VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

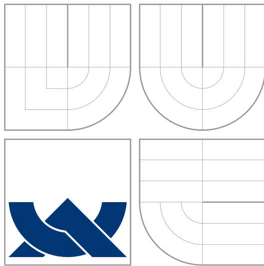
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

PŘEKLADAČ PRO PLATFORMU EDKDSP

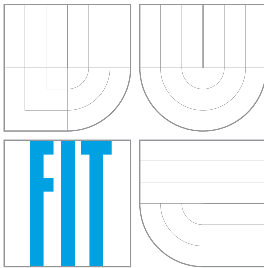
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — RÓBERT BARUČÁK
AUTHOR

BRNO 2012



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

PŘEKLADAČ PRO PLATFORMU EDKDSP

COMPILER FOR EDKDSP PLATFORM

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — RÓBERT BARUČÁK
AUTHOR

VEDOUCÍ PRÁCE — Ing. KAREL MASAŘÍK, Ph.D.
SUPERVISOR

BRNO 2012

Abstrakt

Cílem bakalářské práce bylo vytvoření překladového systému pro platformu EdkDSP. Prezentovány jsou dva odlišné přístupy ke konstrukci překladového systému určeného pro multiprocesorovou platformu. Práce je založena na překladačové infrastruktuře LLVM. Výsledkem jsou dvě funkční verze překladového systému, které generují kód využívající všechny hardwarové prostředky poskytované cílovou platformou. Vytvořená řešení mají sadu omezení, která jsou diskutována v textu práce.

Abstract

Goal of this bachelor's thesis was to create a compiler system for EdkDSP platform. Two different approaches to construction of compiler system for multiprocessor platform are presented. Compiler is based on LLVM compiler infrastructure. As a result, two versions of compiler system utilising hardware capabilities of EdkDSP were created. Developed solutions have a few constraints which are discussed in this paper.

Klíčová slova

EdkDSP, multiprocesorové systémy, překladač, LLVM, Clang, backend, GCC, akcelerace, MicroBlaze, PicoBlaze

Keywords

EdkDSP, multiprocessor systems, compiler, LLVM, Clang, backend, GCC, acceleration, MicroBlaze, PicoBlaze

Citace

Róbert Baručák: Překladač pro platformu EdkDSP, bakalářská práce, Brno, FIT VUT v Brně, 2012

Překladač pro platformu EdkDSP

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana doktora Masaříka

....
Róbert Baručák
15. mája 2012

Poděkování

Ďakujem pánovi inžinierovi Husárovi a pánovi doktorovi Masaříkovi za konzultácie a odbornú pomoc. Ďalej chcem poďakovať celému týmu Lissom za úvod do problematiky a pomoc.

© Róbert Baručák, 2012.
Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Obsah

1 Úvod 3

2 Akcelerátory a platforma EdkDSP 4
2.1 Základné delenia akcelerátorov . . . 4
2.1.1 Delenia podľa spojenia s procesorom . . . 4
2.1.2 Delenie podľa spôsobu implementácie akcelerovanej funkcie . . . 5
2.2 Prehľad paralelných architektúr . . . 5
2.3 Platforma EdkDSP . . . 6
2.3.1 Basic Computing Element . . . 6
2.3.2 Rozdiely medzi jednotlivými verziami platformy EdkDSP . . . 6
2.3.3 Parametre platformy . . . 8
2.3.4 WAL knižnica . . . 8
2.3.5 PBBCELIB . . . 9
2.4 Preklad pre EdkDSP . . . 9
2.4.1 Pbcc . . . 10
2.5 μClinux . . . 11
2.5.1 BusyBox . . . 11
2.6 Vývojová doska . . . 12

3 Architektúra prekladača LLVM 13
3.1 Vnútorná reprezentácia kódu . . . 13
3.1.1 Trojadresný kód . . . 14
3.1.2 SSA forma . . . 14
3.2 LLVM priechody . . . 15
3.3 Generovanie kódu pomocou LLVM backendu . . . 16
3.3.1 Selekcia inštrukcií . . . 16
3.3.2 Plánovanie a formátovanie . . . 16
3.3.3 Alokácia registrov . . . 16
3.3.4 Vkladanie prológu a epilógu . . . 17
3.3.5 Emitovanie kódu . . . 17

4 Návrh a implementácia 18
4.1 Návrh prekladového systému . . . 18
4.2 Frontend . . . 19
4.3 Akceleračný priechod LLVM . . . 20
4.3.1 Registrácia priechodu . . . 20
4.3.2 Vlastnosti akcelerovateľného cyklu . . . 21
4.3.3 Extrakcia informácií o cykle . . . 22

4.3.4 Generovanie firmwaru pre BCE 23
4.4 Manažér akcelerácie 28
4.4.1 Integrácia manažéra akcelerácie do akcelerovaného programu 29
4.5 Prekladový skript 30

5 Výsledky 31
5.1 Preklad vzorového programu 31
5.2 Porovnanie výkonu akcelerovanej a neakcelerovanej aplikácie 33

6 Záver 35

A Inštalácia prekladového systému 38
A.1 Inštalácia systému LLVM a akceleračného priechodu 38
A.2 Inštalácia ostatných súčastí systému 39

Kapitola 1

Úvod

Široké rozšírenie multiprocesorových systémov do praxe prináša zvýšené nároky na prekladače a programátorov. Multiprocesorové systémy sú sľubným spôsobom, ako urýchliť a zefektívniť vykonávanie programov pomocou paralelizácie. Rôzne platformy ponúkajú rozličné spôsoby, ako akcelerovať výpočty. V prípade platformy EdkDSP existuje špecializovaná knižnica funkcií, ktorá zabezpečuje paralelné vykonávanie kódu na špecifických procesoroch. Implementácia programu tak, aby správne a efektívne využíval prostriedky ponúkané platformou, je netriviálna úloha.

Cieľom bakalárskej práce je navrhnutie a vytvorenie špecializovaného akceleračného prekladača jazyka C pre túto platformu. Prekladač transformuje bežný kód v jazyku C tak aby v maximálnej možnej miere využíval prostriedky poskytnuté platformou bez nutnosti zásahu programátora. Výsledkom je ušetrenie času potrebného na napísanie a urýchlenie programu. Umožňuje programátorovi sústrediť sa na implementovaný algoritmus, nie na špecifiká akcelerácie a danej platformy.

Konštruovať celý prekladač od základu by bolo zbytočné, ako základ je využitý prekladový systém LLVM vo verziách 2.8 a 3.0. Použité sú aj mnohé optimalizácie, ktoré ponúka systém LLVM. Funkcionalita transformácie je riešená ako špecializovaný priechod prekladu pomocou LLVM.

Práca je súčasťou medzinárodného projektu SMECY [2] (Smart Multicore Embedded Systems), bola vytvorená v rámci výskumného tímu Lissom [3]. Využíva sa platforma navrhnutá v UTIA (Ústav teorie informace a automatizace AV ČR). EdkDSP je heterogénna multiprocesorová platforma. Počas vývoja prekladača prešlo EdkDSP významnými zmenami, ktoré boli dôvodom vývoja dvoch odlišných verzií prekladača.

Cieľom práce je oboznámenie čitateľa s problematikou vývoja prekladového systému pre netradičnú architektúru. Rozpracované sú dva mierne odlišné prístupy k riešeniu problému.

Kapitola 2

Akcelerátory a platforma EdkDSP

Podľa Dake Liu [15] je akcelerátor hardwarový modul pripojený k procesoru, ktorý zvyšuje výkon alebo zväčšuje funkčné možnosti procesoru. Isté funkcie sú vykonávané v akcelerátore a nie v procesore samotnom. Akcelerátorom môže byť čokoľvek od jednoduchého hardwaru ovládaného pomocou nastavovania portov až po procesor, ktorý obsahuje vlastnú inštrukčnú sadu. Existuje niekoľko prípadov, keď je použitie akcelerátora výhodné.

Ak procesor nemá dostatočný výkon pre spracovanie niekotorých výpočtovo náročných operácií, akcelerátor prevezme ich vykonávanie. Ďalším vhodným príkladom je nahradenie niektorých operácií, ktoré absentujú v inštrukčnej sade procesoru. Napríklad ak v procesore chýbajú inštrukcie pre prácu s číslami s plávajúcou desatinnou čiarkou, tieto môžu byť vykonané akcelerátorom.

Využitie akcelerátorov je výhodné aj pri snahe o významné zníženie spotreby. Ak je akcelerátor schopný prebrať značnú časť výpočtov, môže klesnúť pracovná frekvencia a teda aj privádzané napätie.

Využitie akcelerácie je časté pri procesoroch, ktoré sú prispôsobené pre konkrétnu aplikáciu. Označujú sa skratkou ASIP (Application-specific instruction-set processor) [15]. Funkcie, ktoré by mali byť akcelerované sú známe a nepodliehajú prílišným zmenám. Z podobných príčin je využitie akcelerátorov obľúbené aj pri DSP aplikáciách. Akcelerujú sa operácie nad vektormi alebo rôzne algoritmy pre kódovanie a dekódovanie dát.

Výber konkrétnych funkcií vhodných pre akceleráciu sa riadi niekoľkými základnými pravidlami. Vhodné sú funkcie, ktorých súčasťou sú iterácie a výpočtovo veľmi náročné funkcie. Aplikáciou často používané funkcie a také, ktorých akceleráciou sa dá výrazne redukovať veľkosť vykonávaného kódu.

2.1 Základné delenia akcelerátorov

2.1.1 Delenia podľa spojenia s procesorom

Akcelerátory sa delia na *voľne* a *pevne* spojené. Voľne spojený akcelerátor je funkčný modul umiestnený za všeobecným rozhraním periférneho modulu. Komunikácia medzi procesorom a akcelerátorom prebieha zapisovaním a čítaním z periférneho registra. Takto spojený akcelerátor môže byť jednoducho pridaný k procesoru bez nutnosti zmeny inštrukčnej sady, pretože komunikácia a ovládanie môže byť obstaraná klasickými vstupno/výstupnými inštrukciami. Nevýhodou tohto prístupu je obmedzená rýchlosť komunikácie s akcelerátorom, vznik omeškania a veľké náklady na réžiu komunikácie.

Naproti tomu pevne spojené akcelerátory sú ovládané z procesoru na úrovni špeciálnych inštrukcií inštrukčnej sady. Môžu byť výrazne rýchlejšie ako voľne spojené, ale poskytujú minimálnu flexibilitu a musia byť zapracované už do návrhu procesora samotného.

2.1.2 Delenie podľa spôsobu implementácie akcelerovanej funkcie

Funkcia akcelerátora môže byť implementovaná ako jediná inštrukcia alebo ako súbor inštrukcií. Väčšie množstvo inštrukcií prináša vyššiu flexibilitu avšak za cenu zvýšenia réžie komunikácie. Ideálna implementácia nevyžaduje žiadnu komunikáciu okrem spúšťania a ukončovania činnosti akcelerátora.

1. Akcelerátory s jedinou inštrukciou sú vhodné pre jednoduché algoritmy. Typickým príkladom je počítanie odmocniny alebo delička. V týchto prípadoch stačí jedna inštrukcia. Tá nesie informácie ako konfiguračné dáta pre akcelerátor, adresy operandov alebo priamo ich hodnoty.

2. Funkcia akcelerátora je implementovaná ako viacero inštrukcií. Nejedná sa o procesor, pretože každá inštrukcia je spúšťaná hlavným procesorom. Inštrukcie sú stále dekódované hlavným procesorom.

   Využitie je pri komplexných výpočtových akcelerátoroch multiplexujúcich viacero operácií. Využívajú sa pri konvolúcii, konvolúcii s automatickou koreláciou atď.

3. K hlavnému procesoru môže byť pripojených viacero pomocných procesorov. Hlavný procesor v tom prípade inicializuje vykonanie úlohy odoslaním identifikátora úlohy a inicializačných dát pomocnému procesoru. Vďaka programovateľnosti pomocného procesora je takýto akcelerátor značne flexibilný.

   Významne sa zvyšujú náklady na réžiu komunikácie a tak isto cena hardwaru. Kontrola hlavného procesoru je slabšia ako pri iných typoch akcelerátorov a doba vykonávania úlohy nemusí byť dopredu známa. Problémové môžu byť aj dátové závislosti v akcelerovaných výpočtoch. Typickým príkladom je architektúra CELL od Sony Toshiba IBM. EdkDSP sa taktiež zaraďuje do tejto kategórie.

2.2 Prehľad paralelných architektúr

Existuje niekoľko základných typov paralelných architektúr, ktoré sa v súčasnosti využívajú. Špecifiká každého typu ich predurčujú pre rôzne typy aplikácií, ale zároveň aj využívanie rôznych prekladových systémov.

1. Škálovateľné procesory a akcelerátory umožňujú pridávanie a odoberanie hardwarových modulov a procesorov. Zmeny by mali mať len limitovanú cenu z hľadiska návrhu architektúry.

2. Väčšina DSP procesorov využíva ILP (Instruction level parallelism). Spracovávajú viacero mikrooperácií paralelne v jednom programe. Sem spadajú rôzne VLIW a superskalárne architektúry.

3. Heterogénne multiprocesory sú také systémy, ktoré obsahujú procesory s viacerými inštrukčnými sadami. Najväčšia efektivita ich využitia sa dosahuje, ak rozličné úlohy sú spracovávané špecializovanými procesormi. Pri špecifických aplikáciách dosahujú

vyšší výkon ako homogénne multiprocesorové platformy. Využitie nachádzajú hlavne pri spracovaní signálov, vektorových video procesoroch a pri rozličných kodekoch.

4. Master-slave multiprocesorové architektúry väčšinou pozostávajú z hlavného kontrolného procesoru a väčšieho množstva DSP subsystémov. Pomocné DSP procesory paralelne spracovávajú vektorové a rôzne iteratívne úlohy. Pomocné procesory často bývajú SIMD procesory. Už spomínaná architektúra CELL pozostáva z PowerPC mikrokontroléru a ôsmich identických SIMD pomocných procesorov. Hlavný procesor iba kontroluje pomocné procesory, ktoré vykonávajú operácie. Podobnou architektúrou je aj EdkDSP, ktorá bude podrobnejšie rozobraná.

5. Podstatnou kategóriou sú aj homogénne multiprocosorové systémy, využívané na paralelné spracovanie signálov.

2.3 Platforma EdkDSP

EdkDSP platforma je implementovaná na XILINX FPGA. Postavená je na hlavnom MicroBlaze a skupine rozličných Basic Computing Element (BCE). Určené sú na akceleráciu špecifických typov operácií s plávajúcou desatinnou čiarkou. Všeobecný prehľad informácií o platforme je v dokumente [4].

2.3.1 Basic Computing Element

BCE akcelerujú finálnu aplikáciu pomocou vykonávania postupnosti hardwarovo podporovaných operácií nad lokálnymi vektormi dát s plávajúcou desatinnou čiarkou. Operácie môže BCE spracovávať po skupinách. Dávka môže pozostávať z jedinej operácie, ale všetky operácie v dávke musia byť podporované dataflow jednotkou BCE.

BCE má k dispozícii vždy aspoň tri dvojportové blokové RAM dátové pamäte pre operandy a výsledky operácií. Verzia EdkDSPv2.0 poskytuje štyri takéto pamäte pre každý BCE. Dataflow jednotka zvláda multiplexovanie a prepájanie zreťazených sčítaní a násobení s plávajúcou desatinnou čiarkou. Vytvára sadu hardwarovo akcelerovaných výpočtových reťazcov. Umožňuje čítanie dvoch vstupných operandov a zapísanie jedného výsledku v každom takte BCE hodín. Špecializovaný hardware zodpovedá za inicializačnú fázu, zreťazovanie operácií a za finálne zapísanie výsledku do lokálnej RAM pamäte. Podporovaná je automatická inkrementácia adries a vyhodnocovanie čítača cyklu.

Základná BCE operácia je typicky vykonávaná nad vektormi operandov s plávajúcou desatinnou čiarkou, ktoré sú uložené v dvoch separátnych RAM pamätiach. Výsledný vektor sa zapíše do tretej pamäte.

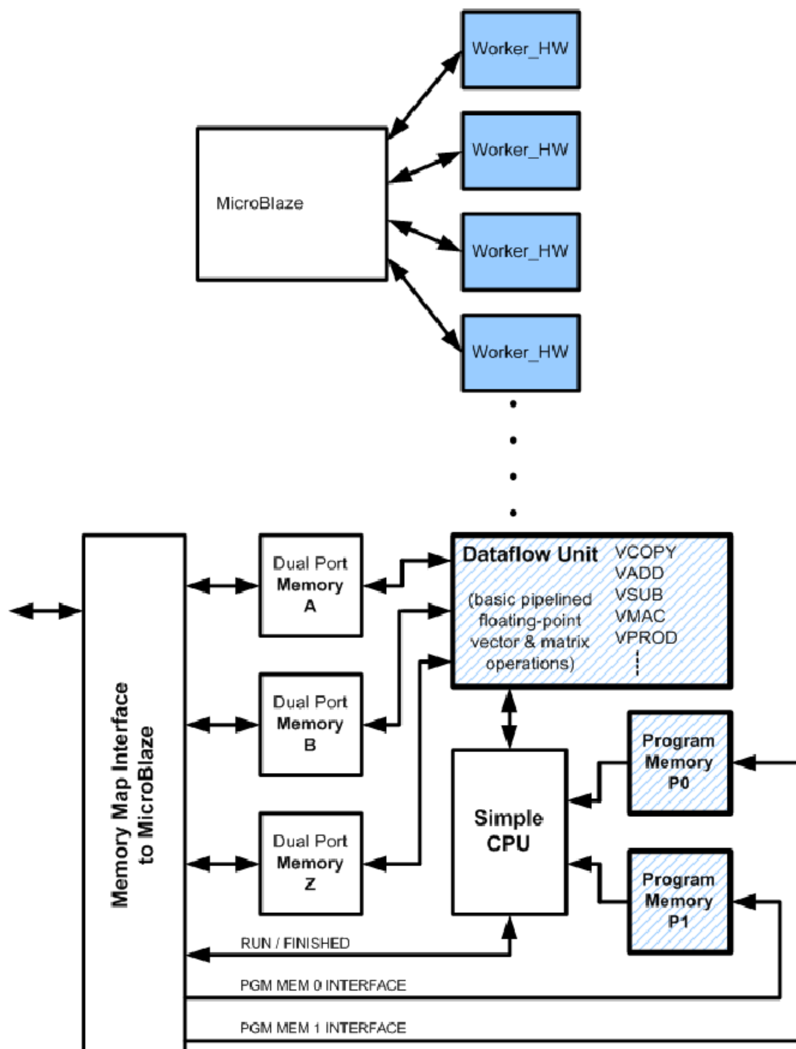
Použitie BCE dataflow jednotiek a preddefinovanej pevnej sady vektorových operácií je umožnené prítomnosťou jednoduchého programovateľného procesoru (PicoBlaze) v každom BCE. Firmware tohto jednoduchého procesoru definuje postupnosť vykonávaných dataflow operácií. Sú vykonávané ako sled hardwarovo akcelerovaných dávkových operácií. Dataflow jednotka je ovládaná jednou VLIW inštrukciou, ktorá je vystavaná firmwarom jednoduchého CPU.

2.3.2 Rozdiely medzi jednotlivými verziami platformy EdkDSP

Súčasťou oboch verzií je hlavný MicroBlaze procesor a sada pomocných BCE jednotiek. Rozdielny je spôsob, akým sa prenášajú dáta jednotlivým BCE. Vo verzii 1.0 (obr. 2.1)

prenos zabezpečuje MicroBlaze procesor, avšak vo verzii 2.0 (obr. 2.2) je k dispozícii DMA jednotka obsluhujúca prenosy dát.

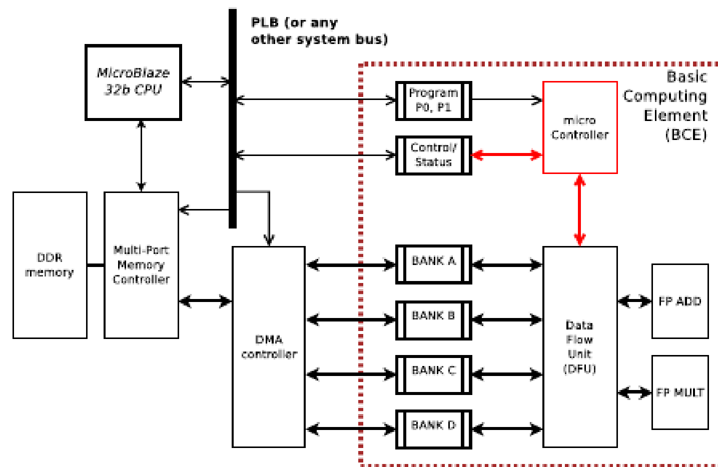
Ďalším podstatným vylepšením je zmena spôsobu práce s vstavanými dvojportovými RAM pamäťami. V prvej verzii sú k dispozícii tri pamäte, každá je rozdelená na štyri banky, ktoré obsahujú 256 slov. Operácie vykonávané dataflow jednotkou majú pevne nastavené vstupné pamäte a výstupnú pamäť. Užívateľsky nastaviteľné sú iba banky z ktorých sa dáta načítajú alebo do ktorých sa ukladajú. Atypické adresovanie a pamäťovo špecifické funkcie spôsobili značné skomplikovanie spôsobu generovania kódu, jeho princípy budú bližšie popísané. Verzia 2.0 priniesla rozšírenie počtu pamäťových bánk na štyri a vylepšenie



Obr. 2.1: Schéma platformy EdkDSPv1.0 [4]

adresovacích metód. Jednotlivé pamäte už nie sú delené na banky a majú priestor na 1024 slov. Jednotlivé operácie dataflow jednotky už nie sú na pevno previazané s pamäťami.

Zmenami prešla aj dataflow jednotka. Kým vo verzii 1.0 sa operácia MAC (Multiply accumulate) dá vykonať zreťazene iba v obmedzenom množstve (13 krát), vo verzii 2.0 takéto obmedzenie neexistuje.



Obr. 2.2: Schéma platformy EdkDSPv2.0

2.3.3 Parametre platformy

Platforma EdkDSP je postavená na XILINX ISE (Integrated Software Enviroment) ver. 11.4 a XILINX EDK (Embedded Development Kit) ver. 11.4. Použitý MicroBlaze procesor je RISC optimalizovaný pre implementáciu v XILINX FPGA. Procesor je konfigurovateľný, zabudovaná verzia obsahuje:

1. tridsaťdva 32b všeobecných registrov,
2. 32b inštrukčné slovo s tromi operandmi a dvoma adresovacími módmi,
3. 32b adresovú zbernicu,
4. 32b verziu rozhrania PLB (Processor local bus) verzie 4.6,
5. jednoduchý synchrónny protokol pre prenosy dát z RAM (LMB),
6. XCL poskytuje rýchle prenosové rozhranie medzi cache pamäťami a externými pamäťovými kontrolérmi. Rozhranie je ovládané pomocným procesorom.

   Každá BCE jednotka je pripojená k MicroBlaze procesoru pomocou zbernice PLB ver. 4.6. MicroBlaze kontroluje prenosy dát medzi externou pamäťou a dvojportovými RAM pamäťami BCE jednotiek. Vo verzii 2.0 k presnosu využíva DMA radič. V budúcnosti budú môcť byť pripojené viaceré verzie BCE jednotiek, vhodné pre rozličné operácie.

2.3.4 WAL knižnica

Worker Abstraction Layer je sada funkcií pre platformu EdkDSP vo forme softwarovej knižnice. WAL musí byť linkovaná k používateľskému programu spúšťanému na MicroBlaze procesore. Informácie o WAL sú dostupné v dokumente [7].

Knižnica bola vytvorená v UTIA pre zjednodušenie a zovšobecneniu prístupu k hardwarovým BCE jednotkám z používateľskej aplikácie. Užívateľ teda pristupuje k hardwarovým akcelerátorom na úrovni programu v jazyku C preloženom pre MicroBlaze procesor.

Použitie WAL knižnice vyžaduje niekoľko krokov:

1. pridať knižnicu do procesu prekladu,
2. pridať hlavičkové súbory knižnice do používateľského programu. Do užívateľskej aplikácie je potrebné dodať aj hlavičkový súbor obsahujúci firmware pre BCE,
3. definovať dátovú štruktúru reprezentujúcu BCE jednotku pomocou makra. Makro definuje ukazatele na zdieľané kontrolné a dátové pamäte. Zabezpečuje pripojenie na konfiguračnú tabuľku definovanú v hardware,
4. inicializovať BCE,
5. použitie BCE zahŕňa nastavenie vykonávaného firmwaru a spustenie požadovanej operácie.

WAL knižnica poskytuje funkcie spojené s inicializáciou, kontrolou BCE a funkcie zamerané na prenos dát medzi MicroBlaze procesorom a BCE. Vo verzii 2.0 ponúka aj funkcie zamerané na ovládanie DMA prenosov.

2.3.5 PBBCELIB

Knižnica PBBCELIB definuje rozhranie medzi PicoBlaze mikrokontrolérom a dataflow jednotkou. Je popísaná v dokumente [8]. Zároveň určuje rozhranie medzi akcelerátorom a hlavným MicroBlaze procesorom zo strany akcelerátora. Poskytuje funkcie na komunikáciu s hlavným CPU a funkcie na parametrizáciu a kontrolu dataflow jednotky. Použitie knižnice sa dá popísať pomocou niekoľkých krokov.

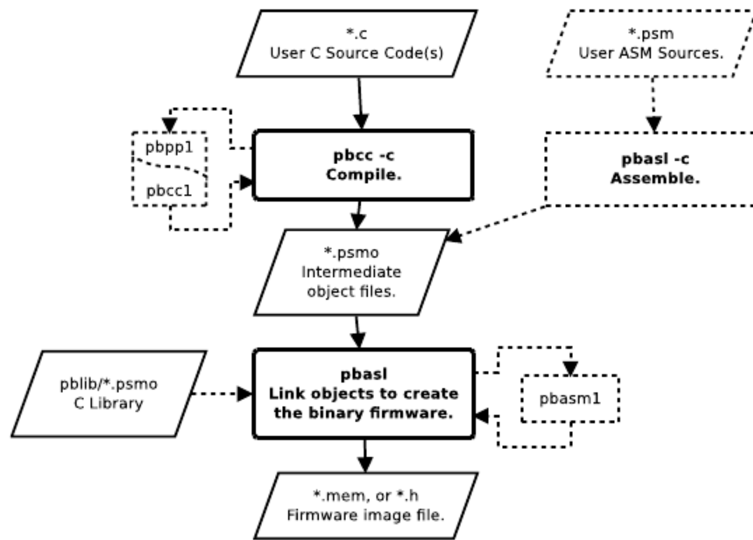
1. Vloženie hlavičkových súborov knižnice do programu pre PicoBlaze.
2. Použitie funkcií v zdrojovom kóde, tie sa dajú rozdeliť na niekoľko skupín. Použité musia byť funkcie na zabezpečovanie komunikácie s hlavným procesorom a prenos dát. BCE jadro musí hlásiť, že je funkčné a vykonáva svoj kód. Poslednou významnou skupinou funkcií je ovládanie dataflow jednotky, nastavovanie operácií, adries operandov a pod.
3. Vzniknutý firmware pre PicoBlaze procesor ovládajúci dataflow jednotku a komunikáciu je násedne preložený a zlinkovaný s programom pre hlavný MicroBlaze procesor, ktorý zabezpečí nahranie firmwaru do BCE jednotky.

Verzia 2.0 značne zjednodušila funkcie určené na komunikáciu s hlavným CPU a zaviedla viaceré možnosti adresovania spracovávaných vektorov.

2.4 Preklad pre EdkDSP

Užívateľský program pre hlavný procesor aj firmwary pre BCE jednotky sú vytvárané v jazyku C. Na preklad pre MicroBlaze procesor sa využíva verzia prekladača GCC určená pre μClinux. K užívateľskému programu sa musí linkovať WAL knižnica. PicoBlaze nepodporujú bežné prekladače jazyka C, preto sa využíva Pbcc prekladač. Informácie o ňom sú dostupné v dokumente [18]. Preložený program má podobu konštantného poľa 1024 celých čísel. Pole tvorí firmware pre PicoBlaze procesor.

2.4.1 Pbcc



Obr. 2.3: Schéma prekladača Pbcc [18]

Pbcc 2.3 je prekladač jazyka C pre PicoBlaze procesor. Preložený program je exportovaný ako hlavičkový súbor, ktorý sa vkladá do programu pre hlavný procesor. Pbcc má viaceré obmedzenia, ktoré neumožňujú využiť plný potenciál jazyka C. V niektorých prípadoch spôsobuje zvláštne správanie. Avšak prekladače jazyka vyššieho ako assembler pre procesor PicoBlaze sú raritné. Pre ilustráciu sú uvedené niektoré obmedzenia dokazujúce, prečo prekladaný program (ergo firmware pre PicoBlaze) v C môže byť len triviálny.

- Použiteľné sú iba základné dátové typy. Medzi ne patria *void*, *char* (8-bit), *int* (16-bit). Podporované sú aj neznamienkové varianty týchto typov, avšak žiadne typy s plávajúcou desatinnou čiarkou.
- Jednorozmerné polia a ukazatele sa dajú používať, rovnako ako ukazateľ na pole. Avšak pole ukazateľov už spadá mimo možnosti Pbcc.
- Nie sú podporované kľúčové slová ako *typedef*, *struct*, *union*, *enum*. Nie sú nutné, nakoľko absentuje podpora užívateľských dátových typov.
- Ternárny operátor ani *sizeof* nie sú implementované.
- Podporované sú iba lokálne premenné alokované na zásobníku. Globálne ani statické premenné sa nedajú využiť. Z uvedených obmedzení vyplýva, že program v PicoBlaze kontrolnom procesore je redukovaný iba na prostriedok slúžiaci pre kontrolu činnosti dataflow jednotky, spravovanie BCE a komunikáciu s hlavným MicroBlaze procesorom.

2.5 μClinux

Programy určené pre akceleráciu na EdkDSP sú spúšťané na verzii μClinuxu od firmy Petalogix. Operačný systém je pužitý kvôli podpore súborového systému a ethernetovej pripojiteľnosti zariadenia k počítaču. Nezanedbateľnou výhodou je podpora vlákien. Vo finálnej verzii, po úspešnej integrácii s ďalšími súčasne riešenými projektmi, bude využívaná paralelizácia pomocou vlákien.

μClinux je operačný systém vhodný pre širokú škálu embedded zariadený odvodený od Linux kernelu 2.6 (pôvodná verzia od 2.0). Využíva sa vo veľkom množstve zariadení od bezpečnostných kamier až po DVD prehrávače. Informácie o systéme sú voľne dostupné na stránke [10].

Napriek mnohým obmedzeniam je systém využívaný, pretože podporuje množstvo štandardných linuxových utilít a v projekte je zahrnutá aj špeciálna verzia štandardnej knižnice jazyka C pod menom uClib. Keďže je vyvíjaný pre mikrokontroléry bez MMU (Memory management unit), tak neposkytuje žiadnu ochranu pamäte. Plný multitasking je dosiahnuteľný, ale treba vziať do úvahy, že *fork()* nie je implementovaný. Namiesto toho je podporovaný *vfork()*, ktorý je rýchlejší, avšak nový proces využíva adresový priestor rodiča.

2.5.1 BusyBox

Bežný balík linuxových utilít by bol príliš veľký a náročný na výkon daného zariadenia. Preto μClinux obsahuje BusyBox, ktorý kombinuje veľké množstvo UNIXových utilít do jedného menšieho spustiteľného súboru. Poskytuje minimalistické verzie väčšiny utilít zahrnutých v GNU coreutils a util-linux. Zoznam podporovaných utilít a ďalšie informácie sú dostupné na stránke [6].

Väčšinou disponujú menším množstvom možností, ale správaním a funkcionalitou sú veľmi blízko svojim plným verziám. BusyBox bol vytvorený tak, aby bola čo najviac optimalizovaná veľkosť a poskytuje komplexné POSIXové prostredie aj pre jednoduchšie systémy.

Pri práci na platforme EdkDSP bola najčastejšie využívaná utilita *tftp* pre prenos súborov z počítača na EdkDSP. Na meranie dĺžky behov jednotlivých programov preložených pre EdkDSP bola využitá utilita *time*, ktorá však nebola dostupná v dodanej verzii BusyBoxu. Po zbavení závislostí na zvyšku BusyBoxu bola preložená ako samostatný spustiteľný súbor pre EdkDSP.

2.6 Vývojová doska

Ako hardware pre platformu EdkDSP bola využitá vývojová doska sp605 2.4 od firmy XILINX. Obsahuje Spartan6 FPGA čip, na ktorom bola EdkDSP syntetizovaná. Doska má 128MB DDR3 RAM pamäte a má sieťovú kartu.

Obr. 2.4: Vývojová doska XILINX sp605, obrázok prevzatý z [1]

Kapitola 3

Architektúra prekladača LLVM

Jadrom prekladača je framework LLVM (Low Level Virtual Machine). Ide o moderný, modulárny a ľahko prispôsobiteľný prekladový systém. Napriek názvu sa nejedná o virtuálny stroj v klasickom slova zmysle.

LLVM začal ako výskumný projekt na University of Illinois. Distribuuje sa s upravenou BSD licenciou a našiel široké uplatnenie v komerčnej aj akademickej sfére. Cieľom bolo vytvoriť moderný prekladový systém schopný spracovávať statickú a dynamickú kompiláciu pre viaceré programovacie jazyky.

Na rozdiel od klasických prekladačov je LLVM navrhnutý modulárne. Na spracovanie vstupného kódu využíva front-endy, ktoré staticky preložia vstupný kód a následne emitujú kód vo formáte virtuálnej inštrukčnej sady (LLVM IR). Táto vnútorná reprezentácia je nezávislá na cieľovej architektúre a v súčasnosti existuje široká škála front-endov podporujúcich mnohé jazyky (C,C++,Java, Haskell, Lua, Python, etc.).

Nad IR reprezentáciou pracujú nástroje existujúce v rámci Core libraries. Príkladom sú rozličné optimalizačné priechody zastrešené programom Opt alebo projekt Polly, ktorý implementuje automatickú paralelizáciu pomocou polyhedrálnych modelov. O prevod IR kódu do strojového jazyka cieľovej architektúry sa starajú backendy. Podporované je veľké množstvo architektúr od najbežnejších (x86, ARM) až po exotické. Informácie o architektúre LLVM som čerpal najmä z [9] a [12].

3.1 Vnútorná reprezentácia kódu

Virtuálna inštrukčná sada je navrhnutá tak, aby bola nízkoúrovňovou reprezentáciou s podporou vysokoúrovňových analýz a transformácií. Aby bolo možné dosiahnuť požadované správanie, táto sada poskytuje extenzívne jazykovo a architekturálne nezávislé typové informácie.

IR reprezentuje virtuálnu architektúru, ktorá zachytáva kľúčové operácie bežných procesorov. Zároveň sa však vyhýba špecifickým vlastnostiam, ako napríklad fyzickým registrom, zreťazeným inštrukciám alebo nízkoúrovňovým volacím konvenciám. LLVM poskytuje neobmedzenú sadu typovaných virtuálnych registrov.

Na premiestňovanie hodnôt z virtuálnych registrov do pamäte a vice versa slúžia výhradne operácie `load` a `store`. Alokovanie dát na zásobníku prebieha pomocou funkcie `alloca`. `Getelementptr` sa využíva na získavanie adries prvkov agregovaných dátových typov.

Keďže LLVM definuje virtuálnu inštrukčnú sadu, táto neponúka žiadne I/O funkcie ani

funkcie spojené s OS.

3.1.1 Trojadresný kód

Väčšina operácií je vo forme trojadresného kódu, všetky aritmetické a logické operácie majú jeden alebo dva operandy a jediný výsledok. Podstatnou výnimkou sú funkcie volacie a `phi` funkcia, ktorá súvisí s SSA formou.

IR disponuje sadou bežných aritmetických a logických operácií: add, fadd, sub, fsub, udiv, sdiv , fdiv , shl, or, xor atd.. Inštrukcie sú polymorfické, takže typ operandu zároveň definuje sémantiku inštrukcie a typ výsledku. Inštrukčná sada je silne typovaná. Podrobnejšie informácie o IR sú uvedené v referenčnom manuáli [13].

3.1.2 SSA forma

Program je v *static single assignment* (SSA) forme, ak každá premenná je definovaná presne jedenkrát a každé jej použitie nasleduje (je dominované) touto definícou. Daný spôsob reprezentácie je výhodný pre mnohé typy optimalizácie kódu. Prináša aj výrazné uľahčenie analýzy dátového toku. Preto LLVM využíva SSA ako primárnu formu reprezentácie kódu.

V praxi sa to prejavuje implicitným vytvorením virtuálneho registra pre každú inštrukciu, ktorá vracia hodnotu. Ak meno nie je stanovené v zdrojovom kóde, vygeneruje sa unikátne. Vďaka tomuto správaniu je vždy možné jednoducho priradiť definíciu ku každému použitiu hodnoty, vzniká takzvaný *use-def* reťazec.

Aby bolo možné zvládať riadiace štruktúry, ako napríklad cykly a podmienky, bolo nutné zaviesť ϕ funkciu. Jej výsledná hodnota je závislá od základného bloku, ktorý predal kontrolu. Na vstupe má dvojice základný blok a premenná. Musí byť vždy uvedená na začiatku základného bloku.

Príklad 1 Využitie ϕ funkcie pri preklade cyklu

Pôvodný kód v jazyku C

```
for( i= 0; i<200; i+= 1){
    b[i]=a[i]+c[i];
}
```

Preložený kód v LLVM IR

```
; <label>:0                                       ; preds = %newFuncRoot, %0
  %i.01 = phi i32 [ 0, %newFuncRoot ], [ %4, %0 ]
  %scevgep = getelementptr float* %b, i32 %i.01
  %scevgep2 = getelementptr float* %c, i32 %i.01
  %scevgep3 = getelementptr float* %a, i32 %i.01
  %1 = load float* %scevgep3, align 4, !tbaa !0
  %2 = load float* %scevgep2, align 4, !tbaa !0
  %3 = fadd float %1, %2
  store float %3, float* %scevgep, align 4, !tbaa !0
  %4 = add nsw i32 %i.01, 1
  %exitcond1 = icmp eq i32 %4, 200
  br i1 %exitcond1, label %.exitStub, label %0
```

3.2 LLVM priechody

Jadrom LLVM infraštruktúry je systém transformačných priechodov. K dispozícii sú mnohé predpripravené statické analýzy a transformácie. Implementované priechody sú k dispozícii v troch hlavných kategóriách.

Analysis priechody získavajú informácie o programe využiteľné v iných priechodoch. Môžu slúžiť pre vizualizáciu a odhaľovanie chýb. *Transforms* využívajú a prípadne invalidujú *analysis* priechody. Všetky *transforms* istým spôsobom modifikujú zdrojový kód. Posledným typom sú *utility* priechody. Vždy poskytujú funkčnosť, ktorá nespadá pod predchádzajúce kategórie. Typickým príkladom je priechod, ktorý extrahuje funkciu do bitového kódu.

Pred začiatkom spracovávania zdrojového kódu, musí prebehnúť niekoľko podstatných analýz. *Mem2reg* transformácia sa používa na propagovanie odkazov do pamäte na prítupy do registrov.

Ďalšou nutnou transformáciou je *Loop-simplify*. Zabezpečuje kanonikalizáciu cyklov pomocou vkladania nových vstupných a výstupných základných blokov do cyklov. Unikátny vstupný blok cyklu garantuje, že existuje jediná vstupná hrana pre cyklus prístupná mimo cyklu samotného. Vloženie nového výstupného bloku umožňuje vznik jedinej výstupnej hrany pre cyklus. Táto transformácia značne zjednodušuje neskoršiu manipuláciu s telom cyklu.

Zjednodušenú analýzu kontrolnej premennej cyklu umožňuje *Indvars* transformácia. Kanonikalizácia kontrolnej premennej prebieha v nasledujúcich troch krokoch.

1. Cyklus je transformovaný tak, aby mal jedinú kontrolnú premennú a tá vždy začína číslom 0 s krokom 1.
2. Prvý ϕ uzol v hlavičke cyklu, je kontrolnou premennou.
3. Ukazateľová aritmetika v cykle (využívajúca opakovanie) je transformovaná tak, aby používala ukazatele do polí.

Táto transformácia je značne silná, ale vyžaduje identifikovateľnú kontrolnú premennú a v ideálnom prípade aj staticky spočítateľný počet opakovaní cyklu. Nakoľko akceleračný priechod pracuje nad cyklami, ktoré vykonávajú aritmetické operácie, tieto prísne pravidlá sú vo väčšine prípadov splniteľné. Pri spočítateľnom počte opakovaní sa výstup z cyklu zabezpečuje porovnávaním kontrolnej premennej a požadovanej výstupnej hodnoty.

Pôvodný cyklus:

```
for (i= 7; i*i < 1000; ++i)
```

Porovnanie s výstupnou hodnotou:

```
for (i= 0; i!= 25; ++i)
```

LLVM vo všeobecnosti poskytuje mocnú sadu nástrojov na transformáciu a analýzu cyklov. Príčinou je množstvo aplikácií, ktoré istým spôsobom optimalizujú takéto výpočty alebo získavajú informácie o závislostiach. Zoznam všetkých dostupných priechodov s ich popisom je dostupný na stránke [17].

3.3 Generovanie kódu pomocou LLVM backendu

Možnosť generovania kódu pre veľké množstvo platforiem a jednoduché vytváranie takéhoto generátora pre nové platformy je jednou z najväčších výhod systému LLVM. Postup generovania kódu je nezávislý na cieľovej architektúre. Nakoľko bol LLVM generátor kódu využitý pri generovaní firmwarov v jednej z verzií prekladača, je účelné bližšie popísať jeho štruktúru. Informácie o generovaní kódu sú dostupné na [11].

3.3.1 Selekcia inštrukcií

Prvý krok generovania kód mení inštrukcie LLVM IR na stromovú reprezentáciu. Nazýva sa DAG (Directed acyclic graph) cieľových inštrukcií, uzly grafu sú tvorené inštanciami triedy `SDNode`. Obsahujú operačný kód a operandy reprezentovanej inštrukcie. Definovaný je aj výsledok inštrukcie, `SDNode` podporuje aj inštrukcie s viacerými výsledkami. V DAG sú obsiahnuté operačné a dátové závislosti. Časti selektoru sú generované na základe súborov popisujúcich cieľovú architektúru a jej inštrukčnú sadu (`*.td` súbory).

Jedným z najpodstatnejších konceptov v SelectionDAG je *legalizácia*. DAG je legálny, ak všetky dátové typy a inštrukcie, ktoré sa v ňom vyskytujú, sú podporované aj cieľovou architektúrou. Mapovanie inštrukcií a dátových typov obsiahnutých v LLVM IR na inštrukcie a dátové typy cieľovej architektúry sa označuje ako *lowering*. Legalizácia pozostáva z dvoch hlavných celkov.

- Legalizácia dátových typov je mocný nástroj, ktorý konvertuje dátové typy na typy natívne podporované cieľovou architektúrou. Skalárne typy sú legalizované pomocou konvertovania menších typov na väčšie (*promoting*) a pomocou rozbíjania väčších celočíselných typov na menšie (*expanding*).

  Vektorové typy sú legalizované pomocou postupného zjednodušovania až na skalárne typy (*scalarizing*) alebo pomocou zaobaľovania elementov do dostupných vektorových typov (*widening*). Informácie o cieľovej architektúre sú odvodzované z popisu jej registrov.

- Legalizácia operácií zaručuje prítomnosť operácií natívnych pre cieľovú architektúru. V backende musia byť popísané vzory jednotlivých podporovaných operácií, legalizátor ich skúsi namapovať na ešte nelegálny DAG. Ak sa operácia nedá namapovať, legalizátor ju skúsi rozdeliť na sekvenciu podporavaných operácií. Všetky zložitejšie mapovania musia byť explicitne popísané užívateľom v backende.

3.3.2 Plánovanie a formátovanie

Plánovacia fáza priradí legálnemu DAGu cieľových inštrukcií poradie jednotlivých inštrukcií. Poradie môže byť volené podľa preferencií architektúry, napríklad snaha o čo najmenšie množstvo presunov medzi registrami.

Po tom ako je správne priradené poradie, rozbije sa legálny DAG a vznikne zoznam cieľových inštrukcií. Zoznam však stále vo väčšine prípadov operuje s virtuálnymi registrami a stále je v SSA forme.

3.3.3 Alokácia registrov

Pozostáva z mapovania programu, ktorý má neobmedzené množstvo virtuálnych registrov na program, ktorý má obmedzené množstvo fyzických registrov. Využívajú sa rozličné tech-

niky farbenia grafu. Odstraňuje všetky referencie na virtuálne registre.

Všetky cieľové architektúry majú obmedzený počet registrov. Ak ich počet nie je dostatočný pre mapovanie, tak niektoré z nich sú mapované do pamäte. Tento postup sa nazýva *spilling*. Dostupné registre sú popísané pomocou registrových tried. Dostupných je niekoľko módov alokácie registrov.

- *Fast* alokátor sa používa na testovacie účely. Registre sú alokované na úrovni základných blokov s minimálnym dôrazom na znovupoužitie.
- *Basic* je alokátor využívajúci inkrementálny prístup a alokovanie podľa rozsahov platnosti. Slúži aj ako základ pre užívateľské alokátory.
- *Greedy* alokátor sa snaží o čo najväčšiu redukciu *spillingu*. Používa sa ako prednastavený alokátor v LLVM.
- *PBQP* je založený na vystavaní PBQP (Partitioned Boolean Quadratic Programming) problému a následného mapovania výsledkov na priradené registre.

3.3.4 Vkladanie prológu a epilógu

Po úspešnom nagenerovaní strojového kódu a namapovaní registrov je možné vypočítať veľkosť miesta, ktoré zaberú jednotlivé funkcie na zásobníku. Do funkcií sa vloží prológ a epilóg. Abstraktné ukazatele do zásobníku sú nahradené fyzickými adresami. V tejto fáze sú spúšťané aj optimalizácie, ako napríklad eliminácia indexu rámca.

3.3.5 Emitovanie kódu

Kód môže byť emitovaný buď vo forme assembleru pre danú architektúru, alebo priamo v podobe inštrukcií procesoru. Pre úspešný preklad stačí implementovať jednu z týchto možností. Emitovanie je založené na loweringu abstraktnej formy, v ktorej je vygenerovaný kód uložený. Podporované je aj paketizovanie VLIWovských funkcií, keďže pri VLIW procesoroch je prekladač zodpovedný za mapovanie na funkčné jednotky.

Kapitola 4

Návrh a implementácia

Kapitola sa zaoberá popisom riešenia akceleračného prekladača a kontrolného systému, ktorý je linkovaný k akcelerovanému kódu. Kontrolný systém má na starosti správu akcelerácie v čase behu programu.

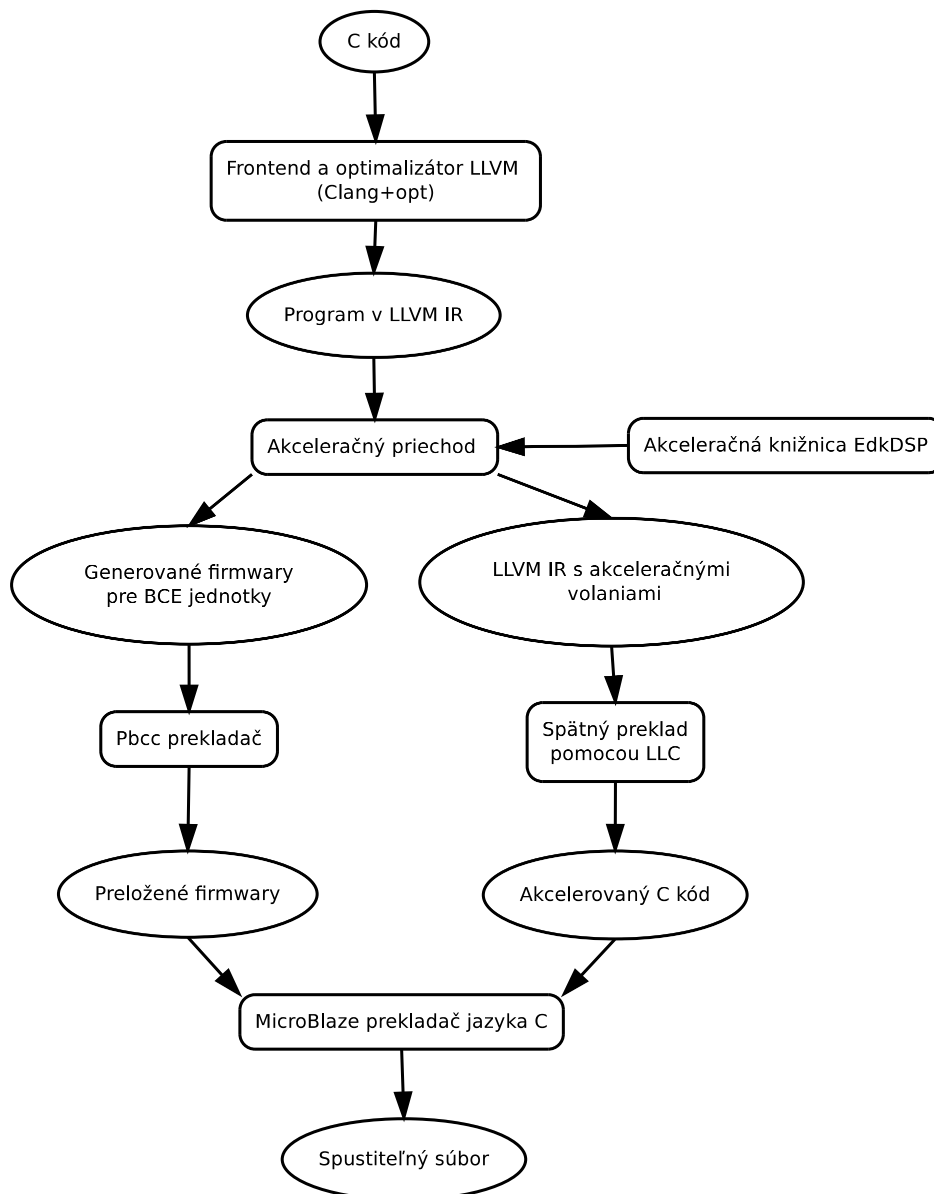
Majoritná časť funkčnosti je tvorená optimalizačným priechodom zabudovaným do prekladového systému LLVM. Časť automatizácie prekladu je riešená pomocou bash skriptov. Popísané budú jednotlivé techniky využité pri generovaní kódu firmwarov, postupy detekcie akcelerovateľných kusov kódu a samotná transformácia akcelerovaného kódu.

4.1 Návrh prekladového systému

RAVAC (Robust automatic vector accelerating compiler) prekladový systém má za úlohu spracovať neupravený program v jazyku C a transformovať ho na program využívajúci všetky dostupné prostriedky platformy EdkDSP.

Funkčný je v súčasnosti preklad pre obe hlavné verzie EdkDSP. Obidve verzie prekladača majú spoločnú štruktúru (obr. 4.1). Líšia sa najmä spôsobom, akým generujú firmwary pre BCE a stavbou kontrolného systému pre riadenie behu akcelerácie.

Pri návrhu bol kladený dôraz na modulárnosť a modifikovateľnosť celého systému. Plná integrácia do projektu SMECY predpokladá, že vstupný kód už bude modifikovaný. Konkrétne sa ráta s anotáciou cyklov, ktoré je výhodné akcelerovať a s vloženými volaniami paralelizačných funkcií. Keďže v súčasnosti tieto nástroje ešte nie sú skompletizované, RAVAC počíta s neanotovaným čistým C kódom.



Obr. 4.1: Schéma prekladového systému RAVAC

4.2 Frontend

Ako frontend slúžiaci na preklad zdrojového kódu do LLVM IR slúži Clang. Ide o relatívne nový prekladač jazykov C/C++. Založený je na rovnakých princípoch ako LLVM a ponúka mnohé výhody oproti GCC. Ponúka lepšiu integrovateľnosť s ďalšími nástrojmi, lepšie analýzy a značne rýchlejší preklad ako GCC.

4.3 Akceleračný priechod LLVM

Akceleračný priechod má na starosti identifikáciu cyklov, ktoré sa dajú akcelerovať pomocou BCE. Následne spúšťa generovanie firmwarov a nahrádza pôvodný cyklus s výpočtom volaniami kontrolného mechanizmu akcelerácie.

V nasledujúcom texte budú priblížené najpodstatnejšie princípy funkčnosti tohto akceleračného priechodu. Informácie nutné pre vytvorenie základu priechodu a jeho začlenenie do systému LLVM boli získané z článku [14].

4.3.1 Registrácia priechodu

LLVM priechody sú podstatnou súčasťou celého systému. Poskytujú rámec pre statické analýzy a transformácie. Všetky priechody sú podtriedou triedy `Pass`. Podľa požadovaných vlastností transformácie by mal vytváraný priechod dediť z jednej z nasledujúcich tried.

- `ModulePass` je najvšeobecnejšia z tried zapúzdrujúcich priechody. Dedenie z `ModulePass` umožňuje pracovanie nad celým programom ako jednou jednotkou. Neumožňuje spracovanie funkcií alebo jemnejších celkov v predvídateľnom poradí.

  Nakoľko nie je známe nič o správaní sa tohto priechodu (spracováva naraz celý program), nie je možná jeho kombinácia s ostatnými spúšťanými priechodmi.

  `ModulePass` samotný môže získať informácie o funkciách (napr. `DominatorTree`) pomocou `getAnalysis` rozhrania.

- `CallGraphSCCPass` je trieda, umožňujúca priechodu spracovávať funkcie zdola - hore podľa grafu volaní (volaní pred volajúcimi). Dedenie z tejto triedy poskytuje rámec pre priechod grafu volaní, umožňuje manažéru priechodov optimalizovať volania priechodu. Priechod prebieha nad *SCC* (silne súvisiacimi komponentami) grafu volaní. Z toho vyplýva obmedzenie, modifikovaná funkcia v jednom volaní priechodu musí byť súčasťou spracovávanej *SCC*. Priechodu nie je umožnené odobrať spracovávanú *SCC*, ale môže ju modifikovať. Táto trieda je modifikáciou `ModulePass`.

- `FunctionPass` na rozdiel `ModulePass` umožňuje priechodom pracovať nad jednotlivými funkciami v programe. Priechod odvodený od `FunctionPass` má predvídateľné správanie naprieč celým programom a má vždy len lokálne účinky. Analyzuje alebo transformuje vždy iba práve spracovávanú funkciu.

  Poradie spracovávaných funkcií nie je špecifikované, určuje ho manažér priechodov v čase behu. Nie je umožnené odoberanie alebo pridávanie funkcií do programu.

- `LoopPass` je trieda, vďaka ktorej môže priechod pracovať nad cyklami v programe. Umožňuje nezávislé spracovanie každého cyclu, ktorý sa vyskytne. Ak sú cykly zanorené, prechádza sa od najvnútornejšieho smerom k vonkajšiemu.

  Trieda umožňuje upravovať, pridávať a vymazávať funkcie, cykly a základné bloky. Jedná sa o veľmi silný nástroj slúžiaci na modifikáciu programu. `LoopPass` sa stal základom pre akceleračný priechod, vďaka svojim rozsiahlym možnostiam.

- `RegionPass` je triedou veľmi podobnou `LoopPass`. Všetky možnosti sú zhodné, avšak `RegionPass` pracuje nad regiónmi kódu vo funkciách. Regióny majú jeden vstupný a jeden výstupný základný blok. Ak sú regióny vnorené spracovávajú sa od najvnútornejšieho k vonkajšiemu.

- `BasicBlockPass` je trieda, ktorá podporuje priechod nad jednotlivými základnými blokmi. Značne obmedzujúci je fakt, že `basicBlockPass` nemôže zasahovať do *CFG* (Control Flow Graph). To znamená, že zmena inštrukcií, ktoré terminujú spracovávaný základný blok, nie je možná.

Dedenie jednej z týchto tried poskytne priechodovému frameworku informácie o požadovaných informáciách pre užívateľský priechod a umožní ho kombinovať v čase behu s ostatnými požadovanými priechodmi.

Telo akceleračného priechodu, ktorý dedí triedu `LoopPass` je tvorené implementovaním virtuálnej metódy `RunOnLoop`. Následne je nutná registrácia pomocou `RegisterPass`. Celý systém priechodov sa spúšťa programom `opt`.

4.3.2 Vlastnosti akcelerovateľného cyklu

Po tom ako je akceleračný priechod zavolaný manažérom priechodov, dostane k dispozícii odkaz na spracovávaný cyklus. Ide o odkaz na instanciu triedy `Loop`. Cyklus musí spĺňať sadu podmienok, aby mohla začať transformácia.

Priechod potrebuje získať dodatočné informácie o programe, respektíve o spracovávanom cykle. Na to slúži mechanizmus `GetAnalysis`, ktorý oznámi manažérovi priechodov zoznam priechodov, ktorých výstupy sú pri bežiacej analýze potrebné. Pre značnú odlišnosť budú uvedené prístupy k overovaniu vlastností u oboch verzií akceleračného priechodu.

Akceleračný priechod pre EdkDSPv1.0

Vstupný blok spracovávaného cyklu nemôže byť uvedený v zozname `ForbiddenBlocks`. Zoznam obsahuje odkazy na bloky, ktoré už boli spracované akceleračným priechodom. Nedá sa aplikovať mechanizmus *zabúdania* na bloky, za ktorý zodpovedá manažér priechodov. Dôvodom je nutnosť zachovania vstupného bloku, ktorý je zároveň vstupom pre *región* akceleračných volaní.

Overenie vyžaduje výsledky už spomínanej `ScalarEvolution` analýzy. Vďaka tejto analýze je možné získať počet opakovaní cyklu. V prípade ak sa táto informácia získať nedá, tak spracovávanie nepokračuje. Cyklus v sebe nemôže obsahovať žiadne ďalšie cykly.

Musí byť jasne identifikovaný unikátny vstupný a výstupný blok. Analýza musí zistiť, že cyklus už bol transformovaný priechodom `LoopSimplify` a má príslušnú formu. Identifikovateľná musí byť aj hodnota, o ktorú sa zvyšuje kontrolná premenná cyklu.

Akceleračný priechod pre EdkDSPv2.0

Vo verzii 2.0 funguje analýza a vytvorenie regiónu akceleračných volaní iným spôsobom. Potrebné sú údaje z analýzy `ScalarEvolution` a `DominatorTree`.

Nutná je správna identifikácia všetkých informácií o cykle, rovnako ako vo verzii 1.0. Rozdielom je, že región akceleračných volaní sa nemôže nadväzovať priamo na vstupný a výstupný blok cyklu, kvôli úplne odlišnému spôsobou generovania kódu a analýzy. Využíva sa funkcia `LoopExtract`, ktorá vytvorí obaľovaciu funkciu okolo spracovávaného cyklu, takzvaný *MinimalWrapper*. Jeho pôvodné telo nahradí volaním tejto funkcie.

Detekcia už spracovaného cyklu nepracuje na princípe zoznamu vstupných blokov. Pracuje na princípe detekcie *MinimalWrapper* funkcie. Ak okolo cyklu detekovaného manažérom priechodov existuje len obalovacia funkcia, je jasné, že už bol spracovaný.

O skontrolovanie vlastností cyklu sa stará trieda `LoopCorrect`, kontrolu spúšťa metóda `IsLoopCorrect`.

4.3.3 Extrakcia informácií o cykle

Vďaka využitým transformáciám zo systému LLVM, sa kontrolná premenná spracovávaných cyklov mení vždy o 1 a jej počiatočná hodnota je 0.

V skutočnom programe sa však takéto cykly vyskytujú iba zriedkavo. Všetky informácie o indexoch do polí, ktoré sú zdrojmi dát pri výpočte sú v LLVM IR zachované. Je úlohou akceleračného priechodu tieto informácie získať a využiť tak, aby akcelerácia pracovala presne nad takými dátami, s ktorými pracoval pôvodný cyklus pred transformáciami. Príklad č.2 ukazuje normalizáciu cyklu.

Príklad 2 Normalizácia netriviálneho cyklu obsahujúceho zmeny indexov polí oproti kontrolnej premennej

---

Pôvodný cyklus

```
for( i= 0; i<127; i+= 2){
    a[i]=(b[i*2+3]+(c[i]*a[i]))+c[i];
}
```

Normalizovaný, upravený cyklus pripravený na akceleráciu. Zvýraznené sú informácie o indexoch do spracovávaných polí.

```
; <label>:0 ; preds = %newFuncRoot, %0
%indvar = phi i32 [ %indvar.next, %0 ], [ 0, %newFuncRoot ]
%1 = mul i32 %indvar, 2
%scevgep = getelementptr float* %a, i32 %1
%scevgep1 = getelementptr float* %c, i32 %1
%2 = mul i32 %indvar, 4
%3 = add i32 %2, 3
%scevgep2 = getelementptr float* %b, i32 %3
%4 = load float* %scevgep2, align 4, !tbaa !0
%5 = load float* %scevgep1, align 4, !tbaa !0
%6 = load float* %scevgep, align 4, !tbaa !0
%7 = fmul float %5, %6
%8 = fadd float %4, %7
%9 = fadd float %8, %5
store float %9, float* %scevgep, align 4, !tbaa !0
%indvar.next = add i32 %indvar, 1
%exitcond = icmp ne i32 %indvar.next, 64
br i1 %exitcond, label %0, label %.exitStub
```

Vždy je známe, ako veľký je vektor spracovávaných dát, keďže je známy počet opakovaní cyklu. Na to, aby sa akcelerácia dostala k správnym dátam, je nutné zistiť, ako je kontrolná premenná modifikovaná pri prístupe k prvkom poľa.

Základom pri získavaní týchto údajov je identifikovanie inštrukcií, ktoré pracujú s pamäťou. Jedná sa vždy o `load` a `store`. Vďaka inštrukcii `getelementptr`, ktorej návratová hodnota určuje adresu prvku, sa dá zistiť, akým spôsobom bola kontrolná premenná modifikovaná. Príklad č.3 názorne ukazuje, ako je modifikovaná kontrolná premenná.

Verzie akceleračného prekladu sa líšia v spôsobe, akým sú tieto informácie získavané.

Príklad 3 Cesta od načítania ku kontrolnej premennej cyklu.

```
%indvar = phi i32 [ %indvar.next, %0 ], [ 0, %newFuncRoot ]
%2 = mul i32 %indvar, 4
%3 = add i32 %2, 3
%scevgep2 = getelementptr float* %b, i32 %3
%4 = load float* %scevgep2, align 4, !tbaa !0
```

Akceleračný priechod pre EdkDSPv1.0

Spôsob analýzy modifikácií je pevne zviazaný so spôsobom, ako prebieha analýza operácií nad vektormi a následné generovanie firmwarov. Mechanizmus bude podrobne rozobraný v sekcii venovanej generovaniu backendu. Nutná je konštrukcia stromu operácií v cykle. V bode kde priechod narazí na `store` alebo `load` spustí sa analýza modifikácií.

Analýzu zabezpečujú metódy `StoreAnalysis` a `bConstruct`. Vytvorí sa nový záznam do poľa `InfoPool`, ktoré obsahuje informácie o jednotlivých inštrukciách, ktoré pracujú s pamäťou a odkaz na pole, s ktorým pracujú. Cieľom je identifikácia všetkých inštrukcií, ktoré ovplyvňujú index do poľa s dátami.

Na základe príkladu č.3 by sa do poľa `InfoPool` uložil záznam o dátovom poli `float* b`, kde index ukazujúci na prvok je kontrolná premenná zmenená násobením konštantou a pričítaním konštanty.

Akceleračný priechod pre EdkDSPv2.0

Charakter zaznamenávaných informácií je zhodný s predchádzajúcou verziou. Keďže najvýraznejším zmenám podliehalo generovanie kódu, bolo nutné zmeniť spôsob, akým sa identifikujú inštrukcie pre prácu s pamäťou.

O identifikáciu informácií sa stará trieda `IndvarMod`. Analýza sa spúšťa volaním metódy `IndvarAnalysis`. Instancii triedy sa predá práve spracovávaný cyklus. Po spustení analýzy sa vyhľadajú inštrukcie pracujúce s pamäťou a vytvoria sa zodpovedajúce záznamy o získaných informáciách.

Počiatočným bodom získavania informácií je inštrukcia pracujúca s pamäťou. Následne je zavolaná metóda `InfoGet` nad touto inštrukciou. Metóda preskúma paramatre a identifikuje index do pamäte. Po identifikácii indexu sa zavolá metóda `TopDown`, ktorá prejde *use-def* reťazec až ku kontrolnej premennej. Pri prechádzaní vytvára záznamy o prípadných modifikáciách kontrolnej premennej. Ak žiadna modifikácia neexistuje a indexom je priamo kontrolná premenná, tak záznam obsahuje iba adresu poľa, s ktorým sa pracuje.

4.3.4 Generovanie firmwaru pre BCE

Generovanie firmwaru je jednou z kľúčových častí akceleračného priechodu. Generovaný firmware musí vykonávať výpočet zhodný s výpočtom spracovávaného cyklu. Ďalej musí mať formu validného programu v jazyku C preložiteľného prekladačom Pbcc a spustiteľného na BCE.

Odlišnosť verzií architektúry umožnila kompletné prepracovanie spôsobov, akým generovanie prebieha. Nakoľko inštrukcie pre dataflow jednotku (EdkDSPv2) nie sú pevne zviazané s konkrétnymi pamäťami, bolo možné využiť LLVM backend pri vytváraní firmwarov. Pre EdkDSPv1 bolo nutné navrhnúť vlastný algoritmus selekcie inštrukcií a mapovania registrov.

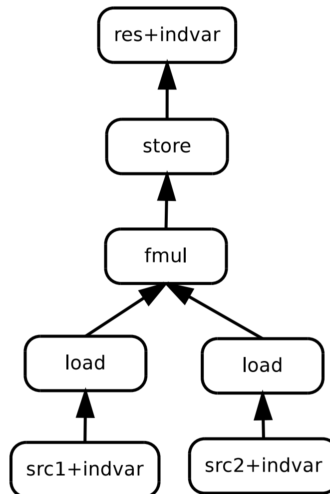
Generovanie firmwaru pre EdkDSPv1.0

Prestavať LLVM backend tak, aby mohol pracovať s inštrukciami, ktoré majú pevne určenú pamäť, s ktorou pracujú, by bola úloha nesmierne náročná, ak nie nemožná. Preto bolo výhodnejšie navrhnúť vlastný systém generovania kódu. Dôraz bol kladený na znovupoužiteľnosť systému pri takto špecifikých platformách, na efektivitu a v neposlednom rade na správnosť generovaného kódu.

Problém generovania kódu je transformovaný na problém mapovania stromov. Na strom operácií cyklu sa mapujú stromy inštruckií pre BCE. Na konci procesu ostane strom operácií BCE s namapovanými pamäťami, ktorý je následne serializovaný a transformovaný na kód v jazyku C. Inšpiráciou pre návrh postupu bol článok [5].

Konštrukcia stromu operácií

Strom operácií je binárny a je tvorený instanciami triedy `BinTree`. Obsahuje odkazy na ľavú a pravú vetvu, odkaz na operáciu, ktorú predstavuje, meno operácie a jej typ. V prípade, ak ide o inštrukciu pracujúcu s pamäťou, je obsiahnutý aj index do poľa `InfoPool`, kde sú uložené informácie o spôsobe prístupu k pamäti.



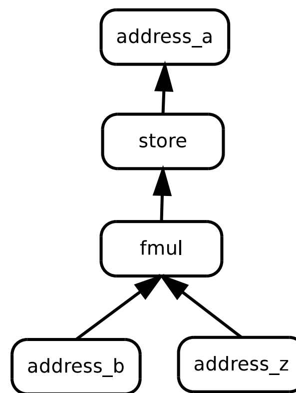
Obr. 4.2: Príklad stromu operácií.

Vystavanie stromu sa začína volaním metódy `bConstruct`, ktorej sa predá identifikovaná inštrukcia `store`. Metóda rekurzívne preskúma všetky operácie, ktoré súvisia s identifikovanou `store` inštrukciou, vytvorí binárny strom operácií, ktoré definujú výsledok. Zastaví sa pri identifikovaní inštrukcií `load`, potom spustí získavanie informácií o `load` inštrukciách. Vytvorený strom operácií je na obrázku 4.2.

Následne sa volá metóda `bTreeStoreInsert`. Upraví strom tak, aby každá operácia ukladala svoj výsledok do virtuálneho registra, pomocou novej inštrukcie `store`. Výsledky operácií teda nie sú priamo operandmi ďalšej operácie, operand je vždy načítaný z virtuálneho registra. Táto reprezentácia značne uľahčuje mapovanie stromov BCE inštrukcií na pôvodný strom.

Mapovanie inštrukcií BCE na strom operácií

Mapovanie prebieha v dvoch úrovniach. Najprv sa identifikuje operácia, ktorú má vykonať inštrukcia. Ak sa zhoduje s jednou z dostupných inštrukcií, skontroluje sa, či existuje inštrukcia s vhodným pamäťovým mapovaním. Metóda `PatterMatchingInit` inicializuje výsledný strom, ktorý bude obsahovať inštrukcie BCE. Vzory stromov, ktoré predstavujú inštrukcie BCE, sú uložené v poli `PatternPool`. Operácia násobenia je znázornená na obrázku 4.3.



Obr. 4.3: Príklad vzoru operácie `VMULT_BZ2A`.

Vzory sú získavané pri štarte akcelerácie a sú užívateľsky definovateľné. Na ich popis bol vytvorený špeciálny formát, ktorým sa dajú operácie s presne definovanými pamäťami popísať. Formát má podobu serializovaného binárneho stromu, ktorý presne zodpovedá stromu predstavujúcemu inštrukciu v poli `PatternPool`.

Postup bol tvorený s cieľom generovania čo najefektívnejšieho kódu, preto každý uzol výsledného stromu inštrukcií je ohodnotený, zvolený je strom s najmenšou výslednou hodnotu, teda s najmenším počtom inštrukcií.

Mapovanie má na starosti metóda `PatternMatching`, ktorú spúšťa metóda `PatterMatchingInit` nad stromom operácií vytvoreným v predchádzajúcom kroku. Metóda je rekurzívna a mapuje strom operácií zhora-dole. Urýchlenie procesu zabezpečuje metóda `MatchResolver`. `PatternMatching` totiž pracuje nad jednotlými vstupnými hranami vzorových inštrukcií. Samotnú zhodu podstromu stromu operácií a vzoru inštrukcie kontroluje metóda `MatchResolver`, ktorá zároveň generuje následný zoznam vstupných hrán pre ďalšie pokračovanie mapovania.

Ak `MatchResolver` určí, že poskytnutá vstupná hrana je koreňom pre strom, ktorý reprezentuje existujúcu inštrukciu, zavolá sa metóda `BankMapping`. Tá určí pamäťovú mapu zvolenej inštrukcie a zistí, či takáto inštrukcia je špecifikovaná v `PatternPool`.

Proces sa končí namapovaním celého stromu operácií, vytvorením optimálneho stromu inštrukcií.

Transformácia stromu inštrukcií na firmware

Generovanie kódu firmwaru prebieha veľmi priamočiaro. Najprv sa do súboru s firmwarom vloží prológ. Nasleduje serializácia práve vygenerovaného stromu s inštrukciami. Tento proces sa spúšťa volaním metódy `FirmwareGenerator`. Jednotlivé uzly stromu inštrukcií sa

transformujú na zodpovedajúci kód určený pre BCE. Následne sa do súboru s firmwarom vloží epilóg a vytváranie firmwaru je ukončené.

Generovanie firmwaru pre EdkDSPv2.0

Nová verzia EdkDSP umožnila využitie LLVM backendu na generovanie kódu firmwaru. Keďže pamäťové miesta už nie sú plne previazané s príslušnými inštrukciami, je možné využiť alokátor registrov dostupný v LLVM. Celý proces generovanie zastrešuje trieda `BackendRunner`. Generovanie sa spúšťa volaním metódy `BackendAnalysis`. Celý proces sa dá rozdeliť na dve hlavné fázy.

1. Najprv je nutné vygenerovať vstupný súbor pre backend, ktorý vytvorí firmware. Izolovanie kódu pre backend je značne komplikovaná činnosť. Samotné extrahovanie tela cyklu je priamočiare vďaka vytvoreniu zaobaľovacej funkcie. Ak by bol však backend spustený nad týmto kódom nevygeneroval by sekvenciu inštrukcií, ktoré kontrolujú dataflow jednotku. Vygenerovaný kód by znova reprezentoval cyklus. Preto musí prebehnúť skalarizácia výpočtu, nakoľko backendy nevedia týmto spôsobom generovať inštrukcie, ktoré pracujú nad vektormi dát.

2. Spustenie backendu nad skalarizovaným kódom. K vygenerovanému kódu je nutné doplniť správny prológ a epilóg. Následne je nutné zistiť, na ktorých adresách v pamäti BCE sa majú nachádzať vstupné dáta a kam v pamäti BCE bude uložený výsledný vektor. O to sa stará skript `fw_tweak.sh`, ktorý je spustený akceleračným priechodom po úspešnom vygenerovaní kostry firmwaru.

Skalarizácia výpočtu v spracovávanom cykle

Vo vstupnom kóde pre backend nemôže ostať nič z pôvodoného cyklu až na operácie, ktoré sa vykonajú v každej iterácii. Operácie nad poľami čísel sa transformujú na operácie nad číselnými premennými. Demontáž spracovávaného cyklu prebieha v niekoľkých krokoch.

1. Rušenie cyklu sa incializuje metódou `LoopPulverizator`. Vytvorí sa kópia obaľovacej funkcie, ktorá obsahuje telo cyklu.

2. Skalarizácia pracuje na princípe identifikácie všetkých inštrukcií, ktoré pracujú s poľami a istým spôsobom využívajú kontrolnú premennú ako index (alebo základ indexu). V praxi je identifikácia jednoduchá, keďže všetky polia sú predávané do obaľovacej inštrukcie ako parametre. Po identifikovaní týchto inštrukcií je meniaci sa index do pamäte nahradení konštantou.

3. Následne sú vystopované všetky inštrukcie, ktoré pracujú s kontrolnou premennou cyklu. Stopovanie prebieha pomocou metódy `ChainDetektor`. Sú označené a pridané do zoznamu zmazávaných inštrukcií pomocou metódy `MarkForKill`.

4. Po odstránení všetkých inštrukcií pracujúcich s kontrolnou premennou, je zmazaný ϕ uzol cyklu definujúci túto premennú.

5. Posledným krokom je opravenie `branch` inštrukcií. Po vykonaní tela bývalého cyklu vždy bezpodmienečne prebehne skok na výstupný blok tela cyklu.

Funkcia so skalarizovaným výpočtom sa následne uloží do súboru, ktorý je predaný backendu pri jeho spustení. Príklad č.4 ilustruje činnosť skalarizácie.

Príklad 4 Príklad skalarizácie výpočtu.

Pôvodoný cyklus

```
; <label>:0                                   ; preds = %newFuncRoot, %0
  %i.01 = phi i32 [ 0, %newFuncRoot ], [ %7, %0 ]
  %scevgep = getelementptr float* %a, i32 %i.01
  %scevgep2 = getelementptr float* %c, i32 %i.01
  %scevgep3 = getelementptr float* %b, i32 %i.01
  %1 = load float* %scevgep3, align 4, !tbaa !0
  %2 = load float* %scevgep2, align 4, !tbaa !0
  %3 = load float* %scevgep, align 4, !tbaa !0
  %4 = fmul float %2, %3
  %5 = fadd float %1, %4
  %6 = fadd float %5, %2
  store float %6, float* %scevgep, align 4, !tbaa !0
  %7 = add nsw i32 %i.01, 1
  %exitcond1 = icmp eq i32 %7, 127
  br i1 %exitcond1, label %.exitStub, label %0
```

Skalarizovaný výpočet s odstránenou `phi` funkciou a bezpodmienečným skokom na výstupný blok.

```
; <label>:0                                   ; preds = %newFuncRoot
  %scevgep = getelementptr float* %a, i32 0
  %scevgep2 = getelementptr float* %c, i32 0
  %scevgep3 = getelementptr float* %b, i32 0
  %1 = load float* %scevgep3, align 4, !tbaa !0
  %2 = load float* %scevgep2, align 4, !tbaa !0
  %3 = load float* %scevgep, align 4, !tbaa !0
  %4 = fmul float %2, %3
  %5 = fadd float %1, %4
  %6 = fadd float %5, %2
  store float %6, float* %scevgep, align 4, !tbaa !0
  br label %.exitStub
```

Backend Codasip určený na generovanie firmwaru

Backend bol postavený na automaticky generovanej kostre vytvorenej pomocou Codasip. Spúšťaný je bezprostredne po vytvorení vstupného súboru metódou `RunBackend` prostredníctvom programu `llc`, cieľovou architektúrou je `codasip`.

Nagenerovaná kostra poskytuje základnú funkčnosť. Aby bol generovaný kód v podobe volaní akceleračnej knižnice určený pre BCE jednotky, bolo nutné upraviť niekoľko súborov. Najmä súbory popisujúcu registrovú sadu a inštrukčnú sadu. Vytvorenie a zaradenie backendu do procesu prekladu vyžadovalo sadu niekoľkých krokov. Prehľadný návod pre písanie nového backendu je uverejnený na stránke [19], popis integrácie nového backendu do systému prekladu sa nachádza v práci [16].

1. Zaradenie vygenerovaného backendu do procesu prekladu. To sa vykoná pomocou registrácie mena cieľovej architektúry do systému LLVM a zaradením zdrojových sú-

borov do procesu prekladu.

2. Dodanie príslušných registrových tried do súboru `CodasipRegisterInfo.td`. Pridaná je trieda všeobecných registrov, ktoré pracujú s celými číslami. Takáto trieda je nutná pre úspešný preklad backendu. Pridaných je osem registrov pre čísla s plávajúcou desatinnou čiarkou. Osem registrov predstavuje osem adresovacích celkov RAM pamäte BCE. Každá zo štyroch dvojportových RAM pamätí bola rozdelená na dve časti, aby bolo možné uložiť väčšie množstvo vektorov a teda vypočítavať komplikovanejšie operácie.

   Meno každého registru pozostáva z mena pamäťovej banky, ktorú predstavuje (MBANK_A, MBANK_B, MBANK_C, MBANK_D) a adresového offsetu do danej pamäte (0, 0xFF).

3. Popis jednotlivých inštrukcií, ktoré sú podporované BCE jednotkami je dodaný do súboru `CodasipInstrInfo.td`. Popis pozostáva z definovania mena inštrukcie, typu a počtu vstupných a výstupných operandov. Následne sa definuje reťazec, ktorý bude vytlačený. Nutné je definovať výberový vzor inštrukcie, ktorý umožní backendu vybrať konkrétnu inštrukciu pri legalizácii. V príklade č.5 je uvedený popis inštrukcie `VADD`.

   Mandatórne bolo aj popísanie základných inštrukcií pre prácu s celými číslami, pre prácu s volaniami, skokmi, ukladaniami a načítaniami. Nevyužívajú sa priamo pri generovaní kódu pre firmware, avšak bez nich sa nedá zabezpečiť správna funkčnosť backendu.

4. V súbore `CodasipGenISelLowering.cpp` boli určené legálne operácie platformy. Bez správneho určenia legality operácie, backend negeneruje požadovaný kód hoci aj keď sú správne popísané inštrukcie.

Po spustení backendu prebehne selekcia inštrukcií, legalizujú sa dátové typy a operácie, plánovanie registrov. Pri emisii kódu vznikne súbor s firmwarom, ktorý je následne upravený skriptom `fw_tweak.sh`.

Posledným krokom akceleračného priechodu je náhrada pôvodoného cyklu systémom volaní manažéra akcelerácie.

4.4 Manažér akcelerácie

Manažér akcelerácie je súhrnné pomenovanie pre sústavu funkcií a dátových štruktúr, ktorých volania sa pridávajú do akcelerovaného programu a zabezpečujú správu akcelerátorov v čase behu. Medzi úlohy patrí inicializácia akcelerátorov, nahranie správneho firmwaru pre konkrétny výpočet, prenos dát do akcelerátora, spustenie výpočtu a následné získanie a predanie výsledku.

Manažér pozostáva z niekoľkých súborov, ktoré sú linkované k prekladanému programu v rozličných štádiách prekladu. Súbor `Manager.c` obsahuje definície funkcií, ktoré sú volané akcelerovaným programom v čase behu. Súbor `fw_incl_core.h` je automaticky generovaný akceleračným priechodom a obsahuje includovania súborov s vygenerovanými firmwarmi. Všetky firmwary sú registrované v hlavnej tabuľke firmwarov, do ktorej pristupuje manažér pri nahrávaní firmwaru do BCE jednotky. Registráciu do tejto tabuľky má na starosti automaticky generovaný kód v súbore `fw_incl.c`.

Príklad 5 Popis inštrukcie VADD, obsahujúci vstupné a výstupné operandy, kód, ktorý sa vytlačí v prípade výberu inštrukcie a výberový vzor

```
def f_3_reg_ops__opc_vadd__rmem__rmem__rmem__:
    \%Popis operandov
    CodasipInst<(outs fpregs:$op0), (ins fpregs:$op2in, fpregs:$op1in),

    \%Generovaný kód
  "\n\n pb2dfu_set_cnt(__indvar__);\n
    pb2dfu_set_inc(DFUAG_0, 1);\n
    pb2dfu_set_inc(DFUAG_1, 1);\n
    pb2dfu_set_inc(DFUAG_2, 1);\n
    pb2dfu_set_fulladdr(DFUAG_0,$op0 \n
    pb2dfu_set_fulladdr(DFUAG_1,$op1in \n
    pb2dfu_set_fulladdr(DFUAG_2,$op2in \n
    pb2dfu_restart_op(DFU_VADD); \n
    pb2dfu_wait4hw(); \n",

   \%Výberový vzor
  [(set fpregs:$op0,
    (f32 (fadd (f32 fpregs:$op2in), (f32 fpregs:$op1in))))]>
  {let Itinerary = Itin1;}
```

Podstatným súborom je `manager_empty.c`, je linkovaný s pôvodoným programom, pred spustením akceleračného priechodu. Cieľom je, aby akceleračný priechod mal možnosť vytvárať volania funkcií manažéra ešte predtým, ako sú vygenerované všetky potrebné súbory, ktoré sú nutné pre plnú integráciu. Týmto spôsobom sú dodané deklarácie funkcií nutných pre náhradu spracovaného cyklu mechanizmom akcelerácie.

4.4.1 Integrácia manažéra akcelerácie do akcelerovaného programu

Integráciu zabezpečuje akceleračný priechod bezprostredne po úspešnom vygenerovaní firmwaru. Postup je mierne odlišný vo verziách akceleračného priechodu, hlavne kvôli odlišnému spôsobu získavania informácií o modifikáciách kontrolnej premennej a použitých pamätiach.

Integrácia manažéra v EdkDSPv1.0

Generovanie volaní sa spúšťa volaním metódy `CallGenerator`. Integrácia prebieha v niekoľkých na seba nadväzujúcich krokoch.

1. Ako prvý musí byť vytvorený základný blok, ktorý bude obsahovať volania manažéra. Blok, ktorý obsahuje telo cyklu sa zmaže a je nahradený práve vytvoreným blokom.

2. Vloží sa volanie funkcie `fw_init_table`. Jej úlohou je kontrola tabuľky obsahujúcej firmwary a v prípade ak nie je inicializovaná, tak spustí príslušný vygenerovaný kód.

3. Funkcia `fw_init_fw` má na starosti inicializáciu BCE a vloženie firmwaru. Návratou hodnotou je index do tabuľky BCE jednotiek, ktorý jednoznačne identifikuje inicializovaný akcelerátor. Výber akcelerátora prebieha priechodom cez tabuľku akcelerátorov,

v ktorej je uložený aj príznak obsadenosti akceleračnej jednotky. Ak akcelerátor nie je obsadený, nastaví sa príznak a vráti sa jeho identifikátor.

4. Nasleduje vloženie volaní funkcií, ktoré zastrešujú prenos vektorov do pamätí akceleračnej jednotky. Počet volaní sa rovná počtu vektorov, s ktorými firmware pracuje. Funkcia sa volá `fw_data_init`. Pomocou nej sa akcelerátoru predávajú aj informácie o adresácii spracovávaných prvkov, ktoré sú prístupné v tabuľke `InfoPool`.

5. Spustenie operácie zabezpečuje funkcia `fw_op_start`.

6. Prenos výsledného vektoru a správne zastavenie akcelerátora má na starosti funkcia `fw_data_get`.

Adresy do pamätí kam majú byť uložené vstupné vektory a adresa, kde bude umiestnený výsledný vektor, sú prístupné prostredníctvom príslušných uzloch vytvoreného stromu inštrukcií pre akcelerátor.

Integrácia manažéra v EdkDSPv2.0

O začlenenie volaní manažéra sa stará trieda `FwIntegrator`. Integrácia sa spúšťa volaním metódy `Integrate`. Vkladajú sa volania rovnakých funkcií ako vo verzii pre EdkDSPv1. Hlavným rozdielom však je, že adresy pre uloženie vstupných vektorov a adresa výsledného vektoru musia byť získané zo súboru. Bol vygenerovaný skriptom `fw_tweak.sh`, ktorý robí dodatočné úpravy v súbore s firmwarom vygenerovaným pomocou backendu. Zároveň sa stará o extrakciu adries, ktoré musia byť známe pri integrovaní volaní manažéra akcelerácie.

4.5 Prekladový skript

Popísaný prekladový systém sa uvádza do chodu spustením skriptu `ravac`. Parametrom pre skript musí byť meno súboru s programom, ktorý sa má akcelerovať. Preklad prebieha v niekoľkých krokoch.

1. Pomocou `clang` je vstupný súbor preložený do LLVM IR.

2. Následne je prilinkovaná kostra manažéra pomocou `llvm-link`.

3. Nad získaným kódom je spustená sada už popísaných transformačných priechodov, pomocou `opt`.

4. Po úspešnej transformácii je pomocou programu `opt` spustený akceleračný priechod.

5. Ak správne prebehne akcelerácia, tak sa výsledný LLVM IR súbor preloží späť do jazyka C pomocou programu `llc`, v tomto prípade je pre volaný backend cieľovou platformou jazyk C.

6. Všetky vygenerované firmwary sú preložené prekladačom `pbcc`.

7. Nakoniec sa akcelerovaný C kód, telo manažéra priechodov a preložené kódy firmwarov preložia a zlinkujú spolu pomocou `microblaze-uclinux-gcc`.

8. Výstupný akcelerovaný binárny súbor sa prenesie do vývojovej dosky a spustí sa.

Proces prekladu je znázornený na obrázku 4.1. Pomocou prepínačov je možné spustiť skript tak, aby generoval len akcelerovaný LLVM IR kód a nevykonával kompletný preklad alebo aby sa spravil preklad neupraveného vstupného súboru bez spustenia akcelerácie.

Kapitola 5

Výsledky

Vytvorené implementácie prešli testovaním zameraným na overenie funkčnosti generovaného kódu. Okrem funkčnosti bolo sledované, ako sa bude meniť doba behu jednotlivých testovacích programov s použitím rôznych prekladačov.

5.1 Preklad vzorového programu

Vzorový program pozostáva z niekoľkých funkcií obsahujúcich cykly s výpočtami pracujúcimi nad vektormi floatov . V nasledujúcom príklade č.6 bude názorne predvedený postup prekladu so zameraním na finálne generované volania a automaticky generované časti manažéra akcelerácie.

Príklad 6 Program obsahuje dva akcelerovateľné cykly s výpočtami.

```
for( i= 0; i<200; i+= 1)
    a[i]=(b[i]+(c[i]*a[i]))+c[i];
  .
  .
  .
for( i= 0; i<250; i+= 3)
    b[i]=a[i+1]*c[i*2]+b[i];
```

Po spracovaní vstupného kódu pomocou frontendu a úvodných transformačných priechodov je v podobe LLVM IR. Predpripravený kód má v sebe deklarácie funkcií (príklad č.7), ktorých volania vkladá akceleračný priechod.

Príklad 7 Deklarácie kontrolných funkcií v LLVM IR.

```
declare void @fw_data_get(i32, i32, i32, i32, float*, i32, i32, i32)
declare i32 @fw_init_fw(i32)
declare void @fw_data_init(i32, i32, i32, i32, float*, i32, i32, i32, float)
declare void @fw_op_start(i32)
declare void @fw_init_table()
```

Akceleračný priechod nahradí telá cyklov novými blokmi s volaniami manažéra (príklad č.8). Parametre funkcií `fw_data_init` a `fw_data_get` zahŕňajú všetky informácie o adreso-

Príklad 8 Bloky s volaniami nahradzujúce telá cyklov.

```
man_block_0:                                          ; preds = %newFuncRoot
  call void @fw_init_table()
  %wrk_id = call i32 @fw_init_fw(i32 0)
  call void @fw_data_init(i32 %wrk_id, i32 255, i32 1, i32 1,
                     float* %b, i32 0, i32 0, i32 200, float 0.000000e+00)
  call void @fw_data_init(i32 %wrk_id, i32 0, i32 1, i32 1,
                     float* %c, i32 0, i32 0, i32 200, float 0.000000e+00)
  call void @fw_data_init(i32 %wrk_id, i32 0, i32 2, i32 1,
                     float* %a, i32 0, i32 0, i32 200, float 0.000000e+00)
  call void @fw_op_start(i32 %wrk_id)
  call void @fw_data_get(i32 %wrk_id, i32 0, i32 1, i32 1,
                     float* %a, i32 0, i32 0, i32 200)
  br label %.exitStub
  .
  .
  .
man_block_1:                                          ; preds = %newFuncRoot
  call void @fw_init_table()
  %wrk_id = call i32 @fw_init_fw(i32 1)
  call void @fw_data_init(i32 %wrk_id, i32 0, i32 1, i32 2,
                     float* %a, i32 1, i32 3, i32 84, float 0.000000e+00)
  call void @fw_data_init(i32 %wrk_id, i32 255, i32 1, i32 2,
                     float* %c, i32 0, i32 6, i32 84, float 0.000000e+00)
  call void @fw_data_init(i32 %wrk_id, i32 0, i32 2, i32 2,
                     float* %b, i32 0, i32 3, i32 84, float 0.000000e+00)
  call void @fw_op_start(i32 %wrk_id)
  call void @fw_data_get(i32 %wrk_id, i32 0, i32 1, i32 2,
                     float* %b, i32 0, i32 3, i32 84)
  br label %.exitStub
```

vaní vektorov, ukazateľ na začiatok poľa v hlavnej pamäti a adresy do pamätí BCE. Známa musí byť aj veľkosť vektoru a spôsob, akým boli jeho prvky adresované v pôvodnom cykle. Zároveň boli akceleračným priechodom vygenerované súbory obsahujúce firmware. V tomto bode sa verzie akceleračného priechodu líšia len pomenovaním firmwarov. Zatiaľ, čo verzia pre EdkDSPv1.0 využíva mená odvodené od operácií vykonávaných firmwarom a poradového čísla spracovávaného firmwaru, verzia pre EdkDSPv2.0 generuje firmwary pomenované podľa obaľovacej funkcie. Pre EdkDSPv1.0 sú to súbory a pre verziu 2.0 `fw_2_foo2__clone.c` a `fw_1_foo__clone.c`.

Automaticky generovaná musí byť aj funkcia, ktorá registruje vytvorené firmwary do poľa, pomocou ktorého k nim manažér pristupuje (príklad č.9). Po ukončení akceleračného priechodu sú všetky časti preložené tak, ako je popísané v predchádzajúcej kapitole.

Príklad 9 Generovaná funkcia registrujúca preložené firmwary.

```
#include "fw_incl_core.h"
void fw_reg(const unsigned int **fwx){
    fwx[0] = fw_1_foo__clone;
    fwx[1] = fw_2_foo2__clone;
}
```

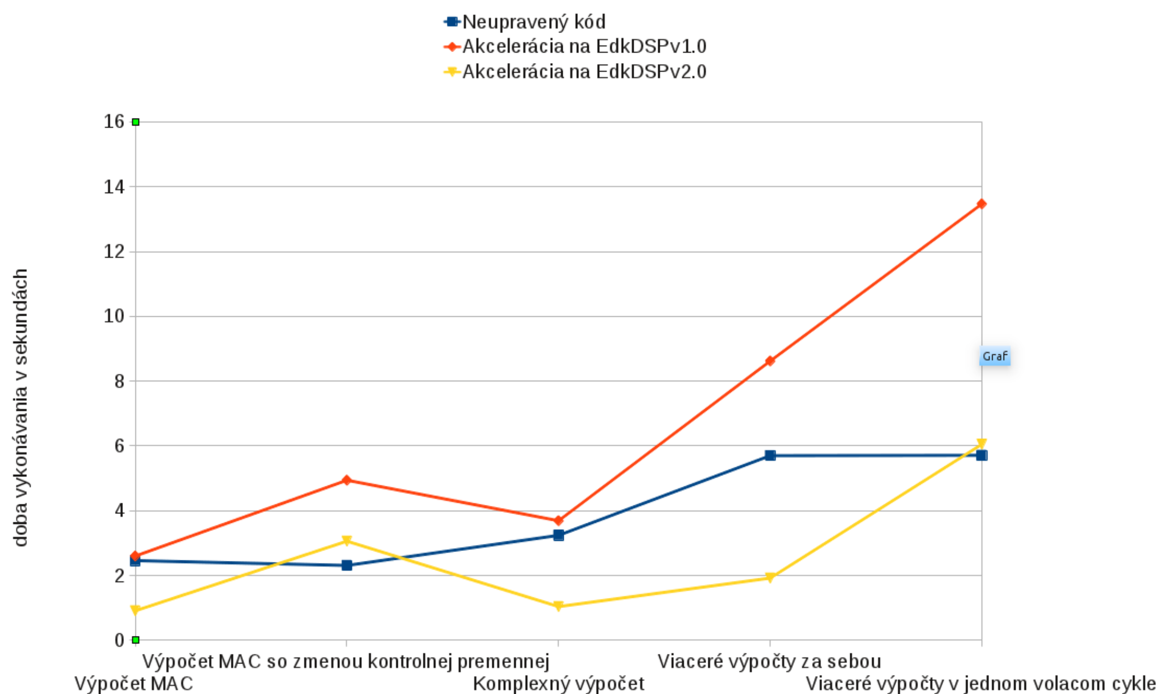
5.2 Porovnanie výkonu akcelerovanej a neakcelerovanej aplikácie

Testovanie prebiehalo s programami, ktoré extenzívne využívajú funkcie obsahujúce akcelerovateľné výpočty. Cieľom testovania bolo zistiť, akým spôsobom ovplyvní čas behu programu zapnutá a vypnutá akcelerácia alebo komplikovanosť výpočtu. Časy behu testovacieho programu boli merané utilitou `time`. Pri meraní sa využil program, ktorý 10000 krát spustí funkciu s výpočtom. Namerané výsledky v tabuľke 5.1 ukazujú možnosti akcelerácie

Tabuľka 5.1: Namerané časy pre jednotlivé testovacie programy.

Program	Neupravený kód	Akcelerácia na EdkDSPv1.0	Akcelerácia na EdkDSPv2.0
Výpočet MAC	real 0m 2.46s	real 0m 2.60s	real 0m 0.91s
Výpočet MAC so zmenou kontrolnej premennej	real 0m 2.31s	real 0m 4.94s	real 0m 3.06s
Komplexný výpočet	real 0m 3.24s	real 0m 3.69s	real 0m 1.04s
Viaceré výpočty za sebou	real 0m 5.70s	real 0m 8.62s	real 0m 1.92s
Viaceré výpočty v jednom volacom cykle	real 0m 5.71s	real 0m 13.47s	real 0m 6.05s

na EdkDSP, prehľadne sú výsledky zobrazené aj v grafe 5.1. Verzia 1.0 má značne menší výkon aj oproti neakcelerovanému kódu, bežiacemu na procesore MicroBlaze. Hlavným dôvodom je absencia DMA radiča, takže všetky prenosy prebiehajú prostredníctvom hlavného procesora. Najmarkantnejšie spomalenie je vidieť pri spustení programu, kde prebiehajú volania výpočtových funkcií v cykle za sebou. Kvôli dátovej intenzite výpočtu a častému premiestňovaniu firmwarov v dôsledku algoritmov manažéra, je program viac ako dvakrát pomalší oproti verzii bez akcelerácie.



Obr. 5.1: Graf znázorňujúci namerané časy pre jednotlivé programy.

Urýchlenie nastáva až pri verzii 2.0 a to najmä pri programoch, kde kontrolé premenné priamo indexujú vektory. Pri výpočte MAC je urýchlenie viac ako dvojnásobné. Čím komplexnejší výpočet je spúšťaný v BCE, tým výraznejšie je urýchlenie. Výpočet komplexnej operácie je rýchlejší skoro trojnásobne. Spomalenie nastáva ak je nutné počítať s modifikovanými indexmi, alebo ak program vyžaduje časté zmeny firmwaru.

Kapitola 6

Záver

Výsledkom práce sú dve odlišné verzie prekladového systému pre verzie platformy EdkDSP. Implementované boli dva odlišné prístupy k realizácii prekladu pre atypickú multiprocesorovú platformu a systém kontroly akcelerácie v čase behu.

Riešenia boli otestované nad sadou progamov, ktoré obsahovali výpočty pracujúce nad poľami čísiel s plávajúcou desatinnou čiarkou. Od použitej verzie platformy a prekladača závisela miera urýchlenia akcelerovaného programu oproti pôvodnému kódu. Vytvorené riešenia majú sadu obmedzení, ktoré kladú nároky na podobu vstupného kódu:

- je nutné aby cyklus mal v čase prekladu známy počet opakovaní,
- v rámci akcelerovaného cyklu môže existovať iba jeden výpočet,
- spracovávané vektory čísel musia mať obmedzenú velkosť (256 `floatov`).

Riešenia generujú funkčný kód, ktorý na rozdiel od nespracovaného kódu využíva všetky hardwarové prostriedky poskytnuté platformou EdkDSP. Výsledky práce si nájdu svoje uplatnenie v rámci projektu SMECY. Vývoj naďalej pokračuje, pričom pripravovaná je sada vylepšení:

- optimalizovanie množstva prenosov dát medzi BCE a procesorom,
- vylepšenie algoritmov manažéra akcelerácie (optimalizovať počet nahrávaní firmwaru),
- vylepšenie podpory spúšťania viacerých akcelerátorov v samostatných vláknach,
- zaujímavá je aj myšlienka generovania dataflow jednotky v čase prekladu programu.

Využitie viacerých BCE naraz je teoreticky možné, limitujúca môže byť WAL.

Jadro činnosti spočívalo najmä v študovaní prekladového systému LLVM, platformy EdkDSP, akceleračných knižníc pre túto platformu a vo vytváraní akceleračného priechodu. Najväčšie problémy pri realizácii spôsobovala v niektorých prípadoch nedostatočná alebo nepresná dokumentácia LLVM. Náročné bolo sprevádzkovanie dodaného prekladového systému pre EdkDSP a správne využitie akceleračnej knižnice.

Získal som hodnotné znalosti v oblasti konštrukcií transformačných priechodov, vytvárania backendov pre systém LLVM a v práci s multiprocesorovými platformami. Cennou skúsenosťou bolo zapojenie sa do práce na medzinárodnom projekte SMECY a možnosť spolupracovať s výskumným týmom Lissom.

Literatúra

[1] Spartan-6 FPGA SP605 Evaluation Kit.
http://www.xilinx.com/products/boards-and-kits/EK-S6-SP605-G.htm.

[2] Webové stránky projektu SMECY. http://www.smecy.eu.

[3] Webové stránky výskumenj skupiny Lissom.
http://www.fit.vutbr.cz/research/groups/lissom/index.html.

[4] SMECY: Platform EdkDSP overview. 2011.

[5] Aho, A. V.; Ganapathi, M.; Tjiang, S. W. K.: Code generation using tree matching and dynamic programming. *ACM Trans. Program. Lang. Syst.*, ročník 11, č. 4, Říjen 1989: s. 491–516, ISSN 0164-0925, doi:10.1145/69558.75700.

[6] Andersen, E.: BusyBox [online]. http://www.busybox.net/, 2012-04-22 [cit. 2012-04-25].

[7] Bartosinski, R.: EdkDSP platform worker abstraction layer [online].
http://sp.utia.cz/index.php?ids=results&id=edkdspsdkapi, 2011-03-11 [cit. 2012-05-01].

[8] Bartosinski, R.: EdkDSP platform PB2 firmware programming interface [online].
http://sp.utia.cz/index.php?ids=results&id=edkdspsdkapi, 2011-03-11 [cit. 2012-05-02].

[9] Brown, A.; Wilson, G.: *The Architecture of Open Source Applications*. 2011, ISBN 978-1-257-63801-7, 432 s.

[10] Dionne, D. J.; Durrant, M.: Embedded Linux/Microcontroller Project [online].
http://www.uclinux.org/, 2012-04-01 [cit. 2012-05-02].

[11] Lattner, C.: The LLVM Target-Independent Code Generator [online].
http://llvm.org/docs/CodeGenerator.html, 2012-04-19 [cit. 2012-05-01].

[12] Lattner, C.; Adve, V.: The LLVM Instruction Set and Compilation Strategy. Tech. Report UIUCDCS-R-2002-2292, CS Dept., Univ. of Illinois at Urbana-Champaign, 2002-08-02.

[13] Lattner, C.; Adve, V.: LLVM Language Reference Manual [online].
http://llvm.org/docs/LangRef.html, 2012-04-19 [cit. 2012-05-04].

[14] Lattner, C.; Laskey, J.: Writing an LLVM Pass [online].
http://llvm.org/docs/WritingAnLLVMPass.html, 2012-04-19 [cit. 2012-04-25].

[15] Liu, D.: *Embedded DSP processor design.* Burlington: Morgan Kaufmann, první vydání, 2008, ISBN 978-0-12-374123-3, 778 s.

[16] Nagy, M.: *Popis mikroprocesorové architektury AVR32 pro překladač LLVM.* Bakalářská práce. Brno: FIT VUT v Brně, 2009. 40 s.

[17] Spencer, R.; Henriksen, G.: LLVM's Analysis and Transform Passes [online]. `http://llvm.org/docs/Passes.html`, 2012-04-19 [cit. 2012-05-04].

[18] Sykora, J.: PicoBlaze Compilation User Guide. dokument dodávaný spoločne s prekladačom Pbcc, 2011-01-31.

[19] Woo, M.; Brukman, M.: Writing an LLVM Compiler Backend [online]. `http://llvm.org/docs/WritingAnLLVMBackend.html`, 2012-04-19 [cit. 2012-05-03].

Dodatok A

Inštalácia prekladového systému

Vytvorený prekladový systém pozostáva z niekoľkých samostatne sa inštalujúcich súčastí. Nutná je kompletná inštalácia systému LLVM s pridaným akceleračným priechodom, prekladač jazyka C pre PicoBlaze a MicroBlaze procesor.

V prípade ak ide o verziu určenú pre EdkDSPv2.0 je nutné do systému LLVM pridať aj backend generujúci firmwary. Obe verzie potrebujú príslušnú verziu WAL knižnice a PBBCELIB.

A.1 Inštalácia systému LLVM a akceleračného priechodu

Funkčný prekladový systém pozostáva z dvoch častí. `LLVM core` predstavuje všetky nástroje a knižnice hlavnej časti prekladového systému. Samostatne dodávaný je frontend, ktorý prekladá vstupný kód do LLVM IR.

Pred samotným prekladom LLVM, je nutné vytvoriť adresár pre objektové súbory a adresár pre hotovú inštaláciu. Vo všeobecnosti je odporúčané mať zdrojové súbory separované od objektových v samostatnom adresári.

Konfigurácia pred prekladom je vykonávaná skriptom `configure`. Uvedený je zoznam základných konfiguračných volieb:

- `--disable-optimized` zakazuje preklad s odstraňovaním debugovacích symbolov a vypína optimalizácie,
- `--enable-targets=` určuje backendy cieľových architektúr, ktoré budú preložené a pridané k programu `llc`,
- `--prefix=` určuje adresu priečinka, do ktorého bude preložené LLVM nainštalované.

Príklad 10 Príklad postupu inštalácie LLVM.

```
cd OBJ_ROOT
SRC_ROOT/configure --disable-optimized --enable-targets=cbe,codasip
                   --enable-assertions --enable-debug-symbols
                   --enable-expensive-checks --prefix=INSTALL_PATH
make
make install
```

Príklad č.10 uvádza postup inštalácie LLVM. V príklade sú využité cesty `SRC_ROOT`, ktorá predstavuje cestu ku koreňovému adresáru zdrojových súborov a `OBJ_ROOT`, kde budú umiestnené objektové súbory. Medzi cieľové backendy sú zahrnuté `cbe`, ktorého cieľovým kódom je jazyk C a `codasip`, ktorý slúži na generovanie firmwarov pri verzii 2.0.

Inštalácia frontendu a akceleračného priechod prebieha pomocou pridania zdrojových súborov s príslušným LLVM make súborom do zdrojových súborov LLVM. Ak sa zdrojové súbory Clangu pridajú do priečinka `SRC_ROOT/tools` pred konfiguráciou, prekladový systém ich preloží automaticky.

Zaregistrovanie a inštalácia akceleračného priechodu vyžaduje pridanie zdrojových súborov do priečinka `SRC_ROOT/lib/Transforms` a následné preloženie. Zdrojové súbory backendu Codasip musia byť pridané v priečinku `SRC_ROOT/lib/Target`.

A.2 Inštalácia ostatných súčastí systému

Pre plnú funkčnosť je nutné pridať prekladače jazyka C pre MicroBlaze a PicoBlaze do premennej `PATH`. Prekladový skript `ravac` musí byť spúšťaný z priečinku `SRC`, kde je umiestnený prekladaný program.

Zdrojový súbory manažéra akcelerácie musia byť v priečinku `SRC/autogen_tmp/link`, knižnice WAL a PBBCELIB sú umiestnené v priečinku `SRC/autogen_tmp/api`. V priečinku `SRC/autogen_tmp` sa nachádzajú súbory priamo generované prekladovým systémom v čase prekladu, napríklad firmwary.