VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ
DEPARTMENT OF COMPUTER SYSTEMS

RYCHLÝ A ČÁSTEČNĚ PŘEKLÁDANÝ SIMULÁTOR PRO APLIKAČNĚ SPECIFICKÉ PROCESORY

FAST AND PARTIALLY TRANSLATED SIMULATOR FOR APPLICATION-SPECIFIC PROCESSORS

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE **PAVEL RICHTARIK**
AUTHOR

VEDOUCÍ PRÁCE **Ing. MARCELA ZACHARIÁŠOVÁ, Ph.D.**
SUPERVISOR

BRNO 2018

Zadání diplomové práce/21156/2017/xricht21

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav počítačových systémů Akademický rok 2017/2018

Zadání diplomové práce

Řešitel: **Richtarik Pavel, Bc.**
Obor: Bezpečnost informačních technologií
Téma: **Rychlý a častečně překládaný simulátor pro aplikačně specifické procesory**
Fast and Partially Translated Simulator for Application-Specific Processors
Kategorie: Počítačová architektura

Pokyny:

1. Seznamte se s problematikou návrhu aplikačně specifických procesorů.
2. Seznamte se s interpretujícím simulačním jádrem používaným v Codasip s.r.o.
3. Seznamte se s aktuálními trendy v oblasti překládaných simulátorů (translated simulators), např. QEMU, Spike, Imperas.
4. Navrhněte nové simulační jádro pro částečně překládaný simulátor.
5. Nově navržené simulační jádro implementujte.
6. Na vybraných benchmarkových testech porovnejte původní simulační jádro s nově navrženým.

Literatura:

- Zdeněk Přikryl. Ph.D. thesis: Advanced Methods of Microprocessor Simulation
- https://www.qemu.org/
- http://www.imperas.com/
- https://riscv.org/software-tools/risc-v-isa-simulator/

Při obhajobě semestrální části projektu je požadováno:

- Splnění bodů 1 až 4 zadání.

Podrobné závazné pokyny pro vypracování diplomové práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva diplomové práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap, které byly vyřešeny v rámci dřívějších projektů (30 až 40% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Zachariášová Marcela, Ing., Ph.D.**, UPSY FIT VUT
Datum zadání: 1. listopadu 2017
Datum odevzdání: 23. května 2018

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav počítačových systémů a sítí
612 66 Brno, Božetěchova 2

L.S.

prof. Ing. Lukáš Sekanina, Ph.D.
vedoucí ústavu

Abstrakt

Hlavným cieľom tejto práce je analyzovať možnosti využitia simulácie pri návrhu aplikačne špecifických procesorov, preskúmať a porovnať rôzne simulačné techniky a využiť získané poznatky pri návrhu nového simulačného nástroja použiteľného pri vývoji a optimalizácii procesorov. Táto práca prezentuje hlavné požiadavky na nový simulátor a popisuje návrh a implementáciu jeho kľúčových častí s dôrazom na dosiahnutie čo najvyššieho výkonu.

Abstract

The major objective of this work is to analyse possibilities of using simulation within the development of application-specific instruction-set processors, to explore and compare some common simulation techniques and to use the collected information to design a new simulation tool suitable for utilization in the processors development and optimization. This thesis presents the main requirements on the new simulator and describes the design and implementation of its key parts with emphasis on the high performance.

Kľúčové slová

simulácia, ASIP, optimalizácia, interpretácia

Keywords

simulation, ASIP, optimization, interpretation

Citácia

RICHTARIK, Pavel. *Rychlý a částečně překládaný simulátor pro aplikačně specifické procesory*. Brno, 2018. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Marcela Zachariášová, Ph.D.

Rychlý a částečně překládaný simulátor pro aplikačně specifické procesory

Prehlásenie

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

.....
Pavel Richtarik
23. mája 2018

Poďakovanie

Rád by som poďakoval pani Ing. Marcele Zachariášovej za jej vedenie a pomoc pri písaní tejto práce. Tiež by som chcel poďakovať pánom Ing. Zdeňkovi Přikrylovi, Ph.D. a Ing. Albertovi Mikóovi za ich odbornú pomoc pri návrhu a implementácii popisovaného simulátora.

Obsah

Kapitola 1

Úvod

Vstavané systémy tvoria neodmysliteľnú súčasť moderného sveta. Svojou činnosťou sa podieľajú na kontrole a riadení množstva procesov, a okrem očividného použitia v elektronických zariadeniach, ako sú mobilné telefóny, prenosné počítače alebo inteligentné televízie, ich je možné nájsť aj na menej nápadných miestach, napríklad v brzdných systémoch automobilov, jadrových elektrárňach, vesmírnych družiciach, ale aj domácich spotrebičoch.

Požiadavky na výpočtovú techniku vo vstavaných zariadeniach sú odlišné od požiadaviek kladených na bežné počítače. Prioritou nie je čo najvyšší výpočtový výkon, kritická je minimálna spotreba a čo najmenšia plocha. Tieto zariadenia sú často napájané z batérií s obmedzenou kapacitou, preto je z hľadiska predĺženia ich životnosti nízka spotreba kľúčová. Veľkosť zohráva rovnako dôležitú úlohu napríklad pri riadení mikroskopických senzorov alebo pri umiestňovaní celého systému na jeden čip (*SoC – System on a Chip*).

Z toho dôvodu sa uplatňuje prístup špecializácie softvéru aj hardvéru na konkrétnu úlohu. Niekedy je postačujúce zvoliť vhodný existujúci programovateľný mikrokontrolér, avšak v prípadoch, kedy je tlak na efektivitu a miniaturizáciu vysoký, je potrebné pristúpiť k aplikačne špecifickému hardvéru, napríklad v podobe zákazníckych integrovaných obvodov. V dnešnej dobe často používanú alternatívu tvoria aplikačne špecifické procesory, ktoré poskytujú možnosť súbežného vývoja softvéru a hardvéru umožňujúcu lepšie pochopenie vzťahov a optimálnejšie rozdelenie úloh medzi týmito dvoma prostrediami.

Jedným zo základných nástrojov používaných pri návrhu aplikačne špecifických procesorov je simulácia, ktorej cieľom je umožniť overenie a otestovanie modelu bez nutnosti jeho fyzickej výroby. To šetrí obrovské finančné prostriedky a dovoľuje rýchle overenie správania navrhnutého konceptu.

Táto práca sa zaoberá problematikou simulácie aplikačne špecifických procesorov, rôznymi prístupmi a spôsobmi jej realizácie. Hlavnou motiváciou je návrh a implementácia nového čiastočne prekladaného simulátora pre firmu *Codasip Ltd.* integrovateľného do existujúceho vývojového prostredia.

Prvá kapitola tvorí krátky úvod do tematiky aplikačne špecifických procesorov, ich návrhu a požívaných nástrojov. Uvedené je základné delenie procesorov podľa architektúry, rozdiely medzi všeobecnými a aplikačne špecifickými procesormi a následne je popísaný proces návrhu a vývoja aplikačne špecifických procesorov spolu s používanými technológiami.

Druhá kapitola rozoberá simuláciu mikroprocesorov ako jeden zo základných nástrojov využívaných pri ich návrhu. Rozdeľuje používané simulačné modely podľa úrovne detailu a vyčleňuje tri typy simulátorov podľa spôsobu implementácie – interpretovaný, prekladaný a čiastočne prekladaný simulátor.

V tretej kapitole nasleduje popis existujúcich simulátorov používaných v praxi. Z otvorených riešení je to *QEMU* a *Spike*, zástupcom komerčnej sféry je simulátor od spoločnosti *Imperas* a simulátory generované v prostredí *Codasip Studio* od spoločnosti *Codasip*. Uvedené sú orientačné rýchlosti jednotlivých simulátorov pre porovnanie.

Do štvrtej kapitoly bol zaradený stručný popis niektorých konštrukcií jazyka *CodAL* používaného na návrh procesorov v prostredí *Codasip Studio*. Rozobraný je predovšetkým spôsob definície inštrukčnej sady podstatný pre model s presnosťou na inštrukcie.

Kapitola 5 obsahuje spolu s definíciou kľúčových požiadaviek návrh nového simulátora, zhrňujúci nadobudnuté poznatky. Na základe identifikácie hlavných problémov v súčasnej implementácii sú prezentované základné koncepty, ktorých použitie by malo zabezpečiť vyšší výkon nového simulačného nástroja porovnateľný s existujúcimi konkurenčnými produktmi.

Konkrétna realizácia navrhnutých optimalizácií je popísaná v šiestej kapitole. Ide o podrobnejšie priblíženie spôsobu implementácie a ukážku vybraných implementačných problémov a ich riešení.

Predposledná kapitola sa venuje testovaniu nového simulátora, pričom sa zameriava najmä na porovnanie výkonu oproti pôvodnému simulátoru pracujúcemu s presnosťou na inštrukcie. Zaoberá sa tiež analýzou správania sa novej implementácie pri rôznom zaťažení. Sumarizácia dosiahnutých výsledkov je prezentovaná prostredníctvom niekoľkých grafov.

Posledná kapitola obsahuje návrhy na zlepšenie do budúcna.

Kapitola 2

Aplikačne špecifické procesory

Aplikačne špecifické procesory (*ASIP – Application Specific Instruction-set Processor*) sú procesory navrhnuté na riešenie úzko špecifikovanej množiny problémov. Typicky sú nasadzované vo vstavaných systémoch, kde prioritou je minimalizácia plochy a spotreby [1]. Optimalizácia procesora pre konkrétnu aplikáciu spočíva jednak v návrhu vhodnej inštrukčnej sady a jednak v efektívnej implementácii jednotlivých inštrukcií na úrovni hardvéru [15].

Procesory pre všeobecné použitie (*General Purpose Processors*), sú vo vstavaných aplikáciach často nepoužiteľné, a to kvôli zbytočnej komplexnosti a s tým súvisiacej vysokej spotrebe, resp. nevyhovujúcej veľkosti. Naopak zákaznícke integrované obvody (*ASIC – Application Specific Integrated Circuits*) bez modifikovateľného softvéru síce ponúkajú najväčšiu možnosť minimalizácie, avšak ich vývoj je nákladný a použitie príliš limitované [22].

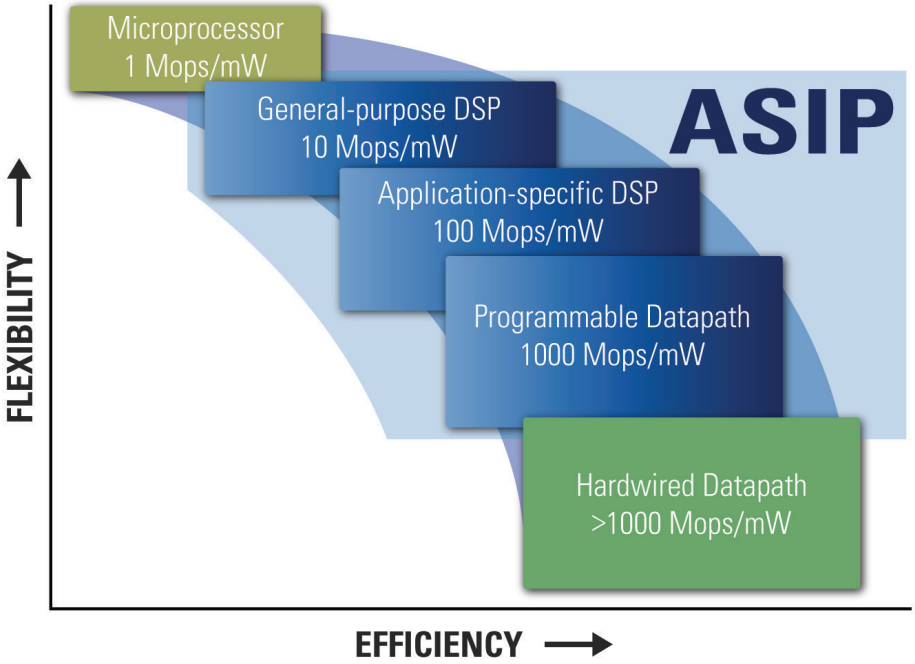
Kompromis tvoria práve aplikačne špecifické procesory (obr. 2.1), ktoré kombinujú výhody oboch prístupov. Obmedzením funkcionality na určitú triedu aplikácií vzniká priestor pre väčšiu optimalizáciu. Pri tom však stále ostáva možnosť modifikovať softvér pre konkrétnu úlohu, čo zvyšuje použiteľnosť procesora, a vďaka čomu klesajú počiatočné náklady na vývoj (*NRE – Non-Reccuring Engineering*) [12] pripadajúce na jeden kus.

Podľa použitej architektúry je možné rozdeliť procesory na:

- *RISC (Reduced Instruction Set Computer)* – inštrukčná sada obsahuje len základné inštrukcie trvajúce typicky jeden takt, zreťazené spracovanie je teda jednoduché a efektívne,
- *CISC (Complex Instruction Set Computer)* – inštrukčná sada obsahuje komplexné inštrukcie trvajúce rôzny počet taktov, plánovanie zreťazeného vykonávania inštrukcií je teda komplikovanejšie,
- *VLIW (Very Long Instruction Word)* – jedno inštrukčné slovo obsahuje viacero inštrukcií, ktoré sú vykonané paralelne v duplikovaných exekučných jednotkách (slotoch).

Príkladom CISC procesorov je architektúra x86 od spoločnosti Intel používaná v osobných počítačoch. Vo vstavaných systémoch sa však uplatňujú skôr procesory RISC s jednoduchou sadou inštrukcií, výsledkom čoho je jednoduchší hardvér. Základná sada môže byť potom doplnená o niekoľko špecializovaných komplexných inštrukcií, ktorých pozitívny efekt je overený simuláciami konkrétnych aplikácií [22].

Procesory VLIW tvoria osobitnú kategóriu svojím špecifickým prístupom k paralelizácii. Tá je realizovaná na úrovni inštrukcií (*Instruction Level Parallelism*), pričom riešenie



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

konfliktov pri plánovaní súbežného vykonávania je realizované kompilátorom už v dobe prekladu – na strane procesora teda táto réžia odpadá. Tieto procesory sa využívajú napríklad pri spracovaní obrazu alebo počítačovom videní, kde má paralelizmus veľký význam [7].

Nevýhodou pri tejto architektúre môže byť veľkosť kódu. Prekladač totiž doplňuje prázdne inštrukcie *NOP* všade tam, kde nie je možné v danom okamihu využiť paralelnú jednotku, typicky z dôvodu časových závislostí medzi inštrukciami. Na zmenšenie veľkosti preloženého programu sa využívajú rôzne techniky, napríklad zavedenie špeciálneho kódovania inštrukcií *NOP* alebo použitie rôzne dlhých inštrukčných kódov za účelom skrátenia dlhého inštrukčného slova (*variable length encoding*). Inou možnosťou je uloženie komprimovanej verzie programu do externej pamäte a jeho dynamická de-komprimácia počas behu s využitím softvéru alebo špecializovaného hardvéru [8].

Súčasťou návrhu procesora je okrem výberu architektúry aj špecifikácia inštrukčnej sady (súvisiaca s vývojom softvéru) a popis zreťazenej linky (*pipeline*), pamäťového systému a podporovaných periférií. S vytvoreným prototypom je možné ďalej experimentovať za účelom optimalizácie – na tieto účely slúžia rôzne simulačné nástroje. Simulácia je tiež základným predpokladom funkčnej verifikácie, teda overenia správnosti modelu.

Nástroje na návrh procesorov

Návrh mikroprocesorov je komplexný proces, pri ktorom je snaha čo najväčšiu časť práce automatizovať. Najmä v počiatočnej fáze, pri skúmaní a porovnávaní rôznych prístupov a architektúr (*design space exploration*) je možnosť rýchleho vytvárania prototypov a ich ohodnocovania pre porovnanie (*benchmarking*) kľúčová. Zároveň je potrebné uvážiť odlišnosti v požiadavkách pri vývoji hardvéru a softvéru – zatiaľčo vývoj hardvéru vyžaduje detailné simulácie s presnosťou na úrovni cyklov a tiež prostriedky na verifikáciu (prepojenie na verifikačné prostredia), softvér je závislý výlučne na inštrukčnej sade a požaduje

funkčný prekladač a rýchly simulátor preloženého kódu, presnosť simulácie na cykly nie je potrebná [9].

V dnešnej dobe je pri vývoji procesorov automatické generovanie nástrojov samozrejmosťou – používajú sa na to integrované vývojové prostredia (*IDE – Integrated Development Environment*), napríklad *ASIP Designer* od spoločnosti Synopsys alebo *Codasip Studio* od firmy Codasip Ltd.

V minulosti boli procesory (tak ako aj iné integrované obvody) navrhované v nízkoúrovňových HDL (*Hardware Description Language*) jazykoch, napr. *VHDL* alebo *Verilog*. Postupne sa však prešlo k vyššej abstrakcii a dnes sa na modelovanie procesorov používajú ADL (*Architecture Description Language*) jazyky, napr. *LISA*, *CodAL* alebo *nML*, ktoré pracujú s komplexnejšími stavebnými prvkami (pamäte, rozhrania, zbernice) a dokážu jednoducho vyjadriť kódovanie a sémantiku jednotlivých inštrukcií. Väčšia štruktúrovanosť modelov popísaných v ADL jazykoch ponúka priestor pre komplexnejšie automatické spracovanie, optimalizáciu a generovanie nástrojovej sady (*toolchain*) pre daný mikroprocesor. Typickými generovanými nástrojmi sú simulátor, prekladač (asembler alebo C/C++), debugger a profiler. Z ADL popisu je tiež možné generovať RTL (*Register-Transfer Level*) model v jazyku HDL, ktorý sa použije pri syntéze výsledného fyzického čipu.

Automatické generovanie nástrojov predstavuje obrovský pokrok v návrhu procesorov. Umožňuje jednoducho a rýchlo preskúmať rôzne konfigurácie modelu, resp. porovnávať rôzne modely bez nutnosti rozsiahlych úprav na najnižšej úrovni. Zároveň uľahčuje proces funkčnej verifikácie – s využitím profilera je možné napríklad sledovať pokrytie testovaného kódu a taktiež je možné generovať kosimulátory, teda rozhrania pre zapojenie vygenerovaného simulátora ako referenčného modelu do štandardných verifikačných prostredí, napr. prostredí UVM (*Universal Verification Methodology*).

Cieľom tejto kapitoly bolo poskytnúť prehľad o rôznych architektúrach aplikačne špecifických procesorov a procese ich návrhu spolu s používanými nástrojmi. Táto práca sa ďalej zaoberá simuláciou, jej rôznymi formami a úrovňami. Získané poznatky sú prenesené do návrhu nového simulátora prezentovaného v záverečnej časti.

Kapitola 3

Simulácia mikroprocesorov

Simulácia je proces experimentovania s modelom za účelom získania nových znalostí o reálnom systéme [17]. Simulátor je v prípade počítačovej simulácie program realizujúci predpísané experimenty.

Simulácia je jedným z najdôležitejších prostriedkov pri vývoji procesora – umožňuje odhaliť chybné alebo neefektívne časti návrhu ešte pred vytvorením reálneho čipu, čím šetrí obrovské finančné prostriedky. Zároveň je podporným nástrojom pre záverečnú časť vývoja – verifikáciu.

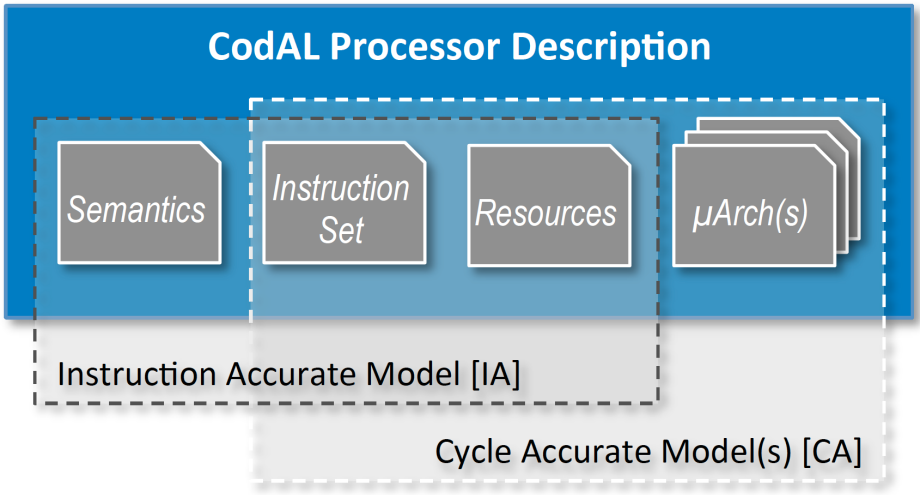
Dôležitými vlastnosťami simulátora sú rýchlosť a presnosť, ktoré sú od seba typicky závislé. Vyššiu rýchlosť je možné dosiahnuť vyššou mierou abstrakcie, zatiaľčo zvýšenie úrovne detailu predstavuje naopak zvýšené výpočtové nároky, a doba trvania simulácie sa teda predlžuje. Preto sa v praxi používa viacero typov simulátorov podľa účelu [9]. Niektoré z nich sú popísané v nasledujúcom texte.

Pri detailnej simulácii orientovanej na správanie hardvéru napríklad pri verifikácii sa používa simulátor s presnosťou na cykly – *Cycle Accurate (CA) Simulator*. Stav modelu procesora (obsahy registrov a hodnoty signálov) je sledovaný v každom hodinovom takte, čo umožňuje ladiť a verifikovať napríklad komunikačné protokoly na zberniciach, funkčnosť zreťazeného spracovania inštrukcií alebo výskyt rôznych hazardov. Tento typ simulácie je výpočtovo náročný a výkon simulátora je zákonite rádovo nižší ako výkon modelovaného procesora.

Podmienkou vývoja softvéru pre nový procesor je funkčný prekladač, detaily hardvérovej implementácie nie sú podstatné. Pri simulácii je kľúčové to, či preložený program pracuje správne (overuje sa sémantika inštrukcií). Načítanie inštrukcie z pamäte, dekódovanie a vykonanie prebieha v jednom kroku, zreťazené spracovanie a oneskorenie pri prístupe do pamäti je zanedbané. Takýto simulátor pracuje s presnosťou na inštrukciu – *Instruction Accurate (IA) Simulator*. Prioritou je vysoká rýchlosť simulácie, ktorá sa môže pri efektívnej implementácii v niektorých prípadoch blížiť k natívnemu výkonu procesora. Vysoká rýchlosť je kľúčová pri simulácií komplexných programov alebo celých operačných systémov na cieľovej platforme.

V prípadoch, kedy je rýchlosť simulácie absolútne kritická a vnútorný stav procesora nie je podstatný, je možné použiť funkčnú emuláciu (*Functional Emulation*) [4]. V tomto prípade sú počas behu simulácie aplikované rôzne radikálne optimalizácie, pri ktorých nemusí byť možné sledovať program inštrukciu po inštrukcii – presnosť je teda menšia ako pri striktnom IA prístupe.

V literatúre sa často zamieňa pojem emulácia a simulácia. Zatiaľčo cieľom simulácie je skúmanie správania a vnútorného stavu modelu v čase, pri emulácii je vnútorný stav



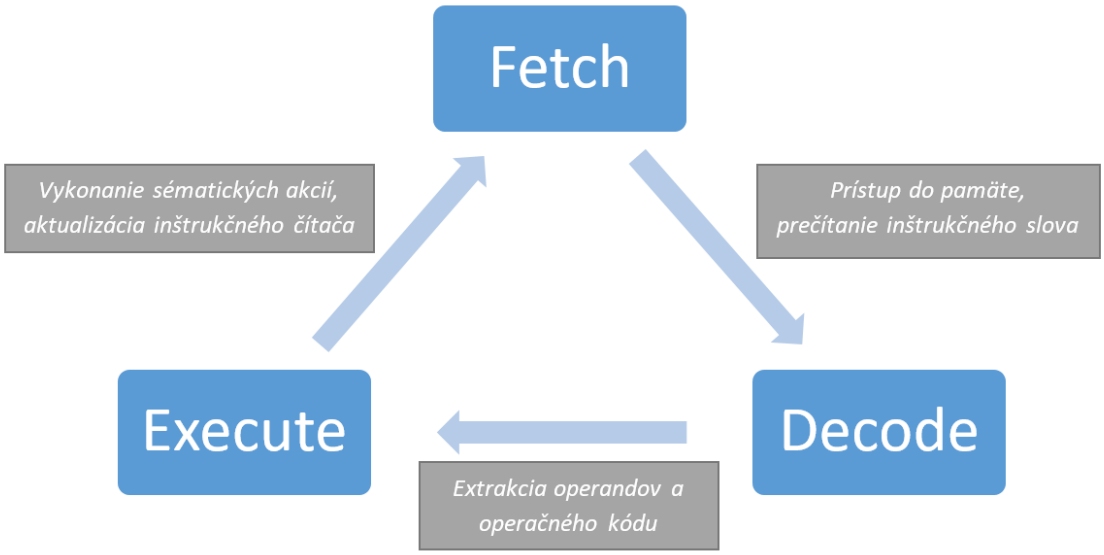
Obr. 3.1: Vzťah IA a CA modelu [5]

zanedbávaný a dôležité je len vonkajšie správanie objektu. Často sa využíva kombinácia oboch prístupov, napríklad pri simulácii procesora zapojeného do širšieho systému. Kým procesorové jadro je modelované detailne, periférie a pamäťový subsystém môžu byť emulované, zaisťujúc požadovanú funkcionalitu (komunikačné rozhranie) ale bez možnosti bližšie skúmať ich vlastnosti [6]. Osobitným pojmom je hardvérová emulácia, kedy časť hardvéru v systéme je nahrádzaná iným hardvérovým prvkom, typicky programovateľným poľom FPGA za účelom overenia správnosti jeho modelu pred vyrobením výsledného čipu. Nejde teda o počítačovú simuláciu ale o reálne zapojenie fyzických obvodov.

Rozdelenie simulátorov na IA a CA má aj ďalšiu výhodu – po špecifikácii inštrukčnej sady je možné paralelne pracovať na vývoji softvéru a hardvéru. Pre vytvorenie IA simulátora a prekladača nie je totiž konkrétna implementácia inštrukcií v hardvéri podstatná. Hardvérový návrh sa môže vyvíjať nezávisle od softvéru, čo skracuje celkovú dobu vývoja produktu. Dodatočne je potom možné s použitím profilovacích nástrojov analyzovať využitie jednotlivých inštrukcií a modifikovať inštrukčnú sadu za účelom zlepšenia celkového výkonu. Môže byť totiž vhodné niektoré málo využité inštrukcie (a s nimi súvisiaci hardvér) eliminovať a ponechať len softvérovú implementáciu. Naopak, niektoré často vykonávané úseky kódu môže byť výhodné preniesť do obvodovej realizácie v hardvéri. Vďaka automatickému generovaniu nástrojov a IA resp. CA simulácii spolu s profilerom je možné prakticky okamžite vyhodnotiť vplyv zmien aplikovaných v modeli na efektivitu celého procesora.

Na obrázku 3.1 je znázornený vzťah a prepojenie IA a CA modelu. Spoločné rozhranie tvorí inštrukčná sada a architekturálne zdroje, ku ktorým IA model pridáva sémantiku a CA model dopĺňa mikroarchitekturálnu implementáciu.

Simulátory procesorov sa líšia aj spôsobom implementácie. Používanými technikami sú interpretácia a kompilácia. Zatiaľčo čisto interpretované simulátory ponúkajú najväčšiu mieru flexibility, ich nevýhodou je často nepostačujúci výkon. Plne prekladané simulátory sú naproti tomu niekoľkonásobne rýchlejšie, avšak ich použitie je obmedzené na daný model procesora a simulovanú aplikáciu. V minulosti boli takéto simulátory programované manuálne, avšak ich implementácia bola náročná a následná validácia problematická. Kompromisom medzi týmito dvoma krajnými riešeniami sú rôzne varianty čiastočne prekladaných simulátorov [10].



Obr. 3.2: Schéma interpretovaného simulátora

3.1 Interpretovaný simulátor

Najjednoduchšou implementáciou simulátora inštrukcií je priama interpretácia, ktorá v podstate verne kopíruje správanie hardvéru. To znamená, že v každom kroku je z pamäte z adresy určenej programovým čítačom načítané inštrukčné slovo, prebehne jeho dekódovanie a vykoná sa príslušná sémantická akcia. Tento prístup je z hľadiska rýchlosti neefektívny, rovnaká inštrukcia musí byť totiž pri každom výskyte znovu dekódovaná a simulačný kód implementujúci jednotlivé inštrukcie je rozptýlený – výkon simulátora je preto limitovaný neustálymi skokmi [18]. Obrázok 3.2 tento postup ilustruje – každá inštrukcia vyžaduje zopakovanie celého cyklu.

Okrem veľkej fragmentácie programu zohráva úlohu aj veľkosť réžie súvisiaca s dekódovaním, ktorá sa odvíja od komplexnosti definície inštrukčnej sady. Zatiaľčo v jednoduchých prípadoch je operačný kód a operandy možné identifikovať obyčajným porovnaním vyhradených bitov, pri inštrukciách s premenlivou dĺžkou alebo zložitejšou štruktúrou môže dekódovanie vyžadovať náročnejší výpočet, a tým, kvôli jeho opakovanému vykonávaniu v každom takte, značne degradovať výkon simulátora.

Výhodou je jednoduchá implementácia nezávislá od interpretovanej aplikácie a implicitná podpora modifikácie programu za behu. Koncept interpretácie je priamo aplikovateľný na akýkoľvek model procesora. Generovanie simulátora je možné vcelku jednoducho automatizovať – hlavná simulačná slučka je pevne daná, špecifické pre daný procesor je len dekódovanie inštrukcií a ich sémantika.

Na tomto princípe je založených množstvo simulačných nástrojov, vrátane IA simulátora generovaného v prostredí *Codasip Studio*.

3.2 Prekladaný simulátor

Prekladaný simulátor je oproti interpretovanému rýchlejší, a to vďaka informáciám o cieľovej aplikácii a eliminácii zbytočnej réžie súvisiacej s interpretáciou každej inštrukcie osobitne.

Vo svojej najprimitívnejšej podobe je prekladaný simulátor závislý na simulovanej aplikácii. Pred vytvorením simulátora je potrebná analýza programu a jeho statický preklad do kódu hostiteľského stroja. Prakticky ide o vytvorenie fragmentov kódu pre každú inštrukciu a následnú substitúciu simulovaných inštrukcií príslušnými fragmentmi.

Analýzu simulovanej aplikácie a proces vytvorenia prekladaného simulátora je možné rozdeliť do 3 základných krokov [10]:

1. dekódovanie inštrukcií (*instruction decoding*),
2. zostavenie sekvencií operácií pre každú inštrukciu (*operation sequencing*),
3. plánovanie vykonávania jednotlivých operácií (*operation instantiation*).

Prvý krok zahŕňa identifikáciu operačného kódu a operandov pre každú inštrukciu v programe. Toto predspracovanie už počas generovania simulátora predstavuje elimináciu réžie spojenej s dekódovaním počas behu (v porovnaní s interpetačným simulátorom).

V závislosti od použitého ADL jazyka, môže byť sémantika inštrukcií rozptýlená, preto je potrebné zhromaždiť všetky elementárne operácie súvisiace s vykonaním konkrétnej inštrukcie. Typicky je potom každá inštrukcia reprezentovaná sekvenciou príkazov a volaní čiastkových funkcií.

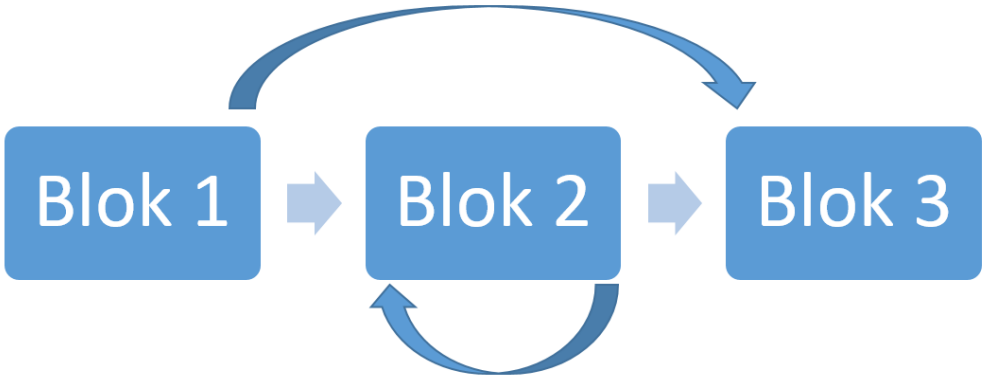
Plánovanie vykonávania jednotlivých operácií je v prípade IA modelu triviálnou úlohou. Ide o náhradu každej inštrukcie programu príslušnou sekvenciou operácií, inštrukcia je totiž nedeliteľná a jej vykonanie predstavuje atomickú udalosť. Vďaka znalosti kódu simulovanej aplikácie a statickej substitúcii inštrukcií dôjde k úplnému rozvinutiu simulačnej slučky.

V prípade modelu s presnosťou na cykly je však plánovanie komplikované, a to kvôli modelovaniu zreťazenej linky, kde dochádza k prekrývaniu viacerých inštrukcií, a navyše sa tu vyskytujú ťažko predvídateľné pozdržania (*stall*) alebo vyprázdnenia (*clear*) linky, ktoré časový plán narúšajú. Statické plánovanie je často nemožné, resp. vyžaduje sofistikované prístupy, kedy je nutné vygenerovať viacero variant časových plánov napríklad v miestach skokových inštrukcií. K voľbe správnej varianty dochádza dynamicky počas behu simulácie [10].

Simulovaný kód je možné rozdeliť do takzvaných základných blokov (*basic blocks*). Ide o ucelené úseky kódu s jedným začiatkom a koncom, čo znamená, že neexistujú skoky do alebo z tela takéhoto bloku. Na získanie informácií o základných blokoch nestačí statická analýza preloženého programu, a to kvôli možným nepriamym skokom. Je potrebný zdrojový kód programu vo vyššom štrukturovanom programovacom jazyku. Simulátor, ktorý je prekladaný po blokoch (*translated simulator*), môže vďaka týmto informáciám pracovať oproti obyčajnej statickej kompilácii podstatne efektívnejšie – v rámci základných blokov sú akceptovateľné agresívnejšie optimalizácie, ide totiž o atomické časti kódu, ktoré môžu byť nahradené jedinou inštrukciou [19]. Z pohľadu simulácie je potom základnou nedeliteľnou jednotkou celý blok, tak ako je to znázornené na obrázku 3.3 – skoky sú možné len na hraniciach blokov (prostredný blok tu môže predstavovať napríklad telo cyklu).

3.3 Čiastočne prekladaný simulátor

Okrem nutnosti vygenerovať nový simulátor pre každú aplikáciu je ďalšou nevýhodou statickej kompilácie neschopnosť simulovať samo-upravujúci kód (*self-modifying code*). Riešením tohto problému je použitie dynamickej kompilácie JIT (*just-in-time compilation*).



Obr. 3.3: Schéma simulácie po blokoch

JIT simulátor funguje spočiatku na princípe interpretácie, a počas toho sa snaží identifikovať často vykonávané úseky kódu (*hot-spots*). Následne sú tieto fragmenty za behu preložené do podoby dynamickej knižnice, ktorá je pripojená k bežiacemu simulátoru a pri každom ďalšom pokuse o vykonanie kódu, ktorý bol identifikovaný ako *hot-spot*, je potom použitý natívny kód z knižnice, čo simuláciu urýchľuje.

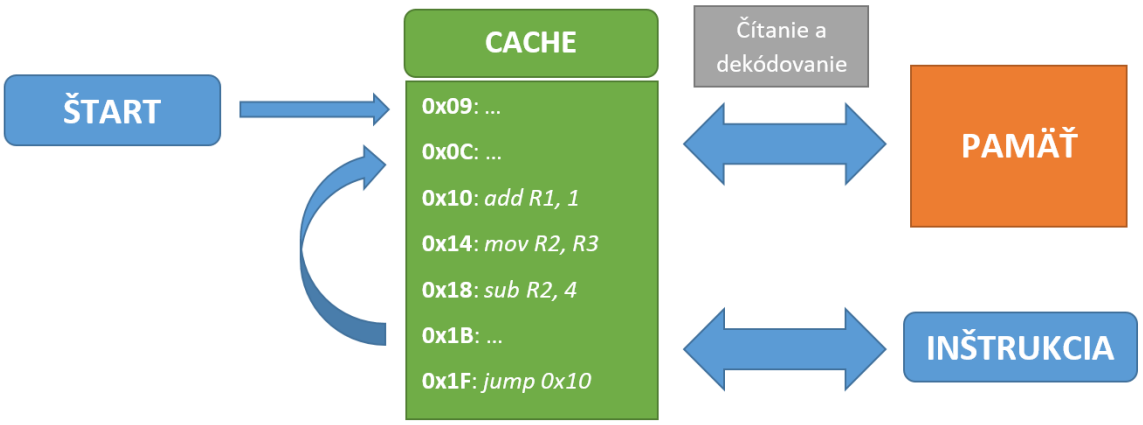
Napriek réžii spojenej s určovaním často vykonávaných úsekov kódu môže byť JIT simulátor stále niekoľkokrát rýchlejší ako čistý interpret. Výhodou oproti staticky prekladanému simulátoru je nezávislosť na simulovanej aplikácii.

Najpokročilejšiu a najefektívnejšiu formu simulácie tvorí kombinácia prekladu po blokoch (*translated simulator*) s využitím JIT kompilácie pre podporu samo-upravujúceho kódu. Pri použití prekladu po blokoch je možné potrebné informácie o začiatkoch a koncoch základných blokov uložiť priamo do preloženej aplikácie v podobe ladiacich informácií. Počas simulácie potom hlavná simulačná slučka neskáče po jednotlivých inštrukciách ale po celých blokoch, ide teda o jej čiastočné rozvinutie. V rámci predspracovania aplikácie je dokonca možné vložiť skokové inštrukcie priamo na konce príslušných blokov, za účelom ich priameho prepojenia. Takýto simulátor je zbavený závislosti na konkrétnej aplikácii, predpokladá však prítomnosť popisu základných blokov v podobe ladiacich informácií, na čo musí byť vybavený prekladač.

Ako príklad pre porovnanie s interpretovaným simulátorom môže poslúžiť obrázok 3.3 – interpretácia by vyžadovala postupné dekódovanie a vykonanie každej inštrukcie, čo znamená prinajmenšom jeden skok pre návrat na začiatok interpretačnej slučky a dva skoky do kódu príslušnej sémantickej akcie a späť. Naproti tomu simulátor prekladaný po blokoch vyžaduje jediný skok na konci bloku.

Podpora samo-modifikujúceho kódu môže potenciálne narúšať štruktúru základných blokov definovanú pri preklade aplikácie. Problémom sú aj nepriame skoky s neznámou cieľovou adresou. Pri kompilácii sa totiž kód v rámci bloku môže kvôli optimalizáciám radikálne odlišovať od pôvodného, a skok je teda možný len na jeho začiatok. Počas behu simulácie je preto potrebná dynamická správa blokov so súvisiacimi podpornými štruktúrami, tak aby bolo umožnené vytváranie, spájanie, rozdeľovanie a rušenie blokov v dôsledku modifikácie kódu podľa potreby.

Simulátoru prekladanému po blokoch sa dá priblížiť približnou identifikáciou základných blokov dynamicky, počas behu programu. Koncept takejto aproximácie ucelených sekvencií v kóde a akcelerácia ich simulácie je predmetom návrhu, ktorým sa zaoberá táto práca.



Obr. 3.4: Simulácia s využitím *cache* tabuľky posledných vykonaných inštrukcií

Hlavnou myšlienkou je zachovať výhody interpretácie, konkrétne flexibilitu a implicitnú podporu modifikácie kódu za behu, a zároveň dosiahnuť čo najvyšší výkon. Nástrojom na dosiahnutie tohto cieľa je použitie krátkej *cache* pamäte naposledy vykonaných inštrukcií s vhodnou štruktúrou. Tým dôjde k čiastočnému rozvíjaniu hlavnej simulačnej slučky v opakujúcich sa fragmentoch, teda typicky cykloch.

Ilustráciou tohto prístupu je obrázok 3.4. Ústredným prvkom je *cache* tabuľka dekódovaných inštrukcií, ktoré sú postupne vykonávané. V prípade výpadku je príslušná inštrukcia načítaná z pamäti a uložená v predspracovanej dekódovanej podobe. Detaily fungovania sú bližšie popísané v kapitole návrhu.

Za zmienku stojí porovnanie miery abstrakcie pri čistej interpretácii a preklade po blokoch. Atomickou udalosťou interpretačnej simulácie je vykonanie jednej inštrukcie, môžeme teda hovoriť skutočne o modeli s presnosťou na inštrukcie. Avšak v momente, kedy sú povolené blokové optimalizácie, informácie o jednotlivých inštrukciách sa strácajú a atomickú udalosť predstavuje celý blok. Nie je možné sledovať stav modelu po každej inštrukcii – môže sa totiž stať, že v rámci optimalizácie dôjde k jej úplnému vypusteniu, resp. že efekt skupiny simulovaných inštrukcií je dosiahnutý jedinou inštrukciou hostiteľského stroja. Idete teda o väčšiu mieru abstrakcie so stratou detailov v prospech vyššieho výkonu.

Kapitola 4

Existujúce simulátory

Táto kapitola popisuje niekoľko vybraných existujúcich nástrojov používaných na simuláciu procesorov. Príkladmi otvorených programov sú *QEMU* [21] a *Spike* [28], z komerčnej sféry je to *ISS – The Imperas Instruction Set Simulator* [13] od spoločnosti *Imperas* a *Codasip Studio* – sada nástrojov od spoločnosti *Codasip* obsahujúca okrem iného aj simulátory.

4.1 QEMU

QEMU je generický *open source* emulátor a virtualizačné prostredie [21]. Dokáže pracovať v dvoch režimoch s rôznou hĺbkou detailu:

- *emulácia celého systému (full-system emulation)* – umožňuje emulovať celý systém (napríklad PC) s jedným alebo viacerými procesormi a rôznymi perifériami,
- *emulácia v užívateľskom režime (user-mode emulation)* – umožňuje spúšťať užívateľské programy preložené pre iný procesor.

Emulácia v užívateľskom móde je typicky jednoduchšia – systémové volania sú prekladané na volania hostiteľského systému a pamäťové operácie využívajú natívnu správu pamäti. Emulácia celého systému zahŕňa softvérovú implementáciu periférnych zariadení a správy pamäte, je teda výpočtovo náročnejšia avšak poskytuje väčšiu mieru kontroly a presnosti simulovaného systému [4].

QEMU emulátor nevyžaduje ovládač v jadre operačného systému (*host kernel driver*). Kód emulovaného systému, resp. programu je počas behu dynamicky prekladaný do natívneho, s podporou modifikácie kódu za behu – využíva teda JIT (*just in time*) kompiláciu. Simulačná réžia zahŕňa preklad systémových volaní, prácu s vláknami a mapovanie obsluhy signálov [20].

Okrem počítačových operačných systémov dokáže QEMU emulovať aj rôzne procesory a inštrukčné sady, napríklad SPARC (*Scalable Processor Architecture*), MIPS (*Microprocessor without Interlocked Pipeline Stages*), ARM [20] alebo RISC-V [3].

Emulované inštrukcie sú dynamicky prekladané do inštrukčnej sady hostiteľského systému po blokoch (*basic blocks*) – pri prvom zaznamenaní neznámeho kódu je vygenerovaný jeho preklad. Jednotlivé bloky sú potom prepojené doplnením skokových inštrukcií. Vytváranie preloženého kódu má na starosti komponent TCG (*Tiny Code Generator*) špecifický pre hostiteľský systém. Pri kóde, ktorý sa môže za behu meniť (*self-modifying code*) musí byť korektne ošetrené zneplatnenie inštrukčnej vyrovnávacej pamäte [20].

Princípom fungovania sa QEMU vzďaľuje simulácii – vnútorné fungovanie procesora je úplne zanedbané, implementácia sa snaží o maximalizáciu výkonu so zachovaním sémantiky inštrukcií, ide teda skôr o emuláciu. Výhodou je vysoká rýchlosť blížiaca sa natívnej a prenosnosť na rôzne platformy (*Linux, BSD, Mac OS X, Windows*) a architektúry.

Napriek snahe pokryť čo najväčšie množstvo simulovaných architektúr sa však QEMU neosvedčilo pri modelovaní zložitejších procesorov, ktorých princíp fungovania sa významnejšie líši od hostiteľského stroja (typicky Intel x86). Pokusy o nasadenie v rámci prostredia Codasip Studio neboli úspešné – podpora procesorov VLIW, DSP (*Digital Signal Processing*) inštrukcií alebo napríklad hardvérových slučiek sa ukázala ako príliš komplexná na implementáciu v podobe automatických generátorov kódu pre platformu QEMU. Použitie tohto výkonného emulátora teda ostáva obmedzené na klasické superskalárne architektúry. Simulácia zložitejších netradičných procesorov vyžaduje individuálnu implementáciu a riešenie rôznych skrytých obmedzení.

4.2 Spike

V roku 2010 bola na univerzite UCB (*University of California at Berkeley*) pre výukové účely navrhnutá nová inštrukčná sada (*ISA – Instruction Set Architecture*) RISC-V. Hlavnými cieľmi bola otvorenosť a podpora čo najväčšej palety architektúr. RISC-V je teda kompletná ale počtom inštrukcií malá inštrukčná sada podporujúca rôzne štandardné aj neštandardné rozšírenia [29].

V posledných rokoch sa z RISC-V ISA stáva pod záštitou *RISC-V Foundation* založenej v roku 2015 otvorený a priebežne dopĺňaný štandard [23]. Jej otvorenosť podporuje konkurenciu medzi výrobcami procesorov v efektívnosti implementácie, pričom jednotná špecifikácia zaručuje kompatibilitu softvéru, a tým poskytuje odberateľom väčšiu voľnosť pri výbere alebo zmene architektúry, resp. dodávateľa [24].

Referenčným simulátorom inštrukčnej sady RISC-V je *Spike* [28]. Nejedná sa o univerzálny simulačný nástroj, jeho funkčnosť je zviazaná s inštrukčnou sadou RISC-V. Umožňuje pozastavenie, vykonávanie po krokoch (inštrukciách alebo riadkoch zdrojového kódu) a zobrazovanie vnútorného stavu (obsahy registrov). Podporované sú aj prerušenia a spracovanie výnimiek. Ide o simuláciu s presnosťou na inštrukcie (IA) a výkon je dostatočne vysoký na simulovanie celého operačného systému.

Súčasťou modelu sú aj vyrovnávacie pamäte *cache* a to inštrukčná, dátová a spoločná L2 cache. Za účelom maximalizácie výkonu je pred vyrovnávacou pamäťou predradená ešte rýchla vyhľadávacia tabuľka TLB (*Translation Lookaside Buffer*), ktorá celkový počet prístupov do cache zmenšuje. To zanáša do modelu nepresnosti, avšak pre základnú analýzu výpadkov sú aj približné informácie postačujúce [11].

Z pohľadu implementácie je zaujímavé jadro simulácie – načítavanie inštrukcií. Použitá je malá tabuľka posledných vykonaných inštrukcií podľa adresy v kombinácii s modifikovaným Duffovým strojom (*Duff's Device*) [26]. Ten funguje na princípe prepadávania sa na nasledujúce návestie *case* v príkaze *switch*. V prípade sekvenčného vykonávania inštrukcií teda odpadá volanie funkcie *step* pre každú inštrukciu – program na začiatku skočí na správnu adresu, a potom, kým je adresa v nasledujúcej položke tabuľky platná, resp. súhlasí so skutočnou hodnotou registra *PC* (*program counter*), pokračuje ďalším návestím. Podľa dokumentácie má táto optimalizácia, teda eliminácia opakovaného volania funkcie, za následok asi dvojnásobné zrýchlenie [28]. Pseudokód je uvedený na obrázku 4.1.

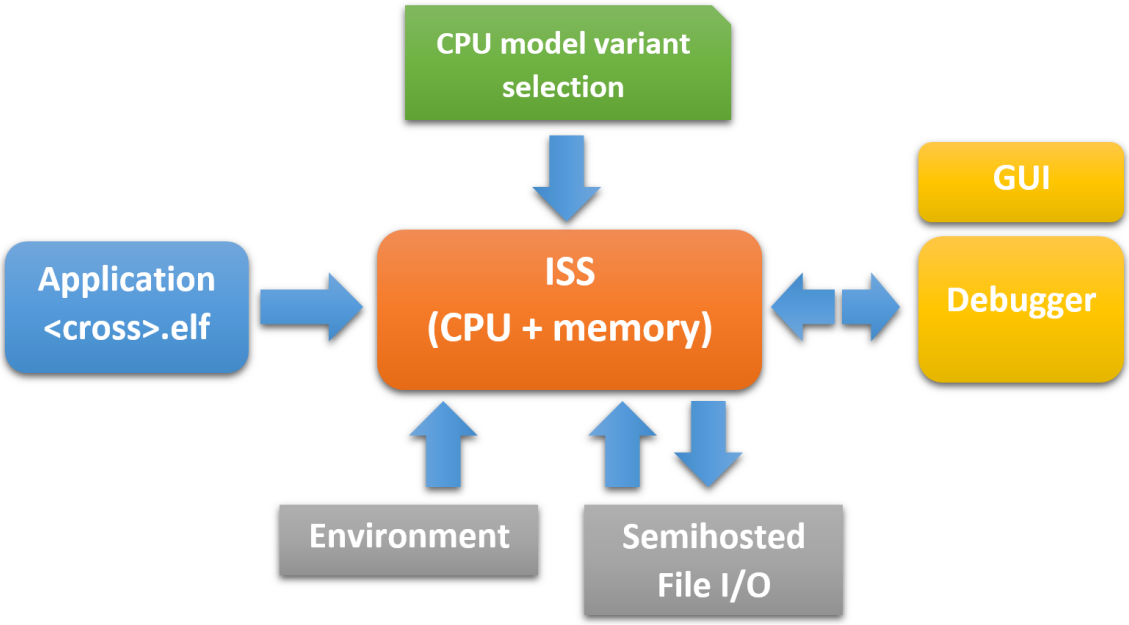
```
switch(pc mod CACHE_SIZE)
{
        ...
        case 8:
                // get the cache entry
                instr = cache.get(pc);
                // execute and retrieve the next PC
                pc = instr->execute();
                // move to the next cache entry
                instr++;
                // wrong cache entry (e.g. after branch)
                if(instr->tag != pc) goto miss;
        ...
}

miss:
        // load the instruction word, decode and insert into the cache
        refill_cache(pc, instr);
```

Obr. 4.1: Pseudokód inštrukčnej cache simulátora *Spike*

Výkon simulátora Spike sa pohybuje v desiatkach miliónov inštrukcií za sekundu. Pre porovnanie, QEMU emulátor pre RISC-V dosahuje stovky miliónov simulovaných inštrukcií za sekundu [4].

4.3 Imperas - ISS

Príkladom komerčného IA simulátora je ISS (*Instruction Set Simulator*) od spoločnosti *Imperas*. Schéma simulátora je znázornená na obrázku 4.2 – zahŕňa simuláciu procesora a pamäti, I/O (vstupno-výstupných) periférií a podporuje zapojenie viacerých procesorov a vstupov z prostredia. Simulované aplikácie majú štandardný formát ELF [27].



Obr. 4.2: Schéma simulátora ISS [13]

Simulátor je používaný softvérovými inžiniermi, ktorí vyvíjajú aplikácie pre nové architektúry a nepotrebujú hardvérové komponenty. Grafické rozhranie tvorí nadstavbu štandardného ladiaceho rozhrania GDB (*The GNU Project Debugger*).

Okrem vývoja softvéru je simulátor použiteľný aj pri automatickom testovaní, a to vďaka možnosti napojenia na skriptovacie prostredia.

Výhodou simulátora od firmy *Imperas* je dostupnosť veľkého množstva pripravených simulačných modelov (vyše 200 rôznych variant CPU) prostredníctvom knižnice *OVP (Open Virtual Platforms) Fast Processor Model Library* [14]. Tieto hotové modely je možné využiť v simuláciách vlastných multiprocesorových systémov.

Keďže ide o komerčný produkt, zdrojové kódy alebo popis architektúry simulátora nie sú verejné. Niektoré technologické informácie sú ale dostupné na internetových stránkach spoločnosti, konkrétne, že ISS využíva JIT kompiláciu a vo veľkej miere pracuje paralelne, čo je dôležité pri multiprocesorových simuláciách, vďaka čomu je údajne až 6-15x rýchlejší ako iné komerčné nástroje [13].

Podľa údajov na stránkach OVP sa ISS v závislosti od simulačného modelu a použitej aplikácie rýchlosťou pohybuje na úrovni 1 až 7 miliárd simulovaných inštrukcií za sekundu. Pre modely RISC-V je uvedený výkon 1 až 4 miliardy inštrukcií za sekundu.

4.4 Codasip Studio

Codasip Studio je integrované vývojové prostredie (*IDE*) určené na návrh a optimalizáciu aplikačne špecifických procesorov. Na základe modelu popísaného v jazyku *CodAL* sú pre navrhnutý procesor automaticky vygenerované nástroje, ktoré okrem iného zahŕňajú CA simulátor s presnosťou na takty a IA simulátor s presnosťou na inštrukcie.

CA simulácia sa snaží o zachytenie komplexných detailov mikro-architektúry. Rozlišuje jednotlivé stupne zreťazenej linky a umožňuje sledovať postupné prečítanie inštrukčného

slova, jeho dekódovanie a následné vykonanie inštrukcie v samostatných taktoch. Navyše počíta s oneskoreniami spôsobenými latenciou pamäte, alebo skokovými inštrukciami. Z dôvodu vysokej detailnosti je CA simulátor oveľa pomalší ako simulátory s presnosťou na inštrukcie – dosahuje len niekoľko stoviek tisíc vykonaných inštrukcií za sekundu.

Naproti tomu, IA simulátor pracuje s vyššou mierou abstrakcie – vykonanie každej inštrukcie trvá jeden takt (jeden simulačný krok). Jeho súčasná implementácia využíva čistú interpretáciu, vďaka čomu je generovanie simulátora pre rôzne modely relatívne jednoduché. Jadro simulácie tvorí jednoduchá slučka, ktorá v každom kroku prečíta jedno inštrukčné slovo z pamäte, dekóduje ho spôsobom predpísaným v modeli, a následne danú inštrukciu interpretuje. Proces interpretácie kopíruje štruktúru inštrukčnej sady, a keďže jazyk *CodAL* umožňuje skladať inštrukcie hierarchicky z menších blokov, postupne je vykonaný simulačný kód pre každý samostatný blok danej inštrukcie.

IA simulátor generovaný v rámci *Codasip Studia* dokáže interpretovať malé desiatky miliónov inštrukcií za sekundu, je teda približne desaťkrát rýchlejší ako simulátor s presnosťou na takty. Jeho architektúra pracujúca na princípe čistej interpretácie mu však nedovolí vyrovnať sa konkurenčným simulátorom využívajúcim rôzne optimalizačné techniky. Napríklad, oproti simulátoru *Spike*, ktorý taktiež využíva interpretačnú metódu, je *Codasip* IA simulátor až 10-15 násobne pomalší.

Kapitola 5

Jazyk CodAL

CodAL (Codasip Architecture Language) je popisný jazyk používaný pri súbežnom návrhu hardvéru a softvéru pre jedno alebo viac procesorové systémy. Jeho syntax je podobná jazyku C, je objektovo orientovaný a deklaratívny (s výnimkou imperatívnych sekcií definujúcich sémantiku). Objekty popisované v tomto jazyku obsahujú informácie potrebné pre rôzne generované nástroje (prekladač, simulátor, debugger...).

Model definovaný v jazyku CodAL obsahuje 4 časti:

- architekturálne zdroje – napr. registre a programový čítač (*program counter*),
- popis inštrukčnej sady – inštrukcie a operandy s textovou a binárnou podobou,
- sémantika – správanie jednotlivých inštrukcií a výnimiek,
- implementácia – skryté zdroje a detailné správanie definujúce mikro-architektúru.

Podľa úrovne detailu je potom možné rozlíšiť IA (*Instruction Accurate*) model s presnosťou na inštrukcie, z ktorého je generovaný prekladač a IA simulátor, a CA (*Cycle Accurate*) model s presnosťou na cykly, z ktorého je možné vygenerovať detailný hardvérový popis v jazyku VHDL alebo CA simulátor.

Z pohľadu návrhu simulátora s presnosťou na inštrukcie je podstatná definícia inštrukčnej sady a architekturálnych zdrojov [5].

Definícia inštrukčnej sady

Jednou z kľúčových entít v jazyku *CodAL* je konštrukcia (objekt) *element*. Jeho definícia sa skladá zo sekcií: *assembler*, *binary*, *semantics* a *return*. Sekcia *assembler* určuje textový zápis objektu pre použitie v jazyku symbolických inštrukcií a sekcia *binary* definuje jeho reprezentáciu vo výslednom binárnom kóde po preklade. V časti označenej ako *semantics* je vyjadrená sémantická akcia spojená s daným objektom, ktorá je extrahovaná pri generovaní prekladača, a taktiež použitá pri interpretácii v IA simulátore. Informácia obsiahnutá v poslednej sekcii *return* je typicky používaná pri spájaní objektu s inými objektmi.

Príklad použitia spomínaných konštrukcií je uvedený na obrázku 5.1. Riadky 1-5 obsahujú definíciu registra *R1*. Rovnakým spôsobom je možné definovať aj ostatné procesorové registre a následne vytvoriť ich množinu, v danom príklade označenú identifikátorom *gpreg*.

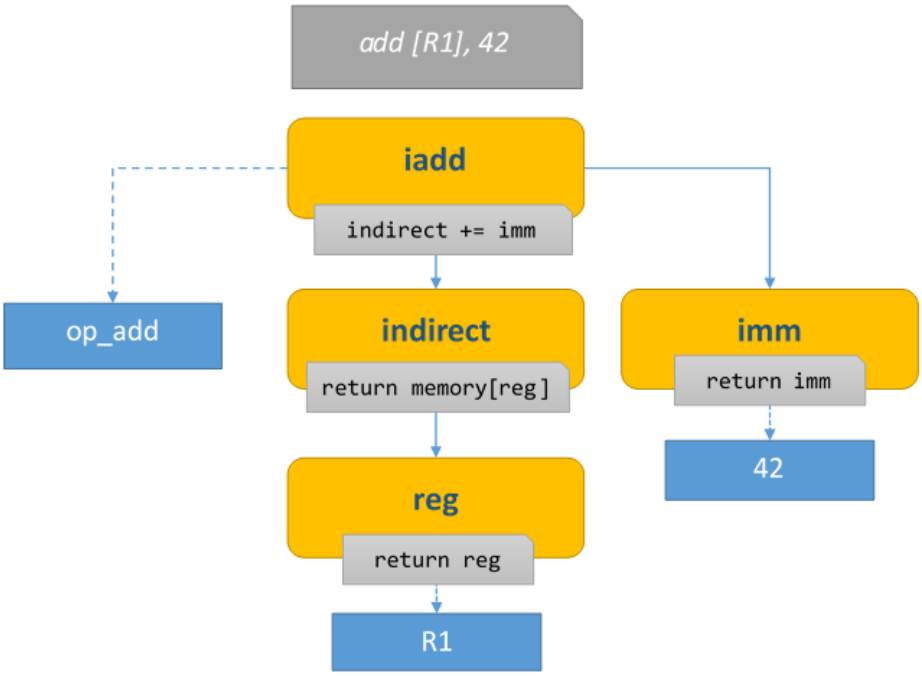
Množina registrov je následne použitá v definícii aritmetickej inštrukcie *iadd* na mieste operandov (riadok 12).

```
1  element reg1
2  {
3      assembler { "R1" };
4      binary { 1:bit[5] };
5      return { 1; };
6  };
7
8  set gpreg += reg1, reg2, ...;
9
10 element iadd
11 {
12     use gpreg as reg_dst, reg_src1, reg_src2;
13     ...
14     semantics
15     {
16         rf_gpr[reg_dst] = rf_gpr[reg_src1] + rf_gpr[reg_src2];
17     };
18 };
19
20 set isa += iadd, imul, idiv, ...;
```

Obr. 5.1: Ukážka kódu v jazyku *CodAL*

Pri preklade sa potom výraz *ADD R1, R2, R3* preloží ako kombinácia *binary* sekcií inštrukcie *i_add* a registrov *reg1*, *reg2* a *reg3*. Interpretácia zase využije *return* sekciu registrov na získanie ich indexov v registrovom poli, vypočíta súčet a uloží výsledok – použitie elementu ako indexu je vidieť na riadku 16.

Takýmto spôsobom je možné hierarchicky definovať celú inštrukčnú sadu (riadok 20), teda množinu obsahujúcu všetky inštrukcie.



Obr. 5.2: Ukážka hierarchickej definície inštrukcie

Inštrukcie a operandy potom v podobe elementov a ich množín vytvárajú hierarchickú štruktúru – príklad časti takéhoto stromu je znázornený na obrázku 5.2. Element *iadd* v tomto prípade modeluje aritmetickú inštrukciu podporujúcu nepriamy prístup do pamäte pomocou adresy uloženej v registri. To je možné dosiahnuť modelovaním príslušného operandu ako samostatného elementu *indirect*, s vlastným kódovaním a sémantikou. Inštrukcia *add [R1], 42* teda zahŕňa prečítanie hodnoty z registra, prístup do pamäte na danú adresu, vykonanie sčítania a uloženie výslednej hodnoty späť do pamäte. Obrázok je uvedený ako zjednodušený ilustračný príklad vyjadrovacích možností jazyka *CodAL* pri popise inštrukčnej sady.

CodAL obsahuje ďalšie konštrukcie ako napríklad sekcie *bundling* pre popis kombinácie viacerých inštrukcií do jedného slova pri architektúrach VLIW. Ďalej umožňuje tento jazyk popísať funkčné jednotky pomocou konštrukcií *event* a definovať stupne zreťazenej linky *pipeline*. Rovnako je možné namodelovať rozhrania medzi komponentmi (napr. medzi pamäťou a procesormi) používajúc rôzne komunikačné protokoly a určiť parametre ako bitová šírka alebo veľkosť či oneskorenie pamäte. To však súvisí s modelovaním hardvéru a informácie tohto druhu sú použité pri generovaní simulátora s presnosťou na takty. Pre simulátor inštrukčnej sady je podstatná sémantika inštrukcií a spôsob ich dekódovania.

Kapitola 6

Návrh simulátora pre Codasip Studio

Hlavnou motiváciou tejto práce je vytvorenie nového simulátora inštrukčnej sady použiteľného pri návrhu, ladení a optimalizácii aplikačne špecifických procesorov v prostredí *Codasip Studio*. Prioritou je rýchlosť simulácie.

6.1 Požadované vlastnosti

Univerzálnosť

Prostredie *Codasip Studio* je súbor nástrojov používaných na návrh aplikačne špecifických procesorov, ktorých architektúra je plne v réžii používateľa. Z existujúceho popisu procesora v jazyku *CodAL* je automaticky vygenerovaná sada nástrojov (*toolchain*) obsahujúca simulátor (IA, CA), profiler, prekladač, atď. Tieto nástroje sú špecifické pre daný procesor a vznikajú spojením a rozšírením predpripravených všeobecných komponentov (zdroje, rozhrania, pamäť...).

Nový simulátor by mal rešpektovať tento koncept. Nejedná sa o špecializovanú simuláciu konkrétnej inštrukčnej sady, ako je to napríklad pri simulátore *Spike*. Je potrebné navrhnúť všeobecné simulačné jadro a implementovať generovanie kódu špecifického pre konkrétny procesor podľa popisu v jazyku *CodAL*.

Pri tom je nutné počítať s potenciálne veľkými rozdielmi v architektúrach, napríklad rôzne dlhé inštrukčné kódy alebo spojenie viacerých inštrukcií v jednom slove pri procesoroch *VLIW* a ich kódovanie (*bundling*).

Rýchlosť

Kritickou požiadavkou na nový simulátor je rýchlosť. Cieľovou skupinou používateľov sú návrhári inštrukčnej sady a softvéroví vývojári, pre ktorých nie sú detaily hardvérovej implementácie kľúčové. Preto je postačujúcou simulácia s presnosťou na inštrukcie (*instruction accurate*). Naopak, podstatné pre vývoj softvéru je rýchle testovanie a možnosť simulácie potenciálne veľkých programov, menovite operačného systému, a to v reálnom čase.

Dôležité je identifikovať hlavné príčiny nedostatočnej rýchlosti existujúceho IA simulátora používaného v prostredí Codasip Studio a preniesť získané poznatky do návrhu nového simulátora. Konkrétne je potrebné optimalizovať plne interpretačný charakter simulácie a v čo najväčšej miere eliminovať duplicitné dekódovanie rovnakých inštrukcií.

Rozšíriteľnosť

Pri simulácii systémov je potrebné umožniť používateľovi kombinovať viaceré modely s rôznou úrovňou detailu podľa aktuálnej potreby. V prípade nového simulátora sa jedná najmä o pamäťový systém.

Codasip Studio štandardne pri IA simulácii používa jednoduchý softvérový model pamäte implementovaný hashovacou tabuľkou. Niekedy je ale potrebná väčšia kontrola nad pamäťovými operáciami, resp. môže byť výhodné využiť existujúci externý, potenciálne hierarchický pamäťový model. To je potrebné pri návrhu nového simulátora zohľadniť – optimalizácie pamäťových prístupov musia počítať s možnosťou užívateľsky definovanej obsluhy.

Podpora prerušení

Bežnou a často neodmysliteľnou súčasťou procesorov je obsluha prerušení. Prerušenia poskytujú procesoru možnosť reagovať na asynchrónne udalosti a tvoria alternatívu k aktívnemu čakaniu na zmenu hodnoty (*polling*). Vo vstavaných systémoch s veľkým dôrazom na minimalizáciu spotreby je to kľúčový mechanizmus – umožňuje prechod procesora do nízko-energetického režimu počas čakania na externú udalosť. Okrem toho sú prerušenia základným predpokladom k implementácii paralelizmu, prepínania procesov a oddelenia užívateľského a systémového (privilegovaného) režimu.

Nový simulátor teda musí obsluhu prerušení podporovať. Simulácia celého operačného systému kladie na efektívnosť implementácie vysoké požiadavky – napríklad frekvencia príchodu prerušenia od časovača spúšťajúceho plánovač procesov v Linuxovom jadre je typicky 100 - 1000Hz a vyžaduje rýchlu obsluhu.

V prípade synchrónnych prerušení súvisiacich s konkrétnou inštrukciou je obsluha jednoznačná – implementácia je možná napríklad vyvolaním a zachytením softvérovej výnimky. Komplikovanejší problém predstavujú externé asynchrónne prerušenia, ktorých kontrola musí byť implementovaná explicitne – buď v rámci hlavnej interpretačnej slučky alebo vložením testovacieho kódu priamo do vygenerovaných sekvencií inštrukcií pri prekladanom simulátore. Minimalizácia výskytu explicitných kontrol môže mať výrazný vplyv na celkový výkon [25].

Debugger a profiler

V rámci procesu vývoja softvéru aj návrhu inštrukčnej sady je potrebným nástrojom *debugger* umožňujúci vykonávanie programu po krokoch (riadkoch zdrojového kódu alebo jednotlivých inštrukciách) a zobrazujúci hodnoty premenných, registrov a obsahu pamäti. Podstatne skracuje dobu lokalizácie potenciálnych chýb v návrhu a implementácii softvéru, a preto je jeho podpora v novom simulátore nutnosťou.

Profiler je efektívnym nástrojom pri identifikácii výpočtovo náročných a často vykonávaných úsekov kódu, tzv. *hotspots*. Takéto úseky sú potenciálnymi kandidátmi na optimalizáciu a ich zefektívnenie má významný vplyv na rýchlosť celej aplikácie. Z hľadiska návrhu inštrukčnej sady dokáže profiler odhaliť nepoužívané inštrukcie, alebo naopak časti algoritmov, ktorých implementácia v hardvéri a sprístupnenie pomocou špecializovaných inštrukcií môže mať pozitívny vplyv na rýchlosť alebo spotrebu. Podpora profilera je teda rovnako žiadúcou vlastnosťou nového simulátora.

6.2 Súčasný Codasip IA simulátor

Existujúca implementácia simulátora s presnosťou na inštrukcie, ktorý je generovaný v rámci prostredia Codasip Studio je založená na čistej interpretácii. To znamená, že v každom kroku je z pamäte načítané inštrukčné slovo, prebehne jeho dekódovanie a zavolanie príslušnej C++ metódy implementujúcej sémantiku danej inštrukcie.

Dekódovanie inštrukcie je časovo náročné – naivná implementácia je založená na stavovom automate vygenerovanom z popisu kódovania inštrukčnej sady. V prípade Codasip Studia a jazyka *CodAL* je z definície inštrukcií vytvorený graf zložený z dvoch typov uzlov – elementov a setov. Pri analýze inštrukčného slova sa vykonáva postupný prechod tohto grafu, a to prostredníctvom volania vygenerovaných metód reprezentujúcich jednotlivé uzly. Sémantika inštrukcie je potenciálne rozptýlená vo viacerých uzloch, keďže každý element v jazyku *CodAL* môže definovať vlastné sémantické akcie (viď príklad na obrázku 5.2).

Veľkou nevýhodou tohto prístupu je opakované dekódovanie rovnakých inštrukcií, kde už len réžia súvisiaca s viacnásobným volaním C++ metód pri prechode grafu má poznateľný vplyv na spomalenie simulácie. Pri rozsiahlych modeloch môže hrať úlohu aj lokalita roztrúseného kódu. V procese optimalizácie by mala byť snaha minimalizovať réžiu dekódovania s cieľom priblížiť sa ideálnemu stavu, kedy jednej simulovanej inštrukcii odpovedá jedna inštrukcia kódu simulátora. To síce neplatí ani pri najrýchlejších prekladaných simulátoroch, avšak taký stav sa dá považovať za ideálny.

6.3 Návrh nového simulátora

Návrh a architektúra nového čiastočne prekladaného simulátora inštrukčnej sady vychádza z pôvodného IA simulátora používaného v prostredí Codasip Studio. Rovnako ako je to v prípade existujúceho riešenia, je potrebné oddeliť generické simulačné jadro a časti špecifické pre konkrétny model, ktoré sú generované na základe popisu v jazyku *CodAL*.

Optimalizácia sa sústreďuje na spôsob dekódovania a vykonávania inštrukcií pri interpretácii. Odstránením redundantných volaní funkcií a zlepšením priestorovej lokality kódu by podľa očakávania malo dôjsť k značnému zrýchleniu.

6.3.1 Zjednotenie sémantiky

Jedným z hlavných problémov súčasného simulátora je fragmentácia kódu vyjadrujúceho sémantiku jednej inštrukcie. Vychádza to z vnútornej stromovej reprezentácie inštrukčnej sady pozostávajúcej z elementov a ich množín. Každý element totiž môže definovať vlastnú sémantickú akciu – príkladom je nepriamy prístup do pamäte, keď elementom je operátor vracajúci hodnotu na špecifikovanej adrese, a tento element je použitý ako operand inej inštrukcie (napríklad aritmetického sčítania, viď obr. 5.2). Vtedy je pri interpretácii v abstraktnom strome vykonaný najprv uzol reprezentujúci prístup do pamäte, a potom v rámci iného uzla (na inom mieste v kóde simulátora) sa realizuje sčítanie.

Takéto usporiadanie je z hľadiska implementácie neefektívne. Je potrebné upraviť proces generovania dekodérov tak, aby každá inštrukcia bola reprezentovaná jednou funkciou zahŕňajúcou sémantiku všetkých hierarchicky vnorených elementov. Táto úprava môže znamenať nárast celkového objemu kódu, keďže aplikačná logika bude duplikovaná všade tam, kde je opakovane použitý ten istý element. Odpadne však volanie funkcií, a teda sa očakáva zrýchlenie. V prípade príliš veľkého nárastu kódu by sa však muselo počítať s vyčlene-

ním niektorých častí kódu do samostatných funkcií, neúmerne dlhé procedúry totiž môžu predstavovať problémy pri preklade [18].

Po úspešnom zjednotení sémantiky každej inštrukcie do ucelenej sekvencie príkazov sa automaticky zlepší priestorová lokalita pri interpretácii, čo je ďalšie pozitívum. Tieto bloky kódu sa dajú považovať za minimalistickú obdobu základných blokov (*basic blocks*) používaných v inteligentných prekladaných (*translated*) simulátoroch. Existujú síce len na úrovni jednej interpretovanej inštrukcie ale je to základný predpoklad k ďalšiemu postupu.

6.3.2 Zjednodušenie dekódovania

V momente kedy je každá inštrukcia reprezentovaná uceleným blokom kódu, je možné zjednodušiť samotný proces dekódovania. Prechod stromom a postupná identifikácia jednotlivých elementov už nie je potrebná. Stačí správne určiť operačný kód inštrukcie, rovnako ako je to v hardvéri, teda použitím bitovej masky. Komplikovanejšia situácia vzniká v prípade rôzne dlhých inštrukcií, to sa ale dá vyriešiť napríklad postupným vyskúšaním rôzne dlhých masiek od najdlhšej po najkratšiu. Hlavné je, že tak ako došlo k zjednoteniu interpretačného kódu pre jednotlivé inštrukcie, minimalizuje a centralizuje sa aj logika dekódovania. Takto ucelený kód má zároveň vyššiu perspektívnosť z hľadiska optimalizácie prekladačom.

6.3.3 Inštrukčná cache

Z implementačného hľadiska má význam zapuzdriť identifikáciu správnej inštrukcie do vyhľadávacej tabuľky. Tá na základe inštrukčného slova vyhľadá funkciu obsahujúcu príslušné sémantické akcie a vráti priamo jej adresu. Tým sa základný interpretačný algoritmus zjednoduší do podoby volania krátkej procedúry *step*:

```
step():
        instr = fetch(pc)
        pc = cache.get(instr).exec()
```

Táto tabuľka môže byť naplnená staticky, už pri generovaní simulátora.

6.3.4 Adresná cache

Napriek zjednodušeniu procesu dekódovania má simulačný algoritmus stále nevýhody. Tabuľka dekódovaných inštrukcií môže dosahovať veľké rozmery a náhodný prístup do nej môže byť kvôli možným častým výpadkom vo vyrovnávacej pamäti (cache) hostiteľského stroja neefektívny. Navyše, každé vyhľadanie inštrukcie zahŕňa bitové operácie za účelom identifikácie operačného kódu.

Vhodnou optimalizáciou je podľa vzoru simulátora *Spike* zavedenie krátkej tabuľky posledných vykonaných inštrukcií podľa adresy, teda hodnoty programového čítača. Vďaka obmedzenej veľkosti tejto tabuľky a organizácii inštrukcií podľa adresy by mala byť lokalita prístupov lepšia ako pri rozptýlenom prístupe, a to najmä v interpretovaných cykloch, resp. všeobecne pri opakovaní kódu.

Zároveň, keďže adresy v *cache* tabuľke nasledujú za sebou, je možné čiastočne rozvinúť simulačnú slučku a pokračovať vykonávaním nasledujúcej inštrukcie v tabuľke, kým je adresa platná, teda kým odpovedá aktuálnej hodnote inštrukčného čítača. Po vykonaní skokovej inštrukcie sa bude líšiť – vtedy je potrebné lineárny prechod ukončiť a správnu adresu nanovo vyhľadať.

Zrýchlenie by malo byť citeľné pri interpretácii sekvenčného kódu bez skokov, čo približne odpovedá myšlienke základných blokov (*basic blocks*) používaných pri prekladaných simulátoroch.

6.3.5 Model pamäti

Simulátory generované v prostredí *Codasip Studio* (IA aj CA) používajú generický a relatívne komplexný pamäťový modul podporujúci napríklad rôzne spôsoby zarovnania, *Big* aj *Little Endian*, užívateľsky definovanú bitovú šírku slova atď. Samostatne modelovaným zdrojom je rozhranie (*interface*), ktoré slúži na pripájanie rôznych komponentov, vrátane pamäti. Rozhrania je možné prepájať a vytvárať tak komunikačné kanály, pričom rôzne typy rozhraní využívajú rôzne protokoly (napr. *AHB 3 Lite*, *AXI 4 Lite* alebo proprietárny protokol *CLB - Codasip Local Bus*).

Z pohľadu CA modelu je typ rozhrania dôležitý kvôli spôsobu implementácie komunikačného protokolu – CA model sa zaoberá hodnotami jednotlivých signálov v každom takte a uvažuje vznik čakacích stavov, napr. v dôsledku vyššej latencie pamäte.

V rámci IA modelu trvá každá pamäťová operácia jeden takt, žiadne čakacie stavy nevznikajú. Nie je potrebné uchovávať žiadny vnútorný stav, implementácia je jednouchšia ako v prípade CA modelu, stále je však potrebné rozlišovať rôzne protokoly – procesor totiž používa rovnaké signály ako v prípade CA modelu, ktoré sú špecifické pre daný protokol a vyžadujú korektnú interpretáciu. Ide spravidla o spôsob interpretácie nezarovnanej adresy a o správne zarovnanie dát určených pre čítanie alebo zápis.

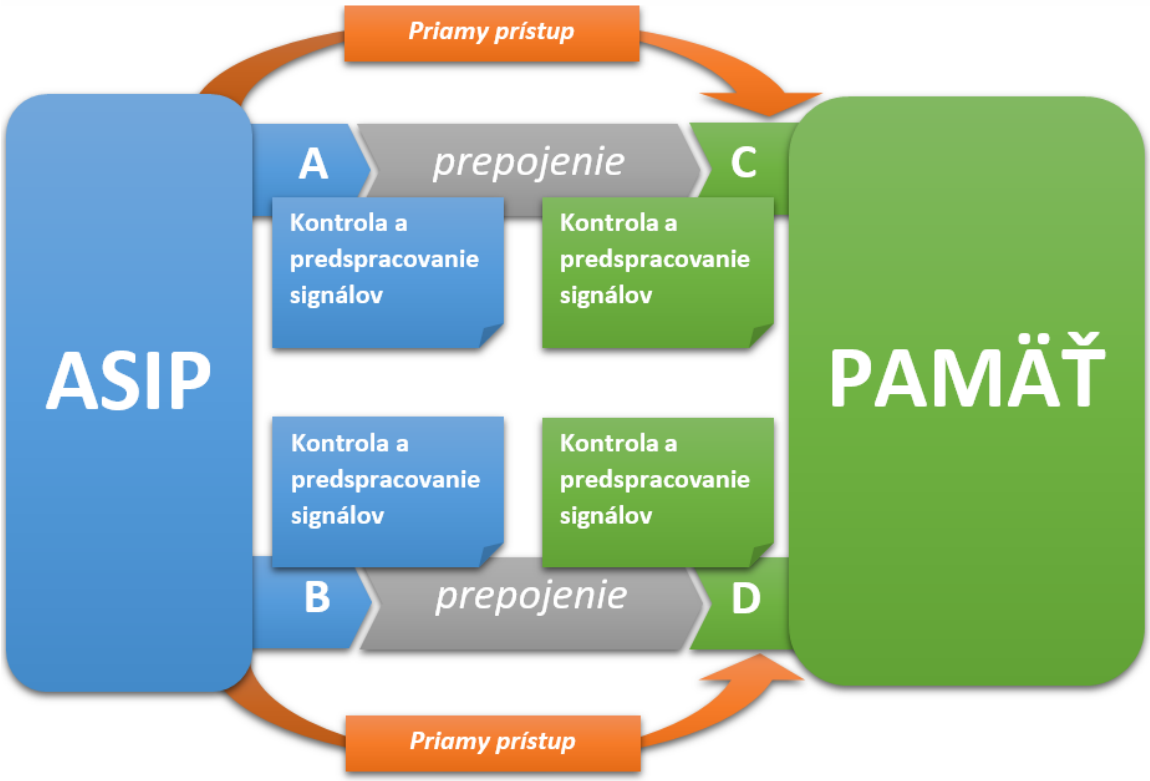
Kód zodpovedný za prístupy do pamäte patrí medzi najčastejšie vykonávané časti simulátora (tzv. *hotspot*) a jeho optimalizácia môže mať značný vplyv na rýchlosť celej simulácie. V súvislosti so zabezpečením vyššie popísanej požiadavky na variabilitu rozhraní, a tiež kvôli potrebe kontroly platnosti hodnôt jednotlivých signálov (napr. zarovnanie adresy alebo požadovaná veľkosť dát) je réžia každého jedného prístupu nezanedbateľná. Navyše, kvôli potrebe viacerých typov rozhraní, je posielanie dát medzi navzájom prepojenými rozhraniami riešené cez virtuálnu metódu, ktorá vyžaduje nepriamy skok cez tabuľku virtuálnych metód, a tým bráni niektorým optimalizáciám bežne vykonávaným pri kompilácii, napr. rozbalenie tela funkcie namiesto volania.

Vhodnou úpravou z hľadiska rýchlosti by mohlo byť nahradenie dvojice (alebo série) rozhraní zapojených za sebou jediným prístupom na koncové rozhranie, čím by sa odstránilo predávanie a viacnásobná kontrola hodnôt signálov cez viacero uzlov. Hlavne by však došlo k eliminácii virtuálneho volania, vďaka čomu by sa celý prístup do pamäte mohol zredukovať na niekoľko inštrukcií bez skoku.

Typická situácia je ilustrovaná na obrázku 6.1. Zobrazené sú dva hlavné bloky – *ASIP* (jadro aplikačne špecifického procesora) s rozhraniami *A* a *B* a pamäť s rozhraniami *C* a *D*. V praxi je jedno rozhranie použité na čítanie inštrukčných slov z programu a druhé na prístup k dátam (čítanie aj zápis). Na každom rozhraní je potrebné kontrolovať správnosť hodnôt signálov, alebo napríklad typ prístupu (pokus o zápis na rozhraní, ktoré podporuje iba čítanie). V niektorých prípadoch je tiež potrebné zarovnať adresu a správne posunúť dátové bity. Nahradením prístupu cez rozhrania a dynamicky vytvorený kanál priamym prístupom na rozhranie pamäte je možné značnú časť réžie eliminovať.

6.3.6 Minimalizácia kódu

Pre maximalizáciu rýchlosti je potrebné odstrániť čo najväčšie množstvo skokov počas behu simulátora, a to najmä v hlavnej simulačnej slučke, ktorá sa opakuje pre každú inštrukciu.



Obr. 6.1: Ilustrácia zapojenia pamäte s použitím rozhraní

Jedným zo spôsobov ako to dosiahnuť, je vloženie tela funkcie na miesto jej volania, teda použitie tzv. *inline* funkcií. Nevýhodou tohto prístupu je ale možný neúmerný nárast veľkosti kódu celého programu alebo jednotlivých funkcií, čo môže byť príčinou ešte väčšieho spomalenia v dôsledku častých výpadkov z inštrukčnej vyrovnávacej pamäte procesora (*L1 cache*). Nadmerné rozvíjanie volaní funkcií môže tiež neúnosne predĺžiť dobu kompilácie alebo dokonca spôsobiť zlyhanie prekladača z dôvodu nedostatku operačnej pamäte.

Je teda potrebné identifikovať funkcie, ktorých rozbalenie na miesta volaní bude prínosom z hľadiska rýchlosti a nespôsobí spomenuté komplikácie. Túto úlohu plní do určitej miery prekladač formou automatického rozbalenia volaní, avšak ručné optimalizácie môžu byť v niektorých prípadoch efektívnejšie.

Príkladom situácie, kedy má programátor možnosť využiť informácie o programe a vytvoriť tak optimálnejší kód, je ošetrenie výnimočných stavov – v prípade simulácie procesora môže byť príkladom obsluha prerušení alebo výnimiek.

Typicky ide o podmienený úsek často vykonávaného kódu (napr. v tele cyklu), pričom vo väčšine prípadov podmienka neplatí. Nahradenie tela podmieneného bloku volaním funkcie žiadnym spôsobom nespomalí beh slučky. Naopak, vďaka nahradeniu celého bloku jedinou inštrukciou volania sa veľkosť kódu zmenší, čo môže napríklad znížiť počet výpadkov z vyrovnávacej pamäte počas behu, a tým zvýšiť celkovú rýchlosť programu.

6.3.7 Obsluha prerušení

Podpora prerušení je bežnou súčasťou mnohých procesorových modelov. Typicky je riešená testovaním príznaku alebo kombinácie príznakov v každom takte, čo však môže mať na rýchlosť simulácie značný negatívny vplyv.

V praxi nie je absolútne nutné modelovať príchod prerušení s presnosťou na takt. V rámci optimalizácie je akceptovateľné kontrolovať výskyt prerušenia len raz za niekoľko taktov, po vykonaní niekoľkých simulovaných inštrukcií.

Aby bolo možné oddeliť kód súvisiaci so spracovaním prerušení od zvyšku simulačnej slučky, je vhodné navrhnúť systém, ktorý umožní autorovi modelu jednoducho popísať detekciu výskytu prerušenia, a tiež definovať sémantické akcie súvisiace s jeho obsluhou. Spôsob vyjadrenia by mal byť pri tom jednotný, nezávislý od konkrétneho modelu.

To je možné dosiahnuť určením vyhradenej funkcie v jazyku *CodAL*, do ktorej používateľ (autor modelu) vloží všetok kód súvisiaci s obsluhou prerušenia. Táto izolovaná procedúra môže byť potom kontrolovane volaná každých niekoľko taktov.

Zároveň, keďže existujú procesorové modely, ktoré prerušenia nepodporujú, je potrebné ponechať používateľovi možnosť túto podporu úplne vypnúť tak, aby nijakým spôsobom simuláciu nezaťažovala.

Kapitola 7

Implementácia

Implementácia nového čiastočne prekladaného simulátora stavia na existujúcom simulátore s presnosťou na inštrukcie, pričom sa za účelom dosiahnutia čo najvyššej rýchlosti snaží o minimalizáciu a vhodnejšiu organizáciu simulačného kódu a zavádza rôzne dodatočné optimalizácie.

Simulátor je reprezentovaný C++ triedou *Sim*, pričom jej zdrojový kód je generovaný na základe popisu procesora v jazyku *CodAL*. Táto kapitola popisuje programovú realizáciu najdôležitejších častí generátora produkujúceho simulačný kód, ako aj použitých generických komponentov nezávislých na konkrétnom procesorovom modeli.

7.1 Tabuľka inštrukcií

V predchádzajúcej kapitole bola popísaná grafová reprezentácia inštrukčnej sady a z nej vyplývajúca komplikovaná štruktúra dekodérov použitá v rámci existujúceho IA simulátora spolu s jej nevýhodami.

V snahe o zoskupenie úzko súvisiaceho simulačného kódu a zjednodušenie procesu dekódovania inštrukčného slova v každom takte je v rámci nového simulátora implementovaná jedno-úrovňová tabuľka všetkých inštrukcií.

Pre každú inštrukciu je vytvorená bitová maska určujúca, ktoré bity tvoria operačný kód, a k tomu samotná hodnota operačného kódu, pričom k jednej inštrukcii môže prislúchať aj viacero operačných kódov a masiek (v prípade aliasov). Dekódovanie inštrukčného slova potom spočíva v nájdení inštrukcie, ktorej operačný kód sa zhoduje s výsledkom bitového súčinu jej masky a daného slova.

Pre každú simulovanú inštrukciu je vygenerovaná C++ funkcia obsahujúca všetky sémantické akcie súvisiace s danou inštrukciou. Adresa príslušnej funkcie je vložená do tabuľky inštrukcií, takže po úspešnom dekódovaní je možne skočiť priamo do bloku kódu, ktorý danú inštrukciu reprezentuje, a vykonať ho ako celok, bez nutnosti ďalšej dynamickej identifikácie čiastkových elementov.

Vygenerovanie funkcií predstavujúcich jednotlivé inštrukcie vyžaduje sploštenie grafu inštrukčnej sady – roztrúsená sémantika vnorených uzlov musí byť rozbalená a vložená priamo do koreňových uzlov reprezentujúcich inštrukcie. Na to je potrebný prechod celým grafom a postupné rozvinutie všetkých prípustných kombinácií čiastkových elementov.

Prechod grafom a zhromaždenie všetkých vnorených uzlov ku každej inštrukcii zabezpečuje existujúci kód v internom modeli. Pre každú inštrukciu tak stačí prejsť plochý zoznam vnorených uzlov, sémantiku každého z nich prepísať do kódu C++ a zabaliť do funkcie re-

#	inštrukcia	kód	maska	implementácia
1	STORE reg	0x01000000	0xFF000000	*&i_store*
2	ADD reg, **reg**, **reg**	0x02000**1**00	0xFF000F00	*&i_add_reg_reg*
3	ADD reg, **reg**, **imm**	0x02000**2**00	0xFF000F00	*&i_add_reg_imm*
4	ADD reg, **reg**, **addr**	0x02000**3**00	0xFF000F00	*&i_add_reg_addr*
5	ADD reg, **imm**, **reg**	0x02000**4**00	0xFF000F00	*&i_add_imm_reg*
6	ADD reg, **imm**, **imm**	0x02000**8**00	0xFF000F00	*&i_add_imm_imm*
7	ADD reg, **imm**, **addr**	0x02000**C**00	0xFF000F00	*&i_add_imm_addr*
...	...	...	...	...

Tabuľka 7.1: Tabuľka inštrukcií

prezentujúcej danú inštrukciu. Výsledkom je celistvý blok simulačného kódu bez zbytočných skokov.

Problémom môže byť neúmerný nárast kódu, ak by bola inštrukčná sada príliš komplexná. Dôvodom je, že každá množina (*set*) sa explicitne rozvinie a vytvorí samostatnú variantu inštrukcie pre každú možnú hodnotu z danej množiny. Ak napríklad inštrukcia obsahuje operand typu množina, ktorý môže byť nahradený tromi možnými elementmi, po rozbalení budú existovať 3 rôzne inštrukcie reprezentované samostatnými blokmi kódu. Ak by však inštrukcia obsahovala dva takéto množinové operandy, výsledkom bude 9 (3 x 3) rôznych variant. A rovnakým spôsobom sa môžu vetviť aj samotné vnorené uzly, čím vzniká ešte väčšie množstvo kombinácií.

Ilustračný príklad možného kódovania inštrukcií je uvedený v tabuľke 7.1. Každá položka obsahuje inštrukčný kód, masku a adresu funkcie reprezentujúcej sémantiku danej inštrukcie. V prípade operácie *ADD* je ukázané možné rozptýlenie bitov tvoriacich operačný kód, z čoho vyplýva potreba bitovej masky. Zároveň je tu ukázané rozvinutie kombinácií operandov.

Predpokladajme, že v jazyku *CodAL* je definovaný jediný element *add* reprezentujúci inštrukciu, zložený z dvoch vnorených množinových operandov *op1* a *op2*, pričom *op1 = {reg, imm}* a *op2 = {reg, imm, addr}*. Pri generovaní funkcií reprezentujúcich sémantiku inštrukcií a plnení inštrukčnej tabuľky vznikne 6 samostatných variant operácie *ADD*. Vetvenie by sa mohlo skomplikovať ešte viac, ak by operand prístupu do pamäte *addr* mohol podporovať rôzne adresné režimy (absolútna adresa, adresa posunutá o hodnotu registra, atď.) – každá varianta obsahujúca operand *addr* by sa rozdelila na niekoľko ďalších inštrukcií.

7.2 Inštrukčná cache

Vyhľadanie správnej inštrukcie pri dekódovaní vyžaduje prejsť celú tabuľku a pre každú položku vykonať bitový súčin a porovnanie, časová zložitosť je teda lineárna. V prípade veľkej množiny inštrukcií môže mať opakovaný lineárny prechod výrazný negatívny dopad na rýchlosť simulácie.

Za účelom eliminácie negatívneho vplyvu zdĺhavého prehľadávania celej množiny inštrukcií je v novom simulátore zavedená krátka tabuľka *cache* organizovaná podľa hodnoty inštrukčného slova. Pri dekódovaní sa najprv vyhľadáva v rýchlej *cache*, a až v prípade výpadku sa pristupuje k prehľadávaniu celej inštrukčnej sady. Nájdená inštrukcia je potom uložená do tabuľky spolu s kľúčom odvodeným z inštrukčného slova.

#	inštrukcia	kód	kľúč
BIG ENDIAN			
1	**ADD R1, R1, 1**	0x02010201	**0x0201**
2	**ADD R1, R1, 2**	0x02010202	**0x0202**
3	**SUB R1, R1, 1**	0x03010201	**0x0201**
LITTLE ENDIAN			
1	**ADD R1, R1, 1**	0x01020102	**0x0102**
2	**ADD R1, R1, 2**	0x02020102	**0x0102**
3	**SUB R1, R1, 1**	0x01020103	**0x0103**

Tabuľka 7.2: Ukážka mapovania inštrukcií na položky v inštrukčnej *cache*

Ako funkcia výpočtu kľúča (*hash*) je použitý jednoduchý zvyšok po delení inštrukčného slova veľkosťou tabuľky. Z toho vyplýva, že úspešnosť vyhľadávacej *cache* tabuľky je do veľkej miery závislá od spôsobu kódovania inštrukcií. Operácia zvyšku po delení totiž spôsobí, že výsledná hodnota kľúča je tvorená niekoľkými spodnými bitmi inštrukčného slova. Ak sa však operačný kód, ktorý je z hľadiska identifikácie inštrukcie rozhodujúci, nachádza na vrchných bitoch inštrukčného slova, môže sa stať, že hodnota kľúča bude tvorená len zakódovanými operandmi. To radikálne degraduje efektivitu vyhľadávania – rovnaká inštrukcia môže zabrať niekoľko položiek v tabuľke, keďže rôzne hodnoty operandov spôsobia vytvorenie odlišných kľúčov (*homonymá*). Zároveň, ak sú na spodných bitoch kódované napríklad registrové operandy, hrozí vysoké riziko vzniku *synoným* – rôzne inštrukcie môžu používať rovnaké registre, ktorých počet je typicky relatívne nízky, a tým pádom sa budú mapovať na rovnakú položku v tabuľke.

Príklad vzniku kolízií v navrhnutej inštrukčnej cache ilustruje tabuľka 7.2. Použitý je rovnaký spôsob kódovania ako v príklade z predchádzajúcej podkapitoly – typ operácie je určený prvým bajtom inštrukčného slova, operandy sú kódované poslednými bitmi.

Pri použití takéhoto kódovania je rozhodujúci spôsob interpretácie sekvencie bajtov – ak uvažujeme *Little Endian*, operačný kód sa nachádza na spodných bitoch, vďaka čomu je vypočítaný kľúč (zvyšok po delení veľkosťou tabuľky) nezávislý od hodnôt operandov (viď inštr. 1 a 2), a zároveň závislý od typu operácie (viď. inštr. 1 a 3). Maximálne nepriaznivá situácia nastáva v prípade *Big Endian* – operačný kód je na najvyšších bitoch a pri výpočte kľúča je úplne zanedbaný. Preto je operáciám *ADD* a *SUB* (inštr. 1 a 2) pridelený rovnaký kľúč, výsledkom čoho je kolízia (*synonymá*). Navyše, rôzne hodnoty operandov v prípade inštrukcií 1 a 2 spôsobia výpočet dvoch rôznych kľúčov pre tú istú operáciu (*homonymá*).

Oba problémy – vznik synoným aj homoným – zvyšujú mieru výpadkov a tým predlžujú priemernú dobu dekódovania, keďže následkom každého neúspešného prístupu (*cache miss*) je lineárne prehľadávanie celej množiny inštrukcií. Hlavnou komplikáciou pri návrhu efektívnejšej funkcie výpočtu kľúča, je príliš vysoká variabilita spôsobu kódovania. Je plne v réžii návrhára inštrukčnej sady definovať, ako budú inštrukcie kódované, a ktoré bity budú niesť operačný kód, pričom umiestnenie týchto bitov v rámci inštrukčného slova nemusí byť nutne jednotné pre celú inštrukčnú sadu. Príklad je uvedený v tabuľke 7.1 – druh operácie je určený najvyššími bitmi inštrukčného slova, avšak konkrétna varianta inštrukcie závisí od hodnoty druhého najnižšieho bajtu. Podobnú schému kódovania využíva napríklad aj inštrukčná sada *RISC-V* – bity tvoriace operačný kód sú rozmiestnené na viacerých miestach v inštrukčnom slove [16].

7.3 Adresná cache

V rámci snahy o ďalšie zefektívnenie procesu dekódovania inštrukcií, je v novom simulátore pred inštrukčnú *cache* predradená ešte jedna medzivrstva – adresná *cache*. Dôvodom jej zavedenia je snaha využiť priestorovú lokalitu interpretovaného kódu. Častou súčasťou bežných programov sú totiž cykly, v ktorých dochádza k opakovanému vykonávaniu rovnakých sekvencií inštrukcií. Ak sú tieto sekvencie relatívne krátke, je možné ich uchovať v tabuľke a vyhľadávať priamo podľa adresy. Tým je takmer úplne eliminovaná réžia dekódovania. Do tabuľky inštrukcií sa pristupuje len v prvej iterácii interpretovaného cyklu, pričom ani potenciálne kolízie medzi jednotlivými inštrukciami nespôsobujú problém. Po naplnení položky v tabuľke adries môže byť príslušná položka v inštrukčnej *cache* zneplatnená a prepísaná inou inštrukciou, jej obsah už nie je potrebný.

Nezanedbateľným prínosom je zníženie počtu prístupov do simulovanej pamäte. Bez použitia adresnej *cache* je nutné v každom takte prečítať nové slovo cez pamäťové rozhranie, čo zahŕňa napríklad réžiu spojenú s kontrolou adresy alebo kopírovanie dát. Naproti tomu, úspešné vyhľadanie v tabuľke adries umožňuje pamäťový systém úplne obísť – adresa funkcie reprezentujúcej danú inštrukciu spolu so vstupom (operandmi) je známa priamo z tabuľky.

Vytvorenie mapovacej *hash* funkcie je jednoduché – uvažujúc sekvencie za sebou idúcich adries, sú horné bity adresy zanedbané, predpokladá sa ich konštantnosť. Zároveň, keďže inštrukcie sú typicky širšie ako jeden bajt, je možné zanedbať aj niekoľko spodných bitov. Konkrétne ide o počet daný dvojkovým logaritmom šírky inštrukcie, teda napríklad pre inštrukčné slová s dĺžkou 4 bajty je možné zanedbať najnižšie 2 bity (adresy sa líšia o 4), pre slová dlhé 8 bajtov to budú 3 bity (adresy sa líšia o 8).

Výsledná hodnota kľúča k do tabuľky je teda získaná ako zvyšok po delení adresy a (bitovo posunutej doprava o dvojkový logaritmus dĺžky inštrukčného slova D v bajtoch) veľkosťou tabuľky N. Výpočet je ilustrovaný vzorcom 7.1.

$$k(a) = (a >> log_2(D)) \mod N \tag{7.1}$$

Veľkosť tabuľky je konštantná. Bola zvolená ako mocnina dvojky (konkrétne 1024), vďaka čomu môže byť časovo náročná operácia zvyšku po delení nahradená bitovým súčinom ($\wedge$). Vzorec 7.2 obsahuje danú úpravu.

$$k(a) = (a >> log_2(D)) \wedge (N - 1) \tag{7.2}$$

V prípade, že inštrukčná sada obsahuje inštrukcie rôznej dĺžky, je na mieste konštanty D použitý najväčší spoločný násobok všetkých podporovaných dĺžok odpovedajúci najmenšej hodnote, o ktorú sa budú adresy v programe líšiť (pre dĺžky 4 a 8 bajtov bude hodnota D rovná 4, pre dĺžky 4, 6 a 8 to bude 2, atď).

Osobitný prístup vyžadujú procesory typu *VLIW*, kde jedno inštrukčné slovo obsahuje niekoľko inštrukcií. Keďže vykonávané su celé n-tice (adresy všetkých skokov musia byť zarovnané na dĺžku celého inštrukčného slova), je výhodné, aby každá položka v adresnej *cache* obsahovala všetkých n inštrukcií tvoriacich jedno slovo. Za týmto účelom je vytvorená varianta vyhľadávacej tabuľky, v ktorej každá položka obsahuje pole inštrukcií (odkazov na funkcie implementujúce dané inštrukcie).

7.4 Simulačná slučka

Hlavná simulačná slučka pozostáva z opakovaného volania metódy *ClockCycle*, ktorej úlohou je v prípade IA simulácie zavolanie procedúry reprezentujúcej hlavnú udalosť *main* definovanú v rámci popisu modelu procesora v jazyku *CodAL*. Udalosť *main* obsahuje sémantické akcie vykonávané v každom takte procesora – typicky ide o prečítanie inštrukčného slova z pamäti, inkrementáciu inštrukčného čítača a aktiváciu dekodéra, ktorý zabezpečí správnu interpretáciu danej inštrukcie. Okrem toho môže obsahovať dodatočnú logiku napríklad v súvislosti s obsluhou prerušení.

V rámci nasadenia inštrukčnej a adresnej *cache* je generátor nového simulátora navrhnutý tak, aby z procedúry *main* odstránil aktiváciu dekodérov, a namiesto toho vložil nový kód zabezpečujúci optimalizovaný prístup s použitím vyhľadávacích tabuliek. Na to je potrebné vykonať analýzu užívateľského kódu v jazyku *CodAL*, ktorý je interne reprezentovaný v podobe stromu. Všetky analýzy a prechody sú realizované s použitím návrhového vzoru *Visitor*, ktorý slúži práve ako abstrakcia prechodu nad zloženou (stromovou) štruktúrou a umožňuje deklaratívne definovať akcie vykonané pre jednotlivé typy uzlov.

Jadro simulácie je teda pozmenené tak, že v rámci každého taktu (volania metódy *ClockCycle*) je na začiatku prečítaná aktuálna hodnota inštrukčného čítača *pc* a na základe nej je pristúpené do adresnej *cache*. V prípade úspešného vyhľadania (*hit*) je priamo vykonaná inštrukcia z tabuľky (zavolá sa príslušná funkcia spolu s uloženými operandmi). Ak sa reálna hodnota adresy v tabuľke líši od požadovanej (*miss*), pristúpi sa do pamäte a prečíta sa inštrukčné slovo z adresy *pc*, ktoré je následne použité pri prístupe do inštrukčnej *cache*. Tá buď priamo vráti dekódovanú inštrukciu (ak je prítomná), alebo prejde celú množinu inštrukcií a pomocou bitovej masky a porovnania identifikuje položku reprezentujúcu hľadanú inštrukciu. Následne zaistí aktualizáciu hodnoty v *cache*.

Po úspešnom vyhľadaní (dekódovaní) je aktivovaná hlavná udalosť *main*, ktorá už však kód zodpovedný za dekódovanie neobsahuje, a v zápätí je vykonaná samotná inštrukcia. Tento koncept predpokladá, že v pôvodnej užívateľom definovanej procedúre *main* sa aktivácia dekodérov nachádza na konci, za aktualizáciou všetkých signálov a inštrukčného čítača. Ak by užívateľský kód obsahoval komplikovanú logiku, kde by aktivácia dekodérov bola prípadne vnorená do tela podmienky, výsledný simulátor by nemusel pracovať správne. To je však obmedzením navrhnutej architektúry v prospech jednoduchosti a maximalizácie výkonu.

Jednotlivé kroky sú zhrnuté v algoritme 7.1.

```
while !finished():
    pc = read_pc();
    if address_cache_miss(pc):
        word = memory_read(pc) # access program memory
        if instr_cache_miss(word):
            instr = all_instructions.find(word) # slow search
            instr_cache_update(word, instr) # update instruction cache
        address_cache_update(pc, instr_cache(word)) # update address cache
    model_main() # activate main() from the model
    address_cache(pc).execute() # execute the instruction
```

Algoritmus 7.1: Simulačná slučka

Čiastočné rozvinutie

Medzi používané techniky pri optimalizácii programov patrí rozvinutie cyklov (*loop unrolling*). Táto transformácia je aplikovaná aj v prípade nového simulátora, s využitím skutočnosti, že položky vo vyhľadávacej tabuľke adries reprezentujúce za sebou idúce adresy v programe sú skutočne uložené v pamäti za sebou. Vďaka tomu je možné v opakovaných prístupoch do *cache* preskočiť výpočet kľúča (*hash*) a špekulatívne prejsť na ďalšiu položku. Úprava zachytáva algoritmus 7.2, pričom vykonanie inštrukcie je zabalené v makre *ACCESS_ADDRESS_CACHE*.

```
#define ACCESS_ADDRESS_CACHE() \
    model_main() \
    addr.execute() \
    addr = next(addr) \
    if !valid(addr, pc): \
        continue

while !finished():
    pc = read_pc();
    # handle cache miss ...
    addr = address_cache(pc)

    ACCESS_ADDRESS_CACHE()
    ACCESS_ADDRESS_CACHE()
    ACCESS_ADDRESS_CACHE()
    # repeat ...
```

Algoritmus 7.2: Rozvinutie simulačnej slučky

7.5 Minimalizácia kódu

Na rýchlosť programu vplýva okrem iného aj jeho celková veľkosť, preto je snaha objem generovaného kódu minimalizovať. V rámci spracovania popisu modelu v jazyku *CodAL* a jeho prekladu do jazyka C++ sú aplikované rôzne transformácie, napríklad eliminácia mŕtveho kódu, nahradenie konštantných výrazov konštantami, odstránenie nepoužitých funkcií, atď.

Generátor nového simulátora pridáva elimináciu zápisov do premenných, ktoré nie sú nikdy čítané, pričom premennými sa rozumejú zdroje procesora, konkrétne registre a signály. Najprv prebehne identifikácia zdrojov, ktoré sú aspoň raz čítané, a teda zápisy do nich musia ostať zachované. Následne je vykonaný prechod, pri ktorom sú nepotrebné zápisy postupne odstraňované, a zároveň, ak je odstránený príkaz jediným príkazom nadradeného zloženého príkazu, dôjde aj k jeho odstráneniu. Príkladom je podmienený zápis do nepoužitého zdroja – eliminovaná je okrem samotného príkazu aj celá podmienka.

Jedným z cieľov, ktoré táto transformácia sleduje, je odstránenie zbytočného prístupu do simulovanej pamäte. Keďže z kódu hlavnej procedúry modelu *main* je odobratá aktivácia dekodéra, register obsahujúci načítané slovo z pamäte sa stáva nepoužitým. Vďaka tomu môžu byť zápisy doňho vymazané, a teda by malo zmiznúť aj prečítanie inštrukčného slova z pamäte.

Problémom sa ukázali byť volania vstavaných funkcií ako *codasip_log*, ktoré produkujú výpisy na štandardný výstup pri ladení. V prípade, že model obsahuje výpis inštrukčného

slova, zápis do príslušného registra, a tým pádom aj čítanie slova z pamäte nemôžu byť odstránené. V záujme výkonu nového simulátora sú preto všetky ladiace výpisy vo výslednom kóde automaticky vynechané.

Rozbalenie volaní funkcií

V rámci odstránenia čo najväčšieho počtu skokov, ktoré simulátor spomaľujú, dochádza k rozbaľovaniu niektorých funkcií na miesta ich volaní. To je do určitej miery v rozpore so snahou dosiahnuť čo najkratší kód (hoci v niektorých prípadoch môže rozbalenie funkcie vo výsledku spôsobiť aj skrátenie výsledného kódu). Preto je nutné rozhodnúť, ktoré funkcie majú byť rozvinuté, a u ktorých by to bolo naopak kontraproduktívne.

Pri testovaní generátora na existujúcich procesorových modeloch sa (podľa očakávaní) neosvedčilo označiť hromadne všetky funkcie ako *inline*. Doba kompilácie sa pri niektorých vygenerovaných simulátoroch predĺžila až nad 30 minút, pričom výsledný program bol pomalší ako bez aplikácie rozvinutia.

V programe však existuje viacero vetiev, ku ktorých vykonaniu dochádza len výnimočne – rozvinutie volaní v rámci nich je zbytočné a neefektívne. Najlepší odhad o tom, ktoré funkcie sa budú volať často, a ktoré len občas, má používateľ, teda autor modelu. Preto je rozhodnutie o aplikácii rozvinutia na ňom. Vo vygenerovanom zdrojovom kóde C++ sú ako *inline* označené len tie funkcie, ktoré sú takto explicitne špecifikované aj v modeli. Generátor však pre každú funkciu v jazyku *CodAL*, ktorá nie je označená ako *inline*, vypíše upozornenie, keďže v prípade niektorých často vykonávaných funkcií môže ich nerozvinutie znamenať pokles výkonu v desiatkach percent. Používateľ má zároveň možnosť experimentálne zistiť, ktoré konkrétne funkcie je výhodné rozvinúť práve pre jeho konkrétny model.

Výnimkou je funkcia reprezentujúca hlavnú udalosť modelu *main*. Tá je fixne označená ako *inline* z dôvodu vysokej frekvencie volaní (pred každou interpretovanou inštrukciou). Problémom je jej veľkosť, keďže v rámci metódy *ClockCycle* je kvôli čiastočnému rozvinutiu simulačnej slučky zavolaná N krát. Jej rozbalenie na mieste každého volania teda spôsobí N-násobný nárast kódu. Vďaka vyššie aplikovaným optimalizáciám, konkrétne odstráneniu aktivácie dekodérov, zbytočných premenných, pamäťových prístupov a pomocných výpisov, sa väčšinou užívateľom napísaná hlavná procedúra dostatočne zjednoduší. Niekedy však môže obsahovať ďalšiu logiku ošetrujúcu rôzne výnimočné stavy testovaním príznakov, resp. nastavovaním iných príznakov – ide napr. o spracovanie prerušení alebo výnimiek. V týchto prípadoch môže byť potrebné upraviť model a vyňať časti hlavnej procedúry do čiastkových funkcií, tak aby bola hlavná procedúra čo najjednoduchšia.

7.6 Prerušenia

V rámci návrhu bola prezentovaná myšlienka oddelenia kódu zodpovedného za obsluhu prerušení od hlavnej procedúry modelu *main*. Nový simulátor preto ponúka možnosť definovať v jazyku *CodAL* funkciu *do_interrupt* zodpovednú za spracovanie prerušenia, pričom volanie tejto funkcie je zabezpečené generátorom. Detekciu príchodu prerušenia môže používateľ popísať definovaním funkcie *is_interrupt*. Oddelenie detekcie od samotnej obsluhy má význam z hľadiska efektívnejšieho usporiadania kódu – vďaka rozbaleniu volania funkcie *is_interrupt*, ktorá typicky obsahuje len jednoduchý test príznaku, prípadne kombinácie príznakov, je réžia spojená s jej volaním eliminovaná. Zároveň, keďže sa dá predpokladať, že vo väčšine prípadov bude výsledok negatívny a obsluha *do_interrupt* sa nevyvolá, je efektívnejšie ju ponechať nerozbalenú, vďaka čomu je výsledný kód kratší.

Detekcia príchodu prerušení je vyňatá na začiatok metódy *ClockCycle* (algoritmus 7.3), dôsledkom čoho je oneskorená (nepresná) obsluha. Čiastočné rozvinutie simulačnej slučky totiž spôsobí, že medzi reálnym príchodom a jeho zistením môže byť vykonaných niekoľko inštrukcií, výhodou je ale vyšší výkon. Miera nepresnosti je obmedzená a akceptovateľná.

V prípade, že používateľ procedúru *do_interrupt* nedefinuje, detekcia príchodu prerušenia je úplne vynechaná a simulácia beží bez zbytočného spomalenia.

```
while !finished():
    if is_interrupt(): # inlined
        do_interrupt() # ordinary call
    # read pc, handle cache miss ...
    ACCESS_ADDRESS_CACHE()
    ACCESS_ADDRESS_CACHE()
    ACCESS_ADDRESS_CACHE()
    # repeat ...
```

Algoritmus 7.3: Detekcia príchodu prerušenia

7.7 Pamäťové prístupy

Pôvodný IA simulátor podporuje flexibilný systém komunikačných rozhraní, ktoré je možné medzi sebou navzájom prepájať. Tieto prepojenia sú vytvárané dynamicky s využitím ukazovateľov, a samotná komunikácia je potom realizovaná volaním virtuálnych metód.

Ukázalo sa, že virtuálne volania majú na rýchlosť simulácie značný negatívny vplyv – dôvodom je okrem samotnej réžie spojenej s nepriamym skokom cez tabuľku virtuálnych metód aj neschopnosť prekladača tieto volania rozbaliť a aplikovať ďalšie pokročilé optimalizácie.

Generátor nového simulátora virtuálne volania odstraňuje využívajúc skutočnosť, že prepojenia medzi jednotlivými rozhraniami sú známe už v dobe prekladu modelu. Ak teda procesor obsahuje rozhranie A, ktoré je pripojené na pamäťové rozhranie B, je možné každý prístup na A nahradiť priamym prístupom na B.

Problémom je, že v prípade niektorých komunikačných protokolov sú dáta na rozhraní A určitým spôsobom transformované (príkladom je bitový posun závislý od zarovnania adresy). Riešením je extrahovanie kódu, ktorý tieto nutné transformácie realizuje, do samostatnej metódy, ktorej volanie je potom vložené spolu s prístupom na B namiesto prístupu na A.

Kapitola 8

Testovanie

Súčasťou vývoja nového simulátora bolo priebežné testovanie. Okrem overovania správnej funkčnosti bol pri každej optimalizácii zároveň sledovaný vplyv na výkon a rýchlosť bola porovnávaná s rýchlosťou pôvodného IA simulátora.

Testovanie nového simulátora prebiehalo na štyroch vybraných procesorových modeloch: *Codasip uRISC*, *Codix Berkelium*, *Codix Helium* a *Codasip uVLIW*. *Codasip uRISC* a *Codasip uVLIW* sú minimalistické demonštračné modely používané na ukážku spôsobu návrhu aplikačne špecifických procesorov s architektúrou *RISC* a *VLIW*. *Codix Berkelium* a *Codix Helium* sú modely reálne používaných mikroprocesorov, pričom konkrétne *Codix Berkelium* je implementáciou štandardu inštrukčnej sady *RISC-V* od spoločnosti *Codasip*.

Pre každý z týchto modelov bol vygenerovaný špecifický prekladač, ktorým boli preložené testovacie programy napísané v jazyku C do binárnej podoby. Po vygenerovaní simulátora bolo následne možné tieto preložené aplikácie spustiť a sledovať dĺžku simulácie.

Keďže nový simulátor podporuje možnosť vykonávania simulácie po krokoch (režim *debugger*), je možné simuláciu pozastaviť a sledovať napríklad hodnoty jednotlivých premenných alebo registrov.

8.1 Porovnanie výkonu

Pri testovaní a porovnávaní výkonu bolo použitých niekoľko testovacích aplikácií a pre simuláciu každej aplikácie bola zaznamenaná doba trvania v sekundách. Časový údaj však pre zmysluplné porovnanie výkonu nestačí, keďže jednotlivé aplikácie majú samy osebe rôznu veľkosť. Preto bol za jednotku porovnania zvolený pomer počtu interpretovaných inštrukcií a doby trvania simulácie, konkrétne milión inštrukcií za sekundu (*MIPS – million instructions per second*).

Pre každú testovaciu kombináciu (procesorový model a testovacia aplikácia) bolo potrebné vygenerovať simulátor, preložiť aplikáciu a spustiť simuláciu. Na základe informácie o počte vykonaných inštrukcií (N) zo simulátora a doby behu (t) v sekundách bola následné vypočítaná a zaznamenaná hodnota *MIPS* podľa vzorca 8.1.

$$MIPS = \frac{N}{t} 10^{-6} \tag{8.1}$$

Celý tento proces je automatizovaný skriptom v jazyku *python*, ktorého parametrami sú procesorový model, zdrojový kód testovacej aplikácie a tiež požadovaný počet opakovaní simulácie. Výstupom je zoznam hodnôt vo formáte *CSV* (*Comma Separated Values*) obsa-

hujúci priemernú dobu trvania spolu s rozptylom, počet inštrukcií a vypočítanú hodnotu *MIPS*.

Meranie výkonu simulátorov – pôvodného aj nového – bolo vykonané na niekoľkých krátkych testovacích aplikáciách:

- *dijkstra.c* - výpočet najkratších ciest medzi uzlami podľa Dijkstrovho algoritmu,
- *quicksort.c* - radenie prvkov v poli algoritmom *Quicksort*,
- *isqrt.c* - výpočet odmocniny bez použitia násobenia alebo delenia,
- *sha.c* - výpočet hodnoty *hash* podľa algoritmu *SHA*,
- *bitcnt.c* - spočítanie jednotkových bitov v slove,
- *crc.c* - výpočet cyklického redundantného súčtu *CRC*.

Zdrojový kód každej z aplikácií obsahuje algoritmus (podľa názvu súboru) zabalený do cyklu s pevne definovaným počtom iterácií. Počet opakovaní bol pre každú aplikáciu experimentálne určený tak, aby celková dĺžka simulácie trvala aspoň niekoľko sekúnd. Pri príliš krátkych simuláciách bolo totiž meranie nepresné a výsledky by boli nepoužiteľné. Prakticky sa doba behu pohybovala v závislosti od aplikácie a modelu v rozmedzí 5 sekúnd až 3 minúty.

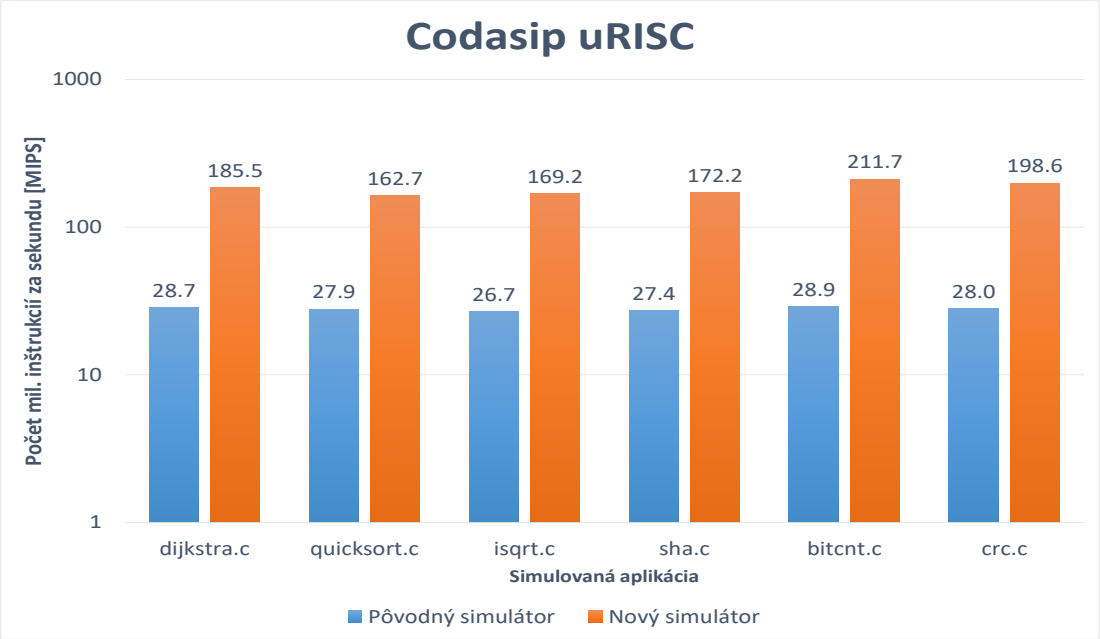
Graf na obrázku 8.1 zobrazuje výsledky sady meraní pre model *Codasip uRISC*. Každá testovacia aplikácia bola postupne spustená v pôvodnom aj novom simulátore, pričom každý beh bol zopakovaný 10-krát a do úvahy bola braná priemerná doba trvania. Je vidieť, že výkon simulátorov nebol významne ovplyvnený konkrétnou aplikáciou. Na druhej strane, z grafu vyplýva, že v prípade všetkých testovacích aplikácií sa podarilo zvýšiť rýchlosť simulácie, a to 6 až 7 násobne.

Ako je vidieť na grafe 8.2, zrýchlenie sa prejavilo aj na ostatných modeloch, a to približne v rovnakej miere (6 až 8-krát). Výnimkou sa ukázal byť model *Codasip uVLIW*, pri ktorom je zrýchlenie len cca 3-násobné. Dôvodom je pridaná réžia súvisiaca s dekódovaním zloženého inštrukčného slova. Architektúra *VLIW* totiž pracuje so širokými slovami (v tomto prípade ide napríklad konkrétne o 160 bitové slovo), ktoré sú v rámci simulátora reprezentované C++ objektmi a akékoľvek operácie s nimi sú výrazne časovo náročnejšie ako pri práci s natívnymi dátovými typmi.

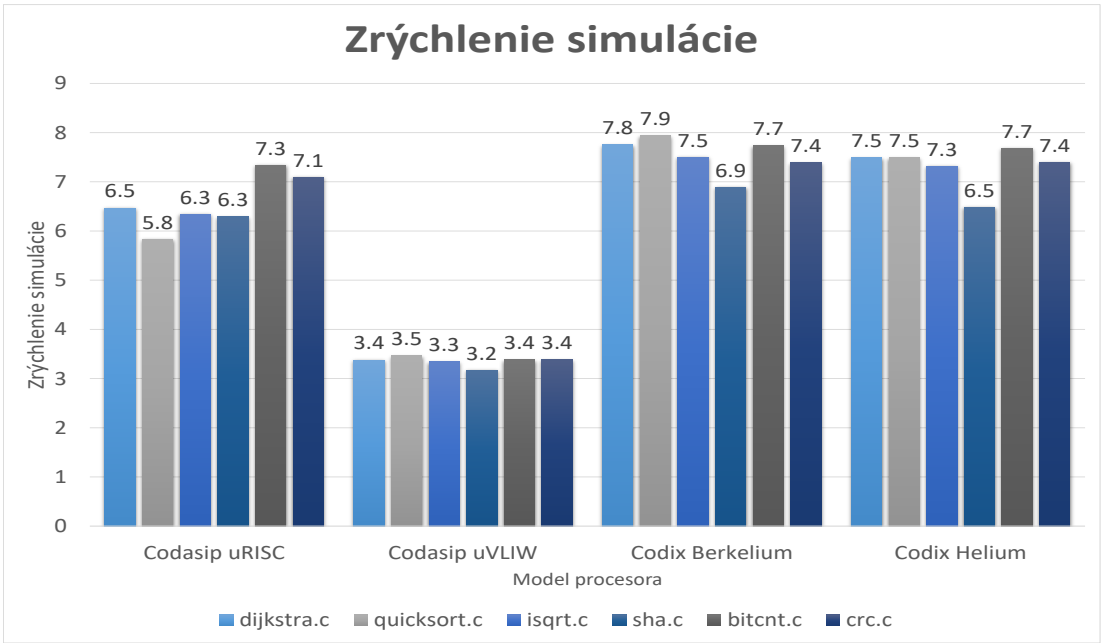
8.2 Vplyv výpadkov z adresnej cache

Aplikácie použité pri predchádzajúcom testovaní boli relatívne krátke – do adresnej vyhľadávacej tabuľky vošiel kód celého programu. V praxi sa však môžu vyskytovať dlhšie a zložitejšie simulované aplikácie, pri ktorých nemusí byť usporiadanie kódu priaznivé. Pre komplexnejšiu predstavu o možnostiach nového simulátora je preto potrebné otestovať aj scenár s veľkým množstvom výpadkov.

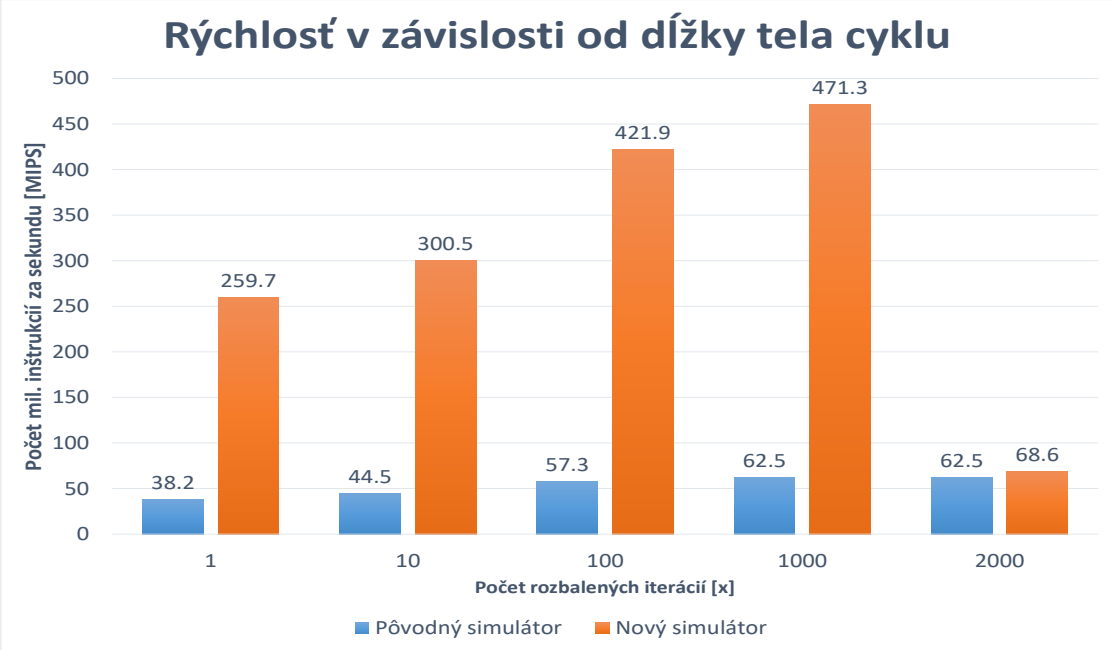
Za účelom zistenia vplyvu výpadkov pri nepriaznivom usporiadaní interpretovaného kódu bola vytvorená aplikácia *nops* (algoritmus 8.1). Ide o program pozostávajúci zo sekvencie prázdnych inštrukcií *NOP*, zabalenej v cykle s pevným počtom iterácií. Obaľujúci cyklus slúži na predĺženie doby behu aplikácie tak, aby bolo možné sledovať vplyvy drobných zmien v štruktúre kódu na rýchlosť simulácie. Konštanta N predstavuje celkový počet inštrukcií *NOP* v programe a symbol X určuje dĺžku sekvencie vykonanú v jednej iterácii.



Obr. 8.1: Výkon nového simulátora oproti pôvodnej implementácii - *Codasip uRISC*



Obr. 8.2: Zrýchlenie simulácie pre jednotlivé modely



Obr. 8.3: Výkon simulátora v závislosti od dĺžky tela cyklu

Zmena X ovplyvňuje počet iterácií tak, aby bol celkový počet vykonaných inštrukcií približne rovnaký – ide o čiastočné rozvinutie tela cyklu (*loop unroll*), techniku často využívanú prekladačmi v rámci optimalizácie.

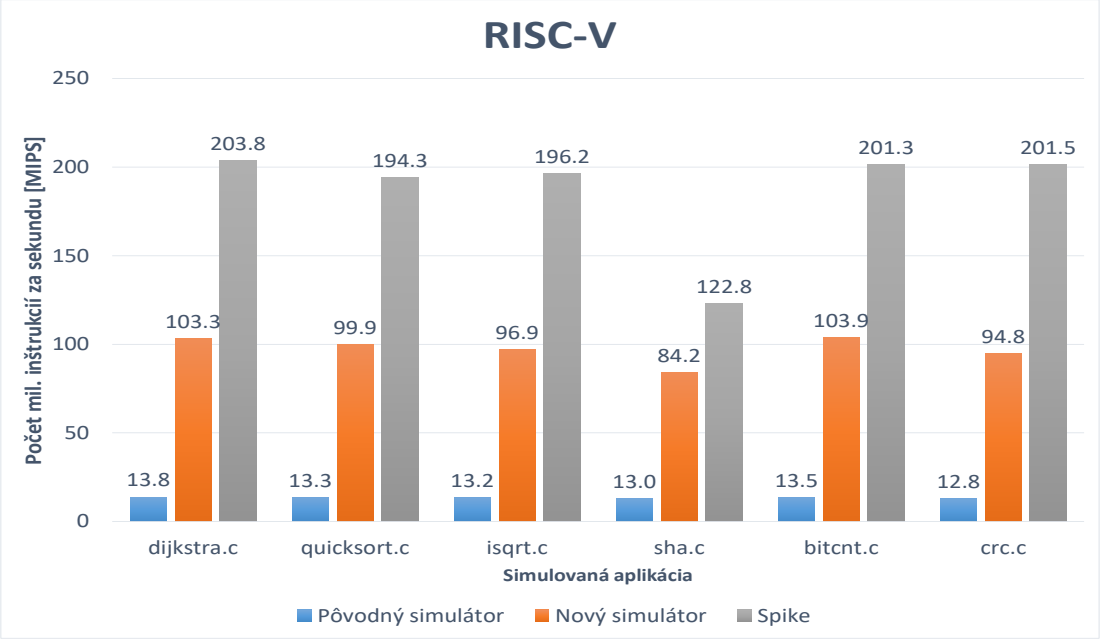
```
int main() {
    for(int i = 0; i < N / X; i++) { NOPS_SEQ(X) // macro }
}
```

Algoritmus 8.1: Testovací program obsahujúci sekvenciu inštrukcií *NOP*

Hodnota N bola po niekoľkých experimentoch stanovená na 1 000 000 000 inštrukcií. Vytvorených bolo niekoľko verzií programu, s rôznymi hodnotami X: 1, 10, 100, 1 000, 2 000. Výsledné namerané hodnoty výkonu simulátorov (pôvodného aj nového) vygenerovaných pre model *Codasip uRISC* sú zhrnuté v grafe 8.3.

Z obrázka vyplýva, že v súlade s očakávaniami, nový simulátor zvyšuje výkon pri interpretácii slučiek – platí, že čím dlhšie je telo cyklu, tým viac inštrukcií v adresnej *cache* je možné vykonať sekvenčne (bez skoku) za sebou. Preto výkon s počtom rozbalených iterácií stúpa.

Zlomovým bodom je úroveň 1 000 inštrukcií v jednej iterácii. Keďže počet položiek v adresnej *cache* je fixne stanovený na 1 024, vykonávanie dlhších cyklov, ktoré sa do *cache* nevojdú, spôsobuje výpadky v každej iterácii, v dôsledku ktorých je nutné opakovane pristupovať do simulovanej pamäte za účelom prečítania reálneho inštrukčného slova. Preto je pri variante s dĺžkou iterácie 2 000 inštrukcií zaznamenaný pokles na 68 *MIPS*. Zatiaľčo pri verzii bez rozvinutia cyklu ($X = 1$), kde každú iteráciu tvorí jediná inštrukcia *NOP*, zohráva úlohu len predčasné ukončenie rozbalenej hlavnej simulačnej slučky, v prípade príliš dlhého tela cyklu (X = 2 000), ktoré sa celé nevojde do vyhľadávacej tabuľky, je hlavnou príčinou spomalenia úplný výpadok z tabuľky a z toho vyplývajúca nutnosť pamäťového prístupu a dekódovania.



Obr. 8.4: Porovnanie výkonu simulátorov (pôvodný, nový a *Spike*)

8.3 Porovnanie so simulátorom Spike

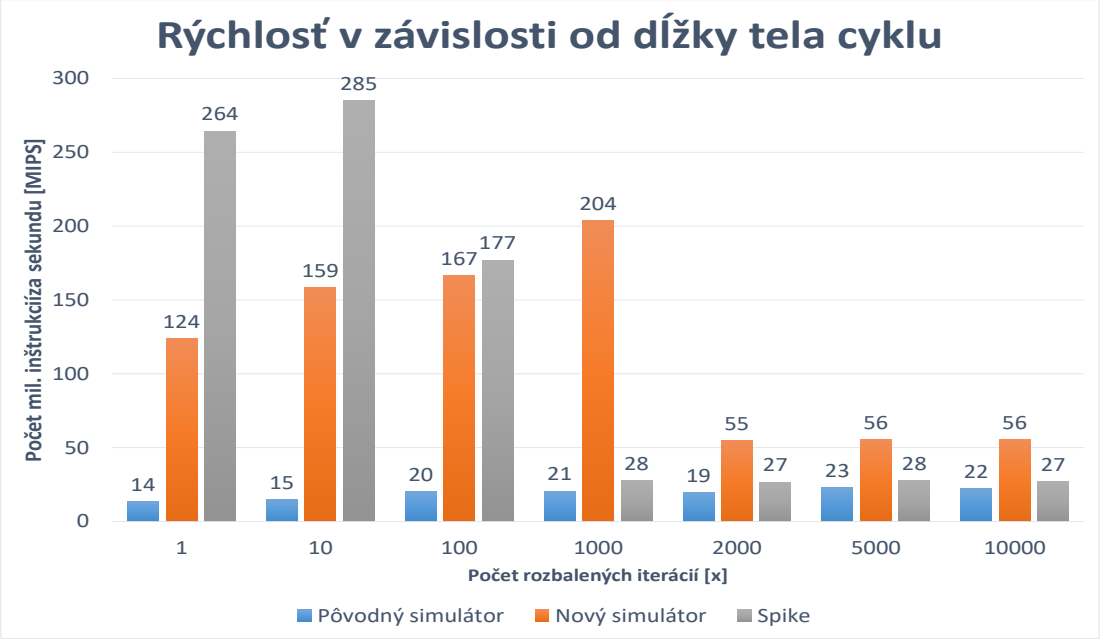
V rámci zhodnotenia úspešnosti vzhľadom ku konkurenčným riešeniam, bol výkon nového simulátora porovnaný so simulátorom *Spike*. Ako bolo prezentované v kapitole 4, ide o referenčný simulátor inštrukčnej sady *RISC-V*. Preto bol pre porovnanie výkonu zvolený model *Codix Berkelium*, ktorého inštrukčná sada taktiež odpovedá špecifikácii *RISC-V*.

Binárne formáty programov preložených pre simulátor *Spike* a simulátor vygenerovaný pre *Codix Berkelium* nie sú úplne totožné, preto boli pri preklade zdrojových kódov použité separátne kompilátory. Keďže interpretovaný kód bol pre oba simulátory rôzny, porovnávané sú znova doby trvaní simulácií vzhľadom k počtu inštrukcií v jednotkách *MIPS*.

Simulátor *Spike* štandardne neumožňuje počet vykonaných inštrukcií zistiť. Preto bolo potrebné upraviť jeho zdrojový kód tak, aby na konci simulácie túto informáciu vypísal, inak by nebolo možné hodnotu *MIPS* vypočítať.

Simulované boli rovnaké aplikácie ako pri predchádzajúcich meraniach výkonu. Graf 8.4 znázorňuje porovnanie rýchlostí pôvodného a nového simulátora, spolu s rýchlosťou konkurenčného simulátora *Spike*. Z výsledkov je vidieť, že nový simulátor je stále 2-krát pomalší ako *Spike*, avšak ide o značné zrýchlenie oproti pôvodnej implementácii, ktorá dosahovala len 6–10% výkonu. Obzvlášť úspešný bol nový simulátor v prípade aplikácie *sha.c*, kde sa rýchlosťou dostal až na úroveň 70% konkurenčného riešenia.

Graf 8.5 znázorňuje porovnanie vplyvu dĺžky tela cyklu na rýchlosť simulácie pre 3 porovnávané simulátory inštrukčnej sady *RISC-V*. Simulovanými aplikáciami sú znova varianty programu *nops.c* s rôznou dĺžkou tela slučky. Kým rýchlosť najpomalšieho *IA* simulátora nie je štruktúrou simulovaného kódu nijak významne ovplyvnená, zaujímavé je porovnanie novej implementácie a simulátora *Spike*. Rýchly simulátor vygenerovaný pre model *Codix Berkelium* vykazuje podobné správanie, ako to bolo pre *Codasip uRISC* (obr. 8.3). Jeho rýchlosť pri rozbaľovaní tela cyklu stúpa, až kým nie je dosiahnutý limit daný veľkos-



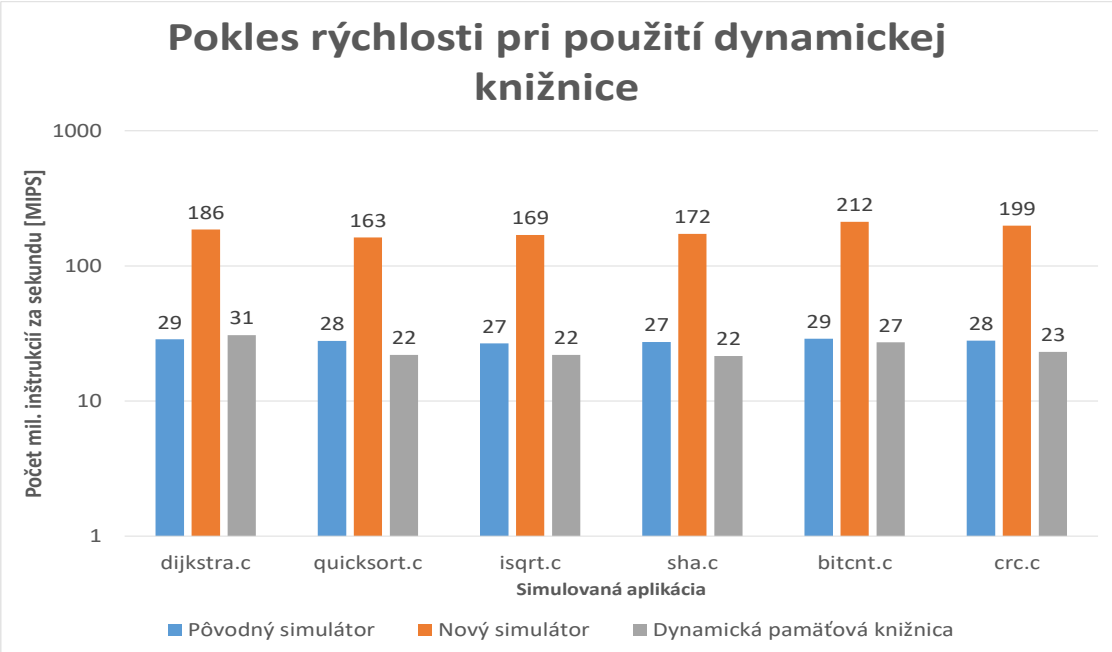
Obr. 8.5: Vplyv dĺžky tela cyklu na rýchlosť simulátorov (pôvodný, nový a *Spike*)

ťou adresnej *cache* (1024). V prípade simulátora *Spike* nastáva zlom o niečo skôr – pri cykle s dĺžkou tela 100 inštrukcií výkon poklesne približne na úroveň nového optimalizovaného IA simulátora a pri 1000 inštrukciách sa rýchlosťou približuje pôvodnému pomalému *IA* simulátoru, pričom je 7-krát pomalší ako nová implementácia. Pri ďalšom predĺžení tela simulovanej slučky na 2000 inštrukcií poklesne aj výkon nového simulátora, avšak napriek tomu je 2-krát rýchlejší ako *Spike*. V tomto stave sa situácia ustáli, následné predlžovanie tela cyklu už nemá na výkon simulátorov žiadny citeľný vplyv.

Z výsledkov simulácií aplikácie *nops.c* nie je možné vyvodiť konkrétne závery o absolútnej rýchlosti simulátorov v praxi, inštrukcia *NOP* totiž nevykonáva žiadnu reálnu operáciu (hoci veľmi podobné výsledky meraní vznikli pri nahradení prázdnej inštrukcie napríklad aritmetickou inštrukciou *ADD*). Hlavným zmyslom testu bolo identifikovať vplyv štruktúry simulovaného kódu na réžiu spojenú s dekódovaním inštrukcie a porovnať rýchlosť simulácie pri vykonávaní dlhej sekvencie inštrukcií bez skokov, oproti vykonávaniu krátkeho cyklu.

8.4 Vplyv použitia dynamickej knižnice

Jednou z požiadaviek na nový simulátor bola možnosť pripojenia externého pamäťového systému. To je možné prostredníctvom dodania externej dynamickej knižnice definujúcej funkcie pre prístup do pamäti (*read*, *write* a *load*). Dopad na rýchlosť simulácie sa však ukázal byť enormný, viď graf 8.6. Merania boli uskutočnené na modeli *Codasip uRISC*, pričom funkcie v pamäťovej knižnici boli definované absolútne minimalisticky. Pamäť bola implementovaná ako pole 32-bitových slov, pričom sa implicitne počítalo so zarovnanou adresou a manipulovaním výlučne s celými slovami bez akejkoľvek kontroly. Napriek tomu, celkový výkon simulátora klesol pod úroveň pôvodnej pomalej implementácie, teda 6-8 násobne. Celé toto spomalenie sa dá pripísať réžii súvisiacej s volaním knižničných funkcií,



Obr. 8.6: Vplyv použitia externej pamäťovej knižnice

ktoré prebieha nepriamo cez tabuľku. Bohužiaľ, túto réžiu nie je možné eliminovať a je potrebné s ňou v prípade nutnosti použitia externej pamäte počítať.

Kapitola 9

Návrhy na zlepšenie

V predchádzajúcom texte bol popísaný návrh nového simulátora a jeho implementácia v podobe funkčného programu. Následne boli prezentované experimenty zamerané na zmeranie a porovnanie výkonu za účelom analýzy a zhodnotenia dosiahnutých výsledkov.

Táto kapitola vychádza zo zistených nedostatkov a zaoberá sa návrhmi na zlepšenie do budúcna.

Dátová cache

Nasadením rýchlej vyhľadávacej tabuľky inštrukcií podľa adresy sa podarilo radikálne znížiť počet prístupov do simulovanej pamäte, ktoré spôsobovali výrazné spomaľovanie simulácie. Problémom ostávajú inštrukcie pracujúce s pamäťou, typicky *load* a *store* (prípadne iné komplexnejšie operácie).

Pre programy, ktoré do pamäte pristupujú často, by mohlo byť výhodné použiť koncept krátkej vyhľadávacej *cache* aj v prípade dát. Medzi hlavné dôvody spomalenia patrí nutnosť kontroly adresy, prípadne požadovanej šírky slova a následný bitový posun dát pri nezarovnanom prístupe. Ak by tabuľka obsahovala dáta už v predspracovanej podobe so správnym zarovnaním, bolo by možné zároveň preskočiť kontroly parametrov a rovno vrátiť výsledok.

Je však otázne, či by réžia súvisiaca s udržovaním konzistencie s hlavnou pamäťou nepreválila nad potenciálnym zrýchlením, a teda či by zavedenie takejto tabuľky malo pozitívny efekt na rýchlosť simulácie. Koncept by bolo potrebné overiť konkrétnymi meraniami výkonu.

Dynamická kompilácia

Na to aby sa výkon simulátora mohol priblížiť rýchlym emulátorom ako je *QEMU* alebo *Imperas ISS* je potrebné pristúpiť k dynamickej kompilácii základných blokov – princíp bol popísaný v kapitole 3.

Existujúca tabuľka adries, ktorá slúži na uchovávanie naposledy vykonaných inštrukcií podľa adresy môže byť využitá pri identifikácii často vykonávaných úsekov kódu, teda blokov, ktorých preklad do natívneho kódu by mohol mať veľký vplyv na rýchlosť celej simulácie.

Kapitola 10

Záver

Táto práca je venovaná téme aplikačne špecifických procesorov a procesu ich návrhu so zameraním na simuláciu ako základný používaný prostriedok. Cieľom bolo analyzovať existujúce prístupy a riešenia, a následne využiť získané poznatky pri návrhu nového simulátora použiteľného v prostredí Codasip Studio.

Text práce rozoberá princípy fungovania, výhody a nevýhody interpretačných aj kompilačných metód simulácie a popisuje vybrané existujúce nástroje využívajúce tieto techniky. Získané informácie sú v kombinácii so znalosťou fungovania existujúceho IA simulátora premietnuté do návrhu nového optimálnejšieho riešenia.

Pri návrhu inštrukčnej sady aplikačne špecifického procesora je potrebné za účelom porovnania a vyhodnotenia vplyvov aplikovaných zmien v čo najkratšom čase simulovať veľké množstvo testovacieho kódu, či už v podobe krátkych vygenerovaných sekvencií inštrukcií alebo komplexných aplikácií, resp. celého operačného systému. Preto je prioritnou požiadavkou na nový simulátor maximalizácia rýchlosti. Identifikácia hlavných problémov v súčasnej implementácii viedla k návrhu kľúčových zmien a optimalizácií, ktoré sa následne stali súčasťou realizácie nového simulačného nástroja.

Výsledkom práce je koncept rýchleho čiastočne prekladaného simulátora aplikačne špecifických procesorov popísaných v jazyku *CodAL*, a implementácia tohto konceptu formou modulárneho programu v jazyku *C++*. Súčasťou výstupu je séria experimentov, ktorých výsledky potvrdzujú úspešnosť nového simulátora z hľadiska dosiahnutého výkonu.

V porovnaní s pôvodným simulátorom používaným v rámci *Codasip Studia* a pracujúcim s presnosťou na inštrukcie, sa podarilo dosiahnuť merateľné zvýšenie rýchlosti simulácie, a to približne 5 až 8-krát v prípade modelov s architektúrou *RISC* a 3 až 3,5-krát pri modeli s architektúrou typu *VLIW*.

Oproti konkurenčnému existujúcemu simulátoru *Spike* sa nový simulátor ukázal byť podľa výsledkov testovaní na niekoľkých krátkych aplikáciách približne o polovicu pomalší, hoci v niektorých krajných prípadoch bola rýchlosť rovnaká alebo dokonca vyššia.

Narozdiel od simulátora *Spike*, ktorý je pevne zviazaný s architektúrou *RISC-V*, má implementácia nového nástroja podobu generátora, ktorý je schopný na základe popisu procesorového modelu automatizovane vytvoriť špecifický simulátor na mieru danej inštrukčnej sady, čo zvyšuje použiteľnosť v praxi.

Výzvou do budúcna je prechod na dynamickú kompiláciu častí simulovanej aplikácie za behu, vďaka čomu by sa výkon mohol priblížiť k rýchlym emulátorom ako je *QEMU* alebo *Imperas ISS*. To je už však nad rámec rozsahu tejto práce.

Literatúra

[1] BOSE, P.: EIC's Message: General-purpose versus application-specific processors. *IEEE Micro*, ročník 24, č. 3, May 2004: s. 5–5, ISSN 0272-1732, doi:10.1109/MM.2004.8.

[2] BURSKY, D.: New Processor Core Options Try Some ARM Wrestling. 2013, [Online; navštívené 12.12.2017].
URL http://www.chipdesignmag.com/bursky/?p=113

[3] CLARK, M.: QEMU with RISC-V. [Online; navštívené 10.11.2017].
URL https://github.com/riscv/riscv-qemu

[4] CLARK, M.: RISC-V QEMU Part 1: Privileged ISA v1.10, HiFive1 and VirtIO.
URL https://www.sifive.com/blog/2017/12/20/risc-v-qemu-part-1-privileged-isa-hifive1-virtio/

[5] Codasip Ltd.: *CodAL Language Reference Manual*. 2017.

[6] CROSTHWAITE, P.; WILLIAMS, J.; SUTTON, P.: A unified emulation/simulation environment for reconfigurable system-on-chip development. In *2011 International Conference on Field-Programmable Technology*, Dec 2011, s. 1–8, doi:10.1109/FPT.2011.6132690.

[7] FISHER, J. A.: *Embedded computing: a VLIW approach to architecture, compilers and tools*. Boston: Morgan Kaufmann Publishers, 2005, iSBN 1-55860-766-8.

[8] HAHN, T. T.; Stotzer, E.; Sule, D.; aj.: *Compilation Strategies for Reducing Code Size on a VLIW Processor with Variable Length Instructions*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, ISBN 978-3-540-77560-7, s. 147–160, doi:10.1007/978-3-540-77560-7_11.
URL https://doi.org/10.1007/978-3-540-77560-7_11

[9] HOFFMANN, A.; KOGEL, T.; NOHL, A.; aj.: A novel methodology for the design of application-specific instruction-set processors (ASIPs) using a machine description language. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, ročník 20, č. 11, Nov 2001: s. 1338–1354, ISSN 0278-0070, doi:10.1109/43.959863.

[10] HOFFMANN, A.; MEYR, H.; LEUPERS, R.: *Architecture Exploration for Embedded Processors with LISA*. Springer, 2010, ISBN 1441953345.

[11] HUANG, P.: Documentation for RISC-V Spike. [Online; navštívené 12.11.2017].
URL https://github.com/poweihuang17/Documentation_Spike

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

[13] Imperas Software Ltd.: ISS - The Imperas Instruction Set Simulator.
URL http://www.imperas.com/iss-the-imperas-instruction-set-simulator

[14] Imperas Software Ltd.: Welcome Page | Open Virtual Platforms. [Online; navštívené 22.11.2017].
URL http://www.ovpworld.org/

[15] LIU, D.: *Application Specific Instruction Set DSP Processors*. Boston, MA: Springer US, 2010, ISBN 978-1-4419-6345-1, s. 415–447, doi:10.1007/978-1-4419-6345-1_16.
URL https://doi.org/10.1007/978-1-4419-6345-1_16

[16] PATTERSON, D. A.; HENNESY, J. L.: *Computer Organization and Design RISC-V Edition: The Hardware Software Interface (The Morgan Kaufmann Series in Computer Architecture and Design)*. Morgan Kaufmann, 2017, ISBN 0128122757.

[17] PERINGER, P.: *Modelování a simulace IMS*, 2012.

[18] PŘIKRYL, Z.: *Advanced Methods of Microprocessor Simulation*. Dizertační práce, Vysoké Učení Technické v Brně, 2012.

[19] PŘIKRYL, Z.; KŘOUSTEK, J.; HRUŠKA, T.; aj.: Fast just-in-time translated simulator for ASIP design. In *14th IEEE International Symposium on Design and Diagnostics of Electronic Circuits and Systems*, April 2011, s. 279–282, doi:10.1109/DDECS.2011.5783094.

[20] QEMU: QEMU version 2.10.50 User Documentation. [Online; navštívené 10.11.2017].
URL https://qemu.weilnetz.de/doc/qemu-doc.html

[21] QEMU: QEMU Project. 2017, [Online; navštívené 10.11.2017].
URL https://www.qemu.org/

[22] RICHTARIK, P.: *Návrh a implementace profileru pro aplikačně specifické procesory*. Bakalářská práce, Vysoké učení technické v Brně, Fakulta informačních technologií, 2016.
URL http://www.fit.vutbr.cz/study/DP/BP.php?id=18737

[23] RISC-V Foundation: About the RISC-V Foundation. [Online; navštívené 13.11.2017].
URL https://riscv.org/risc-v-foundation/

[24] RISC-V Foundation: About the RISC-V ISA. [Online; navštívené 13.11.2017].
URL https://riscv.org/risc-v-isa/

[25] SPINK, T.; WAGSTAFF, H.; FRANKE, B.: Efficient Asynchronous Interrupt Handling in a Full-system Instruction Set Simulator. *SIGPLAN Not.*, ročník 51, č. 5, Červen 2016: s. 1–10, ISSN 0362-1340, doi:10.1145/2980930.2907953.
URL http://doi.acm.org.ezproxy.lib.vutbr.cz/10.1145/2980930.2907953

[26] TEWS, H.: Verifying Duff's device.

[27] TIS Committee: Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification Version 1.2. [Online; navštívené 25.11.2017].
URL https://www.uclibc.org/docs/elf.pdf

[28] WATERMAN, A.; LEE, Y.: Spike. [Online; navštívené 12.11.2017].
URL https://github.com/riscv/riscv-isa-sim

[29] WATERMAN, A.; LEE, Y.; PATTERSON, D. A.: The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.0. Technická Zpráva UCB/EECS-2014-54, EECS Department, University of California, Berkeley, May 2014.
URL http://www2.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-54.html