VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

SIMULÁTOR PAMĚŤOVÉHO PODSYSTÉMU

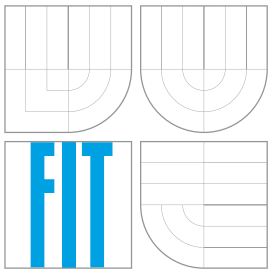
SEMESTRÁLNÍ PROJEKT
TERM PROJECT

AUTOR PRÁCE
AUTHOR

Bc. PETR HOLÁŠEK

BRNO 2013

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

SIMULÁTOR PAMĚŤOVÉHO PODSYSTÉMU

MEMORY SUBSYSTEM SIMULATOR

SEMESTRÁLNÍ PROJEKT
TERM PROJECT

AUTOR PRÁCE — Bc. PETR HOLÁŠEK
AUTHOR

VEDOUCÍ PRÁCE — Dr. Ing. PETR PERINGER
SUPERVISOR

BRNO 2013

Abstrakt

Tato práce popisuje problematiku zpracování paměťových stop a jejich využití v simulaci a vývoj modulárního simulátoru paměťového podsystému založeného na paměťových stopách. Simulátor podporuje také využití pro výukové účely díky vestavěné vizualizaci, pomocí které lze sledovat toky dat v paměťové hierarchii.

Abstract

This theses describes aspects of memory traces processing and its applications in simulation and development of modular memory subsystem simulator based on memory traces. The simulator also supports use for educational purposes thanks to built-in visualization which can be used to track data flows in memory subsystem hierarchy.

Klíčová slova

paměťový podsystém, paměťová stopa, simulace

Keywords

memory subsystem, memory trace, simulation

Citace

Petr Holášek: Simulátor paměťového podsystému, semestrální projekt, Brno, FIT VUT v Brně, 2013

Simulátor paměťového podsystému

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Dr. Ing. Petra Peringera. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Petr Holášek
19. února 2013

Poděkování

Děkuji vedoucímu práce Dr. Ing. Petru Peringerovi za veškeré připomínky, za pomoc při určování směru této práce a za poskytnutí dodatečných pramenů.

Obsah

Kapitola 1

Úvod

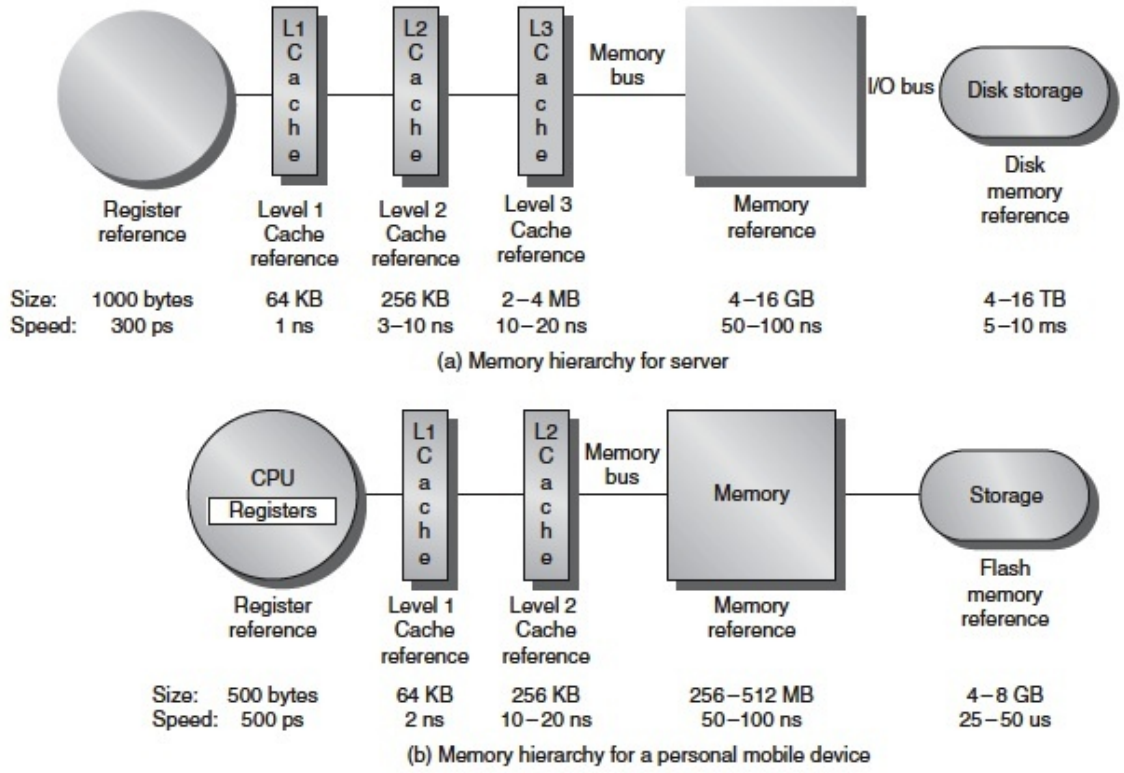
Ústředním tématem této diplomové práce je zkoumání možností simulace přístupů aplikací do paměti a následné vytvoření modulárního simulátoru použitelného i pro výukové účely. Napříč tomu, že přístup do paměti je téma poměrně široce zkoumané, hardware počítačů se neustále vyvíjí a navzdory zvyšujícím se rychlostem procesorů a kapacitám pamětí je také mnoho problémů a komplikací, které tento vývoj doprovází. Také v oblasti programování aplikací dochází k neustálým změnám, i když základní vzorce přístupů jsou stále stejné. V úvodu práce se věnuji popisu základních pojmů a komponent paměťové hierarchie a popisu získání, redukce a zpracování paměťových stop, které tvoří vstup simulátoru. V návrhu jsou diskutovány vhodné možnosti redukce paměťových stop a architektura simulátoru. V kapitole o implementaci je popsána implementace simulátoru a přidružené funkčnosti jako například extrakce paměťových stop z aplikací. V kapitole Experimenty jsou popsány experimenty a parametrické studie prováděné pomocí simulátoru. V závěru práce jsou zhodnoceny dosažené výsledky a možnosti dalšího postupu.

1.1 Paměťová hierarchie

Různé typy paměťových úložišť jsou definovány dvěmi hlavními vlastnosti *cenou* a *rychlostí*, které jsou zpravidla nepřímo úměrné. Pro přístupy do paměti programů běžících na procesoru platí následující dva pojmy *časová lokalita* a *prostorová lokalita*. *Časová lokalita* se zakládá na pravděpodobnosti, že adresa paměti, která byla použita, bude brzy použitá znova a bylo by tedy výhodné ji mít uloženou ve rychleji dostupné paměti. *Prostorová lokalita* znamená vysokou možnost, že budou v určitém časovém úseku použity sousední adresy. Pokud tedy program začíná přistupovat do paměti, jeho první přístupy směřují do úložišť umístěných v hierarchii nejvýše (pevný disk, páskové mechaniky, ...), poté již většina přístupů s využitím výše definovaných pojmů směřuje do rychlejších úložišť blíže procesoru (hlavní paměť, cache).

1.1.1 Typy pamětí

Během let se objevilo mnoho různých typů paměti, v dnešních počítačích se ale jedná téměř vždy o paměti SRAM *(Static Random Access Memory)* a DRAM *(Dynamic Random Access Memory)*. Liší se od sebe zejména rychlostí a cenami výroby a použití [5].



Obrázek 1.1: Paměťová hierarchie (zdroj: Hennessy, Patterson)

SRAM

Buňka paměti SRAM je tvořena šesti tranzistory. Pro udržení informace je třeba mít buňku stále připojenou k napájení, nicméně po přečtení obsahu buňky není třeba informaci obnovovat. Její hlavní nevýhoda je výrobní cena. Z tohoto důvodu je SRAM použitá pouze pro cache a registry CPU [7].

DRAM

Paměť DRAM potřebuje k uchování jednoho bitu informace pouze jeden tranzistor. V průběhu vývoje docházelo ke zvyšování kapacit a nastal problém s jednoduchým adresováním tak vysokého počtu buněk, který byl vyřešen maticovým uspořádáním. Půlka adresy se použije pro adresování sloupce, druhá půlka pro řádek. Hlavní nevýhoda DRAM spočívá v nutnosti stále obnovovat informaci zapsanou v buňce. Dle normy JEDEC (Joint Electron Device Engineering Council) je nutné znovu nabít DRAM buňku každých $64ms$ [5]. Tato činnost je automaticky vykonávána pomocí hardware obsaženém v paměťovém řadiči a před aplikacemi (i jádrem) je skryta. Paměť typu DRAM se používá v současné době v hlavní paměti počítače.

1.1.2 Cache

V průběhu vývoje procesorů a pamětí se začal zvětšovat rozdíl mezi jejich rychlostmi takovým způsobem, že paměťová sběrnice začala omezovat celkovou rychlost systému. Použití SRAM pro hlavní paměť by však bylo ekonomicky neúnosné, protože DRAM paměť

je řádově levnější. Z těchto důvodů se přistoupilo k zavedení SRAM cache *(vyrovnávací paměti)*, která omezuje nutnost přístupů do pomalejší hlavní paměti. Na běžných počítačích v současnosti je její velikost zhruba tisícinová oproti velikosti hlavní paměti. Důvodem, proč může být velikost cache o několik řádů menší než velikost hlavní paměti jsou *časová lokalita* a *prostorová lokalita*.

Asociativita

Asociativita cache udává počet položek na jednom řádku cache. *Plně-asociativní* cache obsahuje pouze jeden řádek, na němž jsou všechny prvky. Má tedy nejnižší *missRatio*, ale nejvyšší *hitTime* pro vyhledání prvku. Procesor tedy může do řádku umístit libovolnou paměť. *Přímo-mapovaná* cache má oproti tomu na každém řádku jen jeden prvek a každé místo v paměti musí být zařazeno do právě jednoho z řádků. Tento typ má však nejnižší čas vyhledání prvku *hitTime*. Kompromisem mezi těmito dvěma druhy jsou *n-cestné* cache, obsahující typicky na každém řádku n prvků.

Úrovně cache

Cache v moderních procesorech je víceúrovňová, což znamená, že existují typicky 3 úrovně [7] různých cache, které jsou se vzrůstající úrovní pomalejší a větší. Pokud procesor neuspěje ani v jedné z nich, přistupuje poté do hlavní paměti.

Režimy zápisu do cache

Při zápisu do cache můžou nastat tyto dvě možnosti:

write-through Data jsou zapsány zároveň do bloku cache i do bloku paměti na nižší úrovni.

write-back Data jsou zapsány pouze do bloku cache. Tento modifikovaný blok je zapsán do hlavní paměti až při jeho výměně.

Pro redukci zpětného zapisování bloků při *write-back* se používá *dirty* bit určující zda je blok modifikovaný nebo ne. *Write-back* pomáhá také zmenšit počet zápisů v paměťové hierarchii vzhledem k tomu, že ne všechny zápisy je do ní nakonec třeba zapisovat vede tedy i k energetickým úsporám a je tím zajímavý např. pro vestavěné systémy. *Write-through* je naopak jednodušší na implementaci než *write-back* a udržuje data stále koherentní [1], což je důležité například pro I/O operace nebo multiprocesory.
Pokud se zapisuje do bloku umístěného v paměťové hierarchii níže než v cache, jsou dvě možnosti chování při *writeMiss*:

Write-allocate Blok je načten do cache a poté zapisován, výpadek zápisu se tedy chová stejně jako výpadek čtení.

No-write-allocate Výpadek čtení nemá vliv na cache a zápis je proveden pouze do nižší vrsty paměťové hierarchie. Jde o méně obvyklou možnost.[7]

[1] ve shodě se stejným blokem paměti v různých vrstvách

1.1.3 Virtuální paměť

Virtuální pamětí nazýváme koncept, který umožní každému běžícímu procesu využívat logický lineární adresový prostor. Jde však jen o abstrakci založenou na faktu, že všechna paměť, která je k dispozici aplikaci, nemusí být nutně umístěna v hlavní paměti, nýbrž také na disku nebo jiných pomalejších úložištích. Tento mechanismus zajišťuje automatické řízení dvou vrstev paměťové hierarchie, *hlavní paměti* a *sekundárního úložiště*. Každá běžící aplikace používá lineární prostor *virtuálních adres*, které jsou bez účasti programátora automaticky mapovány do prostoru *fyzických adres* v *hlavní paměti*, popř.*sekundárního úložiště*. Tento děj je před programátorem aplikace skryt. Před zavedením virtuální paměti museli programátoři sami zajistit, aby paměťové požadavky aplikace nepřekročily dostupnou fyzickou paměť a paměť různých aplikací byla navzájem vyloučena. Zavedení virtuální paměti také velice zjednodušilo sdílení paměti mezi jednotlivými procesy a *relokaci* kódu, tedy běh stejného programu na různých adresách fyzické paměti. Systémy používající virtuální paměť mohou být rozděleny do dvou hlavních kategorií, používající *segmenty* nebo *stránky*.[7]

Segmentovaná virtuální paměť

Paměť je rozdělena do bloků proměnné velikosti nazývaných *segmenty*. Nejmenší možný segment má velikost 1 byte, největší se pohybuje mezi 2^{16} až 2^{32} v závislosti na typu procesoru. Adresování využívá dvě slova ve tvaru `<segment>:<offset>`. Mezi nevýhody patří hlavně složitost výměny bloků z důvodu nutnosti vyhledání spojitého volného prostoru v paměti o proměnných velikostech.

Stránkovaná virtuální paměť

Adresový prostor je rozdělen do bloků stejné velikosti nazývaných *stránky*. Pokud dotazovaná *stránka* není k dispozici v hlavní paměti, je vyvolán *výpadek stránky* (*page fault*) a operační systém přesune požadovanou stránku z disku do hlavní paměti. Po dobu tohoto přesunu obvykle jádro OS přepne provádění na jiný proces. Není-li v hlavní paměti místo na požadovanou stránku, je na základě pravidel určených OS vyřazena nejméně potřebná [2] stránka z hlavní paměti a přesunuta na disk.[7] Existuje také hybridní přístup, kdy uvnitř paměťového podsystému existuje více různých velikostí stránek (o velikostech druhých mocnin nejmenší velikosti stránky).[5]

TLB

Během překladu virtuálních adres na adresy fyzické je používána struktura nazývaná *tabulka stránek*, která obsahuje fyzické adresy seřazené v několikaúrovňové struktuře. Vyhledávání v ní je však pomalé, proto byla zavedena TLB (Translation lookaside buffer), obsahující určitý počet nejpoužívanějších překladů adres. Pokud daná TLB nepoužívá rozšířené tagy pro virtuální adresy, které označují ke kterému procesu (nebo jádru) patří, musí být TLB při každém přepnutí procesu vyprázdněna. To je také důvodem, proč mají TLB v dnešních procesorech řádově jen stovky záznamů, více se jich jednoduše vzhledem k přepínání procesů obvykle nepoužije. [5]

[2] podle stanovené politiky systému virtuální paměti

1.2 Paměťové stopy (memory traces)

Paměťová stopa je záznam o přístupech aplikace do paměti ve tvaru

```
I  39dae78fe0,2
 S 39db1b1400,4
I  39dae78fea,4
 L 39db1b13a8,8
I  39dae78fee,3
I  39dae78ff1,6
```

kde má každý řádek následující význam:

- `I addr,s` značí čtení instrukce z adresy `addr` a délce `s`
- `S addr,s` značí zápis dat na adresu `addr` a délce `s` prováděný poslední instrukcí
- `L addr,s` značí čtení dat z adresy `addr` a délce `s` prováděné poslední instrukcí

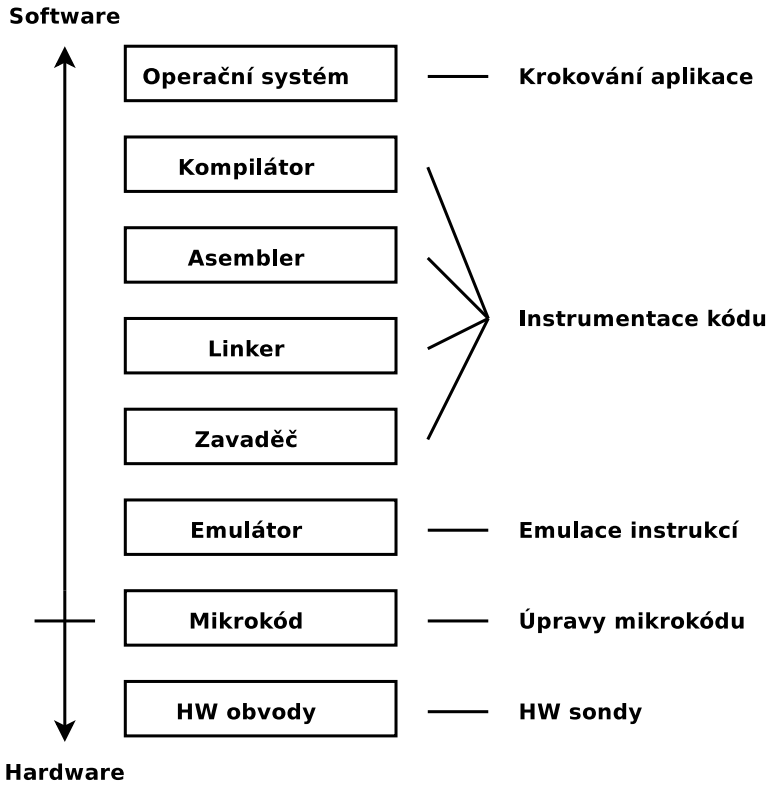
Pro dosažení přesné simulace by měla paměťová stopa obsahovat co nejpřesnější záznam přístupů aplikace do paměti systému a ideálně splňovat následující vlastnosti: *úplnost*, *podrobnost* a co nejméně *zkreslení*.
Úplnost znamená pokrytí všech přístupů do paměti, tedy nejen aplikací v uživatelském prostoru, ale také jádra OS a různých démonů běžících na pozadí.
Vhodně *podrobná* paměťová stopa by měla obsahovat kromě záznamů o přistupovaných adresách také informace navíc také stav tabulky stránek pro zachycení překladu adres virtuální paměti, označení přepnutí kontextu procesů, časovou značku, typ přístupu a velikost přistupované paměti. Pokud paměťová stopa obsahuje všechny tyto informace, lze ji považovat za ideálně *podrobnou*. Pokusem k ideální *podrobnosti* jsou WET (*Whole Execution Traces*), které reprezentují statickou reprezentaci aplikaci s označením dynamického chování každého příkazu. Příkaz je chápán buď na úrovni zdrojového kódu, mezikódu nebo asembleru. Dynamické chování označuje datové a řídící závislosti, časovou značku, údaje o překladu adresy a zpracovávané hodnoty.[14]
Paměťová stopa je *zkreslená* tehdy, obsahuje-li informace o přístupech neuspořádané nebo obsahuje-li přístupy navíc. Tyto problémy se obvykle netýkají paměťových stop jedné aplikace běžící v jednom procesu, ale spíše záznamu přístupů více aplikací najednou nebo operačního systému.[13]

1.2.1 Sběr paměťových stop

Přístupy do paměti mohou být zachytávány na různých úrovních systému.

Zachytávání pomocí HW sond

Sledování přístupů do paměti na paměťové sběrnici zaručuje *úplnost* paměťové stopy vzhledem k tomu, že snímací sondu nelze nijak fyzicky obejít. Je však nutné mít externí zachytávací zařízení s dostatečně velkým bufferem na výsledky. Pokud se buffer během sledování naplní (což je vysoce pravděpodobné), je nutné z něj data přesunout na větší záložní úložiště. Tento přesun obvykle způsobí určitou pauzu ve snímání a paměťová stopa je tedy určitým způsobem *zkreslená*. Další nevýhodou je také vyšší cena zařízení.[13]



Obrázek 1.2: Zachytávání přístupů do paměti na různých úrovních

Úpravy mikrokódu procesoru

Mikrokód procesoru je programovatelná volatilní paměť, do které lze ukládat programy v kódu atomických mikroinstrukcí, pomocí jejichž sekvencí jsou vykonávány standardní instrukce procesoru. I v současné době někteří výrobci [1] kontruují procesory jako RISC, aby s pomocí mikrokódu simulovali chování CISC. Je tedy možné do mikrokódu zabudovat pro instrukce přístupu do paměti dodatečný kód, který zapíše záznam o přístupu do externího bufferu podobně jako u HW sondy 1.2.1. Nevýhody tohoto přístupu jsou téměř nulová přenositelnost a typ zkreslení nazývaný *dilatace času* projevující se delším provádění instrukcí, což způsobí více přerušení časovače v jeho průběhu . To vyvolá častější volání kódu pro obsluhu přerušení časovače a dochází k abnormálnímu chování cache a TLB. Mikrokód také není schopen zachytit přístupy do paměti, které se vyhýbají procesoru, jako například DMA (*Direct memory access*).

Emulace instrukční sady

Emulace instrukční sady je svým přístupem podobná úpravě mikrokódu, ovšem s tím rozdílem, že k úpravě provádění instrukcí dochází na straně SW systému. Výhodou je vysoká flexibilita, kdy lze simulovat i běh více vláknových aplikací a emulovat přístupy do paměti i pro I/O podsystémy. [4] Nevýhoda spočívá ve velkém zpomalení provádění (obvykle v řádu několika stovek až tisíců procent) [13] a vysoké paměťové náročnosti.

Instrumentace kódu

Tento přístup spočívá ve statické úpravě programu před jeho během. Může probíhat na zdrojové, objektové nebo binární úrovni. Nejjednodušší přístup instrumentace na úrovni zdrojového kódu je prováděn během sestavovací a linkovací fáze kompilace, nevýhodou může však být nedostupnost zdrojového kódu aplikace. Mírně složitější je instrumentace na úrovni objektového kódu, vzhledem k tomu, že relokační tabulky a tabulky symbolů jsou stále přítomny. Nejsložitější, ale naopak nejjednodušší z hlediska uživatele, je instrumentace na úrovni spustitelných binárních souborů, zde je však třeba netriviálně zpětně doplnit tabulku symbolů a analýzu relokovaných symbolů. Mezi první široce použitelné aplikace patřilo Pixie[12] pro procesory MIPS. V současnosti lze pro po pokročilou instrumentaci kódu použít například framework Valgrind [10].

Krokování aplikace

Většina OS podporuje deguggery, pomocí kterých lze krokovat aplikaci po jednotlivých instrukcích a postupně sestavovat paměťovou stopu (např. aplikace `gdb` používající systémové volání `ptrace` vyskytující se v operačních systémech unixového základu). Výhodou tohoto přístupu je nízká náročnost, vysoká přenositelnost a jednoduchost použití. Nicméně tento přístup se dále nerozvíjel vzhledem k vyšší efektivitě výše popsaných přístupů [13].

1.2.2 Redukce paměťových stop

Velikost paměťových stop je pro většinu déle běžících aplikací velice rychle rostoucí a v průběhu času bylo vyvinuto mnoho metod, jak jejich velikost zredukovat. Kompletní paměťovou stopu lze bezztrátově zredukovat například při optimalizovaném generování pouhých diferencí v přistupovaných adresách a následnou komprimací. Tento přístup však vyžaduje čas na dekomprimaci a komprimační poměry nejsou tak vysoké jako u paměťových stop redukovaných ztrátově.[8] [13] Bezztrátová komprese má využití pokud je třeba mít zachyceny všechny reference a je třeba se vyhnout simulačním chybám. Principy ztrátové redukce se zaměřují na redukci, při které byly vynechány méně relevantní reference, které neměly velký vliv na na zkoumané děje v paměťovém podsystému. U tohoto přístupu však může simulace skončit chybou. Pro simulace virtuální paměti zaměřující se na stránkování, je základní metodou redukce *blocking*, spočívající ve sloučení referencí přistupující do jedné jednotky jako je blok cache nebo stránka. Tato redukce je však použitelná pouze k simulacím, kde nerozhoduje čas a pořadí přístupů.[8] Mezi další způsoby redukce patří například metoda *mazání zásobníku*, která vylučuje D prvních referencí na LRU zásobníku z důvodu, že jejich částečná redukce neovlivní stálé umístění v cache nebo v hlavní paměti. Dalším přístupem je samplování stopy buď v čase nebo prostoru, kdy jsou do stopy vybírány pouze reference objevující se v opakujících se ohraničených časových intervalech nebo rozmezích adres[11] [13].

1.2.3 Zpracování paměťových stop

Hlavní otázka v oblasti zpracování paměťových stop je zda zpracovávat při jednom průchodu paměťovou stopou všechny možné paměťové konfigurace, které chceme zkoumat, nebo se budou procházet a měřit postupně. Pro paralelní měření více konfigurací existuje například *zásobníkový algoritmus*, který umístí jednotlivé řádky zkoumané cache do zásobníku a udržuje si pole zásahů pro jednotlivé řádky. Úspěšně zasáhnutý řádek se poté přesunuje na

vrchol zásobníku a jeho místo se zaplní posunutím spodní části. Ze statistiky zásahů jednotlivých řádků je možné poté vypočítat počet zásahů a tedy *missRatio* pro jednotlivé plně asociativní cache s různými počty řádků.[13] Tento postup lze použít jen pro jednodušší konfigurace, kdy se sledují pouze cache o jediné úrovni. V ostatních případech se musí udržovat každý model paměťového podsystému, který je předmětem simulace, zvlášť.

1.3 Existující simulátory paměťových podsystémů

1.3.1 Cachegrind

Nástroj Cachegrind je součástí frameworku pro instrumentaci kódu Valgrind schopný simulovat instrukční a datové L1 cache a obecnou cache L2. Pro moderní procesory se třemi úrovněmi cache Cachegrind simuluje cache L1 a L3, nazvanou LL (*last-level*) cache. Důvodem výběru L1 je její obvykle nízká asociativita, lze tedy dobře sledovat situace, kdy je vykonáván kód špatně optimalizovaný pro přístup do cache a počet výpadků cache je vysoký. LL je vybrána z důvodu nejvýznamnějšího překrývání přístupů do hlavní paměti vzhledem k obvykle největší velikosti. Součásti funkcionality Cachegrindu je i profiler předvídání skoků, kterému se však v následujícímu textu nebudu věnovat.[10]
Zde lze vidět ukázkový výstup pro quicksort na poli náhodně generovaných 10000 prvku:

```
==9471== I   refs:      18,321,256
==9471== I1  misses:           814
==9471== LLi misses:           808
==9471== I1  miss rate:       0.00%
==9471== LLi miss rate:       0.00%
==9471==
==9471== D   refs:       6,219,587  (3,828,621 rd   + 2,390,966 wr)
==9471== D1  misses:         3,723  (    2,754 rd   +       969 wr)
==9471== LLd misses:         1,494  (      635 rd   +       859 wr)
==9471== D1  miss rate:        0.0% (      0.0%     +       0.0%  )
==9471== LLd miss rate:        0.0% (      0.0%     +       0.0%  )
==9471==
==9471== LL refs:            4,537  (    3,568 rd   +       969 wr)
==9471== LL misses:          2,302  (    1,443 rd   +       859 wr)
==9471== LL miss rate:         0.0% (      0.0%     +       0.0%  )
```

Narozdíl od běžného použití valgrindu, kdy se používá překlad s parametrem `-g`, je pro simulaci v Cachegrindu aplikace přeložena naopak i s optimalizačními přepínači z důvodu sledování chování reálného kódu. Pomocí přidruženého nástroje `cg_annotate` lze získat detailní statistiky o zásazích cache v rámci jednotlivých funkcí programu nebo dokonce jeho jednotlivých řádků. Na následujícím příkladu výstupu `cg_annotate` lze vidět, že Cachegrind správně diagnostikoval cache na testovacím stroji a jak přiřadil statistiky k jednotlivým řádkům kódu. Typy simulované cache lze také zadat libovolné jako parametry příkazového řádku.

```
I1 cache:         32768 B, 64 B, 8-way associative
D1 cache:         32768 B, 64 B, 8-way associative
LL cache:         2097152 B, 64 B, 8-way associative

   ...
```

```
    Dr  D1mr DLmr      Dw

  .     .    .       .  void quicksort(int list[],int m,int n)
     0     0    0  53,460  {
     .     .    .       .     int key,i,j,k;
 26,730    0    0       0     if( m < n)
     .     .    .       .     {
 13,364    0    0  13,364        k = choose_pivot(m,n);
 26,728    0    0   6,682        swap(&list[m],&list[k]);
 20,046    0    0   6,682        key = list[m];
  6,682    0    0   6,682        i = m+1;
  6,682    0    0   6,682        j = n;
  ...
```

Charakter simulace

Simulovaná cache je *write-allocate*, což znamená, že v případě *writeMiss* je zapisovaný blok načten také do cache. Jde o vlastnost většiny moderních cache. S instrukcemi měnícími obsah paměti je zacházeno stejně jako s instrukcemi pro čtení, což zjednodušuje implementaci. Je to možné vzhledem k faktu, že čtení původního obsahu zaručuje umístění modifikované paměti do cache. Dále je LL cache *inkluzivní* ve smyslu, že obsahuje všechny záznamy L1 a obě z nich používají algoritmus LRU pro nahrazování bloků. Pokud paměťová reference zasahuje přes dva bloky cache, nastává výpadek i pokud je jeden z nich v cache. [9]

Implementace simulátoru a instrumentace kódu

Během svého běhu Cachegrind uchovává 3 hlavní datové struktury:

Global Cache State obsahující stavy Li1, Ld1 a LL. Neobsahuje přímo jejich obsahy, které jsou pro určení zásahů nepotřebné, ale pouze adresy umístěných bloků.

Cost Centre Table je 3-úrovňová tabulka udržující statistiky pro každý zdrojový soubor, funkci a řádek kódu.

Instr-info Table je tabulka obsahující informace o každé prováděné instrukci neměnné v čase instrumentace. Jde o optimalizaci, aby při samotném provádění bylo nutné volat simulační funkce s co nejméně argumenty a zvýšila se rychlost provádění instrumentovaného programu.

Valgrind začíná zpracování všech binárních souborů nebo interpretovaných skriptů analýzou x86 instrukcí, jejich uložením do pole `instr_info` a převedením do RISC bytekódu *UCode*, který je dále zpracováván pluginem, v tomto případě Cachegrindem. Ten poté zařadí *UCode* instrukce do následujících kategorií a provede odpovídající instrumentaci původní instrukce `movl` přidáním *UCode* instrukcí ve tvaru:

```
0x3A965CEF: movl (%ecx),%eax

MOVL t0, t14   # přidáno uložení adresy
LDL (t0), t4
```

```
PUTL t4, %EAX
MOVL $0xB01E8748, t16  # přidáno uložení adresy instr_info pro daný řádek
CCALLo 0xB101992B(t16, t14) # přidáno volání C callbacku s uloženými adresami
INCEIPo $2
```

Nakonec se kód přeloží pomocí JIT (*Just-in-time*) zpět do x86 instrukcí a instrumentovaný kód je spuštěn ve stejném adresovém prostoru s `valgrindem` a použitým pluginem. [9]

Omezení simulace

- Valgrind není schopen instrumentovat jaderný kód, není tedy možné sledovat kód uvnitř systémových volání a jaderné obsluhy signálů.
- Neexistuje možnost sledovat paralelně běžící ostatní procesy na systému, takže nelze modelovat chování celého systému.
- Simulace probíhá pouze na virtuálních adresách, ačkoli na reálných cache se pracuje s fyzickými adresami.
- Rozvržení paměti je vlivem instrumentace kódu při simulaci v Cachegrindu odlišné od reálného, což může způsobovat určité zkreslení.
- Jediný podporovaný algoritmus nahrazování bloků v cache je LRU, ačkoli reálná cache může používat i jiný.

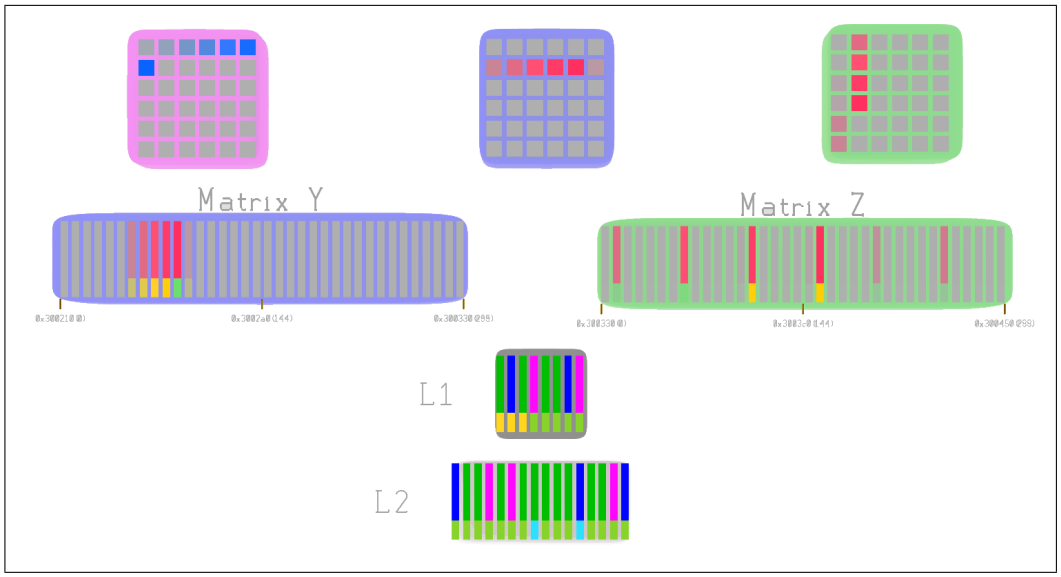
I přes výše popsané nevýhody lze pomocí Cachegrindu poměrně jednoduše najít v profilované aplikaci místa vhodná k další optimalizaci pro práci s cache a získat zajímavé statistiky o zkoumaném kódu. Díky jeho začlenění ve frameworku Valgrind je jeho použití velice jednoduché a plugin může být snadno upraven.

1.3.2 Memory Trace Visualizer (MTV)

Tento simulátor je zaměřen na simulaci a vizualizaci paměťových stop. Obsahuje vlastní simulátor cache postavený na instrumentaci kódu, podobně jako Cachegrind 1.3.1, která však je narozdíl od něj schopen simulovat děje v cache po krocích. MTV obsahuje funkci sledování "teploty"jednotlivých řádků podle počtu zásahů. Zkoumanou aplikaci lze také krokovat a vizualizovat obsah cache na jednotlivých řádcích. Vedle cache obsahuje simulátor také vizualizaci přístupových vzorů ve virtuálním paměťovém prostoru v čase. Simulátor je zaměřen hlavně na vizualizaci, neobsahuje tedy žádné obsáhlé statistiky, nebo parametrické studie.[2]

1.3.3 Dinero IV

Dinero IV je konfigurovatelný cache simulátor pracující s paměťovými stopami ve formátu pixie[12]. Oproti Cachegrindu1.3.1 podporuje až pět úrovní cache, kde každá z nich může mít jinak nastavené velikosti bloku a asociativity, algoritmy pro výměnu bloků, způsoby přednačítání instrukcí, pravděpodobnost selhání přednačítání a nastavení *write-allocate* a *write-back* politik1.1.2. Paměťová hierarchie jsou modelovány jako stromové struktury, kde jsou v listech umístěny zdroje paměťových referencí (procesory) a v kořenech hlavní paměti.



Obrázek 1.3: Násobení matic v aplikaci Memory Trace Visualizer

Simulace začíná umístěním všech referencí do nejvyšší úrovně cache[3], nižší úrovně jsou poté již plněny během simulace podle pořadí instrukcí a zvolené konfigurace. [6]. Dinero není zaměřen na simulaci v čase, jeho vstupem jsou pouze reference a výstupem statistiky o *missRate* a *hitRate* nakonfigurovaných cache. Nevýhodou Dinero je nutnost dodat paměťové reference a složitější ovládání, výhodou je velmi vysoká konfigurovatelnost. Ukázka výstupu:

```
l1-I/Dcaches
 Metrics              Total   Instrn   Data    Read   Write    Misc
 -----------------    ------  ------  ------  ------  ------  ------
 Demand Fetches       266310  189397   76913   70449    6456       8
  Fraction of total   1.0000  0.7112  0.2888  0.2645  0.0242  0.0000
 Prefetch Fetches     259846  189397   70449   70449       0       0
  Fraction            1.0000  0.7289  0.2711  0.2711  0.0000  0.0000
 Total Fetches        526156  378794  147362  140898    6456       8
  Fraction            1.0000  0.7199  0.2801  0.2678  0.0123  0.0000
 Demand Misses         52751     121   52630   49476    3151
  Demand miss rate    0.1981  0.0006  0.6843  0.7023  0.4881
   Compulsory misses    1748     110    1638      31    1604
   Capacity misses     10056       2   10054    9542     512
   Conflict misses     40947       9   40938   39903    1035
 Prefetch Misses       51568     602   50966   50966       0
  PF miss rate        0.1985  0.0032  0.7234  0.7234  0.0000
   PF compulsory misses  787     575     212     212       0
   PF capacity misses   9296       2    9294    9294       0
   PF conflict misses  41485      25   41460   41460       0
 Total Misses         104319     723  103596  100442    3151
  Total miss rate     0.1983  0.0019  0.7030  0.7129  0.4881
```

[3]Obvykle L2-L5

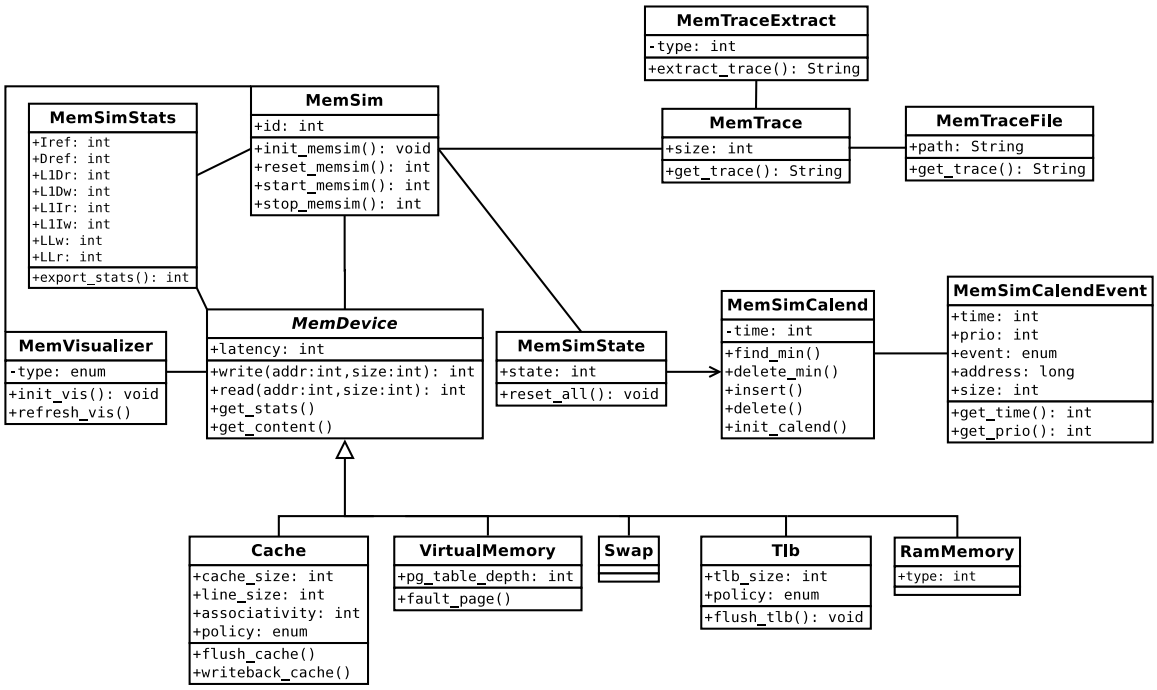
Kapitola 2

Návrh

Vstupem simulátoru je *paměťová stopa* uložená v souboru nebo binární soubor. Pokud je vstupem binární soubor obsahující debugovací informace, při volitelném dodání zdrojového souboru bude možné jednotlivé řádky krokovat a sledovat vizualizaci a statistické hodnoty v kontextu konkrétního řádku. Pro získání *paměťové stopy* je navrhováno použití frameworku Valgind, konkrétně pluginu `lackey`[10]. Simulátor paměťového podsystému bude implementován zřejmě také pomocí frameworku Valgrind. Důvodem je hlavně možnost provádění experimentu s binárními soubory a sledování vazby kódu na konkrétní řádky zdrojového kódu. Valgrind je pro instrumentaci tohoto typu velmi dobře vybaven a předejde se vymyšlení a implementaci nového nástroje pro instrumentaci kódu. Vzhledem k výukovému zaměření simulátoru bude jeho těžiště v možnosti konfigurace simulovaného podsystému a také ve vizualizaci, které budou vyžadovat také nezanedbatelné úsilí. Co se týká simulace cache, rád bych dosáhl flexibility nastavení simulátoru Dinero IV[6]. Při implementaci využiji také jeho stromovou abstrakci paměťového podsystému1.3.3. GUI simulátoru bude poskytovat všechny potřebné pohledy na simulaci najednou, tzn. paměťovou stopu, zdrojový kód programu, vizualizaci, informace o simulaci a nastavení parametrů simulace. K návrhu a implementaci GUI bude použit Qt4 Toolkit. Pro vizualizaci bude pravděpodobně využita třída Qt Painter a kromě tradičních způsobů vizualizace jako grafy nebo tabulky plánuji použít také alespoň jednu méně obvyklou metodu z [3]. Po dokončení první verze programu s funkční vizualizací a simulací paměti cache bych rád rozšířil možnosti simulace virtuální paměti a TLB. Toto rozšíření také cestu k automatickému provádění různých parametrických studií uvnitř simulátoru, ideálně propojených s vizualizačním modulem. V prvních fázích implementace bude VM vedena pouze jako modul v paměťové hierarchii poskytující zpoždění. Ve fázi experimentů by poté již bylo vhodné mít alespoň základní simulaci *stránkovací virtuální paměti* vzhledem k tomu, že u validace modelů bude největší problém se získáním hodnot z reálne cache. Dokonce to v současné době považuji za nemožné. Linuxové jádro naopak poskytuje poměrně dost informací o překladu stránek a časech. V poslední fázi práce na projektu bude vytvořena sada jednoduchých aplikaci a *paměťových stop*, které budou ve výchozím stavu dodávány s aplikací a použitelné pro výuku.

2.1 Objektový návrh simulátoru

Jádrem simulátoru je třída `MemSim`, pomocí které lze spouštět celou simulaci paměťové stopy. Paměťové stopy jsou načítány buď přímo ze vstupních souborů reprezentovaných třídou `MemTraceFile` nebo získány kompilací a analýzou zdrojového kódu pomocí instance



Obrázek 2.1: UML návrh jádra simulátoru

třídy `MemTraceExtract`. Poté jsou reprezentovány objektem třídy `MemTrace`. Modelovaný paměťový podsystém je reprezentován instancemi tříd `Cache`, `RamMemory`, `VirtualMemory`, `Swap` a `Tlb`, následníky abstraktní třídy `MemDevice`. Průběh simulace je řízen asociativní třídou `MemSimState`, pomocí které je přistupováno ke kalendáři událostí `MemSimCalend` a jsou postupně prováděny plánované akce reprezentované objekty třídy `MemSimCalendEvent`. Dále návrh obsahuje třídu `MemVisualizer` schopnou získat změny uvnitř podtříd `MemDevice` mezi jednotlivými událostmi kalendáře a na základě volání z GUI je vizualizovat. Třída `MemSimStats` obsahuje statistická data získaná během simulace na základě událostí kalendáře a paměťových zařízení v podtřídách `MemDevice`.

Kapitola 3

Závěr

V semestrální projektu byly popsány principy současných paměťových podsystémů a jejich vlastnosti. V úvodní části byly popsány součásti paměťové hierarchie s důrazem na paměť cache, jejiž modelování bude v implementaci důležitým prvkem. Dále byla popsány principy virtuální paměti, která bude ve výsledném simulátoru také hrát svoji roli. Následoval popis paměťových stop, způsoby jejich získání, redukce a zpracování. Úvod byl uzavřen porovnáním třech existujících simulátorů paměťového systému. V návrhové části byl představen UML diagram simulátoru a navrhované využití poznatků z úvodní kapitoly v simulátoru.

Literatura

[1] Linux* Processor Microcode Data File. 2013.
URL
http://downloadcenter.intel.com/Detail_Desc.aspx?lang=eng&DwnldID=14303

[2] Choudhury, A. I.; Potter, K. C.; Parker, S. G.: Interactive Visualization for Memory Reference Traces. *Computer Graphics Forum*, ročník 27, č. 3, May 2008: s. 815–822.

[3] Choudhury, I.: *Visualizing Program Memory Behavior Using Memory Reference Traces*. Dizertační práce, 2012.

[4] Cmelik, R. F.; Keppel, D.: Shade: A Fast Instruction-Set Simulator for Execution Profiling. Technická Zpráva SMLI 93-12, UWCSE 93-06-06, 1993.

[5] Drepper, U.: What Every Programmer Should Know About Memory. 2007.

[6] Edler, J.; Hill, M. D.: Dinero IV: trace-driven uniprocessor cache simulator. 2002.

[7] Hennessy, J.; Patterson, D.: *Computer Architecture: A Quantitative Approach*. The Morgan Kaufmann Series in Computer Architecture and Design, Elsevier Science, 2011, ISBN 9780123838728.
URL http://books.google.cz/books?id=v3-1hVwHnHwC

[8] Kaplan, S. F.; Smaragdakis, Y.; Wilson, P. R.: Trace reduction for virtual memory simulations. *SIGMETRICS Perform. Eval. Rev.*, ročník 27, č. 1, Květen 1999: s. 47–58, ISSN 0163-5999, doi:10.1145/301464.301479.
URL http://doi.acm.org/10.1145/301464.301479

[9] Nethercote, N.: *Dynamic binary analysis and instrumentation*. Dizertační práce, 2004.

[10] Nethercote, N.; Seward, J.: Valgrind: a framework for heavyweight dynamic binary instrumentation. *SIGPLAN Not.*, ročník 42, č. 6, Červen 2007: s. 89–100, ISSN 0362-1340, doi:10.1145/1273442.1250746.
URL http://doi.acm.org/10.1145/1273442.1250746

[11] Pleszkun, A.: Techniques for compressing program address traces. In *Microarchitecture, 1994. MICRO-27. Proceedings of the 27th Annual International Symposium on*, nov.-2 dec. 1994, ISSN 1072-4451, s. 32 – 39, doi:10.1109/MICRO.1994.717407.

[12] Smith, M.: *Tracing with pixie*. Computer Systems Laboratory, Stanford University, 1991.

[13] Uhlig, R. A.: Trap-driven Memory Simulation. 1995.

[14] Zhang, X.; Gupta, R.: Whole execution traces and their applications. *ACM Trans. Archit. Code Optim.*, ročník 2, č. 3, Září 2005: s. 301–334, ISSN 1544-3566, doi:10.1145/1089008.1089012.
URL http://doi.acm.org/10.1145/1089008.1089012