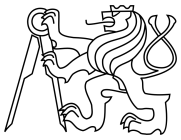
ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ



ZADÁNÍ DIPLOMOVÉ PRÁCE

Název:	Analyzátor činnosti cache v simulátoru GEM5
Student:	Bc. Jakub Pavčo
Vedoucí:	Ing. Jiří Kašpar
Studijní program:	Informatika
Studijní obor:	Počítačové systémy a sítě
Katedra:	Katedra počítačových systémů
Platnost zadání:	Do konce letního semestru 2016/17

Pokyny pro vypracování

Prozkoumejte možnosti trasování paměťových operací při simulaci výpadků v hierarchii pamětí cache v simulátoru GEM5. Doplňte trasovací data o data potřebná pro klasifikaci výpadků cache z hlediska typů sdílení mezi vlákny výpočtu (lokální vs. sdílené použití, jen čtení vs. čtení a zápis, počty sdílených čtení a zápisů). Vytvořte nástroj pro analýzu a klasifikaci přístupů do paměti z trasovacího logu simulátoru GEM5, soustřeďte se na analýzu režimu sdílení proměnných a odpovídajících cache bloků. Nástroj ověřte na běhu standardních benchmarků PARSEC a SPLASH v simulátoru GEM5 v režimu syscall emulation.

Seznam odborné literatury

Dodá vedoucí práce.

L.S.

prof. Ing. Róbert Lórencz, CSc.
vedoucí katedry

prof. Ing. Pavel Tvrdík, CSc.
děkan

V Praze dne 20. listopadu 2015

České vysoké učení technické v Praze
Fakulta informačních technologií
Katedra počítačových systémů

Diplomová práce

Analyzátor činnosti cache v simulátoru GEM5

Bc. Jakub Pavčo

Vedoucí práce: Ing. Jiří Kašpar

9. ledna 2017

Poděkování

Děkuji vedoucímu práce Ing. Jiřímu Kašparovi za vedení, cenné rady, pomoc s orientací v problematice a velkou ochotu mi pomoci. Děkuji rodině za podporu v průběhu studia a při tvorbě diplomové práci. Děkuji Kristýně Bucháčkové za korektury textu.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou, a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla, a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu), licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 9. ledna 2017 .....

Odkaz na tuto práci

Pavčo, Jakub. *Analyzátor činnosti cache v simulátoru GEM5.* Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2017.

Abstrakt

Práce se zabývá simulací činnosti paměti cache. Cílem práce je vyvinout analyzátor činnosti paměti cache a ověřit analyzátor na úlohách benchmarku PARSEC a SPLASH.

Práce se skládá ze čtyř kapitol. První kapitola se zabývá analýzou simulátoru gem5 a trasovacími možnostmi tohoto simulátoru. Jelikož trasovací možnosti nedostačovaly účelům diplomové práce, byly přidány nové trasovací příznaky. Druhá kapitola rozebírá využití adresního prostoru a analyzuje funkce ovlivňující adresní prostor. Třetí kapitola popisuje návrh struktur vyvíjeného analyzátoru. Součástí je podkapitola popisující proces získávání dat z mapy sestavení analyzovaného programu. Poslední kapitola se zabývá benchmarky PARSEC a SPLASH. Jsou zde analyzovány dvě úlohy benchmarku PARSEC.

Klíčová slova simulace, paměť cache, analýza paměti, alokace a dealokace paměti, gem5

Abstract

The thesis concernes simulation of operations cache memory. Goal of the thesis is development cache memory analyzer and verify it at bechmarks PARSEC and SPLASH programs.

The thesis contains four chapter. First chapter analyses simulator gem5 and its tracing possibilities. Second chaper analyses using of address space and analyses functions thats modify address space. Third chapter describes design of development analyzer. Process of getting informations from map file is explained in this chapter. Last chapter deal with benchmarks PARSEC and SPLAST. Two PARSEC programs are analysed here.

Keywords simulation, cache memory, analysis of memory, alocation and dealocation of memory, gem5

Obsah

Seznam obrázků

Seznam tabulek

Úvod

Počítač již využíváme a zdokonalujeme po desetiletí. V průběhu vývoje vznikal i software k simulaci počítače. V takovémto simulátoru lze jednodušeji ověřit, jak se vyvíjený software chová než při běhu reálného počítače. Simulací lze také otestovat chování softwaru na komponentách počítače, které zrovna nemáme dostupné.

Počítač je složen z komponent, které spolupracují. Je možné simulovat celý počítač nebo se zaměřit jen na určité části. Pokud se potřebujeme zabývat jednou částí počítače, lze jiné části v jisté míře zanedbat nebo úplně vynechat. Podle toho, co bylo třeba simulovat, se vyvíjely i simulátory. Existují simulátory simulující celý počítač od procesoru až po periferní zařízení, ale také simulátory zaměřené na jednu část počítače např. procesoru.

Jelikož má každý simulátor odlišný detail simulace jednotlivých součástí počítače, je třeba si vybrat simulátor, co nejvíce vyhovující účelu simulace. Pro analýzu činnosti cache pamětí je důležité zvolit simulátor, který bude přesně simulovat činnost paměťového subsystému. Naopak při analýze činnosti cache pamětí nebude třeba simulátor, který bude přesně simulovat činnost periferního zařízení. Dále jsou stručně popsány některé simulátory a zdůvodněn výběr vhodného simulátoru pro účely této práce.

SimNow

AMD SimNow simulator je kompatibilní x86 platformní simulátor pro rodinu procesorů AMD. Nabízí přesný model počítačového systému pro simulaci programu i operačního systému. Umožňuje rychlou simulaci celého počítačového systému i se standardními ladícími nástroji jako jsou break–pointy, náhledy do paměti a ladění po jednotlivých krocích. Simulátor dovoluje pracovat s BIOS, paměťovými parametry a více procesorovou simulací. [1]

Každý procesor obsahuje obousměrnou, 512 řádků dlouhou, 64B L1 cache. Při prohledávání technické dokumentace není zmínka o L2 cache. Ovladač

paměti zahrnuje dva kanály DDR, které jsou spojené v jedno 128b rozhraní. Paměťové zařízení neprodukuje logovací zprávy. [1]

Simulátor má dvě verze a to veřejnou a plnou. Veřejná se od plné liší omezením některých funkcí, či úplným znepřístupněním. [1]

SimOS

SimOS je prostředí pro studování hardwaru a softwaru moderních počítačových systémů. SimOS simuluje celý stroj. Simulace hardwaru počítače s dostatečnou rychlostí a detailem běhu standardních operačních systémů a aplikačních programů. Model simulátoru SimOS je navržen pro sbírání statistických údajů o chování simulátoru. SimOS obsahuje sofistikované mechanismy pro zjišťování těchto statistik pro specifikované skupiny jako například procesy nebo transakce. [2]

Jelikož v dnešní době rostou rozdíly mezi rychlostmi procesorů a paměti, stává se důležitější cache paměť. Simulátor umožňuje studovat výkon pro různé konfigurace cache pamětí. Simulátor umí simulovat 2 úrovňovou paměť cache, jejíž chování lze nastavit pomocí parametrů. Lze nastavit celkovou velikost, velikost řádku, stupeň asociace, čas nalezení i nenalezení hodnoty a to pro L1 instrukční i datovou cache i pro L2 datovou cache. [2]

OVPsim

OVPsim nabízí infrastrukturu pro popis platforem s jedním nebo více procesory obsahujícími sdílenou paměť a sběrnice u libovolné topologie a periferních zařízeních. OVPsim simulátor umí simulovat libovolný multiprocesor se sdílenou pamětí i heterogenní multiprocesorové platformy. OVPsim nabízí schopnost spojit se s externím ladícím nástrojem, který podporuje GNU GDB RSP protokol. [3]

Simulátor nabízí dva druhy paměťových modelů: plný a transparentní. Plný model implementuje paměť a využívá k tomu i komponenty jako jsou paměti cache. Tento model simuluje chování paměti počítače i s cache koherenčním protokolem. Transparentní model neimplementuje paměť jako v počítači, ani zde není použita cache paměť takže nemohou být modelovány koherenční protokoly. Požadavky na čtení a zápis jsou přesměrovány na model za pamětí, který jen uchovává data. Tento model lze využít jako velmi rychlý výkonnostní monitor. [4]

Simulátor je volně k užití pro nekomerční účely. [3]

gem5

gem5 simulátor je modulární platforma pro vývoj architektury počítačových systémů, zahrnující jak systémovou úroveň architektury, tak i mikro–architekturu procesoru. Tento simulátor vznikl spojením dvou simulátorů: M5 a GEMS.

Simulátor nabízí čtyři různé modely CPU, kde každý má unikátní vyvážení rychlosti vůči přesnosti. Všechny CPU modely mohou probíhat v jednom ze dvou módů: *System-call emulation* a *Full system*. Rozdíl je v tom, že v režimu *System-call emulation* jsou zařízení a systémová volání pouze emulovány. Simulátor podporuje množství architektur procesorů jako Aplha, ARM, MIPS, Power, SPARC a x86. [5]

gem5 simulátor obsahuje 2 různé modely paměťového systému: Klasický a Ruby. Klasický paměťový systém nabízí rychlý a snadno konfigurovatelný paměťový systém. Model paměťového systému Ruby nabízí flexibilní infrastrukturu zaměřenou na široké množství paměťových systémů. Ruby obsahuje cache paměti i různé koherenční protokoly. [6]

Výběr simulátoru

Simulátory SimNow a OPVsim sice nabízí pro veřejnost možnost bezplatného používání, ale software je proprietární. Pokud by nebyla v těchto verzích podpora získání všech potřebných informací o testovaném programu, tak nelze upravit trasovací výpis o potřebná data. SimOS je vyvinut na akademické půdě, ale podle internetové stránky tohoto softwaru [2] již 10 let není u tohoto simulátoru uvedena žádná vývojářská aktivita. Simulátor gem5 je nejlepší možnost díky vhodné úrovni simulace paměti s cache pamětí. Platforma je navržena modulárně, je tedy možné doplnit nebo navrhnout vlastní modul, který by pro potřeby diplomové práce chyběl. V případě, že by trasovací údaje nebyly dostatečné, poté je možné je doplnit, jelikož je software licencován pod opensource licencí.

Rozbor zadání diplomové práce

Podle první věty zadání: „Prozkoumejte možnosti trasování paměťových operací při simulaci výpadků v hierarchii pamětí cache v simulátoru gem5.“ je třeba porozumět, jak funguje simulátor gem5, pro porozumění paměťovému systému, který má být trasován. Dále prozkoumat trasování simulátoru a zaměřit se na paměťový systém. Jelikož je potřeba detailního simulování chování cache pamětí, klasický paměťový systém nebude stačit a musím použít paměťového systému Ruby.

Další částí zadání je: „Doplňte trasovací data o data potřebná pro klasifikaci výpadků cache z hlediska typů sdílení mezi vlákny výpočtu (lokální vs. sdílené použití, jen čtení vs. čtení a zápis, počty sdílených čtení a zápisů).“. Již dříve analyzované trasovací možnosti simulátoru rozšířím o potřebné údaje k analýze činnosti cache. Simulátor umožňuje definovat vlastní příznak k trasování, který lze doplnit do kódu simulátoru.

V zadání následuje: „Vytvořte nástroj pro analýzu a klasifikaci přístupů do paměti z trasovacího logu simulátoru gem5, soustřeďte se na analýzu režimu sdílení proměnných a odpovídajících cache bloků.“. Implementuji program,

který z připraveného trasovacího výstupu naplní struktury jednotlivých částí paměti. V těchto strukturách bude uvedeno, jak se přistupovalo k jednotlivým částem paměti a co se dělo s cache bloky v průběhu vykonávání programu. Následovat bude analýza struktur, jejímž vyhodnocením bude klasifikace výpadků cache z hlediska typů sdílení.

Poslední část zadání: „Nástroj ověřte na běhu standardních benchmarků PARSEC a SPLASH v simulátoru gem5 v režimu syscall emulation.“ referuje, abych otestoval vyvinutý nástroj na běhu benchmarků PARSEC a SPLASH. Simulátor v režimu syscall emulation umí spouštět pouze staticky slinkované programy. Programy určené k analýze bude tedy třeba staticky zkompilovat. Při kompilaci bude třeba i nechat kompilátor vytvořit mapu sestavení. Z této mapy poté analyzátor načte rozložení paměti analyzovaného programu.

Kapitola 1

Analýza funkcí simulátoru gem5

Kapitola obsahuje popis základních vlastností simulátoru gem5. V první sekci je popsána simulace procesoru, režimy simulace, paměťové systémy a doménově specifický jazyk SLICC. V další sekci je podrobněji popsán paměťový systém Ruby, který je stěžejní pro analýzu činnosti cache pamětí. V poslední sekci je analýza trasovacích možností simulátoru a rozšíření trasování o potřebné údaje.

1.1 Popis simulátoru

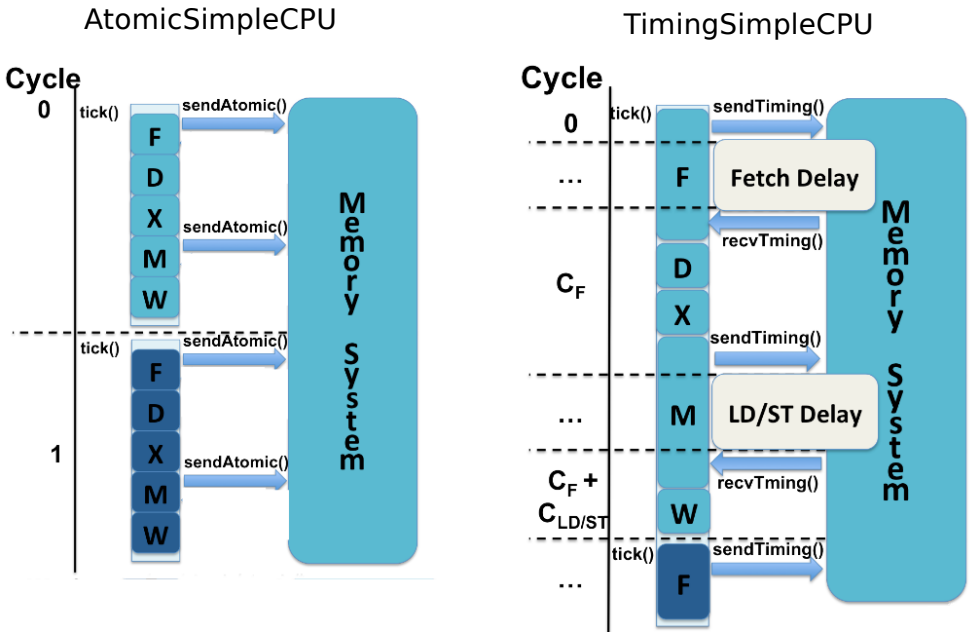
Na vývoji simulátoru gem5 se podílely akademické instituce i komerční firmy, jako například AMD, ARM, HP, MIPS, Princeton, MIT, aj.

Tento simulátor vznikl spojením systémového simulátoru M5 a simulátoru paměťového systému GEMS. Simulátor M5 nabídl vysoce konfigurovatelný simulační framework, mnoho ISA a rozličné CPU modely. Ze simulátoru GEMS je převzat detailní a flexibilní paměťový systém, který zahrnuje podporu mnoha cache koherenčních protokolů.

1.1.1 Modely CPU

Simulátor gem5 nabízí celkem čtyři modely CPU ze tří kategorií. Modely jsou od nejjednoduššího a nejrychlejšího po nejsložitější, ale také nejpomalejší. Díky tomu si každý může zvolit jemu odpovídající model, který bude vyhovovat účelu simulace, ale nebude zbytečně zpomalovat simulaci kvůli detailům, které nejsou sledovány.

Pokud není vhodný ani jeden model CPU, lze vyvinout vlastní model CPU. Vlastní model CPU lze otestovat nástrojem pro kontrolu funkčnosti CPU, ale jen pro jedno jádro a jedno vlákno.



Obrázek 1.1: Obrázek s modely SimpleCPU zobrazující stavy pipeline při vykonávání instrukce (zdroj: [7])

1.1.1.1 SimpleCPU model

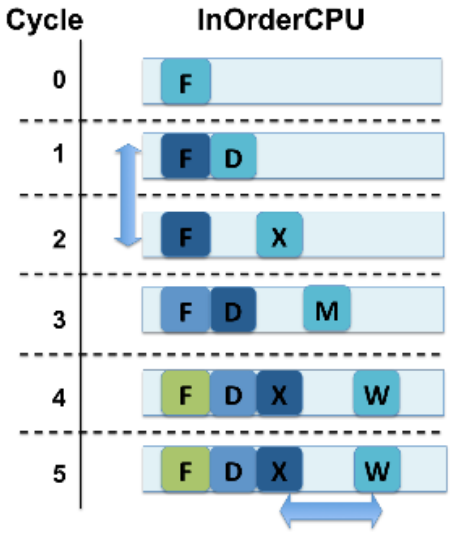
SimpleCPU model je čistě funkcionální, v programovém pořadí provádějící model vhodný pro případy, ve kterých není třeba využít detailnější model.

Tento model simuluje v každém tiku hodin instrukci. V každém cyklu procesoru je tedy zpracována celá instrukce ve všech fázích.

V simulátoru jsou implementovány dva SimpleCPU modely: AtomicSimpleCPU a TimingSimpleCPU.

AtomicSimpleCPU simuluje atomický přístup k paměti. Model využívá odhadovaný čas latence k určení odhadu času, jak dlouho bude trvat paměťová operace fáze procesoru čtení/zápis. Na obrázku 1.1 vlevo jsou zobrazeny dva cykly procesoru. Obě fáze (Fetch, Memory) mají danou časovou délku, která se nemění.

TimingSimpleCPU simuluje přístup k paměti s časem. Procesor se při paměťové operaci zastaví a čeká, než paměťový systém zpracuje požadavek a vrátí řízení procesoru. Na obrázku 1.1 vpravo je obrázek tohoto modelu. Je na něm vidět, že obě fáze (Fetch, Memory), které využívají paměť, trvají déle než ostatní operace, které paměť nevyužívají. Čas čekání na data zobrazují na obrázku čtecí zpoždění (Fetch Delay a LS/ST Delay).



Obrázek 1.2: Obrázek stavů pipeline při vykonávání instrukce modelu InOrder (zdroj: [7])

1.1.1.2 InOrder model

Model InOrder simuluje trvání všech stavů pipeline. Pipeline má 5 úrovní. V každém tiku hodin může instrukce postoupit o jednu úroveň dále. To se ovšem nestane, pokud se v tomto stavu nestihla vykonat nebo nějaká instrukce v následujícím stavu zůstává.

Na obrázku 1.2 je zobrazeno provádění instrukcí a jak prochází jednotlivými stavy. V cyklu 1 se instrukce ve stavu F (fetch) nestihla celá načíst, zůstává tedy v tomto stavu i v dalším cyklu a nemůže se načítat další instrukce. V cyklu 4 se instrukce ve stavu X (execute) nestihla provést, zůstává v tomto stavu další cyklus a instrukce ve stavech F (fetch) a D (decoce) musí v těchto stavech také setrvat.

1.1.1.3 O3CPU model

O3CPU model procesoru je nejvíce složitý model. Tento procesor již nemusí zachovávat programové pořadí instrukcí.
Vlastnosti O3CPU modelu podle [6]:

- Predikce skoku – Používají se lokální, globální a turnajová.
- Přeuspořádání bufferu – obsluhuje instrukce, které dosáhly konce, a zajišťuje jejich programové pořadí.
- Instrukční fronta – Udržuje instrukce k provedení a plánuje již připravené instrukce.

- Načítací–ukládací fronta – Udržuje paměťové operace, které dosáhly konce, pro případ špatné predikce.
- Funkční jednotka – Zajišťuje časování provádění instrukcí.

Pipeline má 7 úrovní:

- Fetch – načtení instrukce
- Decode – dekódování instrukce
- Rename – přejmenování instrukce používající fyzický registr souboru
- Issue – vystavení instrukce
- Execute – provedení instrukce
- Writeback – zápis vypočtené hodnoty
- Commit – obsluha chyby instrukce a informování při chybné predikci

1.1.2 Režimy simulace

Simulátor má dva režimy simulace: Full system (Úplný systém) a Syscall emulation (Emulace systémových volání).

Full system– tento režim simuluje celý počítač, kde je operační systém součástí simulačního prostředí. Režim zahrnuje hardware i zařízení, která jsou simulována. V tomto režimu lze využívat všech privilegovaných instrukcí a lze spouštět dynamicky slinkované programy. Nejdříve musí být spuštěn simulovaný operační systém. Potom se lze pomocí terminálu připojit k simulovanému systému a provádět akce. Výhodou režimu je jeho komplexnost. Nevýhodou jsou pak časové nároky, které náročnější simulace vyžaduje. Tento režim je zatím plně podporován pro architektury ALPHA a ARM.

Syscall emulation – pro použití tohoto režimu není třeba simulovat celý operační systém. Režim simuluje program a v případě, že se má provést systémové volání, tak je emulováno. Většinou je v těchto případech použito volání hostujícího operačního systému. Omezením tohoto režimu je nutnost zkompilovat program staticky. Výhodou je rychlost, jelikož není potřeba simulovat i operační systém.

1.1.3 Paměťové systémy

Simulátor gem5 obsahuje dva různé paměťové modely: Klasický a Ruby. Klasický paměťový systém byl převzat ze simulátor M5, zatímco paměťový systém Ruby je postaven na paměťovém systému Ruby simulátoru GEMS.

Klasický paměťový systém je rychlý a jednoduše konfigurovatelný systém. Cache koherence je udržována pomocí abstraktního MOESI snooping protokolu, kde změna stavu nastává okamžitě.

Zde jsou uvedeny výhody a nevýhody podle dokumentace [6]:
Výhody

- Fast–forwarding – Model podporuje atomický přístup, který je rychlejší než detailní přístup.
- Rychlost – V porovnání s paměťovým systémem Ruby je tento model rychlejší.
- Lehce konfigurovatelný – Konfiguruje se pomocí souboru psaného v Pythonu, kde lze definovat paměťový systém, na který je automaticky aplikován abstraktní koherenční protokol.

Nevýhody

- Flexibilita koherence cache – Model je omezen na abstraktní MOESI snooping protokol, na jehož změnu je třeba vynaložit velké úsilí.
- Přesnost koherence cache – Nejsou modelovány přechodné stavy, model tedy není tak detailní jako Ruby.

Paměťový systém Ruby nenabízí takovou simulační rychlost, ale zaměřuje se na schopnost infrastruktury přesného simulovaní velkého množství paměťových systémů. K tomuto účelu Ruby podporuje specifický doménový jazyk SLICC, kde může být definováno mnoho různých typů cache koherenčních protokolů.

Podle dokumentace [6] jsou výhody a nevýhody paměťového systému Ruby opačně než u Klasického paměťového systému. U Ruby jsou výhodami přesnost a flexibilita cache a nevýhodami nepodporování fast–forwarding, nižší rychlost a složitější konfigurace.

Více podrobně je paměťový systém Ruby rozebrán v následující sekci 1.2.

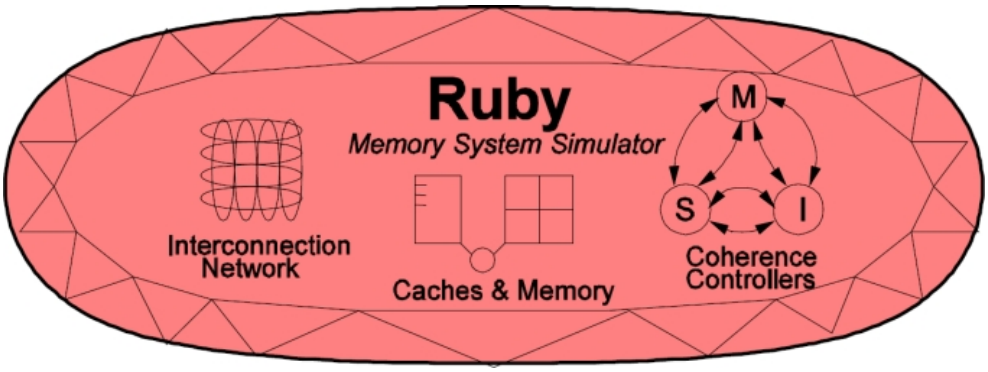
1.2 Paměťový systém Ruby

Ruby implementuje detailní simulační model paměťového systému. Model zahrnuje hierarchii cache pamětí s různými nahrazovacími strategiemi, implementaci koherenčního protokolu, propojovací sítě, DMA a paměťové řadiče, různé sequencery, které iniciují paměťové požadavky a obsluhují odpovědi. Model je modulární, flexibilní a vysoce konfigurovatelný. Tři hlavní aspekty tohoto modelu [6]:

- Oddělený návrh – Například specifikace koherenčního protokolu je oddělená od nahrazovací strategie a mapování cache indexu, topologie je samostatně specifikována od její implementace.
- Bohatá konfigurovatelnost – Většina aspektů týkajících se funkcionality a časování paměťové hierarchie může být kontrolována.

- Rychlý návrh – Vyšší specifický jazyk SLICC lze použít ke specifikaci funkcionality různých řídících prvků a není třeba zasahovat do kódu simulátoru.

Obrázek 1.3 zobrazuje hlavní komponenty Ruby podle [8]. Paměťový systém obsahuje ještě další komponenty, které v této úrovni vhledu nejsou zobrazeny. Pro účely diplomové práce jsou v následujících podsekcích popsány komponenty propojovací sítě, SLICC a sekvencer.



Obrázek 1.3: Hlavní komponenty paměťového systému Ruby (zdroj: [8])

1.2.1 Propojovací sítě

Propojovací síť spojuje dohromady různé komponenty paměťového systému. Jak je vidět na obrázku 1.4, propojovací síť spojuje L1 cache, L2 cache a paměťové ovladače.

V simulátoru je naimplementováno několik topologií propojovací sítě. Jsou to sběrnice, mřížka, toroid a klient – klient.

Pro řízení toku a implementaci mikroarchitektury směrovačů jsou implementovány dva modely:

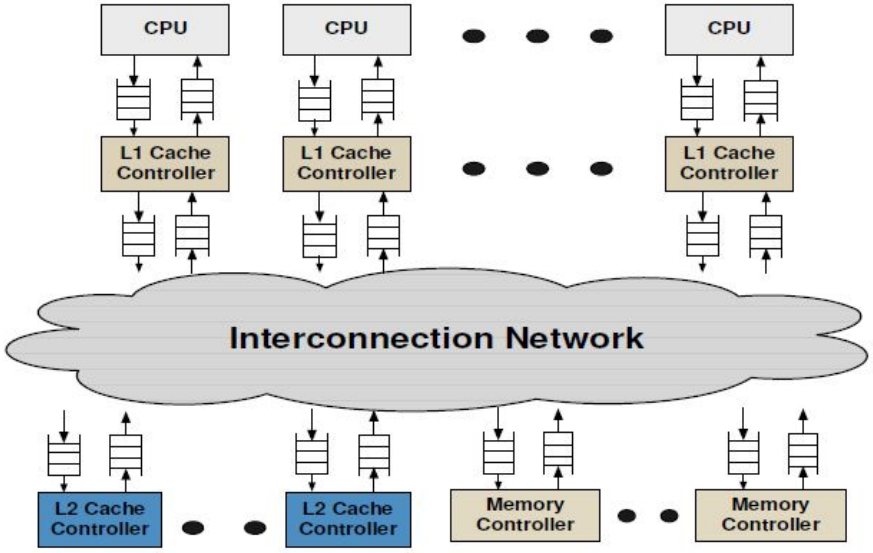
- Jednoduchý – Jednoduchý model idealizovaných směrovačů. Řízení toku je implementováno monitorováním na výstupu před odesláním.

- Garnet – Model simulující chování směrovačů v detailnější úrovni. Nevýhodou tohoto modelu je pomalejší provádění simulace.

Pro účely analyzování chování cache paměti postačuje jednoduchý model.

1.2.2 SLICC

SLICC je zkratka pro Specifikační jazyk pro implementaci koherence cache (Specification Language for Implementing Cache Coherence). Je to specifický



Obrázek 1.4: Propojovací síť paměťového systému Ruby (zdroj: [6])

doménový jazyk sloužící ke specifikaci koherence cache paměti. Kompilací tohoto jazyka vzniká kód pro různé ovladače paměťového systému. Lze také vygenerovat tabulky přechodů ve formátu HTML.

SLICC je založen na myšlence specifikování individuálních ovladačů stavových automatů, které reprezentují komponenty, jako například cache ovladače. Každý ovladač je koncepční stavový automat pro každý paměťový blok, který obsahuje:

- Stavy – množina možných stavů všech cache bloků
- Události – podmínky reagující na stavové události
- Přechody – kartézský součin stavů a přechodů
- Akce – specifické operace vykonané během události

Před začátkem transakce musí být všechny zdroje dostupné. Tento preventivní krok se snaží předcházet zablokování uprostřed transakce.

SLICC je syntakticky podobný jako C nebo C++, ale je v některých ohledech limitován. Například nelze definovat lokální proměnné nebo nelze využít cyklů. Na druhou stranu jsou přidány jazykové konstrukce pro vložení zprávy do fronty a načtení informací z další zprávy ve frontě.

1.2.3 Sekvencer

Sekvencer je komponenta paměťového systému, která je mezi jádrem procesoru a L1 instrukční a datovou cache. Každé jádro je připojeno přes vlastní sekvencer, který obsluhuje pouze toto jádro. Skrz sekvencer projde každý paměťový požadavek nejméně dvakrát: jednou předtím než požadavek obdrží L1 cache a jednou poté, co je požadavek obsloužen L1 cache.
Sekvencer je zodpovědný za:

- vkládání a vedení evidence všech paměťových požadavků do L1 cache,
- alokaci zdrojů a kontrolu dodržování maximálního počtu zatím nevyřízených paměťových požadavků,
- kontrolu a správnou obsluhu atomických operací,
- kontrolu, jestli obsluha požadavku neuvázla,
- navrácení výsledku L1 cache do správného portu jádra.

1.3 Vlákna

V režimu Syscall emulation chybí plánovač vláken. [5] Pro simulaci vláken je třeba využít implementaci speciálních vláken, která jsou navržena pro tento režim.

Knihovna pro vlákna je nahrána v mercurial repozitáři gem5 projektu. Následující informace jsou ze souboru README, který je umístěný v repozitáři m5threads projektu na adrese `http://repo.gem5.org/m5threads/`.

1.3.1 Implementace vláken

Implementace vláken je obsažena v knihovně m5threads. Knihovna byla vyvinuta jako náhrada za NPTL/LinuxThreads implementaci libpthread. Při implementaci byla snaha o co nejmenší používání systémových volání, aby bylo možné zůstat v uživatelské úrovni jak jen je to možné. Implementace se neobešla bez dvou systémových volání: *clone* pro vytvoření nového vlákna a *exit* pro ukončení vlákna.

Implementace není shodná s implementací pthreads, jelikož tato knihovna nesplňuje tak širokou funkcionalitu. Knihovna implementuje podmnožinu specifikace pthreads. Tuto část zahrnuje:

- Vytvoření a ukončení vlákna,
- čekání na ukončení vlákna pomocí join,
- regulérní mutexy (již ne rekurzivní nebo jiné ojedinělé),

- regulérní zámky pro čtení–zápis,
- bariéry,
- podmíněné proměnné,
- klíče (key_create/delete, get/setspecific).

Knihovna již neimplementuje:

- Funkci pthread_cancel a funkce příbuzné,
- funkci pthread_kill,
- nic, co je spojené se signály,
- funkce thread_cleanup_XXX, pthread_unwind.

Pokud program použije nenaimplementovanou funkci, poté zhavaruje.

1.3.2 Alokace paměti pro vlákno

Všechny paměťové oblasti, které jsou potřeba pro vytvoření nového vlákna, jsou uloženy v jednom paměťovém segmentu. Jeho struktura je na obrázku 1.5. Paměťový segment se namapuje pomocí funkce mmap jako privátní anonymní oblast (více v sekci 2.4). Poté jsou naplněny struktury v jednotlivých částech paměti. Nakonec je s tímto segmentem zavolána funkce clone(), která vytvoří nové vlákno.

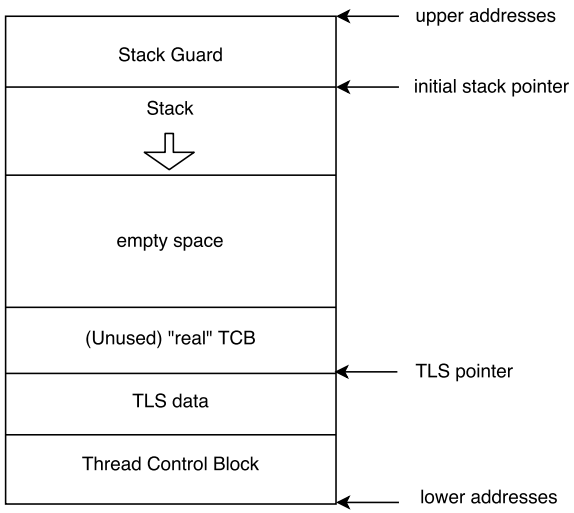
Úsek označený jako *(Unused) "real"TCB* je zachován kvůli kompatibilitě a inicializován hodnotami nula. Běžně je využíván implementacemi LinuxThreads a NPTL a mohou zde být uloženy některé proměnné jako například errno.

1.3.3 Omezení implementace

Nevýhodou použití vláken v režimu Syscall emulation je, že nelze spustit více nových vláken než kolik je jader procesoru bez jednoho. To je způsobeno tím, že v tomto režimu je třeba namapovat každé vlákno na jedno jádro. Jedno jádro je třeba pro hlavní vlákno, zbytek lze obsadit dalšími vytvořenými vlákny.

Implementace knihovny byla omezena na architektury x86 a SPARC. V roce 2012 byla přidána podpora architektury ARM. Pro každou architekturu je třeba naimplementovat thread-local storage (TLS). Implementace pro ostatní architektury nyní není.

Další omezení vyplývají z nenaimplementovaných částí uvedených v sekci 1.3.1.



Obrázek 1.5: Obrázek struktury paměťového segmentu pro nové vlákno (zdroj: [9])

1.4 Analýza trasování paměťových operací

V simulátoru gem5 jsou tři možnosti ladění:

- ladění založené na trasování
- ladění založené na ladícím nástroji gem5
- ladění simulováním kódu na vyšší úrovni gem5

Jelikož je třeba automatizovaně získat informace o chování simulovaného programu a tyto informace dále zpracovat, tak je nejvhodnější varianta ladění založené na trasování.

1.4.1 Ladění založené na trasování

Tento způsob ladění je založen na textovém výstupu činnosti simulátoru do ladícího souboru. Pokud není specifikován výstupní ladící soubor pomocí parametru *--debug-file*, tak je výstup spojen s výstupem simulovaného programu.

Ladící příznak (dále jen příznak) je řetězec, který slouží k určení informací, jež se vypíší do výstupního ladícího souboru. Jejich úplný seznam lze získat při spuštění simulátoru s parametrem *--debug-help*. K určení, které příznaky má simulátor použít, je parametr *--debug-flags*. Za tímto parametrem napíšeme rovnítko a poté výčet požadovaných příznaků oddělených čárkou. Toto vše je bez mezer.

V ladícím výpisu jsou některé informace uváděny automaticky. Příkladem je například číslo cyklu, ve kterém příkaz nastal. V některých případech tuto informaci nepotřebujeme nebo nám může být i na obtíž. Proto lze některé informace z výpisu odstranit tím, že uvedeme odpovídající příznak s mínus na začátku.

Pro vytvoření ladícího výstupu v kódu je určena funkce *DPRINTF*. Tato funkce má za parametry příznak, formátovací řetězec a proměnné určené k výstupu. Funkce je vždy spjata s uvedeným příznakem a textový výstup je do ladícího souboru zahrnut pouze v případě, že byla simulace spuštěna s daným příznakem.

Pokud je potřeba sledovat událost a žádný dosavadní příznak to neumožňuje, lze do kódu doplnit potřebné informace k sledování a jejich výpis právě pomocí *DPRINTF*. Lze přidat vlastní příznak a to pomocí funkce *DebugFlag()* s názvem příznaku do nějakého SConscript souboru (doporučuje se co nejblíže místu, kde je příznak používán). Pokud se příznak využívá v zdrojových kódech napsaných v C++, je třeba ještě v tomto souboru připsat *"#include debug/<název příznaku>.hh"*.

Simulátor je možné zkompilovat v několika verzích. Některé verze funkci *DPRINTF* nezahrnují do výsledného binárního souboru. Pro případy ladění je třeba používat verzi simulátoru *gem5.opt*.

Výstupní soubor ladění může být velmi velký, ale vzhledem k datům v něm uloženým je dobře zkomprimovatelný. Pro zapnutí komprese stačí na konec názvu souboru specifikovaném parametrem *--debug-file* připsat příponu *.gz*. Simulátor pak pozná, že výstup má být komprimován a výsledkem je zkomprimovaný soubor programem gzip.

1.4.2 Rozbor možností trasování

Při spuštění simulátoru s příznakem k vypsání podporovaných ladících příznaků je vypsáno více než 200 řádků. Příznaky jsou zobrazeny ve dvou kategoriích, kterými jsou základní a složené. Příznaků je celkem 166.

První základní kategorie zobrazuje příznaky jako seznam s jejich popisem. Jen málo z nich je popsáno, k čemu přesně jsou určeny. Když už se u příznaku nachází popis, potom je jen velmi stručný, shrnutý do několika slov.

Složené příznaky jsou strukturovány do 14–ti kategorií, podle jejich určení. Komentáře k těmto příznakům nejsou žádné, ale lze přibližně odhadnout jejich účel podle kategorie a jména. Mezi kategoriemi je i kategorie nazvaná Ruby.

Jelikož v popisu příznaků není většinou uvedeno, k čemu přesně příznak slouží, a nenašel jsem tyto údaje ani v dokumentaci simulátoru, rozhodl jsem se prozkoumat kategorii příznaků Ruby. Zde je, podle mne, největší šance najít vhodný příznak, který by zajistil potřebná data pro realizaci programu analyzující činnost chování cache pamětí. V této kategorii je zařazeno celkem 13 příznaků.

Na běhu testovacího programu jsem postupně zapínal příznaky z kategorie Ruby. Dále jsou uvedeny zjištěné informace o jednotlivých příznacích s ukázkami z výpisu.

RubyCache je příznak pro sledování ovladačů jednotlivých cache pamětí. Výpis obsahuje vyhledávání fyzické adresy v komponentách paměti. Na níže přiloženém bloku výpisu je vidět, jak se postupně vyhledává adresa v L1 cache, poté L2 cache a nakonec v hlavní paměti.

```
system.ruby.l1_cntrl0.L1Dcache: No tag match for address
    : [0x1040, line 0x1040]
system.ruby.l1_cntrl0.L1Icache: No tag match for address
    : [0x1040, line 0x1040]
system.ruby.l1_cntrl0.L1Icache: Allocate clearing lock
    for addr: [0x1040, line 0x1040]
system.ruby.l2_cntrl0.L2cache: No tag match for address:
     [0x1040, line 0x1040]
system.ruby.l2_cntrl0.L2cache: Allocate clearing lock
    for addr: [0x1040, line 0x1040]
system.ruby.dir_cntrl0.directory: Looking up address: [0
    x1040, line 0x1040]
```

Tento příznak se hodí k určení, která část paměti data obsahuje.

RubyGenerated sleduje vývoj stavu paměti pro hledanou adresu v jednotlivých částech paměti. V následují ukázce jsem vzal stejný požadavek jako v předchozí. Je zde vidět, jak požadavek prochází jednotlivými částmi paměti a postupně se mění koherenční stav paměti v komponentách.

```
system.ruby.l1_cntrl0: [L1Cache_Controller 0], Time: 3,
    state: NP, event: Ifetch, addr: [0x1040, line 0x1040]
system.ruby.l2_cntrl0: [L2Cache_Controller 0], Time: 11,
     state: NP, event: L1_GET_INSTR, addr: [0x1040, line
    0x1040]
system.ruby.dir_cntrl0: [Directory_Controller 0], Time:
    19, state: I, event: Fetch, addr: [0x1040, line 0
    x1040]
system.ruby.dir_cntrl0: [Directory_Controller 0], Time:
    63, state: IM, event: Memory_Data, addr: [0x1040,
    line 0x1040]
system.ruby.l2_cntrl0: [L2Cache_Controller 0], Time: 70,
     state: IS, event: Mem_Data, addr: [0x1040, line 0
    x1040]
system.ruby.l1_cntrl0: [L1Cache_Controller 0], Time: 77,
     state: IS, event: Data_all_Acks, addr: [0x1040, line
     0x1040]
```

```
system.ruby.l1_cntrl0: [L1Cache_Controller 0], Time: 81,
    state: S, event: Ifetch, addr: [0x1040, line 0x1040]
```

Z výpisu lze určit, ve které komponentě se paměťový blok nachází a v jaké stavu se nachází.

TraceProtocol příznak obsahuje podobné informace jako příznak *RubyGenerated*. Tento výpis je jinak formátovaný a stručnější.

```
0        Seq                Begin        >        [0x1058,
    line 0x1040] IFETCH
0     L1Cache               Ifetch      NP>IS      [0x1040,
    line 0x1040]
0     L2Cache         L1_GET_INSTR      NP>IS      [0x1040,
    line 0x1040] [NetDest (4) 1 - 0 - 0 - - ]
0   Directory               Fetch       I>IM      [0x1040,
    line 0x1040]
0   Directory         Memory_Data       IM>M      [0x1040,
    line 0x1040]
0     L2Cache            Mem_Data      IS>SS      [0x1040,
    line 0x1040]
0        Seq                Done        >        [0x1058,
    line 0x1040] 77 cycles
0     L1Cache        Data_all_Acks       IS>S      [0x1040,
    line 0x1040]
0        Seq                Begin        >        [0x1058,
    line 0x1040] IFETCH
0        Seq                Done        >        [0x1058,
    line 0x1040] 3 cycles
0     L1Cache               Ifetch       S>S      [0x1040,
    line 0x1040]
```

RubyMemory zajišťuje výpis informací o činnosti hlavní paměti. Ve výpisu je vidět přijetí požadavku, jeho zpracování a odeslání odpovědi v daném čase.

```
system.ruby.dir_cntrl0.memBuffer: New memory request
        1: 0x001040 R arrived at        8000 bank =  10
    sched N
system.ruby.dir_cntrl0.memBuffer: Refresh bank   0
system.ruby.dir_cntrl0.memBuffer: Mem issue request
        1: 0x001040 R bank= 10 sched N
system.ruby.dir_cntrl0.memBuffer: Enqueueing msg 0
    x001040 R back to directory at           63000
system.ruby.dir_cntrl0.memBuffer: Peek: memory request
        1: 0x001040 R sched Y
```

Lze tento příznak využít ke zjištění, zda-li byla data načtena z hlavní paměti, ale k tomu by se daly využít jiné příznaky, které poskytují více informací o tom, odkud data pocházejí.

RubyNetwork zaznamenává činnost propojovací sítě. U každého přepínače je uvedeno, že obdržel zprávu. Poté je uvedena zpráva. Dále je vypsáno směrování zprávy a její odeslání na výstupní port. Pomocí tohoto příznaku lze sledovat obsah posílaných dat mezi jednotlivými částmi paměti, ale to není cílem této práce. Zde je uveden přepínač u hlavní paměti, který právě odesílá data zpátky směrem k L1 cache. Podíváme li se na poslední položku *Time*, poté ji můžeme srovnat s časem uvedeným u zpracování požadavku hlavní pamětí příznaku *RubyGenerated*. Časy jsou shodné, díky ideálnímu modelu propojovací sítě.

```
PerfectSwitch-2: incoming: 1
PerfectSwitch-2: Message: [ResponseMsg: Addr = [0x1040,
    line 0x1040] Type = MEMORY_DATA Sender = Directory-0
    Destination = [NetDest (4) 0 - 1 - 0 - - ]
    DataBlk = [ 0xbf ... 0x0 ] Dirty = 0 AckCount = 0
    MessageSize = Response_Data Time = 63000000 ]
PerfectSwitch-2: dst: [NetDest (4) 0 - 0 - 1 - - ]
PerfectSwitch-2: dst: [NetDest (4) 1 - 1 - 0 - - ]
PerfectSwitch-2: Checking if node is blocked ...outgoing
    : 1, vnet: 1, enough: 1
PerfectSwitch-2: Enqueuing net msg from inport[0][1] to
    outport [1][1].
```

Tento příznak nenabízí užitečné informace, které by se nedaly zjistit z předchozích příznaků.

RubySlicc slouží k sledování činnosti koherenčního protokolu implementovaného pomocí doménového jazyka SLICC. První tři řádky poukazují na hledání, kde požadovaná data jsou. Další tři řádky reprezentují žádost o data. Následuje řádek odeslání dat z hlavní paměti. Poté už je přijetí dat L1 ovladačem. Data jsem zkrátil na první a poslední byte.

```
system.ruby.l1_cntrl0: MESI_CMP_directory-L1cache.sm
    :214: NotPresent
system.ruby.l2_cntrl0: MESI_CMP_directory-L2cache.sm
    :212: NotPresent
system.ruby.dir_cntrl0: MESI_CMP_directory-dir.sm:153:
    Read_Write
system.ruby.l1_cntrl0: MESI_CMP_directory-L1cache.sm
    :544: address: [0x1040, line 0x1040], destination: [
    NetDest (4) 0 - 1 - 0 - - ]
system.ruby.l2_cntrl0: MESI_CMP_directory-L2cache.sm
    :342: Addr: [0x1040, line 0x1040] State: NP Req:
```

```
    L1Cache-0 Type: GET_INSTR Dest: [NetDest (4) 0  - 1
    - 0  -  - ]
system.ruby.l2_cntrl0: MESI_CMP_directory-L2cache.sm
    :174: machineID: L2Cache-0, requestor: L1Cache-0,
    address: [0x1040, line 0x1040]
system.ruby.dir_cntrl0: MESI_CMP_directory-dir.sm:309: [
    MemoryMsg: Addr = [0x1040, line 0x1040] Type =
    MEMORY_READ Sender = Directory-0
    OriginalRequestorMachId = L2Cache-0 DataBlk = [ 0xbf
    ... 0x0 ] MessageSize = Control Prefetch = No ReadX =
     0 Acks = 0 Time = 19000000 ]
system.ruby.l1_cntrl0: MESI_CMP_directory-L1cache.sm
    :814: [ 0xbf ... 0x0 ]
system.ruby.l1_cntrl0: MESI_CMP_directory-L1cache.sm
    :806: [ 0xbf ... 0x0 ]
system.ruby.l1_cntrl0: MESI_CMP_directory-L1cache.sm
    :806: [ 0xbf ... 0x0 ]
```

Výpis obsahuje informace o činnosti koherenčního protokolu, ale to není cílem této práce.

RubySystem příznak slouží k sledování zpráv systému. Vzhledem k dosavadnímu cíli práce není potřeba tyto zprávy sledovat.

RubyPort příznak slouží k sledování, co se děje v komponentě sekvencer. Je zde vidět, že požadavek prošel přes sekvencer dvakrát, jak je popsáno k kapitole o sekvenceru 1.2.3.

```
system.ruby.l1_cntrl0.sequencer-slave0: Timing access
    caught for address 0x1058
system.ruby.l1_cntrl0.sequencer-slave0: Request 0x1058
    issued
system.ruby.l1_cntrl0.sequencer-slave0: Hit callback
    needs response 1
system.ruby.l1_cntrl0.sequencer-slave0: Sending packet
    back over port
system.ruby.l1_cntrl0.sequencer-slave0: Hit callback
    done!
system.ruby.l1_cntrl0.sequencer-slave0: Timing access
    caught for address 0x1058
system.ruby.l1_cntrl0.sequencer-slave0: Request 0x1058
    issued
system.ruby.l1_cntrl0.sequencer-slave0: Hit callback
    needs response 1
system.ruby.l1_cntrl0.sequencer-slave0: Sending packet
    back over port
```

```
system.ruby.l1_cntrl0.sequencer-slave0: Hit callback
    done!
```

Výpis zachycuje pouze komunikaci mezi procesorem a L1 cache pamětí, a to jen v omezeném rozsahu.

RubyQueue příznak zaznamenává přenos zpráv mezi jednotlivými prvky ve frontách. Mezi každými dvěma spojenými prvky je fronta, kterou musí zpráva projít. Výpis je zde obsáhlejší, takže v porovnání s příkladem u příznaku *RubyGenerated* je zde výpis na 98 řádků. Zde lze například sledovat z jaké části paměti požadavek pochází nebo i jaká instrukce požadavek vyvolala. Tento příznak se zabývá zprávami na až moc detailní úrovni, která je pro práci zbytečná.

Příznaky **RubyCacheTrace**, **RubyDma**, **RubyPrefetcher**, **RubySequencer** a **RubyTester** nevypsaly žádný výstup. Nejspíše je to způsobeno tím, že testovaný program nevykonával činnost, kterou tyto příznaky monitorují.

Žádný z příznaků nepokrývá potřebné informace pro účel práce. Nenašel jsem ani vhodnou kombinaci více příznaků. Proto je pro účely práce trasování rozšířeno. Rozšíření je popsáno v další sekci.

1.5 Rozšíření trasování paměťových operací

Jak je uvedeno na konci předchozí části textu, v simulátoru gem5 nyní není vhodné trasování paměťových operací k splnění požadavků vyvíjeného programu.

Jelikož typ sdílení proměnných závisí jak na virtuální adrese, tak i na fyzické adrese, je třeba mít tyto informace propojené. Právě fyzická adresa spojuje proměnné do cache bloků. Proto je třeba vytvořit nový trasovací příznak, který tyto informace propojí.

V již implementovaných příznacích trasování chybí párování paměťových operací s instrukcemi. Tyto informace jsou potřebné pro určení, v jaké části programu se provedla paměťová operace.

U výpisů jsou také hodnoty důležitých registrů. Jsou to program counter a stack pointer.

Změny v kódu simulátoru gem5 jsou zaznamenány pomocí verzovacího nástroje mercurial. Výstup nástroje mercurial je uložen na přiloženém CD.

Pro účely trasování jsou přidány příznaky **ExecMemory** a **ExecMemory2**. Příznaky byly implementovány pro TimingSimple model CPU.

Zapnutím přidaných trasovacích příznaků přibudou další trasovací informace v trasovacím souboru, které lze rozdělit na dvě skupiny:

- operace jader
- činnost cache

Výpis informací je po řádcích.

Každý řádek výpisu trasování lze rozdělit na části, které obsahuje. Na dalším řádku jsou shrnuté části, jak jdou za sebou:

```
time: system: instruction with data
```

První položka čas(time) je přidána trasovacím nástrojem a lze ji pomocí trasovacích příznaků vypnout. Následuje položka systém(system). Tato položka označuje, jestli se jedná o operaci jádra nebo činnost cache. Operace a dodatečná data (instruction with data) určují, o jakou variantu se jedná, a obsahuje informace o operaci.

V následujících podkapitolách jsou ukázky bez hodnot času(*time*) na začátku výpisu, aby byly ukázky kratší.

1.5.1 Operace jader

Ve výpisu označují tyto operace v druhé části text *system.cpuX*, kde X označuje číslo jádra, kterého se instrukce týká.

```
time: system.cpuX: instruction, data
```

Instrukce(instruction) je jedna z následujících:

- StartInts nebo EndInst – nepaměťové operace
- StartMemInst nebo EndMemInst – paměťová operace
- ReadMem – operace čtení
- WriteMem – operace zápisu
- StartCall – volání funkce
- StartRet – návrat z funkce

Dále jsou rozebrány jednotlivé instrukce i s ukázkou jejich výpisu.

1.5.1.1 Nepaměťové operace

Slouží k trasování všech činností procesoru, které nepracují s pamětí nebo neslouží k volání funkcí. Výpis těchto operací je párový.

```
system.cpu0: StartInst(1,1), pc=0x401058, sp=0
    x7fffffffee80,    XOR_R_R : xor    ebp, ebp, ebp
system.cpu0: EndInst(1,1), pc=0x401058, sp=0
    x7fffffffee80
```

StartInst uvozuje zahájení zpracování instrukce. *EndInst* ukončuje zpracování instrukce v jádře. Na konci instrukce jsou závorky se dvěma čísly oddělenými čárkou. Jelikož může být jedna instrukce jádra rozdělena na několik mikroinstrukcí jdoucích za sebou, slouží toto značení k určení, které instrukce spolu souvisí. Na pozici čísel je buď číslo nula nebo jedna. První pozice znamená, že je instrukce první. Druhé číslo znamená, že instrukce je poslední. Jednička označuje pravdu, nula nepravdu. Následuje ukázka posloupnosti instrukcí spolu souvisejících:

```
system.cpu0: StartInst(1,0), pc=0x402737, sp=0
    x7fffffffeda0,    JMP_I : rdip    t1, %ctrl153,
system.cpu0: StartInst(0,0), pc=0x402737, sp=0
    x7fffffffeda0,    JMP_I : limm    t2, 0
    xfffffffffffffe93
system.cpu0: StartInst(0,1), pc=0x402737, sp=0
    x7fffffffeda0,    JMP_I : wrip    , t1, t2
```

Toto určení větších instrukcí se využívá i u dalších instrukcí.

Dále jsou ve výpisu vypsány ukazatele jádra. Ukazatel *program counter*, který označuje právě vykonávanou instrukci, má označení *pc*. Ukazatel *stack pointer*, který označuje vrchol zásobníku jádra, má označení *sp*. Dále je na řádku *StartInst* vypsána instrukce jádra, které se výpis týká.

Po analýze nepaměťových operací jsem nalezl, že u některých nepaměťových operacích se v rámci ní změní hodnota ukazatele stack poiner. Je to způsobeno tím, že je změněna hodnota stack pointeru odečtením nebo přičtením hodnoty. Takto se buď zvětší nebo smrští zásobník, aniž by se zapsala nebo přečetla data. Při návrhu struktur reprezentujících zásobník bude třeba tuto skutečnost zohlednit.

1.5.1.2 Paměťové operace

Paměťové operace jsou svázány s operacemi čtení (ReadMem) a zápisu (WriteMem). Také se v bloku paměťové operace mohou objevit bloky nepaměťových operací, které jsou v rámci složitějších paměťových operací potřeba k výpočtu nebo řízení.

V ukázce je příklad paměťové operace čtení. V řádku s *ReadMem* je zde přidána cílová virtuální adresa čtení označená *va*. Také je zde velikost čtení *size*. Posledním údajem na řádce je příznak čtení *flag*. Na dalším řádku s *sendData* je položka *read*, která označuje načtená data. Poté následují informace o fyzické adrese, kterou jsme četli. Dále už jsou informace, které se opakují.

```
system.cpu0: StartMemInst(1,0), pc=0x40105d, sp=0
    x7fffffffee80,    POP_R : ld    t1, SS:[rsp]
system.cpu0: ReadMem, pc=0x40105d, va=0x7fffffffee80,
    size=0x8, flags=0x2
```

```
system.cpu0: sendData, read=1, pa=[0xcee80, line 0xcee80
    ]
system.cpu0: EndReadMem, pc=0x40105d, pa=[0xcee80], data
    =0x1
system.cpu0: EndMemInst(0,1), pc=0x40105d, sp=0
    x7fffffffee88
```

V případě zápisu je jen drobná změna. V následující ukázce již není uveden první a poslední řádek.

```
system.cpu0: WriteMem, pc=0x401065, va=0x7fffffffee78,
    size=0x8, data=0x0, flags=0x2
system.cpu0: sendData, read=0, pa=[0xcee78, line 0xcee40
    ]
system.cpu0: EndWriteMem, pc=0x401065, pa=[0xcee78]
```

Oproti operaci čtení je zde změna v umístění položky *data*. Ta je již na řádku *WriteMem* a udává zapisovanou hodnotu. Položka *read* je v následujícím řádku rovna nule, jelikož žádná data nejsou načítána.

1.5.1.3 Volání funkce

Jelikož volání funkce je složitější operace, skládá se z více instrukcí. Volání funkce uvozuje řádek s *StartCall* a ukončuje řádek s instrukcí *EndMemInst*. K tomu slouží i nepaměťové operace, které jsem v ukázce vynechal. V tomto bloku je vždy i paměťový zápis, kdy se ukládá ukazatel k návratu po ukončení funkce (*return address*). Hodnoty *si* a *di* jsou vstupní parametry funkce. Jaká funkce je volána v tomto bloku nelze poznat. Zjistí se to až při následující instrukci podle *program counter* ukazatele.

```
system.cpu0: StartCall(1,0), pc=0x40260c, sp=0
    x7fffffffeda0, di=0x7fffffffeea0, si=0x1,
    CALL_NEAR_I : st    t7, SS:[rsp + 0xffffffffffffff8]
system.cpu0: WriteMem, pc=0x40260c, va=0x7fffffffed98,
    size=0x8, data=0x402611, flags=0x2
system.cpu0: sendData, read=0, pa=[0xced98, line 0xced80
    ]
system.cpu0: EndWriteMem, pc=0x40260c, pa=[0xced98]
system.cpu0: EndMemInst(0,1), pc=0x40260c, sp=0
    x7fffffffed98
```

1.5.1.4 Návrat z funkce

Instrukce pro návrat *StartRet* je uvozující a blok ukončuje *EndMemInst*. Jako v ukázce při volání funkce jsem i zde odstranil řádky nepaměťových operací.

Uvnitř tohoto bloku je čtení. Jedná se o načtení návratové adresy *return address*, kde se má pokračovat ve vykonávání programu po návratu z funkce.

```
system.cpu0: StartRet(1,0), pc=0x4393d9, sp=0
    x7fffffffed98, ax=0x7fffffffefd9,    RET_NEAR : ld
    t1, SS:[rsp]
system.cpu0: ReadMem, pc=0x4393d9, va=0x7fffffffed98,
    size=0x8, flags=0x2
system.cpu0: sendData, read=1, pa=[0xced98, line 0xced80
    ]
system.cpu0: EndReadMem, pc=0x4393d9, pa=[0xced98], data
    =0x402611
system.cpu0: EndMemInst(0,1), pc=0x4393d9, sp=0
    x7fffffffeda0
```

1.5.2 Činnost cache

Ve výpisu označují tyto operace v druhé části text *system.ruby*.

```
time: system.ruby: data
```

Pokud je pouze uvedeno *system.ruby*, jde o obecné hlášení cache. Infomace mohou pocházet i od jednotlivých částí cache a jejich název je potom specifikován za tečkou, například *system.ruby.l1_cntrl0*.

Obecný výpis *system.ruby* se vypisuje po paměťové operaci. Hlásí stav cache bloku, se kterým právě paměť pracovala. Jsou zde informace, o který blok se jedná, číslo 4 označuje stav cache bloku a posledním údajem na řádce je seznam cache úrovní, kde je právě blok uložen.

```
system.ruby: cachelineState, [0xced80, line 0xced80], 4,
    L1:0
```

Stav cache bloku může nabývat hodnot od 1 do 5 a mají tyto významy:

- 1 – Cache blok je uložen v paměti.
- 2 – Cache blok je v právě v jedné paměti cache v režimu čtení.
- 3 – Cache blok je sdílen ve více pamětech cache v režimu čtení.
- 4 – Cache blok je právě v jedné paměti cache v režimu zápisu.
- 5 – Cache blok může být přítomen ve více pamětech cache pro čtení i zápis.

Dále následují výpisy jednotlivých pamětí cache o zpracování paměťového požadavku. V následující ukázce je vidět, že adresa k paměťové opraci nebyla načtena v žádné úrovni cache. Ovladače L1 i L2 cache nenašly požadovaná data. Poslední řádek udává, že jsou data načtena do L1 cache.

```
system.ruby.l1_cntrl0: L1miss [0xcec00, line 0xcec00]
system.ruby.l2_cntrl0: L2miss [0xcec00, line 0xcec00]
system.ruby.l1_cntrl0: L1unblock [0xcec00, line 0xcec00]
```

Kapitola 2

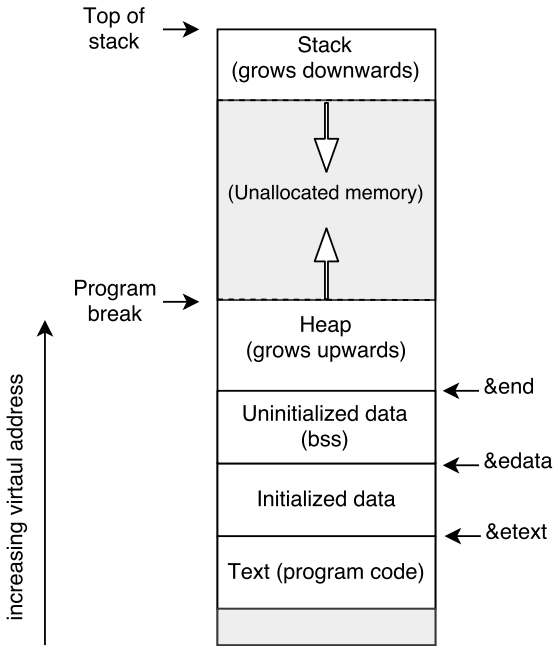
Rozbor použití adresního prostoru procesu

Cílem vyvíjeného analyzátoru je určit typy sdílení proměnných. K tomu je třeba nejdříve prozkoumat strukturu uložení proměnných v paměti. S tím souvisí i analýza, jak se paměť může během vykonávání programu měnit. Dále je třeba analyzovat životní cyklus proměnné.

Virtuální paměť procesu se skládá z několika částí, které se nazývají segmenty. Jsou to:

- *Textový* segment, který obsahuje instrukce zdrojového kódu programu. Tento segment je určen pouze pro čtení.
- *Inicializovaný datový* segment obsahuje globální a statické proměnné, které jsou přímo inicializované počátečními hodnotami. Obsah těchto proměnných je načten při zavádění programu do paměti.
- *Neinicializovaný datový* segment obsahuje globální a statické proměnné, které nejsou přímo inicializovány. Z historických důvodů je tento segment často nazýván *bss*, což je odvozeno z „block started by symbol“.
- *Zásobník* je dynamicky zvětšující se i zmenšující se segment obsahující zásobníkové rámce. Pro každé zavolání funkce je vytvořen jeden rámec, který obsahuje parametry, lokální proměnné a návratovou hodnotu.
- *Halda* je oblast paměti, která může být dynamicky alokována za běhu programu. Konec haldy je nazýván *program break*.

Tyto segmenty jsou umístěny ve virtuální paměti. Jak jsou segmenty rozmístěny je zobrazeno na obrázku 2.1 (adresy na obrázku jsou uvedeny pro lepší orientaci). Mezi dynamicky se měnícími segmenty haldy a zásobníku je nepřiřazený prostor, který lze dynamicky alokovat i uvolňovat.



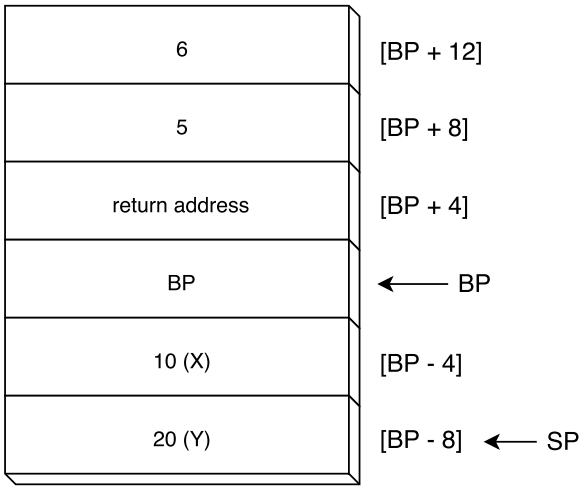
Obrázek 2.1: Členění virtuální paměti procesu (zdroj: [10])

Segmenty se dělí na menší struktury nazývající se stránky. Stránka je nejmenší část paměti, o kterou lze zmenšit nebo zvětšit segment.

Možnosti, jak zvětšit nebo zmenšit velikost segmentu a jak alokovat nebo dealokovat paměť podle [10] jsou:

- když zásobník dosáhne spodní meze dříve alokovaného prostoru,
- když je paměť alokována nebo dealokována v prostoru haldy,
- když je použita funkce brk(), sbrk() nebo funkce z rodiny malloc,
- když je použita funkce k připojení (shmat()) nebo odpojení (shmdt()) sdílené paměti v System V,
- když je použita paměťová mapovací funkce mmap() nebo munmap().

Dále jsou popsány tyto možnosti práce s pamětí, kromě funkcí pro manipulaci se sdílenou pamětí System V. Diplomová práce je zaměřena na chování paměti jednoho procesu, proto není třeba se zabývat sdílenou pamětí.



Obrázek 2.2: Rámec zásobníku funkce se dvěma parametry a dvěma lokálními proměnnými (zdroj: [11])

2.1 Paměť zásobníku

Zásobník se nachází v horním konci paměťového prostoru. Podle obrázku 2.1 hned pod vstupními parametry a proměnnými prostředí. Paměť zásobníku roste směrem dolů proti paměti haldy.

Zásobník se zvětšuje a zmenšuje podle toho, jak jsou volány funkce. Při zavolání funkce se zásobník zvětší, při návratu z funkce se smršťí. Aktuální pozici, kde se nachází vrchol zásobníku, určuje speciální registr *stack pointer*.

2.1.1 Zásobníkový rámec

Paměť se alokuje a dealokuje po zásobníkových rámcích. Zásobníkový rámec obsahuje:

- parametry funkce
- lokální proměnné
- návratovou adresa programu (return address)
- base pointer
- zálohu registrů

Návratová adresa programu slouží při návratu z funkce. Na této adrese se po návratu z funkce začne vykonávat kód. Base pointer odkazuje do paměti zásobníku na předchozí base pointer. Podle toho lze určit, kde je base pointer předchozího rámce a tím určit předchozí rámec.

2.1.2 Funkce alloca()

Funkce alloca() slouží k alokaci dynamických proměnných na zásobníku. Volání funkce v jazyce C z [10]:

```
#include <alloca.h>
void *alloca(size_t size);
```

Jediným parametrem je požadovaná velikost paměti k alokaci. V případě úspěchu je návratovou hodnotou ukazatel na alokovanou paměť.

Výhodou alokace v segmentu zásobníku je, že alokovaná paměť je součástí kontextu rámce zásobníku funkce, ve kterém byla paměť alokována. Při návratu funkce se takto alokovaná paměť automaticky uvolní, při smrštění zásobníku. Výhodou oproti funkcím rodiny malloc 2.3 je menší režie, díky jednoduššímu uspořádání paměti zásobníku [10].

2.2 Funkce brk() a sbrk()

Velikost haldy se dynamicky mění. Začíná za segmentem neinicializovaných dat a končí takzvaným *program break*. Tento program break je adresa, kam až sahá halda. Hodnotu program break lze měnit pomocí volání funkcí brk() a sbrk(). Funkce změní adresu program break a následně lze přistupovat k paměti. Paměťový prostor lze pomocí těchto funkcí jak rozšířit, tak zmenšit.

Použití těchto funkcí v jazyku C z [10]:

```
#include <unistd.h>
int brk(void *end_data_segment);
void *sbrk(intptr_t increment);
```

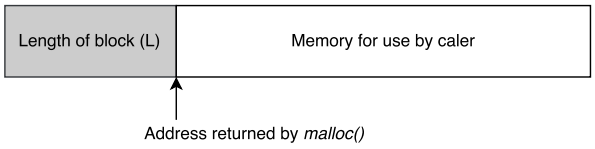
Funkce **brk()** se pokusí nastavit program break na zadanou adresu v parametru *end_data_segment*. Adresa je zaokrouhlena nahoru k nejbližšímu násobku velikosti stránky. Při úspěšné změně program break funkce vrátí hodnotu 0. Pokud nastane chyba, poté vrátí hodnotu -1.

Funkce **sbrk()** se pokusí posunout program break o danou velikost specifikovanou ve struktuře parametru *increment*. Pokud se povede rozšířit haldu, poté funkce vrátí předchozí adresu program break. Pokud funkce neuspěje, vrací hodnotu -1.

Pokud zavoláme funkci takto *sbrk(0)*, poté se nemění program break, ale funkce pouze vrátí adresu, kde se právě program break nachází.

2.3 Funkce rodiny malloc

Podle manuálových stránek [12] jsou v této rodině funkce malloc(), calloc(), realloc(), free(). Tyto funkce slouží k alokaci a uvolnění paměti v oblasti haldy.



Obrázek 2.3: Struktura alokované paměti funkcí rodiny malloc (zdroj: [10])

Funkce přímo nemění velikost haldy změnou program break. Pokud není v segmentu haldy dostatek místa, pak volají funkce uvedené v kapitole 2.2, které rozšíří haldu. Poté je paměť využita těmito funkcemi.

Tyto funkce mají vlastní systém přidělování paměti. Udržují si seznam paměťových rozsahů, které jsou volné. Při požadavku o přidělení paměti vyhledají dostatečně velký souvislý úsek paměti(záleží dle strategie, jak se paměť vybírá), jehož počáteční adresu vrátí v návratové hodnotě. Tuto přidělenou paměť si poté přiřadí k již alokované paměti. Při uvolnění paměti je tento úsek paměti zařazen mezi volnou část paměti.

Funkce **malloc()** slouží k alokování paměti o velikosti počtu bajtů specifikovaných parametrem *size*. Paměť přidělená touto funkcí není inicializována.

Funkce **calloc()** se využívá k alokaci paměti pole stejně velkých prvků. Velikost přidělené paměti je tedy součin počtu prvků (parametr *numitems*) a velikosti prvku (parametr *size*). Paměť je inicializována na hodnoty 0.

Funkce **realloc()** slouží k rozšíření paměti, která byla dříve přidělena jakoukoli funkcí z rodiny funkcí malloc. Parametr *ptr* určuje paměťový úsek a požadovanou velikost určuje parametr *size*. Pokud se za úsekem paměti, který se má rozšířit, nachází dostatečně velký úsek nealokovaná paměti, potom se úsek rozšíří. Pokud nelze jen rozšířit zvětšovaný úsek, poté se najde úsek paměti dostatečné délky a sem se zkopírují původní data. Úsek paměti, o který byl původní rozšířen, není inicializovaný.

Funkce **free()** slouží k uvolnění paměti, která byla přidělena nějakou jinou funkcí z této rodiny funkcí. Uvolněná paměť je zařazena na seznam volné paměti. Aby funkce poznala, jak dlouhý úsek paměti je uvolněn, jsou k tomu při každé alokaci uložena dodatečná data. Před alokovanou paměť je uložena velikost alokovaného úseku, jak je zobrazeno na obrázku reffig:FreeStruct. Odsud funkce zjistí, jak velký úsek paměti je uvolněn.

Použití těchto funkcí v jazyku C z [10]:

```
#include <stdlib.h>
void *malloc(size_t size);
void *calloc(size_t numitems , size_t size);
void *realloc(void *ptr , size_t size);
void free(void *ptr);
```

2.4 Funkce mapující paměť mmap()

Funkce mmap() slouží k mapování do virtuálního paměťového prostoru procesu. Pracuje s virtuálními stránkami.
Jsou dva typy mapování dle zdroje obsahu [10]:

- **Souborové** mapování mapuje specifikovanou část souboru přímo do virtuální paměti procesu.
- **Anonymní** mapování neodpovídá žádnému souboru. Takto namapovaná paměť je inicializována na hodnoty 0.

Dalším dělením je mapování dle rozsahu sdílení namapovaného prostoru [10]:

- **Privátní** oblast paměti – Paměť je pouze ve virtuálním adresním prostoru procesu, který si ji namapoval.
- **Sdílená** oblast paměti – Paměť může být sdílena mezi více procesy. Toho je docíleno tak, že virtuální stránky paměti různých procesů odpovídají jednomu rámci fyzické paměti.

Diplomová práce je zacílena na činnost jednoho procesu, proto se variantou sdílené paměti nezabývá.

2.4.1 Použití mmap()

Použití této funkce v jazyku C z [10]:

```
#include <sys/mman.h>
void *mmap(void *addr, size_t length, int prot, int
    flags, int fd, off_t offset);
```

Prvním parametrem *addr* je adresa virtuální paměti, kde by měla být paměť namapována. Buď lze uvést hodnotu 0 a lokalitu vybere systém. Nebo lze uvést adresu a systém se snaží hledat prostor v blízkosti této adresy. Poslední možností je zadat adresu a v parametru *flags* specifikovat, že paměť má být namapována právě na tuto adresu. Parametr *length* určuje požadovanou velikost paměti, kterou chceme vyčlenit pro mapování.

Parametr *prot* určuje režim ochrany paměti. Možnosti jsou: žádná, čtení, zápis, vykonávání. Možnosti lze kombinovat. Pomocí parametru *flags* lze specifikovat chování funkce. Možnosti, které se týkají diplomové práce, jsou popsány dále.

Parametr *fd* slouží k předání otevřeného popisovače souboru, který má být namapován. Soubor musí být otevřen ve stejném režimu, ve kterém je nastaven režim ochrany podle *prot*. Například pokud lze do mapované paměti lze zapisovat, poté musí být otevřen soubor pro zápis. Parametr *offset* udává byte, od kterého se má začít mapovat soubor. Nahraje se *length* bytů, pokud

tolik bytů ještě do konce souboru existuje. Pokud soubor od *offset* obsahuje méně bytů, poté se nahraje až do konce a zbytek paměti je inicializován na hodnotu 0.

2.4.2 Mapování na konkrétní adresu paměti

Zadaná adresa při volání funkce mmap() je brána jako rada, kde se má mapovat. Pro namapování na konkrétní adresu paměti (zarovnané na začátek stránky) je určen příznak *MAP_FIXED*, který je třeba přidat mezi příznaky parametru *flags* při volání funkce.

Při volání funkce mmap() s tímto příznakem a adresou paměti, kam se dříve něco namapovalo, se přepíše původní mapování. Příkladem využití je namapovat úsek paměti dle volného místa, které najde funkce. Poté pomocí mapování na konkrétní místo namapovat různé úseky. Tímto bude zajištěno, že úseky nic nepřemapují a budou namapovány dle požadavků.

2.4.3 Vrácení paměti funkcí munmap()

Funkce k navrácení namapované paměti ve virtuální prostoru procesu je munmap(). Tato funkce zruší mapování dané paměťové oblasti.
Použití této funkce v jazyku C z [10]:

```
#include <sys/mman.h>
int munmap(void *addr, size_t length);
```

Funkce navrátí paměť od adressy *addr* až do adresy *addr* + *length*. PNěkteré specifikace vyžadují, aby začátek a konec rozsahu paměti k uvolnění byl zarovnán na začátek a konec stránky. Pokud to specifikace pouze doporučuje, stejně musí funkce udané meze zarovnat dle stránek, jelikož je to nejmenší jednotka, se kterou tato funkce pracuje.

Lze navrátit jen část namapované paměti. Lze namapovat funkcí mmap() například 3 stránky a následně funkcí munmap() navrátit prostřední stránku. První a třetí stránka zůstane beze změny.

2.4.4 Změna velikosti mapování

K změně velikosti již namapované paměti slouží funkce mremap(). Tato funkce již dříve namapovanou paměť zvětší nebo zmenší na požadovanou délku.
Použití této funkce v jazyku C z [10]:

```
#define _GNU_SOURCE
#include <sys/mman.h>
void *mremap(void * old_address , size_t old_size ,
    size_t new_size , int flags , ...);
```

Funkce se pokusí mapování na adrese *old_address* o velikosti *old_size* změnit mapování o velikosti *new_size*. Při požadavku o zvětšení mapování

se funkce v základním chování pokusí mapování lineárně rozšířit. Pokud to nelze, poté funkce vrací chybový návratový kód.

Parametr příznaků *flags* slouží k upřesnění chování funkce. Jsou dva příznaky:

- **MREMAP_MAYMOVE** – Při zadání tohoto příznaku může funkce při rozšiřování mapování hledat vhodný paměťový rozsah na nové adrese. Je tedy možné, že nově namapovaný prostor bude na jiné adrese, než kde byl původní.
- **MREMAP_FIXED** – Příznak je obdobný jako *MAP_FIXED* funkce mmap(). Při jeho zadání se původní mapování přesune na specifikovanou adresu. Pokud se na této adrese nacházelo nějaké mapování, poté je ztraceno. Tento příznak lze použít pouze současně s příznakem *MREMAP_FIXED*

2.4.5 Nelineární mapování

Lineární mapování umísťuje stránky do fyzické paměti za sebou v pořadí, jak na sebe navazují. Nelineární mapování umí umístit stránky v definovaném pořadí, které může změnit pořadí stránek. Nelineární mapování tedy nemusí zachovávat souslednost jednotlivých stránek. [10]

Pro potřeby nelineárního mapování je v linuxových jádrech systémové volání remap_file_pages(). Toto volání slouží k přesunutí části namapované paměti v rámci namapovaných bloků funkcí mmap().

Toto volání lze využít pouze na namapované bloky v režimu sdílení mezi procesy [10]. Diplomová práce je zacílena na činnost jednoho procesu, proto se touto variantou dále nezabývá.

2.5 Možné typy sdílení mezi vlákny

Vlákna mohou sdílet proměnné na logické a fyzické vrstvě.

2.5.1 Sdílení pohledem logické vrstvy

V případě logické vrstvy lze určit, jestli jednu proměnnou využívá více vláken. Vlákna mohou proměnnou číst, nebo mohou do proměnné zapisovat, a nebo mohou vykonávat s proměnnou obě operace. Podle toho lze proměnné rozdělit do kategorií, jestli například proměnnou využívalo pouze jedno vlákno nebo proměnnou více vláken pouze četlo.

Ke klasifikaci jsem vybral tyto typy proměnných, které bude analyzátor klasifikovat:

- proměnná čtená pouze jedním vláknem

- proměnná čtená více vlákny
- proměnná zapisovaná pouze jedním vláknem
- proměnná zapisovaná více vlákny
- proměnná čtená i zapisovaná jedním vláknem
- proměnná čtená i zapisovaná více vlákny
- proměnná čtená jedním vláknem a zapisována více vlákny (jeden čtenář, více písařů)
- proměnná čtená více vlákny a zapisována jedním vláknem (jeden písař více čtenářů)

2.5.2 Sdílení pohledem fyzické vrstvy

Na fyzické vrstvě je sdílení proměnných spjato s cache bloky. To způsobuje, že proměnné v jednom cache bloku se navzájem ovlivňují. Podle [13] jsou klasifikovány 4 cache výpadky:

- *Cold start miss* – Nastává při prvním přístupu k cache bloku.
- *True sharing miss* – Jedná se o načtení cache bloku do jádra, které chce zapisovat do proměnné v tomto cache bloku, a cache blok byl předtím načten v jiném jádře, kde je zneplatněn. Pokud cache blok migruje mezi cache pamětmi kvůli přístupu ke stejné proměnné, kterou používalo jiné vlákno, potom se jedná o True sharing miss.
- *False sharing miss* – Podobně jako u True sharing miss zde jde o přesun cache bloku do jiného jádra a v původním jádře je blok zneplatněn. V případě False sharing miss se jedná o přístup k jiné proměnné, než jakou používalo předchozí jádro.
- *Eviction miss* – Tento případ nastává, pokud je cache blok vytěsněn z cache paměti jádra, a poté se přistupuje k proměnné z vytěsněného cache bloku.

Podle zadání se má analyzátor soustředit na režim sdílení proměnných a jim odpovídajících cache bloků. Tomuto nejvíce odpovídá true share miss a false share miss, proto se analyzátor zaměří na toto sdílení.

Kapitola 3

Návrh a realizace analyzátoru

Tato kapitola se zabývá návrhem struktur pro zachycení, jak je využívána paměť sledovaného programu. V první části jsou analyzovány potřebné struktury a požadavky, které jsou na tyto struktury kladeny. Následuje podkapitola s popisem, jak se pojmenovávají dynamicky vytvářené rozsahy. Poté následuje popis implementace těchto struktur a jejich realizace ve zvoleném programovacím jazyce. Dále je popsána analýza a realizace načítání dat z mapy sestavení.

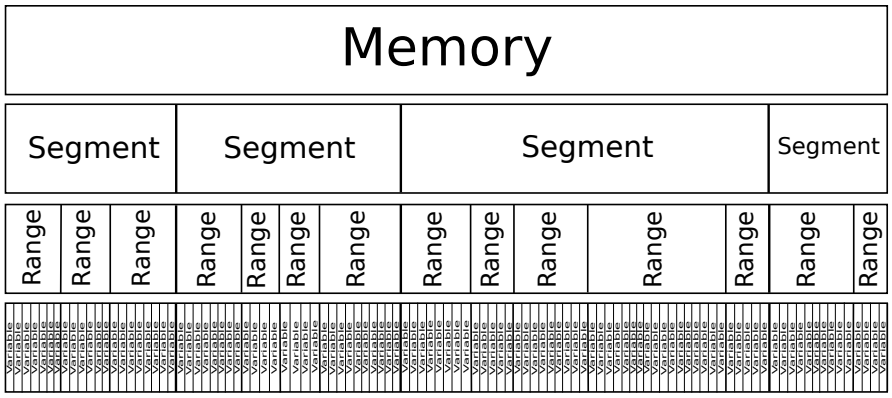
Jelikož je simulátor gem5 implementován v jazycích C++ a Python, rozhodl jsem se nástroj využívající trasovací výstup tohoto simulátoru implementovat v jednom z těchto jazyků. Vzhledem k rozsahu velikosti trasovacích souborů jsem zvolil jazyk C++ pro jeho vyšší rychlost.

3.1 Návrh a požadavky paměťových struktur

Paměťový prostor se organizuje do logických celků, které spolu souvisí. Příkladem je rozdělení do segmentů jako je na obrázku 2.1. Paměť uvnitř segmentů se dále dělí, buď přímo na proměnné, nebo na rozsahy alokované pomocí funkcí rodiny malloc 2.3. Rozsahy alokované rodinou funkcí malloc se dále skládají z proměnných. V každé vrstvě je tedy paměť spojena do celků specifických pro danou vrstvu. Celky ve stejné vrstvě mají specifickou obsluhu a chování, které je mezi vrstvami odlišné. Cílem tohoto členění je zpřehlednit a usnadnit využití fyzické paměti, která takto získává logické členění.

Pro účely programu je paměť brána v pohledu 4 vrstev. Paměť (Memory) lze brát jako jeden celek. Tento celek je poté složen ze segmentů (Segment). Segmenty jsou souborem rozsahů (Range), buď alokovaných funkcemi rodiny malloc, nebo promněnných. Každý tento rozsah je dále členěn na proměnné (Variable). Tyto vrstvy jsou zobrazeny na obrázku 3.1.

Paměťové struktury jsou navrženy tak, aby pokryly paměť ve všech čtyřech úrovních, jak je zobrazeno na obrázku 3.1.



Obrázek 3.1: Vrstvy virtuální paměti počítače navržené k realizaci programu.

3.1.1 1. vrstva – Paměť (Memory)

První vrstva pokrývá celý paměťový prostor. Je určená k seskupení segmentů v druhé vrstvě a jejich zastřešení do jednoho místa, odkud se řídí. Tato vrstva je přístupovým bodem k paměti.

Třída realizující tuto vrstvu bude zpracovávat požadavky na paměť jako jsou například změny paměti po provedení funkcí uvedených v kapitole 2, čtení paměti a zápis do paměti, centrální sběr naměřených statistik nebo zobrazení stavu paměti.

3.1.2 2. vrstva – Segmenty (Segment)

V druhé vrstvě jsou segmenty. Segmentů je několik typů, jak je rozebráno v kapitole 2. Každý uchovává jiný druh dat a jinak s nimi zachází. Proto je třeba implementovat více druhů segmentů.

Základní požadavky na segmenty:

- Spravování rozsahů v jim přidělené části paměti.
- Zpracování čtení a zápisu v segmentu a předání informace o čtení nebo zápisu odpovídajícímu rozsahu ve třetí vrstvě.
- Sbírání statistik paměti jim přidělené.
- Vypsáníinforamcí o sobě a zprostředkování výpisů informací rozsahů, které spravují.
- Přemístění se na jinou adresu, kdyby byly přemístěny funkcí mremap() poslanou v 2.4.4.

Od ostatních segmentů se liší textový segment. Tento segment uchovává kód programu. Není jako jediný dělen na rozsahy, ale do vstupních bodů a jejich implementace. Vstupní bod označuje začátek funkce, poté následuje kód funkce. Tento celek je v práci označován jako TextEntry. TextEntry bude uchovávat jiná data než rozsah u ostatních segmentů, což je popsáno v další části u popisu třetí vrstvy.

Ostatní segmenty lze rozdělit na neměnné a průběžně měnící se. Neměnné jsou před spuštěním programu nahrány do paměti a dále se jejich struktura nemění. Průběžně měnící se segmenty během vykonávání programu mohou mít různé množství rozsahů. Jsou to halda, zásobník a segmenty vytvořené funkcí mmap (déle označované jako mmap segmenty). Všechny tyto segmenty musí být schopné přidávat a odebírat rozsahy podle příkazů z první vrstvy.

Zásobníkový segment se od ostatních průběžně se měnících segmentů odlišuje tím, že v něm jsou ukládány specifické rozsahy určité struktury zvané rámce zásobníku (StackFrame). Tyto rámce na sebe navazují a to od nejvyšší adresy rozsahu zásobníku směrem k nejnižší adrese.

3.1.3 3. vrstva – Rozsahy (Range)

Rozsahy jsou mezistupni mezi segmenty a proměnnými. V různých segmentech plní rozsahy různé funkce. Každý rozsah musí zodpovídat za paměť jemu přidělenou a spravovat operace nad touto pamětí. Rozsah musí umět:

- Zpracovat čtení v tomto paměťovém rozsahu a zápis do tohoto paměťového rozsahu i s předáním operace odpovídající struktuře ve čtvrté úrovni.
- Vypsat a sbírat statistiky o paměti v jemu přidělené části programu.
- Až na dále uvedenou výjimku musí každý rozsah umět přidat a odebrat proměnnou.
- Přesunout se na jinou adresu pro případ, že by se volala funkce realloc() nebo se přemístil celí segment funkcí mremap() popsaných v kapitole 2.

Odlišné rozsahy jsou v textovém segmentu a to vstupní body funkcí s jejich tělem, dále označené jako TextEntry. Tyto rozsahy uchovávají pouze informace o funkci, ale již není třeba, aby byly dále členěné. V tomto segmentu jsou uloženy pouze data instrukcí. Upravený trasovací výstup simulátoru vypisuje pouze paměťové operace datové části paměti. V instrukční části paměti programu se pohybuje pouze ukazatel aktuální pozice v paměti Program Counter (PC). Bylo by tedy zbytečné zde aplikovat čtvrtou vrstvu pod TextEntry.

Specifický je i rozsah v segmentu zásobník. Je to rámec zásobníku StackFrame. Tento rámec má specifickou strukturu popsanou v 2.1. Bohužel z výpisu trasování nelze zjistit počet parametrů funkce. Funkce jsou od sebe v trasovacím výpisu odděleny výpisem volání funkce a návratem z funkce. Mezi

tímto jsou paměťové operace, ale nelze rozlišit, jestli patří k tělu funkce nebo přípravě parametrů volané funkce. Jak je zobrazeno na obrázku 2.2, tak oddělovačem parametrů a lokálních proměnných jsou base pointer (BP) a návratová hodnota (return address). Ve výpisu je součástí volání funkce i uložení návratové hodnoty. Proto adresa návratové hodnoty definuje v programu rámec funkce. Vůči této adrese jsou i relativně adresovány ostatní proměnné na zásobníku.

Některé rozsahy lze identifikovat jménem, které je uvedené v mapě sestavení programu. U těchto rozsahů se musí uložit jméno, díky kterému ho lze poté identifikovat. Rozsahy vytvořené během vykonávaní programu už lze identifikovat jen názvem funkce, která ho vytvořila, a kontextem volaných funkcí, když byl rozsah vytvořen.

3.1.4 4. vrstva – Proměnné (Variable)

Proměnné jsou nejnižší vrstva pohledu na paměť. Tato struktura reprezentuje proměnné, se kterými sledovaný program pracuje. Každá proměnná musí uchovávat své statistiky čtení a zápisů, podle kterých se bude určovat, v jakém režimu je využívána.

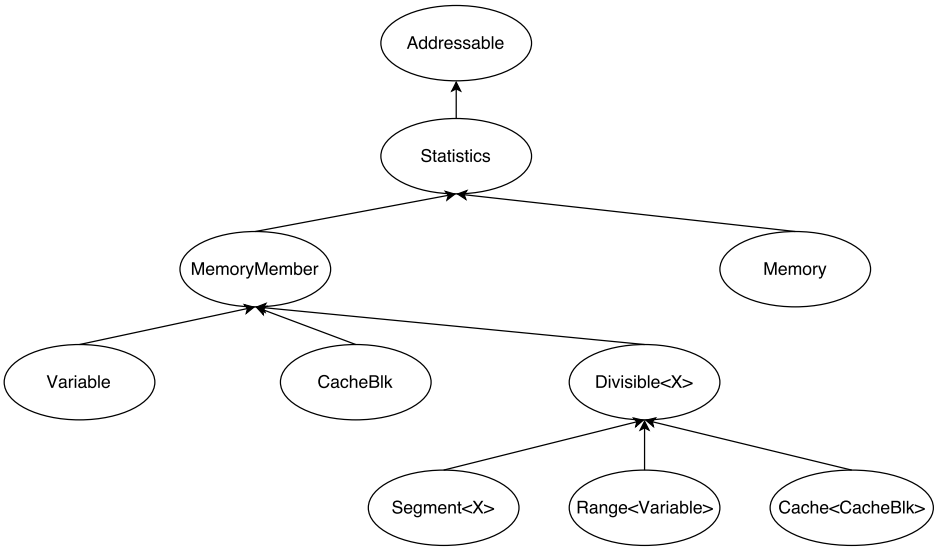
Každá proměnná je také svázána s cache blokem, ve kterém se nachází. Musí si s ním udržovat spojení, aby mohla při sběru statistik zjistit, jak je s ní zacházeno v rámci cache bloku. Jelikož velikost proměnné může být větší než velikost cache bloku, musí tomu být uzpůsobené ukládání referencí.

3.2 Identifikace dynamicky měněných rozsahů

Jelikož rozsahy alokované na haldě, namapovaná paměť a rámce zásobníku mohou za běhu programu vznikat i zanikat, je třeba někde uchovat informace o již neexistujících rozsazích. Také je třeba identifikovat rozsahy, které v paměti pořád jsou reprezentovány strukturami, aby šlo určit, kde a kdy tyto rozsahy vznikly.

Rozsahy alokované na haldě budou pojmenovány podle funkce, která je alokovala, a podle ofsetu v kódu funkce, kde se zavolá alokační funkce. Pro lepší orientaci je za lomítkem uvedena funkce, která paměť přidělila. Například funkce mojeAlokace() hned po spuštění zavolá funkci malloc(). Název alokovaného rozsahu bude *mojeAlokace()+0x0/malloc*. Proměnné budou pojmenovány v rámci rozsahu a to relativně vůči adrese alokace. Pokud bude rozsah obsahovat proměnné o velikosti 4 byty, potom se bude první proměnná jmenovat *mojeAlokace()+0x0/0x0*, další proměnná v rozsahu se bude jmenovat *mojeAlokace()/0x4*, další *mojeAlokace()+0x0/0x8* a tak dále. Funkce, která rozsah alokovala zde již uvedena není. Je to redundantní informace, kterou lze dohledat podle funkce a ofsetu.

Rámce zásobníku jsou pojmenovány podle funkce, pro kterou byly vytvořeny. Proměnné jsou pojmenovány relativně vůči adrese návratové adresy. Jak



Obrázek 3.2: Struktura dědičnosti tříd implementující paměťové struktury.

je popsáno v 2.1, proměnné s nižší adresou než návratová adresa jsou lokální proměnné a proměnné s vyšší adresou jsou parametry funkce. Jelikož z trasovacích výpisů gem5 je problém odlišit konec lokálních proměnných a začátek parametrů nové funkce, jsou parametry funkce v bloku funkce, která ji volala. Při přístupu k parametrům jsou tyto proměnné přejmenovávány správným názvem.

3.3 Implementace paměťových struktur

Detailní popis všech tříd, jejich metod a proměnných je přiložen na CD v podobě dokumentace vygenerované programem Doxygen. V této podkapitole jsou popsány hlavní části nejdůležitějších tříd.

Pro každou úroveň paměti je vytvořena třída, která implementuje požadavky na paměťovou strukturu. U těchto tříd jsou ještě další potomci podle potřeby jednotlivých struktur na dané úrovni. Společnou funkcionalitu implementují třídy Addressable, Statistics a MemoryMember. Strukturu dědičnosti jednotlivých tříd zobrazuje následující obrázek 3.2. Pro přehlednost zde nejsou všechny třídy zobrazeny. Jsou zde zobrazeny třídy, které jsou uvedeny v jednotlivých úrovních paměti.

Následující část se věnuje jednotlivým třídám a jejich popisu.

3.3.1 Třída Addressable

Tato třída je prarodičem všech tříd v paměti. Zde jsou implementovány základní metody pro každou třídu v paměti. V této třídě je implementován me-

chanismus výpisů informací o činnosti paměti a chyb. Také zde jsou abstraktní třídy, které jsou vyžadovány od všech členů paměti.

Každý prvek v paměti uchovává informace o přístupu. Jsou zde definovány dvě proměnné, které slouží jako bitmapa přístupů. Z bitmapy lze vyčíst, které procesory paměťový prvek četly nebo do něho zapisovaly. K ovládání jsou implementovány funkce sloužící k přidávání informací do bitmapy a čtení bitmapy.

Abstraktní metody k implementaci jsou operace čtení a zápisu. Také každý odvozený člen musí implementovat funkci, která vrací jeho velikost. Addressble definuje virtuální metody k určení horní a spodní hranice jeho paměťového prostoru. Pokud si třídy neimplementují vlastní chování, potom spodní hranicí je adresa třídy a horní je adresa sečtená s velikostí prvku. Pro výpis je zde přetížená funkce výpisu, kde lze stanovit hloubku výpisu, kolik vrstev se má vypsat.

K výpisu informací o průběhu zpracovávání trasovacího výpisu je zde implementována statická funkce. Je definováno 6 kategorií rozdělujících zařazení výpisu:

- *rwErrorOn* – výpis informace, kdy došlo k pokusu o čtení nebo zápis neplatné adresy paměti
- *errorOn* – výpis informace, kdy došlo k neplatnému pokusu manipulovat s pamětí
- *rwOperationOn* – výpis informací o čtení paměti nebo zápisu do paměti
- *memoryOperationOn* – výpis informací o změně struktury paměti
- *functionCallOn* – výpis volání funkcí
- *cacheActivityOn* – výpis aktivity cache prvků

U každé kategorie lze nastavit, jestli se má nebo nemá vypisovat. K určení času události je zde proměnná uchovávající čas. Proměnná je statická a musí se o ni starat objekt, který bude zpracovávat trasovací soubor.

3.3.2 Třídy Statistics a MemoryMember

Třída Statistics je abstraktní a vnáší do prvků, které jsou z ní odvozené, metody ke sběru statistik. V této třídě jsou také definovány datové struktury sloužící k ukládání statistik.

Problémem sběru statistik je zajistit, aby se neztratily statistiky tříd, které jsou mazány v průběhu. Jelikož odkaz tříd je takový, že vyšší vrstva má přístup k nižším vrstvám, ale opačně přístup není umožněn, aby nevznikly křížové odkazy, proto musí být zajištěn sběr statistik jinak. Ve třídě Statistics je statická proměnná *Statistics *Worker*, která se bude starat o zpracování takto odstraněných tříd.

Pro úroveň segmentů, rozsahů a proměnných jsou zde implementovány metody k nastavení statistik a jejich odeslání ke zpracování zodpovědné třídě. Pokud je proměnná Worker přiřazena třídě, poté tato třída bude zpracovávat statistiky. Pokud proměnná zůstala prázdná, statistiky odeslané během provádění programu budou ztraceny.

Třída MemoryMember je potomkem třídy Statistics a jejím významem je pouze to, že deklaruje abstraktní metody třídy Statistics. Jsou to metody určené ke zpracování statistik. Tímto jsou třídy ve stromu dědičnosti pod MemoryMember odstíněny od metod ke zpracování statistik za běhu, jelikož by je stejně implementovaly zbytečně. Statistiky zpracovává pouze objekt v proměnné Worker, kterým ale tyto třídy nejsou.

Jelikož struktury pro uložení statistik souvisí se sběrem statistik, jsou u třídy definovány i datové struktury pro uchovávání statistik. Pro každou úroveň je definována jedna struktura.

Struktura Variable uchovává statistiky o proměnné. Její struktura je:

```
struct VariableStatistics{
        uint64 read, write;
        uint64 readVector, writeVector;
        uint64 cacheOperations;
        uint64 cpuChangeInit;
        uint64 cpuChangeAll;
        uint64 cacheRead, cacheWrite;
        uint64 cacheReadVector, cacheWriteVector;
        uint32 cacheBlocksCount;
}
```

Statistiky proměnných obsahují počet čtení proměnné a zápisů do proměnné. Dále jsou bitmapy procesorů, které zacházely s proměnnou. Poté jsou informace o cache bloku, ve kterém se proměnná nacházela. *cacheOperations* udává počet čtení cache bloku a zápisů do cache bloku. Dále je proměnná *cpuCachngeInit*, která udává počet přesunů cache bloku mezi cpu, které se udály kvůli této proměnné. Proměnná *chpuChangeAll* je celkový počet změn cpu daného cache bloku. Následně jsou proměnné uchovávající počet paměťových operací cache bloku a bitmapy přístupů pro cache blok. Proměnná *cacheBlocksCount* udává, přes kolik bloků byla proměnná uložena.

Struktura Range uchovává statistiky o rozsahu. Její struktura je:

```
struct RangeStatistics{
        string name;
        uint64 addr;
        bool myDefined;

        uint64 variables;
        uint64 others;
```

```
        uint64 notUsed;
        uint64 readOnlySingle;
        uint64 readOnlyMulti;
        uint64 writeOnlySingle;
        uint64 writeOnlyMulti;
        uint64 rwSingle, rwMulti;
        uint64 oneWriterMultiReader;
        uint64 oneReaderMultiWriter;

        uint64 acrossMoreBlocks;
        uint64 falseSharing;
        static const float falseShareRate = 1;
        static const uint64
           falseSharingMinimalOperations = 1;
        static const float trueShareRate = 1;
    static const uint64 trueSharingMinimalOperations =
       1;
    uint64 trueSharing;
}
```

Tato struktura uchovává jednotlivé počty typů proměnných, které jsou sledovány. První tři proměnné uchovávají informace o rozsahu, ze kterého statistiky pochází. Je to z důvodu, aby bylo možné alespoň částečně určit původ statistik. Proměnná *myDefined* značí, jestli rozsah je definován programátorem v analyzovaném programu.

V druhém bloku jsou proměnné uchovávající počty zastoupení jednotlivých typů. Postupně shora dolů proměnné určují:

- *variables* – celkový počet proměnných v rozsahu
- *others* – počet nekategorizovaných proměnných
- *notUsed* – počet proměnných bez jediného zápisu nebo čtení
- *readOnlySingle* – počet proměnných pouze čtených jedním vláknem
- *readOnlyMulti* – počet proměnných pouze čtených více vlákny
- *writeOnlySingle* – počet proměnných pouze zapisovaných jedním vláknem
- *writeOnlyMulti* – počet proměnných zapisovaných více vlákny
- *rwSingle* – počet proměnných čtených i zapisovaných jedním vláknem
- *rwMulti* – počet proměnných čtených i zapisovaných více vlákny

- *oneWriterMultiReader* – počet proměnných zapisovaných jedním vláknem
- *oneReaderMultiWriter* – počet proměnných čtených více vlákny a počet proměnných čtených jedním vláknem a zapisovaných více vlákny

Poslední blok proměnných je vyhrazen statistikám týkajících se vztahu proměnných a využívání cache bloku. Proměnná *acrossMoreBlocks* označuje počet proměnných, které se nevešly do jednoho cache bloku a její části byly rozloženy ve více cache blocích. Další proměnná *falseSharing* označuje počet proměnných, jejichž cache blok migruje mezi jádry, ale způsobuje to přístup k jiným proměnným ve stejném cache bloku. K uchování počtu proměnných true sharing je proměnná *trueSharing*. K budoucímu rozšíření jsou zde ještě u každého druhu sdílení další dvě proměnné. Proměnnou *SharingMinimalOperations* lze využít k definování minimálního počtu migrací cache bloku a tím detekovat proměnné, u kterých se sdílení vyskytuje častěji. Proměnná *ShareRate* určuje hranici od jakého poměru má být sdílení zaznamenáno. Podíl změn jádra vyvolaných touto proměnnou musí být menší než *falseSharing* krát celkový počet změn jádra cache bloku.

Vyhodnocení statistik proměnné a její kategorizování do jedné z výše uvedených typů implementuje metoda přiřazená struktuře RangeStatistics. Přiřadit proměnnou do režimu sdílení mezi vlákny se provádí na základě počtu přistupujících vláken. Jestli je proměnná v režimu true nebo false sharing se určuje podle počtu migrací cache bloku vyvolaných proměnnou a celkového počtu migrací.

Jelikož je více typů proměnných, tak při výpisu statistik rozsahu se vypisují pouze typy, které obsahují alespoň jednu proměnnou tohoto typu.

Struktura Segment uchovává statistiky o segmentu. Její struktura je:

```
struct SegmentStatistics{
        SegmentType type;
        uint64 segmentAddress;
        uint64 segmentSize;
        uint64 ranges;
        uint32 maxRangeList;
        RangeStatistics **rangeList;
}
```

Statistiky segmentu jsou navrženy tak, aby uchovávaly statistiky rozsahů. Nijak je průběžně nezpracovávají. Takto lze poté vyhledávat statistiky konkrétního rozsahu. Potom lze dohledávat statistiky rozsahů, které definoval uživatel a dále statistiky zpracovávat.

Struktura Memory uchovává statistiky o celé paměti. Její struktura je:

```
struct MemoryStatistics{
        SegmentStatistics *start, *data, *dataro, *heap,
            *mainStack;
        uint64 stackCount;
        SegmentStatistics **stacks;
        uint64 mmapCount, mmapMaxCount;
        SegmentStatistics **mmapSegments;
}
```

Jelikož některé typy segmentů jsou v paměti programu pouze jednou, je pro ně vyčleněna proměnná. Proměnná *stacks* slouží k ukládání segmentů zásobníků vláken inicializovaných za běhu programu. Každé vlákno má své místo a lze zjistit, které statistiky patří kterému vláknu. Všechny statistiky mapovaných segmentů jsou uložené v poli odkazů *mmapSegments*.

3.3.3 Třída Divisible

Tato třída implementuje pole prvků a operace s nimi. Je to generická třída, aby prvky mohly být takového typu, který odpovídá požadavkům dědičné třídy. Třída je předkem všech tříd, které jsou v druhé a třetí vrstvě a skládají se z dalších prvků nižší vrstvy.

Prvky jsou do pole přidávány podle spodní adresy prvku a jsou v poli vzestupně seřazeny. Pokud se třídy odvozené z Divisible nechovají specificky jsou určeny adresou zděděnou z Addressable a velikostí, kterou určuje Divisible posledním prvkem.

Třída implementuje pole adres na prvky. K tomuto poli implementuje metody, které s ním manipulují. Jsou zde metody k přidání prvku, jeho vyhledání i smazání. V daném paměťovém rozsahu také lze prvky vyhledávat nebo zjistit jejich počet.

3.3.4 Třída Memory

Třída Memory sdružuje všechny segmenty. Paměťové požadavky směřují na tuto třídu, která zprostředkuje jejich splnění správným segmentem. V této třídě je realizováno zpracování mapy sestavení. Také zde je implementováno zpracování trasovacího souboru.

Třída implementuje metody k práci s pamětí. Jsou zde implementovány funkce k mapování segmentů. Dále jsou zde funkce k alokování rozsahů. Také je zde implementovaný zápis a čtení. Hlavním úkolem těchto metod je správně určit segment, kterého se operace týká a nechat odpovídající objekt vykonat příslušnou operaci. K vyhledávání segmentů je implementována funkce, která vrací odkaz na RangeSegment. Tím je Textový segment vyčleněn z vyhledávání. Jelikož je účel textového segmentu odlišný, toto vyčlenění nevadí.

Třída obsahuje několik typů výpisů struktury paměti a statistik. V obou případech lze vypsat vše nebo nechat vypsat jednotlivé segmenty. Lze nechat vypsat informace podle typu segmentu nebo podle adresy. Lze řídit i hloubku výpisu, kolik vrstev se má vypsat.

V této třídě jsou implementovány metody ke sběru statistik. Podle úrovně, ze které statistiky pochází, jsou předány o jednu vrstvě výše. Podle adresy je nalezen odpovídající rozsah nebo segment a jemu jsou statistiky předány ke zpracování. Pokud jsou to statistiky segmentu, potom je zpracuje přímo třída Memory.

3.3.5 Třída Segment

Třída Segment je abstraktní a generická třída. Je to způsobeno tím, že je zde více odlišně se chovajících segmentů.

Generická je proto, aby nemusela specifikovat generický typ při dědění z předka Divisible. Textový segment uchovává úplně jiné rozsahy zvané TextEntry, čímž se odlišuje od ostatních segmentů. Abstraktní metodou je funkce k inicializování rozsahů. Každému segmentu přichází stejná data, ale každý s daty nakládá jinak, proto je třída abstraktní.

Třída implementuje společné metody ke sběru statistik. Také implementuje metody čtení a zápisu. Jelikož v některých segmentech při paměťové operaci lze přistoupit do paměťového prostoru, kde se nyní ještě nenalézá rozsah, jsou zde inicializovány 2 proměnné, které tuto informaci řídí. Každý odvozený segment jen v konstruktoru třídy Segment nastaví, jestli se smí při čtení a zápisu vytvořit nový rozsah.

Na obrázku 3.3 je zobrazena struktura tříd všech segmentů. N8sleduje popis jednotlivých tříd.

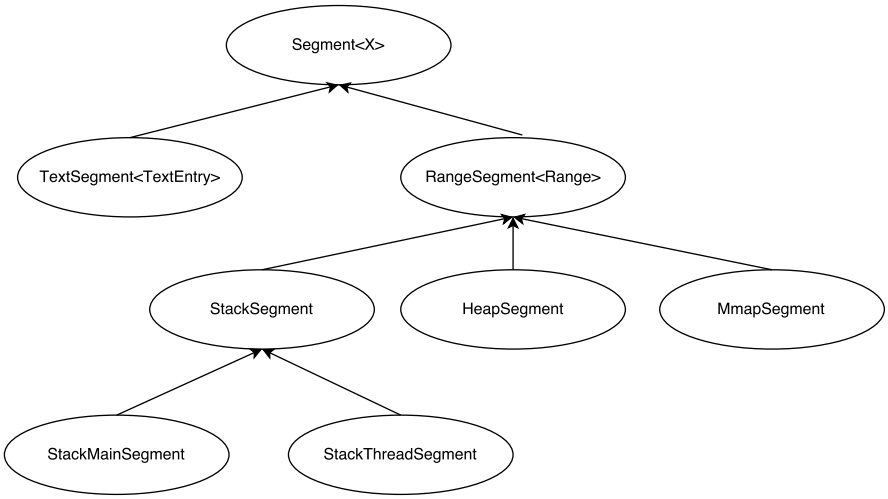
3.3.5.1 Třída TextSegment

Textový segment, kromě implementace inicializační funkce, má ještě jednu funkčnost navíc. Musí kontrolovat přidávané funkce, jestli se nejedná o vybranou funkci pracující s pamětí. Jsou to funkce malloc(), calloc(), realloc(), free(), mmap(), munmap(), brk() a sbrk() rozebrané v kapitole 2. U těchto funkcí ukládá vstupní adresy, aby se poté při provádění činnosti sledovaného programu mohlo reagovat na změny v paměti způsobené těmito funkcemi.

3.3.5.2 Třída RangeSegment

Třída RangeSegment spojuje všechny segmenty kromě Textového. Nastavuje tedy generický typ rodičovské třídy Segment na třídu Range.

Jelikož ne všechny segmenty podporují alokaci prostoru při přístupu k paměťovému prostoru, jsou v třídě dvě proměnné, které tuto informaci uchovávají. Jedna proměnná udává, zda při čtení nealokovaného prostoru zde vytvořit rozsah. Druhá proměnná udává to samé pro zápis do nealokovaného prostoru.



Obrázek 3.3: Struktura zobrazující dědičnost jednotlivých tříd realizujících segmenty

Na základě toho pak rozhodují alokační funkce, zda se mohou provést. Jsou zde implementovány funkce k alokaci, přesunu a uvolnění rozsahu paměti.

Jsou zde také implementovány funkce k alokaci rozsahu při paměťové operaci mimo existující rozsah v daném segmentu. V případě, že to segment podporuje, se poté vytvoří požadovaný rozsah.

Instancemi této třídy jsou statické segmenty inicializované z mapy sestavení. Velikost tohoto segmentu je dána velikostí rozsahů v ní obsažených. Určení velikosti určuje tedy předek Divisible. Instance této třídy nepodporují vytváření rozsahů při paměťové operaci. Pokud by tedy paměťová operace směřoval mimo nějaký rozsah, vyvolala by chybové hlášení(pokud je vypisování tohoto výstupu zapnuto).

Dále jsou specifické třídy s upraveným chováním.

3.3.5.3 Třída HeapSegment

Tato třída je rozšířena o proměnnou DataBreak, která určuje délku tohoto segmentu. Jelikož se během vykonávání programu může velikost haldy měnit, je zde i implementována funkce na zvětší a zmenšení haldy.

3.3.5.4 Třída MmapSegment

Třída MmapSegment reprezenzuje dynamicky mapované úseky paměti funkcí mmap(). Proto musí tato třída implementovat chování mapovaných úseků funkcí mmap().

Funkce mmap() alokuje danou délku segmentu, proto je velikost této třídy dána proměnnou. Jelikož se může velikost segmentu měnit, je zde implemen-

tována metoda ke změně velikosti. Také se může stát, jak je uvedeno v 2.4.3, že bude třeba segment rozdělit, aby byla uvolněna pouze část segmentu.

Jelikož v tomto segmentu lze přímo přistupovat k paměti programu a nemusí se před tím žádat o alokaci programu, vytváří se zde při paměťových operacích automaticky rozsahy. Toto vytváření rozsahů při paměťové operaci je řízeno tak, aby každý paměťový přístup nebyl samotný v jednom rozsahu. Proto se takto vytvořené navazující rozsahy spojují v jeden.

3.3.5.5 Třída StackSegment

Třída realizující zásobník má specifické chování a bylo třeba předefinovat virtuální metody. Jsou to metody určující spodní a horní hranici, jelikož zásobník roste směrem k nižší adrese.

V této třídě se využívají speciální rozsahy StackFrame, které jsou potomky třídy Range.

Paměť určená pro zásobníky vláken se mapuje až při vytvoření vlákna funkcí mmap(), kdy je známa velikost tohoto segmentu. Tento segment má specifickou strukturu, kterou vytváří implementace vláken vytvořená pro simulátor gem5. Tato struktura je popsána v 1.3.2. Při vykonávání programu se tedy přistupuje i k dalším datům než jen k zásobníku. Kvůli tomuto chování je rozlišen zásobník hlavního vlákna a zásobníky vláken vytvořených za běhu.

Třída StackMainSegment se může rozšiřovat pokud je paměťový prostor pod hlavním zásobníkem volný. Přídání StakcFrame při volání funkce je zde jednoduchý. Přidá se jen nový rámec zásobníku do pole rozsahů v Divisible. Odebrání rámce je odebráním prvku s nejnižší adresou.

Třída StackThreadSegment má danou velikost a už ji nelze změnit. V prostoru segmentu u nižší adresy jsou uložena data pro vlákno. Je třeba v tomto paměťovém prostoru udržovat rozsah, který bude uchovávat proměnné. Proto je zde přidání nového rámce komplikovanější o to, že je třeba vložit jej mezi poslední rámec zásobníku a rozsah s daty.

3.3.6 Třída Range

Tato třída implementuje rozsahy ve třetí úrovni paměti. Jsou zde implementovány abstraktní funkce třídy Addressable jako je například čtení a zápis, navrácení velikosti a výpis. Také je zde implementován sběr statistik ze čtvrté vrstvy proměnných, které jsou v daném rozsahu.

Jelikož velikost rozsahu může být v průběhu vykonávání programu měněna, jsou k tomu implementovány potřebné funkce. Rozsah se umí zvětšit i zmenšit podle požadované hodnoty. V případě, že by byl rozsah přesunut, umí změnit svou adresu v paměťovém prostoru. V tomto případě také informuje všechny proměnné ve svém úseku paměti, že se jim změnila adresa.

Jelikož by se mohlo stát, že při uvolnění části segmentu, ve kterém je rozsah umístěn, bude uvolněna jen část rozsahu, je implementována metoda

k rozdělení rozsahu. Také je implementována metoda k uvolnění proměnných v daném úseku paměti rozsahu.

Třída StackFrame je odvozena z třídy Range, ale je uzpůsobena chování paměti zásobníku. Adresou rámce je nejvyšší adresa, která je dnem zásobníkového rámce. Jelikož z trasovacího výpisu nelze poznat, kde končí proměnné funkce a kde parametry volané funkce, je zde na dno rámce uložena proměnná *retAddr*. Další proměnné rámce se přidávají směrem k nižší adrese.

3.3.7 Třída Variable

Třída Variable reprezentuje prvky v poslední vrstvě paměti. Při čtení a zápisu program pracuje právě s fyzickou pamětí, které odpovídají tyto proměnné. Vyšší vrstvy využívá program k logické organizaci proměnných.

Proměnné ve fyzické paměti jsou spojeny s odpovídajícími cache bloky. Jelikož se může stát, že proměnná je ve více cache blocích, je odkaz na bloky uložen polem referencí. Jelikož fyzickou adresu proměnné lze zjistit až při paměťové operaci s proměnnou, je třeba, aby se proměnná při běhu spojila s cache blokem. Proto ve chvíli, kdy je zjištěna fyzická adresa proměnné, získá proměnná odkaz na třídu cache paměti. U této třídy zažádá o odkaz na cache bloky, které spravují jeho adresu. Třída cache vrátí seznam cache bloků, ve kterých se proměnná nachází.

Před odesíláním statistik ještě třída požádá o statistiky cache bloky, ve kterých se nachází. Tyto údaje zahrne do svých a vše odešle třetí vrstvě ke zpracování. Zpracování informací o proměnné implementuje metoda statistik rozsahu v třídě Statistics.

3.3.8 Třída Cache

Třída Cache slouží jako organizační prvek cache paměti. Cache je potomkem třídy Divisible a je složena z cache bloků, které reprezentuje třída CacheBlk. Třída Cache přijímá informace o činnosti cache paměti a déle je předává cache blokům, kterých se to týká.

Jako každá třída odvozená od třídy Addressable i tato implementuje funkce pro čtení a zápis. Jedná se o nalezení cache bloků na fyzické adrese, které se paměťová operace týká, a zavolání jejich metody odpovídající paměťové operace. Pokud by byla paměťová operace přes více bloků, budou všechny tyto bloky informovány.

Dalšími metodami jsou funkce pro přidávání a odebíraní proměnných. Funkce pro přidání proměnné vrací seznam cache bloků, do kterých byla proměnná přidána. Tento seznam následně slouží proměnné, aby mohla zjišťovat stav cache bloku při sběru statistik.

3.3.9 Třída CacheBlk

Třída CacheBlk reprezentuje jednotlivé cache bloky. Tato třída implementuje metody pro čtení a zápis. Také jsou zde metody pro získání statistik o cache bloku.

Aby mohl cache blok informovat všechny proměnné, které jsou v něm umístěny, uchovává si jejich seznam. Tento seznam je realizován proměnnou třídy Divisible<Variable>, která uchovává odkazy na všechny proměnné. Jsou implementovány metody pro přidání a odebrání proměnné.

Při zápisu a čtení cache bloku se vyhodnocuje, jestli cache blok migroval do cache paměti jiného jádra. Cache blok informuje všechny proměnné, které jsou zde umístěny, o migraci. Nynější verze programu předpokládá, že každé vlákno je v samostatném jádře. Program neuvažuje možnost HyperThreading, kdy mohou být 2 vlákna namapována v jednom jádře.

3.4 Načítání dat z mapy sestavení

Mapa sestavení jsou informace linkeru o tom, kde je umístěna jaká část paměti. Tuto informaci vypíše do souboru. K získání této mapy je třeba přidat při sestavování programu, které se provádí v rámci kompilace, parametr *-Wl,-Map,nazev*. Příklad kompilace souboru foo.c pro jeho následnou analýzu programem.

```
gcc -static -o foo -Wl,-Map,foo.map foo.c
```

Pro účely této práce jsou zajímavé úseky s pamětí segmentů. Podle [14] jsou tyto kategorie:

- *.text* – kód programu
- *.rodata* – data určená pouze ke čtení
- *.data* – data ke čtení i zápisu, která jsou inicializována
- *.bss* – data ke čtení i zápisu, která jsou inicializována na nulu

Paměť v úseku *.text* se načte do textového segmentu. Paměť v úseku *.dataro* se načte do segmentu s daty jen pro čtení označovaného DataRO. Paměťové úseky *.data* a *.bss* jsou načteny do segmentu pojmenovaného Data.

Spojení dat do logických celků je v mapě sestavení realizováno odsazením.

Následuje ukázka začátku úseku *.text* pro textový segment (formátování je upraveno, kvůli přehlednějšímu vzhledu):

```
.text      0x00000000004003d0    0x94678
 .text     0x00000000004003d0      0x71 libc.a(check_fds.o)
 .text     0x0000000000400441      0x14 libc.a(malloc.o)
 .text     0x0000000000400455      0xa3 /tmp/cchWVznr.o
```

```
                0x0000000000400455                     my_function
                0x0000000000400475                     main
```

První řádek uvozuje úsek kódu programu. Další řádky začínající odsazeným *.text* jsou knihovny, které mohou dále obsahovat funkce.

Toto je ukázka začátku úseku *.data* (formátování je upraveno, kvůli přehlednějšímu vzhledu):

```
.data           0x00000000006c1080    0x16b0
 .data          0x00000000006c1080      0x4 /usr/.../crt1.o
                0x00000000006c1080            data_start
 .data          0x00000000006c1084      0x0 /usr/.../crti.o
 *fill*         0x00000000006c1084      0x4
 .data          0x00000000006c1088      0x8 /tmp/cchWVznr.o
                0x00000000006c1088            my_variable1
                0x00000000006c108c            my_variable2
```

První řádek uvozuje úsek dat, která jsou inicializována a mohou být jak čtena, tak do nich lze zapisovat. Následují další řádky, které začínají také *.data*, ale s již odsazenými. Tyto řádky označují moduly, které následující paměť požadovaly. Dále jsou vypsány proměnné. Velikost proměnných je potřeba spočítat podle rozdílu adres. Velikost poslední proměnné je třeba spočítat podle velikosti modulu a kolik paměti již bylo využito. Některé moduly jsou bez proměnných. Také se ve výpisu objevují řádky začínající **fill**, které označují zarovnání paměti.

Obě ukázky zobrazují i programátorem definované funkce a data. V první ukázce jsou to funkce *main* a *my_function* a v druhé ukázce proměnné *my_variable1* a *my_variable2*. Všechny tyto informace jsou načteny ze souboru umístěném v */tmp*. Paměť načtená z této složky by měla být programátorem definována.

Velikost haldy lze zjistit z následujícího úseku mapy sestavení:

```
.ldata          0x0000000000ac5338    0x0
                0x0000000000ac5338      . = ALIGN ((. != 0x0) ? 0
                    x8 : 0x1)
                0x0000000000ac5338      . = ALIGN (0x8)
                0x0000000000ac5338      _end = .
                0x0000000000ac5338      PROVIDE (end, .)
                0x0000000000ac5338      . = DATA_SEGMENT_END (.)
```

Právě na posledním řádku je uveden konec datového segmentu, takzvaný Program break. Ten určuje adresu konce haldy.

Kapitola 4

Ověření programu na běhu benchmarků

Prověření funkčnosti programu kromě vlastních testovacích úloh proběhlo na standardizovaných sadách úloh. Zvoleny jsou dvě a to PARSEC a SPLASH.

Tato kapitola se věnuje úpravě úloh, aby bylo možné je spustit v simulátoru gem5 v režimu SE. Dále jsou uvedeny a diskutovány statistiky jednotlivých testovacích úloh.

4.1 Benchmark PARSEC

The Princeton Application Repository for Shared-Memory Computers (PARSEC) je benchmark navržený pro vícevláknové programy. Pozornost je zaměřena na pracovní zátěž pro novou generaci programů sdílejících paměť na více procesorových čipech. [15]

PARSEC obsahuje 13 rozdílných úloh [16]:

- *blackscholes* – výpočet optimálních cen pomocí PDE (Partial Differential Equation)
- *bodytrack* – vyhledávání těla osoby
- *canneal* – optimalizace ceny směrování při návrhu čipu
- *dedup* – komprese nové generace s deduplikací dat
- *facesim* – simulace pohybu lidského obličeje
- *ferret* – serverová aplikace pro porovnávání na základě obsahové podobnosti
- *fluidanimate* – animace dynamiky kapalin pomocí SPH (Smoothed Particle Hydrodynamics) metody

- *freqmine* – identifikace frekvence nálezů vzorů v transakční databázi
- *raytrace* – raytracing v reálném čase
- *streamcluster* – výpočet optimálního shlukování bodů
- *swaptions* – finanční výpočetní aplikace
- *vips* – zpracování obrázků
- *x264* – H.264 video dekódování

Úlohy se snaží pokrýt větší množství domén. Tato sada úloh není zaměřená na jeden konkrétní problém.

V práci je využit balíček PRASEC verze 3.0, který by měl nyní být podle oficiálních stránek nejnovější.

4.1.1 Struktura balíčku PARSEC

Zde je popsána adresářová struktura PARSEC benchmarku.

```
parsec-3.0/
├── bin/ ..... spustitelné programy
├── config/ ..... globální konfigurační soubory
├── log/ ..... trasovací soubory kompilací a běhů
├── man/ ..... manuálové stránky pro PARSEC
└── pkgs/
    ├── apps/ ..... složky s úlohami
    ├── kernels/ ..... složky s úlohami
    ├── libs/ ..... potřebné knihovny pro úlohy
    └── tools/ ..... nástroje
```

Dále je struktura adresářů jednotlivých úloh.

```
[PACKAGENAME]/ ..... složka s úlohou PACKAGENAME
├── inputs/ ..... archiv se vstupy
├── inst/ ..... složky s jednotlivými instalacemi
├── obj/ ..... složky s dočasnými soubory pro instalace
├── parsec/ ..... lokální konfigurační soubory
└── src/ ..... zdrojové kódy úlohy
```

Ve složkách inputs, inst a obj jsou podsložky s verzí, pro kterou je instance určena. Jsou rozlišeny kompilace pro různé druhy paralelizace a různé druhy instrukčních sad.

Rozšíření o vytvoření mapy sestavení by ji mělo generovat do složky inst k odpovídající verzi úlohy. Uzpůsobení PARSEC úloh, aby šly spustit v gem5 režimu SE, a generování mapy sestavení je popsáno v další části textu.

4.1.2 Testovací prostředí

Od vedoucího diplomové práce Ing. Kašpara jsem měl k dispozici server s sestaveným simulátorem gem5. Na tomto serveru již dříve studenti testovali své diplomové práce týkající se simulátoru gem5. Jsou zde tedy implementovány všechny nedostatky, na které v průběhu narazili.

Na serveru je operační systém Ubuntu 12.10. Verze gem5 simulátoru je 2.0 a využíval jsem verzi sestavení opt. K překladu analyzátoru je využíván kompilátor g++ verze 4.7.2.

Konfigurační soubor pro simulaci programu v gem5 jsem používal připravený v adresáři gem5 *configs/example/se.py*.

4.1.3 Kompilace úloh

Oficiální stránky gem5 se zabývají pouze spuštěním v režimu FS. Pro režim SE bude třeba zkompilovat úlohy staticky. Také bude třeba získat mapy sestavení programů. Další komplikací jsou omezení knihovny realizující m5threads vlákna, která jsou popsána v 1.3.3.

Spuštěním úloh benchmarku PARSEC v režimu SE se dříve zabýval Ing. Robert David v diplomové práci Simulation of multiprocessor/multicore system in GEM5 simulator [17]. Velká část diplomové práce je zaměřena právě na zprovoznění těchto úloh. U deseti úloh našel řešení, jak úlohy spustit, u 3 se mu to nepodařilo kvůli časové náročnosti. Jsou to úlohy ferret, raytrace a vips. Těmi se dále zabývat nebudu. Využiji poznatků ke kompilaci úloh a ještě je rozšířím o získání map sestavení programů.

Podle návodu k užívaní úloh PARSEC [18] slouží k instalaci úloh nástroj parsecmgmt, který se nachází ve složce bin/.

V návodu je dále doporučeno přidávat instrukce pro kompilování do konfiguračních souborů. Ve složce config u každé úlohy se nachází globální konfigurace. Zde je upraven soubor *gcc-pthreads.bldconf*, který se týká kompilace úloh využívající pro paralelizaci vlákna. U každého souboru jsou přidány příznaky pro statickou kompilaci, pro vytvoření mapy sestavení a načtení implementace knihovny m5threads.

Je třeba doplnit proměnné CXXFLAGS a LDFLAGS. Zde je jejich rozšíření:

```
CXXFLAGS=${CXXFLAGS} -static -Wl,-Map,map
LDFLAGS=-L$HOME/m5threads ${LDFLAGS} -Xlinker -Map=
    $HOME/map/name.map
```

Takto upravené proměnné je třeba přidat do prostředí při kompilaci. V souboru je rozšířena proměnná *build_env*, kde jsou upraveny předchozí dvě proměnné prostředí.

```
build_env="version=pthreads CXXFLAGS=\"${CXXFLAGS} -
    static -Wl,-Map,map\" LDFLAGS=\"-L$HOME/m5threads ${
    LDFLAGS} -Xlinker -Map=$HOME/map/name.map\""
```

Mapa sestavení je vygenerována v domovském adresáři ve složce map. Knihovna m5threads musí být umístěna v domovském adresáři.

Pokud daná úloha potřebovala ještě nějaké změny, jsou popsány přímo v části textu u této úlohy.

4.1.4 Blackscholes

Úloha nevyžaduje další úpravy, kromě zmíněných v 4.1.3.

Úlohu jsem nejdříve analyzoval z pohledu závislosti na počtu vláken. Naměřené počty typů proměnných v jednotlivých segmentech jsou zaznamenány v následující tabulce 4.1.

Tabulka 4.1: Tabulka zobrazuje počty typů proměnných v segmentech úlohy Blackscholes. R označuje read, W označuje write a O označuje Only.

Segment	Var types	2 thr	4 thr	8 thr	16 thr
Start	RO single cpu	38	38	38	38
DataRO	RO single cpu	363	361	361	347
	RO multi cpu	41	43	43	57
	WO single cpu	13	13	13	13
	RW single cpu	3	3	3	3
	One W multi R	1	1	1	1
Data	RO single cpu	43	44	44	44
	WO single cpu	323	323	323	322
	RW single cpu	82	84	90	103
	One W multi R	13	13	13	13
Heap	WO single cpu	115	115	115	115
	RW single cpu	134	134	134	134
	RW multi cpu	113	115	119	127
	One W multi R	1	1	1	1
	False shared	21	24	28	36
	True shared	1	0	0	0
Main stack	Not used	2392	2400	2416	2448
	RO single cpu	313	313	313	313
	WO single cpu	1299	1299	1299	1299
	RW single cpu	8668	8670	8674	8682
Thread stack A	Not used	3205	1605	805	405
	RO single cpu	5	5	5	5
	WO single cpu	9	9	9	9

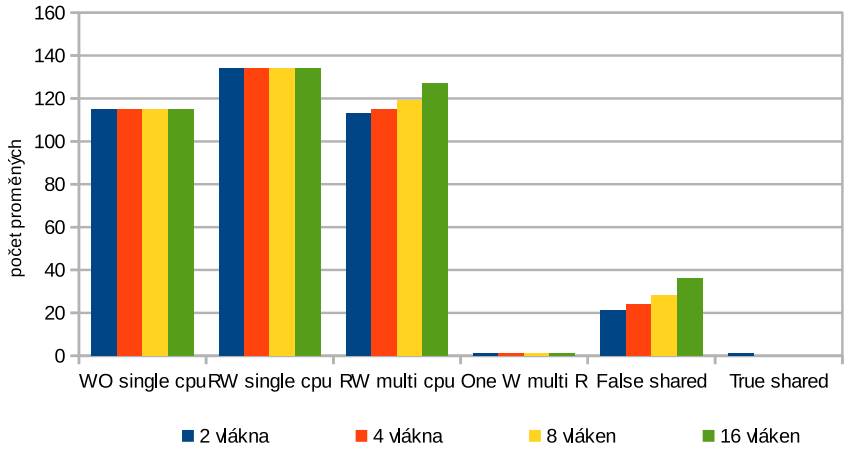
	RW single cpu	20	20	20	20
	RW multi cpu	2	2	2	2
	False shared	7	7	7	7
Thread stack B	Not used	3206	1606	806	406
	RO single cpu	5	5	5	5
	WO single cpu	9	9	9	9
	RW single cpu	20	20	20	20
	RW multi cpu	2	2	2	2
Mmaps	RO single cpu	1025	1025	1025	1025
	WO single cpu	297	297	297	297
	WO multi cpu	2	2	2	2
	RW single cpu	9	9	9	9
	False shared	1	1	1	1
	True shared	1	1	1	1

V segmentu dataRO není žádné sdílení na úrovni cache bloků. U žádného typu proměnných není vidět, že by byla závislá na počtu vláken. Zajímavých je 13 proměnných v režimu pouze zápisu jedním vláknem. V segmentu data se mění o více než jedna jenom počty proměnných v režimu čtení a zápisu jedním vláknem. Počet těchto proměnných roste s počtem vláken, ale nevidím v tom závislost. Proměnné v těchto segmentech se nesdílí na úrovni cache bloků. Proměnné jsou buď v režimu čtení více vlákny, nebo jeden písař a více čtenářů. Model jeden písař více čtenářů by mohl sdílení vyvolat, ale podle další analýzy písař pouze inicializoval proměnné, aby je čtenáři následně mohly číst.

V segmentu heap se již nachází sdílené proměnné v rámci cache bloků. Pro lepší orientaci jsou počty proměnných zobrazeny v grafu 4.1. Zajímavé je, že pouze v případě dvou vláken je jedna proměnná v režimu true shared, v ostatní případech se nevyskytuje žádná. Porovnáme-li dvě konkrétní verze, například 4 a 8 vláken, je zde vidět rozdíl v počtu false shared proměnných. Je to právě rozdíl počtu vláken. Nejspíše nějaký alokovaný rozsah na haldě obsahuje jednu proměnnou pro každé vlákno.

V případě segmentů zásobníků vláken klesá počet nepoužitých proměnných *Not used*. Tyto proměnné jsou návratové adresy funkcí. V analyzátoru je tato proměnná svázaná s rámce zásobníku. Proměnná se čte ve chvíli, kdy se rámec ruší, což v analyzátoru překryje tuto paměťovou operaci. Zápis proměnné probíhá ve chvíli volání funkce a vytváření zásobníku. I zde je paměťová operace překryta. Lze tedy ze snižujícího se počtu nepoužitých proměnných usoudit, že s vyšším počtem vláken každé vlákno provede méně volání funkcí. Odečetla by se hodnota 5, potom by se při zdvojnásobení počtu vláken počet volání funkcí snížil na polovinu.

Zajímavostí je, že se ve statistikách zásobníků vláken nachází 2 různé celkové počty. Tyto dva vzory se liší tím, že jeden vzor zavolá o jednu funkci méně, což je vidět v počtu nepoužitých proměnných, ale neobsahuje proměnné v režimu false shared.



Obrázek 4.1: Graf zobrazuje počty proměnných v segmentu halda pro počty vláken 2, 4, 8 a 16.

V případě mmap segmentů sloučených do jedné statistiky nejsou vidět žádné změny v závislosti na změně počtu vláken.

Dále budu analyzovat segment heap, jelikož je zde nejvíce true a false shared proměnných. K analýze jsem zvolil úlohy s osmy vlákny. Z běhu analyzátoru jsem získal statistiky pro heap segment:

```
Statistic of segment HEAP at address 6e3a78, size 400000
    (contains 9 range statistics)
  Range statistics fopen+0x20/malloc at 6e54f0:
    Context: /__libc_start_main/main/fopen
    variables: 38
      Write only single cpu: 19
      RW single cpu: 19
  Range statistics main+0x431/malloc at 6e5640:
    Context: /__libc_start_main/main
    variables: 8
      RW multi cpu: 8
      False shared: 8
  Range statistics fopen+0x20/malloc at 6e5c20:
    Context: /__libc_start_main/main/fopen
    variables: 38
      Write only single cpu: 19
      RW single cpu: 19
      False shared: 4
  Range statistics main+0x9e/malloc at 6e5730:
    Context: /__libc_start_main/main
```

```
    variables: 144
      Write only single cpu: 48
      RW single cpu: 96
  Range statistics main+0xb5/malloc at 6e5980:
    Context: /__libc_start_main/main
    variables: 16
      RW multi cpu: 15
      One writer multi readers: 1
      False shared: 16
  Range statistics _dl_init_paths+0x3e/malloc at 6e4010:
    Context: /__libc_start_main/__libc_init_first/
       _dl_non_dynamic_init/_dl_init_paths
    variables: 5
      Write only single cpu: 5
  Range statistics _dl_init_paths+0x83/malloc at 6e4040:
    Context: /__libc_start_main/__libc_init_first/
       _dl_non_dynamic_init/_dl_init_paths
    variables: 24
      Write only single cpu: 24
  Range statistics main+0x246/malloc at 6e54f0:
    Context: /__libc_start_main/main
    variables: 16
      RW multi cpu: 16
  Range statistics main+0x1e8/malloc at 6e59d0:
    Context: /__libc_start_main/main
    variables: 80
      RW multi cpu: 80
```

Na haldě bylo v průběhu vykonávání programu alokováno 9 rozsahů. Dva rozsahy jsou alokovány funkcí fopen a dva rozsahy jsou alokovány funkcí _dl_init_paths. Ostatních 5 rozsahů alokovala úloha ve funkci main. Podíváme se dále na tyto rozsahy.

Ve dvou rozsazích jsou proměnné, které jsou sdílené v rámci cache bloku. Jedná se o rozsahy *main+0xb5* a *main+0x9e* oba alokované ve funkci main. Oba dva rozsahy byly uvolněny, takže musím najít místo v log souboru, kde jsou uvolněny a tam najít statistiky o jejich proměnných.

Nejdříve se podíváme na rozsah *main+0xb5*. Tento rozsah obsahuje 16 proměnných. Zde je prvních 6 proměnných (dalších 10 zde není uvedeno, jelikož pro názornou ukázku těchto šest stačí), další pokračují v sekvenci:

```
VAR main+0xb5/0x0 size=4, pa=0x108980(0x108980)
    (000000001-000000011)(r=1,w=101,c=99)
    (000000001-111111111)&(r=16,w=1601,c=1598)
VAR main+0xb5/0x4 size=4, pa=0x108984(0x108980)
    (000000001-000000010)(r=1,w=100,c=99)
```

```
    (000000001-111111111)&(r=16,w=1601,c=1598)
VAR main+0xb5/0x8 size=4, pa=0x108988(0x108980)
    (000000001-000000100)(r=1,w=100,c=99)
    (000000001-111111111)&(r=16,w=1601,c=1598)
VAR main+0xb5/0xc size=4, pa=0x10898c(0x108980)
    (000000001-000000100)(r=1,w=100,c=99)
    (000000001-111111111)&(r=16,w=1601,c=1598)
VAR main+0xb5/0x10 size=4, pa=0x108990(0x108980)
    (000000001-000001000)(r=1,w=100,c=99)
    (000000001-111111111)&(r=16,w=1601,c=1598)
VAR main+0xb5/0x14 size=4, pa=0x108994(0x108980)
    (000000001-000001000)(r=1,w=100,c=99)
    (000000001-111111111)&(r=16,w=1601,c=1598)
```

Na první řádku proměnné jsou informace o její velikosti, fyzické adrese. V závorce je adresa řádku v cache paměti. Na druhém řádku je mapa vláken, které proměnnou četly, a které do proměnné zapisovaly. V druhé závorce jsou počty čtení, zápisů a kolik migrací cache bloku tato proměnná vyvolala. Na třetím řádku jsou informace o cache bloku, ve kterém se proměnná nachází. Tyto informace mají stejnou strukturu jako na předchozím řádku. Každá proměnná téměř při každém zápisu vyvolala migraci cache bloku. Nyní je otázkou, jestli by se vyplatilo tyto proměnné zarovnat na velikost cache bloku a tím eliminovat false sharing. Nevýhodou by bylo zvýšení počtu cache bloků. To by mohlo zapříčinit vytěsnění jiných cache bloků z cache paměti a třeba i zpomalit celkový běh programu. Jestli by toto zarovnání na cache bloky přineslo zrychlení, by bylo třeba experimentálně změřit.

Dalším rozsahem, kde jsou proměnné sdílené v rámci cache bloku, je *main+0x431*. Tento rozsah obsahuje 8 proměnných:

```
VAR main+0x431/0x0 size=4, pa=0x108640(0x108640)
    (000000010-000000001)(r=1,w=2,c=1)
    (111111110-000000001)&(r=8,w=9,c=6)
VAR main+0x431/0x4 size=4, pa=0x108644(0x108640)
    (000000100-000000001)(r=1,w=1,c=0)
    (111111110-000000001)&(r=8,w=9,c=6)
VAR main+0x431/0x8 size=4, pa=0x108648(0x108640)
    (000001000-000000001)(r=1,w=1,c=0)
    (111111110-000000001)&(r=8,w=9,c=6)
VAR main+0x431/0xc size=4, pa=0x10864c(0x108640)
    (000010000-000000001)(r=1,w=1,c=0)
    (111111110-000000001)&(r=8,w=9,c=6)
VAR main+0x431/0x10 size=4, pa=0x108650(0x108640)
    (000100000-000000001)(r=1,w=1,c=0)
    (111111110-000000001)&(r=8,w=9,c=6)
```

```
VAR main+0x431/0x14 size=4, pa=0x108654(0x108640)
    (001000000-000000001)(r=1,w=1,c=0)
    (111111110-000000001)&(r=8,w=9,c=6)
VAR main+0x431/0x18 size=4, pa=0x108658(0x108640)
    (010000000-000000001)(r=1,w=1,c=0)
    (111111110-000000001)&(r=8,w=9,c=6)
VAR main+0x431/0x1c size=4, pa=0x10865c(0x108640)
    (100000000-000000001)(r=1,w=1,c=0)
    (111111110-000000001)&(r=8,w=9,c=6)
```

Celkový počet migrací cache bloku je vyšší než počet migrací, které vyvolaly proměnné v tomto rozsahu. V cache bloku musela být umístěna ještě další proměnná z následujícím rozsahu, která vyvolala zbývajících 5 migrací. V případě těchto proměnných se již nejeví perspektivní je zarovnávat na začátek cache bloků, jelikož každá proměnná je jednou přečtena a jednou zapsána. Nejspíše tento rozsah sloužil k předání hodnot z výpočetních vláken do hlavního vlákna.

4.1.5 Bodytrack

U této úlohy je třeba dalších zásahů, jelikož využívá dalších knihoven. Implementoval jsem postup z [17] pro upravení kompilace. Dále je uvedeno, že při běhu nastává chyba při systémovém volání *madwise*. Je třeba zakomentovat uvolňování paměti v knihovně. Po této úpravě by měla úloha fungovat.

Trasovací soubor úlohy je moc velký a nevešel se na disk, proto není úloha analyzována.

4.1.6 Canneal

Při běhu úlohy nastává chyba. Podle [17] stačí zakomentovat uvolňování paměti v metodě destruktoru jedné třídy a běh úlohy proběhne úspěšně.

Měření jsem provedl pro dvě a tři vlákna. Pro čtyři a více vláken hlásí gem5 simulátor nedostatek fyzické paměti.

Jelikož nyní nejsem schopen získat naměřená data pro více než 3 vlákna, není tato úloha analyzována. Analýza této úlohy bude provedena poté, co se podaří získat data pro více vláken.

4.1.7 Dedup

Tuto úlohu se mi nepodařilo zkompilovat. V [17] žádné speciální zásahy pro spuštění kompilace nejsou uvedeny, pouze změny uvedené v 4.1.3.

4.1.8 Facesim

Podle [17] běh programu v gem5 trvá v řádu dní. Trasovací soubor by v takovémto případě byl obrovský, proto není úloha analyzována.

4.1.9 Fluidanimate

Tato úloha nepotřebuje ke spuštění dalších zásahů, kromě již uvedených změn popsaných v 4.1.3.

Běh úlohy je dlouhý a generuje velký trasovací soubor, který se nevejde na disk testovacího serveru.

4.1.10 Freqmine

Tato úloha využívá k paralelizaci OpenMP, nikoliv m5threads.

4.1.11 Streamcluster

Pro spuštění této úlohy není třeba dalších úprav, kromě dříve uvedených v 4.1.3.

I u této úlohy jsem nejdříve analyzoval závislost sdílení proměnných na počtu vláken. Naměřené údaje jsou v tabulce 4.2. V této tabulce je uveden pouze první a druhý segment zásobníků vláken. První se od druhého liší množinou typů sdílených proměnných. V případech s více vlákny segmenty zásobníků vláken obsahovaly stejné typy proměnných a počty jednotlivých typů byly podobné.

Tabulka 4.2: Tabulka zobrazuje počty typů proměnných v segmentech úlohy Streamcluster. R označuje read, W označuje write a O označuje Only. Thr je zkrácené thread, což je vlákno.

Segment	Var types	2 thr	4 thr	6 thr	8 thr	10 thr
Start	R only single cpu	42	42	42	42	42
DataRO	R only single cpu	425	425	426	425	426
	R only multi cpu	33	33	33	33	32
	W only single cpu	16	16	16	16	16
	RW single cpu	6	6	6	6	6
	one W multi R	2	2	2	2	2
Data	R only single cpu	54	52	52	52	52
	R only multi cpu	3	3	3	3	3
	W only single cpu	1566	1566	1564	1566	1564
	RW single cpu	250	250	252	250	252
	RW multi cpu	4	8	8	8	8
	one W multi R	26	24	24	24	24
	False shared	27	27	27	27	27

	True shared	2	3	3	3	3
Heap	R only single cpu	12	12	12	12	12
	W only single cpu	78	59	59	59	59
	RW single cpu	49	27	28	32	34
	RW multi cpu	60	59	69	66	77
	one W multi R	84	101	106	125	123
	False shared	72	90	120	135	131
	True shared	4	3	5	4	3
Main Stack	Not used	1987	2004	2020	2036	2052
	R only single cpu	1343	1343	1343	1343	1343
	W only single cpu	1170	1196	1196	1187	1196
	W only multi cpu	2	2	2	2	2
	RW single cpu	2587	2603	2607	2610	2615
	RW multi cpu	6	6	6	6	6
	one W multi R	2	2	2	2	2
	False shared	191	185	185	185	185
Thr Stack 1	Not used	34856	45499	56500	67121	73770
	R only single cpu	28161	28161	28136	28001	28106
	W only single cpu	32137	36051	36170	36161	36295
	RW single cpu	21369	24131	25668	26011	26172
	RW multi cpu	1	1	1	1	1
	False shared	6	6	6	6	6
Thr Stack 2	Not used	34298	48617	62153	65187	70934
	R only single cpu	28155	28135	27565	27945	28035
	W only single cpu	28361	28984	29292	29292	29405
	RW single cpu	13532	14482	14547	14551	14582
	RW multi cpu	1	1	1	1	1
Mmaps	R only single cpu	12	12	12	12	12
	R only multi cpu	0	0	0	0	1
	W only single cpu	660	670	676	4816	10252
	W only multi cpu	7	6	6	614	1025
	RW single cpu	71	60	69	1163	1708
	RW multi cpu	14	35	58	10688	19267
	one W multi R	27	27	33	983	1820
	False shared	84	118	145	17581	33388
	True shared	3	0	1	1	0

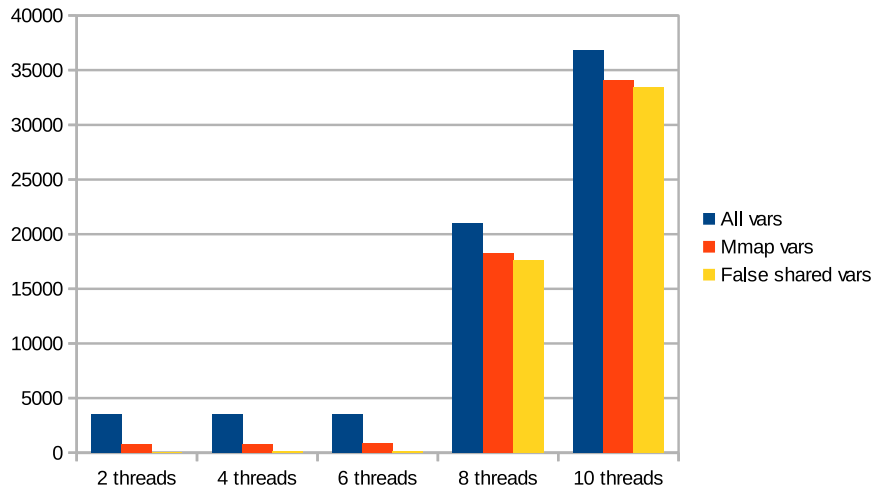
Při pohledu na statistiky jednotlivých segmentů vidím, že při zvyšování počtu vláken roste počet proměnných v segmentech heap a mmaps.

V případě haldy zde rostou počty proměnných *RW multi* a *One W more R*. Oba typy mohou zapříčinit migraci cache bloku, což také způsobují vzhledem k rostoucímu počtu false shared proměnných.

Statistiky mmaps jsou součty statistik všech mmap segmentů. Zde se počty proměnných zvyšují více než u haldy. Nejvyšší nárůst proměnných je mezi

během s šesti vlákny a během s osmi vlákny. Tento nárůst je skokový. Mezi osmi a deseti vlákny je navýšení počtů u většiny proměnných přibližně dvojnásobné. Mezi osmi a deseti vlákny také vzrostl počet proměnných v režimu false shared téměř na dvojnásobek. Počet false shared proměnných je vysoký vzhledem k celkovému počtu proměnných v segmentech mmap.

Celkový počet proměnných a počet proměnných mmap segmentů je porovnaný s množstvím false shared proměnných v mmap segmentu v grafu 4.2. Poměr proměnných v mmap segmentech s rostoucím počtem vláken začne převažovat nad ostatními proměnnými. Téměř všechny proměnné v mmap segmentech jsou na úrovni cache bloků sdílené. Kolik procent všech proměnných a kolik procent proměnných mmap segmentů tvoří false shared proměnné ze segmentů mmap je v tabulce 4.3. V případech s dvěma, čtyřmi a šesti vlákny podíl false shared proměnných vůči všem proměnným programu není příliš vysoký. Vůči proměnným v mmap segmentech je vyšší, ale stále je méně než pětina. Od osmi vláken vzroste zastoupení false shared proměnných velmi radikálně. V případě s deseti vlákny je více než 90 % proměnných typu false sharing. Téměř všechny proměnné v mmap segmentech jsou sdílené.



Obrázek 4.2: Graf zobrazuje množství všech proměnných, proměnných v segmentech mmap a false shared proměnných v segmentech mmap. Do součtu všech proměnných nejsou započteny proměnné v zásobníkových segmentech.

Nyní se podíváme na statistiky proměnných mmap segmentů. K další analýze jsem zvolil běh programu s osmi vlákny. Podle další analýzy byly mapovány tři segmenty. Do jednoho se přistupovalo při inicializaci programu. Do druhého se přistupovalo během ukončování vláken. Třetí segment byl využíván v průběhu činnosti vláken, které v těchto segmentech prováděly paměťové operace. Tento segment obsahuje 634 statistik rozsahů, ale v případě mapovaného segmentu nemusí být toto číslo relevantní. V tomto segmentu lze přímo

Vůči všem proměnným	2,40 %	3,38 %	4,08 %	83,73 %	90,69 %
Vůči mmap proměnným	10,62%	14,57%	16,98%	96,20%	97,96%

Tabulka 4.3: Tabulka zobrazuje procentuální zastoupení proměnných v režimu false sharing. První řádek je poměr těchto proměnných ve všech proměnných programu. Na druhém řádku je zastoupení false shared proměnných v mmap segmentech. Do součtu všech proměnných nejsou započteny proměnné v zásobníkových segmentech.

přistupovat k paměti bez alokace, což může způsobit, že rozsah pojme množinu sousedních proměnných. V segmentu se nachází velké množství rozsahů, které vznikly v průběhu vykonávání funkce *pgain()*. Je jich 438 a vždy mají 36 proměnných. Všechny proměnné jsou false shared. Zdroj [16] uvádí, že právě funkce *pgain()* je pro tuto úlohu stěžejní. Zde tedy vznikají false shared proměnné v této úloze. Jelikož je tato funkce stěžejní pro výpočet, tvoří false shared proměnné takové velké množství z celkového počtu proměnných programu.

4.1.12 Swaptions

Úloha nevyžaduje další změny, kromě změn uvedených v 4.1.3.

Měření jsem provedl pro dvě a tři vlákna. Pro čtyři vlákna hlásí gem5 simulátor nedostatek fyzické paměti.

Jelikož nyní nejsem schopen získat naměřená data pro více než 3 vlákna, není tato úloha analyzována. Analýza této úlohy bude provedena poté, co se podaří získat data pro více vláken.

4.1.13 X264

Tato úloha by podle [17] neměla vyžadovat žádnou další úpravu. Kompilace úlohy proběhne v pořádku. Při běhu úlohy nastane chyba a program havaruje.

4.2 Benchmark SPLASH

SPLASH-2 je banchmark zaměřený ke studiu centralizovaných a distribuovaných multiprocesorů se sdíleným adresním prostorem. [19]

Benchmark SPLASH-2 obsahuje 12 úloh:

- Barnes
- Cholesky
- FFT
- FMM

- LU
- Ocean
- Radiosity
- Radix
- Raytrace
- Volrend
- Water-Nsq
- Water-Sp

Na oficiální stránkách dokumentace gem5 [6] je i stránka o používání SPLASH benchmarku v gem5. Jsou zde uvedeny dvě možnosti, jak spustit SPLASH úlohy v režimu SE. První je využít implementaci PARMACS maker pro režim FS a implementovat obsluhu nutných syscall volání. Druhou možností je vzít nebo vytvořit vlastní implementaci PARMACS knihovny. Existuje jen stará implementace pro architekturu Tru64.

Zprovoznění SPLASH úloh v režimu SE je za těchto okolností velmi náročné a přesahuje rámec této diplomové práce.

Závěr

V první části práce jsme analyzovali funkčnost simulátoru gem5. Jelikož trasovací možnosti pro účely práce nebyly dostatečné, rozšířili jsme simulátor o trasovací příznaky ExecMemory a ExecMemory2. Dále jsme analyzovali použití adresního prostoru procesu. Zde jsme si určili funkce ovlivňující paměť. Také jsme zde rozebrali možné typy sdílení proměnných mezi vlákny. Následoval návrh struktur vyvíjeného analyzátoru. Navržené struktury jsme implementovali v jazyce C++. V poslední fázi práce jsme analyzovali úlohy benchmarků PARSEC a SPLASH. Zprovoznění benchmarků v simulátoru v režimu SE se ukázalo jako obtížné. Povedlo se analyzovat dvě úlohy z benchmarku PARSEC.

Vyvinutý analyzátor jsme ověřili a funkčnost demonstrovali při analýze úloh Blackscholes a Streamcluster z benchmarku PARSEC. V úloze Blackscholes při analýze závislostí počtu proměnných dle typů sdílení na počtu vláken jsme neobjevili žádné závislosti. Dále jsme se zabývali analýzou sdílení proměnných na fyzické vrstvě. Zaměřili jsme se na proměnné v režimu false sharing. Tyto proměnné se vyskytovaly v segmentu haldy. Identifikovali jsme dva rozsahy s těmito proměnnými. U jednoho z rozsahů by stálo za zvážení upravit uložení dat v rámci cache bloků. Při analyzování změny počtu vláken u úlohy Streamcluster jsme pozorovali, jak s rostoucím počtem vláken rostl i počet proměnných sdílených v rámci cache bloku. V případě osmi a deseti vláken byla většina proměnných sdílena v rámci cache bloku v režimu false sharing. Tyto proměnné jsme identifikovali v namapovaném segmentu.

Touto ukázkou použití analyzátoru jsme předvedli jen část jeho možností. Analyzátor lze dále využít ke zkoumání lokálních proměnných funkcí. Další funkčností analyzátoru je měřené statistiky jen v definovaných úsecích. Lze se tedy zaměřit jen na paralelní úsek vykonávání kódu. Tyto možnosti využití nejsou z časových důvodů plně otestována, proto nejsou v práci zahrnuty.

Budoucí práce

Možností k rozšíření a dalším pracím je mnoho. Zde jsou ty, kterými se budu v blízké době zabývat:

- **Analýza zbývajících úloh benchmarku PARSEC** – Některé úlohy nejsou analyzovány, jelikož jsme narazili na limity testovacího serveru. Po odstranění těchto limitů lze získat trasovací soubory a ty analyzovat.
- **Rozšiřování analyzátoru** – Analyzátor lze rozšířit a vylepšit v mnoha ohledech, které jsem během práce z časových důvodů nestihl implementovat. Příkladem jsou nyní neimplementované dva výpadky cache bloku *cold start miss* a *eviction miss*.
- **Rozšíření ovládání simulátoru** – Simulátor se ovládá pomocí příkazů, které lze zapsat do souboru. Pro snadnější analýzu částí programu by bylo vhodné rozšíření funkcionality. Například navržení detekce částí, které by mohly uživatele zajímat, protože nyní musí uživatel tyto úseky identifikovat sám.
- **Implementace vytvořených trasovacích příznaků do aktuální verze gem5** – Verze gem5 simulátoru na testovacím serveru není aktuálně nejnovější. V aktuální verzi gem5 by mohly být opraveny některé chyby, které působily potíže.

Literatura

[1] Advanced Micro Devices, Inc.: *AMD SimNow^{TM} Simulator 4.6.1.* [online]. [cit. 2016-03-09]. Dostupné z: http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2012/10/SimNowUsersManual4.6.1.pdf

[2] Computer Systems Laboratory, Stanford University: *The SimOS Simulation Environment.* [online]. [cit. 2016-03-09]. Dostupné z: https://web.archive.org/web/20050830235028/http://simos.stanford.edu/userguide/

[3] Imperas Software Limited: *Technology OVPsim.* [online]. [cit. 2016-03-10]. Dostupné z: http://www.ovpworld.org/technology_ovpsim

[4] Imperas Software Limited: *OVP VMI Memory Model Component Function Reference.* [online]. [cit. 2016-03-10]. Dostupné z: http://www.ovpworld.org/documents/OVP_VMI_Memory_Model_Component_Function_Reference.pdf

[5] Binkert, N.; Beckmann, B.; Black, G.; aj.: *The GEM5 Simulator.* [online]. [cit. 2016-03-10]. Dostupné z: http://dl.acm.org/citation.cfm?doid=2024716.2024718

[6] *GEM5 official web page.* [online]. [cit. 2016-03-10]. Dostupné z: http://www.gem5.org/Main_Page

[7] Saidi, A.; Hansson, A.: *Simulating Systems not Benchmarks.* [online]. [cit. 2016-03-15]. Dostupné z: http://gem5.org/dist/tutorials/hipeac2012/gem5_hipeac.pdf

[8] Beckmann, B.; Binkert, N.; Saidi, A.; aj.: *The gem5 Simulator – ISCA 2011.* [online]. [cit. 2016-03-18]. Dostupné z: http://www.gem5.org/dist/tutorials/isca_pres_2011.pdf

[9] Sanchez, D.: *m5threads – A pthread library for the M5 simulator*. Stanford University, [cit. 2016-03-23].

[10] Kerrisk, M.: *The Linux Programming Interface : A Linux and UNIX System Programming Handbook*. San Francisco: No Starch Press,US, první vydání, 2010, ISBN 1-59327-220-0.

[11] Irvine, K. R.: *Assembly Language for x86 Processors*. Florida International University, School of Computing and Information Sciences, 7 vydání, 2015, ISBN 0-13-376940-2.

[12] Kerrisk, M.: *Linux Programmer's Manual – malloc(3)*. [online]. [cit. 2016-05-09]. Dostupné z: `http://man7.org/linux/man-pages/man3/malloc.3.html`

[13] Bianchini, R.; Kontothanassis, L.: *Algorithms for Categorizing Multiprocessor Communication Under Invalidate and Update–Based Coherence Protocols*. Department of Computer Science – University of Rochester.

[14] Chamberlain, S.; Taylor, I. L.: *The GNU linker*. [cit. 2016-12-04]. Dostupné z: `https://www.eecs.umich.edu/courses/eecs373/readings/Linker.pdf`

[15] Princeton University: *PARSEC [online]*. [cit. 2016-12-11]. Dostupné z: `http://parsec.cs.princeton.edu/index.htm`

[16] Bienia, C.: *Benchmarking Modern Multiprocessors*. Dizertační práce, Princeton University, 2011.

[17] Robert, D.: *Simulation of multiprocessor/multicore system in GEM5 simulator*. Diplomová práce, České Vysoké Učení Technické v Praze, Fakulta Informačních Technologií, 2014.

[18] Bienia, C.; Li, K.: *The PARSEC Benchmark Suite Tutorial [online]*. Princeton University, [cit. 2016-12-11]. Dostupné z: `http://parsec.cs.princeton.edu/download/tutorial/2.0/parsec-2.0-tutorial.pdf`

[19] Woo, S. C.; Ohara, M.; Torrie, E.; aj.: The SPLASH-2 Programs: Characterization and Methodological Considerations. *SIGARCH Comput. Archit. News*, ročník 23, č. 2, Květen 1995: s. 24–36, ISSN 0163-5964, doi:10.1145/225830.223990. Dostupné z: `http://doi.acm.org/10.1145/225830.223990`

Příloha **A**

Seznam použitých zkratek

PARSEC The Princeton Application Repository for Shared-Memory Computers

SPLASH Stanford ParalleL Applications for SHared memory

BIOS Basic Input/Output System

CPU Central Processing Unit

SLICC Specification Language for Implementing Cache Coherence

FS Full System

SE Syscall Emulation

MOESI Modified Owned Exclusive Shared Invalid

Příloha B

Uživatelská příručka analyzátoru

Analyzátor na příkazové řádce...

B.1 Kompilace analyzátoru a potřebný software

Pro využívání analyzátoru ho nejdříve musíte sestavit. K získávání trasovacích souborů je třeba nainstalovat simulátor gem5. Simulátor je nutné rozšířit o trasovací příznaky, které analyzátor potřebuje.

B.1.1 Sestavení analyzátoru

Sestavení analyzátoru se provede příkazem *make* ve zdrojovém adresáři analyzátoru.

Sestavení analyzátoru bylo testováno v systému Debian s g++ verzí 4.9.2-10 a systému Ubuntu s g++ verzí 4.7.2.

B.1.2 Instalace gem5

Zde je shrnutí informací ze stránky oficiálního webu gem5 na adrese `http://gem5.org/Dependencies`.

Doporučenými operačními systémy pro běh gem5 jsou linuxové platformy a Max OS X. Unixové platformy a Windows platformy by také měly být podporovány, ale nejsou tolik otestovány.

Závislosti na externích nástrojích jsou:

- g++ verze 4.8 nebo novější
- Python verze 2.6 - 2.7 (nejsou podporovány verze 3.x)
- SCons verze 0.98.1 nebo novější
- SWING verze 2.0.4 nebo novější

- zlib aktuální verze
- m4

Simulátor lze získat pomocí verzovacího programu hg, a poté sestavit nástrojem scons. Následuje ukázka:

```
hg clone http://repo.gem5.org/gem5
cd gem5

export M5_PATH=$HOME/gem5
export TARGET_ISA=x86

mkdir build
scons -j Y build/X86/tests/opt/quick
# Y je pocet vlaken
```

Pro běh vícevláknových programů využívající vlákna, je třeba zkompilovat a při kompilaci analyzovaných úloh načíst knihovnu m5threads. Tu získáte takto:

```
hg clone http://repo.m5sim.org/m5threads

# aby nasledujici kompilace nehavarovala, je treba do
   pthread.c pridat radek:
#include <linux/sched.h>

gcc -c m5threads/pthread.c -o m5threads/pthread.o
```

B.1.3 Rozšíření trasovacích příznaků

Před sestavením gem5 je třeba upravit některé soubory se zdrojovými kódy. Na CD v adresáři */src* je soubor *gem5_changes* s veškerými změnami vygenerovaný verzovacím nástrojem hg. Tyto změny je třeba implementovat.

B.2 Ovládání analyzátoru

Analyzátor při spuštění přijímá parametr, kde je soubor s příkazy. Pokud žádný parametr není použit, potom analyzátor rovnou přechází do interaktivního režimu.

Příkazy lze zapsat do souboru, který analyzátor zpracuje, nebo je zapisovat interaktivně v příkazové řádce. Při zpracování souboru z příkazy jsou ignorovány prázdné řádky a řádky začínající symbolem #. Po zpracování všech řádek příkazového souboru se analyzátor přepne do interaktivního režimu. Pokud analyzátor načte příkaz *exit*, potom se ukončí.

Pro vypsání nápovědy slouží příkaz Help nebo symbol ?. Analyzátor vypíše seznam všech příkazů. Příkazy lze rozdělit na konfigurační a vykonávající. Konfigurační příkazy by měly být zadány dříve, než se spustí běh analyzátoru. Konfigurační příkazy jsou:

- **CpuNumber** – Nastaví celkový počet vláken (Hlavní vlákno je třeba také započítat).
- **PrintSetting** – Nastavuje, které kategorie dat se mají vypisovat na standardní výstup. Příkaz je popsán dále v této podkapitole.
- **MapFile** – Slouží ke specifikaci cesty, kde se nachází mapa sestavení.
- **TraceFile** – Slouží ke specifikaci cesty, kde se nachází soubor s trasovacími daty.
- **FinalStatisticOutput** – Určuje cestu a název souboru, kam budou uloženy finální statistiky při ukončení analyzátoru.

Příkaz nastavuje kategorie k určené k výpisu. Jsou to kategorie popsané v seznamu 3.3.1. Příkaz má strukturu:

```
PrintSetting rwError t
```

Druhé slovo určuje název kategorie. Třetí položka je písmeno, které slouží k zapnutí nebo vypnutí tohoto výpisu. Pokud je zde uvedeno písmeno t nebo číslo 1, potom je daný výpis zapnut. Pokud je zde jakýkoliv jiný symbol, je tento výpis vypnutý.

Druhou skupinou příkazů jsou takové, které vyvolávají činnost analyzátoru. Jsou zde příkazy k zahájení analýzy, vypsání stavu paměti a vypsání statistik paměti.

- **Work** – Zahájí vykonávání trasovacího souboru.
- **WorkToTime** – Zahájí vykonávání trasovacího souboru dokud je čas provedených operací menší nebo roven zadanému času příkazu.
- **WorkInFunction** – Sběr dat do statistik probíhá pouze v definované funkci. Při volání této funkce se vynulují veškeré statistiky. Po návratu z této funkce se ukončí vykonávání trasovacího souboru.
- **Clean** – Vynuluje všechny statistiky.
- **PrintMemory** – Vypíše aktuální stav celé paměti.
- **PrintSegment** – Vypíše stav paměti definovaného segmentu. Segment se uvádí názvem např. Data nebo Heap, pokud je pouze jediný segment tohoto názvu. Segmenty zásobníků se specifikují číslem za položkou Stack-X. Pro vypsání všech mapovaných segmentů slouží identifikátor Mmap.

- **PrintMyDefined** – Vypíše paměť, která byla definována analyzovaným programem.
- **PrintStatistics** – Vypíše statistiky všech struktur v paměti.
- **PrintStatisticsSegment** – Vypíše statistiky definovaného segmentu.
- **PrintStatisticsSummary** – Sloučí statistiky celé paměti do jedné celkové, kterou vypíše.
- **PrintStatisticsMyDefined** – Vypíše statistiky částí paměti, které jsou definovány analyzovaným programem.
- **PrintStatisticsFunction** – Vypíše statistiky lokálních proměnných a parametrů funkcí.
- **PrintStatisticsFunctionMyDefined** – Vypíše statistiky funkcí, které jsou definovány v analyzovaném programu.

Funkce určené k výpisu mohou mít ještě parametr *depth=X*, kde X je číslo. Podle tohoto čísla se provede počet vnoření při výpisu. Například když by bylo X rovno dvěma u vypsání paměti, potom by se vypsala první a druhá vrstva. Detaily o rozsazích se už nevypíší, jelikož jsou ve třetí vrstvě.

Funkce končící textem *MyDefined* vypisují jen struktury, které by měly být definovány nebo alokovány uživatelem. Jedná se o globální proměnné definované v kódu analyzovaného programu a alokovanou paměť rodinnou funkcí malloc ve funkcích, které jsou definovány v kódu analyzovaného programu.

Sběr statistik v případě funkcí je odlišný. Jelikož funkce může být volána mnohokrát je zajímavý pohled, v jakých režimech byly proměnné při jednotlivých voláních funkce. Například, jestli při každém volání funkce byl parametr využívám stejně nebo se v některých volání funkce typ parametru lišil.

B.3 Jak získat trasovací soubor

Analyzovaný program je třeba staticky slinkovat. Toho dosáhnete tak, že mezi parametry kompilace nástroje gcc nebo g++ přidáte parametr *–static*.

Dalším důležitým výstupem kompilátoru je mapa sestavení. Mapu sestavení získáte tak, že mezi parametry kompilátoru přidáte:

```
-Wl,-Map,nazev_mapy
```

Pro spuštění více vláknového programu v gem5, je třeba při kompilaci načíst knihovnu implementující m5threads. Její získání je popsáno v B.1.2.

Například kompilace mého programu foo.c by vypadala následovně:

```
gcc -c foo.c -o foo.o
gcc --static -Wl,-Map,foo.map -o foo foo.o m5threads/
    pthread.o
```

Pokud máte takto připravený binární soubor analyzovaného programu, můžete ho spustit v gem5 simulátoru. Příkaz ke spuštění vypadá takto:

```
gem5_bin --debug-file=trace_file --debug-flags=
    ExecMemory,ExecMemory2 system_config_file --caches --
    ruby --num-cpus=X -o "parameters" -c binary_file
~/gem5/build/X86/gem5.opt --debug-file=foo.trace --debug
    -flags=ExecMemory,ExecMemory2 ~/gem5/configs/example/
    se.py --caches --ruby --num-cpus=8  -o "params" -c
    foo
```

První příkaz zobrazuje obecně popsané části spuštění simulátoru. Druh příklad ukazuje, jak by vypadalo spuštění simulace našeho dříve zkompilovaného foo pro osm vláken. Předpokládá se, že gem5 je nainstalován v domovském adresáři podle B.1.2. Pro změnu systému, který má simulovat vykonávaný program lze změnit system_config_file. Zde je využit ukázkový konfigurační soubor předpřipravený vývojáři gem5.

B.4 Popis výpisu analyzovaných dat

Řádky, které odpovídají činnosti vlákna, začínají číslem tohoto vlákna. Řádky o událostech v paměti cache jsou uvozeny slovem *cache*. Poté následuje buď dvojtečka nebo znaménko porovnání. Znaménko větší než (>) znamená, že se volá funkce. Znaménko menší než (<) znamená návrat z funkce. Poté následuje odsazení prázdnými znaky. Jsou to dvě mezery, za každé vnoření do funkce. poté jsou informace, co se událo.

Ukázka volání funkcí main a následně funkce foo:

```
0> main
0:   MemoryOperation ...
0>   foo
0:     MemoryOperation ...
0:     MemoryOperation ...
0<   foo
0< main
```

Příloha C

Obsah přiloženého CD

- `readme.txt` ..... stručný popis obsahu CD
- `exe` ..... adresář se spustitelnou formou implementace
  - `geman` ..... analyzátor
  - `blackscholes.map` ..... mapa sestavení blackscholes
  - `blackscholes.trace` ..... trasovací soubor blackscholes pro 8 vláken
  - `commands` ..... příkazový soubor pro analyzátor
- `src`
  - `impl` ..... zdrojové kódy implementace
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
  - `documentation` ..... dokumentace analyzátoru
  - `gem5_changes` ..... soubor se změnami v gem5
- `text` ..... text práce
  - `thesis.pdf` ..... text práce ve formátu PDF