VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

GENEROVANÉ PEEPHOLE OPTIMALIZACE V PŘEKLADAČI LLVM

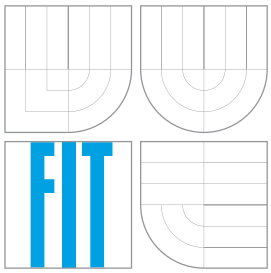
DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

STANISLAV MELO

BRNO 2016

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

GENEROVANÉ PEEPHOLE OPTIMALIZACE V PŘEKLADAČI LLVM

GENERATED PEEPHOLE OPTIMIZATIONS IN LLVM COMPILER

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

STANISLAV MELO

VEDOUCÍ PRÁCE
SUPERVISOR

Prof. Ing. TOMÁŠ HRUŠKA, CSc.

BRNO 2016

Abstrakt

Jednou z důležitých vlastností aplikačně specifických procesorů je jejich výkon. Aby byl co největší, musí se překladač přizpůsobit potřebám procesoru, pre který bude překládat a generovat co nejefektívnejší kód. Jedním ze způsobů přizpůsobení se je hledání vhodných instrukcií, které je možné implementovat jako jednu instrukci s více výstupy. Následně se dá vygenerovaný kód zefektivnit pomocí peephole optimalizátoru, který vyhledává skupiny instrukcí, které může směnit za jejich efektivnejší varianty. Tato práce popisuje problém nacházení a výběru instrukcí s více výsledky. Taktéž poskytuje stručný přehled několika nejznámějších algoritmů na řešení tohoto problému. Na závěr skoumá uplatnění a začlenění řešení spolu s peephole optimalizacemi, do překladače LLVM.

Abstract

One of the important feature of application specific processors is performance. To maximize it, the compiler must adapt to needs of processor that it is going to compile for and it must generate the most efficient code. One of the ways to do that is to search for appropriate instructions that can be implemented as one instruction with multiple outputs. Afterwards the generated code can be parsed through peephole optimizations that search for instruction patterns and replace them with other instructions to make code more effective. This paper describes the problem of finding and selecting suitable candidates for multiple output instructions. It also provides a brief overview of the few best known algorithms that solve this problem. Eventually it examines possibilities of incorporating this optimizations to LLVM compiler.

Klíčová slova

peephole optimalizace, instrukce s více výsledky, LLVM, generování kódu

Keywords

peephole optimizations, multile output instructions, LLVM, code generation

Citace

Stanislav Melo: Generované peephole optimalizace v překladači LLVM, diplomová práce, Brno, FIT VUT v Brně, 2016

Generované peephole optimalizace v překladači LLVM

Prohlášení

Prohlašuji, že jsem tento semestrální projekt vypracoval samostatně pod vedením pana prof. Ing. Tomáše Hrušky, CSc. Další informace mi poskytl konzultant, pan Ing. Adam Husár Ph.D. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Stanislav Melo
25.5.2016

Poděkování

Chtel bych se podekovat svému vedoucímu prof. Ing. Tomáši Hruškovi CSc. a mému konzultantovi Ing. Adamu Husárovi Ph.D. za připomínky a odbornou pomoc při konzultacích k téhle práci.

Obsah

1 Prekladač LLVM **5**
1.1 LLVM backend - llc 6

2 Teoretické základy, predstavenie problematiky a návrhy riešenia **9**
2.1 Základné pojmy 9
2.2 Existujúce algoritmi 10

3 Algoritmus systému Cburg **13**
3.1 Vstupné dáta 13
3.2 Identifikácia kandidátov 14
3.3 Výber MOI kandidátov 14

4 Implementácia nachádzania MOI inštrukcií **17**
4.1 Vzor inštrukcie 17
4.2 Prvý krok algoritmu 18
4.3 Hľadanie vzoru v grafe a jeho nahradenie 19

5 Peephole optimalizácie **20**
5.1 Základný princíp peephole optimalizácií 20
5.2 Spôsoby implementácie optimalizátoru a vyhľadávania vzorov 21
5.3 Použitie 23

6 Peephole optimalizácia na veľkosť kódu **24**
6.1 Optimalizácia na veľkosť kódu 24
6.2 Reprezentácia vzorov 24
6.3 Implementácia prechodu 25

7 Všeobecná peephole optimalizácia **26**
7.1 Príklady vzorov 26
7.2 Formát vzoru 28
7.3 Reprezentácia vzoru v optimalizácií 28
7.4 Implementácia optimalizácie 29

8 Generovanie vzorov inštrukcií **31**
8.1 Generátor backendu 31
8.2 Vzory pre MOI inštrukcie 32
8.3 Vzory pre peephole optimalizátor 33

9 Dosiahnuté výsledky **34**

A Obsah CD **40**

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 hodinky, fotoaparáty alebo rôzne smart zariadenia. Procesory v malých, jednoúčelových zariadeniach vykonávajú špecifickú činnosť neustále dokola. Je snaha aby ju vykonávali čo najrýchlejšie, s čo najmenšou spotrebou energie či iných zdrojov, a aby ich výroba bola čo najlacnejšia. Najvhodnejšie by bolo, keby jeden procesor vykonával jednu činnosť, pre ktorú by bol maximálne optimalizovaný. Navrhnúť procesor však nieje jednoduché a takýto prístup je cenovo neefektívny. Preto sa používajú jednoduché procesory s obvyklou inštrukčnou sadou, ktoré môžu byť doplnené o špeciálne inštrukcie.

Tie slúžia na urýchlenie nejakej činnosti, ktorú bude procesor často vykonávať, vzhľadom na svoje zameranie. Procesor zvyčajne obsahuje hardwarovú podporu na ich vykonávanie, Inak by ich používanie nemalo žiadny prínos. Namiesto nich by sa použilo zopár obyčajných inštrukcií. Program, ktorý sa musí dozvedieť o týchto špecialitách procesoru je prekladač. Musí poznať všetky špeciálne inštrukcie, ktoré dovoľuje procesor používať, vedieť identifikovať tú časť kódu, kde je možné ich použiť a naplno využiť výhod, ktoré nová inštrukcia ponúka. Na miestach, kde sa špeciálne inštrukcie nedajú použiť bude postupovať zvyčajným spôsobom. Časť tejto práce sa zaoberá spôsobom ako nájsť vhodnú časť kódu, ktorá sa dá nahradiť za špeciálnu inštrukciu s viacerými výstupmi.

Strojový kód, ktorý je prekladačom vygenerovaný, nesmie byť zbytočne veľký. Častokrát sa totiž stáva, že vygenerovaný kód obsahuje duplicitné príkazy, všeobecnejšie inštrukcie na miestach, kde by sa dali použiť špecifickejšie, ktoré sú častokrát kratšie, alebo iný takzvaný mŕtvy kód. Na odstránenie, alebo zlepšenie týchto inštrukcií nieje väčšinou potrebné poznať detailne štruktúru programu. Preto sa dajú na ich odstránenie použiť peephole optimalizácie. Tie dokážu prejsť celý program inštrukciu po inštrukcii a problémové časti programu zlepšiť. Danou vecou sa zaoberá druhá časť práce.

Kapitola 1 obsahuje popis prekladača LLVM, v ktorom budú dané optimalizácie implementované. Rozdelenie prekladača na časti a vysvetlenie ich činnosti.

Každý prekladač musí riešiť problém výberu inštrukcií výsledného kódu, preto existuje mnoho algoritmov, ktoré daný problém riešia. V kapitole 2 budú predstavené niektoré z nich, ktoré by sa dali použiť aj na výber inštrukcií s viacerými výsledkami. Kapitola tiež obsahuje objasnenie pojmov, ktoré sú pri popise potrebné.

Vybraný algoritmus bude predstavený v kapitole 3. Tá obsahuje pseudokód algoritmu a jeho vysvetlenie. Tiež sa tam nachádza popis, akým budú vstupné dáta algoritmu reprezentované.

Kapitola 4 obsahuje popis implementácie a fungovania vyhľadávania MOI inštrukcií. Taktiež predstavuje forát vzoru, ktorý reprezentuje hľadanú inštrukciu.

Kapitola 5 obsahuje všeobecný popis peephole optimalizácií, rôzne existujúce spôsoby implementácie a reprezentácie dát, s ktorými budú pracovať.

Kapitola 6 obsahuje popis implementácie menšej z peephole optimalizácií, ktorá sa zameriava na veľkosť výsledného kódu.

Kapitola 7 opisuje implementáciu druhej peephole optimalizáci, ktorá užívateľovi dovoluje narhnúť vlastný vzor. Taktiež popisuje formát, akým je vzor pri vyhľadávaní reprezentovaný, a formát určený pre zadávanie vzoru užívateľom. Taktiež ukazuje prípady, kde sa dá táto optimalizácia použiť.

Kapitola 8 ukazuje spôsob, akým sa vzory pre spomínané optimalizácie automaticky generujú.

Kapitola 9 obsahuje výsledky, ktoré optimalizácie pri testovaní dosiahli.

V rámci semestrálneho projektu boli vyhotovené kapitoly 1, 2, 3 a 5. Taktiež boli navrhnuté spôsoby reprezentácie vzorov pre všetky optimalizácie a bola implementovaná veľká časť prechodu na nachádzanie MOI inštrukcií a peephole optimalizátoru zameraného na veľkosť kódu.

Záver sumarizuje obsah práce a určuje jej ďalšie rozšírenie v rámci diplomovej práce.

Kapitola 1

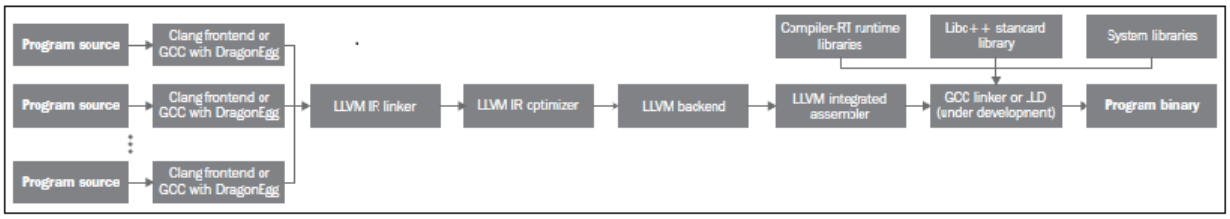
Prekladač LLVM

Prekladač LLVM začínal ako výskumný projekt na University of Illinois v roku 2000, pričom prvá verzia bola vydaná v roku 2003. Pôvodne bol názov skratkou, ktorá znamenala Low Level Virtual Machine, v preklade nízko úrovňový virtuálny stroj [8, 10]. Dnes sa projekt natoľko rozvinul, že už nemá s virtuálnymi strojmi takmer nič spoločné a jeho názov nieje vnímaný ako skratka. LLVM projekt je zbierka modulárnych a znovu použitelných knihovní a jednotlivých nástrojov prekladača. Jeho úlohou je poskytnúť moderný prístup prekladu, založený na vnútornom medzikóde prekladača, nazvaného LLVM IR (Intermediate Representation), ktorý je schopný podporovať preklad statických a dynamických programovacích jazykov. LLVM IR je jadrom projektu, do ktorého sa prekladá program z programovacieho jazyka a je výstupom frontendu prekladača. Je to kód, ktorý používa trojadresné inštrukcie v SSA forme. Single Static Assignment, čiže SSA forma je taká, v ktorej je každá hodnota priradená do jedinečnej premennej, ktorá ju definuje [10]. Každé použitie hodnoty môže byť vďaka tomu vystopované až ku svojej definícií. To zjednodušuje optimalizácie kódu, ktoré sa nad ním vykonávajú. LLVM IR je tiež dostatočne vysokoúrovňový, že umožňuje preklad do viacerých odlišných architektúr. Dnes projekt LLVM zastrešuje radu menších projektov, z ktorých sú mnohé používané veľkou radou komerčných a open source projektov [8, 10]. Medzi jeho podprojekty patrí napríklad:

Clang – Tento projekt predstavuje implementáciu frotendu LLVM prekladača pre jazyky C/C++/Objective-C. Jeho vstupom je zdrojový kód nad ktorým vykonáva lexikálnu, syntaktickú a sémantickú analýzu. tým skontroluje správny zápis programu vo vstupnom jazyku a vygeneruje kód s rovnakou funkciou v nízkoúrovňovom jazyku LLVM IR. Clang tak isto slúži aj ako ovládač pre ostatné časti LLVM prekladača. Dokáže ich automaticky spúšťať, predávať im parametre zadané užívateľom a tým zapuzdruje celý proces prekladu. Užívateľ zadá iba zdrojový kód a na konci získa spustiteľný program.

LLVM Core Libraries – Knižnice jadra sú prvou časťou, ktoré boli v rámci projektu LLVM implementované. Je to skupina knižníc, ktoré implementujú zdrojovo a cieľovo nezávislí optimalizátor nad LLVM IR. V tejto časti budú vytvorené aj popisované optimalizácie. Preto sa touto časťou budeme podrobnejšie zaoberať neskôr. Tieto knižnice tiež poskytujú prostriedky na generovanie back-endu prekladača, pre konkrétnu architektúru. Tá je popísaná špeciálnym spôsobom v súboroch s príponou .td.

Dragonegg – Slúži na integráciu LLVM optimalizátoru a generátoru kódu s parserom GCC. Vznikol v dobe keď neexistoval Clang a tak LLVM prekladač nemal vlastný frontend. Hoci to už neplatí, projekt prežil dodnes. Umožňuje aby boli pomocou LLVM kompilovatelné aj iné programovacie jazyky, napríklad Fortran a Ada. Tie sú podporované frontendmi GCC.



Obr. 1.1: Ukážka prekladu programu nástrojmi projektu LLVM [10]

LLDB – Low Level Debbuger je ladiaci nástroj založený na knižniciach LLVM a Clangu. Preto je veľmi vhodný na ladenie projektov kompilovaných týmto prekladačom. Je rýchlejší a omnoho efektívnejší pri využívaní pamäte ako GDB.

Compiler-rt – projekt obsahuje vysoko optimalizované implementácie nízkoúrovňových operácií, ktoré môže použiť generátor kódu. Použije ich vtedy, ak cieľová architektúra nemá krátku postupnosť jednoduchých inštrukcií, ktorými by implementovala funkčnosť kľúčových inštrukcií LLVM IR kódu. Tiež poskytuje implementácie knižníc pre dynamické testovacie nástroje.

Jednoduchý spôsob ako si predstaviť preklad programu pomocou LLVM nástrojov ukazuje obrázok 1.1. Ten predstavuje jednu z konfigurácií v akej je možné prekladač použiť. Jednotlivé komponenty môžeme preorganizovať a využiť ich v inom zoskupení. Napríklad nemusíme použiť LLVM IR linker ak nechceme využiť optimalizácie pri linkovaní.

Komunikácia medzi hociktorými dvomi časťami môže prebiehať dvomi spôsobmi [10]:

Pomocou pamäte – to sa deje cez jeden ovládací nástroj ako Clang. Ten používa každú časť prekladača ako knihovňu a závisí na dátových štruktúrach vytvorených v pamäti aby výstup jednej časti predali ako vstup druhej časti.

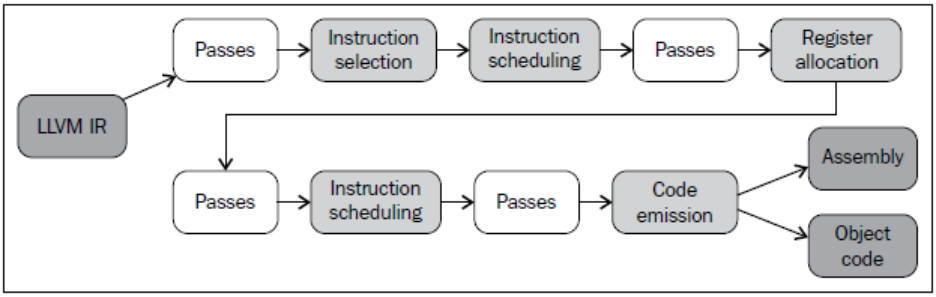
Pomocou súborov – To sa deje ak užívateľ spúšťa malé samostatné nástroje, ktoré zapisujú výstup do súborov uložených na disku. Potom musí manuálne spustiť ďalšiu časť prekladu, ktorá dostane daný súbor ako svoj vstup.

1.1 LLVM backend - llc

Takmer celá funkcionalita llc je začlenená do znovupoužitelných knižníc, len malá časť kódu sa nedá požiť opakovane. Množina knižníc sa skladá z dvoch častí, časti nezávislej na architektúre a z časti závislej na architektúre [10]. Nezávislé knižnice slúžia na generovanie, spracovanie, či zápis assembleru do súboru a implementáciu assembleru a dissasembleru. Patria tam tiež knižnice, ktoré implementujú výber inštrukcií a generovanie kódu. Knižnice na vygenerovanie backendu prekladača pre konkrétnu architektúru, na základe jej jednoduchého popisu. Aspoň ich všeobecnú časť. Takmer každá z nich má aj svoje architektúrne špecifickú verziu, ktoré sa vygenerujú. Naviac k architektúrne špecifickým ešte patria knižnice na popis architektúry potrebný pri generovaní backendu a knižnica na zaregistrovanie architektúry v LLVM generátore kódu.

Keďže peephole optimalizácie a výber inštrukcií sa budú vykonávať v backende, táto kapitola priblíži jeho štruktúru a kroky prekladu zdrojového kódu.

Samotný backend sa skladá z množiny analýz a transformačných prechodov, ktoré postupne konvertujú kód do výslednej podoby [10]. LLVM podporuje množstvo výsledných architektúr, pričom každý z backendov má spoločné rozhranie. To je časťou architektúrne nezávislého generátora kódu. Každá architektúra potom musí špecializovať jeho všeobecné



Obr. 1.2: Ukážka časti prekladu v backende prekladača LLVM [10]

triedy, aby implementovala svoje špecifické inštrukcie. Ako bolo spomenuté v predchádzajúcej kapitole, táto časť prekladača má za úlohu transformovať kód vo forme LLVM IR do assembleru. To sa deje pomocou transformačných prechodov popísaných v obr. 1.2. Počas spracovania sa kód pretransformuje do rôznych iných podôb, než sa z neho stane assembler.

Šedé obdĺžniky predstavujú prechody, ktoré sú nevyhnutné. Často sú zložené z viacerých menších prechodov. Biele obdĺžniky sú voliteľné prechody, ktoré slúžia na zefektívnenie vygenerovaného kódu.

Prvá z povinných fáz je výber inštrukcií. Táto fáza slúži na konvertovanie LLVM IR reprezentácie načítanej v pamäti na orientovaný acyklický graf. Uzly grafu predstavujú inštrukcie špecifické pre danú architektúru. Hrany označujú dátové závislosti medzi inštrukciami. Každý graf reprezentuje inštrukcie jedného základného bloku. Táto fáza je potrebná, aby mohol generátor kódu použiť pri výbere strojových inštrukcií algoritmus založený na nachádzaní vzorov tvaru stromu. Na konci tejto fázy má graf ako uzly strojové inštrukcie, namiesto inštrukcií LLVM IR kódu.

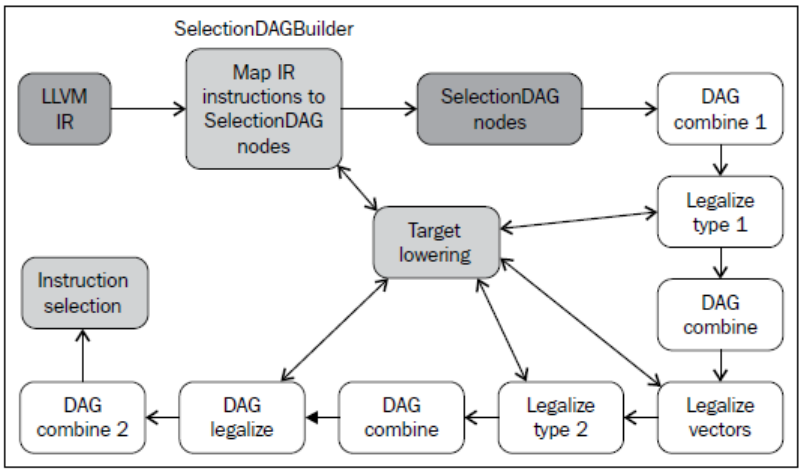
Nasledujúca fáza má za úlohu nahradiť grafovú reprezentáciu inštrukcií naspäť na trojadresný kód. Grafová reprezentácia totiž neobsahuje informácie o poradí inštrukcií, ktoré na sebe nezávisia. Prvá časť plánovania inštrukcií sa tiež nazýva plánovanie pred registrovou alokáciou. Snaží sa usporiadať inštrukcie, pričom chce čo najväčšmi využiť paralelizmus pri ich používaní. Výsledkom je trojadresný kód strojových inštrukcií.

LLVM IR používa nekonečný počet virtuálnych registrov. V skutočnosti to však nieje možné. Fáza alokácie registrov preto nahradzuje virtuálne registre za konečný počet registrov špecifických pre danú architektúru. Ak je registrov málo, automaticky generuje ukladanie ich obsahu na zásobník.

Druhá časť plánovania inštrukcií sa vykoná po alokácií registrov. Prekladač má preto informácie o skutočných registroch a dokáže rozoznať ďalšie problémy a meškania spojené s používaním určitých typov registrov.

Fáza emisie kódu premieňa inštrukcie triedy MachineInstr za strojové inštrukcie triedy MCInstr. Tá reprezentácia je vhodnejšia pre assembler a linker.

Samotný výber inštrukcií sa dá rozdeliť na menšie časti [10]. Ich štruktúra je načrtnutá v obrázku 1.3. Najprv sa namapujú inštrukcie IR na architektúrne nezávislé uzly acyklického grafu. Už v tejto fáze potrebujú niektoré IR inštrukcie informácie závislé na architektúre. Napríklad ako predávať argumenty a ako sa vracať z funkcie. Preto sa používajú metódy z triedy Target lowering, ktoré dokážu tieto informácie o architektúre získať. Uzly, ktoré sú vytvorené z IR reprezentácie ešte nie sú pripravené na výber inštrukcií, ale musia prejsť ďalšími transformáciami. Kombinačné prechody slúžia na optimalizáciu, pri ktorej



Obr. 1.3: Ukážka štruktúry výberu inštrukcií [10]

sa nahradzujú suboptimálne časti grafu za ich optimálne varianty, vždy keď je to výhodné. Potom nasleduje legalizácia typov, ktorá zaručuje, že uzly budú pracovať iba s dátovými typmi, ktoré architektúra podporuje. Môže nastať situácia, v ktorej architektúra podporuje určitý dátový typ, ale nie inštrukciu s týmto typom. Vtedy je potrebné inštrukciu zmeniť. Inštrukciu s menším dátovým typom je potrebné propagovať, čiže použiť inštrukciu s väčším dátovým typom. Inštrukciu s väčším dátovým typom je zase potrebné rozšíriť, čiže použiť viacej daných inštrukcií s menšími dátovými typmi. Tieto situácie riešia legalizačné priechody. V ktoromkoľvek kroku sa môžu použiť inštrukcie, ktoré potrebujú informácie špecifické pre architektúru, preto sú prepojené s triedou Target lowering. Tesne pred výberom inštrukcií je pridaný prechod na vyhľadávanie inštrukcií s viacerými výsledkami. Ten, podobne ako kombinačné prechody, nájde skupinu uzlov v grafe, ktoré reprezentujú špeciálnu inštrukciu, a nahradí ich za jeden uzol, ktorý je rozpoznaný pri výbere inštrukcií.

Kapitola 2

Teoretické základy, predstavenie problematiky a návrhy riešenia

Algoritmy na výber inštrukcií pracujú nad grafom dátových závislostí. Ten je v tomto prípade reprezentovaný orientovaným acyklickým grafom. Začiatok kapitoly je preto venovaný definícií a vysvetleniu pojmu graf a pojmov s ním súvisiacich. Tie sú potom použité v ďalších častiach kapitoly. Nasledujúca časť obsahuje teoretický popis problému vyhľadávania a výberu inštrukcií. Na záver kapitoly sú predstavené niektoré spôsoby, ktoré sa v danej oblasti používajú.

2.1 Základné pojmy

Orientovaný acyklický graf nazývaný tiež DAG, z anglického Directed Acyclic Graph, G 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.

DAG, ktorý sa nachádza v LLVM obsahuje dva ďalšie typy hrán. Jeden typ sa nachádza pri inštrukciách `load` a `store`. Ten predstavuje poradie inštrukcií, v ktorom za sebou nasledujú, aj keď nie sú prepojené dátami. Druhý typ hrán predstavuje závislosť jednej inštrukcie na druhej. Ak sú inštrukcie prepojené touto hranou, musia nasledovať hneď po sebe. Týmto typom hrany sa pri nemusíme zaoberať.

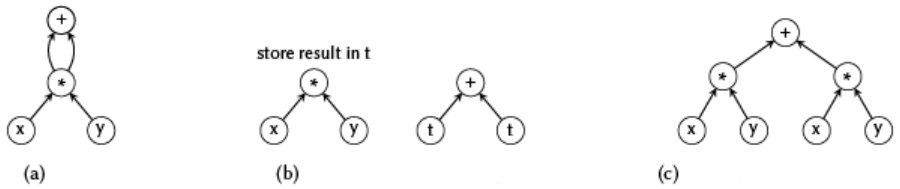
Vďaka tomu, že na reprezentáciu programu sa v tejto časti používa DAG, môžu mať uzly grafu viac výstupných hrán. Medzivýsledky výrazov môžu teda byť zdielané a znovu použité v tom istom grafe. Taktiež inštrukcie architektúry môžu mať viacej výsledkov, tým pádom aj vzory, ktoré ich reprezentujú môžu mať viacej koreňov.

Jeden DAG predstavuje jeden základný blok. Základný blok [9] 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.

Nájdenie vzoru inštrukcie v grafe je len časťou problému. Rovnaké časti grafu môžu vyhovovať pre viaceré inštrukcie. Potom je potrebný optimálny výber inštrukcií, pričom vybrané inštrukcie musia pokryť celý graf programu.

Optimálny výber inštrukcií sa dá definovať nasledovne [3].

Definícia 1 *Výber inštrukcií, ktorý vyberá z množiny inštrukcií I s cenou c_i je optimálny, ak pre každý vstupný program P vyberie množinu $S \subset I$ pre ktorú platí: S implementuje P*



Obr. 2.1: Ukážka spôsobu vytvárania stromov z DAGu (a) DAG; (b) delenie; (c) duplikácia [3]

a neexistuje žiadna množina S', ktorá tiež implementuje P a pre ktorú platí $\sum_{s' \in S'} c_{s'} < \sum_{s \in S} c_s$.

Výhody, ktoré nám DAG ponúka vo všeobecnosti a možnostiach modelovania vzorov, sú za cenu zložitejšieho algoritmu na ich nachádzanie a výber. Optimálne pokrytie DAGu je NP-úplný problém [6]. Napríklad dôkaz autorov Koes a Goldstein [6] používa redukciu SAT problému na problém pokrytia DAGu s čo najnižšou cenou. SAT problém je rozhodnúť, či booleovská formula v konjuktívnej normálnej forme je rozhodnutelná. Je dokázané, že tento problém je NP-úplný. Tým aj každý problém, ktorý sa v polynomiálnom čase dá redukovať na SAT je NP-úplný. Najprv pretransformujeme formulu SAT problému na DAG. Pokiaľ dokážeme pokryť tento DAG vzormi s jednotnou cenou, tak, že celková cena je rovná počtu uzlov, tak existuje priradenie hodnôt pre formulu SAT problému také, že formula je pravdivá.

2.2 Existujúce algoritmi

Prvé z algoritmov sa snažili použiť čo najviac poznatkov získaných z výberu inštrukcií nad stromovými grafmi. Napríklad hoci sa vyhľadáva nad DAGom, vzory inštrukcií sú reprezentované iba pomocou stromov [3]. To umožňuje použiť algoritmi z výberu inštrukcií reprezentovaných stromovými grafmi, pričom niektoré z nich majú lineárnu časovú zložitosť. Samotný DAG reprezentujúci program je tiež prevedený na strom. To sa dá dosiahnuť dvomi spôsobmi. Prvý je delenie hrán. Ak z nejakého uzlu vychádza viacero hrán, výsledok daného uzlu sa uloží a pri uzloch, ktoré ho používajú sa podvýraz nahradí za jeden uzol, v ktorom je výsledok uložený. Nevýhodou tohto prístupu je, že sa neuplatnia zložitejšie vzory, lebo graf programu sa môže rozdeliť na mnoho malých stromov. Druhý spôsob je duplikácia spoločných uzlov so všetkými jeho predchodcami. Nevýhodou je, že vygenerovaný kód je neefektívny, lebo obsahuje mnoho operácií, ktoré sa opakujú. Tieto spôsoby sú znázornené na obr. 2.1. Pri používaní tohto postupu bol navrhnutý algoritmus, ktorý sa snaží vyrovnane používať delenie a duplikáciu uzlov. Na výber medzi nimi požíva porovnanie ceny, ktorá je možnosti priradená. Cena sa vypočíta ako odhad veľkosti vygenerovaného kódu, pri použití duplikácie alebo delenia hrán, a času jeho vykonania.

Existujú metódy, ktoré nepotrebujú meniť DAG programu na stromy, ale vykonávajú vyhľadávanie vzorov priamo na ňom [3]. Medzi ne patrí aj metóda použitá v LLVM prekladači. V nej sa architektúrne nezávislý DAG prepíše na architektúrne závislý DAG. Stále však používa vzory tvaru stromov. Vzory sú popísané pomocou popisného systému nástroja TableGen, ktorý ich pretransformuje na úplné stromové vzory. Tie sa použijú pri generovaní

časti prekladača zodpovednej za výber inštrukcií. Tá najprv vykoná lexikografické usporiadanie vzorov grafu. Najprv podla znižujúcej sa komplexity, potom podla zvyšujúcej sa ceny a nakoniec podla zvyšujúcej sa veľkosti vzoru. Potom je každý vzor pretvorený na malý, rekurzívny program, ktorý kontroluje, či sa uzly zhodujú s uzlami vzoru. Ďalej sú tieto programíky skompilované do byte-kódového formátu a zostrojujú párovaciu tabuľku. Tá sa používa pri výbere inštrukcií. Výber prebieha ako prechádzanie uzlov grafu, pri ktorom program konzultuje uzol s tabuľkou, ktorá mu hovorí, aké uzly môžu nasledovať ako jeho potomok pre ktorý vzor.

Problém výberu inštrukcií, môže byť tiež modelovaný ako problém integerového programovania (IP) [3, 13]. To je metóda na riešenie kombinačných optimalizačných problémov. Množina prípustných riešení je ohraničená lineárnymi nerovnicami a úlohou je vybrať najmenšiu, alebo najväčšiu hodnotu, ktorá je riešením rovnice popisujúcej problém. Riešenie problému IP býva zvyčajne dlhšie, výhod je v možnosti ľahkého pridania ďalších podmienok, ktoré musí podgraf mapovaný na vzor spĺňať. Tie však musia byť vyjadrené pomocou lineárnych nerovníc, čo nemusí byť jednoduché. V prípade inštrukcií s viacerými výsledkami je možné použiť algoritmus IP, ktorý pracuje so SIMD inštrukciami. V tomto prípade algoritmus predpokladá, že SIMD inštrukcia je tá, ktorá vykonáva aspoň dve operácie, pričom každá z operácií má disjunktnú množinu vstupov. Lineárne nerovnice, ktoré opisujú omedzenia na výber SIMD.

$$\sum_{r_j \in R(n_i)} x_{ij} = 1 \tag{2.1}$$

$$x_{ij} \leq \sum_{r_l \in R_m(n_k)} x_{kl} \tag{2.2}$$

$$\sum_{j:(n_i;n_j)\in P} y_{ij} = \sum_{r_k \in R_{hi}(n_i)} x_{ik} \tag{2.3}$$

$$\sum_{j:(n_j;n_i)\in P} y_{ji} = \sum_{r_k \in R_{lo}(n_i)} x_{ik} \tag{2.4}$$

Podľa [3, 13] nerovnice 2.1 a 2.2 zaisťujú validný výber inštrukcií vo všeobecnosti. Nerovnice 2.3 a 2.4 sú špecifické pre SIMD inštrukcie. Rovnica 2.1 zaručuje, že pre každý uzol v DAGu existuje pravidlo. $R(n_i)$ predstavuje množinu pravidiel, ktorých vzory sa zhodujú na uzle n_i. x_{ij} je booleovská premenná, ktorá určuje, či je uzol n_i pokrytý pravidlom r_j. Nerovnica 2.2 vyžaduje, aby pravidlo vybrané pre n_k, potomka uzlu n_i, redukovalo n_k na rovnaký neterminál, ktorý je vyžadovaný pravidlom vybranom pre n_i. N_k je m-tý potomok uzla n_i a $R_m(n_k)$ je množina pravidiel, ktoré uzol n_k redukujú na m-tý nonterminál pravidla r_j. Rovnice 2.3 a 2.4 vyžadujú, aby aplikovanie SIMD inštrukcie naozaj pokrylo SIMD pár a že každý uzol môže byť pokrytý najviac jednou takou inštrukciou. y_{ij} je premenná typu boolean, ktorá indikuje, či dva uzly grafu n_i a n_j môžu byť spojené do SIMD inštrukcie. $R_{hi}(n_i)$ a $R_{lo}(n_i)$ sú množiny pravidiel, ktoré sú aplikovateľné na uzol n_i, ktoré operujú na hornej a dolnej časti registra, ktorý obsahuje výsledok inštrukcie. Úlohou je maximalizovať počet SIMD inštrukcií. Toho sa dosiahne maximalizáciou nasledujúcej funkcie.

$$f = \sum_{n_i \in N} \sum_{r_j \in S(n_i)} x_{ij} \tag{2.5}$$

N je množina uzlov, ktoré obsahuje DAG a $S(n_i) = R_{hi}(n_i) \cup R_{hi}(n_i) \subset R(n_i)$.

Keďže nie všetky inštrukcie sa dajú popísať pomocou stromu, používajú sa tiež postupi pri ktorých aj vzory inštrukcií majú formu DAGu. Také vyhľadávanie a pokrývanie sa dá robiť dvomi spôsobmi [3]. Rozdeliť DAG na stromy a tie spárovať individuálne. Pritom je možné použiť už predstavené metódy. Po nájdení stromových vzorov je nutné ich znovu spojiť a overiť, či sa našli všetky časti danej inštrukcie. Druhý spôsob je vyhľadať priamo DAG v DAGu. Ten sa takmer nepoužíva, pretože je rovnako, alebo takmer rovnako zložitý, ako izomorfizmus podgrafov. Keďže graf je všeobecnejší ako DAG, nemalo by zmysel sa obmedzovať iba na podmnožinu grafov. V roku 2001 Arnold a Corporaal [1] prišli so spôsobom, ktorý využíva prvú metódu delenia vzorov na stromy. Tiež dovoľuje označiť výstup inštrukcie explicitne, nemusí sa nachádzať iba v koreni vzoru. V tomto princípe sa DAG rozdelí tak, že každý výstupný uzol vzoru sa stane koreňom stromu. Ten reprezentuje určitý podvzor. Potom sa použije vyhľadávací algoritmus s časovou zložitosťou $O(mn)$, ktorý nájde stromové podvzory v DAGu. Po vyhľadávaní je potrebné spojiť vhodné kombinácie podvzorov. To sa dosiahne tým, že sa udržuje pole pre každý podvzor, ktorý bol nájdený a potom sa kontroluje, či niektoré z nich nepatria k sebe a nekolidujú navzájom. Inými slovami uzli podvzorov, ktoré v pôvodnom vzore predstavujú ten istý uzol, musia pokrývať ten istý uzol v DAGu reprezentujúcom program. Výber vzorov je robený pomocou dynamického programovania, ktoré je popísané v [3]. To pri vyhladávaní redukuje uzly programu na neterminály pomocou najlacnejšieho, aplikovatelného pravidla. Cena redukcie predstvuje cenu pravidla plus cenu redukcie uzla na neterminály, ktoré práve aplikované pravidlo používa. Cena konkrétnej redukcie sa tým zvyšuje. Na konci, ak je možné redukovať vzor na počiatočný neterminál viacerými spôsobmi, je možné vybrať ten najlepší.

Kapitola 3

Algoritmus systému Cburg

Algoritmus bude vychádzať z toho, ktorý je popísaný v [14]. Je to spôsob použitý v generátore kódu Cburg predstavený v roku 2007, pričom bol upravený v roku 2011, tak, že gramatika, ktorú používa na popis vzorov inštrukcií umožňuje popísať inštrukcie s viacerými výsledkami, nazývané tiež MOI (Multiple Output Instruction), ktoré zdieľajú operandy. Vstupom algoritmu je DAG operačných uzlov a výstupom je označenie uzlov pravidlami, podľa ktorých budú v ďalšej fázy generované strojové inštrukcie.

3.1 Vstupné dáta

Cburg využíva na popis vzorov stromovú gramatiku, ktorá má tri typy inštrukcií. Prvý typ sú inštrukcie s jedným výsledkom. Pravidlá gramatiky, ktoré ich označujú, vyzerajú ako pravidlá takmer vo všetkých gramatikách. Upravuje, ktorý neterminál sa rozgeneruje na ktorú inštrukciu. Navyše obsahujú informácie o cene použitia daného pravidla a akčnú časť. To je úsek C kódu, ktorý vykoná emitovanie assembleru.

Pravidlá pre MOI inštrukcie sú veľmi podobné, každé z nich obsahuje dva nontermiály a dve inštrukcie, na ktoré sa príslušné nonterminály rozgenerujú. Časť pravidla patriaca jednému nonterminálu predstavuje čiastočné pravidlo. Ak je k operandu navyše priradené číslo, značí to, že operand druhej inštrukcie s rovnakým číslom predstavuje uzol, ktorý dané vzory zdieľajú.

Mnou navrhovaná optimalizácia obsahuje vlastnú štruktúru ktorá reprezentuje jeden uzol grafu. Obsahuje položky, ktoré predstavujú všetky informácie, ktoré je potrebné pri identifikácií vzorov. Je to rekurzívna dátová štruktúra, jedným z jej prvkov je pole štruktúr rovnakého typu. To predstavuje všetkých potomkov uzlu. Výsledné operandy sú označené špeciálny typom uzlu, pri ktorom nezáleží o akú operáciu sa jedná, kontroluje sa iba jej typ, teda či listová operácia vzoru prijíma operand so správnym typom. Na rozdiel od generátora Cburg, všetky operandy sú označené indexom, ktorý určuje poradie daného operandu pre výsledný uzol. Operandy s rovnakým indexom predstavujú rovnaký uzol v grafe. Každá hľadaná inštrukcia sa skladá z pola daných štruktúr, pričom každý prvok pola predstavuje jeden z koreňov DAGu. To je podobné systému Cburg, keďže každý koreň predstavuje jeden z neterminálov a ten v sebe obsahuje informácie o strome, ktorého je koreňom a ktorý predstavuje čiastočné pravidlo.

Práca programu sa dá rozdeliť na štyri základné kroky. Prvým z nich je identifikovanie kandidátov vhodných na MOI inštrukcie. Uzly DAGu predstavujúceho program, môžu byť totiž pokryté čiastočnými pravidlami MOI inštrukcií, alebo pravidlami jednoduchých

inštrukcií s jedným výstupom. Druhý krok skúma identifikovaných kandidátov a vyberá z pomedzi nich tých vhodných na MOI inštrukcie. V nasledujúcom kroku je porovnávaná výhodnosť nájdených MOI inštrukcií, oproti tomu, keby boli implementované ako jednoduché inštrukcie. Posledný krok vyhľadáva uzly, ktoré boli v kroku 2 označené ako MOI, ale v kroku 3 premenené na jednoduché inštrukcie. Tým sú potom priradené správne vzory jednoduchých inštrukcií.

Usporiadanie prekladača LLVM je trochu iné. V pripravovanej optimalizácií je potrebné vyhľadať iba MOI inštrukcie, pokrytie ostatných uzlov bežnými inštrukciami sa vykoná v nasledujúcom kroku prekladu. Tiež sa predpokladá, že implementácia časti kódu MOI inštrukciami bude vždy lepšia, ako keby bola implementovaná pomocou bežných inštrukcií. Preto sa nieje potrebné zaoberať tretím a štvrtým krokom Cburg generátoru kódu.

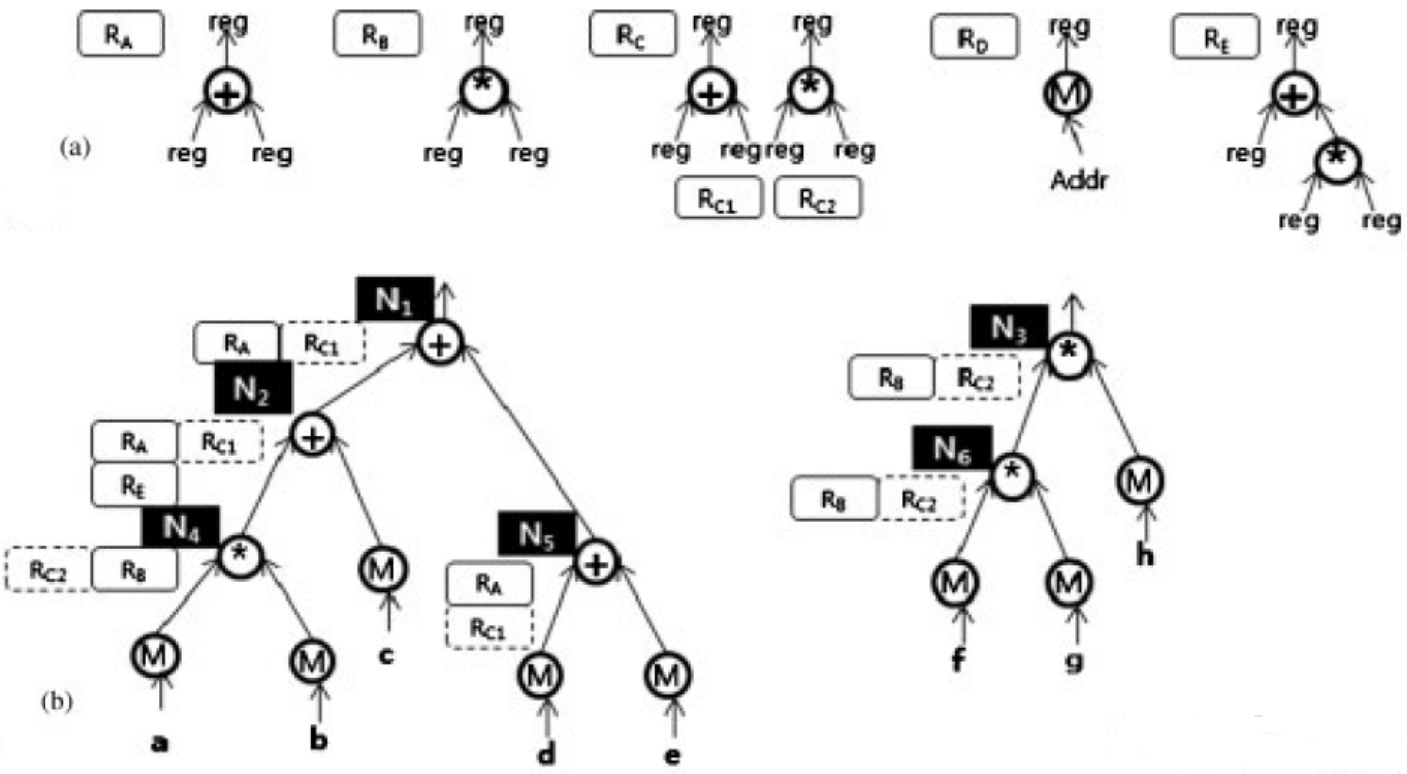
3.2 Identifikácia kandidátov

Úlohou tohto kroku je označiť každý uzol grafu pravidlom, podľa ktorého bude táto časť grafu redukovaná na konkrétnu inštrukciu. Aj keď neuvažujeme vyhľadávanie jednoduchých inštrukcií, môžu byť uzly označené viacerými pravidlami, ak sa vzory viacerých MOI inštrukcií prekrývajú. Po tomto označení sa vytvorí tabuľka čiastočných pravidiel. Tá priraďuje každému čiastočnému pravidlu množinu uzlov, ktoré ním môžu byť pokryté. Vďaka nej je jednoduché vytvoriť MOI kandidátov jednoducho kombináciou čiastočných pravidiel tým, že spárujeme odpovedajúce uzly DAGu pokryté čiastočnými inštrukciami. Každé spárovanie grafov predstavuje kandidáta na MOI inštrukciu. Týmto môže vzniknúť veľké množstvo MOI kandidátov. Na nich sa aplikuje kontrola dátovej závislosti a kandidáti, ktorý ju nesplnia budú vyradený. Daná kontrola skúma, či medzi uzlami jednej MOI inštrukcie neexistuje dátová závislosť, ktorá nieje vo vzore popísaná. V príklade zobrazenom na obr. 3.2 sú preto vylúčené uzly MC_2 a MC_5.

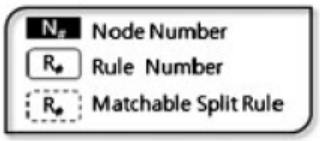
3.3 Výber MOI kandidátov

Pri identifikácií kandidátov sa môže stať, že niektorý z nich si budú navzájom prekážať. Vtedy je potrebné vybrať najvhodnejšieho kandidáta. Pri výbere kandidátov berieme do úvahy dva faktory: zisk kandidáta a prekážanie ostatným kandidátom. Zisk sa určuje pre každého MOI kandidáta zvlášť, prekážanie zase medzi viacerými z nich. Potom čo sú analyzované všetky vlastnosti vzoru, dá sa problém výberu formulovať ako problém nezávislej množiny s maximálnou váhou (MWIS – Maximum Weighted Independent Set), ktorý je NP-úplný [3], preto je potrebné na jeho riešenie použiť heuristiku. Nakoniec výsledkom tohto kroku je množina MOI kandidátov, ktorý nekolidujú zo žiadnym iným kandidátom a môžu byť prevedené na MOI inštrukcie. Program rozoznáva dva spôsoby, ako si kandidáti môžu prekážať: prekážanie pri pokrytí a prekážanie pri plánovaní. Vzťah prekážania si reprezentujeme grafom $IG = (V, E)$ kde V je množina všetkých MOI kandidátov a každá neorientovaná hrana $(v_i, v_j) \in E$ medzi dvomi uzlami $v_i, v_j \in V$ reprezentuje, že uzly si navzájom prekážajú.

Dva alebo viacej MOI kandidátov si prekážajú pri pokrývaní grafu vtedy, keď označujú ten istý uzol grafu. Pre jeden uzol nemôže byť vybraných viacero kandidátov. Inak by bol uzol pokrytý viackrát. To by nezachovalo sémantiku programu. Pre každú takúto závislosť je v grafe IG vygenerovaná hrana medzi kandidátmi.



Obr. 3.1: Ukážka vzorov inštrukcií a príklad pokrytia: (a) pravidlá; (b) DAG [14]

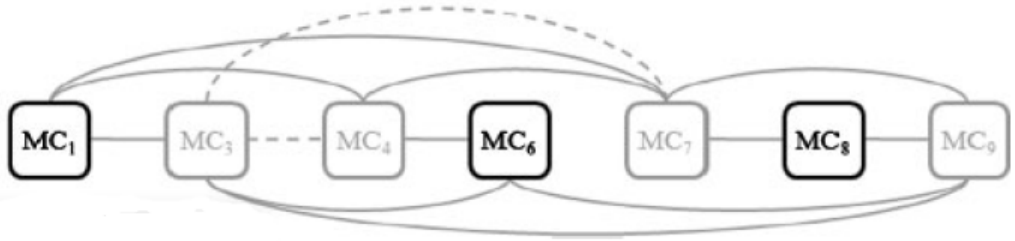


(c)

Split Rule	Candidate IR Nodes
R_{C1}	N_1, N_2, N_5
R_{C2}	N_3, N_4, N_6

(d) $MC_1 = \{ N_1, N_3 \}$ ~~$MC_2 = \{ N_1, N_4 \}$~~ $MC_3 = \{ N_1, N_6 \}$
$MC_4 = \{ N_2, N_3 \}$ ~~$MC_5 = \{ N_2, N_4 \}$~~ $MC_6 = \{ N_2, N_6 \}$
$MC_7 = \{ N_5, N_3 \}$ $MC_8 = \{ N_5, N_4 \}$ $MC_9 = \{ N_5, N_6 \}$

Obr. 3.2: Ukážka vzorov inštrukcií a príklad pokrytia:(c) mapa čiastočných pravidiel; (d) MOI kandidáti pre R_C [14]



Obr. 3.3: Ukážka výberu MOI kandidátov pomocou grafu [14]

Prekážaním si pri plánovaní inštrukcií sa nieje potrebné zaoberať. LLVM totiž plánuje inštrukcie v inom priechode a až po ich výbere.

Dôležitým faktorom výberu vhodného kandidáta je jeho zisk. Generátor kódu Cburg umožňuje zadať cenu jednotlivých pravidiel popisujúcich vzory inštrukcií. Na ich základe potom počíta celkový zisk vybranej inštrukcie, ktorý sa skladá z dvoch častí: ušetrenej ceny a duplikačnej ceny. Ušetrená cena predstavuje rozdiel medzi sumou cien, ak by boli dané uzly pokryté jednoduchými inštrukciami a cenou pokrytia MOI inštrukciou. Cena duplikácie predstavuje cenu, ak je treba niektorý z výrazov obsiahnutých vo vzoroch reprodukovať na iných miestach grafu. Ako cenu pravidla je možné použiť čokoľvek, napríklad množstvo uzlov, ktoré vzor pokryje.

V našom príklade máme šesť množín kompatibilných kandidátov. $\{MC_1, MC_6, MC_8\}$, $\{MC_1, MC_9\}$, $\{MC_3, MC_8\}$, $\{MC_4, MC_8\}$, $\{MC_4, MC_9\}$, $\{MC_6, MC_7\}$. Napríklad množina s jediným prvkom, napríklad $\{MC_1\}$ je vždy valídna, lebo nekoliduje zo žiadnym iným prvkom v množine. Tá však nebude vybraná, lebo iné množiny majú väčší zisk. V našom príklade je množina s najväčším ziskom $\{MC_1, MC_6, MC_8\}$. Problém výberu tejto množiny Môže byť jednoducho modelovaný ako problém MWIS (Maximum Weighted Independent Set), ktorý sa snaží vybrať množinu nezávislých uzlov grafu s čo najväčšou váhou. Váhu uzlu, predstavuje zisk MOI inštrukcie, ktorú predstavuje a váha množiny je súčet ziskov inštrukcií. Nezávislá množina je množina uzlov grafu, ktoré nie sú navzájom prepojené.

Ako bolo spomenuté, problém MWIS je NP-úplný, preto na jeho riešenie použijeme heuristiku GWMIN2 [14]. Pracuje tak, že konštruuje nezávislú množinu výberom uzla s najmenším počtom hrán, odstráni ho a jeho hrany z grafu a iteruje nad zbytkom grafu, pokiaľ nieje prázdny. V každej iterácií vyberá uzol v taký, že

$$zisk(v) = \frac{W(v)}{\sum_{w \in N^+_{G(v)}} W(w)} \forall v \ \in V$$

je maximalizovaný. notácia $N_{G(v)}$ znamená susedné uzly uzlu v v grafe G a množina $N^+_{G(v)}$ je $\{v\} \cup N_{G(v)}$. Algoritmus má lineárnu časovú zložitosť závislú na počte hrán a uzlov grafu. Obrázok 3.3 znázorňuje proces výberu MOI inštrukcií pre náš príklad.

Kapitola 4

Implementácia nachádzania MOI inštrukcií

Táto kapitola obsahuje konkrétnu implementáciu nachádzania inštrukcií s viacerými výsledkami. Toto nachádzanie sa vykonáva nad orientovaným acyklickým grafom, ktorý reprezentuje jeden základný blok prekladaného programu. Jeho výsledkom je upravený graf, obsahujúci nový uzol, ktorý reprezentuje nájdenú MOI inštrukciu. Bezprostredne po tomto prechode začne prekladač zamieňať inštrukcie vnútorného kódu za konkrétne inštrukcie architektúry.

4.1 Vzor inštrukcie

Implementácia začiatku prechodu sa nachádza v súbore `CodasipMoiLowerer.inc`. Prípona `.inc` znamená, že pri preklade bude obsah tohto súboru vložený na miesto, kde sa nachádza príkaz `#include CodasipMoiLowerer.inc`.

Tento súbor obsahuje implementáciu funkcie `LowerMoiPatterns`. Tá prijíma ako parameter graf, nad ktorým bude prebiehať vyhľadávanie. Obsahuje cyklus, v ktorom sa volá metóda na vyhľadávanie pre jednotlivé vzory. Tento obsah musí byť vygenerovaný, pretože dopredu nieje známe, koľko vzorov bude architektúra obsahovať. Vyhľadávanie jedného vzoru sa nad grafom opakuje dovtedy, pokiaľ sa v ňom vyskytuje. Druhú časť súboru tvoria definície konkrétnych tried, ktoré reprezentujú vzory jednotlivých MOI inštrukcií. Vyzerajú nasledovne.

```
struct MoimMoiGroup7Class: public DagPattern
{
  MoimMoiGroup7Class()
  {
    mRoots << (PNode(ISD::ADD, MVT::i32) + PDummyOper(MVT::i32,1)
      + PDummyOper(MVT::i32,2));
    mRoots << (PNodeExtL(ISD::EXTLOAD, MVT::i32, MVT::i16, 1, 0)
      + PDummyOper(MVT::i32,1));
    icode = CodasipISD::MoiGroup7;
    PrepareMeta();
  }
};
MoimMoiGroup7Class MoimMoiGroup7;
```

Ako je vidieť, je to štruktúra, ktorá je rozšírením triedy `DagPattern` definujúcej spoločné vlastnosti pre všetky vzory. Obsahuje iba konštruktor, v ktorom napĺňa dáta používané pri vyhľadávaní, uzlami reprezentujúcimi hľadanú inštrukciu. Uzly typu `PdummyOper` označujú operand predchádzajúceho uzlu, ktorý už nebude súčasťou novej inštrukcie, preto nás jeho typ nezaujíma. Je však potrebné skontrolovať jeho dátový typ. Operand tiež obsahuje identifikátor v podobe čísla. To značí poradie operandu v novej inštrukcií. Ak majú dva operandy vo vzore zhodné číslo, ide o rovnaký operand. Konštruktor následne zavolá funkciu, ktorá pomôže inicializovať dáta vyhľadávania. Nakoniec sa vytvorí premenná reprezentujúca tento vzor, ktorá sa použije v cykle popísanom vyššie.

Implementácia štruktúry `DagPattern` sa nachádza v súbore `CodasipBaseMoiLowerer.h` Táto štruktúra reprezentuje všeobecné rozhranie vzorov. Obsahuje metódy a dáta, ktoré sú potrebné pri vyhľadávaní, ako napríklad zoznam koreňov inštrukcie a zoznam už nájdených uzlov. Celá hľadaná inštrukcia sa vlastne skladá zo zoznamu koreňov, pričom koreň je sám štruktúra typu `PNode`.

Tá reprezentuje jeden hľadaný uzol. Medzi dáta, ktoré obsahuje, patrí číselná identifikácia typu uzlu, rôzne dodatočné parametre, ktoré môžu uzly rôznych typov obsahovať a ktoré je potrebné pri nachádzaní kontrolovať a dátový typ výsledku. Obsahuje tiež zoznam ďalších uzlov typu `PNode`. Týmto je vyjadrená dátová závislosť. Uzly, ktoré sú obsiahnuté v inom uzle, predstavujú v grafe jeho parametre. Tak isto je vyjadrená dátová závislosť aj v grafe základného bloku, ktorý tvorí LLVM.

4.2 Prvý krok algoritmu

Hľadanie každého vzoru začína určitou prípravou dát. Prvým krokom je pripravenie vhodnej skupiny uzlov grafu, ktoré by mohli byť koreňmi. Už pri vytváraní vzoru pomocou funkcie `PrepareMeta`, program zistí, koľko a akých uzlov daný musí graf programu obsahovať, aby v ňom bolo možné vzor nájsť. Koreňové uzly sú tie, ktorých výsledok bude konečným výsledkom inštrukcie. Najprv teda prejde všetky inštrukcie grafu a roztriedi ich do matice, kde jeden index predstavuje typ uzlu a druhý konkrétnu uzol daného typu, ktorý sa nachádza v grafe základného bloku. Aby sa čo najskôr odhalilo, či sa vzor dá splniť, skontrolujú sa počty. Ak graf obsahuje dostatočný počet správnych uzlov, ktoré vyžadujú korene vzoru, môže vyhľadávanie pokračovať.

Je veľmi pravdepodobné, že program bude obsahovať viacej uzlov správneho typu, ako bude vzor potrebovať pre svoje koreňové uzly. Preto pre každý z koreňových uzlov spravíme nasledovné. Z množiny uzlov rovnakého typu vyberieme požadovaný počet. Tento výber urobíme všetkými možnými spôsobmi. Ak máme napríklad vzor s dvomi koreňovými inštrukciami `add`, tak zo zoznamu všetkých uzlov typu `add` v grafe vyberáme dvojice. Najprv prvý prvok zoznamu s druhým, prvý s tretím, prvý so štvrtým atď. Potom druhý prvok s tretím, druhý so štvrtým atď. Tým vlastne vytvoríme všetky možné dvojprvkové kombinácie z množiny uzlov `add`. Dvojice, kde je uzol sám zo sebou nepotrebujeme, pretože ten by nakoniec reprezentoval inštrukciu, ktorá by mala o jeden výsledok menej. Tým vznikne zoznam kombinácií. Následne v nich poprehadzujeme prvky. Tak vznikne zoznam permutácií pre každý typ koreňového uzla. Tieto permutácie sa pre každý vzor vytvárajú pre daný základný blok iba raz. Pri opätovnom hľadaní toho istého vzoru v tom istom bloku sa táto časť preskočí. Vzory MOI inštrukcií, ktoré sa v architektúrach vyskytujú a teda sú týmto algoritmom vyhľadávané, majú malý počet koreňových uzlov. Najviac zaznamenaných boli dva. Preto pri vytváraní kombinácií nedochádza k rapídnemu nárastu stavov, ktoré treba prehľadať.

4.3 Hľadanie vzoru v grafe a jeho nahradenie

Hľadanie sa vykonáva postupne po jednotlivých typoch koreňových uzlov. Program si udržuje určitý stav vyhľadávania, teda zoznam uzlov, ktoré boli nájdené ako súčasť vzoru a zoznam operandov okrajových uzlov vzoru, ktorý budú aj operandmi novej inštrukcie. Vždy, pred začatím skúmania nového typu koreňov, je stav vyhľadávania uložený na zásobník. Vyhľadávanie postupuje jednotlivými permutáciami. Vo vzore zoberie koreň zodpovedajúceho typu a začne porovnávať samostatný uzol grafu s uzlom vzoru pomocou metódy `Match`.

Úlohou metódy `Match` je porovnávanie všetkých vnútorných vlastností uzlov. Začína sa to dátovým typom výsledku. Ak je tento uzol vo vzore označený ako `PdummyOper`, okrem typu je treba skontrolovať iba jeho poradie. Ak Operand novej inštrukcie s týmto poradím už bol nájdený, musí to byť uzol ktorý sa práve kontroluje. Ďalšie kontroly sa nad ním nevykonávajú a je zaradený medzi operandy novej funkcie. Inak sa kontroluje typ uzlu a podľa neho aj ostatné vlastnosti, ktoré obsahuje. Metóda `Match` sa vykonáva nad všetkými operandmi daného uzlu a takto sa zanoruje do stromovej štruktúry vzoru, pokiaľ nenarazí na uzol typu `PdummyOper`, alebo pokiaľ sa operandy neprestanú zhodovať.

Po nájdení daného koreňa program aktualizuje zoznam nájdených inštrukcií a nových operandov. Pokračuje ďalším prvkom permutácie, kde postupuje podobne. Ak sa ďalší prvok nezhoduje, je potrebné prejsť na inú permutáciu. Tým pádom sú nájdené uzly predchádzajúceho prvku permutácie neplatné a stav štruktúr obsahujúcich nájdené uzly je potrebné obnoviť. Na to sa využíva zásobník. Naopak, ak sa ďalší prvok permutácie zhoduje, je ešte potrebné vykonať test, či nevznikli nepovolené dátové závislosti. Nepovolená dátová závislosť je, ak výsledok uzla vzoru používa uzol, ktorý do vzoru nepatrí, a zároveň jeho výsledok zase používa iný uzol vzoru. Nahradením takéhoto vzoru by sa z acyklického grafu stal cyklický, čo predstavuje výrazný problém pri plánovaní inštrukcií. Jednej z inštrukcií by totiž chýbal operand. Na vykonanie tejto kontroly sa použije metóda `isPredecessorOf`, ktorá dokáže o dvoch uzloch zistiť, či je jeden predchodcom druhého.

Úspešným nájdením všetkých koreňov daného typu, teda tej správnej permutácie, sa program presunie na skúmanie permutácií iného typu koreňových uzlov. Samozrejme predtým je potrebné uložiť stav nájdených uzlov, nových operandov a permutácie predchádzajúceho typu, na zásobník. Ak sa pri novom type koreňov nenájde vhodná permutácia, je možné sa vrátiť o krok späť a pokúsiť sa pre predchádzajúci typ koreňov nájsť ďalšiu vyhovujúcu permutáciu.

Po nájdení vzoru je možné pristúpiť k vytvoreniu nového uzlu. Vytvorí sa teda nový objekt triedy `SDValue`, kde typ uzlu je číslo zadané pri vytvorení vzoru. Ako operandy sa použijú uzly, ktoré boli pre tento účel nájdené a uložené do špeciálneho zoznamu nových operandov. Pomocou metódy `ReplaceAllUsesOfValueWith`, ktorú poskytuje LLVM, sú výsledky operandov presmerované do novej inštrukcie. Podobne, pre jednotlivé výsledky nového uzlu, ktoré môžu mať rozličné typy, sa použije `ReplaceAllUsesOfValueWith`. Tým sú výsledky novej inštrukcie prepojené so zbytkom grafu a výsledky uzlov, ktoré predstavujú korene, sú nepoužité. Vďaka tomu vznikne izolovaná časť uzlov, ktorá nieje so zbytkom grafu nijako spojená a dá sa jednoducho odstrániť metódou `RemoveDeadNodes`.

Kapitola 5

Peephole optimalizácie

Generátor kódu akéhokoľvek prekladača nikdy nedokáže vygenerovať optimálny kód. Ten často obsahuje operácie, ktoré by sa dali zjednodušiť, teda zapísať menším počtom inštrukcií, dokonca často obsahuje redundantné, ba až nedosiahnuteľné inštrukcie. Jednou z jednoduchých a efektívnych metód, ako kód zefektívniť sú peephole optimalizácie. Táto kapitola popisuje princíp peephole optimalizacií a niekoľko spôsobov, akými bola implementovaná a použitá v iných prekladačoch. Tiež ukazuje spôsob akým môže vzory reprezentovať užívateľ.

5.1 Základný princíp peephole optimalizácií

Kód, ktorý bol vygenerovaný pri preklade, obsahuje veľa redundancií, alebo iné neefektívne inštrukcie, hlavne okolo hraníc základných blokov. Zamedziť týmto prípadom pri generovaní kódu je veľmi zložité, preto sa vytvoril nový priechod prekladača nazvaný peephole optimalizátor, ktorý bude tieto prípady riešiť [11]. Používa pritom okno, tzv peephole, ktoré je obsahuje určitý malý počet inštrukcií, zvyčajne dve až tri. Optimalizácia skúma iba tieto inštrukcie a o ich okolie sa nezaujíma. Ak odpovedajú vzoru, tak ich zamení za výhodnejšiu verziu. Po preskúmaní sa okno posunie o jednu inštrukciu ďalej, pričom jeho veľkosť zostáva nezmenená.

Peephole optimalizácie môžu byť použité pri takmer každom programovacom jazyku a architektúre. Dajú sa pomocou nich optimalizovať všeobecné prípady aj tie, na architektúre závislé. Sú tiež efektívne a jednoduché na implementáciu.

Spôsob, ktorý sa pri implementácií peephole optimalizátoru najčastejšie definoval Lamb v [7]. Táto implementácia je založená na vzoroch, ktoré sa majú v kóde vyhladať. Optimalizátor sa skladá z troch komponent. Kolekcii vzorov, parseru vzorov a modulu na porovnávanie. Generátor kódu na začiatku generuje assembler ako dvojsmerne viazaný zoznam inštrukcií, aby sa dali jednoducho prechádzať. Optimalizátor potom používa vzoroy a snaží sa ich porovnávať s inštrukciami programu kde je to možné. Vzory sú zovšeobecnené špeciálne prípady, preto obsahujú abstraktné symboly a premenné, ktoré sú konkretizované až pri porovnávaní so skutočnými inštrukciami. Vyzerajú nasledovne:

```
< inštrukcia vzoru 1 [podmienka (premenná)]>
...
< inštrukcia vzoru n>
=
<inštrukcia, za ktorú sa zamení inštrukcia vzoru 1>
```

```
...
< inštrukcia , za ktorú sa zamení inštrukcia vzoru n>
```

V tomto prípade je podmienka pri prvej inštrukcií dobrovoľná, preto je v hranatých zátvorkách a na jej kontrolu je potrebná premenná, ktorej sa týka. Tieto podmienky sa nachádzajú iba v časti vzoru. Vzor je nájdený vtedy a len vtedy, ak sa všetky inštrukcie vzoru zhodujú s po sebe nasledujúcimi inštrukciami programu a zároveň sú splnené všetky podmienky, ktoré sa vo vzore nachádzajú. Sekcie v hranatých zátvorkách sa podla Lamba [7] môžu nachádzať aj v časti, ktorá reprezentuje nové inštrukcie, zamenené za vzor. V tom prípade však nereprezentujú podmienky, ale ďalšie akcie, ktoré sa pri zámene vykonajú. Napríklad môže byť obtiažne reprezentovať negáciu premennej v assembleri. V tom prípade je výhodnejšie, aby sa vykonala funkcia, ktorá to zariadi.

Princíp definície vlastných optimalizácií podporuje aj prekladač GCC. Ten umožňuje definovať vzory niekoľko za sebou nasledujúcich inštrukcií pomocou RTL (register transfer language). Vzory sú však dosť zložité a na ich implementáciu je potrebná určitá znalosť RTL.

Čo sa týka vyhľadávania vzorov v programe, existuje niekoľko stratégií [2]. Takmer každá z nich však začína prehľadávanie od konca. To znamená, že ako prvá sa hľadá posledná inštrukcia vzoru a postupuje sa späť k prvej, nie podľa postupu vykonávania programu, ako to používali prvé peephole optimalizácie. Výhodou tohto prístupu je, že optimalizácie sa môžu hneď vykonávať aj nad novo vytvorenými inštrukciami, ktoré vznikli ako výsledok predchádzajúceho použitia niektorého pravidla , a ich predchodcami. Tým nemusí byť optimalizácia spustená znovu ako v prípade hľadania zhody od začiatku.

5.2 Spôsoby implementácie optimalizátoru a vyhľadávania vzorov

Ako bolo spomenuté v predchádzajúcej kapitole, existuje niekoľko spôsobov vyhľadávania vzorov v zdrojovom kóde. Peephole optimalizátory sa tiež líšia vo svojom návrhu.

Pôvodne boli peephole optimalizácie implementované ručne programátorom, ktorý musel najprv identifikovať vhodné vzory, ktoré je možné zjednodušiť, a navrhnúť, za čo sa budú nahrádzať, potom implementovať optimalizáciu. Prvý prekladač, ktorý využíval tieto optimalizácie bol podla [2] vytvorený McKeemanom . Bol to ručne písaný prechod v prekladači jazyka Gogol.

Tento druh optimalizácií však môže využívať aj iné nástroje alebo časti prekladaču na generovanie vzorov z popisu cieľovej architektúry. Tento prístup má svoje výhody. Napríklad dokáže vygenerovať veľké množstvo vzorov, teda výsledný kód je optimálnejší. Taktiež prekladač môže byť ľahko prepísaný na podporu inej architektúry. Napríklad PO [5] je optimalizátor s automaticky generovanými pravidlami. Vstupom PO je popis architektúry a prekladaný program. Potom postupuje nasledovne. Najprv zistí účinok každej inštrukcie assembleru, ktorá môže byť pre danú architektúru vygenerovaná, a reprezentuje ju ako vzor prechodu medzi registrami. Tým sa vytvorí obojsmerná prekladová gramatika medzi týmito dvomi reprezentáciami. Potom je každá dvojica po sebe nasledujúcich inštrukcií analyzovaná a skonvertovaná na equivalentné vzory prechodov konkretizovaných registrov. To pomocou gramatiky získanej v predchádzajúcej analýze. Následne sú tieto vzory zjednodušené. Výsledné jednoduché vzory sú opäť pomocou gramatiky spätne prevedené na inštrukcie assembleru. PO môže byť rôzne modifikované, napríklad aby dokázalo zameniť

tri za sebou nasledujúce inštrukcie za jednu, alebo inštrukcie, ktoré nenasledujú hneď za sebou.

Tanenbaum [2] predstavil iný prístup, ktorý vykonáva peephole optimalizáciu nad vnútorným kódom prekladača. Optimalizácia by mala byť nezávislá na frontende a backende. Vďaka tomu je tiež prenositeľný na rôzne architektúry a nepotrebuje pri tom žiadny jej popis. Pri nachádzaní vhodných inštrukcií používa množinu ručne písaných vzorov. Dizajn tejto optimalizácie neponúka žiadne špecifické stratégie. Výhodou tohto prístupu je, že obchádza problémy s alokáciou registrov. Na druhej strane, niektoré prípady, ktoré je potrebné zoptimalizovať sa vyskytujú až po alokácií registrov.

Všetky tieto prípady používajú podobný systém vyľadávania vzorov, pomocou Knuth-Maris-Prattovho a Boyer-Moorovho algoritmu. To sú klasické prístupy k vyhľadávaniu reťazcov v texte. V tomto prípade majú aj vzory podobu textu. Obsahujú špecifické poradie znakov a premenné. Vzory sú reprezentované regulárnymi výrazmi, ktoré poskytujú formát na vyjadrenie poradia znakov, ktoré sa majú vyhľadať. To umožňuje abstrakciu nad jednoduchými znakmi. Ak reťazec reprezentuje valídny výskyt vzoru, potom je vzor nájdený. Ak sa nezhoduje čo len jeden znak, vzor sa nenašiel. Často sa používa deklaratívna špecifikácia pravidiel. To umožňuje jednoduchý zápis pravidiel a rýchle spracovanie pravidla. Možnosť združovať regulárne výrazy, spätne sa na ne odkazovať a tým používať už definované vzory v iných, predstavuje veľkú výhodu. Programátor nemusí ukladať vstup porovnávania do pomocných premenných, čo vedie iba ku komplikovanému kódu. Takýto prístup založený na vyhľadávaní reťazcov je efektívny, ale problém je, že iba spracováva syntax programu a nemá žiadnu informáciu o sémantike inštrukcie, ktorú sa práve snaží nájsť. Hľadanie vzorov by však nemalo byť považované iba za prehľadávanie reťazca.

Pre objektovo orientované programovacie jazyky, v ktorých je implementovaný optimalizátor, je najmenšou jednotkou informácie objekt. Ten je charakterizovaný svojimi dátami a operáciami, ktoré sa nad nimi dajú vykonávať. Na použitie vyhľadávania vzorov s reťazcami a regulárnymi výrazmi musí programátor pracovať na nižšej úrovni abstrakcie. Absencia vyhľadávania vzorov v objektovo orientovaných jazykoch je cítiť hlavne pri spracovávaní zdrojového kódu. Visser [12] popísal prístup ako podporovať vyhľadávanie vzorov aj v objektovo orientovaných jazykoch bez nutnosti ich rozšírenia. Vzor je v tomto prístupe vytvorený ako objekt. Ten obsahuje metódy, ktoré s ním manipulujú, napríklad určujú splniteľnosť podmienok zadaných so vzorom. Vyhľadávanie pracuje s objektami a väčšinou nieje výhodné používať algoritmy na vyhľadávanie textu, pretože inštrukcia je reprezentovaná iným spôsobom. Tento prístup implementuje vyhľadávanie vzorov ako vyhľadávanie objektov v objektovom grafe.

Z predchádzajúcich prípadov vieme, že dosiahnuť prenositeľnosť a platformovú nezávislosť sa peephole optimalizáciam podarilo dosiahnuť. Jeden z problémov, ktorý však zostal bola rýchlosť. V snahe prekonať ho bolo potrebné znížiť celkovú dobu trvania generovania kódu a vykonávania optimalizácií. Ako bolo spomenuté na začiatku sekcie, vykonávať tieto optimalizácie pri generovaní kódu by viedlo k príliš zložitému kódu, v ktorom by sa muselo ošetriť veľa prípadov. Takže cieľom bolo zachovať efektivitu oboch prechodov bez prílišného zamerania algoritmov na jeden z nich.

Koncept kombinácie týchto dvoch fáz, teda generovania kódu a peephole optimalizácie do jednej, podla [2] popísali Fraser a Wendt. Predstavili prepisovací systém založený na všeobecných pravidlách, ktorý vykonáva nachádzanie vzorov pomocou hashovania. Vzory tiež obsahujú podmienky, ako ich predstavuje Lamb, popísané vyššie. Tento systém šetrí čas tým, že dosiahne efektívnu generáciu kódu v jednej fáze a nie v niekoľkých jednotlivých fázach.

Najnovšie práce sa väčšinou zaoberajú superoptimalizáciami [4], teda nahradením časti kódu skutočne optimálnou postupnosťou inštrukcií, generovaním pravidiel, alebo verifikáciou existujúcich optimalizácií.

5.3 Použitie

Použitie peephole optimalizácií sa dá rozdeliť do niekoľkých základných skupín [2].

Eliminácia redundantného kódu.

Generátory kódu dokážu vygenerovať značné množstvo zbytočného kódu, ktorý môže byť odstránený bez toho aby to malo vplyv na program. Napríklad inštrukcie, ktoré nasledujú za podmienkou, ktorá nebude nikdy splnená. Inštrukcie načítania a ukladania, ktoré pracujú s tým istým registrom ešte predtým, než bol zmenený. Zbytočné inštrukcie, ako napríklad pričítanie nuly, dvojitá negácia alebo niečo iné.

Vylepšenie dátového toku.

Generátor kódu môže vygenerovať inštrukciu podmieneného skoku, ktorá skáče na iný skok. Program môže byť zefektívnený tým, že sa zmení adresa skoku, v niektorých prípadoch sa môže druhý skok vyradiť. Ak je pred skokom inštrukcia porovnania, môže byť vyradená a skok nahradený podmieneným skokom, ktorý zachová logiku programu.

Zjednodušenie algebraických výrazov.

Napríklad šírenie konštanty, alebo skladanie konštanty. Tieto prípady je možné vypočítať už pri preklade. Niektoré operácie, napríklad násobenie číslom, ktoré. Je mocnina dvoch, sa dajú nahradiť za jednoduchšie varianty ako posun doľava. Logické operácie je tiež možné zjednodušiť pomocou booleovej algebry. Nakoniec je možné preorganizovať niektoré inštrukcie využívajúc princíp komutativity a asociativity a tým môžu vzniknúť nové možnosti algebraických optimalizácií.

Architektúrne špecifické inštrukcie

Na získanie najefektívnejšieho kódu je potrebné čo najviac využiť špeciálne inštrukcie architektúry. Tie napríklad dokážu pomocou jednej inštrukcie implementovať zložitejšie operácie, na ktoré by bolo inak treba niekoľko inštrukcií. Napríklad inštrukcia ktorá pri návrate z funkcie upraví začiatok zásobníku a zároveň skočí na návratovú adresu, alebo použitie inštrukcie s rovnakou sémantikou, ktorá je však menšia vzhľadom na veľkosť kódu. Použitie špecifických inštrukcií môže byť podmienené, napríklad používaním špeciálneho registru, či hodnotou konštanty, ktorú používa. Preto je peephole optimalizácia najlepším miestom pre tento druh zámeny. Program je už pred ňou zložený z platných inštrukcií, preto ak podmienky nie sú splnené, nieje potrebné generovať iné inštrukcie, ale iba ponechať stávajúce. Optimalizácie, ktoré sú implementované vrámci tejto diplomovej práce sa zaoberajú práve touto skupinou. Prekladač LLVM totiž už obsahuje prechody, ktoré riešia predchádzajúce typy inštrukcií. Sú implementované na úrovni vnútorného kódu prekladača.

Kapitola 6

Peephole optimalizácia na veľkosť kódu

Táto kapitola opisuje implementáciu peephole optimalizácie v LLVM. Optimalizácia sa zameriava na zmenšenie veľkosti kódu. Využíva pritom fakt, že architektúra obsahuje viacero inštrukcií, ktoré majú rovnakú sémantiku, ale rozdielnu veľkosť. V tomto prípade je vzory možné generovať, bez zásahu užívateľa.

6.1 Optimalizácia na veľkosť kódu

V niektorých architektúrach sa stáva, že obsahujú inštrukcie s rovnakou sémantikou viackrát. Často sa líšia iba v type operandov. Jedna inštrukcia prijíma oba operandy ako hodnotu registru, iná zase berie druhý operand ako konštantu. Ďalšia má rovnaké typy operandov, ale ten, ktorý predstavuje okamžitú hodnotu, je reprezentovaný na menšom počte bitov. Preto hodnota, ktorú obsahuje môže byť iba z menšieho rozsahu. Vďaka tomu je však na reprezentáciu inštrukcie potrebný menší počet bitov. Takýto prístup je možné použiť aj pri registroch. Súčasťou ich názvu býva číselné označenie. Menšie čísla sa dajú reprezentovať na menšom počte bitov, čo taktiež zmenšuje veľkosť inštrukcie. Na úrovni spracovania inštrukcií v prekladači LLVM sa dá táto vlastnosť reprezentovať pomocou registrových tried. Tie môžu združovať registre, ktorých číslo sa dá reprezentovať na rovnakom počte bitov.

Architektúra, na ktorej je tento prístup uplatnený najviac má takmer pre každú aritmetickú a logickú inštrukciu aj jej menší ekvivalent. Dokonca aj inštrukcie vzoru obsahujú svoje zmenšené verzie.

6.2 Reprezentácia vzorov

Každá peephole optimalizácia potrebuje mať funkciu, ktorá určí, aké vzory sa majú vyhľadať. Táto ich nájde v metóde `GetSmallerInst`. Optimalizácia nahrádza jednotlivé inštrukcie za ich menšie varianty, preto metóda obsahuje jeden veľký príkaz `switch`, ktorý obsahuje vetvu pre každú inštrukciu architektúry. Ten podľa operačného kódu aktuálnej inštrukcie naplní pole operačnými kódmi inštrukcií s rovnakou sémantikou, ale menšou veľkosťou. Ak inštrukcia nemá svoj menší variant, príkaz `switch` vráti prázdne pole. Užívateľ nemusí poznať, ktoré inštrukcie sú sémanticky ekvivalentné, obsah metódy sa vygeneruje automaticky pri tvorbe prekladača.

Ďalšia metóda, ktorej obsah sa generuje automaticky, je `GetImmProperties`. Jej obsah je veľmi podobný predchádzajúcej metóde. Príkaz `switch` však obsahuje iba tie inštrukcie, ktoré majú operand typu okamžitá hodnota. Metóda poskytuje informácie o tom na koľkých bitoch je hodnota reprezentovaná, či sa jedná o znamienkovú hodnotu a či hodnota nieje zaokrúhlená na určitý počet bitov. To všetko sú parametre potrebné pre funkciu, ktorá určuje, či môže byť konkrétna hodnota reprezentovaná na určitom počte bitov.

6.3 Implementácia prechodu

Optimalizácia je implementovaná ako prechod nad strojovými inštrukciami prekladaného programu. Využíva možnosti, ktoré ponúka LLVM na vytvorenie takého prechodu. Konkrétne dedí triedu `MachineFunctionPass`, ktorá dovoľuje prechádzať kód a vykonávať úpravy v kóde funkcií.

Optimalizácia začína tým, že prechádza každý základný blok funkcie a v ňom každú inštrukciu od konca bloku. Následne použitím metódy `GetSmallerInst` zistí, či existuje menšia, sémanticky ekvivalentná inštrukcia. Pokiaľ ano, program pokračuje skontrolovaním ostatných podmienok. Ak bolo nájdených viacej ekvivalentných inštrukcií, program pokračuje skúmaním, či je inštrukciu možnú zameniť, aj vtedy ak už bola nájdená jej náhrada. To ale iba za podmienky, že nová náhrada by bola menšia ako predchádzajúca. Keďže náhradné inštrukcie niesú zoradené podľa veľkosti otestuje sa každá, ktorú metóda `GetSmallerInst` ponúkne.

Hoci sú inštrukcie sémanticky zhodné, treba skontrolovať aj ich operandy. To začína kontrolou ich typov. Ak bol ako prvý operand register, musí byť aj prvý operand menšej inštrukcie register, atď.. Naviac, menšie inštrukcie, môžu obsahovať iba hodnoty menšieho rozsahu ako pôvodné, preto je potrebné pri registroch skontrolovať v akej registrovej triede sa nachádza. Ak v tej, ktorú používa aj nová inštrukcia, je to v poriadku. Pri okamžitých hodnotách je tak isto potrebné skontrolovať, či sa na novom počte bitov dá reprezentovať daná hodnota. Na to slúži funkcia `CheckImmImpl`, ktorá potrebuje hodnotu operandu a vlastnosti novej inštrukcie získané z metódy `GetImmProperties`.

Môže sa stať, že nová inštrukcia potrebuje, aby výsledok bol uložený v rovnakom registri v ktorom je jej prvý operand. Táto potreba je reprezentovaná v popise inštrukcie a ak ju pôvodná inštrukcia nespĺňa, nemôže sa nahradiť za menší variant. Existujú prípady, v ktorých táto podmienka nieje splnená, ale inštrukcia je komutatívna. Preto ak pri týchto testoch neuspeje a je komutatívna, vymenia sa jej parametre a kontroly sa zopakujú. Nakoniec, ak sa našla vhodná náhrada, tak sa zmení deskriptor inštrukcie a operandy zostanú zachované.

Keďže tento prechod dokáže zamienať aj inštrukcie skoku, vykoná sa viackrát. Najprv po alokácií registrov a naplánovaní inštrukcií. Potom, keďže sú známe výsledné veľkosti blokov, sa vykoná kontrola skokov. Nakoniec sa zase vykoná tento prechod, ale iba pre inštrukcie skoku, ktoré nahradí za ich menšie verzie, ak je to možné.

Kapitola 7

Všeobecná peephole optimalizácia

Niektoré prípady nevýhodných inštrukcií závisia čisto na architektúre. Preto je potrebné mať systém, ktorému dokáže sám užívateľ zadať, ktoré kombinácie inštrukcií sú neefektívne a ako ich zoptimalizovať. Tieto prípady nieje možné automaticky generovať. Nasledujúca kapitola popisuje implementáciu takejto peephole optimalizácie

7.1 Príklady vzorov

Táto peephole optimalizácia bola testovaná na architektúre, ktorá obsahuje 32 a 64 bitové registry a inštrukcie, ktoré ich používajú. Na prechod z 32 bitovej hodnoty na 64 bitovú sa používajú inštrukcie kopírovania, konkrétne `SUBREG_TO_REG`. Pri prechode opačným smerom zase inštrukcia `COPY`. Pri kopírovaní pomocou `SUBREG_TO_REG` však inštrukcia zoberie hodnotu znamienkového bitu a skopíruje ju do ďalších tridsiatich dvoch bitov. Ak je hodnota znamienka jedna, tak sa nové bity naplnia hodnotou jedna. Keď sa následne na daný register pozerá ako na 64 bitový, tak je jeho hodnota iná ako v 32 bitovom registry. Preto je potrebné vygenerovať naviac inštrukciu, ktorá túto hodnotu skopíruje do výsledného registru, ale pritom zachová znamienko. Takou je inštrukcia `zero_ext_32_64`. Problémom je, že v skutočnosti sa skladá z dvoch inštrukcií assembleru a to posunu doľava o 32 bitov a následne spätnému logickému posunu doprava o rovnaký počet bitov. Tým sa prvých 32 bitov naplní nulami a hodnota registru bude rovnaká aj na 64 bitoch.

Existuje však niekoľko prípadov, pri ktorých je zo sémantiky použitých inštrukcií jasné, že pred kopírovaním hodnoty do väčšieho registru sa v menšom nebude nachádzať záporná hodnota, preto je možné inštrukciu nulového rozšírenia nahradiť klasickými inštrukciami `SUBREG_TO_REG` a `COPY`. LLVM dokonca obsahuje transformácie nad kódom, ktoré tieto inštrukcie napokon eliminujú, takže v skutočnosti sa ušetria dve inštrukcie.

Prvý vzor je nasledujúci.

```
r1 = i_load_32_opc_loadw_regs32_simm12_regs    r2,    imm
r3 = zero_ext_32_64        r1
```

Tento vzor obsahuje inštrukciu load na načítanie hodnoty z pamäte, pričom adresa hodnoty je zadaná pomocou operandov. Vypočíta sa ako súčet obsahu registra, ktorý je prvý operand a okamžitej hodnoty. Druhá inštrukcia je rozšírenie a kópia obsahu 32 bitovéh registru do 64 bitového. Ako vstupný operand používa výsledok predchádzajúcej inštrukcie. Danú sekvenciu je možné nahradiť inštrukciou

```
r3 = i_load_opc_loadwu_regs_simm12_regs    r2,    imm
```

Tá tiež načíta 32 bitovú hodnotu z pamäte ale následne ju rozšíri na 64 bitov, bez zmeny znamienka. V tom prípade, ak by aj bol znamienkový bit nastavený na hodnotu jedna, bude zvyšných 32 bitov naplnených nulou, čím sa zachová načítaná hodnota.

Identický vzor predstavujú inštrukcie

```
r1 = i_load_32_opc_loadw_regs32_simm12_regs_SPEC_CLONE
    r2 ,    imm
r3 = zero_ext_32_64    r1
```

Tie sa nahradia za

```
r3 = i_load_opc_loadwu_regs_simm12_regs_SPEC_CLONE    r2 ,    imm
```

Tento prípad sa od predchádzajúceho líši iba v tom, že hodnota druhého operandu inštrukcie load je pevne daná ako nula.

Ďalší vzor redstavujú inštrukcie

```
r1 = i_logi_32_opc_andi_regs32_regs32_simm12    r2 ,    imm
r3 = zero_ext_32_64    r1
```

Prvá vykonáva logickú operáciu and nad hodnotou uloženou v registre a okamžitou hodnotou. Druhá inštrukcia rozširuje 32 bitový výsledok prvej na 64 bitový. Ak je hodnota druhého parametru taká, že neobsahuje na znamienkovom bite hodnotu jedna, tak po operácií and bude aj znamienkový bit výsledku obsahovať nulu. V tom prípade je možné obsah registra rozšíriť na 64 bitov aj znamiekovo, hodnota sa nezmení. Preto je možné danú inštrukciu nahradiť za inštrukciu

```
r3 = i_comp_2reg_imm_opc_andi_regs_regs_simm12    r2 ,    imm
```

Tá vykonáva operáciu and a jej výsledok rozšíri z 32 na 64 bitov.

Posledný vzor obsahuje inštrukciu, ktorá bitovú reprezentáciu hodnoty registra logicky posunie doprava o počet miest zadaný okamžitou hodnotou ako druhým parametrom. Následne je výsledok zase rozšírený druhou inštrukciou na 64 bitov.

```
r1 = i_comp_2reg_imm_shift32_opc_srliw_regs32_regs32_shifthalf_imm
    r2 ,    imm
r3 = zero_ext_32_64    r1
```

V prípade ak sa hodnota bitovo posunie aspoň o jedno miesto znamená, že bitová reprezentácia danej hodnoty nebude na znamienkovom bite obsahovať jednotku. Preto posúvanú hodnotu netreba explicitne rozširovať nasledujúcou inštrukciou vzoru. Namiesto toho je možné ponechať prvú inštrukciu bezo zmeny a nahradiť poslednú. Inštrukcie, ktoré sú po náhrade sú nasledovné:

```
r1 = i_comp_2reg_imm_shift32_opc_srliw_regs32_regs32_shifthalf_imm
    r2 ,    imm
r3 = SUBREG_TO_REG    r1
```

Vzor sa síce nahradí zase za dve inštrukcie, ale ako bolo spomenuté, `SUBREG_TO_REG` sa eliminuje v neskoršom priechode.

7.2 Formát vzoru

Na to, aby mohol užívateľ vytvárať svoje vlastné optimalizácie, musí existovať neaký mechanizmus, pomocou ktorého dokáže definovať vzor. Niektoré spôsoby boli predstavené v predchádzajúcej kapitole. Spôsob, ktorý bol implementovaný sa nimi inšpiruje a vyzerá nasledovne.

```
peephole_pattern pat1, ok (0),
{ imm_0 = immop(), reg_gpr64_1 = regop(reg_gpr64),
  reg_gpr32_1 = regop(reg_gpr32), reg_gpr32_2 =
  regop(reg_gpr32) },

"foundInst[0]->getOperand(2).getImm() -> 0"
,
i_comp_2reg_imm_shift32_opc_srliw_regs32_regs32_shifthalf_imm_
    reg_gpr32_1, reg_gpr32_2,    imm_0;
zero_ext_32_64    reg_gpr64_1,    reg_gpr32_1;
,
i_comp_2reg_imm_shift32_opc_srliw_regs32_regs32_shifthalf_imm_
   reg_gpr32_1, reg_gpr32_2,    imm_0;
SUBREG_TO_REG     reg_gpr64_1,    0,    reg_gpr32_1,    1;
```

Prvý riadok obsahuje definíciu nového vzoru spolu s jeho názvom a prípadnými inými parametrami, napríklad prioritou vzoru. Hranaté zátvorky obsahujú definíciu operandov, ktoré budú vo vzore použité. V tomto prípade ide o jednu okamžitú hodnotu, jeden 32 bitový a jede 64 bitový register. Za nimi, v úvodzovkách, nasledujú podmienky, ktoré musí vzor inštrukcie spĺňať, aby mohol byť nájdený. Je jedno pre ktorú inštrukciu vzoru musia podmienky platiť, je potrebné ich uviesť ich v tejto časti. Zatiaľ je možné vyjadriť ich iba v jazyku C, preto musí užívateľ poznať aspoň dátové štruktúry, v ktorých sú nájdené časti vzoru uchovávané, aby sa mohol odkázať na správne operandy. V tomto prípade musí byť posledný operand druhej inštrukcie vzoru, kladný. Toto pole nemôže zostať prázdne ani ak vzor žiadne podmienky nemá. Vtedy je potrebné zadať do úvodzoviek true. Za prvou čiarkou nasledujú inštrukcie vzoru. Za druhou čiarkou sú inštrukcie, ktoré majú vzor nahradiť. Prvý operand inštrukcie predstavuje vždy výsledok. Ak majú operandy rovnaký identifikátor, znamená to, že sa na danom mieste nachádza ten istý operand. Tým sa dá jednoducho vyjadriť, že druhá inštrukcia vzoru používa výsledok prvej inštrukcie vzoru, alebo, že inštrukcia, ktorá nahradí prvú inštrukciu vzoru, bude mať rovnaké operandy.

7.3 Reprezentácia vzoru v optimalizácií

Aby sa so vzorom dalo v optimalizácií pracovať, musí byť spracovaný a informácie, ktoré obsahuje reprezentované pomocou objektov.

Základná trieda je nazvaná `Pattern`. Tá je spoločná pre všetky vzory, ktoré ju dedia. Ako dáta obsahuje pole inštrukcií, ktoré slúžia ako vzor a iné pole nových inštrukcií, ktoré nahradia tie zo vzoru. Tieto polia obsahujú objekty triedy `PatternInstr`.

Okrem dát obsahuje trieda `Pattern` aj užitočné metódy na prácu so vzorom. Medzi ne patria jednoduché metódy na napĺňanie dát objektu, získavanie informácií ako sú operačný kód, či jednotlivé objekty triedy `PatternInstr`, ktoré sú uložené v poliach objektu `Pattern`. Medzi zložitejšie z metód patria tie, ktoré zisťujú, či zadaná inštrukcia vzoru zdieľa nejaký

operand s inou inštrukciou vzoru, alebo nejakou inštrukciou, ktorá bude vzor nahradzovať. Tie sa využijú pri určovaní operandov nových inštrukcií. Taktiež dokáže zistiť, či je inštrukcia vzoru dátovo prepojená s inou inštrukciou vo vzore, ktorá ju predchádza, alebo je na nich úplne nezávislá. Nakoniec trieda `Pattern` definuje virtuálnu metódu `SpecialCondition`, ktorá slúži na vyhodnotenie podmienok, ktoré sú zadané spolu so vzorom.

Trieda `PatternInstr` predstavuje jednotlivé inštrukcie. O každej z nich vie jej operačný kód a identifikátory jej operandov. Tie sú reprezentované iba celými číslami začínajúcimi od nuly, pričom prvý z nich predstavuje výsledok inštrukcie. Slúžia na vyjadrenie dátových závislostí medzi vzorom a časťou, ktorá ho má nahradiť, ako aj medi inštrukciami vzoru samými. Poslednou dátovou položkou je index, ktorý určuje, ktorá inštrukcia vzoru je sémanticky ekvivalentná s touto inštrukciou. Táto hodnota sa zadáva iba pre tie inštrukcie, ktoré nahradzujú nejakú jednu inštrukciu vo vzore. Nakoniec obsahuje trieda iba jednoduché metódy na zadávanie a získavanie zadaných dát.

Poslednou je trieda reprezentujúca konkrétny vzor. Tá je rozšírením triedy `Pattern` a obsahuje konštruktor, ktorý napĺňa dáta triedy potrebnými informáciami získanými zo zadaného vzoru. Implementuje tiež funkcionalitu abstraktnej metódy `SpecialCondition`, ktorá je vlastne príkaz `return` s argumentom, ktorý bol zadný pri špecifikácií vzoru v úvodzovkách. Pre každý vzor existuje iná trieda, preto je potrebné ich deklarácie generovať. Trieda pre vzor zobrazený v tejto kapitole vyzerá nasledovne.

```
struct pat1Struc: public Pattern
{
  pat5Struc()
  {
    int opId1[] = {0, 1, 2};
    addPatIns(PatternInstr(Codasip::i_2reg_imm_shift32_opc_srliw,
      opId1, 3));
    int opId2[] = {3, 0};
    addPatIns(PatternInstr(Codasip::zero_ext_32_64, opId2, 2));
    int opId3[] = {0, 1, 2};
    addResIns(PatternInstr(Codasip::i_2reg_imm_shift32__opc_srliw
      , opId3, 3), 0);
    int opId4[] = {3, 4, 0, 5};
    addResIns(PatternInstr(Codasip::SUBREG_TO_REG, opId4, 4), 1);
  }
  bool SpecialCondition(std::vector<MachineInstr*> foundInst)
  {
    return (foundInst[0]->getOperand(2).isImm() &&
            foundInst[0]->getOperand(2).getImm() > 0);
  }
};
```

7.4 Implementácia optimalizácie

Rovnako ako predchádzajúca peephole optimalizácia na veľkosť kódu, aj táto je implementovaná ako rozšírenie triedy `MachineFunctionPass`, ktorú dedí. Optimalizácia začína postupným prehľadávaním programu od prvej inštrukcie prvého základného bloku. Pri každej z nich skontroluje prvú inštrukciu všetkých zadaných vzorov. Ak sa operačné kódy

rovnajú, optimalizácia vie s kontrolou ktorého vzoru má pokračovať. V tomto prístupe nehľadí na to, či sa typy operandov vzorovej a výslednej inštrukcie zhodujú, ale predpokladá, že užívateľ zadal korektné vzory. Po nájdení prvej inštrukcie sa zameria na konkrétny vzor.

V našom prípade majú všetky vzory svoje vzorové inštrukcie dátovo previazané a pomocou metód triedy `Pattern` to dokáže optimalizácia zistiť. Toho následne využije. LLVM uchováva o každom operandovi, ktorý je register, záznam, v akých inštrukciách bol definovaný a použitý. Navyše sa táto optimalizácia vykonáva pred alokáciou registrov, kód je teda v SSA forme. To znamená, že výsledok každej inštrukcie je uložený do nového virtuálneho registru. Optimalizácia teda začne prechádzať jednotlivé použitia výsledného registru, čo umožňuje preskočiť všetky inštrukcie, ktoré sa medzi použitiami nachádzajú. Tým dokáže nájsť vzory, kde inštrukcie nenasledujú bezprostredne za sebou, môžu sa dokonca nachádzať v rôznych základných blokoch bez väčšej námahy. Je pri tom jasné, že sa pohybuje v smere toku programu a nie proti nemu, lebo nájdená prvá inštrukcia daný register definuje a ten nemôže byť pred svojou definíciou použitý.

Pri tomto hlbšom prehľadávaní sa môže stať, že je register používaný aj inou inštrukciou, ktorá do vzoru nepatrí. V tomto prípade musíme zistiť, či sa táto inštrukcia nahradí za nejakú s ekvivalentnou sémantikou. V tom prípade vieme, že aj po nahradení vzoru bude v programe inštrukcia, ktorá dokáže tento výsledok vyprodukovať a teda ešte stále je možné vzor zameniť. To je možné vďaka údaju v triede `PatternInstr`. Ak taká inštrukcia existuje, je potrebné si toto použitie výsledku zapamätať, aby sa dal neskôr tento operand nahradiť za správny.

Ak týmto spôsobom dokážeme nájsť všetky inštrukcie vzoru, pristupujeme ku kontrole podmienky zadanej vo vzore. Tá využíva dátové štruktúry, ktoré obsahujú nájdené inštrukcie.

Ak je aj tento krok úspešný, začína nahradenie vzoru. Pomocou objektu triedy `MachineBuilder` začne optimalizácia vytvárať nové inštrukcie. Pri vytvorení prechádza identifikátory operandov vo vzore a skúma, či inštrukcia nemá spoločný operand s nejakou inštrukciou vzoru. V tom prípade by nemusela vytvárať nový, ale použil by sa už existujúci. Ak to nieje možné, zisťuje, či nemá operandy rovnaké s nejakou už vytvorenou novou inštrukciou. Ako posledný prípad vytvára nové operandy. Táto situácia nastáva iba pri jednom identifikovanom vzore. Jedná sa o inštrukciu `SUBREG_TO_REG` a operandy, ktoré treba pridať sú okamžité hodnoty. Tento prípad sa dá preto ľahko ošetriť. Po vytvorení nových inštrukcií je potrebné správne previazať hodnoty ich výsledkov, pretože výsledný register inštrukcie sa vždy vytvorí nový. Nakoniec nesmie optimalizácia zabudnúť aj na inštrukcie, ktoré niesú súčasťou vzoru, ale používajú výsledok nejakej jeho inštrukcie. Často sa stane, že nová inštrukcia produkuje 64 bitový výsledok, ale tá, ktorá ho používa očakáva 32 bitový. V tom prípade je potrebné vygenerovať novú `SUBREG_TO_REG` inštrukciu, ktorá sa v neskoršom priechode eliminuje. Ako posledný krok sa vymažú inštrukcie vzoru.

Po úspešnom nájdení a zamenení inštrukcií, sú iterátori používané na prehľadávanie kódu invalidované. Preto sa daný základný blok začne prehľadávať od začiatku. Vďaka tomu sa nájdu aj vzory, ktoré môžu vzniknúť pri predošlích zámenách.

Kapitola 8

Generovanie vzorov inštrukcií

Codasip Studio je vývojové prostredie, ktoré umožňuje automatickú generáciu prekladača a iných potrebných nástrojov, z popisu architektúry. Popis architektúry, pre ktorú sa má prekladač vytvoriť, začína v jazyku CodAL. Užívateľ v ňom špecifikuje registre, pamäť a hlavne inštrukcie, ktoré bude architektúra používať. Na vytvorenie prekladača LLVM je potrebný iný popis architektúry. Nástroj, ktorý zabezpečuje generovanie potrebných informácií pre LLVM z preloženého modelu, sa nazýva generátor backendu. Táto kapitola opisuje, ako vytvára generátor backendu vzory, ktoré sú potrebné pre peephole optimalizáciu a nachádzanie MOI inštrukcií.

8.1 Generátor backendu

Popis architektúry začína v jazyku CodAL. Ten umožňuje definovať architektúru procesoru. Výsledkom prekladu tohto jazyka je súbor instruction_semantics.sem, ktorý obsahuje všetky tieto informácie vo forme, ktorú dokáže generátor backendu spracovať. Napríklad inštrukcie v ňom vyzerajú nasledovne.

```
instr i_addiw__opc_addiw__regs32__regs32__simm12__, ok (0),
{ regs32_0 = regop(regs32), regs32_1 = regop(regs32),
  imm_2 = immop() },
%5 = i32 regop(regs32_1);
%6 = i32 immop(imm_2,1,12);
%4 = add(%5,%6);
%7 = i1 -1;
regop(regs32_0) = %4;
,
"addw" regs32_0~"," regs32_1~"," imm_2,
imm_2[11,0] regs32_1[4,0] 0b000 regs32_0[4,0] 0b0011011 ,
"",
"el:i_addiw(el:opc_addiw, el:regs32, el:regs32, el:simm12)",
""
,
{{0}}
```

Ako je vidieť, informácie o inštrukcii obsahujú jej názov, typy parametrov, binárne kódovanie a zápis ako bude inštrukcia vyzerať v assembleri. Najväčšou časťou je popis

sémantiky inštrukcie, pomocou jednoduchých príkazov, ktoré predstavujú uzly acyklického grafu, ktorý používa LLVM pri nachádzaní inštrukcií.

Na vytvorenie backendu LLVM však treba popísať architektúru v inom formáte do príslušných súboroch s koncovkou .td. Tie potom použije nástroj TableGen, ktorý z nich vygeneruje backend prekladača. Preto existuje generátor backendu, ktorý najprv všetky tieto informácie načíta do svojich vútorných dátových štruktúr. Následne ich spracuje a podľa potreby vypisuje informácie do .td súborov. Niektoré súbory taktiež potrebujú dodatočné informácie, ktoré dokáže poskytnúť iba generátor backendu. Preto sa vytvorí ich .td verzia, v ktorej sa volá niektorá z metód generátora, na výpis potrebních informácií. Súbory s vygenerovaným textom sú následne preložené, čím vznikne konečný prekladač jazyka C.

8.2 Vzory pre MOI inštrukcie

Prvým krokom je implementovať inštrukciu v jazyku CodAL, ktorá dokáže vykonať viac operácií naraz, pričom jednotlivé výsledky sa uložia do rozdielnych registrov. Generátor backendu následne inštrukcie načíta a uchováva o nich rôzne informácie. Na to aby bola inštrukcia rozpoznaná ako vhodný kandidát na použitie v programe, musí splniť zopár podmienok, ktoré určuje metóda `IsMoiLowSuitable`. Ak ich splniť nedokáže, generátor backendu ju nezaradí medzi vhodné inštrukcie a pri preklade programu nebude nikdy použitá. Na rozpoznávanie je triede `Instructions`, ktorá reprezentuje inštrukcie rozpoznané generátorom beckendu, pridaná metóda `IsMoiLowSuitable`.

Základnou podmienkou je, aby mala inštrukcia viac ako jeden výsledok. Následne sa nemôže jednať o inštrukciu skoku, alebo porovnania. Samozrejme je to tiež zbytočné pre `NOP` inštrukciu, hoci je veľmi nepravdepodobné, že by inštrukcia `NOP` obsahovala viacej výsledkov. Keďže LLVM nepodporuje funkcie, ktoré by mali dva výsledky, ktoré sa ukladajú do dvoch všeobecných registrov. Jeden z nich musí byť preto uložený do fixného registru, teda musí mať explicitne určený konkrétny register, do ktorého sa výsledok uloží. Ako posledný krok je kontrola sémantiky inštrukcie. Povolené sú iba tie, ktorých sémantika obsahuje iba jednoduché operácie, ako sčítanie, odčítanie a iné aritmetické a operácie. Taktiež povoľuje uloženie, alebo načítanie hodnoty z pamäte a iné. Ak inštrukcia spĺňa všetky tieto požiadavky, je označená príznakom `isMoi`.

Hlavná metóda, ktorá sa stará o samotný výpis vzorov do súboru obsahujúceho kód nahradenia, je `PrintMOI_Classes`. Je to metóda triedy `BEGwrapper`, ktorá obaľuje implementáciu generátoru backendu a obsahuje hlavne metódy na vypisovanie potrebných informácií do zdrojových kódov backendu prekladača LLVM. Táto metód prechádza všetky identifikované inštrukcie architektúry a pre tie, ktoré boli identifikované ako MOI vypíše deklaráciu novej triedy vzoru, ktorý inštrukcia reprezentuje. Vzor je predstavený v sekcii **??**. Tiež má za úlohu vypísať koreňové uzly sémantiky inštrukcie. Následne zavolá pomocnú funkciu `PrintMOI_Tree`. Tá sa postará o výpis zbytku sémantiky, pretože sémantika obsahuje tie uzly, ktoré sa budú v grafe vyhľadávať.

`PrintMOI_Tree` je volaná rekurzívne, lebo uzly v sémantike sú taktiež uložené v rekurzívnej dátovej štruktúre. Každé jej nové volanie vygeneruje vo vzore zátvorky, ktoré reprezentujú zanorenie uzlu. Následne pre každý uzol vygeneruje jeho identifikátor a dátový typ. Pri uzloch typu `LOAD` a `STORE` musí rozlíšiť či sa nejedná o rozširujúce načítanie alebo uloženie zaokrúhlenej hodnoty. Ak áno, je potrebné vygenerovať dodatočné informácie, ako z akého typu sa bude hodnota rozširovať, alebo zužovať na aký typ. Taktiež musí byť pri ukladaní a načítaní určené aj zarovnanie adresy. Ak narazí na uzly, ktoré reprezentujú operandy inštrukcie, priradí im zvlášť vyhradený identifikátor, dátový typ operandu a

číslo určujúce poradie operandu v inštrukcií.

8.3 Vzory pre peephole optimalizátor

Peephole optimalizátor na veľkosť kódu taktiež využíva generované vzory. Tie mu určujú aká menšia, ale pri tom sémanticky ekvivalentná inštrukcia existuje pre každú inštrukciu architektúry.

Podobne ako v predchádzajúcom prípade, funkcia prechádza všetky inštrukcie a ak nájde k práve skúmanej nejakú s menšou veľkosťou, pokračuje porovnávaním sémantiky. Porovnávanie prebieha uzol po uzle, pričom všetky, okrem uzlov operandov, sa musia zhodovať. Kontrola operandov potom prebieha v samotnej optimalizácií. Ak prebehli testy v poriadku, vygeneruje sa odpovedajúci vzor.

Kapitola 9

Dosiahnuté výsledky

Peephole optimalizácia na veľkosť kódu bola testovaná na architektúre `codix_helium`. Pri teste sa zaznamenávala veľkosť preloženého programu. Optimalizácia priniesla výrazné zmenšenie preložených programov pri každom programe na ktorom bola testovaná. Hlavným dôvodom je, že daná architektúra obsahuje pre takmer každú inštrukciu aj aspoň jednu, ktorá je menšia, ale zároveň sémanticky ekvivalentná. Výsledné hodnoty sú vyjadrené v bajtoch v tabuľke9.1.

Tabuľka9.2 znázorňuje výsledky použitia všeobecnej peephole optimalizácie. Tá bola testovaná na architektúre `codix_berkelium`. Vzory, ktoré sa v programoch vyhľadávali sú popísané v kapitole **??**. Nie každý testovaný program dané vzory obsahoval, preto sa podarilo dosiahnuť zlepšenia iba v malom počte prípadov. Navyše Vzory sa v programoch vyskytovali v malých množstvách, jeden až desať prípadov v jednom programe. Preto je dosiahnuté zlepšenie, v niektorých prípadoch, tak malé.

Pri testovaní sa zaznamenávala a porovnávala veľkosť výsledného kódu. Keďže optimalizácia aj znižovala počet výsledných inštrukcií programu, mohlo dôjsť aj k zníženiu počtu inštrukcií, ktoré boli pri vykonávaní programu odsimulované. Tento fakt však nemohol byť potvrdený, pretože architektúra obsahovala určité chyby. Testy preto často skončili s chybou bez ohľadu na to, či bola spustená peephole optimalizácia alebo nie. Následná inšpekcia výsledného assembleru potvrdila, že optimalizácia pracovala správne.

Tabuľka 9.3 a 9.4 patria prechodu na nachádzanie inštrukcií s viacerými výsledkami. Tento prechod bol testovaný na architektúre `codasip_urisc`, ktorá obsahuje inštrukcie načítania a ukladania hodnoty registru do pamäte a zároveň inkrementujú hodnotu registra, ktorý obsahuje adresu pamäťového miesta. Vzory takýchto inštrukcií sa taktiež našli iba v niektorých príkladoch. Vyskytovali sa v množstvách od dvoch do päťdesiat nájdených vzorov v programe.

Pri testoch sa zaznamenávala veľkosť výsledného programu, ako aj počet inštrukcií, ktoré simulátor pri vykonávaní programu odsimuloval. Takmer v každom teste, v ktorom sa vzor našiel a špeciálna inštrukcia použila, boli obe z týchto hodnôt horšie. Manuálna inšpekcia kódu assembleru zistila, že problém spôsobuje použitie fixného registru pre jeden z výsledkov. Pred jeho použitím v spomínanej inštrukcii sa najprv jeho hodnota zálohuje do iného registru a po vykonaní inštrukcie a použití inkrementovanej hodnoty sa zase spätne doňho načíta. Prípadne môže byť výsledný kód ešte inak zmenený, v závislosti na tom, ktoré registri sa pre aké inštrukcie použijú. Výsledný kód preto neobsahuje o inštrukciu menej, ale aspoň o inštrukciu naviac.

Tabuľka 9.1: veľkosť programu v bajtoch pre `codix_helium`

	bez optimalizácie	s optimalizáciou
VersaBench-ecbdes	46266	34104
Stanford-Towers	5582	4114
Stanford-Queens	4434	3190
Stanford-Puzzle	6470	4842
Stanford-Perm	5318	3746
Stanford-IntMM	4662	3398
Stanford-Bubblesort	4550	3270
Shootout-sieve	3654	2650
Shootout-methcall	3914	2826
Shootout-matrix	4830	3526
Shootout-lists	4890	3522
rc4	5114	3670
quicksort	3798	2802
Misc-salsa20	4642	3566
md5	9270	7510
isqrt	3670	2654
dhrystone	5694	4262
crc	3582	2630
coremark	12398	8918
blowfish	61950	44620
bitcnt	3458	2514

Tabuľka 9.2: veľkosť programu v bajtoch pre `codix_berkelium`

	s optimalizáciou	bez optimalizácie
local1	6160	6224
local2	6384	6400
local3	6608	6632
local4	6336	6360
local5	1076	1084
local6	956	964
local7	6808	6824
local8	948	956
rc4	2932	2980
Misc-salsa20	2924	2940
crc	1404	1444
coremark	17448	17496
blowfish	82432	82744

Tabuľka 9.3: veľkosť programu v bajtoch pre `codasip_urisc`

	veľkosť kódu s opt.	veľkosť kódu bez opt.	počet nájdených vzorov
VersaBench-ecbdes	56368	56176	21
Shootout-methcall	4476	4456	6
rc4	6208	6196	3
md5	11328	11292	4
isqrt	4204	4184	3
dhrystone	6564	6512	9
coremark	15332	15208	29
blowfish	75176	75208	42
local1	4064	4064	4
local2	3968	3972	2
local3	3988	3984	2
local4	3844	3844	2
local5	4008	4008	2
local6	4120	4092	6

Tabuľka 9.4: počet odsimulovaných inštrukcií pre `codasip_urisc`

	počet ins. s optim.	počet ins. Bez opt.	počet vzorov
VersaBench-ecbdes	19382743	19342743	21
Shootout-methcall	4467101	4433767	6
rc4	397001	397587	3
md5	1016762	983530	4
isqrt	618076	618076	3
dhrystone	5350191	4500190	9
coremark	4485962	4275887	29
blowfish	15631113	15630704	42
local1	154	149	4
local2	105	105	2
local3	115	114	2
local4	68	68	2
local5	88	88	2
local6	376	284	6

Záver

Práca sa zaoberá implementáciou optimalizácií v zadnej časti prekladača LLVM. V rámci práce bola naštudovaná problematika výberu inštrukcií s viacerými výsledkami a problematika peephole optimalizácií.

Práca obsahuje implementáciu troch optimalizačných priechodov. Jeden z nich slúži na vyhľadávanie inštrukcií s viacerými výsledkami, zvyšné dva sú peephole optimalizácie zameriavajúce sa na zmenšenie veľkosti výsledného kódu. Jeden z prechodov povoľuje užívatelom definovať si vlastné vzory, zatiaľčo druhý si dokáže potrebné informácie sám vygenerovať. Akýkoľvek prekladač LLVM je možné rozšíriť o optimalizačné prechody popísané v tejto práci.

Ďalej obsahuje návrh a popis implementácie vzorov, ktoré budú obsahovať všetky potrebné informácie pre spomenuté optimalizačné prechody. Taktiež je v práci definovaný návrh popisu, pomocou ktorého by užívateľ mohol vytvoriť svoje vlastné vzory, ktoré chce v rámci peephole optimalizácie zameniť. Pre platformu CodasipStudio, ktorá umožňuje vygenerovať prekladač pre architektúru popísanú v jazyku CodAL, je implementované automatické generovanie vyššie spomenutých vzorov.

Následne bola implementácia odtestovaná na sade benchmarkov. Výsledky ukázali, že obe peephole optimalizácie priniesli zlepšenie, inštrukcie s viacerými výsledkami naopak zhoršenie. Konkrétne hodnoty a vysvetlenie zhoršenia sú uverejnené v kapitole 9.

Práca môže byť rozšírená o implementáciu ďalšieho peephole prechodu, ktorý bude zamieňať vzory zadané užívateľom. Bude však pracovať nad kódom, ktorý vznikne po alokácií registrov, aby bolo možné optimalizovať aj tie prípady, ktoré vzniknú až po tomto prechode. Taktiež môže byť rozšírená o implementáciu generátora, ktorý dokáže transformovať vzor optimalizácie popísaný užívateľsky prívetivým spôsobom z kapitoly 7.2, na vzor používaný peephole optimalizátorom.

Literatúra

[1] Arnold, M.; Corporaal, H.: Automatic detection of recurring operation patterns. In *Proceedings of the seventh international workshop on Hardware/software codesign*, ACM, 1999, s. 22–26.

[2] Bhatt, M. C. H.; Bhadka, D. H. B.: Peephole Optimization Technique for analysis and review of Compile Design and Construction. *IOSR Journal of Computer Engineering (IOSR-JCE)*, ročník 9, č. 4, 2013.

[3] Blindell, G. H.: Survey on Instruction Selection: An Extensive and Modern Literature Review. *CoRR*, ročník abs/1306.4898, 2013.
URL http://arxiv.org/abs/1306.4898

[4] Buchwald, S.: Optgen: A generator for local optimizations. In *Compiler Construction*, Springer, 2015, s. 171–189.

[5] Davidson, J. W.; Fraser, C. W.: The design and application of a retargetable peephole optimizer. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, ročník 2, č. 2, 1980: s. 191–202.

[6] Koes, D. R.; Goldstein, S. C.: Near-optimal instruction selection on dags. In *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*, ACM, 2008, s. 45–54.

[7] Lamb, D. A.: Construction of a peephole optimizer. *Software: Practice and Experience*, ročník 11, č. 6, 1981: s. 639–647, ISSN 1097-024X, doi:10.1002/spe.4380110608.
URL http://dx.doi.org/10.1002/spe.4380110608

[8] Lattner, C.: LLVM [online]. `http://www.llvm.org`, 2013-07-7 [cit. 2016-10-1].

[9] 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.

[10] Lopes, B. C.; Auler, R.: *Getting Started with LLVM Core Libraries*. Packt Publishing, 2014, ISBN 1782166920, 9781782166924.

[11] McKeeman, W. M.: Peephole Optimization. *Commun. ACM*, ročník 8, č. 7, Červenec 1965: s. 443–444, ISSN 0001-0782, doi:10.1145/364995.365000.
URL http://doi.acm.org/10.1145/364995.365000

[12] Visser, J.: Matching objects without language extension. 2006.

[13] Wilson, T.; Grewal, G.; Halley, B.; aj.: An integrated approach to retargetable code generation. In *Proceedings of the 7th international symposium on High-level synthesis*, IEEE Computer Society Press, 1994, s. 70–75.

[14] Youn, J. M.; Lee, J.; Paek, Y.; aj.: Fast graph-based instruction selection for multi-output instructions. *Software: Practice and Experience*, ročník 41, č. 6, 2011: s. 717–736, ISSN 1097-024X, doi:10.1002/spe.1034.
URL http://dx.doi.org/10.1002/spe.1034

Príloha A

Obsah CD

Priložené CD obsahuje nasledujúcu adresáre a súbory:

- /info.txt – súbor obsahujúci informácie, ako nástroje spustiť a odtestovať.
- /run/ – preložené prekladače testovaných platforiem, na ktorých je možné odskúšať funkčnosť optimalizácií.
- /src/benchmarks – benchmarky na ktorích boli optimalizácie testované.
- /src/local/berkelium – testovacie súbory s krátkzmi programami, na ktorých sa dá overiť platnosť peephole optimalizácií.
- /src/local/urisc – testovacie súbory s krátkzmi programami, na ktorých sa dá overiť platnosť vyhladávania MOI inštrukcií.