rozšířen o sufix z parametrů. Nakonec se zpracují větve, každá z nich začíná prefixem určeným podle názvu operace nebo skupiny u CASE. Poté se u každé zpracují všechny elementy ve větvi, přičemž prefix se vloží před první element, sufix (opět rozšířený) za poslední element.

Funkce substcr_suffix funguje tak, že nejdříve najde stavy automatu, kde se vyskytuje rozhodovací element určený předaným jménem a poté prochází sémantické akce pravidel

vycházejících z těchto stavů a do nepodmíněných přechodů vloží novou sémantickou akci, která přidá sufix do názvu instrukce.

5.2 Generování událostí

O generování událostí pro informace o instrukcích se z části postarala CODINGROOT sekce, konkrétně generuje události zjištění názvu instrukce a zjištění adresy instrukce. O generování informací o využití zdrojů se postarají samy zdroje. Buď při základní simulaci ve formě tříd (už se prakticky nepoužívá), nebo při rychlé simulaci pomocí maker (makra se zatím také příliš nepoužívají). Událost zrušení instrukce není generována automaticky, musí se zavolat ručně v BEHAVIOR sekci.

Události pro plánování operací jsou generovány pomocí sémantických akcí v časovacím automatu. Tyto sémantické akce jsou typu PROFILER_CODE a do výsledného kódu se zahrnou pouze při povoleném profilování. Obsahem těchto sémantických akcí je přímo kód, který se má do generovaného výsledku vložit. Zahájení instrukce se vkládá těsně před sémantickou operaci typu ACTDOP, jejímž obsahem je operace main. Vložený kód je:

```
profilerl_new_id();
```

Událost dokončení instrukce je vnitřně generována jádrem profileru, kde se pro zjištění této události používá počítání referencí. Spuštění naplánované instrukce je podobné, vkládá se rovněž před operaci typu ACTDOP, ale tentokrát není požadovaným obsahem "main", ale název operace definované v ISAC modelu. Není možné dělat prostý test na "main", protože ACTDOP může obsahovat i operace virtuálně vytvořené například kvůli substitucím v sekci CODINGROOT. Sémantická operace je pak ve tvaru

```
profilerl_set_operation("název operace");
```

Poslední událostí je naplánování operace. Tato plánování se vkládají do časovacího automatu na základě analýzy aktivační sekce právě spouštěné operace. Při výskytu akce ACTDOP se zjistí, zda má operace aktivační sekci. Pokud ne, nic se neděje. V opačném případě se pro operaci vytvoří strom plánování operací včetně podmínek aktivací, který představuje fragment časovacího automatu. Tento strom se pro operaci vytváří jen jednou a při příštím výskytu operace se použije již vytvořený strom. Časovací automat se pak doplní podle tohoto stromu, v automatu se hledají odpovídající přechody a jim se pak přiřadí plánování operací.

Podmíněné aktivace z CODINGROOT sekce je nutno řešit speciálním způsobem. V generování podle stromu se podmínky pro CODINGROOT sekci přeskakují. Pro zpracování v současném simulátoru jsou dvě základní možnosti. První možností je generovat plánování operací přímo v dekódovacím stromu, již se tam nastavují podmínky pro časovací automat. Druhou možností je podmínkám přiřadit seznam operací, které se mají aktivovat. Tyto podmínky jsou jediným způsobem komunikace mezi dekodérem a automatem, takže množinu operací určují jednoznačně.

Podmíněné aktivace mají v obou případech ještě jeden problém. V okamžiku "vykonávání" akce TRANSCOND nemusí být nutně nastaveno správné ID. Proto se při vybírání nového ID pomocí new_id nebo set_operation u operací s aktivační sekcí navíc vytvoří pro příslušnou akci ACTDOP akce set_plan, která si pamatuje ID instrukce podle zadaného čísla, které je jedinečné pro každou akci

ACTDOP s plánováním. Toto zapamatované ID se pak pomocí get_plan použije před plánováním u TRANSCOND. Za touto akcí se pak generují akce plan_operation.

5.3 Knihovna profileru

Tato knihovna se stará o zpracování událostí a generování statistik, které pak předá ve formě XML. Před použitím je třeba knihovnu inicializovat, tím se vytvoří objekt profileru, který je pak použit ve všech dalších operacích s knihovnou. Funkce knihovny samy o sobě nic nedělají, pouze předávají své parametry příslušným metodám třídy profileru.

Jádro profileru pro usnadnění své práce používá další třídy, které shromažďují informace o instrukcích, případně je vhodně organizují nebo sčítají. Hlavní třída potom pouze zajišťuje komunikaci mezi těmito moduly a generuje XML statistiky.

5.3.1 Instrukce

Základem profileru jsou informace o instrukcích. Ty jsou prakticky jen strukturou, které lze nastavovat a číst všechny prvky. Pro usnadnění práce generátoru názvů instrukcí má tato operace přidávací sémantiku. Tato třída má navíc metodu import, která slouží ke sčítání statistik z více instrukcí do jedné souhrnné instrukce. Součástí informací o instrukci je i využití zdrojů touto instrukcí. U indexovaných zdrojů (např. paměť) se počítají pouze souhrnné informace, ne pro každý index zvlášť.

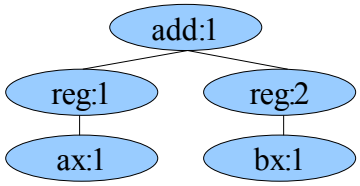
5.3.2 Tabulka běžících instrukcí

Tabulka slouží ke sledování běhu instrukcí. Primárně se stará o zpracování událostí typu plánování operací. Dále poskytuje přístup k datům instrukce, která je právě označena jako aktuální. Tabulka je indexována podle ID instrukce, funkcionalita front operací je realizována jinde.

5.3.3 Strom instrukcí

Strom obsahuje sumární informace o instrukcích, které jsou hierarchicky uspořádány podle názvů instrukcí importovaných do stromu. Položky na každé úrovni jsou identifikovány svým názvem, který odpovídá prvku názvu na příslušném místě v dekódovacím stromu. Protože některé instrukce mají ve stejné úrovni více instancí téže skupiny (operace), je třeba rozlišovat i pozici tohoto názvu v dekódovacím stromu. Celá identifikace uzlu stromu se pak skládá ze základního názvu a pozice. Položky, které mají stejnou identifikaci jsou sčítány, takže je identifikace v rámci nadřazeného uzlu unikátní.

Instrukce s názvem add(reg(ax())reg(bx())) je pak ve stromu reprezentována jako:



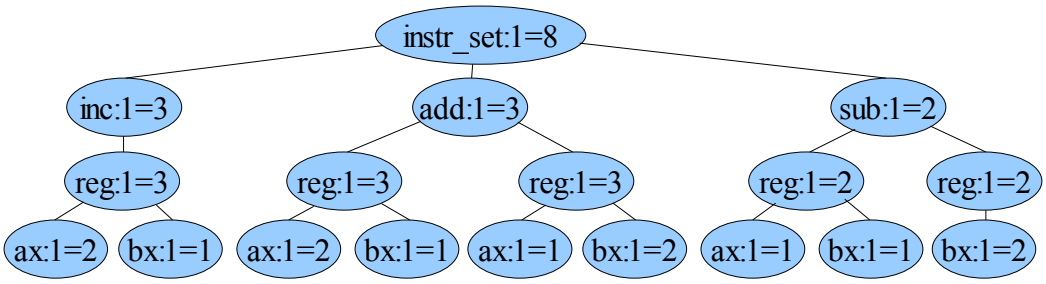
Ilustrace 5: Převod názvu do stromu instrukcí

Import probíhá tak, že se tímto stylem z názvu jakoby vytvoří nový strom a tyto dva stromu jsou pak uzel po uzlu sumovány. K sečtení uzlů dojde, pokud mají oba stejnou identifikaci, jinak se ve výsledném stromu objeví oba uzly každý s vlastními statistikami.

Pro názornost ukážu příklad výsledného stromu instrukcí. Do stromu instrukcí se importují instrukce:

```
inc ax
inc ax
inc bx
add ax, bx
add ax, ax
add bx, bx
sub ax, bx
sub bx, bx
```

Strom, kde v uzlu bude název, pozice parametru a počet dekódování, pak bude vypadat takto:



Ilustrace 6: Příklad kompletního stromu instrukcí

Tento styl sumování instrukcí má nevýhodu v tom, že v něm není možné rozlišit informace pro jednotlivé případy u víceparametrových instrukcí, například add(reg(ax())reg(bx())). Budou pouze vidět statistiky pro instrukce, kdy u add byl na prvním místě ax a na druhém cokoliv, nebo na prvním místě cokoliv a na druhém bx. Tento nedostatek se u instrukcí s jediným parametrem neprojeví.

5.3.4 Jádro profileru

Jádro profileru zpracovává všechny události a většinou je i posílá dál. Druhým úkolem jádra je generovat XML výstup ze získaných statistik.

Zahájení instrukce nejdříve aktuálnímu ID sníží počet referencí a pokud klesne na nulu, zavolá se událost konce instrukce. Poté se vygeneruje nové ID, které se pak společně s událostí předá tabulce běžících instrukcí, která tak nové ID nastaví jako aktuální a vytvoří prostor pro informace o instrukci. Událost konce instrukce je generovaná vnitřně jádrem profileru. Při jejím výskytu se aktuální instrukce naimportuje do stromu instrukcí a poté se událost předá tabulce instrukcí, kde se uvolní zdroje zabírané instrukcí.

Spuštění naplánované instrukce se na začátku zachová podobně, tj. sníží počet referencí a případně se generuje konec instrukce. Z fronty operace určené událostí se odebere první ID a tabulce instrukcí se předá událost spuštění, ovšem místo operace se použije vybrané ID, čímž se toto ID nastaví na aktuální. Naplánování operace přidá aktuální ID z tabulky instrukcí do fronty určené operací získanou z události a tomuto ID se zvýší počet referencí.

Pro plánování operací existují pomocné "události", které slouží k zapamatování ID pro vybranou sémantickou akci, která způsobí plánování operací. Při výskytu podmíněné sémantické akce se pak toto zapamatované ID použije. Smyslem je přiřazení správného ID k podmínkám.

Výstupní statistiky ve formátu XML obsahují především strom instrukcí. Dále se počítá pokrytí instrukční sady a počet vykonání na jednotlivých adresách, tj. klasické profilování.

```
<profiler>
  <resource_usage>
    <operations>
      <operation>
        <name>Příslušná část názvu instrukce</name>
        <executed>Počet vykonání instrukce</executed>
        <reads>Počet čtení instrukcí</reads>
        <writes>Počet zápisů instrukcí</writes>
        <resources>
          <resource>
            <name>Název zdroje</name>
            <reads>Počet čtení instrukcí z tohoto zdroje</reads>
            <writes>Počet zápisů instrukcí do tohoto zdroje</writes>
            <executes>Počet vykonání instrukce z tohoto zdroje</executes>
          </resource>
        </resources>
        <operations> ... </operations>
      </operation>
      ...
    </operations>
  </resource_usage>
  <coverage>
    <instructionset>
      <used>Počet použitých instrukcí</used>
      <total>Celkový počet instrukcí</total>
      <percent>Procento použitých instrukcí</percent>
    </instructionset>
  </coverage>
  <executions>
    <execution>
      <address>Adresa začátku instrukce</address>
      <count>Počet vykonání</count>
    </execution>
    ...
  </executions>
</profiler>
```

6 Závěr

Cílem této práce bylo navrhnout a implementovat profilovací nástroj pro generické procesory.

Při vývoji jsem zahodil několik nápadů, například původní pojmenování instrukcí stačilo pouze pro jednoduché procesory, kdy instrukcemi byly operace se sekcemi CODING, ASSEBMLER a BEHAVIOR.

Další schopností, kterou ovšem není zatím možné implementovat, jsou sledování spouštění funkcí v programu, protože chybí překladač jazyka C a ladící informace.

Vlastností, která byla implementována jen částečně, je sledování využití zdrojů, kde je problémem generování událostí. Třídní přístup už se nepoužívá, makra jsou použita pouze v mapě adresového prostoru a poslední možností jsou informace v hlavičce BEHAVIOR sekce, které jsou teprve navrhovány.

Hlavní cíl této práce, který se pravděpodobně příliš měnit nebude, je správné generování událostí ze simulátoru. Naopak velký prostor pro rozšíření je ve zpracování statistik získaných z těchto událostí.

Literatura

[1] Performance analysis [online]. [cit. 2. 5. 2008]. Dostupné na WWW: <http://en.wikipedia.org/wiki/Performance_analysis>

[2] RealView Profiler [online]. [cit. 2. 5. 2008]. Dostupné na WWW: <http://www.arm.com/products/DevTools/RVP.html>

[3] GNU gprof [online]. [cit. 2. 5. 2008]. Dostupné na WWW: <http://www.delorie.com/gnu/docs/binutils/gprof_toc.html>

[4] John Levon: OProfile manual [online]. [cit. 2. 5. 2008]. Dostupné na WWW: <http://oprofile.sourceforge.net/doc/index.html>

[5] Zdeněk Přikryl: Specifikace profileru (interní dokument)

Seznam příloh

Příloha 1. CD/DVD se zdrojovými texty profileru, technické zprávy a s programovou dokumentací.