

# VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ  
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY  
DEPARTMENT OF INFORMATION SYSTEMS

## ZRYCHLENÍ VYKONÁVÁNÍ SOFTWARE POMOCÍ AUTOMATICKÝCH INSTRUKČNÍCH ROZŠÍŘENÍ

BAKALÁŘSKÁ PRÁCE

BACHELOR'S THESIS

AUTOR PRÁCE

AUTHOR

STANISLAV MELO

BRNO 2013



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ  
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ  
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY  
DEPARTMENT OF INFORMATION SYSTEMS

# ZRYCHLENÍ VYKONÁVÁNÍ SOFTWARE POMOCÍ AUTOMATICKÝCH INSTRUKČNÍCH ROZŠÍŘENÍ

SOFTWARE EXECUTION ACCELERATION USING AUTOMATIC INSTRUCTION SET  
EXTENSIONS

EXTENSIONS

BAKALÁŘSKÁ PRÁCE

BACHELOR'S THESIS

AUTOR PRÁCE

AUTHOR

STANISLAV MELO

VEDOUCÍ PRÁCE

SUPERVISOR

Ing. KAREL MASARÍK, Ph.D.

BRNO 2013

## Abstrakt

Jednou z důležitých vlastností aplikačně specifických procesorů je jejich výkon. Aby byl co největší, musí se procesor přizpůsobit potřebám aplikace, kterou bude vykonávat. Jedním ze způsobů přizpůsobení se je hledání vhodných instrukcí, které se následně spojí do jedné speciální instrukce. Daná instrukce je pak implementována v hardwaru jako jeden funkční blok, takže se vykoná rychleji. Tahle práce popisuje problém nalezení a vybrání vhodných kandidátů na instrukční rozšíření. Také poskytuje stručný přehled pár nejznámějších algoritmů na řešení tohoto problému. Dále se práce detailněji zabývá algoritmem single-cut a jeho implementací. Na závěr je zjištěné zrychlení, kterého se dosáhlo na testovaných programech.

## Abstract

One of the important feature of application specific processors is performance. To maximize it, the processor must adapt to needs of application that it is going to perform. One of the ways to do that is to search for appropriate instructions that can be joined into one special instruction. This instruction is then implemented in hardware as a single function block. This bachelor's thesis describes problem of finding and selecting suitable candidates for instruction-set extension. It also provides brief overview of the few best known algorithms that solve this problem. Moreover the thesis deals with the single-cut algorithm and its implementation in more detail. Eventually the achieved speedup is found on tested programs.

## Klíčová slova

instrukční rozšíření, algoritmus single-cut, LLVM IR, základní blok

## Keywords

instruction-set extension, single-cut algorithm, LLVM IR, basic block

## Citace

Stanislav Melo: Zrychlení vykonávání softwaru pomocí automatických instrukčních rozšíření, bakalářská práce, Brno, FIT VUT v Brně, 2013

# Zrychlení vykonávání softwaru pomocí automatických instrukčních rozšíření

## Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Karla Masaříka Ph.D. Další informace mi poskytl Ing. Adam Husár. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....  
Stanislav Melo

15.5.2013

## Poděkování

Chtěl bych poděkovat svému vedoucímu Ing. Karlu Masaříkovi Ph.D. a mému konzultantovi Ing. Adamu Husárovi za jich připomínky a odbornou pomoc při konzultacích k této práci. Dále bych chtěl poděkovat své rodině za podporu při studiu.

© Stanislav Melo, 2013.

*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

<b>1</b>	<b>Vstavane systémy a prekladače</b>	<b>3</b>
1.1	Vstavane systémy . . . . .	3
1.2	Štruktúra prekladačov . . . . .	4
1.3	Začlenenie práce . . . . .	5
1.4	LLVM IR . . . . .	5
<b>2</b>	<b>Teoretické základy, predstavenie problematiky a návrhy riešenia</b>	<b>7</b>
2.1	Základné pojmy . . . . .	7
2.2	Problémy nachádzania ISE . . . . .	7
2.3	Používané algoritmy . . . . .	9
2.3.1	MaxMiso . . . . .	9
2.3.2	Single-cut . . . . .	9
2.3.3	ISEGEN . . . . .	9
2.3.4	Genetické algoritmy . . . . .	10
<b>3</b>	<b>Praktická ukážka algoritmu single-cut</b>	<b>12</b>
3.1	základný blok v LLVM IR . . . . .	12
3.2	Teória algoritmu . . . . .	12
3.3	Praktická ukážka . . . . .	15
<b>4</b>	<b>Implementácia algoritmu</b>	<b>16</b>
4.1	Základný modul ise_pass . . . . .	16
4.2	Modul ise_algorithm . . . . .	17
4.3	Modul function_finish . . . . .	17
<b>5</b>	<b>Testovanie programu a jeho výsledky</b>	<b>18</b>
5.1	Výsledky testov . . . . .	18
5.2	Nedostatky programu . . . . .	18
<b>6</b>	<b>Záver</b>	<b>20</b>
<b>A</b>	<b>Obsah CD</b>	<b>22</b>
<b>B</b>	<b>Návod na preloženie priechodu optimalizátora</b>	<b>23</b>
<b>C</b>	<b>Návod na odtestovanie priechodu</b>	<b>24</b>
<b>D</b>	<b>Parametre prechodu</b>	<b>25</b>

# Úvod

V dnešnej dobe sa stretávame s procesormi takmer všade. Nachádzajú sa v rôznych zariadeniach. Od tých zložitejších ako napríklad počítače, mobilné telefóny či autá, až po tie jednoduchšie ako sú digitálne hodinky, fotoaparáty alebo mp3 prehrávače. Procesory v malých, jednoúčelových zariadeniach vykonávajú špecifickú činnosť neustále dokola. Je snaha aby ju vykonávali čo najrýchlejšie, s čo najmenšou spotrebou energie a ich výroba bola čo najlacnejšia. Najvhodnejšie by bolo, keby jeden procesor vykonával jednu činnosť, pre ktorú by bol maximálne optimalizovaný. Ich návrh však v dnešnej dobe nieje dostatočne rýchly, aby sa tento princíp dal využiť. Procesor musí totiž prejsť niekoľkými iteráciami vývoja, pokiaľ je výsledok dostatočne optimálny. Návrh taktiež neprebíha plne automaticky, ale je potrebný zásah odborníkov. Takisto cena návrhu je prekážkou. Preto sa používajú všeobecnejšie návrhy procesorov, pre určité druhy aplikácií s možnosťou dodatočného prispôbena sa pre konkrétnu aplikáciu. Hľadanie a používanie inštrukčných rozšírení je jedným zo spôsobov, ako procesor prispôbiť programu. Ide o nachádzanie vhodných skupín inštrukcií, ktoré sa združia do jednej. To sa deje nad kódom programu s vhodnou úrovňou abstrakcie, napríklad v nízkoúrovňovom medzikóde. Potom sa nová inštrukcia premietne do funkčného bloku hardwaru, ktorý sa dá vytvoriť napríklad v čipe FPGA. Keďže bude implementovaná hardwarovo, vykoná sa rýchlejšie.

Na úvod do problematiky obsahuje kapitola 1 základné informácie o vstavaných systémoch a prekladačoch, aj rozdelenie a štruktúru prkladačov. V ďalšej časti hovorí, do ktorej ich časti vyhľadávanie ISE patrí a ako spolupracuje s ostatnými časťami. Na záver predstavuje projekt LLVM a medzikód LLVM IR, s ktorým algoritmus pracuje.

Kapitola 2 teoreticky vysvetľuje problém nachádzania inštrukčných rozšírení. Preto je jej začiatok venovaný objasneniu pojmov, ktoré sú k tomu potrebné. Nakoniec po samotnom vysvetlení predstavuje kapitola aj niektoré algoritmy, ktoré túto problematiku riešia.

Podrobný popis algoritmu single-cut sa nachádza v kapitole 3. Okrem toho obsahuje aj praktickú ukážku jeho práce na jednoduchom príklade.

Kapitola 4 sa venuje implementácii daného algoritmu v jazyku C++, ako aj implementáciu podporných funkcií potrebných pre jeho vykonanie a pre transformovanie výsledkov algoritmu na funkciu v jazyku LLVM IR.

Analýza výsledkov a porovnanie rýchlosti vykonávania programu bez a s inštrukčnými rozšíreniami sa nachádza v kapitole 5.

Záver sumarizuje výsledky práce a predkladá návrhy na jej ďalšie vylepšenie a rozšírenie.

# Kapitola 1

## Vstavané systémy a prekladače

Vstavané zariadenia sú čoraz častejšou súčasťou nášho života. Snažia sa čo najviac priblížiť svojej úlohe aj pomocou optimalizácii v prekladačoch. Jednou z nich je nachádzanie inštrukčných rozšírení. Preto začiatok kapitoly stručne predstavuje tieto zariadenia a konštrukciu prekladačov. Ukazuje tiež ako je nachádzanie prepojené s ostatnými časťami prekladača.

### 1.1 Vstavané systémy

Vstavané systémy môžeme definovať ako systémy, ktoré sú integrované, vstavané, do väčších systémov a vykonávajú neakú špecifickú funkciu [8]. Spracovanie informácií týmto systémom nieje priamo viditeľné a nieje to fakt, ktorý zákazníka zaujíma. Človek si nekúpi mobil kvôli značke a rýchlosti hodín procesoru. On ani nemusí vedieť, že daný systém nejaké informácie spracováva, lebo vstavané zariadenia majú väčšinou limitované užívateľské rozhranie. To neobsahuje obrazovky či klávesnicu, ale pár tlačidiel a jednoduchý číslcový displej. Podľa [8] existuje pár vlastností, ktoré identifikujú vstavané systémy.

Prvou vlastnosťou je efektívnosť zariadenia. Vstavané systémy musia byť čo najviac efektívne po všetkých stránkach. Tie, ktoré sú prenosné, ako mobilné telefóny či kalkulačky, sa musia pri napájaní spoliehať na batériu. Preto je na energetickú efektívnosť kladený dôraz. Tiež je dôležité minimalizovať váhu zariadenia a čo najlepšie využiť priestor na čipe, aby zariadenie nenaberalo na veľkosti a malo prijateľnú cenu. Pri svojej veľkosti a s obmedzenými zdrojmi musí byť čo najvýkonnejšie. To platí hlavne pre grafické procesory, ktoré vykonávajú výpočtovo náročné úlohy.

Druhá vlastnosť je zvyšujúce sa množstvo softvéru. Špeciálny hardvér vykonáva iba niektoré periférne operácie. Hlavnou príčinou je flexibilita softvéru, ktorá umožňuje jednoduché vylepšenia existujúcich systémov, neskoré zmeny v návrhu a skracuje čas vývoja. Zákazníkom tiež poskytuje viacej funkcionality. Hoci je vykonávanie programov softvérovo pomalšie, pre väčšinu dnešných zariadení sú existujúce procesory dostatočne výkonné.

Kompatibilita súborov inštrukcii je ďalšou vlastnosťou. Na rozdiel od procesorov v PC doméne, vo vstavaných systémoch je udržiavanie kompatibility inštrukcii medzi zariadeniami odsunuté na vedľajšiu koľaj. Vďaka tomu majú systémy možnosť používať svoje inštrukcie, ktoré im viacej vyhovujú. Napríklad procesory na spracovanie signálov môžu používať optimalizované inštrukcie obsahujúce násobenie, modulo adresovanie, paralelné operácie a špeciálne aritmetické módy, ktoré vykonávanie programu urýchlia. Keďže je každý systém špecializovaný na určitú funkciu, existuje ich veľké množstvo.

Spôľahlivosť je ďalšia z vlastností, ktorá je dôležitá. Ľudia očakávajú, že ich zariadenia budú fungovať v každom čase. Hlavne ak sa jedná o systémy kritické pre bezpečnosť.

Zaujímavým faktom je, že funkcia vstavaných zariadení je známa už pri dizajne a nie je požadovaná flexibilita neskoršieho pridania funkčnosti. To umožňuje návrhárom dizajn prispôbiť a dosiahnuť čo najväčšiu efektívitu zariadenia.

Efektívitu a ďalšie vyššie spomenuté vlastnosti je možné dosiahnuť navrhnutím procesoru špeciálne pre vybranú aplikáciu. Kvôli cene a času vývoja sa tento spôsob nedá uplatniť. Namiesto toho sa vyvíjajú všeobecnejšie procesory, ktoré sa neskôr aplikáciám prispôbujú.

## 1.2 Štruktúra prekladačov

Prekladač je program, ktorý zoberie zdrojový kód v istom jazyku a preloží ho na funkčne ekvivalentný kód v inom jazyku [1]. Skladá sa z dvoch častí. Prvá je analýza zdrojového kódu, známa tiež ako front-end a druhá je generovanie cieľového kódu, inými slovami back-end. Väčšina prekladačov obsahuje aj voliteľnú časť optimalizátor, ktorý sa nachádza medzi analýzou a generovaním. Rozdelenie je prevzaté z [1].

Samotná analýza má za úlohu rozobrať vstupný program, skontrolovať jeho syntax a sémantiku a vrátiť ekvivalentný program v medzikóde prekladača. Tiež zbiera potrebné informácie o programe a ukladá ich do tabuľky symbolov. Analýza sa ďalej delí na nasledujúce časti.

Lexikálna analýza. Jej vstupom je zdrojový kód, ktorý prechádza znak po znaku a hľadá v ňom reťazce, ktoré majú v danom jazyku zmysel, ako kľúčové slová, názvy premenných a iné. Nájdene výrazy reprezentuje v podobe tokenov, ktoré sa skladajú z identifikátora a hodnoty. Ak je výrazom premenná, identifikátor lexému môže byť napríklad `id` a jeho hodnotou ukazovateľ do tabuľky symbolov.

Reťazec tokenov ďalej preberá syntaktický analyzátor. Jeho hlavnou úlohou je kontrolovať, či jednotlivé tokeny môžu za sebou nasledovať tak, ako nasledujú. Podľa definovaných pravidiel sa snaží zostaviť abstraktný syntaktický strom, kde uzol predstavuje operáciu a jeho potomkovia argumenty danej operácie. Ak sa podarí reťazec tokenov s neakou jeho časťou, tak syntax tej časti programu je v poriadku.

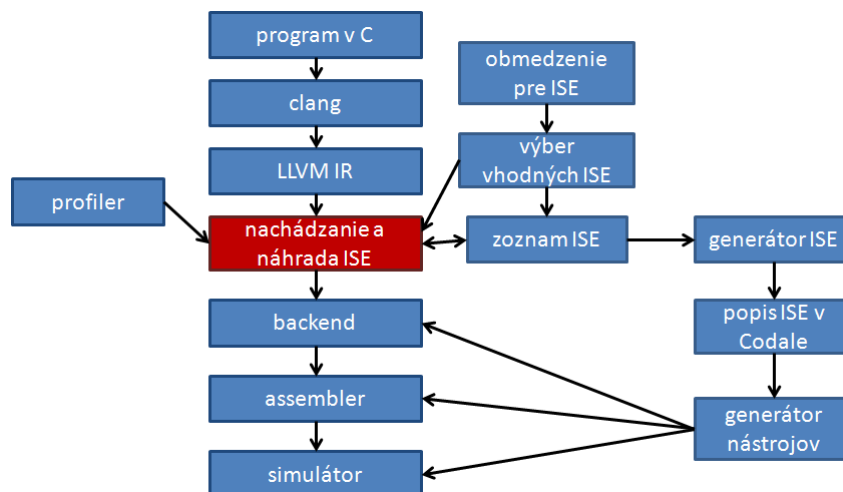
Sémantický analyzátor tiež pracuje s abstraktným syntaktickým stromom a v spolupráci s tabuľkou symbolov kontroluje sémantickú kompatibilitu programu s definíciou jazyka. To zahŕňa napríklad kontrolu definícií a typov premenných.

Poslednou časťou je generátor vnútorného kódu prekladača. Od neho závisí aké optimalizácie sa budú dať na programe vykonať a ako dlho vykonávanie potrvá, alebo ako ľahko sa program preloží do strojového kódu. Existujú mnohé vnútorné reprezentácie, niektoré sú vysokoúrovňové, niektoré nízkoúrovňové, alebo iné [10]. Preto sa môže táto časť v prekladači vyskytovať viackrát a pri každej reprezentácii sa môžu vykonať iné optimalizácie.

Medzikód vstupuje do ďalšej veľkej časti prekladača a tou je optimalizátor. Jeho úlohou je optimalizovať medzikód podľa možností, a prepínačov zadaných pri začiatku kompilácie.

Nakoniec prekladač pomocou generátoru výstupného kódu zostaví program v cieľovom jazyku, ktorý je najčastejšie strojovým kódom.

Drvivá väčšina statických prekladačov je založená na tomto trojfázovom dizajne. Vďaka tomu je jednoduchšie vytvárať prekladače pre ďalšie jazyky, pretože pre rovnakú architektúru stačí vytvoriť nový front-end, taktiež pri podpore novej architektúry treba vytvoriť iba back-end, čo šetrí čas a náklady. Programátori, ktorí sa venujú vývoju jednej časti prekladača, tiež nemusia mať veľké znalosti o jeho iných častiach.



Obr. 1.1: Začlenenie algoritmu na vyhľadávanie ISE do prekladu programu.

V predchádzajúcej časti bol spomenutý problém prispôsobenia sa vstavaných systémov úlohe. Jednou z ciest na urýchlenie návrhu procesorov je používanie popisovacích jazykov. Vďaka nim je možné vytvoriť programy, ktoré automaticky generujú prekladač, linker, assembler a iné nástroje, ktoré sú potrebné na preklad softvéru. Vďaka tomu niesú potrební odborníci na ich vývoj.

Ďalšia možnosť je použiť tzv. retargetable compilers. To sú prekladače, ktoré sú ľahko modifikovateľné, aby produkovali výstupný kód pre stroj s inou inštrukčnou sadou [8]. Výsledný kód je však menej kvalitný a prekladač vie robiť iba tie optimalizácie, ktoré sa dajú robiť pre väčšinu procesorov. Nemôže pri tom využiť jedinečné vlastnosti jednotlivých architektúr a ich inštrukcií.

### 1.3 Začlenenie práce

Algoritmus na získanie ISE bude súčasťou optimalizátoru prekladača. Jeho vstupom ako aj výstupom je medzikód v jazyku LLVM verzia 3.2, ktorý je výstupom front endu prekladača. Pre svoj chod potrebuje poznať obmedzenia architektúry a profil programu, aby vedel, ktoré základné bloky sa najviac oplatí optimalizovať. Generátor ISE pre nájdené rozšírenia vygeneruje popis hardwaru v jazyku CodaL. Výsledok sa potom pridá do programu v podobe pluginov viz 1.1. Nájdené rozšírenia sa premietnu na hradlové pole FPGA.

### 1.4 LLVM IR

Projekt LLVM [6] (Low Level Virtual Machine) vznikol ako výskumný projekt na University of Illinois. Jeho cieľom bolo poskytnúť modernú prekladovú stratégiu, ktorá je schopná podporovať statický aj dynamický preklad ľubovoľných programovacích jazykov.

V dobe, keď LLVM projekt začínal, princíp modularity nebol veľmi využívaný, pretože prekladače boli vytvárané ako monolitické, úzko prepojené masy kódu, ktoré je ťažké diferencovať. V dnešnej dobe predstavuje LLVM kolekciu knižníc a nástrojov s dobre definovanými rozhraniami na vybudovanie jednoduchého prekladača.

Tieto prekladače taktiež používajú trojfázový dizajn. Vďaka modularite front end komunikuje s optimalizátorom iba pomocou medzikódu. Ten je tiež vstupom back endu, čiže jednotlivé časti je možné vyvíjať oddelene a využívať vyššie spomenuté výhody. Optimalizátor pracuje nad medzikódom a skladá sa z rôznych modulov. Programátor si z nich vyberie ktoré potrebuje, podľa prekladaného jazyka a iných kritérií, a len tie použije. Pre projekt LLVM prináša jeho medzikód päť hlavných výhod [7].

- Perzistentné informácie o programe. Prekladač uchováva LLVM podobu programu po dobu života aplikácie, aby bolo možné program optimalizovať aj za behu, či medzi jednotlivými behmi.
- Generovanie kódu offline. Program je možné preložiť do strojového kódu aj v režime offline.
- Profilovanie a optimalizácia založená na používaní programu. Počas behu programu zbiera LLVM framework profilovacie informácie, ktoré môže využiť pri optimalizácií.
- Transparentný model behu programu. Systém nešpecifikuje nejaký konkrétny objektový model, či výnimočnú sémantiku. Tým umožňuje prekladať akýkoľvek jazyk.
- Rovnaká kompilácia celého programu. Jazyková nezávislosť umožňuje optimalizovať a prekladať kód celej aplikácie jednotným spôsobom.

Ako medzikód LLVM prekladače používajú LLVM IR [6] (Internal Representation). Ten je navrhnutý na vykonávanie analýz a transformácií v rámci optimalizácií.

LLVM IR je nízkoúrovňová, inštrukčná sada, ktorá sa používa vo všetkých fázach prekladu LLVM prekladačmi. Snaží sa byť univerzálnym medzikódom, tým že je dostatočne nízkoúrovňový, aby sa naň daly namapovať konštrukcie vysokoúrovňových jazykov. Má veľmi dobre definovanú sémantiku a je podobný RISC, čiže podporuje lineárne sekvencie jednoduchých inštrukcií, ktoré sú reprezentované trojadresným kódom.

Tiež je to `load/store` architektúra, čiže program prenáša dáta medzi registrami a pamäťou pomocou inštrukcií `load` a `store` s pomocou typovaných ukazovateľov.

Na rozdiel od väčšiny RISC inštrukčných sád je LLVM IR silne typovaný jazyk a niektoré detaily stroja sú abstrahované. Napríklad volanie funkcií pomocou inštrukcie `call`, ktorá potrebuje ukazovateľ na funkciu a prípadné argumenty, a návrat z nich pomocou `ret`. Ďalší príklad je, že nepoužíva pevnú sadu pomenovaných registrov. Namiesto toho má nekonečný počet dočasných registrov, ktorých názov začína znakom percento.

LLVM IR je navrhnutý na použitie v troch formách. Ako bytecode, ako pamäťová reprezentácia používaná pri preklade a ako čitateľná reprezentácia assembleru. Všetky tieto formy sú navzájom ekvivalentné a je možné prevádzať kód z jednej formy do druhej. Syntax a sémantiku daného jazyka je možné nájsť na [5].

## Kapitola 2

# Teoretické základy, predstavenie problematiky a návrhy riešenia

Algoritmy na vyhľadávanie inštrukčných rozšírení pracujú nad grafom dátových závislostí. Začiatok kapitoly je preto venovaný definícií a vysvetleniu pojmu graf a pojmov s ním súvisiacich. Tieto pojmy sú potom použité v ďalších častiach kapitoly. Nasledujúca časť obsahuje teoretický popis problému nachádzania ISE. Na záver kapitoly sú predstavené niektoré známejšie algoritmy, ktoré sa v danej oblasti používajú. Všetky pojmy v nasledujúcej časti sú definované podľa [4, 11, 3].

### 2.1 Základné pojmy

Graf dátových závislostí je priamy acyklický graf  $G$ , ktorý je vyjadrený ako dvojica  $G = (V, E)$ .  $V$  predstavuje množinu uzlov, pričom jeden uzol reprezentuje konkrétnu inštrukciu kódu.  $E$  je množina hrán medzi uzlami. Hrany predstavujú dátové závislosti, čiže dáta, ktoré do inštrukcie vstupujú a výsledky inštrukcií. Nachádzanie ISE nad týmto grafom je hľadanie najlepšieho rezu  $S$ . Rez  $S$  je definovaný ako podgraf  $G : S \subseteq G$ . Počet uzlov, ktorých hrany vstupujú do rezu  $S$  označujeme  $IN(S)$  a  $OUT(S)$  označuje počet uzlov, ktorých hrany rez opúšťajú.  $N_{in}$  predstavuje počet vstupných portov do súboru registrov daného procesoru. Podobne, hodnota  $N_{out}$  je počet výstupných portov. Dané hodnoty sa pre rôzne mikroarchitektúry líšia.

Základný blok [8] je maximálna množina za sebou idúcich inštrukcií, do ktorej sa dá vstúpiť iba počiatočnou inštrukciou, žiadne skoky doprostred bloku, a vystúpiť iba koncovou inštrukciou, čiže inštrukcia vetvenia môže byť iba ako posledná inštrukcia bloku.

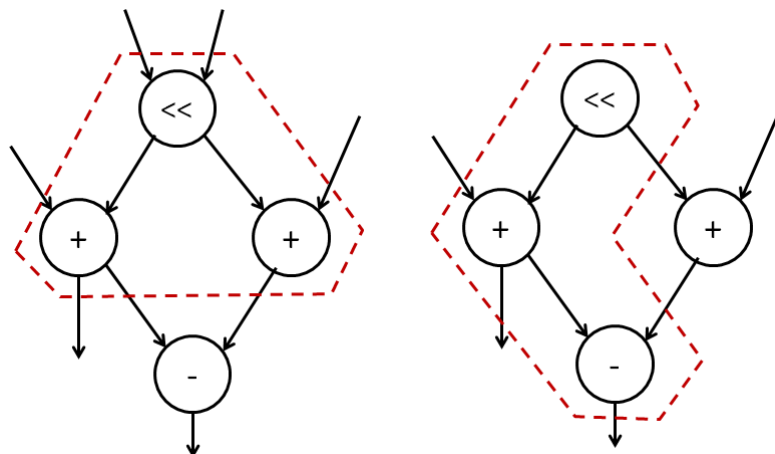
### 2.2 Problémy nachádzania ISE

Nachádzanie inštrukčných rozšírení môžeme formálne vyjadriť pomocou nasledujúcich problémov.

#### Problém 1: Identifikácia

Tento problém zodpovedá nájdeniu jedného inštrukčného rozšírenia nad jedným základným blokom. Teoreticky je vyjadrený ako nájdenie rezu  $S$ , ktorý splňuje nasledujúce pravidlá:

1.  $IN(S) \leq N_{in}$



Obr. 2.1: Ukážka konvexného (vľavo) a nekonvexného (vpravo) rezu.

2.  $OUT(S) \leq N_{out}$
3.  $F \cap S = \emptyset$
4.  $S$  je konvexný

Pravidlo 1 hovorí, že počet vstupov špeciálnej inštrukcie nesmie presiahnuť počet vstupných portov súboru registrov. Inštrukcia totiž potrebuje načítať všetky dáta na jej spracovanie naraz, aby sa mohla vykonať. To by nebolo možné ak by bol počet vstupných portov menší.

Pravidlo 2 je podobné ako predchádzajúce. Rozdiel je iba v tom, že sa jedná o počty výstupov a výstupných portov.

Pravidlo 3 hovorí, že niektoré príkazy sa nemôžu nachádzať v špeciálnej inštrukcii.  $F$  je množina zakázaných uzlov, ktoré tieto príkazy predstavujú pričom  $F \subseteq V$ . Množina najčastejšie obsahuje inštrukcie, ktorých implementácia v hardvéri by nebola pre výkon prínosná, alebo je pre návrh nežiadúca. Napríklad ak sú pamäťové porty vo funkčných blokoch nežiadúce, patria inštrukcie pracujúce s pamäťou ako `load` a `store` do množiny  $F$ .

Posledné pravidlo 4 zaručuje konvexnosť rezu. Rez je konvexný ak v grafe dátových závislostí neexistuje cesta medzi dvoma uzlami v reze  $S$ , ktorá obsahuje uzol, ktorý rezu  $S$  nepatrí. Inak by totiž nebolo možné naplánovať poradie inštrukcií tak, aby sa špeciálna inštrukcia vykonala naraz a mohla byť implementovaná ako jeden funkčný blok. Najprv by sa vykonala jej prvá časť, potom by sa musela vykonať iná inštrukcia a až potom by sa mohlo dokončiť vykonávanie špeciálnej inštrukcie. Príklady konvexného a nekonvexného rezu sú zobrazené a obrázku 2.1. Riešením problému 1 sú niektoré algoritmy na nachádzanie ISE, napríklad `single-cut`.

### Problém 2: Selekcia

Pre čo najväčšie zrýchlenie nám nestačí nájsť jedno rozšírenie. Chceme prehľadávať všetky základné bloky, a ak je možné tak aj niekoľkokrát. Túto myšlienku vyjadruje problém 2 nasledovne.

Nad grafmi  $G_i^+$  všetkých základných blokov s vlastnosťami mikroarchitektúry  $N_{in}, N_{out}$  a  $F$  nájsť maximálne  $N_{instr}$  rezov  $S_j$ , ktorých prínos je čo najväčší.  $N_{ise}$  je maximálny počet inštrukcií, ktoré je možné alebo žiadúce v danej mikroarchitektúre implementovať.

Podľa [11] riešenie oboch problémov môžeme zovšeobecniť do troch krokov.

1. Nájdenie jedného inštrukčného rozšírenia nad jedným základným blokom.
2. Nájst množinu  $N$  neprekrývajúcich sa rezov nad jedným základným blokom.
3. Nájdenie množiny  $N$  nad viacerými základnými blokmi.

Krok 1 zodpovedá problému 1 a jeho riešením sú jednotlivé algoritmy ako napríklad *single-cut*. Posledné dva kroky zodpovedajú problému 2. Jeho riešením môže byť opakovanie algoritmu, ktorý rieši problém 1 niekoľkokrát a nad viacerými základnými blokmi. Daný prístup môže byť značne časovo náročný, preto existujú rôzne heuristiky, ktoré časovú náročnosť znižujú, napríklad algoritmus *iterativeselection*.

## 2.3 Používané algoritmy

### 2.3.1 MaxMiso

Algoritmus MaxMiso, ako uvádza [2], pracuje nad grafom dátových závislostí, v ktorom sa snaží nájsť rez  $S$ . V tomto prípade je  $S$  spojitý podgraf typu MISO. Skratka MISO znamená Multiple Input Single Output, v preklade: viacero vstupov, jeden výstup. Z toho vyplýva, že hoci hodnota rezu  $IN(S)$  obmedzená nie je, tak pre hodnotu  $OUT(S)$  platí  $OUT(S) = 1$ . Názov MaxMiso naznačuje, že algoritmus sa snaží nájsť čo najväčší rez  $S$ , pričom MaxMiso je definovaný ako graf MISO, ktorý nie je úplne zahrnutý v žiadnom inom MISO grafe. V rozšírenej verzii práce [2] je dokázané, že dva MaxMiso grafy sa nemôžu čiastočne prekrývať. Preto potom, čo je nájdený MaxMiso rez, sú jeho uzly vyradené z množiny prehľadávaných uzlov a algoritmus hľadá ďalší rez. Daná vlastnosť je základ pre lineárne časovo náročný algoritmus, pričom náročnosť závisí na počtu inštrukcií v základnom bloku.

Najväčšou nevýhodou MaxMiso algoritmu je obmedzenie počtu výstupov rezu  $S$ . To obmedzuje veľkosť rezu a tým je dosiahnuté zrýchlenie menšie ako pri iných prístupoch. Naopak lineárna zložitosť je veľkou výhodou.

### 2.3.2 Single-cut

Algoritmus single-cut pracuje podobne ako MaxMiso. Snaží sa nájsť čo najvhodnejší rez, ktorý rešpektuje pravidlá v sekcii 2.2. Na rozdiel od MaxMiso nemá pevne zadané obmedzenie na hodnotu  $OUT(S)$ . Dané hodnoty závisia iba na parametroch mikroarchitektúry  $N_{in}$  a  $N_{out}$ . To mu umožňuje nachádzať väčšie rezy. Algoritmus do rezu postupne pridáva inštrukcie a potom skúma, či sa daná inštrukcia bude v rez nachádzať alebo nie. V Najhoršom prípade preskúma všetky možné kombinácie inštrukcií. Časová zložitosť je preto exponenciálna. Daný problém výrazne znižujú vyššie spomenuté pravidlá. Zložitosť môžeme vyjadriť ako  $2^{|V|}$ , pričom  $|V|$  je počet uzlov. Princíp algoritmu je podrobnejšie vysvetlený v kapitole 3.

### 2.3.3 ISEGEN

Algoritmus ISEGEN [3] pracuje trochu inak ako predchádzajúce postupy. Vykonáva rozdeľovanie inštrukcií na hardvérové a softvérové. Hardvérové sú hľadané ISE a budú sa vykonávať na špeciálnych funkčných blokoch, zatiaľ čo softvérové bude vykonávať jadro

procesoru. Samotné rozdeľovanie inštrukcií má na starosť prírastková funkcia, ktorá vypočíta odhadovaný zisk ak by sa inštrukcia vykonala v hardvéri. Hlavná myšlienka riadenia preklápania inštrukcií je prevzatá z Kerninghan-Lin min-cut partitioning heuristiky.

Algoritmus začína cyklom, v ktorom najprv aktualizuje najlepší rez, ktorý je na začiatku prázdny. Potom sa zanorený do ďalšieho cyklu, ktorý prechádza jednotlivé uzly, až pokiaľ nie sú všetky označené. Pre každý uzol grafu tak vypočíta zrýchlenie, ktoré by program dosiahol, keby bola inštrukcia vykonaná v hardvéri. Výpočet zrýchlenia má na starosti prírastková funkcia. ISEGEN porovná výsledné hodnoty a vyberie inštrukciu s najväčším prírastkom zrýchlenia, ktorú preklopí a označí. Operácia preklopenia znamená, že ak bola inštrukcia plánovaná ako softvérová, bude patriť medzi hardvérové a naopak. Ďalej musí algoritmus skontrolovať, či rez spĺňa dané podmienky. Ak áno, upraví najlepší rez tým, že doňho pridá, alebo z neho vyjme daný uzol podľa toho, ako dopadla operácia preklopenia. Tým skončí vnútorný cyklus. Potom algoritmus zisťuje, či je nový rez lepší ako predošlý a to pomocou funkcie na odhad zrýchlenia. Víťaz sa stane najlepším rezom. Posledným krokom je odznačenie všetkých uzlov, aby sa mohol vnútorný cyklus opakovať. Potom skončí aj vonkajší cyklus.

Samotná prírastková funkcia je váženým súčtom piatich faktorov:

- Odhadované zrýchlenie. Ak rez nespĺňa podmienku konvexnosti, je hodnota faktoru  $-\infty$ .
- Porušenie vstupno-výstupných podmienok. Hodnota faktoru závisí na tom ako veľmi rez prekračuje povolené hodnoty
- Obmedzenie konvexnosti. Čím viac susedov uzlu je v reze prítomných, tým viac je žiadúce, aby do rezu patril aj daný uzol.
- Veľkosť rezu. Rozvíja rez v tých smeroch v ktorých má lepšie podmienky na rast.
- Nezávislé rezy. Najlepším rezom nemusí byť jeden spojený podgraf, ale môže ním byť viacero medzi sebou nespojených podgrafov, tak prehľadávanie musí expandovať aj v horizontálnom smere.

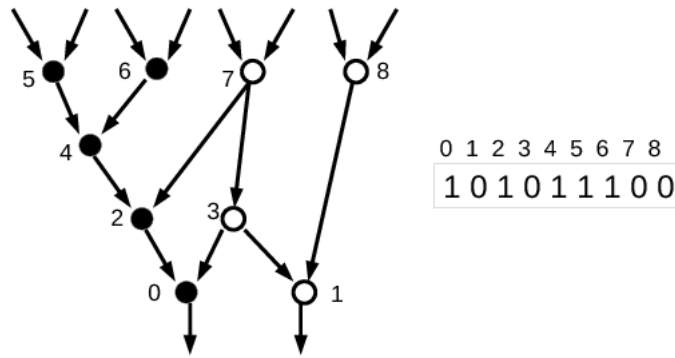
Jednotlivé váhy sú konštanty, ktorých najvhodnejšia hodnota bola zistená pokusmi.

Algoritmus ISEGEN kladie omnoho väčší dôraz na prírastkovú funkciu a odhadované zrýchlenie ako ostatné postupy. Tiež umožňuje nachádzanie ISE aj nad nespojitými podgrafmi. Podľa [3] je časová zložitosť v najhoršom prípade  $o(|V| * |E|)$ , kde  $|V|$  je počet uzlov a  $|E|$  je počet hrán. Algoritmus produkuje výsledky, ktoré sú veľmi podobné tým, čo našli experti pri manuálnom vyhľadávaní.

### 2.3.4 Genetické algoritmy

Sú to stochastické algoritmy, ktoré aplikujú princípy genetickej dedičnosti z prírody, kde prežije iba najschopnejší jedinec [9]. Používajú sa pri optimalizácii cieľovej funkcie, pričom o nej nemáme veľa údajov. Nemusíme dokonca vedieť či je funkcia spojitá, alebo ako sa na množine vstupných hodnôt správa. Majú niekoľko výhod, napríklad pri viacrozmerom prehľadávaní, kde existuje viacero lokálnych optimálnych riešení, je veľmi pravdepodobné, že genetické algoritmy vyberú globálne optimálne riešenie. Algoritmus, počas svojej činnosti, udržiava množinu najoptimálnejších riešení, pričom tie nevhodné postupne vymierajú.

Keďže nachádzanie ISE prebieha nad grafom dátových ciest, je potrebné tento graf nejako prezentovať. Najčastejšie pomocou poľa bitov, kde je každý uzol vyjadrený jedným



Obr. 2.2: Ukážka kódovania grafu v genetickom algoritme.

bitom. Indexi poľa predstavujú poradie uzlov v topologicky usporiadanom grafe a hodnota bitu určuje, či daný uzol patrí do rezu alebo nie. Obrázok 2.2, prevzatý zo [11] z obsahuje ukážku danej reprezentácie.

Predstavený postup je prevzatý z [11] Na začiatku algoritmus vygeneruje nejakú počiatočnú populáciu. Na to môže použiť rôzne heuristiky. Tento postup konkrétne používa algoritmus, ktorý na začiatku vyberie konvexné zhľuky z náhodne vybraných uzlov a pridáva príslušné uzly, pokiaľ sú splnené podmienky zo sekcie 2.2. Tento prístup je kombinovaný s MaxMiso spomenutým vyššie.

Ďalším krokom je vyhodnotenie populácie. Pre tento účel sa používa špeciálna funkcia na vyhodnotenie spôsobilosti, tzv. fitness funkcia. Tá sa veľmi podobá na funkciu, ktorú optimalizuje, v tomto prípade  $M(S)$ . Fitness hodnota rezu  $S$ , ktorý splňuje všetky podmienky, je rovná hodnote cieľovej funkcie  $M(S)$ . Keďže grafy dátových závislostí často obsahujú veľké zhľuky, ktoré majú zopár vstupov, veľký počet medzivýsledkov a niekoľko výstupov, dovoľuje táto implementácia aj také rezy, ktoré porušujú vstupné a výstupné obmedzenia. Rezy tým majú možnosť zmenšiť sa alebo dorásť do požadovaných podmienok. Porušenie podmienok sa ale trestá určitým prírastkom vo fitness funkcií.

Tak, ako v prírode prežijú tí najschopnejší, vyberie algoritmus, podľa výsledkov, tie najlepšie riešenia, zatiaľ čo tí slabí vymrú.

Na vybrané uzly sa aplikujú genetické operácie kríženia a mutácie. Kríženie je zamieňanie uzlov z rôznych základných blokov, na dosiahnutie lepšieho rezu. Prakticky sa časť reťazca, ktorý označuje prítomnosť uzlov v reze, prehodí s tou istou časťou reťazca iného rezu. Mutácie zase spôsobujú náhodné zmeny jednotlivých bitov v reťazci. Každý bit sa podľa pravdepodobnosti intenzity mutácie preklopí na opačnú hodnotu.

Použitie daných operácií môže vytvoriť rezy, ktoré nespĺňajú potrebné podmienky. Preto ich program musí skontrolovať. Prekročenie počtu vstupov a výstupov upravuje fitness funkcia.

Pretože genetické algoritmy konvergujú rýchlo, operácia kríženia stráca svoju efektivitu. Tomuto zabráňuje výmena jedincov. Operácia výmeny vyberie niektorých jedincov populácie a nahradí ich novými, ktorý sú generovaný spôsobom ako počiatočná populácia.

Všetky operácie, okrem generovania počiatočnej populácie sa v algoritme niekoľkokrát opakujú. Výhodou tohto prístupu je, že rieši prvý aj druhý problém zo sekcie 2.2 ako aj jeho rýchla konvergencia a schopnosť nájsť kvalitné riešenia. Na druhej strane, jeho implementácia je o poznanie zložitejšia a je v nej zahrnuté množstvo iných algoritmov.

## Kapitola 3

# Praktická ukážka algoritmu single-cut

Táto bakalárska práca sa zaoberá implementáciou algoritmu single-cut. Preto je v kapitole algoritmus podrobne popísaný, ako teoreticky, tak aj prakticky na jednoduchom základnom bloku.

### 3.1 základný blok v LLVM IR

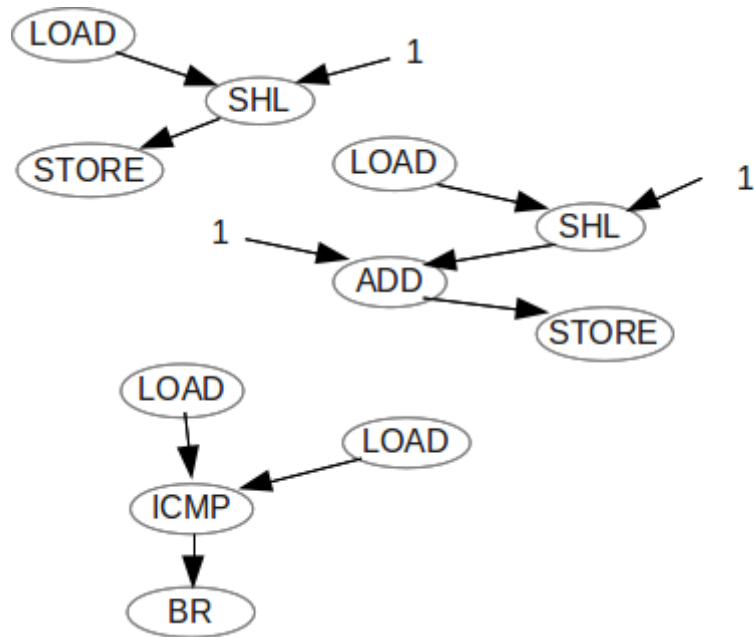
Vybraný blok *for.body* z programu *isqrt* vyzerá v LLVM IR nasledovne:

```
1  %4 = load i32* %a, align 4
2  %sh12 = shl i32 %4, 1
3  store i32 %sh12, i32* %a, align 4
4  %5 = load i32* %a, align 4
5  %sh13 = shl i32 %5, 1
6  %add4 = add i32 %sh13, 1
7  store i32 %add4, i32* %e, align 4
8  %6 = load i32* %r, align 4
9  %7 = load i32* %e, align 4
10 %cmp5 = icmp uge i32 %6, %7
11 br i1 %cmp5, label %if.then, label %if.end
```

Daný kúsok kódu najprv načíta do pomocnej premennej hodnotu z pamäte uloženú na adrese zadanej hodnotou *a*. Potom do premennej *sh12* priradí túto hodnotu bitovo posunutú doľava o jeden bit. Na riadku 3 získanú hodnotu zase zapíše na to isté miesto v pamäti, špecifikované hodnotou *a*. Následne sa načítanie a bitový posun opakuje, navyše na riadku 6 pripočíta k posunutému číslu jednotku. Výsledok uloží do pamäte. Nakoniec načíta dve hodnoty, ktoré na riadku 10 porovná a zistí, či je jedna väčšia, alebo rovná ako druhá. Podľa výsledku skočí na ďalšiu časť programu. K danému kódu je potrebné vytvoriť graf dátových závislostí, ktorý je zobrazený na obrázku [3.1](#).

### 3.2 Teória algoritmu

Algoritmus single-cut identification [11, 1] pre výber inštrukčných rozšírení pracuje nad binárnym stromom, kde uzly reprezentujú jednotlivé inštrukcie a ich hodnota to, či patria do



Obr. 3.1: Graf dátových závislostí

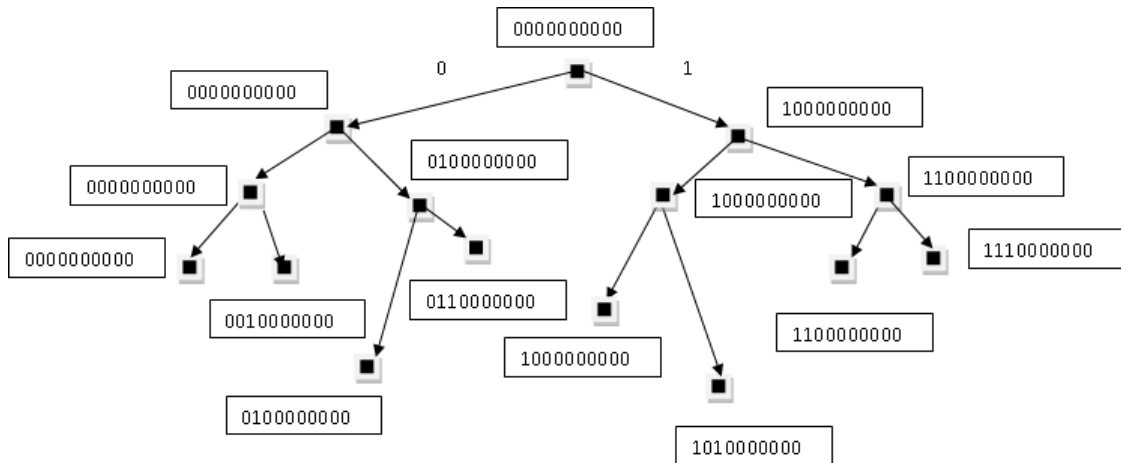
špeciálnej inštrukcie. Koreň stromu predstavuje prázdnu množinu (nebola vybraná žiadna inštrukcia na vytvorenie ISE). Každý ďalší level je jedna inštrukcia podľa topologického zoradenia. Hrany stromu môžu mať hodnotu 0 alebo 1. Uzol, ktorý je na hrane 1 predstavuje možnosť, že inštrukcia do rezu patrí, 0 znamená opak. Hodnota uzlu znázorňuje pole cut z algoritmu. Algoritmus postupne prechádzame program podľa topologického usporiadania inštrukcii zospodu a jednotlivé príkazy pridáva do špeciálnej inštrukcie. Časť binárneho stromu tohto príkladu by mohol vyzeráť tak, ako na obr 3.2.

Sám algoritmus ma pseudokód:

```

1  identification() {
2      for (i = 0; i < NODES; i++) cut[i] = 0;
3      topological_sort();
4      search(1, 0);
5      search(0, 0);
6  }
7
8  search(current_choice, current_index) {
9      cut[current_index] = current_choice;
10     if (current_choice == 1) {
11         if (forbidden()) return;
12         if (!output_port_check()) return;
13         if (!permanent_input_port_check()) return;
14         if (!convexity_check()) return;
15         if (input_port_check()) {
16             calculate_speedup();
17             update_best_solution();
18         }

```



Obr. 3.2: binárny strom reprezentujúci výber inštrukcií

```

19     }
20     if ((current_index + 1) == NODES) return;
21     current_index = current_index + 1;
22     search (1, current_index);
23     search (0, current_index);
24 }

```

Program začne funkciou `identification` v ktorom inicializuje pole `cut`. Pole `cut` predstavuje jednotlivé inštrukcie, topologicky zoradené zospodu, grafu dátových závislostí. Ak je na istom indexe hodnota jedna, znamená to, že inštrukcia na danej pozícii bude zaradená do špeciálnej inštrukcie. Potom topologicky zoradí jednotlivé inštrukcie z grafu dátových ciest. Následne začne prehľadávať podstromy. Najprv na vetvu jedna potom vetvu nula. Funkcia `search` prijíma parametre `current_choice`, ktorý predstavuje hodnotu vetvy, po ktorej program pokračuje a `current_index`, ktorý označuje poradie pridávanej inštrukcie a level zanorenia binárneho stromu.

Ak chce uzol pridať, skontroluje všetky potrebné podmienky. Ak rez niektorú nespĺňa, tak funkcia končí a pri jej ďalšom spustení sa na index príslušnej inštrukcie priradí hodnota 0. Algoritmus rozlišuje permanentné a dočasné vstupy. Permanentné sú tie, ktoré budú v reze figurovať vždy, napríklad vstupy z iného základného bloku. Dočasné vstupy sú tie, ktorých počet sa môže zmeniť pridaním ďalšej inštrukcie. Ak je porušená podmienka počtu vstupov (dočasných a permanentných spolu), tak funkcia `search` nekončí hneď, ale pokračuje v prehľadávaní potomka uzlu, hoci hodnotu zatiaľ nájdeného rezu neupraví. Inak program vypočíta zrýchlenie a aktualizuje hodnotu najlepšieho rezu. Nakoniec zistí, či má uzol potomkov a ak má, tak rekurzívne volá seba na prehľadávanie detí. To sa opakuje pokiaľ neprehľadá všetky uzly. Tento prístup má vďaka tomu exponenciálnu časovú zložitosť, ktorú môžeme vyjadriť ako  $2^{|V|}$ , pričom  $|V|$  je počet uzlov stromu. Prax však ukázala, že podmienky pomáhajú strom výrazne prerezať a tým zložitosť znížiť.

### 3.3 Praktická ukážka

V tomto prípade bude hodnota `NODES = 11`. Funkcia `search` prijíma parametre `current_choice`, ktorý má na začiatku hodnotu 1 a `current_index`, s hodnotou 0. Do poľa `cut` na index, ktorý sa rovná hodnote zanorenia v binárnom strome, pri prvom priechode má hodnotu 0, sa pridá hodnota vetvy, po ktorej program pokračuje. Obmedzenie na počet vstupných a výstupných hodnôt môžeme pre tento prípad zvoliť na 4 a 2.

Keďže je hodnota premennej `current_choice` rovná 1, program ďalej testuje vhodnosť inštrukcie `BR`. Daná inštrukcia patrí medzi zakázané, preto funkcia `search` pri kontrole `forbidden()` skončí. Následne sa spustí znovu s parametrami 0 a 0. Vtedy sa hodnota v poli `cut` na indexe 0 prepíše z 1 na 0. Potom sa zvýši hodnota `current_index` a keďže uzol nieje koncovým uzlom stromu, spustí sa funkcia `search` najprv pre jeho pravý, potom pre ľavý podstrom.

Nové parametre funkcie pri preskúvaní inštrukcie `ICMP` sú 1 a 1. Do poľa `cut` sa na index 1 zapíše hodnota 1. Potom prechádza nový rez kontrolami. Inštrukcia `ICMP` nepatrí medzi zakázané a jej pridaním bude rez naďalej konvexný. Počet výstupných portov rezu bude 1, počet trvalých vstupných portov 0 a počet vstupných portov 2, čím neporuší žiadnu podmienku. Program následne spočíta predpokladané zrýchlenie a aktualizuje hodnotu najlepšieho riešenia. Index sa preto zvýši o jedna a program bude pokračovať funkciou `search` s indexami 1 a 2.

Pridanie inštrukcií `LOAD` bude prebiehať podobne. Keďže v grafe dátových závislostí nemajú predchodcov, hodnota vstupných portov rezu sa bude iba znižovať. Oba uzly budú pridané.

Program postúpi na ďalší podgraf. Funkcia `search` začína s parametrami 1 a 4. Inštrukcia `STORE`, podobne ako ostatné, nepatrí medzi zakázané a keďže nepridáva výstupný port a počet vstupných portov je 1. Bude k rezu pridaná.

Pri pridávaní inštrukcie `AND` sa zmení počet permanentných vstupov na 1, lebo `AND` má ako jeden operand konštantu. Stále však splňuje limity a preto bude do rezu pridaná.

Nové parametre funkcie pri pridávaní inštrukcie `SHL` sú 1 a 6. Pri kontrole sa zistí, že počet výstupných portov je 1, počet permanentných vstupných portov 2 a obyčajných vstupných portov 3. Podmienka konvexnosti porušená nieje a uzol tiež nepatrí medzi zakázané inštrukcie, takže bude do rezu pridaný.

Preskúvanie inštrukcií `LOAD` a `STORE` neprinesie nič nové a inštrukcie budú bez problémov do rezu zaradené.

Pri pridávaní inštrukcie `SHL` sa hodnota počtu vstupných portov zvýši na číslo 4. Ak by bola hranica počtu vstupných portov nastavená na 3, bola by podmienka porušená. V tom prípade by funkcia neskončila, iba by sa neaktualizoval najlepší výsledok na hodnotu znázorňujúcu tento rez. V poli `cut` by na indexe 9 nebola hodnota 0 ale 1 pričom v poli reprezentujúcom najlepší rez, by na tom istom indexe bola hodnota 0. Program by pokračoval ďalej, akoby bol uzol pridaný. Keďže počet vstupných portov je obmedzený na 4, daný prípad nenastane a uzol je do rezu pridaný.

Posledný uzol `LOAD` je tiež pridaný. Hodnota `current_index` je v tejto chvíli 10 a keďže je splnená podmienka  $(\text{current\_index} + 1) == \text{NODES}$ , tak funkcia skončí a spustí sa `search` pre parametre 0 a 10. Ani tam sa neudeje nič mimoriadne a po jej skončení nasleduje `search` pre parametre 0 a 9. Takto sa bude funkcia postupne vracaať, až sa dostane zase na začiatok. Najlepší rez, ktorý našla je reprezentovaný polom s hodnotami 0111111111.

## Kapitola 4

# Implementácia algoritmu

Táto kapitola obsahuje popis rozdelenia programu na moduly, ich implementáciu a funkciu, ktorú v programe zastávajú. Program bol implementovaný v jazyku C++. Hlavným dôvodom pre výber daného jazyka bolo, že program používa knižnice LLVM, ktoré sú v ňom implementované.

Samotný program sa skladá z troch hlavných modulov: `ise_pass`, `ise_algorithm` a `function_finish`. Jednotlivé moduly sa skladajú z funkcií, ktoré sú si blízke tým, že sa používajú v rovnakých častiach programu. Keďže je program iba ďalším prechodom LLVM optimalizátora nad kódom v jazyku LLVM IR, je na jeho spustenie potrebný optimalizátor, ktorý treba spustiť so správnymi parametrami ako cesta ku knižnici či názov prechodu.

### 4.1 Základný modul `ise_pass`

Tento modul je základom programu. Obsahuje funkciu `main`, takže program ním začína. Má dve hlavné úlohy.

Prvou je, po načítaní súboru zistiť, koľkokrát sa ktorý základný blok vykoná. Na to používa profilovacie informácie, zistené pri simulácii programu, ktoré načíta zo súboru. Podľa toho bloky zoradí od najpoužívanejšieho po najmenej používaný blok. Následne funkcia spracuje argumenty programu a podľa nich začne jednotlivé bloky spracúvať, čiže prvýkrát zavolá funkciu `search`. Modul obsahuje iba jej volanie, nie samotnú implementáciu.

Druhou hlavnou úlohou modulu je vytvorenie novej funkcie, ktorá reprezentuje funkčný blok v hardvéri. Prvý problém, ktorý sa s touto úlohou spája je zistenie parametrov a návratových hodnôt novej funkcie. K tomu sa používajú pomocné funkcie z iného modulu. Ak program pozná dátové typy všetkých parametrov a návratových hodnôt, je možné vytvoriť novú funkciu. Jej naplneniu a dokončeniu sa zase venuje iný modul. Tento iba volá jeho funkcie s potrebnými parametrami. Hlavnú pozornosť venuje modul zase odstráneniu inštrukcií, ktoré sú v reze a prepojenie novej funkcie so základným blokom. To znamená vytvoriť a správne umiestniť volanie novej funkcie. Nájsť správne premenné a použiť ich ako parametre novej funkcie. Nájsť a prepojiť výsledok funkcie s operáciami, ktoré ho očakávajú ako parameter.

Tieto úlohy sú implementované ako prechádzanie základného bloku, pri ktorom modul používa pomocnú funkciu na zistenie predkov a potomkov inštrukcie z grafu dátových závislostí a metód nastavujúcich parametre z knižnice LLVM. Ak funkcia vracia viacej výsledkov v podobe štruktúry, vytvorí program inštrukcie `extractvalue`, aby mohol k jednotlivým hodnotám pristupovať. V takom prípade je však často nutné inštrukcie v bloku topologicky

usporiadať. To sa vykoná zase pomocou inej funkcie z iného modulu.

## 4.2 Modul `ise_algorithm`

Hlavnou úlohou tohto modulu je implementácia funkcie `search`, ktorá je základom algoritmu `single-cut`. Funkcia je implementovaná tak, ako je opísaná v časti 3.2. Navyiac je však možné, pomocou parametrov programu, meniť maximálny počet vstupov a výstupov rezu. Je tiež možné hľadať aj rezy bez týchto obmedzení a nastaviť inštrukcie `LOAD` a `STORE` ako nežiadúce. Obsahom modulu sú aj funkcie, ktoré `search` využíva. To sú `isForbidden`, `isConvex`, `numberOfOutputs`, `numberOfInputs` a `numberOfPermanentOutputs`. Každá z nich kontroluje určitú vlastnosť rezu.

Najpoužívanejšími funkciami sú však `getAncestor` a `getPredecessor`, ktoré zisťujú predkov a potomkov uzlu, podľa grafu dátových závislostí. Sú to základné kamene vyššie spomenutých funkcií, ale používajú sa aj pri vytváraní novej ISE funkcie. Ich implementácia je založená na postupnom prechádzaní základného bloku a volaní metód nad inštrukciami z LLVM knižnice.

Keďže rez je reprezentovaný ako pole `int` hodnôt a základný blok ako kontajner inštrukcií, je občas treba previesť index poľa na konkrétnu inštrukciu. To majú na starosti funkcie `getInstruction` a `getTsePosition`.

## 4.3 Modul `function_finish`

Posledný modul obsahuje funkcie, ktorých hlavnou úlohou je vytvorenie novej funkcie reprezentujúcej ISE inštrukciu. Patrí tam funkcia `getParams` na získanie parametrov novej funkcie ako aj `getReturns` s podobným účinkom pre návratové hodnoty. Podľa jej výsledkov sa tiež určuje, či funkcia vracia iba jednu hodnotu alebo štruktúru.

Funkcia `createReturn` vytvára `ret` inštrukciu novej funkcie. Rozlišuje, či funkcia vracia `void`, jednu hodnotu, alebo štruktúru. Ak vracia štruktúru, tak vytvorí inštrukcie `insertvalue` na jej naplnenie a potom štruktúru vráti.

Keďže inštrukcie z pôvodného základného bloku sú do nového kopírované, tak dočasné premenné, do ktorých sa ukladajú výsledky inštrukcií, majú iný názov ako dočasné premenné, ktoré sú operandy týchto inštrukcií. Inštrukcie treba vzájomne poprepájať. To robia funkcie `bindParams` a `bindBasicBlock`.

Poslednou funkciou je `topologicalSort`. Zavolá sa iba v prípade, že nová ISE funkcia vracia štruktúru. Jej úlohou je usporiadať inštrukcie v pôvodnom základnom bloku tak, aby boli pred volaním novej funkcie dostupné všetky jej parametre a jej výsledok aby sa používal až po jej volaní. Funkcia sa vykoná pred prepojením pôvodného bloku s novou ISE funkciou.

Základom všetkých vymenovaných funkcií sú vyššie spomenuté funkcie `getAncestor` a `getPredecessor`.

## Kapitola 5

# Testovanie programu a jeho výsledky

Táto kapitola obsahuje opis testovania výsledného programu. Testovanie prebieha tak, že sa program odsimuluje a meria sa jeho rýchlosť. Následne sa vykoná nachádzanie inštrukcií v tomto programe, pre ktoré sa v popise procesoru vygenerujú hardvérové prvky. Potom sa program simuluje znovu s nahradenými prvkami a výsledky sa porovnávajú.

### 5.1 Výsledky testov

Tabuľka 5.1 obsahuje počet cyklov hodín procesoru pri spustení algoritmu, nad testovacími súborami s parametrom `-st/ld`. Tabuľka 5.2 zase obsahuje dĺžku simulácie. Z uvedených výsledkov je vidieť, že program sa niekoľkonásobne zrýchlil po pridaní špeciálnych inštrukcií. V prípade programu `crc.c` sa čas trvania skrátil takmer štvornásobne.

Iné príklady sa zatiaľ nepodarilo úspešne otestovať pre menšiu chybu v prechode, tá bude však zanedlho odstránená.

	<b>crc.c</b>	<b>quicksort.c</b>
bez ISE	16400071	11927
s ISE	4100059	7441

Tabuľka 5.1: Tabuľka počtov hodinových cyklov procesoru

	<b>crc.c</b>	<b>quicksort.c</b>
bez ISE	6.18 s	5 ms
s ISE	1.89 s	4 ms

Tabuľka 5.2: Tabuľka časov trvania simulácií

### 5.2 Nedostatky programu

Pri testovaní programu sa odhalili niektoré jeho nedostatky. Medzi ne patrí neschopnosť programu spracovať špeciálne inštrukcie, ktoré sa nachádzajú na dvoch nespojených grafoch

dátových závislostí. Hoci bol program pôvodne navrhnutý na spracovanie aj týchto udalostí a dokáže vygenerovať funkciu vracajúcu štruktúru, pri simulácii sa takýto program neukončí a niekde sa zacyklí. Pravdepodobná príčina bude v nesprávnom preberaní výsledkov zo štruktúry v pôvodnom bloku.

Ďalším problémom je príliš dlhé vyhľadávanie špeciálnych inštrukcií v programe `sha.c`. Ten totiž obsahuje veľké základné bloky, ktoré sú pre algoritmus `single-cut` prekážkou, pre jeho exponenciálnu zložitosť. Tomu neprispieva ani kontrola konvexnosti rezu, ktorá musí preskúmať veľké množstvo uzlov.

Tiež z testovacieho programu `isqrt.c` sa nedalo vygenerovať nijaké rozšírenie, lebo obsahuje 8-bitové inštrukcie `LOAD` a `STORE`.

# Kapitola 6

## Záver

Výsledkom tejto bakalárskej práce je prechod optimalizátoru LLVM prekladača, ktorý identifikuje inštrukčné rozšírenia pomocou algoritmu *single – cut* a reprezentuje ich v jazyku LLVM IR pomocou nových funkcií. Tie sú potom implementované hardvérovo, čo zrýchli vykonávanie programu.

Výber inštrukcií je do určitej miery možné ovplyvniť pomocou parametrov. Tie ovplyvňujú minimálnu veľkosť novej inštrukcie. Umožňujú tiež zadávať vlastnosti mikroarchitektúry procesoru, ktoré sú pre algoritmus potrebné. Priechod bol implementovaný aj s podporou výberu nad nespojitými grafmi dátových závislostí. Testovanie programu však ukázalo, že táto vlastnosť ešte nieje plne funkčná. Pre dosiahnutie čo najväčšieho zrýchlenia, sa základné bloky, nad ktorými sa vyhľadávanie ISE vykoná, nevyberajú manuálne. Namiesto toho sú zoradené podľa počtu ich použití pri simulácii. Preto je potrebné program pred priechodom odsimulovať a získať jeho profil.

Prechod bol vytvorený v jazyku C++ s použitím LLVM knižníc. Na jeho vývoj nebol použitý žiaden framework, iba nástroje LLVM prekladača potrebné na jeho odtestovanie.

Samotné testovanie ukázalo, že nový program dosiahol zrýchlenie výkonu oproti jeho predchádzajúcej verzii. Preto sa dá domnievať, že tento postup je použiteľný v praxi. Dosiahnuté výsledky by bolo zaujímavé porovnať s inými prácami, ktorá rieši rovnakú úlohu, avšak na identifikáciu rozšírení využívajú rozdielny algoritmus.

Ako ďalší krok vo vývoji programu by bolo vhodné opraviť vyhľadávanie inštrukčných rozšírení nad viacerými grafmi dátových závislostí. Tým by sa rozhodne dosiahlo ďalšieho zrýchlenia výsledného programu. Tiež by bolo vhodné pretransformovať rekurzívne volanie funkcie `search` na iteráciu, aby sa pri skúmaní veľkých blokov zbytočne nezaplňoval zásobník programu.

# Literatúra

- [1] Aho, A.: *Compilers: Principles, Techniques, & Tools*. Pearson international edition, Addison Wesley, 2007, ISBN 9780321491695.
- [2] Alippi, C.; Fornaciari, W.; Pozzi, L.; aj.: A DAG-based design approach for reconfigurable VLIW processors. In *Proceedings of the conference on Design, automation and test in Europe*, ACM, 1999, str. 57.
- [3] Biswas, P.; Banerjee, S.; Dutt, N.; aj.: ISEGEN: generation of high-quality instruction set extensions by iterative improvement. In *Design, Automation and Test in Europe, 2005. Proceedings*, 2005, ISSN 1530-1591, s. 1246 –1251 Vol. 2, doi:10.1109/DATE.2005.191.
- [4] Ienne, P.; Leupers, R.: *Customizable Embedded Processors*. Morgan Kaufman Publishers, 2007, iISBN 0-12-369526-0.
- [5] Infrastructure, T. L. C.: LLVM Language Reference Manual [online]. <http://llvm.org/docs/LangRef.html#llvm-language-reference-manual>, 2013-03-11 [cit. 2013-3-4].
- [6] Lattner, C.: LLVM [online]. <http://www.aosabook.org/en/llvm.html>, 2012-07-7 [cit. 2013-3-4].
- [7] Lattner, C.; Adve, V.: LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.
- [8] Leupers, R.; Marwedel, P.: *Retargetable compiler technology for embedded systems: tools and applications*. Norwell, MA, USA: Kluwer Academic Publishers, 2001, ISBN 0-7923-7578-5.
- [9] Luren, P.: Jemný úvod do genetických algoritmů [online]. <http://cgg.mff.cuni.cz/pepca/prg022/luner.html>, 2005-7-26 [cit. 2013-3-4].
- [10] Muchnick, S. S.: *Advanced Compiler Design Implementation*. Morgan Kaufmann Publishers, 1997, ISBN 9781558603202.
- [11] Pozzi, L.; Atasu, K.; Ienne, P.: Exact and approximate algorithms for the extension of embedded processor instruction sets. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, ročník 25, č. 7, 2006: s. 1209–1229.

# Príloha A

## Obsah CD

Priložené CD obsahuje nasledujúcu adresárovú štruktúru

- /thesis/pdf – text bakalárskej práce vo formáte .pdf
- /thesis/tex – nepreložený text bakalárskej práce
- /src/test – zdrojové kódy vytvoreného priechodu
- /examples – zdrojové kódy testovaných programov a skript na ich odtestovanie

## Príloha B

# Návod na preloženie priechodu optimalizátora

Táto kapitola obsahuje popis, ako preložiť a začleniť nový priechod do optimalizátora. Očakáva sa, že na počítači je nainštalovaný LLVM prekladač a všetky jeho potrebné súčasti.

Postup:

1. Skopírovať adresár `test` z priloženého CD do adresára, ktorý obsahuje prechody optimalizátora pre LLVM-3.2, napríklad: `compilerdev/build-3.2/lib/Transforms/`
2. Vo vyššie spomenutom adresári upraviť súbor Makefile tak, že do premennej `PARALLEL_DIRS` pridáte názov priečinku s priechodom, čiže `test`.
3. Spustiť Makefile

## Príloha C

# Návod na odtestovanie priechodu

Na tento účel je možné použiť skript `test.sh`, ktorý však obsahuje absolútne cesty. Je ho preto potrebné pred použitím upraviť podľa potrieb užívateľa. Pri testovaní prechodu na jednotlivých súboroch treba postupovať nasledovne.

1. Preložiť testovaný program do LLVM IR.  
`codix_ia-llvmc --save-temps bitcnt.c`
2. Pridať doňho profilovacie inštrukcie pomocou optimalizátora.  
`codix_ia-opt -insert-optimal-edge-profiling bitcnt.ll -o bitcnt.inst`
3. Preložiť na spustiteľný program.  
`codix_ia-llc bitcnt.inst -o bitc.inst.s`  
`codix_ia-llvmc bitcnt.inst.s libc.a libm.a libsim.a libcomp.a`  
`CommonProfiling.c OptimalEdgeProfiling.c`
4. Odsimulovať program.  
`codix_ia-intersim2 -i a.xexe -n n -x x`
5. Podrobiť pôvodný program v LLVM IR optimalizačnému prechodu.  
`codix_ia-opt -load cesta k preloženému prechodu/test.so -profile-loader`  
`-ise -st/ld -inst 4 bitcnt.ll -o out --debug-pass=Structure`
6. Z výsledného súboru `out` skopírovať definície ISE funkcií a vložiť ich do nového súboru.
7. Vo výslednom súbore `out` nahradiť definície ISE funkcií ich deklaráciami.
8. Spustiť program `isegen` pre súbor s definíciami ISE funkcií a výsledný súbor `ises.codal` umiestniť do adresáru `model` simulovaného procesoru.
9. Pomocou `CodasipStudia` vygenerovať všetky nástroje, získať súbor `swtlissom/project`  
`/$USER/codix_ia/gencompiler/compiler/inlines.h` a preložiť ho do LLVM IR.
10. Zlinkovať spolu upravený súbor `out` a `inlines.ll` a ich výsledok zo optimalizovať  
`codix_ia-opt -O3 out.bc -o out_opt.bc.`
11. Výsledok preložiť do assembleru, vytvoriť z neho spustiteľný súbor a odsimulovať.

## Príloha D

# Parametre prechodu

Nový optimalizačný prechod môže prijímať určité parametre.

- -st/lid – Určuje, či budú patriť inštrukcie **LOAD** a **STORE** medzi zakázané. Ak nieje zadaný, inštrukcie medzi zakázané patria.
- -in x – Určuje maximálny počet vstupných portov rezu. Ak nieje zadaný, počet portov je neobmedzený.
- -out x – Určuje maximálny počet výstupných portov rezu. Ak nieje zadaný, počet portov je neobmedzený.
- -inst x – Určuje minimálny počet inštrukcií, ktoré musí rez obsahovať. Ak nieje zadaný, minimálny počet je jedna.