VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

OPTIMALIZACE LLVM IR PRO ÚČELY ZPĚTNÉHO PŘEKLADU

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE Bc. JAROSLAV KOLLÁR
AUTHOR

BRNO 2015

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

OPTIMALIZACE LLVM IR PRO ÚČELY ZPĚTNÉHO PŘEKLADU

LLVM IR OPTIMIZATIONS FOR DECOMPILATION

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — Bc. JAROSLAV KOLLÁR
AUTHOR

VEDOUCÍ PRÁCE — Ing. PETER MATULA
SUPERVISOR

BRNO 2015

Abstrakt

Tato práce se zabývá návrhem a implementací optimalizací ve střední části zpětného překladače vyvíjeného firmou AVG Technologies. Úlohou těchto optimalizací je zlepšit čitelnost produkovaného zdrojového kódu a současně vylepšit některé navržené optimalizace v zadní části zpětného překladače. V úvodu jsou poskytnuty základní informace o reverzním inženýrství a zpětných překladačích, které slouží pro účel uvedení do dané problematiky. Poté se nachází popis stavu zpětného překladače před zahájením této diplomové práce. Dále následuje hlavní část práce, která se věnuje popisu návrhu a implementace jednotlivých navržených optimalizací. Poté následuje popis testování optimalizací a shrnutí dosažených výsledků. V závěru práce je diskutován budoucí možný vývoj.

Abstract

This master's thesis describes the design and implementation of optimizations in the middle-end part of a retargetable decompiler developed by AVG Technologies. The purpose of these optimizations is to improve readability of the produced source code and improve existing optimizations in the back-end part. In the introduction, basic information about reverse engineering and decompilation is provided. Then, a description of state of the retargetable decompiler before this work is given. After that, the main part of this work is presented, which describes the design and implementation of the proposed optimizations. Then, the techniques that were used for testing are described. This description is followed by a summary of the achieved results. The present work is concluded by a discussion of possible future development.

Klíčová slova

Reverzní inženýrství, zpětný překladač, zpětný překlad, střední část zpětného překladače, optimalizace, čitelnost zdrojového kódu.

Keywords

Reverse engineering, decompiler, decompilation, middle-end part of a decompiler, optimization, source code readability.

Citace

Jaroslav Kollár: Optimalizace LLVM IR pro účely zpětného překladu, diplomová práce, Brno, FIT VUT v Brně, 2015

Optimalizace LLVM IR pro účely zpětného překladu

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně za odborného vedení Ing. Petra Matuly a konzultanta Ing. Petra Zemka, Ph.D. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

. . . . . .
Jaroslav Kollár
25. května 2015

Poděkování

Děkuji svému vedoucímu Ing. Petru Matulovi a konzultantovi Ing. Petru Zemkovi, Ph.D. za odborné vedení, za poskytnuté rady a za čas, který mi při tvorbě práce věnovali.

© Jaroslav Kollár, 2015.
Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Obsah

1 Úvod **3**

2 Reverzné inžinierstvo a spätný preklad **5**

2.1 Softvérové reverzné inžinierstvo a jeho použitie 5

2.2 Spätný preklad 6

3 Spätný prekladač vyvíjaný spoločnosťou AVG Technologies **8**

3.1 LLVM 9

3.2 Predná časť 9

3.3 Stredná časť 10

3.4 Zadná časť 10

4 LLVM IR **12**

5 Stav spätného prekladača pred zahájením tejto práce **16**

5.1 Stredná časť 16

5.2 Zadná časť 17

5.3 Ciele diplomovej práce 17

6 Návrh optimalizácií **18**

6.1 Usporiadanie PHI uzlov 18

6.1.1 Syntax, sémantika a použitie inštrukcie phi 19

6.1.2 Problémy a ich riešenie 21

6.1.3 Návrh algoritmu 24

6.2 Odstránenie mŕtvych priradení do globálnych premenných 27

6.2.1 Syntax a sémantika inštrukcií súvisiacich s mŕtvymi priradeniami . . 29

6.2.2 Špecifické prípady, pri ktorých nedochádza k optimalizácii 30

6.2.3 Priechod základnými blokmi a ich spracovanie 31

6.2.4 Priechod funkciami a ich spracovanie 33

6.2.5 Informácie uchovávané počas analýzy 34

6.2.6 Princíp odstraňovania mŕtvych priradení 40

6.3 Konverzia globálnych premenných na lokálne 41

6.3.1 Syntax a sémantika inštrukcií súvisiacich s konverziou globálnych premenných na lokálne premenné 44

6.3.2 Špecifické prípady, pri ktorých nedochádza k optimalizácii 45
6.3.3 Princíp konvertovania globálnej premennej na lokálnu premennú . . 45
6.4 Odstránenie nedosiahnuteľných funkcií 45
6.4.1 Syntax a sémantika inštrukcie súvisiacej s odstraňovaním nedosiahnuteľných funkcií 46
6.4.2 Špecifické prípady, pri ktorých nedochádza k optimalizácii 46
6.4.3 Princíp identifikácie nedosiahnuteľných funkcií 47
6.5 Odstránenie kódu za volaniami funkcií, ktoré sa nevracajú 47
6.5.1 Syntax a sémantika inštrukcií súvisiacich s touto optimalizáciou . . . 48
6.5.2 Princíp identifikácie volania funkcie, ktorá sa nevracia 48
6.5.3 Spôsob zabezpečenia odstránenia kódu za volaniami funkcií, ktoré sa nevracajú 49

7 Implementácia optimalizácií 50
7.1 Usporiadanie PHI uzlov 52
7.2 Odstránenie mŕtvych priradení do globálnych premenných 53
7.3 Konverzia globálnych premenných na lokálne 55
7.4 Odstránenie nedosiahnuteľných funkcií 55
7.5 Odstránenie kódu za volaniami funkcií, ktoré sa nevracajú 57

8 Testovanie optimalizácií 58
8.1 Referenčné testy 58
8.2 Jednotkové testy 60
8.3 Regresné a nočné testy 62
8.4 Zhodnotenie 63

9 Dosiahnuté výsledky a ich zhodnotenie 64
9.1 Usporiadanie PHI uzlov 64
9.2 Odstránenie mŕtvych priradení do globálnych premenných 65
9.3 Konverzia globálnych premenných na lokálne 66
9.4 Odstránenie nedosiahnuteľných funkcií 66
9.5 Odstránenie kódu za volaniami funkcií, ktoré sa nevracajú 67
9.6 Celkové zhodnotenie 67

10 Záver 68

A Obsah DVD 74

Kapitola 1

Úvod

V dnešnej dobe je v rámci informačných technológií často kladený dôraz na bezpečnosť. Medzi jedno z najväčších rizík je možné zaradiť rôzne druhý škodlivého softvéru. Typickými príkladmi sú klasické počítačové vírusy, trójske kone, internetové červy a iné. Všeobecné označenie týchto škodlivých programov je možne pomocou pojmu malvér [34]. V súčasnosti dochádza k veľkému nárastu škodlivých softvérov, ako aj k vzniku nových druhov. Veľký vzostup malvéru je možné vidieť pri mobilných zariadeniach a tabletoch. Ide predovšetkým o operačný systém Android [6]. Typickými úlohami škodlivého softvéru je najčastejšie získavanie citlivých dát od užívateľov, poškodzovanie súborov, znepríjemňovanie práce na počítači a podobné [34]. Jednou z možných techník obrany je nespúšťanie a neotváranie nedôveryhodných materiálov. Populárna a osvedčená technika je využívanie antivírusových programov. Antivírusové programy prehľadávajú súbory, sledujú správanie aplikácií a aktivitu systému na pozadí. Pri týchto činnostiach súčasne porovnávajú či napríklad chovanie niektorej aplikácie neodpovedá správaniu sa napadnutého programu. V súboroch zase hľadajú sekvenciu inštrukcií, ktorá odpovedá známemu malvéru [13]. Schopnosť detekovať malvér je však limitovaná kvalitou antivírusového programu. Preto je dôležité dbať na vývoj techník, ktoré umožnia vylepšiť kvalitu programov bojujúcich s malvérom. Jednou z techník, ktorú môžu použiť tvorcovia aplikácii proti malvéru, je softvérové reverzné inžinierstvo. Ako príklad je možne uviesť využitie spätného prekladu nad získaným binárnym súborom. O spätný preklad sa postará spätný prekladač, pričom jeho úlohou je z tohto súboru vygenerovať program vo vysokoúrovňovom jazyku. Následne je možné tento kód preskúmať a vyhodnotiť jeho činnosť [20]. Táto technika je výhodná v tom, že daný súbor nie je potrebné ani spustiť.

Na základe uvedeného je zrejmé, že spätný prekladač v rukách tvorcov antivírusových programov môže znamenať veľký prínos. Za týmto účelom je vyvíjaný aj spätný prekladač spoločnosťou AVG Technologies. Prostredníctvom verejne prístupnej webovej služby si je možné tento spätný prekladač vyskúšať [2]. Spätný prekladač vyvíjaný spoločnosťou AVG Technologies je rozdelený na tri časti. Ide o prednú časť, strednú časť a zadnú časť. Úlohou prednej časti je prevod binárneho kódu platformovo závislej aplikácie do platformovo nezávislého kódu. Výstupom prednej časti je kód v jazyku LLVM IR [23], ktorý spĺňa uvedenú vlastnosť [44]. LLVM IR predstavuje prechodnú reprezentáciu, ktorá poskytuje typovú bezpečnosť, nízkoúrovňové operácie, flexibilitu a schopnosť prevodu z množstva vysokoúrovňových jazykov. Cieľom jeho tvorcov bolo dosiahnuť jednoduchý nízkoúrovňový jazyk, ktorý je typovaný a rozšíriteľný [23]. V strednej časti dochádza k optimalizovaniu

tohto LLVM IR kódu, pričom cieľom je ho upraviť do čo najvhodnejšej podoby pre zadnú časť spätného prekladača. Toto optimalizovanie je vykonané prostredníctvom vstavaných optimalizácií nachádzajúcich sa vo frameworku LLVM [44]. LLVM framework bol navrhnutý ako inovatívny framework pre prekladače a pre tvorbu nástrojov na nich založených. Obsahuje množinu jazykovo nezávislých inštrukcií, veľké množstvo vstavaných optimalizačných algoritmov a analýz a už spomínanú prechodnú reprezentáciu LLVM IR [23]. Stredná časť je vo veľkej miere závislá na tomto frameworku. Zadná časť spätného prekladača zabezpečuje generovanie vysokoúrovňového kódu. Nachádzajú sa tu aj optimalizácie, ktorých úlohou je zvýšiť čitateľnosť produkovaného kódu. Aktuálne sú podporované dva vysokoúrovňové jazyky a to jazyk C a modifikovaná verzia jazyka Python [44].

Ako už bolo zmienené, v zadnej časti spätného prekladača sa nachádzajú optimalizácie, ktorých úlohou je zvýšiť čitateľnosť produkovaného kódu. Niektoré tieto optimalizácie sú však príliš pomalé na veľkých binárnych súboroch (súbory o veľkosti niekoľkých MB). Hlavnou úlohou tejto diplomovej práce je presun týchto optimalizácií do strednej časti a prípadné rozšírenie ich funkčnosti. Pri návrhu tohto presunu sa vychádzalo z toho, že LLVM framework poskytne lepšiu podporu pre tieto optimalizácie, a tak dôjde k ich zrýchleniu.

V rámci tejto diplomovej práce bolo navrhnutých a implementovaných niekoľko optimalizácií. Ich detailný popis bude prezentovaný v neskorších častiach textu. Ide napríklad o optimalizáciu riešiacu usporiadanie PHI uzlov. Jej úlohou je riešiť problém spojený s paralelným vyhodnocovaním inštrukcie `phi` nachádzajúcej sa v jazyku LLVM IR [45]. Výstup spätného prekladu je však v jazyku C, ktorý je vyhodnocovaný sekvenčne. Preto je potrebné aby táto optimalizácia transformovala kód LLVM IR do funkčne ekvivalentnej sekvenčnej podoby. Ďalšou optimalizáciou je optimalizácia odstraňujúca mŕtve priradenia do globálnych premenných. Pod pojmom mŕtve priradenie do globálnej premennej rozumieme situáciu, pri ktorej dôjde k priradeniu hodnoty do globálnej premennej a táto hodnota nie je nikde v kóde z nej prečítaná. Takéto priradenie môže byť odstránené. Optimalizácia konvertujúca globálne premenné na lokálne patrí medzi ďalšie navrhnuté a implementované optimalizácie v tejto diplomovej práci. Jej úlohou je konvertovať globálne premenné na lokálne všade tam, kde je to možné. To znamená vytvárať nové lokálne premenné vo funkciách a nahrádzať použitia globálnych premenných pomocou týchto lokálnych premenných. Ďalšou optimalizáciou je optimalizácia odstraňujúca nedosiahnuteľné funkcie. Táto optimalizácia odstraňuje funkcie, ktoré nie sú nikdy volané. Poslednou navrhnutou a implementovanou optimalizáciou je optimalizácia odstraňujúca kód za volaniami funkcií, ktoré sa nevracajú. Jej úlohou ako už z názvu vyplýva je zabezpečiť odstránenie kódu za takýmito funkciami. Jednou z takých funkcií je napríklad `exit()` zo štandardnej knižnice jazyka C.

Práca je štruktúrovaná nasledovne. Kapitola 2 obsahuje základné informácie o reverznom inžinierstve a jeho použití v rámci informačných technológii, ako aj informácie o spätnom preklade. Detailný popis spätného prekladača vyvíjaného spoločnosťou AVG Technologies sa nachádza v kapitole 3. V tejto kapitole je možné nájsť aj základné informácie o frameworku LLVM. V kapitole 4 sa nachádza popis prechodnej reprezentácie LLVM IR. Stav spätného prekladača pred zahájením tejto diplomovej práce a uvedenie jej cieľov obsahuje kapitola 5. Návrhu a popisu jednotlivých optimalizácií sa venuje kapitola 6. Kapitola 7 popisuje implementáciu navrhnutých optimalizácií. Popis techník využitých pri testovaní týchto optimalizácií obsahuje kapitola 8. V kapitole 9 sa nachádza zhodnotenie dosiahnutých výsledkov navrhnutých a implementovaných optimalizácií. Záverečná kapitola 10 obsahuje zhodnotenie práce a diskutuje možný budúci vývoj.

Kapitola 2

Reverzné inžinierstvo a spätný preklad

Táto kapitola je založená na [10, 15].

Pod pojmom reverzné inžinierstvo chápeme proces, ktorého úlohou je získať chýbajúce znalosti o skúmanom predmete. Najčastejšie ide o znalosti, ktoré nie sú voľne dostupné. V niektorých prípadoch sú to zničené alebo stratené informácie. Tento pojem sa datuje už od dôb priemyselnej revolúcie a nie je striktne viazaný na modernú dobu. Reverzné inžinierstvo je veľmi často spájané s armádou. Tá ho využíva pri skúmaní technológií nepriateľa a následne sú potom tieto získané informácie využité pre tvorbu nových vlastných technológií. Počas studenej vojny a v obidvoch svetových vojnách existuje veľa príkladov upozorňujúcich práve na použitie reverzného inžinierstva. Moderná doba však prináša novú formu reverzného inžinierstva a to softvérové reverzné inžinierstvo. Pri reverznom inžinierstve je dôležitá aj otázka právnej legálnosti. V niektorých štátoch sú určité aktivity spadajúce pod reverzné inžinierstvo považované za nezákonné [10].

V tejto kapitole je možné nájsť základné informácie o problematike reverzného inžinierstva. Konkrétne podkapitola 2.1 sa venuje softvérovému reverznému inžinierstvu a jeho použitiu v praxi. Ďalej sa v tejto kapitole nachádza popis spätného prekladu, pretože spätný preklad predstavuje jednu z techník, ktoré reverzné inžinierstvo využíva. Spätnému prekladu sa venuje podkapitola 2.2. V tejto podkapitole sa nachádza aj popis výhod využitia spätného prekladača oproti disassembleru.

2.1 Softvérové reverzné inžinierstvo a jeho použitie

Softvérové reverzné inžinierstvo (ďalej len SRE z angl. *software reverse engineering*) a jeho použitie môžeme rozdeliť na dva prípady. V prvom prípade je v praxi používané ako analýza určitého softvérového systému s cieľom získať implementačné detaily, prípadne jeho návrh. Môže byť aplikovaná buď iba na jeho určitú časť alebo na celý systém [7].

V druhom prípade ide o prevod informácie z nízkoúrovňového formátu (napr. aplikačného binárneho kódu) na čitateľnejšiu formu na určitej úrovni abstrakcie. Takýto kód na vyššej úrovni abstrakcie je lepšie čitateľný a zrozumiteľnejší [5].

V dnešnej dobe sa často používajú tieto techniky na detekciu a analýzu malvéru. Rovnako ich je možné využiť pri odhaľovaní bezpečnostných chýb a slabín kreditných kariet. SRE je však často používané tak, že porušuje niektoré zákony. Napríklad ide o odstraňovanie ochrany proti kopírovaniu z hier, CD nosičov a podobne. Zneužívanie SRE k získavaniu cudzích zdrojových kódov a implementačných detailov je tiež časté a nepatrí medzi ojedinelé prípady jeho využitia. Tieto získané informácie sú potom využívane pri tvorbe novej technológie, prípadne dochádza len k šíreniu získaných informácií [35].

Typickými príkladmi nástrojov využívaných pri SRE sú disassembler a spätný prekladač. Disassembler prevádza strojový kód do jazyka symbolických inštrukcií [31]. Cieľom spätného prekladača je preklad binárneho kódu závislého na určitej platforme do zdrojového kódu vysokoúrovňového jazyka. Spätný prekladač je možné chápať oproti dissasembleru ako program na vyššej úrovni. Je to z toho dôvodu, že výstupom spätného prekladača je kód na vyššej úrovni, než kód produkovaný dissasemblerom [9]. Ku skomplikovaniu využitia SRE je možné použiť tzv. obfuskátor. Ten slúži k prevodu zdrojového kódu do zdrojového kódu v tom istom jazyku, avšak vykoná na ňom niekoľko zmien tak, aby bol takýto kód ťažko čitateľný. Obidva kódy sú však funkčne ekvivalentné [33]. Ďalšou možnosťou je využitie tzv. protektorov. Tie na výsledný binárny kód použijú kompresiu alebo ho pomocou šifrovacích techník zašifrujú a vložia kód na jeho automatické rozšifrovanie pri spustení [30].

2.2 Spätný preklad

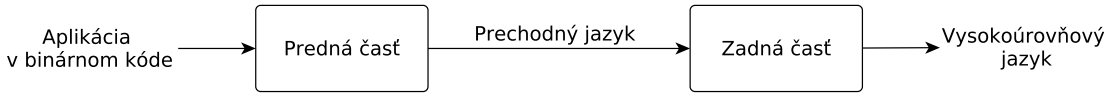
Spätný preklad je reverzný proces bežného prekladu, pričom jeho cieľom je vytvorenie vysokoúrovňového kódu z binárneho kódu. Pre tento vysokoúrovňový kód by však malo platiť, že je funkčne ekvivalentný s kódom v binárnej podobe. Hlavná motivácia spätného prekladu je tá, že tento kód na vyššej úrovni abstrakcie je v porovnaní s tým na nižšej úrovni čitateľnejší [9].

Spätný preklad vykonáva spätný prekladač. Pri spätnom prekladači je však potrebné uviesť, že binárny kód na ktorom je vykonaný spätný preklad je platformovo závislý. Spätný prekladač vykonáva reverzný proces prekladača [9]. Tento reverzný proces nie je ale možné chápať ako vrátenie krokov, ktoré vykonal prekladač. Takéto vrátenie krokov prekladača nie je možné z toho dôvodu, že v binárnom kóde sa nenachádzajú informácie o tom, aké kroky pri preklade vykonal prekladač. S procesom spätného prekladu je však spojený jeden významný problém. Tento problém súvisí s tým, že kód v binárnej podobe obvykle neobsahuje explicitné informácie akými sú napríklad definície dátových štruktúr, názvy premenných, prípadne informácie o iných dátových typoch. Často sa vedú polemiky o tom či je vôbec možné zostrojiť kvalitný spätný prekladač [10]. Vytvorenie zdrojového kódu vysokoúrovňového jazyka, ktorý identicky vykonáva funkcie pôvodného programu, je však pre niektoré jazyky možné [9]. Tento kód sa však vo väčšine prípadoch nepodobá na originálny zdrojový kód a býva veľmi ťažko čitateľný. Vo výslednom kóde chýbajú napríklad komentáre, názvy pôvodných premenných, deklarácie pôvodných dátových štruktúr. Takisto jedným z dôvodov, prečo výsledný kód po spätnom preklade neodpovedá štruktúre pôvodného kódu, sú optimalizácie vykonávané prekladačom [10].

Spätný prekladač ako už bolo uvedené je možné považovať za ďalší vývojový stupeň po dissassembleru. Úlohou spätného prekladača je vytvoriť kód na vyššej úrovni abstrakcie, než kód v jazyku symbolických inštrukcií. Takýto kód je jednoduchšie čitateľný a analyzovateľný. Hlavnou myšlienkou spätného prekladu je získať čo najpodobnejší výstupný kód s kódom,

ktorý bol preložený do binárnej podoby. Spätný prekladač je však možné použiť aj na vytvorenie výstupného kódu v inom jazyku, než v akom bol pôvodný kód napísaný [18].

Jedna z možných štruktúr spätného prekladača je uvedená na obrázku 2.1. Táto štruktúra je veľmi podobná prekladaču. Jednotlivé časti však pracujú v opačnom poradí. Úlohou prednej časti je dekódovať inštrukcie v nízkoúrovňovom jazyku a ich prevod do prechodnej reprezentácie. Prechodná reprezentácia predstavuje vyššiu formu abstrakcie, než pôvodný nízkoúrovňový jazyk. Nezávislosť reprezentácie na nejakej konkrétnej architektúre je možné chápať ako ďalšiu veľkú výhodu. Úlohou zadnej časti je z prechodnej reprezentácie vygenerovať zvolený vysokoúrovňový jazyk [10].



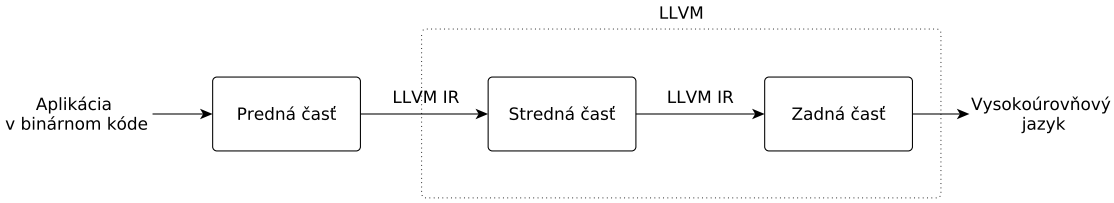
Obr. 2.1: Jedna z možných štruktúr spätného prekladača.

Kapitola 3

Spätný prekladač vyvíjaný spoločnosťou AVG Technologies

Spätný prekladač vyvíjaný spoločnosťou AVG Technologies je platformovo nezávislý na architektúre (MIPS, ARM, Intel x86, atď.) a súborových formátoch (ELF, PE, Mach-O, atď.), generujúci zvolený vysokoúrovňový jazyk. Jeho využitie je plánované pri analýze malvéru napríklad pre mobilné zariadenia, tablety a iné podobné zariadenia. Momentálne sú podporované architektúry Intel x86, ARM, inštrukčné rozšírenie Thumb, MIPS, PIC32 a PowerPC. Medzi podporované súborové formáty patrí ELF a PE. Výstup je možný v jazyku C alebo v modifikovanej verzii jazyka Python [44]. Tento spätný prekladač a jeho funkcionalitu si je možné vyskúšať prostredníctvom verejne prístupnej služby [2].

Spätný prekladač vyvíjaný spoločnosťou AVG Technologies sa skladá z troch častí. Z prednej, strednej a zadnej časti. Ich prepojenie demonštruje obrázok 3.1. Úlohou prednej časti je prevod binárneho kódu platformovo závislej aplikácie do platformovo nezávislého kódu. Kód v jazyku LLVM IR predstavuje takýto nezávislý kód, a preto je výstupom prednej časti. Stredná časť nad kódom získaným z prednej časti vykoná optimalizácie, pričom jej cieľom je získať kód čo najvhodnejší pre zadnú časť spätného prekladača. Tieto optimalizácie sú vykonávané nad kódom v jazyku LLVM IR a výstupom je optimalizovaný kód opäť v tomto jazyku. Úlohou zadnej časti je vyprodukovať vysokoúrovňový kód v čo najčitateľnejšej podobe [44].



Obr. 3.1: Štruktúra spätného prekladača vyvíjaného spoločnosťou AVG Technologies [44].

Táto kapitola je štruktúrovaná nasledovne. Podkapitola 3.1 obsahuje popis frameworku LLVM, na ktorom je založená stredná časť a čiastočne zadná časť. Podkapitola 3.2 sa venuje prednej časti. Strednú časť popisuje podkapitola 3.3. Táto časť spätného prekladača je z hľadiska tejto diplomovej práce najvýznamnejšia, pretože optimalizácie navrhnuté v rámci tejto

diplomovej práce sú určené práve pre túto časť spätného prekladača. Záverečná podkapitola 3.4 sa venuje zadnej časti.

3.1 LLVM

LLVM framework bol navrhnutý ako inovatívny framework pre prekladače a pre tvorbu nástrojov na nich založených. Obsahuje množinu jazykovo nezávislých inštrukcií, veľké množstvo vstavaných optimalizačných algoritmov a analýz a prechodnú reprezentáciu LLVM IR. LLVM IR spĺňa takzvanú *static single assignment* (ďalej len SSA) formu [23]. Pre SSA formu platí, že do každej premennej môže dôjsť iba práve k jednému priradeniu. Súčasne musí byť splnená podmienka, že každá premenná je definovaná pred jej použitím [37]. LLVM IR predstavuje prechodnú reprezentáciu, ktorá poskytuje typovú bezpečnosť, nízkoúrovňové operácie, flexibilitu a schopnosť prevodu z množstva vysokoúrovňových jazykov. Cieľom jeho tvorcov bolo dosiahnuť jednoduchý nízkoúrovňový jazyk, ktorý je typovaný a rozšíriteľný [23]. Detailný popis jazyka LLVM IR sa nachádza v samostatnej kapitole 4. Pre nasledujúcu časť textu však nie je potrebná znalosť uvedenej kapitoly. Tá je potrebná až v neskorších častiach venujúcich sa navrhnutým a implementovaným optimalizáciám.

3.2 Predná časť

Predná časť spätného prekladača je jediná časť, ktorá je platformovo závislá. Jej hlavnou úlohou je prevod binárneho kódu platformovo závislej aplikácie do sekvencie LLVM IR inštrukcií [19].

V prednej časti dochádza k odstraňovaniu staticky linkovaného kódu z binárneho kódu, pretože tento kód zbytočne zvyšuje neprehľadnosť výsledného výstupného kódu. Týka sa to napríklad knižničných funkcií typu `printf`, kde je všeobecne známe, čo tieto funkcie vykonávajú. Takto upravený binárny kód je vstupom pre takzvaný inštrukčný dekodér. Jeho funkcionalita je podobná disassembleru, avšak výstupom nie je kód v jazyku symbolických inštrukcií, ale sekvencia inštrukcií odpovedajúca reprezentácii LLVM IR. Táto sekvencia inštrukcií je odlišná od LLVM IR, ale pripomína ju a predstavuje vnútornú reprezentáciu prednej časti, pričom platí, že je platformovo nezávislá [16].

V prednej časti spätného prekladača sa nachádza niekoľko analýz. Jednou z týchto analýz je analýza toku riadenia programu. Táto analýza je zodpovedná za rozpoznávanie základných blokov a vytvorenie takzvaného grafu toku riadenia. Ďalšou jej úlohou je rozpoznanie všetkých skokových inštrukcií a rozpoznanie cieľových adries týchto skokových inštrukcií. Ďalšia analýza, ktorá sa v tejto prednej časti nachádza je analýza funkcií. Jej základnou úlohou je rozpoznať funkcie. V prednej časti sa ešte nachádza analýza dátového toku. Táto analýza je zodpovedná za rekonštrukciu dátových typov, rekonštrukciu argumentov funkcií ako aj ich návratových hodnôt. Okrem týchto uvedených analýz existujú v prednej časti ďalšie analýzy. Tie však nemá zmysel popisovať, pretože tie vyššie uvedené patria medzi najpodstatnejšie [16].

Poslednou inštanciou prednej časti je generátor LLVM IR kódu. Tento generátor generuje LLVM IR kód z vnútornej reprezentácie tak, aby získaná LLVM IR reprezentácia bola čo najvhodnejšia pre ďalšie časti spätného prekladača [16].

3.3 Stredná časť

Vstupom strednej časti spätného prekladača je kód v jazyku LLVM IR. Rovnako je kód v tomto jazyku aj výstupom pre túto časť. Úlohou strednej časti je optimalizovať kód v jazyku LLVM IR tak, aby bol čo najvhodnejší pre zadnú časť a zvýšil tak schopnosť generovania čitateľnejšieho kódu zadnou častou spätného prekladača.

V strednej časti je využívaný nástroj `opt` [20]. Tento nástroj poskytuje LLVM framework a pomocou neho je možné spúšťať veľké množstvo vstavaných optimalizácií a analýz. Zároveň je možné pomocou neho spúšťať i vlastné implementované optimalizácie. Vďaka tomuto nástroju dochádza vo veľkej miere k optimalizovaniu kódu v jazyku LLVM IR [24].

V strednej časti sa nachádza aj analýza idiómov, ktorá prispieva k čitateľnejšiemu kódu. Táto analýza už bola vytvorená pred zahájením tejto diplomovej práce a zároveň platí, že nie je súčasťou LLVM frameworku. Je však spúšťaná pomocou nástroja `opt`. Analýza je potrebná z dôvodu, že prekladač sa pri preklade snaží niektoré inštrukcie nahradiť za iné inštrukcie, ktoré sú rýchlejšie vykonateľné počítačom. Často však vďaka tomu dôjde k použitiu viacerých inštrukcií miesto jednej. Z toho vyplýva, že takto dochádza k zhoršovaniu čitateľnosti kódu, a preto táto analýza idiómov sa snaží vrátiť zmeny vykonané prekladačom [16].

V strednej časti sa okrem vstavaných optimalizácií a analýzy inštrukčných idiómov nachádzajú navrhnuté a vytvorené optimalizácie v tejto diplomovej práci. Úlohou týchto optimalizácii je rovnako zlepšiť LLVM IR kód pre zadnú časť spätného prekladača, čo následne vedie k čitateľnejšiemu vysokoúrovňovému kódu. Tieto optimalizácie sú spúšťané rovnako ako tie vstavané pomocou nástroja `opt`. Ich detailný popis návrhu obsahuje kapitola 6.

3.4 Zadná časť

Poslednou časťou spätného prekladača je zadná časť. Úlohou tejto časti je produkovanie vysokoúrovňového kódu tak, aby bol tento kód čo najčitateľnejší. Vstupom zadnej časti je LLVM IR kód. Jej výstupom je kód vysokoúrovňového jazyka. V zadnej časti dochádza k prevodu LLVM IR do BIR. BIR je vnútornou reprezentáciou tejto časti spätného prekladača, pričom je vytvorený na základe kódu LLVM IR. Prechodná reprezentácia BIR umožňuje zachytiť všetky prvky do pamäti, pričom nie je potreba žiadnej textovej reprezentácie. Jej dôležitou schopnosťou je, že dokáže modelovať všetky podstatné konštrukcie jazyka LLVM IR z hľadiska spätného prekladača. Okrem toho je schopná zoskupovať viacero príkazov a výrazov do jedného, ktoré sú následne prevedené do vysokoúrovňového jazyka. Ďalšou vlastnosťou BIR je umožnenie štruktúrovania kódu. Tieto postupy nie sú možné v LLVM IR, pretože LLVM IR je nízkoúrovňový jazyk. BIR následne slúži ako vstup pre konkrétny generátor výsledného jazyka [44].

V zadnej časti spätného prekladača sa nachádzajú optimalizácie, ktorých úlohou je zlepšiť čitateľnosť produkovaného kódu. Ide napríklad o optimalizáciu zjednodušujúcu aritmetické výrazy. Táto optimalizácia nahrádza napríklad súčet dvoch čísel za výsledok tohto súčtu. Ďalej je to optimalizácia konvertujúca bitové posuny na násobenie a delenie. Odstraňovanie prebytočných zátvoriek v aritmetických výrazoch je takisto riešené optimalizáciou nachádzajúcou sa v zadnej časti spätného prekladača. Ďalšou optimalizáciou je optimalizácia odstraňujúca mŕtvy kód. Mŕtvy kód predstavuje taký kód, ktorý sa nikdy nevykoná.

Okrem uvedených optimalizácií sa nachádza v zadnej časti ešte veľké množstvo ďalších optimalizácií [4].

Výstupom zadnej časti je vysokoúrovňový kód. Aktuálne podporované jazyky sú jazyk C a modifikovaná verzia jazyka Python. Tento Python je rozšírený napríklad o ukazovatele a referenčné operátory z jazyka C. Keďže jazyk Python neobsahuje dátový typ pole ani štruktúra, tak sú tieto konštrukcie nahradené inými. Pole je nahradené dátovým typom list a štruktúra pomocou slovníku [44].

Kapitola 4

LLVM IR

Táto kapitola sa venuje popisom niektorých konštrukcií jazyka LLVM IR. Napríklad ide o funkcie, moduly, premenné a pod. Táto časť však nenahrádza dokumentáciu syntaxe týchto konštrukcií. Tá je dostupná na stránkach LLVM frameworku [23]. Miesto toho sa bude nasledujúca časť textu venovať jednoduchým príkladom, ktoré demonštrujú tieto konštrukcie. Rovnako bude obsahovať základné informácie, ktoré sú potrebné v neskorších kapitolách pre pochopenie danej problematiky a ukážok kódov v jazyku LLVM IR.

Programy v jazyku LLVM IR sú tvorené takzvanými modulmi. Moduly obsahujú funkcie, globálne premenné, definície dátových typov a pod [23]. V prípade spätného prekladača vyvíjaného spoločnosťou AVG Technologies je celý program v jednom module [16].

LLVM poskytuje veľké množstvo dátových typov. Tabuľka 4.1 zhrňuje tie najdôležitejšie typy, ktoré budú využívané v príkladoch v neskorších kapitolách.

Tabuľka 4.1: Príklady dátových typov v LLVM IR [23].

Typ	Popis
`i32`	32 bitová celočíselná hodnota
`i64`	64 bitová celočíselná hodnota
`iX`	X bitová celočíselná hodnota
`float`	32 bitová hodnota s pohyblivou rádovou čiarkou
`double`	64 bitová hodnota s pohyblivou rádovou čiarkou
`i32 *`	Ukazovateľ na 32 bitovú celočíselnú hodnotu
`[40 x i32]`	Pole obsahujúce štyridsať 32 bitových celočíselných hodnôt
`{ i32, i32, i32 }`	Štruktúra zložená z troch 32 bitových celočíselných hodnôt

V LLVM IR kóde sa nachádzajú aj funkcie [23]. Napríklad funkcia s menom `func`, návratovým typom `i32` a jedným parametrom `a` typu `i32` vyzerá v kóde LLVM IR ako na príklade 4.1. Na príklade je možné vidieť aj volanie tejto funkcie. Toto volanie je realizované prostredníctvom inštrukcie `call`.

Funkcia je zložená z takzvaných základných blokov. Bloky obsahujú inštrukcie, pričom každý blok musí byť ukončený ukončovacou inštrukciou. Ukončovacie inštrukcie spôsobia pri vyhodnocovaní bloku prechod do iného bloku. Ide napríklad o inštrukcie `br` (skok na iný blok), `ret` (návrat z funkcie) a pod. Každá funkcia ma takzvaný vstupný blok. Tento blok

```
define i32 @func(i32 %a) {
...
  ...
  call i32 @func(i32 3)
  ...
}
```

Obr. 4.1: Príklad zápisu funkcie a jej volania v jazyku LLVM IR.

je len jeden. Ide o prvý vyhodnocovaný blok vo funkcii, v ktorej sa nachádza [23]. Funkcia obsahujúca jeden vstupný blok s názvom `entry` a blokom `bb`, obsahujúcim inštrukciu `ret`, ktorá spôsobí ukončenie vyhodnotenia funkcie a návrat hodnoty 0 je znázornená na ukážke 4.2. Ako je možné vidieť, tak v prípade inštrukcie skoku je adresa skoku uvedená názvom bloku, na ktorý je tento skok vykonaný, pričom pred týmto názvom bloku je uvedený znak "`%`".

```
define i32 @func() {
entry:
  br label %bb

bb:
  ret i32 0
}
```

Obr. 4.2: Príklad obsahujúci základne bloky vo funkcii.

Premenné v jazyku LLVM IR je možné rozdeliť na lokálne a globálne. Rozsah platnosti týchto premenných odpovedá typickým konvenciám ako v iných jazykoch, napríklad jazyku C. Názvy globálnych premenných začínajú znakom "`@`". Napríklad premenná `@glob` predstavuje globálnu premennú s názvom `glob`. Lokálne premenné sa líšia od globálnych tak, že ich pomenovanie začína znakom "`%`". Lokálna premenná s názvom `loc` by bola v jazyku LLVM IR zapísaná ako `%loc` [23].

Uveďme si teraz pre ilustráciu príklad zložitejšieho kódu LLVM IR. Jeden z možných kódov zápisu výpočtu faktoriálu v jazyku LLVM IR je možné vidieť na príklade 4.3. Na riadku 3 dochádza k porovnaniu obsahu hodnoty predanej do funkcie s hodnotou 0. Na základe výsledku tohto porovnania dôjde na riadku 4 k skoku na blok `recurse.end` v prípade, že predávaná hodnota do funkcie je rovná 0. V opačnom prípade dôjde k skoku na blok `recurse`. V bloku `recurse` na riadkoch 7 a 8 je použitá inštrukcia `phi`. Jej detailný popis sa nachádza pri popise optimalizácie, ktorá usporiadava uzly PHI, konkrétne v oddiele 6.1.1. Pre tento príklad však stačí uviesť, že inštrukcia `phi` vyberie hodnotu z množiny určených hodnôt na základe predchádzajúceho bloku, z ktorého došlo k skoku na tento blok. Napríklad pre skok z bloku `entry` to bude hodnota nachádzajúca sa v parametre `X`. Táto inštrukcia je potrebná z dôvodu zachovania SSA formy. Riadok 9 obsahuje inštrukciu pre sčítanie, riadok 10 inštrukciu pre násobenie. Na riadku 11 dochádza opäť k porovnaniu s nulou a na základe toho na riadku 12 dochádza k pokračovaniu výpočtu a skoku na blok `recurse`, alebo k ukončeniu a skoku na blok `recurse.end`. Riadok 16 vykonáva vrátenie výsledku z funkcie. Na uvedenom príklade je možné vidieť, že kód jazyka LLVM IR spĺňa SSA formu. Do každej premennej je hodnota priradená v kóde iba raz a súčasne platí, že vždy došlo k priradeniu hodnoty do premennej pred načítaním hodnoty z tejto premennej.

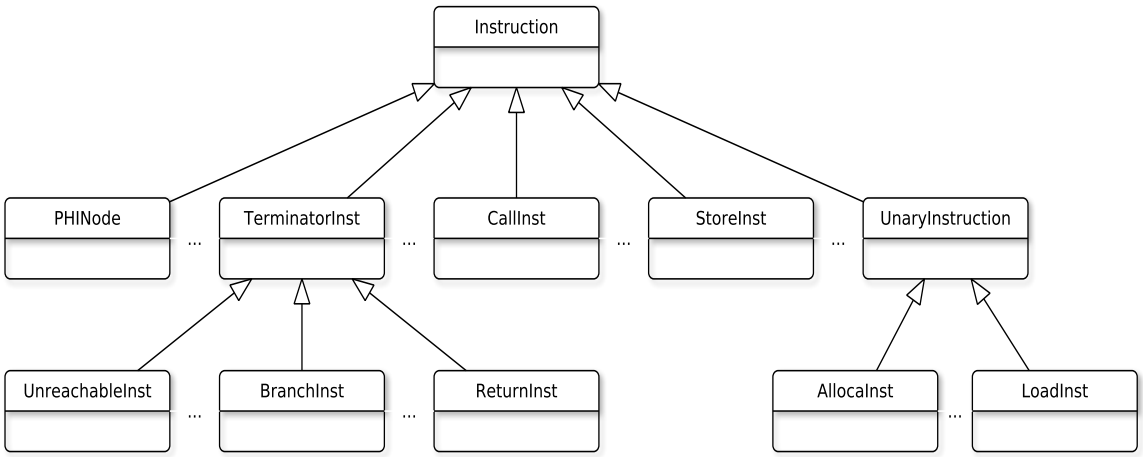
```
1  define i32 @factorial(i32 %X) {
2  entry:
3    %1 = icmp eq i32 %X, 0
4    br i1 %1, label %recurse.end, label %recurse
5
6  recurse:
7    %X2 = phi i32 [%2, %recurse], [%X, %entry]
8    %acc = phi i32 [%3, %recurse], [1, %entry]
9    %2 = add i32 %X2, -1
10   %3 = mul i32 %X2, %acc
11   %4 = icmp eq i32 %2, 0
12   br i1 %4, label %recurse.end, label %recurse
13
14 recurse.end:
15   %res = phi i32 [1, %entry], [%3, %recurse]
16   ret i32 %res
17 }
```

Obr. 4.3: Príklad výpočtu faktoriálu zapísaného v jazyku LLVM IR.

LLVM IR je možné vytvárať programovo. LLVM poskytuje rozhranie v jazyku C++ pre vytváranie LLVM IR kódu. Pre prístup k rozhraniu v jazyku C++ je možné použiť aj iné jazyky, ako napríklad jazyk Python [21].

LLVM obsahuje triedy slúžiace ako abstrakcia rôznych prvkov prechodnej reprezentácie LLVM IR a vďaka ním je možné vytvárať jednotlivé prvky tohto jazyka. Trieda `Module` reprezentuje modul. Funkciu reprezentuje trieda `Function`. Základné bloky predstavuje trieda `BasicBlock`.

Inštrukcie používané v LLVM IR sú reprezentované tiež triedami. Na UML diagrame 4.4 sú znázornené hlavne triedy, ktoré predstavujú využívané inštrukcie pri navrhnutých optimalizáciách. Bázovou triedou pre všetky inštrukcie je trieda `Instruction`. Bázovou triedou pre všetky inštrukcie ukončujúce základný blok je trieda `TerminatorInst`. Trieda `CallInst`



Obr. 4.4: UML diagram tried znázorňujúci vzťahy medzi triedami reprezentujúcich inštrukcie.

predstavuje konkrétnu triedu pre inštrukciu volania funkcie. Trieda `BranchInst` reprezentuje inštrukciu skoku a trieda `ReturnInst` inštrukciu ukončenia toku programu pre danú funkciu. Rovnako konkrétne triedy reprezentujúce inštrukcie sú triedy `PHINode`, `StoreInst`, `LoadInst`, `AllocaInst` a `UnreachableInst`. Bázovou triedou pre unárne inštrukcie je trieda `UnaryInstruction` [27]. Syntax a sémantika inštrukcií využívaných v konkrétnych optimalizáciách bude vysvetlená v kapitolách prislúchajúcim týmto optimalizáciám.

Kapitola 5

Stav spätného prekladača pred zahájením tejto práce

Táto kapitola sa venuje stavu spätného prekladača pred zahájením tejto diplomovej práce. Venuje sa však iba častiam, ktoré súvisia s cieľmi tejto práce.

V podkapitole 5.1 je uvedený stav strednej časti pred začatím tejto diplomovej práce. Podkapitola 5.2 popisuje problémy spojené s optimalizáciami v zadnej časti. Vďaka týmto problémom bolo potrebné navrhnúť nové optimalizácie v strednej časti. Zhrnutie cieľov tejto diplomovej práce sa nachádza v podkapitole 5.3.

5.1 Stredná časť

Stredná časť spätného prekladača je založená na frameworku LLVM [16]. Tento framework obsahuje veľké množstvo vstavaných optimalizácií. Vzhľadom na to, že LLVM bolo pôvodne navrhnuté ako inovatívny framework pre prekladače, tak nie všetky vstavané optimalizácie sú pre spätný preklad prínosné. Úlohou prekladača je previesť vysokoúrovňový kód do binárnej aplikácie. Spätný prekladač vykonáva opačný proces. Je logické, že nie všetky vstavané optimalizácie sú prínosné pre spätný preklad. LLVM framework však obsahuje aj optimalizácie, ktoré môžu byť prínosné pre spätný preklad. Napríklad ide o optimalizáciu odstraňujúcu mŕtve častí kódu. Pod mŕtvym kódom je možné rozumieť časť kódu nemajúcu vplyv na beh programu. Optimalizácia mŕtvych cyklov takisto patrí medzi optimalizácie, ktoré vedú k zväčšeniu čitateľnosti kódu. Úlohou tejto optimalizácie je odstrániť také cykly z kódu, ktoré neovplyvňujú beh programu. Jednoduchá propagácia konštánt patrí medzi ďalšiu prínosnú optimalizáciu. V prípade, že existuje inštrukcia sčítajúca dve konštantné hodnoty, tak táto optimalizácia nahradí túto inštrukciu za výsledok zo sčítania. Táto optimalizácia samozrejme optimalizuje aj iné inštrukcie okrem tých na sčítanie, ale je zachovaný popísaný princíp pri tomto optimalizovaní [25].

Pred začatím tejto diplomovej práce boli niektoré vstavané optimalizácie zapnuté a spúšťané prostredníctvom nástroja `opt`. Súčasne sa v tejto časti spätného prekladača nachádzala aj analýza idiómov, popísaná v podkapitole 3.3.

5.2 Zadná časť

Ako už bolo spomínané, tak zadná časť obsahuje optimalizácie, ktorých úlohou je zlepšiť čitateľnosť produkovaného kódu [44]. Niektoré tieto optimalizácie sú však príliš pomalé na veľkých binárnych súboroch (súbory o veľkosti niekoľkých MB). Konkrétne ide o optimalizáciu odstraňujúcu mŕtve priradenia a optimalizáciu konvertujúcu globálne premenné na lokálne. Okrem týchto dvoch uvedených optimalizácií boli presunuté do strednej časti ďalšie dve optimalizácie. Ide o optimalizáciu odstraňujúcu nedosiahnuteľné funkcie a optimalizáciu odstraňujúcu kód za funkciami, ktoré sa nevracajú. Tieto dve optimalizácie netrpia problémami spojenými s ich pomalosťou nad veľkými súbormi. Vďaka ním dochádza k odstráneniu kódu, ktorý neovplyvňuje beh programu. Takýmto odstránením dôjde k zmenšeniu množstva kódu potrebného spracovať inými optimalizáciami a časťami spätného prekladača, a tak dôjde k urýchleniu spätného prekladu.

5.3 Ciele diplomovej práce

Jednou z úloh tejto diplomovej práce bolo preštudovať všetky dostupné vstavané optimalizácie z frameworku LLVM a posúdiť ich prínos pre spätný preklad. Pred začatím tejto analýzy už bolo veľké množstvo vstavaných optimalizácií zapnutých, preto bolo nutné aj potvrdiť ich dôvod zapnutia, prípadne navrhnúť ich vypnutie. Okrem posúdenia prínosu týchto optimalizácií bolo pri tejto analýze potrebné porozumieť aj princípom ich tvorenia a následné získané poznatky využiť pri tvorbe nových optimalizácií.

Hlavným cieľom tejto diplomovej práce bol presun niektorých optimalizácií zo zadnej časti do strednej časti. Pri návrhu tohto presunu do strednej časti sa vychádzalo z toho, že LLVM poskytne lepšie prostriedky pre realizáciu týchto optimalizácií. Zároveň presunutím optimalizácií odstraňujúcich kód neovplyvňujúci beh programu dôjde k redukcii kódu, ktorý je potrebný spracovať ostatnými časťami spätného prekladača. Okrem toho bola navrhnutá aj jedna optimalizácia, ktorá rieši problémy spojené s ekvivalentnosťou funkcionality kódu LLVM IR s vysokoúrovňovým kódom.

Návrhu týchto optimalizácií sa venuje kapitola 6. Popis ich implementácie je možné nájsť v kapitole 7. Každú navrhnutú a implementovanú optimalizáciu bolo potrebné otestovať. V kapitole 8 sa nachádza popis tohto testovania ako aj vyvinutie nového testovacieho frameworku v strednej časti spätného prekladača. Zhodnotenie dosiahnutých výsledkov sa nachádza v kapitole 9.

Kapitola 6

Návrh optimalizácií

V tejto kapitole sa nachádza podrobný popis návrhu jednotlivých optimalizácií. Pri návrhoch je kladený dôraz na funkčnú ekvivalentnosť výsledného kódu pred a po aplikovaní optimalizácií. Každá podkapitola sa venuje jednej optimalizácii, pričom obsahuje jej popis, problémy ktoré boli riešené, ako aj podmienky pri ktorých dochádza k optimalizovaniu kódu.

Implementačné riešenia optimalizácií sú popísané v nasledujúcej kapitole 7. Spôsobom testovania navrhnutých optimalizácií sa venuje kapitola 8. V kapitole 9 sa nachádza zhodnotenie dosiahnutých výsledkov navrhnutých optimalizácií.

6.1 Usporiadanie PHI uzlov

S touto optimalizáciou úzko súvisia takzvané uzly PHI. PHI uzly sa využívajú pri SSA pre spôsob umožnenia výberu hodnoty [11]. Vo vysokoúrovňovom jazyku, ako je napríklad jazyk C, je možné použiť pre výber hodnoty príkaz `if`. Na ukážke 6.1 je vidieť, že do premennej `b` je priradená hodnota jedna alebo dva. Na základe vyhodnotenia podmienky príkazu `if` sa rozhodne, ktorá hodnota to bude. Táto uvedená ukážka však nespĺňa podmienky SSA, pretože dochádza k dvojitému priradeniu hodnoty do premennej `a`.

```
a = 1;
if (v < 10)
    a = 2;
b = a;
```

Obr. 6.1: Ukážka demonštrujúca kód, ktorý nespĺňa podmienky SSA.

Uvedený príklad 6.1 je možné upraviť do varianty kódu spĺňajúceho podmienky SSA. Úprava zahŕňa vloženie funkcie PHI, ktorá reprezentuje uzol PHI. Výsledok tejto úpravy sa nachádza na ukážke 6.2. Na nej je vidieť, že do premennej `b` sa priradí hodnota na základe výsledku predchádzajúceho porovnania podmienky `if`, pričom dôjde k výberu buď premennej `a1` alebo `a2` na základe bloku kódu, z ktorého dôjde k volaniu funkcie PHI. To znamená, že ak sa podmienka vyhodnotí ako pravdivá, tak funkcia PHI vráti hodnotu 2. V opačnom prípade vráti hodnotu 1.

```
a1 = 1;
if (v < 10)
    a2 = 2;
b = PHI(a1, a2);
```

Obr. 6.2: Ukážka demonštrujúca kód, ktorý spĺňa podmienky SSA.

V LLVM sa používa pre implementáciu PHI uzlov inštrukcia `phi`. S touto inštrukciou však súvisí jeden významný problém a to, že inštrukcie `phi` sú vyhodnocované paralelne v rámci jedného základného bloku [45]. Príklad demonštrujúci paralelné vyhodnocovanie inštrukcií `phi` sa nachádza až v oddiele 6.1.1, pretože je vhodný až po vysvetlení syntaxe a sémantiky tejto inštrukcie. Výstupom spätného prekladača je napríklad jazyk C, ktorý sa však vyhodnocuje sekvenčne. Preto je potrebné upraviť kód v jazyku LLVM IR do podoby takej, že tento kód bude odpovedať ekvivalentnému sekvenčnému kódu. Úlohou tejto optimalizácie je vykonať úpravu kódu tak, aby bola zabezpečená ekvivalentnosť paralelného a sekvenčného vyhodnotenia.

Túto optimalizáciu nie je úplne možné považovať v pravom zmysle za optimalizáciu, pretože jej primárnym cieľom nie je vytvoriť čitateľnejší kód. Jej hlavnou úlohou ako už bolo zmienené je vyriešiť paralelné vyhodnocovanie inštrukcií `phi`. Na druhú stranu však vďaka riešeniu dôjde k úprave kódu tak, že nový upravený kód bude možné spracovať sekvenčne, čo má na čitateľnosť určite pozitívny prínos. Táto optimalizácia rovnako ako iné navrhnuté optimalizácie využíva výhody poskytované nástrojom `opt`. Vzhľadom na vyššie uvedené je možné implementáciu riešiacu problémy spojené s paralelným vyhodnotením inštrukcií `phi` považovať za optimalizáciu.

Sémantike, syntaxi a možnému použitiu inštrukcie `phi` sa venuje oddiel 6.1.1. Popis problémov, ktoré rieši táto optimalizácia a spôsob ich riešenia sa nachádza v oddiele 6.1.2. Oddiel 6.1.3 sa venuje navrhnutému algoritmu, ktorý rieši uvedené problémy v oddiele 6.1.2 po algoritmickej stránke.

6.1.1 Syntax, sémantika a použitie inštrukcie phi

Syntax inštrukcie `phi` sa nachádza na ukážke 6.3. Inštrukcia `phi` je určená typom prichádzajúcej hodnoty (`<ty>`), prichádzajúcou hodnotou (`<val0>`) a predchodcom bloku (`<label0>`), v ktorom sa konkrétna inštrukcia `phi` nachádza. Počet prichádzajúcich hodnôt v danej inštrukcii `phi` je neobmedzený [23, viď sekcia phi Instruction].

```
<result> = phi <ty> [ <val0>, <label0> ], ...
```

Obr. 6.3: Syntax inštrukcie `phi` [23, viď sekcia phi Instruction].

Okrem dodržania správneho syntaktického zápisu inštrukcie `phi` je nutné pre jej korektnosť dodržať aj nasledujúce podmienky:

1. Inštrukcia `phi` musí obsahovať dvojice `<val0>`, `<label0>` pre všetky predchádzajúce bloky, ktoré predchádzajú tomu, v ktorom sa táto inštrukcia nachádza [25, viď sekcia -verify: Module Verifier].

2. Inštrukcia `phi` musí obsahovať aspoň jednu dvojicu `<val0>`, `<label0>` [25, viď sekcia -verify: Module Verifier].

3. V prípade, že `<val0>` predstavuje nekonštantnú hodnotu, čiže ide napríklad o premennú, tak musí byť táto premenná už v predchádzajúcom kóde LLVM definovaná. Tento fakt vyplýva z definície SSA formy [37].

4. Inštrukcie `phi` sa musia nachádzať len na začiatku bloku. Platí, že pokiaľ sa vyskytne iná inštrukcia ako inštrukcia `phi`, tak za touto inštrukciou už nesmú nasledovať žiadne `phi` inštrukcie [25, viď sekcia -verify: Module Verifier].

5. Vstupný blok funkcie nesmie obsahovať žiadne `phi` inštrukcie, pretože tento blok nemá predchodcov [23, viď sekcia Functions].

Sémantika inštrukcie `phi` je nasledovná. Hodnota priradená do `<result>` je vybraná podľa toho, z akého predchádzajúceho bloku sa dostalo vykonávanie inštrukcií do bloku, v ktorom je daná inštrukcia `phi` definovaná [23, viď sekcia phi Instruction]. V prípade, že by inštrukcia `phi` bola zadefinovaná ako na ukážke 6.4, tak ak by bol vykonávaný predchádzajúci blok aktuálneho bloku blok `first`, tak do premennej `A` sa priradí hodnota 1. Pokiaľ by bol vyhodnocovaným predchodcom blok `second`, tak sa priradí hodnota 2.

```
%A = phi i32 [ 1, %first ], [ 2, %second ]
```

Obr. 6.4: Ukážka výberu hodnoty na základe predchádzajúceho bloku.

Dôležitý fakt o `phi` inštrukciách je ten, že všetky tieto inštrukcie v tom istom bloku sú spracované paralelne [45]. Pre lepšie pochopenie paralelnosti inštrukcií `phi` je uvedený príklad 6.5. Stanovme si napríklad, že premenná `A` obsahuje pred vykonaním uvedených inštrukcií v príklade hodnotu 3. Tým, že sa vykonajú všetky priradenia paralelne, tak premenné nadobudnú nasledujúce hodnoty: `A = 1, B = 2, C = 3`. Paralelnosť v tomto príklade sa prejaví u premennej `C`, ktorá je nastavená na predchádzajúcu hodnotu `A` a nie na hodnotu 1, ktorá bola do nej priradená na začiatku príkladu. Samozrejme všetky ostatné priradenia v uvedenom príklade sú rovnako paralelne vyhodnotené, avšak na nich sa tento jav neprejaví.

```
%A = phi i32 [ 1, %pred ]
%B = phi i32 [ 2, %pred ]
%C = phi i32 [ %A, %pred ]
```

Obr. 6.5: Príklad paralelného priradenia hodnôt pomocou inštrukcií `phi`.

Ako už bolo spomínané v úvode tejto podkapitoly, tak inštrukcia `phi` je využiteľná napríklad pri modelovaní podmieneného skoku v jazyku C, ktorý je reprezentovaný príkazom `if`. Kód na ukážke 6.2 by v jazyku LLVM IR vyzeral ako na ukážke 6.6. Tento uvedený LLVM IR kód je však tiež ešte zjednodušený tak, aby bolo vidieť v ňom podobnosť s kódom na ukážke 6.2. Zjednodušenie pre čitateľnosť je v tomto prípade realizované tým, že kód neobsahuje alokačné časti premenných, obsahuje ručne upravené názvy blokov a doplnené komentáre.

Inštrukciu `phi` je však možné použiť aj pri cykle `for`. Príklad 6.7 v jazyku LLVM IR to názorne demonštruje. Tento príklad je však pre názornosť upravený na tú najpodstatnejšiu

```
pred:
  store i32 1, i32* %1
  %a = load i32* %1                                  ; a1 = 1
  %2 = icmp slt i32 %v, 10                           ; v < 10
  br i1 %2, label %true, label %afterIf              ; if (v < 10)

true:
  store i32 2, i32* %3
  %a.true = load i32* %3                             ; a2 = 2
  br label %afterIf

afterIf:
  %b = phi i32 [ %a, %pred ], [ %a.true, %true ] ; b = PHI(a1, a2)
  ret i32 %b                                         ; return b
}
```

Obr. 6.6: Kód z ukážky 6.2 v jazyku LLVM IR.

časť. V jazyku C by tento zjednodušený príklad predstavoval cyklus `for` s ôsmymi iteráciami. Ako je vidieť, tak inštrukcia `phi` je použitá pre rozhodnutie o tom, či do premennej `i` je priradená inkrementovaná hodnota `i` alebo ide o prvý priechod, pri ktorom dôjde k nastaveniu premennej `i` na hodnotu 0.

```
for:
  %i = phi i64 [ %i.next, %for ], [ 0, %init ]
  %i.next = add i64 %i, 1
  %exitcond = icmp eq i64 %i.next, 8
  br i1 %exitcond, label %exit, label %for
```

Obr. 6.7: Ukážka cyklu `for` v jazyku LLVM IR.

6.1.2 Problémy a ich riešenie

Majme nasledujúci príklad 6.8 v jazyku LLVM IR. Na ňom je možné vidieť, že pri prvom priechode blokom `phiBB` dôjde k paralelnému priradeniu hodnôt nasledovne: `A = 1, B = 2`. Následne riadenie programu prejde do bloku `cond`, kde dôjde k porovnaniu na rovnosť premennej `A` s hodnotou 2. Nakoľko premenná `A` neobsahuje hodnotu 2, nastane skok opäť na blok `phiBB`. V druhom priechode opäť dôjde k paralelnému priradeniu do premenných, ale tento krát budú priraďované uložené hodnôt v premenných `A` a `B`. Po vyhodnotení inštrukcií `phi` budú obsahovať premenné nasledujúce hodnoty: `A = 2, B = 1`. Podmienka na rovnosť v bloku `cond` bude tento kráť pravdivá, a tak dôjde k skoku na blok `exit` a ukončeniu funkcie.

Ukážka 6.9 zobrazuje výsledok po spätnom preklade nad príkladom 6.8. Pri tomto spätnom preklade boli vypnuté všetky optimalizácie, aby došlo k čo najmenšiemu ovplyvneniu výsledku spätného prekladu. V uvedenej ukážke sú ručne doplnené komentáre, aby bolo možné lepšie vidieť súvislosť s príkladom 6.8. Na začiatku ukážky dochádza k priradeniu hodnôt 1 a 2 pri definícií premenných `v1` a `v2`. Tieto premenné predstavujú tie z kódu LLVM IR a v komentároch je uvedené, o ktoré ide. Priechod toku programu v jazyku C a nastavovanie príslušných hodnôt do premenných zatiaľ odpovedá kódu v jazyku LLVM IR. V tele cyklu

```
define void @func() {
entry:
  br label %phiBB

phiBB:
  %A = phi i32 [ %B, %cond ], [ 1, %entry ]
  %B = phi i32 [ %A, %cond ], [ 2, %entry ]
  br label %cond

cond:
  %0 = icmp eq i32 %A, 2                ; Je A rovné dvom?
  br i1 %0, label %exit, label %phiBB ; Skok na základe porovnania.

exit:
  ret void
}
```

Obr. 6.8: Príklad kódu v jazyku LLVM IR, ktorý je pomocou spätného prekladu preložený a zobrazený v ukážke 6.9.

`while` však dochádza k rozdielu. Po vykonaní priradení v tele cyklu budú mať premenné nasledujúce hodnoty: `v1 = 2, v2 = 2`. Je teda vidieť, že obsah premenných je nastavený na iné hodnoty v prípade jazyka LLVM IR a jazyka C. Pokiaľ by bol v uvedených príkladoch výpis týchto premenných, tak by došlo k viditeľnej funkčnej neekvivalentnosti týchto dvoch kódov. Úlohou optimalizácie je vyriešiť tento problém a zabezpečiť ekvivalentné vyhodnocovanie kódu LLVM IR a vysokoúrovňových kódov.

```
void func(void) {
    int32_t v1 = 1;      // A = 1
    int32_t v2 = 2;      // B = 2
    while (true) {
        if (v1 == 2) {   // A == 2
            break;
        }
        v1 = v2;         // A = B
        v2 = v1;         // B = A
    }
    return;
}
```

Obr. 6.9: Ukážka kódu v jazyku C získaného spätným prekladom z príkladu 6.8.

Aby bolo možné zabezpečiť ekvivalentnosť paralelného a sekvenčného vyhodnotenia, tak museli byť vyriešené dva problémy. Prvý problém najlepšie demonštruje nasledujúca ukážka 6.10.

```
%A = phi i32 [ %D, %pred ]
%B = phi i32 [ %A, %pred ]
```

Obr. 6.10: Ukážka prvého problému pri paralelnom vyhodnocovaní inštrukcií `phi`.

Vezmime si napríklad hodnoty tak, že `A = 1, D = 2`. V prípade paralelného vyhodnotenia

inštrukcií `phi` obsahujú premenné v ukážke 6.10 nasledujúce hodnoty: `A = 2, B = 1`. Pokiaľ je vyhodnotenie chápané sekvenčne, tak hodnoty uložené v premenných budú iné a to konkrétne: `A = 2, B = 2`. Je vidieť, že dochádza k rozdielu výsledkov medzi paralelným a sekvenčným vyhodnotením. Tento problém je možné vyriešiť správnym usporiadaním inštrukcií `phi`. Pre príklad 6.10 by išlo o riešenie na ukážke 6.11. Pri paralelnom i sekvenčnom vyhodnocovaní dôjde tento krát k dosiahnutiu ekvivalentných výsledkov. Je to zabezpečené tým, že inštrukcie boli zoradené podľa závislostí. Inštrukcia, v ktorej sa nachádza priradzovaná premenná, ktorá je v inej `phi` inštrukcii modifikovaná, sa musí nachádzať pred tou, ktorá túto hodnotu modifikuje. Tento spôsob riešenia už bol navrhnutý a využíval sa pri konverzii LLVM IR do BIR v zadnej časti spätného prekladača.

```
%B = phi i32 [ %A, %pred ]
%A = phi i32 [ %D, %pred ]
```

Obr. 6.11: Ukážka riešenia prvého problému pri paralelnom vyhodnocovaní inštrukcií `phi`.

Toto riešenie však ešte nie je dostačujúce a tým narážame na druhý problém, ktorý je charakteristický cyklickou závislosťou premenných. Ten je zobrazený na príklade 6.12. V tomto prípade už nie je možné vytvoriť žiadne také usporiadanie, ktoré by zabezpečilo ekvivalentnosť sekvenčného vyhodnotenia s paralelným.

```
...
pred
  ...
  br label %bb

bb:
  %A = phi i32 [ %B, %pred ]
  %B = phi i32 [ %C, %pred ]
  %C = phi i32 [ %A, %pred ]
  ...
```

Obr. 6.12: Ukážka druhého problému pri paralelnom vyhodnocovaní inštrukcií `phi`.

Preto bolo navrhnuté nasledujúce riešenie pre druhý problém. Toto riešenie je možné vidieť na príklade 6.13. Vytvorením nového bloku `bb.pre.phi2seq` obsahujúceho ďalšiu inštrukciu `phi`, úpravou inštrukcií skoku a správnou modifikáciou `phi` inštrukcií v bloku s cyklickou závislosťou, dôjde k zabezpečeniu ekvivalentnosti vyhodnotenia sekvenčného riešenia s paralelným. Táto ekvivalentnosť je zaručená vytvorením novej pomocnej premennej `B.phi2seq.tmp`, ktorá je umiestnená v bloku `bb.pre.phi2seq`, ktorý vznikol ako predchodca toho, v ktorom sa cyklická závislosť nachádza. Vďaka tomuto vytvoreniu dôjde k zrušeniu cyklickej závislosti. Pri tomto riešení je využitý fakt, že pokiaľ rozdelíme `phi` inštrukcie do viacerých blokov, tak rozdelíme množinu inštrukcií vykonávaných paralelne na podmnožiny určené príslušnými blokmi v ktorých sa nachádzajú. Toto riešenie je však navrhnuté tak, aby vznikol čo najmenší počet potrebných blokov a súčasne došlo k vzniku čo najmenšieho počtu pomocných premenných. Pre overenie funkčnosti tohto riešenia pre ukážku 6.12 stanovme, že premenné obsahujú hodnoty pred vykonaním inštrukcií `phi` nasledovne: `A = 1, B = 2, C = 3`. Po vykonaní inštrukcií `phi` budú premenné obsahovať nasledujúce hodnoty: `A = 2, B = 3, C = 1`. Tieto hodnoty budú však nadobudnuté len v prípade paralelného vyhodnotenia. V prípade sekvenčného by boli odlišné. V ukážke 6.13,

ktorá je riešením problému cyklickej závislosti sú počiatočne hodnoty pred vykonaním inštrukcií `phi` rovnaké, ako v ukážke 6.12. Po vykonaní inštrukcií `phi` v príklade 6.13 budú premenné obsahovať nasledujú hodnoty: `B.phi2seq.tmp = 2, A = 2, B = 3, C = 1`. Tento výsledok bude platiť v prípade sekvenčného aj paralelného spracovania. Hodnoty premenných `A`, `B`, `C` z ukážky 6.13 sú ekvivalentné s tými v ukážke 6.12. Táto ekvivalentnosť platí i v prípade sekvenčného spracovania, takže toto riešenie splňuje požadovaný cieľ.

```
...
pred
  ...
  br label %bb.pre.phi2seq

bb.pre.phi2seq
  %B.phi2seq.tmp = phi i32 [ %B, %pred ]
  br label %bb

bb:
  %B = phi i32 [ %C, %bb.pre.phi2seq ]
  %C = phi i32 [ %A, %bb.pre.phi2seq ]
  %A = phi i32 [ %B.phi2seq.tmp, %bb.pre.phi2seq ]
  ...
...
```

Obr. 6.13: Ukážka riešenia druhého problému pri paralelnom vyhodnocovaní inštrukcií `phi`.

Po eliminácii cyklických závislostí je však ešte potrebné vykonať úpravy zmienené pri riešení prvého problému, čím dôjde k vhodnému usporiadaniu jednotlivých inštrukcií `phi`. Preto je dôležité v prvom rade rozdeliť cyklické závislosti a následne aplikovať usporiadanie popísané pri prvom probléme.

6.1.3 Návrh algoritmu

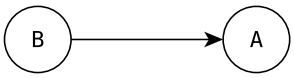
Pri návrhu bol kladený dôraz na efektívnosť algoritmu pre túto optimalizáciu. Z tohto dôvodu bola zvolená reprezentácia závislosti premenných v orientovanom grafe a následne vykonanie topologického radenia nad týmto grafom. Pod pojmom topologického radenia rozumieme také radenie, ktorého účelom je postupne vyberať uzly z grafu, ktoré nemajú žiadneho predchodcu a po každej iterácii pokračovať nad uzlami, ktoré ešte neboli vyriešené [36]. V tomto navrhnutom algoritme je však vyhovujúcejšie hľadať uzly, ktoré nemajú žiadnych následníkov. Je to z dôvodu, že graf v navrhnutom algoritme je postavený nasledujúco. Premenná, ktorá nemá žiadneho následníka predstavuje premennú, ktorá nie je nikde priradená. V prípade, že je inštrukcia `phi` definovaná ako v ukážke 6.14, tak orientovaný graf odpovedajúci tejto inštrukcii je zobrazený na obrázku 6.15.

```
%A = phi i32 [ %B, %pred ]
```

Obr. 6.14: Ukážka inštrukcie `phi` pre demonštráciu grafu topologického radenia.

V grafe 6.15 sa nachádzajú dva uzly, konkrétne `A` a `B`, pričom uzol `B` je zdrojový a uzol `A` cieľový. Zdrojový uzol predstavuje tú premennú, ktorá je priraďovaná. Cieľový uzol predstavuje premennú, do ktorej sa priraďuje. Pokiaľ by uvedená inštrukcia `phi` obsahovala aj ďalšie

priradenie premennej pomocou iného predchádzajúceho bloku, tak pre toto priradenie by bola vytvorená ďalšia hrana charakterizujúca toto priradenie.



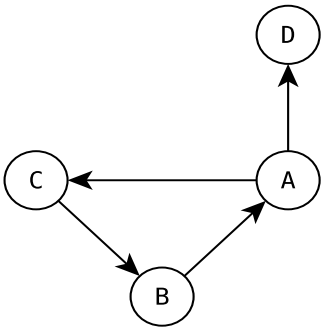
Obr. 6.15: Orientovaný graf závislosti premenných pre ukážku 6.14.

Nasledujúca časť textu sa venuje popisu algoritmickej stránke riešenia problémov uvedených v oddiele 6.1.2. Ako už bolo zmienené, najprv je potrebné vyriešiť cyklické závislosti premenných. Ukážka 6.16 obsahuje cyklickú závislosť medzi použitými premennými v inštrukciách `phi`.

```
pred:
  ...
  br label %bb

bb:
  %A = phi i32 [ %B, %pred ]
  %B = phi i32 [ %C, %pred ]
  %C = phi i32 [ %A, %pred ]
  %D = phi i32 [ %A, %pred ]
  ...
...
```

Obr. 6.16: Ukážka obsahujúca cyklickú závislosť premenných v inštrukciách `phi`.

Graf odpovedajúci tejto závislosti premenných je možné vidieť na obrázku 6.17. Na grafe je vidieť jednoznačná cyklická závislosť. Nad takýmto grafom by nebolo možné vytvoriť topologické radenie. Je preto potrebné identifikovať cyklus v grafe.



Obr. 6.17: Graf odpovedajúci cyklickej závislosti premenných z ukážky 6.16.

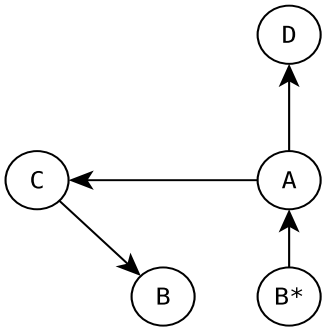
K tomuto účelu bol zvolený priechod do hĺbky - DFS (z angl. *depth-first search* [40]) cez všetky uzly. Cyklus je spoznaný tak, že pri priechode je navštívený ten uzol, ktorý už bol raz navštívený pri tomto priechode. Následne dochádza k odobratiu hrany spôsobujúcej cyklus. Potom algoritmus pokračuje iteráciou cez ďalšie hrany, pokiaľ daný uzol nejaké ešte obsahuje. Pre lepšie pochopenie si uvedieme jeden priechod DFS. Začnime v uzle `B`,

nasledujeme uzlom `A`, prechádzame do uzlu `C` a dostávame sa opäť do `B`, odstraňujeme hranu vedúcu z `B` do `A`. Táto hrana predstavuje inštrukciu `phi`: `%A = phi i32 [ %B, %pred ]`. Túto inštrukciu `phi` si algoritmus poznačí ako inštrukciu spôsobujúcu cyklus. Na ňu je potom možné uplatniť úpravu, ktorá bola popísaná v oddiele 6.1.2, konkrétne pri popise riešenia druhého problému. Pôvodný kód z ukážky 6.16 je touto zmienenou úpravou zmenený na kód v ukážke 6.18.

```
pred:
  ...
  br label %bb.phi2seq.pre

bb.phi2seq.pre
  %B.phi2seq.tmp = phi i32 [ %B, %pred ]
  br %.bb

bb:
  %A = phi i32 [ %B.phi2seq.tmp, %bb.phi2seq.pre ]
  %B = phi i32 [ %C, %bb.phi2seq.pre ]
  %C = phi i32 [ %A, %bb.phi2seq.pre ]
  %D = phi i32 [ %A, %bb.phi2seq.pre ]
  ...
...
```

Obr. 6.18: Ukážka LLVM IR kódu po vykonaní úpravy nad príkladom 6.16.

Zároveň je však potrebné aby došlo k úprave grafu na taký, ktorý odpovedá aktuálnemu kódu v jazyku LLVM IR. Graf na obrázku 6.19 znázorňuje upravený kód z ukážky 6.18, pričom uzol `B*` predstavuje pre prehľadnosť grafu premennú `B.phi2seq.tmp`. Je vidieť, že cyklus bol odstránený. Usporiadanie inštrukcií však ešte nie je správne, pretože premenná `A` je prepísaná hneď na začiatku bloku a táto premenná je používaná pri priradení do `C` a `D`.



Obr. 6.19: Upravený graf po odstránení cyklických závislostí premenných.

Ako bolo už zmienené, tak je na záver ešte potrebné usporiadať tieto inštrukcie `phi` do korektného poradia. Dôvody a ciele tohto usporiadania už boli popísané v oddiele 6.1.2, konkrétne pri popise prvého problému. K zoradeniu inštrukcií `phi` je opäť využívaný priechod DFS, pričom je spúšťaný nad každým uzlom. Pri tomto priechode dochádza k ukladaniu si uzlov, ktoré neobsahujú žiadneho následníka a predstavujú nejakú `phi` inštrukciu. Pre lepšie pochopenie si uvedieme jeden priechod DFS. Začnime v uzle `B*`. Potom pokrač-

jeme po následníkovi do uzla A, následne do uzla D, pričom uzol D nemá žiadneho následníka, tak si ho poznačíme. Následne sa vrátime do uzla A, odkiaľ pokračujeme cez C do B. Poznačíme si B, ďalej C, potom A a nakoniec B*. Týmto poznačovaním dôjdeme k získaniu poradia uzlov a to: D, B, C, A, B*. Z tohto zoradenia je následne možné zostaviť výsledok, pričom pre pripomenutie napríklad hrana z A do D predstavuje, že premenná A je priradená do D. Nasledujúci príklad 6.20 demonštruje výsledné zoradenie. V tomto príklade je zobrazený pre jednoduchosť len jeden blok, v ktorom dochádzalo k zoraďovaniu, pričom toto zoraďovanie bolo realizované nad ukážkou 6.18. Je možné vidieť, že pri tomto kóde už platí, že paralelné spracovanie inštrukcií phi je ekvivalentné s tým sekvenčným.

```
bb:
  %D = phi i32 [ %A, %bb.phi2seq.pre ]
  %B = phi i32 [ %C, %bb.phi2seq.pre ]
  %C = phi i32 [ %A, %bb.phi2seq.pre ]
  %A = phi i32 [ %B.phi2seq.tmp, %bb.phi2seq.pre ]
  ...
...
```

Obr. 6.20: Výsledné zoradenie inštrukcií phi, pri ktorom platí ekvivalentnosť paralelného a sekvenčného spracovania inštrukcií phi.

Navrhnutý algoritmus sa skladá z dvoch častí. Prvá časť rieši problém cyklických závislostí a druhá časť správne zoradenie inštrukcií phi. Obidve tieto časti spúšťajú DFS priechody nad uzlami grafu. Z dôvodu, aby tieto priechody DFS boli efektívnejšie, bol vyššie uvedený algoritmus vylepšený o poznačovanie si spracovaných uzlov a tým pádom nedochádza k zbytočnému spusteniu DFS priechodov už nad spracovanými uzlami. Takisto vďaka tomuto označovaniu nedochádza k opätovnému priechodu ciest a uzlov, ktoré už boli raz navštívené. DFS priechody boli zvolené z dôvodu, že vďaka ním je riešenie topologického radenia realizovateľné s lineárnou časovou zložitosťou. Lepšiu časovú zložitosť pre vykonanie topologického radenia nie je s pomocou známych algoritmov možné dosiahnuť [36].

6.2 Odstránenie mŕtvych priradení do globálnych premenných

Táto optimalizácia je jednou z tých, ktoré boli navrhnuté za účelom presunu optimalizácií do strednej časti zo zadnej časti spätného prekladača. Dôvody presunu boli popísané v kapitole 6. Aby mohlo dôjsť k tomuto presunu, bolo potrebné navrhnúť túto optimalizáciu tak, aby pokrývala celú funkčnosť optimalizácie mŕtvych priradení nachádzajúcej sa v zadnej časti spätného prekladača. Súčasne bolo nutné dbať pri návrhu aj na rýchlosť danej optimalizácie, a tak ju navrhnúť čo najefektívnejšie. Optimalizácia mŕtvych priradení v zadnej časti spätného prekladača nepokrýva odstraňovanie úplne všetkých mŕtvych priradení, a preto vznikol zároveň priestor pre rozšírenie tejto funkčnosti v strednej časti spätného prekladača.

Pod pojmom mŕtve priradenie do globálne premennej rozumieme situáciu, pri ktorej dochádza k priradeniu hodnoty do globálnej premennej a následne dôjde k opätovnému priradeniu nejakej hodnoty do tejto globálnej premennej, čím dôjde k prepísaniu predchádzajúcej uloženej hodnoty. Dôležité je ale splniť ešte jednu podmienku, aby išlo o mŕtve priradenie.

Priradená hodnota do globálnej premennej a následne prepísaná nesmie byť nikde v kóde prečítaná z tejto globálnej premennej. Ukážka 6.21 demonštruje príklad mŕtveho priradenia v jazyku C.

```
int g = 0;

int main() {
  g = 2; // Mŕtve priradenie.
  ...    // Globálna premenná g nie je nikde priradená
         // do inej premennej.
  g = 4;
}
```

Obr. 6.21: Ukážka obsahujúca mŕtve priradenia do globálnej premennej g.

Pokiaľ by ukážka 6.21 bola upravená napríklad na ukážku 6.22, tak v tomto prípade by už nešlo o mŕtve priradenie. Je to z toho dôvodu, že hodnota 2 je priradená do globálnej premennej g, následne je táto hodnota priradená do lokálnej premennej v1, kde je táto hodnota použitá k výpisu na štandardný výstup. Odstránenie priradenia g = 2 by spôsobilo zmenu chovania programu, keďže na štandardný výstup by bola vypísaná hodnota 0.

```
int g = 0;

int main() {
  g = 2; // Nie je mŕtve priradenie.
  int v1 = g;
  printf("%d", v1);
}
```

Obr. 6.22: Ukážka neobsahujúca mŕtve priradenia do globálnej premennej g.

Za mŕtve priradenie je možné považovať aj nasledujúcu situáciu uvedenú na ukážke 6.23. V tomto prípade by mohlo byť takéto priradenie skôr označené ako zbytočné, avšak pre jednotnosť sa bude aj pre takéto priradenie používať pojem mŕtve priradenie. Mŕtve priradenie je to však len za predpokladu, že funkcia func nie je nikde volaná, prípadne ak aj je, tak nikde v kóde nie je táto konkrétna hodnota prečítaná z globálnej premennej g.

```
int g = 0;

void func() {
  ...
  g = 3; // Mŕtve priradenie.
  return;
}
```

Obr. 6.23: Ukážka obsahujúca mŕtve priradenia do globálnej premennej g na konci funkcie.

Syntax a sémantiku inštrukcií úzko súvisiacich s touto optimalizáciou popisuje oddiel 6.2.1. Návrh optimalizácie je z dôvodu jej zložitosti rozdelený do niekoľkých oddielov. Oddiel 6.2.2 obsahuje popísané špecifické prípady, pri ktorých nemá dochádzať k optimalizovaniu mŕtvych priradení do globálnej premennej. Tieto situácie bolo nutné zakomponovať do navrhovaného algoritmu. Oddiel 6.2.3 obsahuje popis akým spôsobom sú prechádzané a spra-

covávané základné bloky. Priechodu funkciami a ich spracovaniu sa venuje oddiel 6.2.4. Získavané a šírené informácie medzi blokmi sú popísané v oddiele 6.2.5. V tomto oddiele sa nachádza aj uvedenie dôvodov, prečo sú tieto informácie potrebné. Posledný oddiel 6.2.6 sa venuje princípu odstraňovania mŕtvych priradení.

6.2.1 Syntax a sémantika inštrukcií súvisiacich s mŕtvymi priradeniami

Pre priradenie hodnoty do globálnej premennej sa v LLVM využíva inštrukcia `store`, ktorej syntax je uvedená na ukážke 6.24.

```
store [volatile] <ty> <value>, <ty>* <pointer>
    [, align <alignment>][, !nontemporal !<index>]
store atomic [volatile] <ty> <value>, <ty>* <pointer> [singlethread]
    <ordering>, align <alignment>
```

Obr. 6.24: Syntax inštrukcie `store` [23, viď sekcia store Instruction].

Najdôležitejšie argumenty tejto inštrukcie pre túto optimalizáciu sú `<value>` a `<pointer>`. Argument `<value>` predstavuje priraďovanú hodnotu. Argument `<pointer>` obsahuje adresu, do ktorej sa priraďuje hodnota [23, viď sekcia store Instruction]. Ukážka 6.25 predstavuje uloženie hodnoty 2 do globálnej premennej `g`. V LLVM sa v prípade globálnych premenných vždy používa k ich prístupu adresa [23], čo je rovnako možné vidieť na tejto uvedenej ukážke.

```
store i32 2, i32* @g
```

Obr. 6.25: Ukážka priradenia hodnoty 2 do globálnej premennej `g`.

Pre prístup k hodnote uloženej v nejakej premennej sa v LLVM používa inštrukcia `load`. Jej syntax je uvedená na ukážke 6.26.

```
<result> = load [volatile] <ty>* <pointer>[, align <alignment>]
    [, !nontemporal !<index>][, !invariant.load !<index>]
    [, !nonnull !<index>]
<result> = load atomic [volatile] <ty>* <pointer> [singlethread]
    <ordering>, align <alignment> !<index> = !{ i32 1 }
```

Obr. 6.26: Syntax inštrukcie `load` [23, viď sekcia load Instruction].

V tomto prípade sú najdôležitejší argumenty `<pointer>` a `<result>`. Argument `<pointer>` predstavuje adresu, z ktorej dochádza k načítaniu hodnoty a argument `<result>` premennú, do ktorej sa priraďuje načítaná hodnota. Príklad 6.27 predstavuje načítanie hodnoty globálnej premennej `g` a jej uloženie hodnoty do lokálnej premennej `loc`.

```
%loc = load i32* @g
```

Obr. 6.27: Príklad získania hodnoty z globálnej premennej `g`.

Syntax LLVM IR neumožňuje načítanie hodnoty a jej okamžité priradenie pomocou inštrukcie `load` do globálnej premennej. Je potrebné využiť najprv inštrukciu `load` a následne načítanú hodnotu uložiť pomocou inštrukcie `store` do globálnej premennej [23]. Príklad 6.28 obsahuje popísaný chybný postup a zároveň aj ten korektný.

```
; Chybný postup:
@g = load i32* %var

; Korektný postup:
%z = load i32* %var
store i32 %z, i32* @g
```

Obr. 6.28: Príklad nesprávneho a správneho priradenia hodnoty z lokálnej premennej do globálnej premennej.

Ukážku 6.21, ktorá demonštrovala príklad mŕtveho priradenia a bola prezentovaný pre jednoduchosť v jazyku C, je možné vidieť na ukážke 6.29 v jazyku LLVM IR.

```
@g = global i32 0

define i32 @main() {
  store i32 2, i32* @g ; Mŕtve priradenie.
  ... ; Premenná g nie je nikde priradená do inej premennej.
      ; Nevyskytuje sa napr. inštrukcia: %v1 = load i32* @g.
  store i32 3, i32* @g;
  ...
}
```

Obr. 6.29: Ukážka 6.21 prevedená do jazyka LLVM IR.

6.2.2 Špecifické prípady, pri ktorých nedochádza k optimalizácii

Nasledujúca časť textu sa venuje špecifickým situáciám, pri ktorých nedochádza k optimalizovaniu a bolo ich potrebné zohľadniť v návrhu algoritmu.

Optimalizácia nepodporuje optimalizovanie takých globálnych premenných a ich priradení, pri ktorých dochádza k prevzatiu adresy tejto globálnej premennej, pretože táto adresa môže byť použitá a tým pádom by iná premenná mohla rovnako predstavovať túto globálnu premennú, ktorej adresa bola prevzatá. Pri vytváraní návrhu bola snaha nájsť čo najvhodnejšiu podporu z LLVM, ktorá by zabezpečila možnú identifikáciu premenných, ktoré pri danom behu toku programu predstavujú nejakú konkrétnu globálnu premennú. Jedinou potenciálne použiteľnou sa ukazovala takzvaná *AliasAnalysis* [26], ktorá sa nachádza vo frameworku LLVM. Bohužiaľ táto analýza nebola dostatočná a neobsahovala potrebnú podporu, ktorá by sa dala využiť pre vylepšenie tejto optimalizácie. Implementácia vlastnej analýzy riešiacej tento problém by prekračovala náročnosť celej diplomovej práce. Vzhľadom na to sa po dohode s vedúcim práce rozhodlo, že pre globálne premenné, pri ktorých dôjde k prevzatiu ich adresy, nebudú odstraňované žiadne mŕtve priradenia. Rovnako z toho istého dôvodu nie sú optimalizované ani globálne ukazovatele. Toto správanie je rovnaké aj v optimalizácii mŕtvych priradení do globálnych premenných v zadnej časti spätného prekladača.

Optimalizácia neumožňuje optimalizovať priradenia, ktoré sú označené ako `volatile`. Tieto priradenia nie sú optimalizované, pretože LLVM vyžaduje po optimalizáciách aby neoptimalizovali `volatile` priradenia [28].

Optimalizácia takisto nepodporuje optimalizovanie agregovaných typov ako napr. štruktúry, polia a pod. Pri poliach nie je vždy možné vyhodnotiť, na ktoré miesto sa priraďuje ľubovoľná hodnota. Je to z toho dôvodu, že niektoré informácie je možné získať až za behu programu. Ukážka 6.30 demonštruje tento prípad. Pre jednoduchosť je uvedená v jazyku C.

```
int array[20];
array[i] = 5;
```

Obr. 6.30: Ukážka znázorňujúca príčiny neoptimalizovania globálnych polí.

V prípade štruktúr by bolo nutné sledovať, či nedochádza k získavaniu adresy nejakej položky štruktúry. Ďalším problémom je možnosť prepísania hodnoty štruktúry nepriamo. Ukážka 6.31 v jazyku C demonštruje tento prípad.

```
struct {
  int p[10];
  int i;
} s = {.p = {0}, .i = 0};
s.p[10] = 123;
printf("%d\n", s.i); // Vypíše sa 123.
return 0;
```

Obr. 6.31: Ukážka znázorňujúca príčiny neoptimalizovania štruktúr.

Z vyššie uvedených dôvodov nedochádza k optimalizovaniu globálnych premenných predstavujúcich agregované typy. Rovnaké správanie je aj v optimalizácii mŕtvych priradení v zadnej časti spätného prekladača.

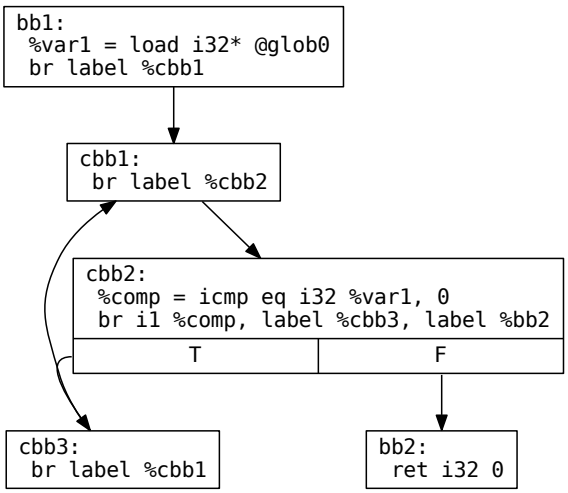
6.2.3 Priechod základnými blokmi a ich spracovanie

Táto optimalizácia odstraňuje mŕtve priradenia až po prebehnutí analýzy určujúcej, ktoré priradenia je možné považovať ako mŕtve a následne ich odstrániť. Táto analýza je vykonávaná prostredníctvom priechodu blokmi a inštrukcií v týchto blokoch. Priechod je realizovaný tak, že sa vždy pre základný blok prechádzajú jeho predchodcovia. Inštrukcie v bloku sú tiež prechádzane pomocou spätného priechodu. Pri priechode je však potrebné niekde začať. Začína sa s blokmi, ktoré nemajú ani jedného následníka. Tieto bloky predstavujú miesta, kde dochádza k návratu toku programu z funkcie.

Analýza počas priechodov blokmi zbiera potrebné informácie, na základe ktorých je neskôr možné určiť, či je priradenie možné považovať za mŕtve priradenie. Vzhľadom na to, že je šírené veľké množstvo informácií pri priechode blokmi, tak je potrebné dbať na čo najmenšie množstvo redundantných priechodov blokmi. Veľké množstvo priechodov by viedlo k spomaľovaniu tejto optimalizácie.

Pri spracovaní bloku je potrebné zohľadniť aj získané informácie z blokov, ktoré predstavujú následníkov pre tento blok. V prípade, že by dochádzalo k vyhodnocovaniu bloku bez toho, aby boli vyhodnotení jeho následníci, bolo by nutné tento blok vyhodnotiť viac než raz. Je to z toho dôvodu, že získané informácie z následníkov ovplyvňujú aj tento blok. Preto bolo nutné navrhnúť tento algoritmus tak, aby spracovával blok len v prípade, že sú spracovaní všetci jeho následníci. Pre tento účel bola vytvorená analýza, ktorá pomocou takzvaného priechodu *post-order* [41] určuje, ktorý blok má byť v akom poradí vyhodnotený.

Medzi blokmi môže dochádzať aj k cyklickým závislostiam. Príklad takejto cyklickej závislosti je znázornený na obrázku 6.32. V prípade cyklov je potrebný viacnásobný priechod blokmi tvoriacich tento cyklus. Je to z toho dôvodu, že cyklus predstavuje takú závislosť, pri ktorej nie je možné dosiahnuť, že každý blok bude mať vyhodnotených všetkých svojich následníkov. Preto je potrebné viacnásobne prechádzať bloky tohto cyklu a vykonávať to do doby, pokiaľ dochádza k zmene získavaných informácií.



Obr. 6.32: Cyklická závislosť medzi základnými blokmi.

Pre zminimalizovanie počtu redundantných navštívení blokov je potrebné uprednostniť spracovanie blokov, ktoré tvoria cyklické závislosti. Tieto cyklické závislosti tvoria takzvané silne súvislé komponenty [39] v grafu toku riadenia. Hlavný prínos pre uprednostnenie spracovania silne súvislých komponent je ten, že po vyhodnotení tejto komponenty môže dôjsť k šíreniu získaných informácií do iných blokov. Pokiaľ by však nedochádzalo prioritne k vyhodnoteniu silne súvislej komponenty, tak by nastávala nasledujúca situácia. V prvom priechode silne súvislej komponenty by došlo k čiastočnému vyhodnoteniu tejto komponenty a získané informácie by boli šírené k ostatným blokom. Potom následne pri ďalšom priechode tohto cyklu by došlo k zmene týchto získaných informácií a bolo by potrebné opäť šíriť túto zmenu do ostatných blokov. Takéto správanie je však veľmi nevýhodné a preto je výhodnejšie uprednostniť vyhodnotenie silne súvislej komponenty. LLVM obsahuje podporu pre vyhľadávanie silne súvislých komponent medzi blokmi a tak je táto analýza využitá. Na základe toho, že je možné určiť, ktorý blok tvorí silne súvislú komponentu, je možné uprednostniť vyhodnotenie týchto blokov pred ostatnými a tým dosiahnuť prioritné vyhodnocovanie cyklov. Samozrejme musí byť splnené, že všetky bloky, ktoré sú následníkmi blokov tvoriacich silne súvislú komponentu, sú už vyhodnotené prostredníctvom analýzy.

6.2.4 Priechod funkciami a ich spracovanie

V spracovávaných základných blokoch sa bežne nachádzajú aj volania funkcií. Tieto volania ovplyvňujú získavané informácie, nakoľko platnosť priradenia do globálnej premennej

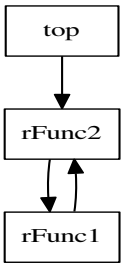
je i mimo funkciu, v ktorej bolo toto priradenie realizované. Opäť z dôvodu aby nedochádzalo k zbytočnému vyhodnocovaniu základných blokov je potrebné zabezpečiť, že pri spracovaní bloku obsahujúceho volanie funkcie bude platiť, že táto funkcia už bola vyhodnotená a všetky jej základné bloky spracované. K zaisteniu toho, aby funkcie boli vyhodnocované v správnom poradí je využitá analýza silne súvislých komponent z LLVM, z ktorej je práve možno získať žiadané poradie pre korektné vyhodnocovanie funkcií. Táto analýza zabezpečuje, že pokiaľ funkcia 1 volá funkciu 2, tak ako prvú funkciu k vyhodnoteniu vráti funkciu 2 a následne funkciu 1.

Existuje však prípad, pri ktorom nie je možné zabezpečiť, že volaná funkcia bude vyhodnotená pred týmto volaním. K takejto situácií dochádza v prípade rekurzie. Funkcie, ktoré sú medzi sebou rekurzívne závislé tvoria silne súvislú komponentu v grafe volaní funkcií. Pri takýchto funkciách je nutné prechádzať celú silne súvislú komponentu do doby, pokiaľ dochádza k zmenám v získavaných informáciách čo i len v jednej funkcii. Na ukážke 6.33 je možné vidieť vzájomnú rekurziu medzi funkciami `rFunc1` a `rFunc2`. Graf volania odpovedajúci ukážke 6.33 je vidieť na obrázku 6.34.

```
define i32 @top() {
bb:
  call i32 @rFunc2()
  ret i32 0
}

define i32 @rFunc1() {
bb1:
  br i1 1, label %return, label %bb2

bb2:
  call i32 @rFunc2()
  ret i32 0

return:
  ret i32 0
}

define i32 @rFunc2() {
bb:
  call i32 @rFunc1()
  ret i32 0
}
```

Obr. 6.33: Ukážka zobrazujúca rekurzívne volanie funkcií.

Analýza z LLVM nad uvedeným príkladom vráti funkcie v nasledujúcom poradí. V prvom prípade vráti funkciu `rFunc1` a potom `rFunc2`. Súčasne informuje o tom, že tieto dve funkcie tvoria silne súvislú komponentu. Následne vráti funkciu `top`. Ako je možné vidieť, je to presne také poradie, aké je potrebné. Najprv dôjde k vyhodnoteniu tejto rekurzie a výsledok vyhodnotenia je potom možné použiť pri volaní funkcie `rFunc2` z funkcie `top`.



Obr. 6.34: Graf volania pre funkcie z ukážky 6.33.

6.2.5 Informácie uchovávané počas analýzy

Nasledujúca časť textu sa bude venovať šíriacim sa informáciám medzi blokmi. Tieto informácie ako už bolo zmienené sú šírené z dôvodu, aby bolo možné určiť, ktoré priradenie do globálnej premennej je možné považovať za mŕtve. Ukážky v nasledujúcom texte obsahujú príklady, pri ktorých sú zobrazené len podstatné časti pre pochopenie príkladu.

Pod šírenými informáciami si je možné predstaviť tri množiny. V nasledujúcich príkladoch sú vždy zobrazené len množiny potrebné pre pochopenie daného príkladu. Obecne ale dochádza k šíreniu všetkých troch množín.

Jednou z uchovávaných informácii je ukladanie si takzvaného *pravého použitia globálnej premennej*. V texte bude množina pravých použití označovaná skratkou PP. Pod pravým použitím sa myslí taká situácia, kde je globálna premenná priraďovaná do nejakej inej premennej. Táto informácia sa využíva pre rozhodovanie o tom, či je priradenie do globálnej premennej možné považovať za mŕtve priradenie. Na ukážke 6.35 je možné vidieť opodstatnenie uchovávania tejto informácie. Po spracovaní inštrukcie `load` sa uchová informácia o pravom použití pre globálnu premennú `g`. Následne sa pri spracovaní inštrukcie `store` využije táto uložená informácia v množine PP a to tak, že k tomuto priradeniu sa uloží informácia o tom, že toto priradenie ovplyvňuje konkrétne uložené pravé použitie v množine PP. Ovplyvnením sa myslí to, že hodnota, ktorá sa uloží do premennej `x` je určená práve týmto priradením do globálnej premennej. Samozrejme po spracovaní inštrukcie `store` je potrebné upraviť množinu PP, pretože toto pravé použitie sa naviazalo už na konkrétne priradenie hodnoty do globálnej premennej.

```
...
store i32 1, i32* @g ; PP = {}
%x = load i32* @g    ; PP = {%x = load i32* @g}
...
```

Obr. 6.35: Príklad zobrazujúci opodstatnenie množiny PP.

Ďalšou uchovávanou informáciou sú takzvané *posledné ľavé použitia*. V texte bude množina posledných ľavých použití označovaná skratkou PLP. Ide o také priradenie do globálnej premennej, ktoré po ukončení behu funkcie zabezpečí, že daná globálna premenná bude mať práve túto hodnotu. Príklad zobrazujúci túto množinu ako aj jej opodstatnenie znázorňuje ukážka 6.36. V tomto príklade inštrukcia `store i32 2, i32* @g` predstavuje posledné ľavé

použitie, pretože po ukončení behu funkcie `func2` bude mať globálne premenná `g` hodnotu 2. Vďaka uloženej informácii o poslednom priradení do globálnej premennej vieme určiť, že priradenie `store i32 2, i32* @g` ovplyvňuje pravé použitie šíriace sa vďaka inštrukcii `%x = load i32* @g`, keďže sa v tomto pravom použití využije priradená hodnota.

```
@g = global i32 0

define i32 @func1() {   ; PP = {}, PLP = {store i32 2, i32* @g}
  call i32 func2()      ; PP = {}, PLP = {store i32 2, i32* @g}
  %x = load i32* @g     ; PP = {%x = load i32* @g}, PLP = {}
  ret i32 0             ; PP = {}, PLP = {}
}

define i32 @func2() {   ; PLP = {store i32 2, i32* @g}
bb:
  store i32 1, i32* @g ; PLP = {store i32 2, i32* @g}
  store i32 2, i32* @g ; PLP = {store i32 2, i32* @g}
  ret i32 0             ; PLP = {}
}
```

Obr. 6.36: Príklad zobrazujúci opodstatnenie množiny PLP.

Poslednou uchovávanou informáciou je informácia o tom, či v danom toku už došlo k uchovaniu nejakého posledného ľavého použitia pre danú globálnu premennú (v texte bude táto množina označovaná skratkou EPLP). Táto informácia sa používa k tomu, aby bolo možné správne identifikovať posledné ľavé použitie. Ukážka 6.37 zobrazuje uvedený princíp. Na základe informácie, že už je uložené posledné ľavé použitie nedôjde k uloženiu inštrukcie `store i32 1, i32* @g` ako ďalšie posledné ľavé použitie.

```
...
bb:
  store i32 1, i32* @g ; PLP = {store i32 2, i32* @g}, EPLP = {g}
  store i32 2, i32* @g ; PLP = {store i32 2, i32* @g}, EPLP = {g}
  ret i32 0            ; PLP = {}, EPLP = {}
```

Obr. 6.37: Príklad zobrazujúci opodstatnenie množiny EPLP.

Uvedené informácie sa postupne šíria a upravujú pri prechode blokmi. Bloky v LLVM však nie vždy na seba nadväzujú lineárne. Je bežné, že blok v LLVM kóde obsahuje viac než jedného následníka, a preto je potrebné tieto získané informácie nejako zlúčiť. Napríklad ak blok obsahuje práve dvoch odlišných následníkov, tak v prípade množiny PP platí vzťah 6.1.

$$\mathrm{PP}^{\mathrm{bloku}} = \mathrm{PP}^{\mathrm{následník1}} \cup \mathrm{PP}^{\mathrm{následník2}} \tag{6.1}$$

Rovnaký princíp platí aj pre množinu PLP. Zlučovanie samozrejme platí i v prípade viacerých než dvoch blokov. Príklad 6.38 demonštruje, prečo je potrebné zjednotenie. V bloku `bb` je nutné uchovať informáciu o obidvoch pravých použitiach v blokoch `l` a `r` a rovnako tak platí aj pre posledné ľavé použitia. Preto dôjde pri poslednej inštrukcii v bloku `bb` k zjednoteniu množiny PLP a PP.

```
...
bb:
  %c = icmp eq i32 %var1, 0      ; PP = {%x = load i32* @g0,
                                 ;   %y = load i32* @g1},
                                 ; PLP = {store i32 1, i32* @g0,
                                 ;   store i32 1, i32* @g1}
  br i1 %c, label %l, label %r   ; PP = {%x = load i32* @g0,
                                 ;   %y = load i32* @g1},
                                 ; PLP = {store i32 1, i32* @g0,
                                 ;   store i32 1, i32* @g1}
l:
  %x = load i32* @g0      ; PP = {%x = load i32* @g0},
                          ; PLP = {store i32 1, i32* @g0}
  store i32 1, i32* @g0   ; PP = {}, PLP = {store i32 1, i32* @g0}
  ...

r:
  %y = load i32* @g1      ; PP = {%y = load i32* @g1},
                          ; PLP = {store i32 1, i32* @g1}
  store i32 1, i32* @g1   ; PP = {}, PLP = {store i32 1, i32* @g1}
  ...
```

Obr. 6.38: Príklad zobrazujúci potrebu zjednotenia pre množiny PP a PLP.

V prípade množiny EPLP platí vzťah 6.2.

$$\mathrm{EPLP}^{\mathrm{bloku}} = \mathrm{EPLP}^{\mathrm{následník1}} \cap \mathrm{EPLP}^{\mathrm{následník2}} \tag{6.2}$$

Dôvod, prečo je potrebný práve prienik je zobrazený na príklade 6.39. Pokiaľ by nedošlo k prieniku, ale napríklad k zjednoteniu, tak by sme stratili informáciu o tom, že v bloku `r` neexistuje ešte žiadna informácia o poslednom ľavom použití. K tomuto prieniku dôjde na poslednej inštrukcii bloku `bb`.

```
...
bb:
  store i32 1, i32* @g          ; PLP = {store i32 1, i32* @g,
                                ;   store i32 2, i32* @g}, EPLP = {g}
  %c = icmp eq i32 %var1, 0     ; PLP = {store i32 2, i32* @g},
                                ; EPLP = {}
  br i1 %c, label %l, label %r  ; PLP = {store i32 2, i32* @g},
                                ; EPLP = {}

l:
  store i32 2, i32* @g ; PLP = {store i32 2, i32* @g}, EPLP = {g}
  ret i32 0            ; PLP = {}, EPLP = {}

r:
  ret i32 0            ; PLP = {}, EPLP = {}
```

Obr. 6.39: Príklad zobrazujúci potrebu prieniku pre množiny EPLP.

Pokiaľ dôjde k spracovaniu vstupného bloku funkcie, tak získané množiny sa uložia pre funkciu, v ktorej sa tento vstupný blok nachádza. Vďaka tomu sú získané potrebné informácie,

ktoré je možno použiť pri volaní tejto funkcie. Pri spracovaní bloku môže nastať situácia, že z daného bloku je volaná nejaká funkcia. Preto je potrebné využiť informácie z analyzovanej funkcie a následne podľa toho príslušne upraviť informácie v tomto bloku. Pri volaní funkcie z bloku platí, že vždy je volaná funkcia už analyzovaná. Akým spôsobom je zabezpečený tento fakt je možné dočítať sa v oddiele 6.2.4.

Pre spracovanie množiny PP z volanej funkcie s množinou PP v danom bloku, v ktorom je volaná funkcia platí vzťah 6.3.

$$\mathrm{PP}^{\text{celkové}} = \left(\mathrm{PP}^{\text{bloku}} \cup \mathrm{PP}^{\text{funkcie}}\right) - \mathrm{EPLP}^{\text{funkcie}} \tag{6.3}$$

Rozdiel o množinu $\mathrm{EPLP}^{\text{funkcie}}$ je potrebný z toho dôvodu, že pokiaľ pre danú globálnu premennú je uchovaný záznam v množine $\mathrm{EPLP}^{\text{funkcie}}$, tak je isté, že neexistuje taký tok programu, v ktorom sa nenaviaže toto pravé použitie globálnej premennej na nejaké posledné ľavé použitie. Nasledujúci príklad 6.40 pre názornosť demonštruje popísané. Realizáciu vzťahu 6.3 je možné vidieť pri volaní funkcie `func2`, pri ktorom dôjde k odstráneniu pravého použitia pre globálnu premennú `g0` z množiny PP na základe toho, že globálna premenná `g0` sa nachádza v množine EPLP volanej funkcie.

```
@g0 = global i32 0
@g1 = global i32 0

define i32 @func1() {    ; PP = {%y = load i32* @g1}
bb:
  call i32 @func2()      ; PP = {%y = load i32* @g1}
  %x = load i32* @g0     ; PP = {%x = load i32* @g0,
                         ;   %y = load i32* @g1}
  %y = load i32* @g1     ; PP = {%y = load i32* @g1}
  ret i32 0              ; PP = {}
}

define i32 @func2() {    ; PP = {}, EPLP = {g0}
bb:
  store i32 1, i32* @g0  ; PP = {}, EPLP = {g0}
  ret i32 0              ; PP = {}, ELP = {}
}
```

Obr. 6.40: Príklad zobrazujúci spracovania volania funkcie pre množinu PP.

Pre spracovanie množiny PLP z volanej funkcie platí nasledujúci vzťah 6.4.

$$\mathrm{PLP}^{\text{celkové}} = \left(\mathrm{PLP}^{\text{funkcie}} - \mathrm{EPLP}^{\text{bloku}}\right) \cup \mathrm{PLP}^{\text{bloku}} \tag{6.4}$$

Rozdiel $\mathrm{EPLP}^{\text{bloku}}$ je potrebný, pretože $\mathrm{EPLP}^{\text{bloku}}$ obsahuje globálne premenné, ktoré už majú v tomto toku kódom uložené posledné ľavé použitie. Ukážka 6.41 znázorňuje princíp spracovania množiny PLP po volaní funkcie. V tejto ukážke množina PLP obsahuje celú inštrukciu, avšak pri tomto rozdiele množín je potrebné uvažovať, že je uchovaná aj informácia o tom pre akú globálnu premennú je toto posledné ľavé priradenie vykonané. Pri volaní funkcie `func2` v bloku `bb` dôjde k uplatneniu vzťahu 6.4. Je vidieť, že po tomto volaní sa do množiny PLP nepridalo posledné ľavé použitie z funkcie `func2` pre globálnu

premennú g0, pretože sa už táto globálna premenná nachádza pri volaní funkcie v množine EPLP, čo značí, že pre túto globálnu premennú je už uložené posledné ľavé použitie.

```
@g0 = global i32 0
@g1 = global i32 0

define i32 @func1() {    ; PLP = {store i32 1, i32* @g0,
                         ;   store i32 1, i32* @g1}, EPLP = {g0, g1}
bb:
  call i32 @func2()      ; PLP = {store i32 1, i32* @g0,
                         ;   store i32 1, i32* @g1}, EPLP = {g0, g1}
  store i32 1, i32* @g0  ; PLP = {store i32 1, i32* @g0}, EPLP = {g0}
  ret i32 0              ; PLP = {}, EPLP = {}
}

define i32 @func2() {    ; PLP = {store i32 2, i32* @g0,
                         ;   store i32 1, i32* @g1}, EPLP = {g0, g1}
bb:
  store i32 2, i32* @g0  ; PLP = {store i32 2, i32* @g0,
                         ;   store i32 1, i32* @g1}, EPLP = {g0, g1}
  store i32 1, i32* @g1  ; PLP = {store i32 1, i32* @g1}, EPLP = {g1}
  ret i32 0              ; PLP = {}, EPLP = {}
}
```

Obr. 6.41: Príklad zobrazujúci spracovania volania funkcie pre množinu PLP.

Pre spracovanie množiny EPLP z volanej funkcie platí nasledujúci vzťah 6.5.

$$\mathrm{EPLP}^{\text{celkové}} = \mathrm{EPLP}^{\text{bloku}} \cup \mathrm{EPLP}^{\text{funkcie}} \tag{6.5}$$

Pokiaľ sa pre danú globálnu premennú nachádza záznam v $\mathrm{EPLP}^{\text{funkcie}}$, tak je jasné, že pre každú cestu toku programu je uložené pre túto globálnu premennú posledné ľavé použitie. Preto je potrebné šíriť túto informáciu a vykonať zjednotenie. Príklad 6.42 demonštruje tento princíp. Pri volaní funkcie `func2` v bloku `bb` dôjde k uplatneniu vzťahu 6.5. Ako je vidieť, tak pri tomto volaní funkcie dôjde k zjednoteniu množín EPLP. Zároveň si je možné pri tomto volaní všimnúť, že správne nedôjde k pridaniu inštrukcií `store i32 1, i32* @g0` a `store i32 1, i32* @g1` do množiny PLP vďaka množine EPLP.

V LLVM kóde sa môžu obecne vyskytovať také funkcie, ktorých definícia nie je dostupná. Ide napríklad o niektoré funkcie z knižníc. Na príklade 6.43 nie je možné určiť, či priradenie do globálnej premennej `g` je možné považovať za mŕtve, alebo nie, pretože premenná `g` môže predstavovať premennú, ktorej pravé použitie je v tejto volanej funkcii. Najčastejším prípadom sú premenné z hlavičkového súboru `errno.h`, ale môže ísť aj o iné premenné. Preto optimalizácia podporuje dva režimy. V agresívnom režime sa nepočíta s tým, že by bolo hocijaké pravé použitie vo volanej funkcii pre nejakú globálnu premennú. V neagresívnej verzii je tomu naopak. Pri neagresívnej verzii dochádza k nastaveniu množiny PP na všetky globálne premenné, pre ktoré sa odstraňujú mŕtve priradenia. Vďaka tomu sa zabezpečí to, že žiadne prvé priradenie od tohto volania funkcie pre danú globálnu premennú nebude optimalizované.

Pri volaní funkcií môže nastávať ešte jedna situácia. Volanie funkcie môže byť realizované nepriamo. Takáto situácia je viditeľná na ukážke 6.44. Pri týchto volaniach nie je možné

```
@g0 = global i32 0
@g1 = global i32 0

define i32 @func1() {    ; PLP = {store i32 2, i32* @g0,
                         ;   store i32 2, i32* @g1},  EPLP = {g0, g1}
bb:
  store i32 1, i32* @g0 ; PLP = {store i32 2, i32* @g0,
                         ;   store i32 2, i32* @g1},  EPLP = {g0, g1}
  store i32 1, i32* @g1 ; PLP = {store i32 2, i32* @g0,
                         ;   store i32 2, i32* @g1},  EPLP = {g0, g1}
  call i32 @func2()     ; PLP = {store i32 2, i32* @g0,
                         ;   store i32 2, i32* @g1} , EPLP = {g0, g1}
  store i32 2, i32* @g1 ; PLP = {store i32 2, i32* @g1}, EPLP = {g1}
  ret i32 0             ; PLP = {}, EPLP = {}
}

define i32 @func2() {    ; PLP = {store i32 2, i32* @g0}, EPLP = {g0}
bb:
  store i32 2, i32* @g0 ; PLP = {store i32 2, i32* @g0}, EPLP = {g0}
  ret i32 0 ;           ; PLP = {}, EPLP = {}
}
```

Obr. 6.42: Príklad zobrazujúci spracovania volania funkcie pre množinu EPLP.

```
declare void @externalCall()

declare void @func() {
...
bb:
  store i32 1, i32* @g
  call void externalCall() ; Volanie funkcie bez definície.
  ...
}
```

Obr. 6.43: Ukážka volania funkcie, ktorá nemá definíciu v module.

určiť, aká funkcia je volaná. Ako je vidieť na tejto ukážke, tak vo funkcii `funcWithCall` dochádza k nepriamemu volaniu funkcie. Funkcie, pre ktoré platí, že môžu byť potenciálne nepriamo volané sú funkcie `func1` a `funcVarArg`, pretože pre obidve funkcie platí, že môžu byť volané jedným argumentom, ktorý je predávaný do tejto funkcie. Funkcia `funcVarArg` predstavuje funkciu s premenlivým počtom parametrov. V tomto prípade je navrhnuté riešenie tohto problému také, že všetky pravé použitia uložené pred nepriamym volaním v množine PP sa uložia a na záver po prejdení všetkých funkcií sa opäť vyhodnotia spolu s poslednými ľavými použitiami uloženými vo funkciách, ktoré mohli byť nepriamo volané z toho miesta. V uvedenom príklade sa konkrétne uloží pravé použitie `%x = load i32* @g0` a to sa následne na záver po vyhodnotení všetkých funkcií priradí k ľavým použitiam, ktoré sa nachádzajú v potenciálne volaných nepriamych funkciách a to `func1` a `funcVarArg`, pretože toto pravé použitie môže nadobúdať hodnoty určené poslednými ľavými použitiami `store i32 1, i32* @g0` a `store i32 2, i32* @g0`. Zároveň je však potrebné v bloku, v ktorom dochádza k nepriamemu volaniu pridať do množiny PP všetky globálne premenné, pre ktoré optimalizujeme mŕtve priradenia. Je to z toho dôvodu, že nevieme určiť to, aké

pravé použitia v nepriamo volanej funkcii ostanú nenaviazané s nejakým priradením.

```
@g0 = global i32 5
@g1 = global i32 5

define i32 @func1(i32 %a) #0 {
bb:
  store i32 1, i32* @g0
  store i32 2, i32* @g1
  ret i32 0
}

define i32 @func2(i32 %a, i32 %b) #0 {
bb:
  store i32 3, i32* @g0
  ret i32 0
}

define i32 @funcVarArg(i32, ...) #0 {
bb:
  store i32 2, i32* @g0
  ret i32 0
}

define i32 @funcWithCall() #0 {
bb:
  ...
  %tmp2 = call i32 %tmp1(i32 %v1) ; Nepriame volanie.
  %x = load i32* @g0
  ret i32 0
}
```

Obr. 6.44: Ukážka nepriameho volania funkcie.

6.2.6 Princíp odstraňovania mŕtvych priradení

Výsledkom analýzy, ktorá prechádza celý kód LLVM IR a šíri si zmieňované informácie je to, že na záver táto analýza vráti množinu priradení a k ním prislúchajúce všetky pravé použitia, na ktoré majú dosah a platí, že priradzovaná hodnota do globálnej premennej môže byť použitá v tomto pravom použití. Pre ukážku 6.45 by bolo výsledkom analýzy taká informácia, že priradenie hodnoty 1 a hodnoty 2 do globálnej premennej `g` môže byť použité v načítaní hodnoty z tejto globálnej premennej v bloku `bb1`.

Zároveň vďaka tejto analýze získame priradenia, ktoré neovplyvňujú ani jedno pravé použitie, a takéto priradenia môžeme považovať za mŕtve priradenia, nakoľko priradzované hodnoty nie sú nikde použité. Vďaka tomu táto optimalizácia na záver odstráni tieto mŕtve priradenia a následne ešte prebehne odstraňovanie takých definícií globálnych premenných, pre ktoré platí, že táto definovaná globálna premenná v kóde neobsahuje žiadne použitie. Takáto situácia nastane v prípadoch, keď optimalizácia mŕtvych priradení odstránila všetky priradenia do takejto globálnej premennej a pre túto globálnu premennú platí, že to boli všetky jej použitia v kóde.

```
@g = global i32 5

define i32 func() {
bb:
  br i1 1, label %l, label %r

l:
  store i32 1, i32* @g
  br label %bb1

r:
  store i32 2, i32* @g
  br label %bb1

bb1:
  %x = load i32* @g
  ret i32 0
}
```

Obr. 6.45: Ukážka pre demonštráciu výsledkov analýzy, ktorá prechádza kód a šíri si potrebné informácie.

6.3 Konverzia globálnych premenných na lokálne

Úlohou tejto optimalizácie je konvertovať globálne premenné na lokálne všade tam, kde je to možné. Lokálne premenné sú vhodnejšie pre ďalšie optimalizácie, pretože skúmanie ich platnosti v kóde je jednoduchšie než v prípade globálnych premenných. Táto optimalizácia je dôležitá z toho dôvodu, pretože každý register, ktorý sa v binárnom kóde použije, je transformovaný na globálnu premennú. Tento spôsob riešenia je využívaný v spätnom prekladači vyvíjaného spoločnosťou AVG Technologies. Vďaka tomu môže vzniknúť veľký počet globálnych premenných, pretože registrov je veľké množstvo a dochádza k ich častému použitiu.

Rozhodnutie o tom, či globálna premenná môže byť vo funkcii konvertovaná na lokálnu nie je jednoduché. Je potrebné si uvedomiť, že pokiaľ dôjde k priradeniu nejakej hodnoty do globálnej premennej a táto hodnota je použitá mimo funkciu, v ktorej došlo k priradeniu tejto hodnoty, tak v takomto prípade nesmie dôjsť ku konverzii tejto globálnej premennej na lokálnu. Takto určené pravidlo o tom, či môže dôjsť ku konverzii alebo nie by však bolo zbytočne veľmi striktné. Pokiaľ dôjde k priradeniu hodnoty do globálnej premennej a všetky použitia tejto hodnoty sú v rámci tejto funkcie, čo je napríklad zabezpečené tak, že táto priradená hodnota je vždy prepísaná inou hodnotou pred jej použitím v iných funkciách, tak je možné aspoň čiastočne optimalizovať toto priradenie a jeho použitia prostredníctvom lokálnej premennej. Obrázok 6.46 zobrazuje kód pred aplikovaním optimalizácie a obrázok 6.47 obsahuje kód po aplikovaní optimalizácie. Tieto dva obrázky názorne vysvetľujú vyššie popísané. Na nich je vidieť, že nedošlo k zmene funkčnej ekvivalentnosti kódu. Na obrázku 6.47 je možné ešte ukázať práve prínos konverzie globálnej premennej na lokálnu. Po tejto konverzii je možné pomocou inej optimalizácie vykonať úpravu kódu tak, že hodnota 2, ktorá je priradená do definície lokálnej premennej `gLoc` sa použije rovno v definícii premennej `x`. Na záver je ešte potrebné vykonať odstránenie lokálnej premennej `gLoc`, nakoľko už nie je využívaná.

```
int g = 0;

void func() {
    g = 2;      // Konvertovateľné.
    int x = g; // Konvertovateľné.
    g = 4;      // Nekonvertovateľné.
    anotherFunc()
}

void anotherFunc() {
    x = g;      // Nekonvertovateľné.
}
```

Obr. 6.46: Ukážka kódu pred optimalizáciou konverzie globálnych premenných na lokálne.

```
int g = 0;

void func() {
    int gLoc = 2; // Optimalizované.
    int x = gLoc; // Optimalizované.
    g = 4;
    anotherFunc();
}

void anotherFunc() {
    int z = g;
}
```

Obr. 6.47: Ukážka kódu po optimalizácii konverzie globálnych premenných na lokálne.

Ako je možné na uvedených príkladoch vidieť, tak dochádza ku konverzii globálnej premennej použitej v priradení a súčasne dochádza ku konverzii v inštrukciách, kde je táto priradená hodnota použitá. Existuje však ešte špeciálny prípad, kedy neexistuje táto závislosť. Ukážka 6.48 ho znázorňuje. Ide o situáciu, kedy platí, že globálna premenná je použitá len v jednej funkcii. Vtedy nemusí platiť, že pred každým priradením hodnoty z globálnej premennej musí existovať priradenie tejto hodnoty do globálnej premennej. V tomto prípade je možné presunúť definíciu globálnej premennej rovno do funkcie a urobiť tak z tejto premennej lokálnu premennú.

```
int g = 1;

void func() {
    x = g;
}
```

Obr. 6.48: Ukážka obsahujúca kód, pri ktorom je globálna premenná použitá len v jednej funkcii.

Táto optimalizácia rieši ešte jeden špeciálny prípad, ktorý sa často vyskytuje v spätne preloženom kóde. Ide o situáciu uvedenú na ukážke 6.49. Na nej je možné vidieť takzvaný vzor. Vzor je tvorený pomocnou premennou, do ktorej je uschovaná hodnota globálnej

premennej. Následne je táto globálna premenná používaná a pri všetkých možných ukončeniach funkcie je uschovaná hodnota späť priradená do globálnej premennej. Tento vzor je častý z toho dôvodu, že globálna premenná v pôvodnom kóde vo veľa prípadoch predstavuje nejaký register. Preň je typické, že je často využívaný k rôznym účelom. Tento vzor potom predstavuje situáciu, kedy je jeho pôvodná hodnota uschovaná, register je využitý k iným účelom a následne je uschovaná hodnota späť do neho priradená. Uvedený vzor na ukážke 6.49 je možné konvertovať na kód, zobrazený na ukážke 6.50. Je vidieť, že došlo k odstráneniu vzoru a súčasne platí, že nedošlo k zmene funkčnej ekvivalentnosti kódu. Zároveň aby toto platilo je však potrebné konvertovať všetky použitia globálnej premennej, ktorá je súčasťou vzoru na použitia obsahujúce lokálnu premennú.

```
int g = 1;

void func() {
    int x;
    int tmp = g; // Začiatok vzoru.
    if (rand() < 5) {
        g = 2
        x = g;
        g = tmp; // Koniec vzoru.
        printf("%d", x);
        return;
    } else {
        g = 4;
        x = g;
        g = tmp; // Koniec vzoru.
        printf("%d", x);
        return;
    }
}
```

Obr. 6.49: Ukážka znázorňuje kód pred aplikovaním optimalizácie a obsahuje takzvaný vzor.

```
int g = 1;

void func() {
    int x;
    int gLoc;
    if (rand() < 5) {
        gLoc = 2
        x = gLoc;
        printf("%d", x);
        return;
    } else {
        gLoc = 4;
        x = gLoc;
        printf("%d", x);
        return;
    }
}
```

Obr. 6.50: Ukážka znázorňuje kód po aplikovaní optimalizácie na kóde z ukážky 6.49.

Aby bolo možné takto vyriešiť vzor je potrebné splniť nasledujúce podmienky:

1. Hodnota globálnej premennej je uschovaná do dočasnej lokálnej premennej a pri každom mieste, kde dôjde k ukončeniu funkcie je táto uschovaná hodnota priradená späť do globálnej premennej.

2. Dočasná premenná nie je nijako inak použitá, to znamená že je použitá len pre uschovanie hodnoty globálnej premennej a pri všetkých možných ukončeniach funkcie je uschovaná hodnota pomocou nej priradená späť do globálnej premennej.

3. Pre globálnu premennou, ktorá tvorí vzor platí, že všetky použitia tejto globálnej premennej vo vnútri vzoru sú nahraditeľné lokálnou premennou.

Táto optimalizácia je rovnako jednou z tých, ktoré boli navrhnuté za účelom presunu optimalizácií zo zadnej časti do strednej časti spätného prekladača. Aby mohlo dôjsť k tomuto presunu, bolo potrebné navrhnúť túto optimalizáciu tak, aby pokrývala celú funkčnosť optimalizácie globálnych premenných na lokálne, ktorá sa nachádza v zadnej časti spätného prekladača. Pri tomto presune ostal priestor pre vylepšenie tejto optimalizácie, nakoľko tá v zadnej časti konvertovala globálnu premennú na lokálnu len v prípade, že všetky použitia tejto globálnej premennej boli nahraditeľné lokálnou premennou v jednej funkcii. Novo navrhnutá optimalizácia umožňuje aj čiastočné optimalizovanie. Je to umožnené tým, že je vždy konvertované priradenie do globálnej premennej spolu s konverziou v inštrukciách, kde je táto priradená hodnota použitá.

Táto optimalizácia využíva rovnakú analýzu, ktorá je využívaná v optimalizácii odstraňujúcej mŕtve priradenia do globálnych premenných. Vďaka tomu je možné túto analýzu, ktorá je najpomalšou časťou týchto dvoch optimalizácií spustiť iba raz nad celým kódom a následne vykonať obidve optimalizácie.

V nasledujúcich oddieloch budú popísané len rozdiely medzi týmito dvomi optimalizáciami, keďže využívajú rovnakú analýzu a princípy, ktoré už boli popísane. Oddiel 6.3.1 popisuje syntax a sémantiku inštrukcií súvisiacich s touto optimalizáciou. Oddiel 6.3.2 popisuje nový špecifický prípad, kedy nesmie dôjsť ku konvertovaniu globálnej premennej na lokálnu. V optimalizácii mŕtvych priradení sa nachádzali oddiely venujúce sa priechodu základnými blokmi a funkciami a boli v samostatnom oddiele popísane informácie, ktoré sú šírené a uchovávané počas analýzy. Uvedené informácie v týchto oddieloch rovnako platia aj pre túto optimalizáciu, a tak nebudú redundantne uvádzané. Oddiel 6.3.3 sa venuje princípu konverzie globálnej premennej na lokálnu premennú.

6.3.1 Syntax a sémantika inštrukcií súvisiacich s konverziou globálnych premenných na lokálne premenné

Rovnako ako pri optimalizácii mŕtvych priradení súvisia s touto optimalizáciou inštrukcie `store` a `load`. Optimalizácia globálnych premenných na lokálne premenné využíva ešte jednu inú inštrukciu navyše. Táto inštrukcia sa využíva pre vytvorenie novej lokálnej premennej. Ide o takzvanú `alloca` inštrukciu. Jej syntax je uvedená na ukážke 6.51. Táto inštrukcia pridelí miesto v pamäti pre novú premennú a toto miesto je uvoľnené po ukončení funkcie, v ktorej sa táto premenná nachádza.

```
<result> = alloca [inalloca] <type> [, <ty> <NumElements>]
           [, align <alignment>]
```

Obr. 6.51: Syntax inštrukcie `alloca` [23, viď sekcia alloca Instruction].

Najdôležitejšie argumenty tejto inštrukcie pre túto optimalizáciu sú `<type>` a `<result>`. Argument `<type>` predstavuje typ novej premennej a `<result>` predstavuje novo vytvorenú premennú [23, viď sekcia alloca Instruction].

6.3.2 Špecifické prípady, pri ktorých nedochádza k optimalizácii

Špecifické prípady uvedené pre optimalizáciu odstraňujúcu mŕtve priradenia rovnako platia aj pre túto optimalizáciu. Vzhľadom na to, že dochádza k náhrade globálnej premennej za lokálnu v inštrukcii `load`, tak pokiaľ je táto inštrukcia označená ako `volatile`, tak nesmie dôjsť k optimalizovaniu tejto globálnej premennej pre nejaké priradenie a jeho pravé použitia.

6.3.3 Princíp konvertovania globálnej premennej na lokálnu premennú

Ako už bolo uvedené pri optimalizácii mŕtvych priradení, tak výsledkom analýzy prechádzajúcej celý kód LLVM IR a šíriacej si zmieňované informácie je to, že na záver táto analýza vráti množinu priradení a k ním prislúchajúce všetky pravé použitia, na ktoré majú dosah a platí, že priradzovaná hodnota do globálnej premennej môže byť použitá v tomto pravom použití. Na základe týchto informácií je možné rozhodnúť, či toto priradenie do globálnej premennej spolu s pravými použitiami je možné nahradiť pomocou lokálnej premennej. Musí platiť, že všetky tieto pravé použitia sa nachádzajú v rovnakej funkcii ako toto priradenie a súčasne ani jedno z týchto pravých priradení nie je dosiahnuteľné z nejakého iného priradenia nachádzajúceho sa mimo túto funkciu.

Následne po uvedenej konverzii sú rozpoznané vzory vyriešené popísaným spôsobom v úvode tejto podkapitoly. Na záver sa skontrolujú všetky globálne premenné a ich použitia a pokiaľ platí, že všetky použitia sa nachádzajú len v jednej funkcii, tak dôjde ku konverzii globálnej premennej na lokálnu popísaným spôsobom uvedeným v úvode tejto podkapitoly. Pokiaľ dôjde k situácii, že všetky použitia globálnej premennej boli konvertované na lokálne, tak dôjde k odstráneniu definície takejto globálnej premennej.

6.4 Odstránenie nedosiahnuteľných funkcií

Hlavnou motiváciou tejto optimalizácie je odstrániť funkcie, ktoré nie sú nikdy volané. Takéto funkcie neovplyvňujú beh programu, a preto môže dôjsť k ich odstráneniu. Odstránením nedosiahnuteľných funkcií dôjde k zmenšeniu kódu LLVM IR, ktorý je potrebný analyzovať inými časťami spätného prekladača, a preto dôjde k jeho urýchleniu. Ako príklad ukážka 6.52 obsahuje nedosiahnuteľnú funkciu `func2`, pretože pre ňu platí, že nedôjde nikdy k jej volaniu.

Syntax a sémantiku inštrukcie úzko súvisiacej s touto optimalizáciou popisuje oddiel 6.4.1. Oddiel 6.4.2 obsahuje popísané špecifické prípady, pri ktorých nedochádza k optimalizova-

```
void func1() { // Nevhodná k odstráneniu.
    ...        // Nevolá func2.
}

void func2() { // Vhodná k odstráneniu.
    ...
}

int main() {
    func1();
}
```

Obr. 6.52: Ukážka obsahujúca nedosiahnuteľnú funkciu `func2`.

niu nedosiahnuteľných funkcií. Oddiel 6.4.3 sa venuje spôsobu určenia nedosiahnuteľných funkcií.

6.4.1 Syntax a sémantika inštrukcie súvisiacej s odstraňovaním nedosiahnuteľných funkcií

Pre zavolanie funkcie sa v LLVM využíva inštrukcia `call`, ktorej syntax je uvedená na ukážke 6.53.

```
<result> = [tail | musttail] call [cconv] [ret attrs] <ty> [<fnty>*]
           <fnptrval>(<function args>) [fn attrs]
```

Obr. 6.53: Syntax inštrukcie `call` [23, viď sekcia call Instruction].

Argumenty inštrukcie `call`, ktoré majú význam pre túto optimalizáciu sú `<ty>`, `<fnty>*` a `<function args>`. Argument `<ty>` určuje návratový typ funkcie. Argument `<fnty>*` musí byť signatúra ukazovateľa na funkciu, cez ktorú je volaná. Môže ísť buď o meno funkcie, alebo premennú obsahujúcu adresu volanej funkcie. Zoznam všetkých predávaných argumentov do volanej funkcie predstavuje argument `<function args>`. Volanie funkcie `func` s návratovým typom `i32` a dvoma predávanými argumentami typu `i32` je zobrazené na ukážke 6.54

```
call i32 @func(i32 1, i32 2)
```

Obr. 6.54: Ukážka volania funkcie v LLVM IR.

6.4.2 Špecifické prípady, pri ktorých nedochádza k optimalizácii

S touto optimalizáciou existujú obmedzenia, kedy nie je možné nedosiahnuteľné funkcie odstrániť. Ide o nasledujúce situácie:

- Pokiaľ nie je známy vstupný bod programu, nie je možné odstrániť žiadnu funkciu, pretože nevieme určiť, či práve táto funkcia nie je vstupným bodom programu. Napríklad pre jazyk C je vstupný bod programu funkcia `main`.

- Odstraňované sú len funkcie, ktoré majú definíciu. Je to z toho dôvodu, že odstránením funkcie, ktorá má iba deklaráciu by nedošlo k žiadnemu urýchleniu iných častí spätného prekladača.
- Adresa funkcie je uložená napríklad do globálnej premennej, položky štruktúry a pod. Nemusí však platiť, že pomocou tejto adresy je realizované nepriame volanie tejto funkcie. V prípade jej odstránenia by nebolo možné zistiť a uložiť si jej adresu.

6.4.3 Princíp identifikácie nedosiahnuteľných funkcií

Ako už bolo spomínané, tak prvým dôležitým krokom je určiť vstupný bod programu. V prípade jazyka C a modifikovaného jazyka Python využívaného v spätnom prekladači vyvíjaného spoločnosťou AVG Technologies je týmto vstupným bodom programu funkcia `main`. Identifikácia funkcie `main` je realizovaná prostredníctvom porovnávania mien funkcií na toto meno. Následne pokiaľ je nájdená funkcia `main`, tak môže dôjsť k vyhľadávaniu všetkých dosiahnuteľných funkcií. Vyhľadávanie je realizované prostredníctvom grafu volania funkcií dostupného z frameworku LLVM. Vďaka tomu je možné určiť všetky funkcie, ktoré sú volané priamo z funkcie `main`, prípadne z funkcií volaných z funkcie `main`. Funkcie však môžu byť volané aj nepriamo prostredníctvom ukazovateľa. Preto je nutné identifikovať všetky nepriame volania funkcií vo funkciách volaných priamo z funkcie `main`. Na základe týchto rozpoznaných nepriamych volaní je potrebné určiť, ktorá množina funkcií môže byť prostredníctvom nejakého konkrétneho nepriameho volania volaná. Toto určenie prebieha na základe porovnávania typov návratových hodnôt funkcií, počtu a typu parametrov funkcií. Takéto riešenie by však stále nebolo dostačujúce. Z volaných funkcií nepriamo môže opäť dochádzať k volaniu ďalších funkcií priamo alebo nepriamo, a preto je potrebné pokračovať v určovaní dosiahnuteľných funkcií až do doby, pokiaľ sú objavované nové funkcie. Na záver môže dôjsť k odstráneniu všetkých definovaných nedosiahnuteľných funkcií, ktorých adresa nie je ukladaná napríklad do globálnej premennej a podobne.

6.5 Odstránenie kódu za volaniami funkcií, ktoré sa nevracajú

V jazyku C existuje niekoľko funkcií, pre ktoré platí, že po ich zavolaní nedôjde k ich návratu [43]. Ide o nasledujúce funkcie:

- `void exit(int status)`
- `void abort()`
- `void longjmp(jmp_buf env, int val)`
- `void _Exit(int status)`
- `void quick_exit(int status)`
- `void thrd_exit(int res)`

Úlohou optimalizácie je identifikovať volanie takýchto funkcií a zabezpečiť odstránenie nedosiahnuteľného kódu za týmito volaniami. Vďaka tomu dôjde k redukcii množstva kódu potrebného k spracovaniu inými časťami spätného prekladača. Nasledujúca ukážka 6.55 obsahuje volanie takejto nevracajúcej sa funkcie. Kód za týmto volaním môže byť odstránený, nakoľko nikdy nedôjde k jeho volaniu.

```
int g = 4;

int main() {
    exit();
    printf("%d", g); // Môže byť odstránené.
}
```

Obr. 6.55: Ukážka obsahujúca volanie funkcia, ktorá sa nevracia.

Syntax a sémantiku inštrukcií úzko súvisiacich s touto optimalizáciou popisuje oddiel 6.5.1. Oddiel 6.5.2 sa venuje spôsobu určenia funkcií, ktoré sa nevracajú. Popis spôsobu zabezpečenia odstránenia nedosiahnuteľného kódu za volaniami funkcií, ktoré sa nevracajú, sa nachádza v oddiele 6.5.3.

6.5.1 Syntax a sémantika inštrukcií súvisiacich s touto optimalizáciou

S touto optimalizáciou súvisí inštrukcia `call`, ktorá už bola popísaná v oddiele 6.4.1. Okrem zmienenej inštrukcie je využívaná inštrukcia `unreachable`. Jej syntax sa nachádza na ukážke 6.56.

```
unreachable
```

Obr. 6.56: Syntax inštrukcie `unreachable` [23, viď sekcia unreachable Instruction].

Inštrukcia `unreachable` sa používa pre informovanie optimalizácií, že kód nachádzajúci sa za ňou je nedosiahnuteľný a nikdy nedôjde k jeho vykonaniu [23].

6.5.2 Princíp identifikácie volania funkcie, ktorá sa nevracia

K určeniu, že volanie funkcie odpovedá volaniu funkcie, pri ktorej nedôjde k návratu funkcie je potrebné zohľadniť nasledujúce skutočnosti. Meno volanej funkcie zodpovedá jednému z mien funkcií, pri ktorých nedôjde k ich návratu. Okrem toho je však potrebné dbať aj na návratový typ volanej funkcie, jej parametre ako aj typy týchto parametrov. Všetky uvedené vlastnosti musia odpovedať nejakej funkcii, pre ktorú platí, že nedôjde k jej návratu. Nemenej dôležitým faktorom je, že pre takúto funkciu sa v spätne preloženom kóde nachádza len jej deklarácia a definícia chýba. Pokiaľ by funkcia obsahovala definíciu, tak by išlo o užívateľsky definovanú funkciu a pri takejto funkcii štandardne dochádza k jej návratu.

6.5.3 Spôsob zabezpečenia odstránenia kódu za volaniami funkcií, ktoré sa nevracajú

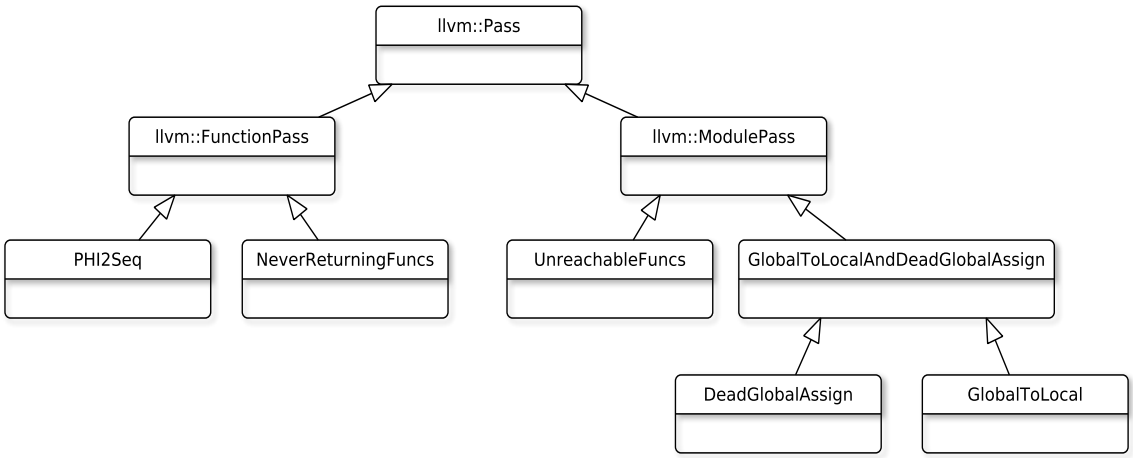
Hlavnou úlohou optimalizácie ako už bolo spomínané je zabezpečiť, že dôjde k odstráneniu nedosiahnuteľného kódu nachádzajúceho sa za volaniami funkcií, ktoré sa nevrátia. Toto je možné realizovať umiestnením LLVM IR inštrukcie `unreachable` za takéto volanie. Následne po umiestnení tejto inštrukcie dôjde spustením ďalších optimalizácií z frameworku LLVM k odstráneniu nedosiahnuteľného kódu a vďaka tomu dôjde k zmenšeniu počtu inštrukcií, ktoré je nutné analyzovať ostatnými časťami spätného prekladača, a tak dôjde k jeho zrýchleniu.

Kapitola 7

Implementácia optimalizácií

Táto kapitola sa venuje popisu implementácie jednotlivých optimalizácií. Každá jedna podkapitola obsahuje popis práve jednej optimalizácie. Všetky optimalizácie sú implementované v jazyku C++ podľa štandardu ISO C++11 [1].

Optimalizácie navrhnuté v tejto práci rovnako ako tie vstavané dedia od tried, ktoré poskytujú rozhranie k modifikovaniu určitých časti kódu. Napríklad trieda `llvm::FunctionPass` je určená ako otcovská trieda pre optimalizácie, ktorých úlohou je meniť telá funkcií. Z tejto triedy dedí optimalizácia usporiadania PHI uzlov (`PHI2Seq`) a optimalizácia odstraňujúca kód za volaniami funkcií, ktoré sa nevracajú (`NeverReturningFuncs`). Otcovská trieda pre optimalizácie modifikujúce modul je `llvm::ModulePass`. Práve takouto je optimalizácia odstraňujúca nedosiahnuteľné funkcie (`UnreachableFuncs`). Pre optimalizáciu odstraňujúcu mŕtve priradenia (`DeadGlobalAssign`) a optimalizáciu konvertujúcu globálne premenné na lokálne (`GlobalToLocal`) bola vytvorená spoločná bázová trieda `GlobalToLocalAndDeadGlobalAssign`. Táto bázová trieda je potrebná z toho dôvodu, aby analýza, ktorá je spoločná pre tieto dve optimalizácie mohla byť spustená iba raz. Obidve uvedené optimalizácie upravujú časti kódu aj mimo funkciu, a preto ich bázová trieda dedí z `llvm::ModulePass`. Bázovou triedou pre všetky optimalizácie je trieda `llvm::Pass` [29]. UML diagram na obrázku 7.1 znázorňuje tieto uvedené vzťahy dedičnosti.



Obr. 7.1: UML diagram tried, zobrazujúci hierarchiu dedičností navrhnutých optimalizácií.

Ako už bolo uvedené, tak jednotlivé optimalizácie dedia z tried LLVM podľa toho, akú časť kódu modifikujú. Tieto triedy poskytujú rozhranie pre toto modifikovanie. Napríklad trieda `llvm::FunctionPass` obsahuje dve virtuálne metódy a jednu čisto virtuálnu metódu, ktoré sú určené k týmto účelom. Metóda `runOnFunction()` je uvedená čisto virtuálna metóda. Z nej je možné vykonávať všetku potrebnú prácu optimalizácie nad konkrétnou funkciou. Jej návratová hodnota určuje, či došlo k zmene LLVM IR kódu. Virtuálna metóda `doInitialization()` je určená k vykonaniu práce pred začatím navštevovania funkcií pomocou metódy `runOnFunction()`. Poslednou virtuálnou metódou je `doFinalization()`. V nej je možné vykonať prácu po navštívení všetkých funkcií. Podobný princíp funguje napríklad aj v prípade dedenia z triedy `llvm::LoopPass`. Účelom tejto triedy je poskytnúť rozhranie pre optimalizácie určené na optimalizovanie cyklov. Čisto virtuálna metóda v tejto triede má samozrejme odlišný názov, virtuálne metódy sú opäť dve a majú rovnaký názov. Líšia sa len v počte a type parametrov. Princíp využitia je však zachovaný a platí i pre iné triedy, ktoré slúžia pre poskytnutie rozhrania optimalizáciám. Trieda `llvm::ModulePass` obsahuje len čisto virtuálnu metódu `runOnModule()`. Pomocou nej je možné upravovať všetko, čo je obsiahnuté v module [29].

Trieda `llvm::Pass` obsahuje virtuálnu metódu `getAnalysisUsage()`, ktorú je možné predefinovať v optimalizáciách. Týmto predefinovaním je v nej možné zabezpečiť spustenie analýz z frameworku LLVM pred samotným vykonaním optimalizácie a následne využívať výsledky týchto analýz.

V navrhnutých a implementovaných optimalizáciách sú využívané rôzne kontajnery, ako napríklad `std::map`, `std::vector` a `std::set` a podobne. Tieto kontajnery sú vyberané v implementáciách tak, aby boli čo najvhodnejšie využiteľné pre potrebné účely.

Celý spätný prekladač je prekladaný pomocou CMake, ktorý predstavuje multiplatformový softvér pre automatizáciu prekladu programov [8]. Preklad je možný pomocou GCC [12] alebo Clang [22].

Spätný prekladač je vyvíjaný na operačnom systéme Linux a Windows. Na obidvoch operačných systémoch je ho možné využiť k spätnému prekladu.

Umiestnenie optimalizácií, analýz navrhnutých v tejto práci a funkcionality, ktorá je používaná viacerými optimalizáciami je nasledujúce:

```
decompiler
└─ src
   └─ opt
      ├─ analyses ..................................... Analýzy.
      ├─ utilities ................................ Zdieľaná funkcionalita.
      └─ optimizations ........................... Optimalizácie.
         ├─ phi2seq
         ├─ unreachable_funcs
         ├─ func_never_returns
         └─ globals
```

Adresár `globals` obsahuje optimalizáciu odstraňujúcu mŕtve priradenia a optimalizáciu konvertujúcu globálne premenné na lokálne premenné.

7.1 Usporiadanie PHI uzlov

Optimalizácia je implementovaná pomocou triedy `PHI2Seq`. Keďže upravuje základné bloky vo funkcii, tak dedí z triedy `llvm::FunctionPass`. Je nutné, aby táto trieda implementovala čisto virtuálnu metódu `runOnFunction()`. Optimalizácia sa nachádza vo vlastnom priestore mien a to konkrétne `phi2seq`.

Využívané analýzy

V optimalizácii je predefinovaná virtuálna metóda `getAnalysisUsage()`, vďaka čomu je možné využiť v tejto optimalizácii analýzu s názvom `llvm::InstructionNamerID`. Jej úlohou je pomenovať dočasnými názvami všetky premenné, ktoré meno nemajú, pretože v kóde LLVM IR sa môžu nachádzať takzvané nepomenované premenné. Toto pomenovávanie premenných je potrebné z toho dôvodu, aby bola zachovaná rovnaká funkčnosť pri ľubovoľnom spustení tejto optimalizácie. Pokiaľ by nebolo realizované, ale premenné by boli uchovávané prostredníctvom ich adries a nie mien, tak by mohlo dochádzať k situácii, že pri usporadovaniu inštrukcií `phi` do správneho poradia by došlo k odlišnému usporadovaniu pri rôznych spusteniach, keďže adresy premenných sa menia pri každom spustení. Samozrejme k tejto zmene usporiadania môže dôjsť len pri inštrukciách `phi`, ktoré nemajú medzi sebou žiadnu závislosť a nie je potrebné, aby bola jedna vyhodnotená pred druhou.

Optimalizácia využíva ešte jednu analýzu, ktorá bola implementovaná v rámci tejto diplomovej práce a je reprezentovaná triedou `VarDependAnalysis`. Pomocou tejto analýzy je možné vytvoriť graf závislostí medzi premennými. Tento graf je realizovaný prostredníctvom `std::map`, kde kľúč je názov premennej a hodnotou je objekt triedy `Node`, ktorý reprezentuje uzol grafu. Táto trieda je súčasťou analýzy `VarDependAnalysis`. Objekt triedy `Node` obsahuje všetkých následníkov tohto uzla, ako aj základné informácie či už bol spracovaný, navštívený a pod. Po vytvorení grafu je možné nad ním spustiť detekciu cyklických závislostí, ktorej výstupom je množina `phi` inštrukcií vhodná k osamostatneniu sa do samostatného bloku, čím dôjde k eliminovaniu tejto cyklickej závislosti a vyriešenie problému paralelného vyhodnocovania cyklicky závislých premenných v inštrukciách `phi`. Cyklické závislosti ako už bolo uvádzané v návrhu sú vyhľadávané prostredníctvom rekurzívneho priechodu DFS. Tento priechod je implementovaný v metóde `visitNodeCycleDetect()`. Metóda `detectCycleVarDependency()` realizuje zmienenú funkcionalitu detekcie cyklických závislostí a zabezpečuje získanie inštrukcií vhodných k osamoteniu do samostatného bloku.

Pokiaľ zostavený graf v rámci tejto analýzy už neobsahuje žiadnu cyklickú závislosť, tak je možné spustiť druhú časť tejto analýzy, ktorej úlohou je nájsť vhodné usporiadanie inštrukcií `phi` a toto zoradenie je vrátené ako výsledok. Opäť je k účelu zoradeniu inštrukcií využívaný rekurzívny priechod DFS, ktorého implementácie sa nachádza v metóde `visitNodeNonCycleVarDependency()`. Toto zoradenie a získanie výsledku je možné realizovať pomocou metódy `detectNonCycleVarDependency()`.

Táto optimalizácia na základe priechodu inštrukcií `phi` zostaví graf závislostí premenných v analýze `VarDependAnalysis`, následne spustí prostredníctvom tejto analýzy vyhodnotenie riešiace cyklické závislosti. Výsledky spracuje a upraví potrebné inštrukcie a bloky, aby došlo k odstráneniu týchto cyklických závislostí. Potom spustí druhú fázu analýzy, vďaka ktorej získa potrebné zoradenie inštrukcií `phi` a toto zoradenie vykoná.

Úpravy LLVM IR kódu

Pridávanie inštrukcií do nových základných blokov, ktoré vznikajú pri tejto optimalizácií je riešené pomocou `llvm::IRBuilder`. Je poskytovaný frameworkom LLVM a je určený práve k budovaniu obsahu základných blokov pomocou inštrukcií.

V tejto optimalizácii dochádza k upravovaniu inštrukcie `phi`, ktorá je reprezentovaná triedou `llvm::PHINode`. Táto úprava zahŕňa zmenu premennej, ktorá spôsobuje cyklickú závislosť medzi premennými a pre túto premennú aj zmenu predchádzajúceho bloku, pri ktorom je táto premenná použitá. Dôvody tejto zmeny sú popísane v oddiele 6.1.2. Okrem tejto inštrukcie dochádza k upravovaniu takzvaných ukončujúcich inštrukcií. Ako príklad je možné uviesť napríklad inštrukciu skoku.

K vytváraniu a napojeniu novo vzniknutých základných blokov je využívaná statická metóda `Create()`, ktorú poskytuje trieda `llvm::BasicBlock`.

7.2 Odstránenie mŕtvych priradení do globálnych premenných

Optimalizácia je implementovaná pomocou triedy `DeadGlobalAssign`. Hlavná funkcionalita sa však nachádza v triede `GlobalToLocalAndDeadGlobalAssign`, z ktorej táto optimalizácia dedí. Tento spôsob je potrebný z toho dôvodu, že LLVM framework neumožňuje spustenie jednej analýzy pre dve optimalizácie súčasne. Ako už bolo uvádzané v návrhu, tak táto optimalizácie využíva analýzu zbierajúcu potrebné informácie z kódu LLVM IR súčasne s optimalizáciou konvertujúcou globálne premenné na lokálne. V triede `DeadGlobalAssign` dôjde len k nastaveniu potrebných informácií pre bázovú triedu a tá následne vykoná danú optimalizáciu. Zároveň je týmto konceptom možné spúšťať len jednu z uvedených dvoch optimalizácií a druhú ponechať vypnutú. Optimalizácia sa nachádza spoločne s optimalizáciou konvertujúcou globálne premenne na lokálne vo vlastnom priestore mien a to konkrétne `globals`.

Trieda `GlobalToLocalAndDeadGlobalAssign` dedí z `llvm::ModulePass`, pretože môže dochádzať k odstraňovaniu definícií globálnych premenných. Zároveň je nutné, aby táto trieda implementovala čisto virtuálnu metódu `runOnModule()`.

Odstránenie mŕtvych priradení do globálnych premenných je realizované prostredníctvom metódy `removeDeadGlobalAssigns()`. Globálne premenné bez použitia sú odstraňované prostredníctvom metódy `removeGlobsWithoutUse()`. Obidve uvedené metódy sa nachádzajú v triede `GlobalToLocalAndDeadGlobalAssign`.

Využívané analýzy

Optimalizácia využíva dve analýzy, ktoré boli implementované v tejto diplomovej práci. Prvú predstavuje trieda `UsesAnalysis`. Úlohou tejto analýzy je poskytovať základné informácie o použitiach napríklad globálnych premenných v LLVM IR kóde. Táto optimalizácia ju využíva pre určenie toho, či ešte existujú použitia globálnych premenných v kóde a na základe toho je možné odstrániť definíciu globálnej premennej.

Trieda `StoreLoadAnalysis` predstavuje hlavnú analýzu pre túto optimalizáciu. Jej úlo-

hou je prechádzať kód a šíriť si potrebné informácie popísané v návrhu tejto optimalizácie a následne poskytovať podporu pre rozhodovanie o prípadnom optimalizovaní mŕtveho priradenia. Spustenie analýzy je realizované prostredníctvom metódy `doAnalysis()`. Analýza sprostredkováva získane výsledky pomocou vlastných prístupových metód nad kontajnermi, ktoré ich uchovávajú. Napríklad metódy `rUsesForLUse_begin()` a `rUsesForLUse_end()` vracajú iterátor do `std::map`, kde kľúčom je globálne priradenie do premennej a hodnotou je `std::set` inštrukcií, v ktorých dochádza k použitiu hodnoty z tohto globálneho priradenia. Pre pravé použitia popísané v návrhu sa jedná o metódy `extRUses_begin()` a `extRUses_end()`. Tento koncept je zachovaný aj pri ďalších získavaných informáciách. Aby nedochádzalo k zbytočnému veľkému prehľadávaniu získaných informácií pri dotaze na nejakú konkrétnu informáciu, tak tieto informácie sú mapované na základe funkcie, s ktorou majú súvislosť. Zároveň sú mapované aj ku konkrétnemu základnému bloku, ku ktorému sa vzťahujú. Vzhľadom na veľkú zložitosť tejto analýzy bola vytvorená trieda `AnalysisInfo`, ktorou úlohou je uchovávať získané informácie a poskytovať k ním potrebné prístupové metódy. Na základe toho je možné logicky oddeliť dátovú časť tejto analýzy od funkčnej.

Analýza `StoreLoadAnalysis` využíva niekoľko ďalších analýz implementovaných v tejto diplomovej práci. Napríklad ide o také, ktorých úlohou je zabezpečiť správne poradie vyhodnocovania funkcií a základných blokov. Pre zaistenie správneho poradia vyhodnocovania funkcií bola implementovaná analýza nachádzajúca sa v triede `FuncTraversalAnalysis`. Trieda `BBTraversalAnalysis` obsahuje analýzu, ktorá zabezpečuje správne poradie vyhodnocovania blokov. Bázovou triedou pre uvedené dve analýzy je trieda `TraversalAnalysis`. Táto trieda obsahuje spoločnú funkcionalita pre uvedené dve analýzy. Tieto dve analýzy poskytujú rozhranie a nadstavbu nad analýzou silne súvislých komponent, ktorá sa nachádza v LLVM frameworku. Vhodnou úpravou uvedenej analýzy je možné zabezpečiť potrebný priechod *post-order* nad základnými blokmi a funkciami. Rozhranie v prípade analýz `BBTraversalAnalysis` a `FuncTraversalAnalysis` je realizované formou dotaz-odpoveď, kedy dochádza k vyžiadaniu nasledujúceho bloku/funkcie, ktorý má byť analyzovaný. Táto funkcionalita je zabezpečená pomocou metód `getNextBB()` v prípade základných blokov a v prípade funkcií ide o `getNextFunc()`. Okrem týchto dvoch uvedených metód sa v týchto analýzach nachádzajú aj také, ktoré umožňujú overiť, či ďalší blok/funkcia je súčasťou silne súvislej komponenty a prípadne podľa potreby iterovať cez túto silne súvislú komponentu do doby, kým je požiadané o ukončenie tejto iterácie. Vďaka tomu je možné prechádzať silne súvislú komponentu pokiaľ v nej dochádza k úpravám zbieraných informácií nad kódom LLVM IR. Význam nutnosti opätovného vyhodnocovania bol popísaný v návrhu optimalizácie. Pred využitím týchto dvoch analýz je ale potrebné spustiť ich vlastné vyhodnotenie, aby dokázali vhodne poskytovať príslušné informácie. Toto je možné realizovať prostredníctvom metód `doBBAnalysis()` v prípade základných blokov a pre prípad funkcií ide o metódu `doFuncsAnalysis()`.

Analýza `IndirectlyCalledFuncsAnalysis` je taktiež využívanou v `StoreLoadAnalysis`. Jej úlohou je zistiť všetky možné funkcie, ktoré sú volané prostredníctvom nepriameho volania. Metóda `getFuncsForIndirectCalls()` nachádzajúca sa v tejto analýze vracia funkcie, ktoré môžu byť nepriamo volané pomocou volania, ktoré je predané v parametre tejto metódy.

Analýza `StoreLoadAnalysis` prechádza celý kód LLVM IR, na základe čoho si získava a uchováva potrebné informácie a po jej vykonaní poskytuje potrebné informácie optimalizácii odstraňujúcej mŕtve priradenia.

Úpravy LLVM IR kódu

Odstraňovanie mŕtvych priradení a definícií globálnych premenných je realizované prostredníctvom metód, ktoré poskytuje framework LLVM.

7.3 Konverzia globálnych premenných na lokálne

Ako už bolo uvádzané v návrhu, tak veľká časť funkcionality je spoločná s analýzou odstraňujúcou mŕtve priradenia. Z tohto dôvodu sa nasledujúca časť textu bude venovať len rozdielnostiam, pretože uvedené informácie v podkapitole 7.2 platia aj pre túto optimalizáciu. Optimalizácia je realizovaná prostredníctvom triedy `GlobalToLocal` a tiež dedí z triedy `GlobalToLocalAndDeadGlobalAssign`.

Pred začatím konverzie globálnych premenných na lokálne je potrebné určiť, v ktorých priradeniach a pre nich patriacich dosiahnuteľných použití priradzovanej hodnoty nesmie dôjsť k náhrade globálnej premennej za lokálnu. Metóda `filterUsesThatCannotBeOptimized()` zabezpečuje popísanú funkcionalitu.

Konverzia globálnych premenných, ktoré sa využívajú len v jednej funkcii sa nachádza v metóde `convertGlobsToLocUseInOneFunc()`. Konverzia globálnych premenných na lokálne pre ostatné prípady je implementovaná v metóde `convertGlobsToLocs()`. Riešenie vzorov je realizované prostredníctvom metódy `removePatternInsts()`. O odstránenie globálnych premenných nemajúcich žiadne použitia sa stará metóda `removeGlobsWithoutUse()`. Všetky tieto uvedené metódy sa nachádzajú v triede `GlobalToLocalAndDeadGlobalAssign`.

Využívané analýzy

Analýzy využívané v tejto optimalizácii sú rovnaké ako uvedené v podkapitole 7.2.

Úpravy LLVM IR kódu

Pridávanie novej lokálnej premennej je uskutočnené pomocou konštruktoru vytvárajúceho objekt triedy `AllocaInst`. Pre konvertovanie globálnej premennej, ktorá sa používa len v jednej funkcii je potrebné ešte využiť inštrukciu `store`, vďaka ktorej sa priradí hodnota z definície globálnej premennej do novo vytvorenej lokálnej premenne. Vytvorenie tejto inštrukcie je realizované prostredníctvom konštruktora nachádzajúceho sa v triede `StoreInst`.

7.4 Odstránenie nedosiahnuteľných funkcií

Táto optimalizácia je implementovaná pomocou triedy `UnreachableFuncs`. Keďže upravuje modul, tak dedí z triedy `llvm::ModulePass`. Je nutné, aby táto trieda implementovala čisto virtuálnu metódu `runOnModule()`. Optimalizácia sa nachádza vo vlastnom priestore mien a to konkrétne `unreachable_funcs`.

V triede `UnreachableFuncs` dochádza ku kontrole toho, či v module je prítomná funkcia s názvom `main`. Následne sú získané všetky dosiahnuteľné funkcie. Potom sú určené funkcie, ktoré nemôžu byť optimalizované, pretože ich adresa sa ukladá napríklad do globálnej premennej. Na záver sú všetky ostatné funkcie odstránené. K určeniu dosiahnuteľných funkcií sa využíva analýza, ktorá bola implementovaná v tejto diplomovej práci a je popísaná nižšie.

Využívané analýzy

Táto optimalizácia využíva jednu analýzu. Konkrétne ide o analýzu, ktorá je implementovaná pomocou triedy `ReachableFuncsAnalysis`. Vďaka nej je možné získať všetky dosiahnuteľné definované funkcie, ktoré môžu byť volané priamo aj nepriamo (prostredníctvom ukazovateľa) z nejakej funkcie. Získanie týchto dosiahnuteľných definovaných funkcií je možné realizovať prostredníctvom statickej metódy `getReachableDefinedFuncsFor()`. Jej návratovou hodnotou je `std::set` definovaných dosiahnuteľných funkcií z funkcie, ktorá je predaná parametrom do tejto metódy. Táto analýza však ešte využíva ďalšie dve analýzy, vďaka ktorým je možné získať všetky dosiahnuteľné funkcie. Obidve tieto analýzy boli rovnako implementované v rámci tejto diplomovej práce. Prvá analýza umožňuje riešiť problematiku nepriamych volaní funkcií. Trieda `IndirectlyCalledFuncsAnalysis` reprezentuje túto analýzu. Pomocou nej je možné získať všetky funkcie, ktoré môžu byť volané nepriamo prostredníctvom nepriamych volaní. Túto funkcionalitu je možné dosiahnuť volaním statickej metódy `getFuncsForIndirectCalls()`. Jej návratovou hodnotou je `std::set` funkcií, ktoré môžu byť pre nepriame volanie predané parametrom volané prostredníctvom ukazovateľa. Druhá analýza, ktorú predstavuje trieda `IndirectCallsFinder`, je umiestnená v anonymnom priestore mien v zdrojovom súbore, v ktorom sa nachádza analýza `ReachableFuncsAnalysis`. Je to z toho dôvodu, že sa nepredpokladá využitie tejto analýzy aj na iných miestach. Jej úlohou je získať všetky nepriame volania, ktoré sa nachádzajú v nejakej funkcii. Trieda pre túto analýzu dedí z triedy `llvm::InstVisitor`. Tá sprístupňuje navštevovanie inštrukcií, ktoré je realizované návrhovým vzorom *visitor* [42]. Vďaka tomu je možné prejsť všetky inštrukcie `call` a určiť, ktoré volania predstavujú nepriame volania. Celú túto procedúru je možné vykonať prostredníctvom metódy `getIndirectCallsFor()`, ktorá vracia množinu nepriamych volaní nachádzajúcich sa vo funkciách predaných parametrom tejto metódy.

Analýza `ReachableFuncsAnalysis` pracuje nasledovne. Vypočíta všetky dosiahnuteľné funkcie, ktoré sú volané priamo. Tento výpočet je realizovaný prostredníctvom analýzy z frameworku LLVM. Ide o analýzu obsahujúcu graf volania funkcií a predstavuje ju trieda `llvm::CallGraph`. Následne v týchto získaných funkciách nájde všetky nepriame volania pomocou analýzy `IndirectCallsFinder`. Nad týmito nepriamymi volaniami spustí analýzu `IndirectlyCalledFuncsAnalysis`, vďaka čomu získa všetky možné funkcie, ktoré môžu byť volané nepriamo z funkcií, ktoré sú volané priamo. Tým však ešte jej úloha nekončí. Opäť je potrebné vykonať popísanú procedúru nad všetkými zatiaľ získanými funkciami, pretože napríklad z nepriamo volanej funkcie môže opäť dôjsť k priamemu volaniu iných funkcií. Popísanú procedúru je potrebné vykonávať do doby, pokiaľ dochádza k nejakým zmenám v množine dosiahnuteľných funkcií.

Úpravy LLVM IR kódu

Odstránenie nedosiahnuteľnej funkcie prebieha v metóde `removeFuncFromModule()`, ktorá sa nachádza v anonymnom priestore mien tejto optimalizácie. Táto metóda obsahuje rôzne volania iných metód, ktoré poskytuje LLVM pre odstránenie funkcie z modulu. Pomocou nich je potrebné zmazať telo funkcie, odstrániť ju z grafu volania a následne ju odstrániť z modulu.

7.5 Odstránenie kódu za volaniami funkcií, ktoré sa nevracajú

Táto optimalizácia je implementovaná pomocou triedy `NeverReturningFuncs`. Keďže upravuje telá funkcií, tak dedí z triedy `llvm::FunctionPass`. Je nutné, aby táto trieda implementovala čisto virtuálnu metódu `runOnFunction()`. Optimalizácia sa nachádza vo vlastnom priestore mien a to konkrétne `never_returning_funcs`.

Trieda pre túto optimalizáciu ešte dedí aj z triedy `llvm::InstVisitor`. Tá sprístupňuje navštevovanie inštrukcií, ktoré je realizované návrhovým vzorom *visitor*. Vďaka tomu je možné prejsť všetky inštrukcie `call` a určit, ktoré volania predstavujú volania funkcií, ktoré sa nevracajú. Ide o rovnaký princíp, aký bol uvedený v podkapitole 7.4.

Optimalizácie pracuje nasledovne. Prejde všetky volania funkcií, ktoré sa nachádzajú vo funkciách v LLVM IR kóde. Tie porovná s `std::map` obsahujúcou funkcie, ktoré sa nevracajú. Jej inicializáciu zabezpečuje metóda `initFuncNeverReturnsMap()`. Mapovaný je vždy názov funkcie na `llvm::Function`. Pokiaľ dôjde k zhode názvu, typu návratovej hodnoty a počtu predávaných argumentov a ich typov, tak pridá inštrukciu `unreachable` na koniec bloku, v ktorom dochádza k volaniu. Pridanie na koniec je potrebné z toho dôvodu, že LLVM neumožňuje pridať ukončovaciu inštrukciu napríklad do stredu bloku. Na záver odstráni všetky inštrukcie nachádzajúce sa medzi volaním funkcie a ukončovacou inštrukciou, pretože nikdy nedôjde k vykonaniu týchto inštrukcií. Ešte je však potrebné vykonať jednu činnosť. Vďaka tomu, že dochádza k zámene inštrukcií skoku za inštrukciu `unreachable`, tak je potrebné túto zmenu reflektovať aj v inštrukciách `phi`. V nich je potrebné odobrať predchádzajúci blok, pokiaľ došlo k pridaniu inštrukcie `unreachable` pre tento blok.

Úpravy LLVM IR kódu

Nahradenie ukončovacích inštrukcií za inštrukciu `unreachable` je realizované v metóde `replaceTerminatorInstsWithUnreachableInst()`. Odstraňovanie inštrukcií medzi volaním funkcie, ktorá sa nevracia a inštrukciou `unreachable` je implementované v metóde `removeInsts()`. Metóda `updatePHINodesIfNeeded()` upravuje inštrukcie `phi` popísaným spôsobom. Uvedené metódy využívajú metódy z LLVM frameworku k splneniu ich úloh.

Kapitola 8

Testovanie optimalizácií

Vytváranie testov je potrebná činnosť, pretože počas vývoja tohto spätného prekladača dochádza k veľa úpravám už stavajúceho kódu, prípadne k implementovaniu nových prvkov. Tieto zmeny môžu mať dopad už aj na funkčné optimalizácie. V zadnej časti spätného prekladača vyvíjaného spoločnosťou AVG Technologies sú aktuálne podporované dve možnosti testovania. Ide o takzvané jednotkové testovanie a referenčné testy. V strednej časti spätného prekladača ale chýbala podpora pre testovanie pomocou referenčných testov. Súčasťou tejto diplomovej práce bolo navrhnúť framework pre toto testovanie a implementovať ho. Rovnako bolo potrebné priebežne vytvárať nové testy pre implementované optimalizácie. Niektoré testy boli vytvorené pred začatím implementácie, aby slúžili ako vhodný príklad uvedomenia si cieľov tejto optimalizácie.

Podkapitola 8.1 obsahuje popis navrhnutého frameworku pre referenčné testovanie. Popis jednotkových testov sa nachádza v podkapitole 8.2. Okrem týchto dvoch spôsobov testovania bolo možné otestovať optimalizácie pomocou takzvaných nočných testov a regresných testov. Týmto testom sa venuje podkapitola 8.3. Podkapitola 8.4 obsahuje popis zhodnotenia testovania a počet navrhnutých testov v tejto diplomovej práci.

8.1 Referenčné testy

Pri referenčnom testovaní dochádza k aplikovaní optimalizácií nad zdrojovým súborom testu a následne získaný výsledok po vykonaní optimalizácií je porovnaný z referenčným výstupom. Navrhnuté a vytvorené referenčné testovanie v strednej časti spätného prekladača vychádza z konceptu a implementácie referenčných testov v zadnej časti.

Vytvorený framework pre referenčne testovanie v strednej časti spätného prekladača umožňuje dva prístupy testovania. V prvom prípade ide o testovanie, pri ktorom sú spúšťané nad referenčnými testami všetky aktuálne preložené optimalizácie. Pri druhom prístupe testovania je možné zvoliť množinu optimalizácií, ktorá bude spustená nad referenčnými testami. Táto množina spustených optimalizácií je určená pomocou súboru `config`. Navrhnuté referenčné testovanie podporuje zoskupovanie testov do adresárov, ktoré charakterizujú danú množinu testov. Najčastejšie ide o adresáre zoskupujúce všetky referenčné testy pre danú optimalizáciu, ale nemusí to byť vždy jediným pravidlom.

Pre vytvorenie jedného referenčného testu je potrebný zdrojový súbor testu v jazyku

LLVM IR. Nad každým referenčným testom následne dôjde k spusteniu nástroja `opt` so špecifikovanou množinou optimalizácií. Výstupom nástroja `opt` je takzvaný *bitcode*. Tento kód je možné transformovať do kódu v jazyku LLVM IR pomocou nástroja `llvm-dis`, ktorý poskytuje framework LLVM [24]. Získaný LLVM IR kód po vykonaní optimalizácií je porovnaný s referenčným súborom odpovedajúcim konkrétnemu testu. Porovnanie je vykonané pomocou štandardného nástroja `diff`. V prípade, že dôjde k rozdielnemu porovnaniu, vypíše sa výstup z programu `diff` a spustí sa nad zdrojovým súborom testu spätný preklad. Pri tomto spätnom preklade je však v čo najväčšom množstve vypnutá funkcionalita zadnej časti spätného prekladača, aby nedochádzalo k zbytočnému ovplyvňovaniu výsledkov touto zadnou časťou. Výstupom tohto spätného prekladu je súbor v jazyku C a aj jazyku Python. Následne sú tieto výstupy porovnané s referenčnými súbormi v príslušných jazykoch určených pre tento test. Porovnanie je opäť realizované pomocou programu `diff` a v prípade rozdielu je vypísaný výstup tohto porovnania. Testovanie na základe výstupov v jazyku C a Python bolo zvolené z dôvodu, že pri pohľade na vysokoúrovňovejší kód môže dôjsť k rýchlejšiemu odhaleniu vzniknutej chyby. Výstup z úspešného porovnávania referenčných testov je zobrazený na ukážke 8.1.

```
Running all tests in /decompiler/...
dead_global_assign/aggresive/complicatedRecursion01  OK (ll)
dead_global_assign/aggresive/cycle01                 OK (ll)
dead_global_assign/aggresive/cycle02                 OK (ll)
...
```

Obr. 8.1: Výsledok úspešného porovnania referenčných testov.

V prípade neúspechu obsahuje výstup z porovnávania referenčných testov názov neúspešného testu a zároveň je zobrazený výstup zo štandardného nástroja `diff`. Výstup z neúspešného porovnávania referenčných testov je viditeľný na ukážke 8.2.

```
Running all tests in /decompiler/...
dead_global_assign/aggressive/complicatedRecursion01 OK (ll)
dead_global_assign/aggressive/cycle01
                                                     ERROR (ll)
  29d28
  <   store i32 2, i32* @glob0
                                                     OK (py) OK (c)
dead_global_assign/aggressive/cycle02                OK (ll)
...
```

Obr. 8.2: Výsledok neúspešného porovnania referenčných testov.

Na ukážke 8.2 je vidieť, že oproti očakávanému referenčnému výstupu optimalizácia ponechala jednu inštrukciu `store`, ktorá mala byť odstránená. Referenčné porovnanie výstupov pre jazyk C a jazyka Python dopadlo v poriadku napriek rozdielnosti v jazyku LLVM IR. Toto je zapríčinené tým, že spätný preklad je realizovaný pomocou štandardného skriptu na to určeného a ten neumožňuje vypnúť optimalizácie v strednej časti spätného prekladača. Vďaka tomu došlo na takomto jednoduchom teste k optimalizácii zbytočných častí kódu, a preto nedošlo potom pri konečnom porovnaní k rozdielnosti. Napriek tomu však toto správanie nie je problematické, nakoľko porovnanie výstupu v jazyku C a jazyku Python je určené iba k možnému urýchleniu nájdenia chyby, pretože vysokoúrovňovejší kód môže byť prehľadnejší.

Pre generovanie referenčných súborov pre jednotlivé testy bol vytvorený skript, ktorý zabezpečuje generovanie referenčných súborov v jazyku LLVM IR, C a Python. Súčasne skript podáva informácie o úspešnosti tohto generovania referenčných súborov. Príklad tohto výstupu je možné vidieť na ukážke 8.3.

```
Generating the reference outputs for all tests in /decompiler/...
/dead_global_assign/aggresive/cycle03   OK (ll) OK (py) OK (c)
/dead_global_assign/aggresive/cycle04   OK (ll) OK (py) OK (c)
/dead_global_assign/aggresive/cycle05   OK (ll) OK (py) OK (c)
...
```

Obr. 8.3: Výsledok úspešného generovania referenčných súborov.

Umiestnenie skriptov a testov je nasledujúce:

```
decompiler
├── bin ....................................Obsahuje skripty pre testovanie.
└── ref_tests ..............................................Referenčné testy.
    ├── all_optimizations
    ├── phi2seq
    ├── dead_global_assign
    ├── global_to_local
    ├── global_to_local_and_dead_global_assign
    ├── unreachable_funcs
    └── never_returning_funcs
```

Na uvedenej adresárovej štruktúre testov si je možné všimnúť, že pre optimalizáciu mŕtvych priradení a optimalizáciou konvertujúcu globálne premenné na lokálne sú vytvorené samostatné adresáre pre testy, avšak nachádza sa aj spoločný adresár pre obidve optimalizácie. Adresár `global_to_local_and_dead_global_assign` obsahuje testy, kedy dochádza k súbežnému behu obidvoch optimalizácií. Nad testami v adresári `dead_global_assign` sú spúšťané testy len s optimalizáciou odstraňujúcou mŕtve priradenia. Rovnaký princíp je zachovaný u adresára `global_to_local`, kde dochádza len k spusteniu optimalizácie konvertujúce globálne premenné na lokálne. Adresár `all_optimizations` obsahuje testy, nad ktorými sú spúšťané všetky preložené optimalizácie nachádzajúce sa v strednej časti.

Pre všetky optimalizácie bolo navrhnutých a vytvorených 127 referenčných testov.

8.2 Jednotkové testy

Hlavnou výhodou testovania pomocou jednotkových testov je možnosť otestovať určité časti samostatne a nezávisle na iných [32]. Napríklad môže ísť o testovanie výstupu metódy, prípadne celej analýzy. V spätnom prekladači sú používané jednotkové testy, ktoré sú písane pomocou testovacie frameworku Google Test [14]. Na príklade 8.4 je možné vidieť, ako taký jednotkový test vyzerá. Ide o jednotkový test pre analýzu použití.

Riadky 2-8 popisujú kód, ktorý bude vstupom pre vyhodnotenie touto analýzou. Konštrukcia tohto kódu pomocou LLVM frameworku sa nachádza na riadkoch 10-18. Na riadku 10 dochádza k vytvoreniu modulu. Vytvorenie funkcie s názvom `func1` je realizované na riadkoch 11-12 a riadky 13-14 zabezpečujú vytvorenie globálnej premennej `glob0`. Na riadku 15

```
1   TEST_F(UsesAnalysisTest, GetUseInfoRightUseTest) {
2   // Testing if analysis returns correct info for instruction.
3   //
4   // @glob0 = global i32 0
5   // define void @func1() {
6   // bb:
7   //   %x = load i32* @glob0
8   // }
9
10  Module *module = new Module("test", getGlobalContext());
11  Constant *funcConstant(module->getOrInsertFunction("func1",
12      Type::getVoidTy(getGlobalContext()), AttributeSet(), NULL));
13  Constant *globConstant(module->getOrInsertGlobal("glob0",
14      Type::getInt32Ty(getGlobalContext())));
15  GlobalVariable *glob = cast<GlobalVariable>(globConstant);
16  BasicBlock *bb = BasicBlock::Create(getGlobalContext(), "bb",
17      cast<Function>(funcConstant));
18  LoadInst *loadInst = new LoadInst(glob, "x", bb);
19  GlobVarSet globs{glob};
20  usesAnalysis.doUsesAnalysis(globs);
21
22  const UsesAnalysis::UseInfo *info = usesAnalysis.getUseInfo(
23      *bb, *loadInst));
24
25  ASSERT_TRUE(info) <<
26      "Expected a non-null pointer.";
27
28  EXPECT_FALSE(info->isLUse) <<
29      "Expected that contained info is a right use.";
```

Obr. 8.4: Príklad jednotkového testu pre analýzu použití.

je realizované potrebné pretypovanie na globálnu premennú. Na riadku 16-17 dochádza k vytvoreniu základného bloku s názvom `bb` a súčasne je tento blok umiestnený do vytvorenej funkcie. Riadok 18 predstavuje vytvorenie inštrukcie `load`. Na riadkoch 19 a 20 dochádza k naplneniu vstupných dát pre analýzu použití a dochádza aj k jej spusteniu. Riadok 22 vyžaduje výsledok analýzy a ukladá ho do premennej `info`. Na riadku 25 sa overuje, či bol získaný nejaký výsledok a riadok 28 testuje, či nejde o ľavé použitie, ale ide o pravé použitie. Pod pojmom pravé použitie pre pripomenutie sa rozumie situácia, kedy dochádza k načítaniu hodnoty z globálnej premennej.

Výsledok ukážkového úspešného testu je zobrazený na obrázku 8.5.

```
[ RUN      ] UsesAnalysisTest.GetUseInfoRightUseTest
[       OK ] UsesAnalysisTest.GetUseInfoRightUseTest (0 ms)
```

Obr. 8.5: Výsledok úspešného testu zobrazeného na obrázku 8.4.

Pokiaľ však popísaný jednotkový test neprejde napríklad z dôvodu, že dôjde k nesprávnemu rozpoznaniu pravého použitia, tak výsledok by vyzeral tak, ako je uvedený na obrázku 8.6.

Ako je možné vidieť, tak písanie jednotkových testov obsahujúcich LLVM IR kód je veľmi

```
[ RUN      ] UsesAnalysisTest.GetUseInfoRightUseTest
/decompiler/.../uses_analysis_tests.cpp:124: Failure
    Value of: info->isLUse
        Actual: true
    Expected: false
        Expected that contained info is a right use.
[  FAILED  ] UsesAnalysisTest.GetUseInfoRightUseTest (0 ms)
```

Obr. 8.6: Výsledok neúspešného testu zobrazeného na obrázku 8.4.

zložité a pracné. Preto optimalizácie sú testovaná referenčnými testami, kde je možné kód zapísať priamo v textovej podobe. Napriek tomu bolo vytvorených niekoľko jednotkových testov, ktoré testujú hlavne kritické časti niektorých analýz a zároveň sú testované niektoré funkcie zo súborov obsahujúcich spoločnú funkcionalitu v adresári `utilities`. Celkovo bolo vytvorených v tejto diplomovej práci 18 jednotkových testov.

Umiestnenie jednotkových testov je nasledujúce:

```
decompiler
└─ src
   └─ tests
```

8.3 Regresné a nočné testy

Okrem jednotkových a referenčných testov, ktoré sú prevažne určené k testovaniu jednotlivých častí spätného prekladača, boli navrhnuté optimalizácie testované pomocou nočných testov a takzvaných regresných testov. Tieto spôsoby testovania sa nachádzali v spätnom prekladači už pred začatím tejto práce a tak ich bolo možné využiť k dôkladnému otestovaniu nových optimalizácií.

Nočné testy predstavujú veľkú sadu testov, ktorá je spúšťaná nad celým spätným prekladačom. Pri nočných testoch sa kontroluje úspešnosť spätného prekladu a kvalita výstupného kódu. Rovnako je možné vidieť doby behu jednotlivých častí spätného prekladača nad týmito testami. Testovanie sa vykonáva na každej podporovanej architektúre s rôznymi vstupmi a variáciami. Tieto nočné testy otestujú spätný prekladač po každej stránke [17].

Cieľom regresných testov je detekcia situácií, pri ktorých niektoré aspekty spätného prekladača prestanú fungovať po pridaní novej funkcionality [38]. Regresné testy zároveň umožňujú písať takzvané kvalitatívne testy. Typickým príkladom kvalitatívneho testu je otestovanie, že do výsledného programu v jazyku C boli vygenerované všetky funkcie, všetky potrebné hlavičkové súbory, relevantné komentáre (napr. detekovaný jazyk), všetky funkcie majú správny počet parametrov, cykly boli identifikované ako `for` cykly a nie `while` a pod. Rovnako je možné skontrolovať zhodnosť výstupu vygenerovaného súboru v jazyku C so zdrojovým súborom v jazyku C, pričom výstupný súbor vznikol pomocou spätného prekladu z binárneho súboru získaného prekladom zo zdrojového súboru [3].

Nočné testy aj regresné testy sú spúšťané pravidelne nad pridanou funkcionalitou, pričom tieto výsledky sa uchovávajú aby bolo možné v prípade potreby dohľadať na správnom mieste vznik chýb [17].

V tejto diplomovej práci nebol pridaný ani jeden regresný ani nočný test, pretože pred za-

čatím tejto diplomovej práce ich bolo dostatočné množstvo a zároveň testovali i presúvané optimalizácie v zadnej časti. Vďaka tomu bolo možné otestovať, či nové navrhnuté optimalizácie sú funkčne ekvivalentné s tými v zadnej časti, nakoľko došlo vždy po presune optimalizácie do strednej časti k vypnutiu ekvivalentnej optimalizácie v zadnej časti.

8.4 Zhodnotenie

Súčasťou tejto diplomovej práce bolo dôkladne otestovať navrhnuté a implementované optimalizácie. Všetky navrhnuté referenčné a jednotkové testy prechádzajú, a tak kontrolujú správnosť implementovaných optimalizácií. Rovnako ani ostatné spôsoby testovania popísané v tejto kapitole nevykazujú žiadnu chybu, ktorá by súvisela s novými navrhnutými optimalizáciami. Všetky referenčné testy v strednej časti spätného prekladača ale aj v zadnej časti boli vždy priebežne po navrhnutí optimalizácie ručne skontrolované, či nová optimalizácia nezaniesla nejaké chyby.

Počet vytvorených referenčných testov zobrazuje tabuľka 8.1. V tabuľke je možné vidieť počet referenčných testov pre jednotlivé optimalizácie, ale aj počet navrhnutých testov pre prípad, kedy dochádza k spusteniu všetkých optimalizácií v strednej časti. Ako už bolo uvádzané, tak referenčné testy pre optimalizáciu odstraňujúcu mŕtve priradenia a optimalizáciu konvertujúcu globálne premenné sú rozdelené do dvoch skupín. Prvou skupinou testov sú také, ktoré sú spúšťané len s jednou z týchto dvoch uvedených optimalizácií. Druha skupina obsahuje testy spúšťané spoločné s obidvomi optimalizáciami. Uvedené rozdelenie je tiež zahrnuté v tejto tabuľke.

Tabuľka 8.1: Počet navrhnutých referenčných testov.

Optimalizácia	**Počet testov**
Testy pre všetky spúšťané optimalizácie	4
Usporiadanie PHI uzlov	13
Odstránenie mŕtvych priradení	49
Konverzia globálnych premenných na lokálne	40
Odstránenie mŕtvych priradení a konverzia globálnych premenných na lokálne	5
Odstránenie nedosiahnuteľných funkcií	10
Odstránenie kódu za volaniami funkcií, ktoré sa nevracajú	6
Spolu	127

Celkovo bolo vytvorených 18 jednotkových testov, z toho 11 testov bolo vytvorených pre analýzu použití. Okrem tejto analýzy je testovaná jednotkovými testami aj analýza závislosti premenných, ktorá je využívaná optimalizáciou realizujúcou usporiadanie PHI uzlov. Konkrétne sa jedná o 4 jednotkové testy. Zvyšok jednotkových testov je určený pre niektoré metódy nachádzajúce sa v zdrojových súboroch v adresári `utilities`.

Kapitola 9

Dosiahnuté výsledky a ich zhodnotenie

Táto kapitola popisuje dosiahnuté výsledky a ich zhodnotenie pre novo navrhnuté a implementované optimalizácie. Každá jedna podkapitola okrem poslednej sa venuje práve výsledkom jednej optimalizácie. Posledná podkapitola 9.6 obsahuje celkové zhodnotenie.

Prínos nových navrhnutých optimalizácií je možné posúdiť z dvoch hľadísk. Prvým hľadiskom je posúdenie prínosu na čitateľnosť výstupného kódu. Vo väčšine prípadov sa jednalo o presun optimalizácií nachádzajúcich sa v zadnej časti spätného prekladača. Vzhľadom na to, že tieto optimalizácie boli používané v spätnom prekladači a zlepšovali čitateľnosť výstupného kódu, tak ich presunom do strednej časti nedošlo k zhoršeniu tejto čitateľnosti. Samozrejme musí platiť, že nové optimalizácie funkčne pokrývajú tie staré. Táto vlastnosť je splnená. Dokonca boli niektoré optimalizácie rozšírené o novú funkcionalitu a zároveň vyriešili niekoľko chýb v optimalizáciách nachádzajúcich sa v zadnej časti. V jednotlivých podkapitolách sa nachádza detailný popis implementovanej rozširujúcej funkcionality, prípadne aké chyby boli objavené a vyriešené v starých optimalizáciách.

Druhým hľadiskom je urýchlenie doby behu spätného prekladača. Väčšina optimalizácií bola presúvaná do strednej časti práve kvôli tomuto účelu. Zrýchlenie bolo testované na nočných testoch. Nočné testy boli špeciálne spúšťané tak, aby dochádzalo k čo najmenšiemu ovplyvňovaniu dosiahnutých výsledkov pre jednotlivé optimalizácie. Celková doba behu nočných testov je cca 15 hodín a 40 minút. Tento čas je dosiahnutý paralelným spúšťaním testov na serveri s 24 procesormi. V prípade jedného procesoru by išlo približne o 376 hodín. V nasledujúcich podkapitolách budú popísané časy, ktoré zobrazujú zrýchlenie spätného prekladača vďaka novým navrhnutým optimalizáciám. Tieto časy budú uvádzané pre sekvenčné spúšťanie testov, čo znamená využitie jedného procesoru. Je to z toho dôvodu, aby bolo možné názornejšie ukázať prínos nových optimalizácií.

9.1 Usporiadanie PHI uzlov

Táto navrhnutá a implementovaná optimalizácia vyriešila problémy spojené s paralelným vyhodnocovaním inštrukcií `phi`. Potvrdzujú to funkčné referenčné testy, ako aj nasadenie tejto optimalizácie v spätnom prekladači.

9.2 Odstránenie mŕtvych priradení do globálnych premenných

Nová optimalizácia bola navrhnutá tak, aby pokryla funkcionalitou optimalizáciu mŕtvych priradení v zadnej časti spätného prekladača. Počas návrhu tejto optimalizácie došlo k odhaleniu niekoľkých chýb v optimalizácii mŕtvych priradení do globálnych premenných nachádzajúcej sa v zadnej časti spätného prekladača. Nová optimalizácia v strednej časti neobsahuje tieto zmienené chyby.

Optimalizácia mŕtvych priradení v zadnej časti spätného prekladača si principiálne neporadí s prípadom uvedeným na ukážke 9.1. Optimalizácia v zadnej časti spätného prekladača totižto neobsahuje podporu pre zistenie, či posledné priradenia vo funkciách sú potrebné mimo funkcií, v ktorých sa nachádzajú. Preto optimalizácia v zadnej časti neoptimalizuje žiadne posledné priradenia a z toho dôvodu by v uvedenej ukážke nedošlo k optimalizovaniu priradenia `g = 2`. Naopak navrhnutá optimalizácia v strednej časti spätného prekladača obsahuje podporu pre tento problém a tak korektne optimalizuje i tieto prípady.

```
int g = 0;

int func1() {
  g = 2;
  return 4;
}

int func2() {
  func1();
  return 4;
}
```

Obr. 9.1: Príklad zobrazujúci rozšírenie funkcionality optimalizácie v strednej časti oproti tej v zadnej časti spätného prekladača.

Rýchlosťou optimalizácia mŕtvych priradení v strednej časti prekonáva optimalizáciu v zadnej časti spätného prekladača. Tabuľka 9.1 zobrazuje porovnanie týchto dvoch optimalizácií. Ako už bolo uvádzané, tak v strednej časti dochádza k súčasnému spusteniu optimalizácie odstraňujúcej mŕtve priradenia a optimalizácie konvertujúcej globálne premenné na lokálne. Z tohto dôvodu sa nachádza v tabuľke výsledný čas obidvoch týchto optimalizácií. Pre zadnú časť je v tabuľke uvedený čas len pre samostatnú optimalizáciu. Na základe uvedeného porovnania je vidieť, že došlo k značnému zrýchleniu tejto optimalizácie. Hlavným dôvodom takéhoto zrýchlenia je to, že väčšina LLVM IR kódu je analyzovaná pomocou jedného priechodu.

Tabuľka 9.1: Porovnanie časov pre optimalizáciu odstraňujúcu mŕtve priradenia.

	Optimalizácia v strednej časti	**Optimalizácia v zadnej časti**
Čas	00h 10m 30s	11h 36m 14s

9.3 Konverzia globálnych premenných na lokálne

Pri navrhovaní tejto optimalizácie ostal aj priestor na jej vylepšenie. Výrazný prínos zabezpečilo navrhnutie optimalizácie tak, že niektoré časti kódu vo funkcii môže byť optimalizované a nahradené lokálnou premennou a iné časti nie. V optimalizácii nachádzajúcej sa v zadnej časti spätného prekladača dochádzalo k posudzovaniu konverzie globálnej premennej na lokálnu vo funkcii podľa toho, či je možné vykonať túto konverziu na všetkých miestach použitia tejto globálnej premennej. Nová optimalizácia sa na túto problematiku díva inak. Posudzuje priradenia hodnoty do globálnej premennej a použitia tejto priradenej hodnoty. Na základe toho je možné čiastočne optimalizovať niektoré časti kódu a nahrádzať v nich globálne premenné za lokálne.

Okrem toho bola zavedená konverzia globálnej premennej na lokálnu v prípade, že je použitie iba v jednej funkcii a pred použitím sa nenachádza žiadne priradenie hodnoty do tejto globálnej premennej.

Rýchlosťou optimalizácia konvertujúca globálne premenné na lokálne v strednej časti prekonáva optimalizáciu v zadnej časti spätného prekladača. Tabuľka 9.2 zobrazuje porovnanie týchto dvoch optimalizácií. Rovnako aj v tomto prípade sa nachádza v tabuľke čas pre súčasný beh optimalizácie odstraňujúcej mŕtve priradenia a optimalizácie konvertujúcej globálne premenné na lokálne. Pre zadnú časť je v tabuľke uvedený čas len pre samostatnú optimalizáciu. Na základe uvedeného porovnania je vidieť, že došlo k značnému zrýchleniu tejto optimalizácie. Hlavným dôvodom takéhoto zrýchlenia je to, že väčšina LLVM IR kódu je analyzovaná pomocou jedného priechodu.

Tabuľka 9.2: Porovnanie časov pre optimalizáciu konvertujúcu globálne premenné na lokálne.

	Optimalizácia v strednej časti	**Optimalizácia v zadnej časti**
Čas	00h 10m 30s	10h 04m 32s

9.4 Odstránenie nedosiahnuteľných funkcií

Optimalizácia odstraňujúca nedosiahnuteľné funkcie v zadnej časti spätného prekladača si principiálne neporadí s prípadom uvedeným na ukážke 9.2, pretože v tomto prípade neoznačí funkciu `func()` ako nedosiahnuteľnú. Dochádza k tomu v prípadoch, kedy funkcia nie je volaná z `main` funkcie a rekurzívne volá samú seba. Nová navrhnutá optimalizácia v strednej časti si poradí aj s týmto prípadom.

```
void func() {
  func();
}

int main() {}
```

Obr. 9.2: Príklad zobrazujúci rozšírenie funkcionality optimalizácie v strednej časti oproti tej v zadnej časti spätného prekladača.

Pri návrhu optimalizácie v strednej časti bola odhalená aj chyba v optimalizácii nachádza-

júcej sa v zadnej časti. Išlo o prípad, kedy táto optimalizácia nesprávne označila niektoré funkcie za nedosiahnuteľné. V optimalizácii v zadnej časti spätného prekladača sa neuvažovalo s tým, že je potrebné iteratívne upravovať dosiahnuteľné funkcie do doby, kým dochádza k zmene množiny dosiahnuteľných funkcií. Opakovaný priechod je potrebný z toho dôvodu, že pokiaľ je označená funkcia ako dosiahnuteľná na základe toho, že dôjde k jej volaniu nepriamo, tak je potrebné opätovne z tejto funkcie zisťovať všetky dosiahnuteľné funkcie či už priamo, alebo nepriamo. Toto sa však v optimalizácii nachádzajúcej sa v zadnej časti neuskutočňovalo. Nová navrhnutá optimalizácia samozrejme tento problém rieši.

Pri presune tejto optimalizácie do strednej časti sa nevychádzalo z toho, že presun je potrebný z dôvodu pomalosti tejto optimalizácie v zadnej časti spätného prekladača. Hlavnou motiváciou presunu bolo odstrániť čo najväčšie množstvo kódu neovplyvňujúceho beh programu skôr, než v zadnej časti. Vďaka tomu dôjde k urýchleniu spätného prekladača, pretože zbytočný kód nie je potrebné následne analyzovať inými časťami spätného prekladača. Vďaka presunu optimalizácie došlo k zrýchleniu behu nočných testov o cca 3.7 %. Pri prevedení percent do sekvenčného času by sa jednalo o zrýchlenie cca 14 hodín.

9.5 Odstránenie kódu za volaniami funkcií, ktoré sa nevracajú

Dôvod presunu tejto optimalizácie do strednej časti je rovnaký ako uvedený v podkapitole 9.4. Vďaka presunu optimalizácie došlo k zrýchleniu behu nočných testov o cca 2.9 %. Pri prevedení percent do sekvenčného času by sa jednalo o zrýchlenie cca 11 hodín.

9.6 Celkové zhodnotenie

Nočné testovanie bolo vďaka novým optimalizáciám urýchlené o cca 11 %. V prípade paralelného spúšťania testov na 24 procesoroch ide o zrýchlenie približne 1 hodinu a 45 minút. Pokiaľ by boli testy spúšťané sekvenčne na jednom procesore, tak zrýchlenie by bolo približne 42 hodín.

Funkčnosť novo navrhnutých optimalizácií dokazujú prechádzajúce testy a zároveň nasadenie týchto optimalizácií, pri ktorom nie sú hlásené žiadne problémy spojené s nimi.

Kapitola 10

Záver

Hlavným cieľom tejto práce bol presun niektorých optimalizácií zo zadnej časti spätného prekladača do strednej časti. Celkovo bolo navrhnutých päť optimalizácií. Konkrétne ide o optimalizáciu riešiacu usporiadanie PHI uzlov, optimalizáciu odstraňujúcu mŕtve priradenia, optimalizáciu konvertujúcu globálne premenné na lokálne, optimalizáciu odstraňujúcu nedosiahnuteľné funkcie a optimalizáciu odstraňujúcu kód za volaniami funkcií, ktoré sa nevracajú. Súčasťou práce bolo preštudovať dostupné optimalizácie nachádzajúce sa vo frameworku LLVM a posúdiť ich prínos pre spätný prekladač. Pre testovanie navrhnutých optimalizácií boli vytvorené skripty umožňujúce ich testovanie pomocou referenčných testov. Ďalej bolo pre tieto optimalizácie vytvorené množstvo referenčných testov a jednotkových testov.

Nové navrhnuté optimalizácie je možné posúdiť z dvoch hľadísk. Prvým hľadiskom je posúdenie prínosu na čitateľnosť výstupného kódu. Vo väčšine prípadov išlo o presun optimalizácií nachádzajúcich sa v zadnej časti spätného prekladača do strednej časti. Vzhľadom na to, že tieto optimalizácie boli používané v spätnom prekladači a zlepšovali čitateľnosť výstupného kódu, tak ich presunom do strednej časti nedošlo k zhoršeniu tejto čitateľnosti. Samozrejme platí, že nové optimalizácie funkčne pokrývajú tie staré. Dokonca boli niektoré nové optimalizácie v tejto diplomovej práci funkčne rozšírené oproti tým nachádzajúcim sa v zadnej časti. Konkrétne ide o optimalizáciu odstraňujúcu mŕtve priradenia, optimalizáciu konvertujúcu globálne premenné na lokálne a optimalizáciu odstraňujúcu nedosiahnuteľné funkcie. Zároveň bolo vyriešených niekoľko chýb nachádzajúcich sa v optimalizáciách v zadnej časti pri novo navrhnutých optimalizáciách. Konkrétne ide o optimalizáciu odstraňujúcu mŕtve priradenia a optimalizáciou odstraňujúcu nedosiahnuteľné funkcie.

Druhým hľadiskom je urýchlenie doby behu spätného prekladača. Nočné testovanie bolo vďaka novým optimalizáciám urýchlené o cca 11 %. V prípade paralelného spúšťania testov na 24 procesoroch ide o zrýchlenie približne 1 hodinu a 45 minút. Pokiaľ by boli testy spúšťané sekvenčne na jednom procesore, tak zrýchlenie by bolo približne 42 hodín.

Cieľom tejto práce bolo urýchliť beh spätného prekladača, konkrétne presunom niektorých optimalizácií zo zadnej časti do strednej časti. Zároveň bolo potrebné nové navrhnuté optimalizácie dostatočne otestovať. Súčasťou práce bolo preštudovať dostupné optimalizácie nachádzajúce sa vo frameworku LLVM a posúdiť ich prínos pre spätný prekladač. Uvedené ciele boli v tejto diplomovej práci úspešne splnené.

Z hľadiska budúceho vývoja je možným rozšírením nových navrhnutých optimalizácií v tejto

práci implementácia analýzy, ktorá rieši problematiku ukazovateľov. Táto analýza by bola prínosná pre optimalizáciu odstraňujúcu mŕtve priradenia a optimalizáciu konvertujúcu globálne premenné na lokálne. V týchto optimalizáciách by bola využiteľná pre určenie toho, aké všetky ukazovatele ukazujú na globálnu premennú.

Literatúra

[1] [ISO/IEC 14882:2011] Information technology – Programming languages – C++. Ženeva, Švýcarsko: Mezinárodní organizace pro standardizaci, 2011.

[2] AVG team: Retargetable Decompiler. [online], [cit. 2015-01-13].
URL https://retdec.com/

[3] AVG team: Regresní testy. Interná wiki projektu zpětného překladače.

[4] AVG team: Soupis optimalizací v backendu. Interná wiki projektu zpětného překladače.

[5] backerstreet.com: Reverse Engineering Resources. [online], [cit. 2014-12-23].
URL http://www.backerstreet.com/cg/work.htm

[6] Caetano, L.: Mobile Malware in 2014. [online], Poslední modifikace 2014-03-25 [cit. 2014-12-29].
URL http://blogs.mcafee.com/consumer/mobile-malware-2014

[7] Cipresso, T.: *Software reverse engineering education*. Diplomová práce, San Jose state university, 2009.
URL http://scholarworks.sjsu.edu/etd_theses/3734

[8] CMake team: CMake. [online], [cit. 2015-01-14].
URL http://www.cmake.org/

[9] DebugMode: Decompilation. [online], Poslední modifikace 2001-10-22 [cit. 2014-12-25].
URL http://www.debugmode.com/dcompile/

[10] Eilam, E.: *Reversing: Secrets of Reverse Engineering*. Wiley Publishing, Inc., 2005, ISBN-10: 0-7645-7481-7.

[11] GCC: Static Single Assignment. [online], [cit. 2014-12-26].
URL https://gcc.gnu.org/onlinedocs/gccint/SSA.html

[12] GCC: GCC, the GNU Compiler Collection. [online], [cit. 2015-04-08].
URL https://gcc.gnu.org/

[13] Gerçek, B.: Jak funguje antivirový program? [online], [cit. 2014-12-23].
URL http://www.symantec.com/region/cz/resources/antivirus.html

[14] Google.com: Google C++ Testing Framework. [online], [cit. 2013-04-06].
URL http://code.google.com/p/googletest/

[15] Kollár, J.: *Optimalizace v zadní části zpětného překladače*. Bakalářská práce, Brno university of technology, 2013.

[16] Křoustek, J.: *Retargetable Analysis of Machine Code*. Dizertační práce, Brno university of technology, 2015.

[17] Křoustek, J.; Ďurfina, L.; Matula, P.; aj.: *Testování zpětného překladače*. Interná technická správa projektu Lissom, 2014.

[18] Křoustek, J.; Ďurfina, L.; Zemek, P.: Generic Source Code Migration Using Decompilation. In *10th Annual Industrial Simulation Conference (ISC'2012)*, EUROSIS, 2012, ISBN 978-90-77381-71-7, s. 38–42.

[19] Křoustek, J.; Ďurfina, L.; Zemek, P.; aj.: Design of a Retargetable Decompiler for a Static Platform-Independent Malware Analysis. *International Journal of Security and Its Applications*, 2011: s. 91–106, ISSN 1738-9976.

[20] Křoustek, J.; Ďurfina, L.; Zemek, P.; aj.: Detection and Recovery of Functions and Their Arguments in a Retargetable Decompiler. In *19th Working Conference on Reverse Engineering (WCRE'2012)*, IEEE Computer Society, 2012, ISBN 978-0-7695-4891-3, s. 51–60.

[21] LLVM Project: The LLVM Compiler Infrastructure. [online], [cit. 2015-01-11].
URL http://llvm.org/

[22] LLVM Project: Clang: A C language family frontend for LLVM. [online], [cit. 2015-04-08].
URL http://clang.llvm.org/

[23] LLVM Project: LLVM Language Reference Manual. [online], Poslední modifikace 2014-12-23 [cit. 2014-12-23].
URL http://llvm.org/docs/LangRef.html

[24] LLVM Project: opt – LLVM optimizer. [online], Poslední modifikace 2014-12-23 [cit. 2014-12-23].
URL http://llvm.org/docs/CommandGuide/opt.html

[25] LLVM Project: LLVM's Analysis and Transform Passes. [online], Poslední modifikace 2014-12-28 [cit. 2014-12-28].
URL http://llvm.org/docs/Passes.html

[26] LLVM Project: LLVM Alias Analysis Infrastructure. [online], Poslední modifikace 2014-12-29 [cit. 2014-12-29].
URL http://llvm.org/docs/AliasAnalysis.html

[27] LLVM Project: LLVM API Documentation. [online], Poslední modifikace 2014-12-29 [cit. 2014-12-29].
URL http://llvm.org/docs/doxygen/html/index.html

[28] LLVM Project: LLVM Language Reference Manual. [online], Poslední modifikace 2014-12-29 [cit. 2014-12-29].
URL http://llvm.org/docs/LangRef.html#volatile

[29] LLVM Project: Writing an LLVM Pass. [online], Poslední modifikace 2014-12-29 [cit. 2014-12-29].
URL http://llvm.org/docs/WritingAnLLVMPass.html

[30] Oreans Technologies: Themida Features. [online], [cit. 2014-12-25].
URL http://www.oreans.com/themida_features.php

[31] Rouse, M.: Disassemble. [online], Poslední modifikace 2005-09 [cit. 2014-12-29].
URL http://whatis.techtarget.com/definition/disassemble

[32] Rouse, M.: Unit testing. [online], Poslední modifikace 2007-02 [cit. 2013-03-29].
URL
http://searchsoftwarequality.techtarget.com/definition/unit-testing

[33] Russell, K. J.: Obfuscation. [online], Poslední modifikace 2006-10 [cit. 2014-12-29].
URL http://searchsoftwarequality.techtarget.com/definition/obfuscation

[34] TechTerms.com: Malware. [online], [cit. 2014-12-27].
URL http://www.techterms.com/definition/malware

[35] Wikipedia contributors: Reverse engineering. [online], Poslední modifikace 2014-04-19 [cit. 2014-12-21].
URL http://en.wikipedia.org/wiki/Reverse_engineering

[36] Wikipedia contributors: Topological sorting. [online], Poslední modifikace 2014-10-23 [cit. 2014-12-26].
URL http://en.wikipedia.org/wiki/Topological_sorting

[37] Wikipedia contributors: Static single assignment form. [online], Poslední modifikace 2014-12-10 [cit. 2013-12-27].
URL http://en.wikipedia.org/wiki/Static_single_assignment_form

[38] Wikipedia contributors: Software testing. [online], Poslední modifikace 2014-12-26 [cit. 2014-12-28].
URL http://en.wikipedia.org/wiki/Software_testing

[39] Wikipedia contributors: Strongly connected component. [online], Poslední modifikace 2014-12-26 [cit. 2015-01-11].
URL http://en.wikipedia.org/wiki/Strongly_connected_component

[40] Wikipedia contributors: Depth-first search. [online], Poslední modifikace 2014-12-30 [cit. 2015-01-14].
URL http://en.wikipedia.org/wiki/Depth-first_search

[41] Wikipedia contributors: Tree traversal. [online], Poslední modifikace 2015-01-14 [cit. 2015-01-14].
URL http://en.wikipedia.org/wiki/Tree_traversal

[42] Wikipedia contributors: Visitor pattern. [online], Poslední modifikace 2015-02-28 [cit. 2015-04-10].
URL http://en.wikipedia.org/wiki/Visitor_pattern

[43] Zemek, P.: Méně známé skutečnosti o C: Funkce, které se nevrací. [online], [cit. 2015-04-05].
URL http://cs-blog.petrzemek.net/node/123

[44] Zemek, P.: *Design of a Language for Unified Code Representation*. Interná technická správa projektu Lissom, 2012.

[45] Zhao, J.; Nagarakatte, S.; Martin, M. M. K.; aj.: Formalizing the LLVM Intermediate Representation for Verified Program Transformations. *12 Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 2012: s. 427–440.
URL http://acg.cis.upenn.edu/papers/popl12_vellvm.pdf

Dodatok A

Obsah DVD

- Zdrojové súbory, obrázky a Makefile pre vygenerovanie textu práce do elektronickej podoby.
- Text práce v elektronickej podobe.
- Zdrojové kódy implementácie optimalizácií navrhnutých v tejto práci.
- Makefile pre preklad jednotkových testov a optimalizácií vytvorených v tejto diplomovej práci.
- Spustiteľná verzia spätného prekladača.
- Referenčné testy vytvorené v rámci tejto diplomovej práce.
- Skripty pre referenčné testovanie a generovanie referenčných testov, ktoré boli vytvorené v rámci tejto diplomovej práce.
- Zdrojové kódy jednotkových testov, ktoré boli vytvorené v rámci tejto diplomovej práce.
- Súbor README s pokynmi pre spustenie spätného prekladu, spustenie testov a informáciami o adresároch na DVD.
- Skript, ktorý zjednodušuje prácu pri zobrazovaní výsledkov jednotkových a referenčných testov.
- Externé knižnice.
- Rôzne balíčky ako sú gnuarm, pspsdk, clang a iné, potrebné k spätnému prekladu.