VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INFORMAČNÍCH SYSTÉMŮ
DEPARTMENT OF INFORMATION SYSTEMS

ALIAS ANALÝZA V PŘEKLADAČI JAZYKA C

ALIAS ANALYSIS IN C COMPILER

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — **DÁVID BOLVANSKÝ**
AUTHOR

VEDOUCÍ PRÁCE — **prof. Ing. TOMÁŠ HRUŠKA, CSc.**
SUPERVISOR

BRNO 2018

Zadání bakalářské práce/19579/2017/xbolva00

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav informačních systémů Akademický rok 2017/2018

Zadání bakalářské práce

Řešitel: **Bolvanský Dávid**
Obor: Informační technologie
Téma: **Alias analýza v překladači jazyka C**
Alias Analysis in C Compiler
Kategorie: Překladače

Pokyny:

1. Seznamte se s aplikačním rámcem pro kompilátory Clang/LLVM a konceptem alias analýzy.
2. Seznamte se s existujícími implementacemi alias analýzy v LLVM.
3. Porovnejte tyto existující implementace a zhodnoťte jejich výhody a nevýhody.
4. Po konzultaci s vedoucím vyberte a implementujte vhodný algoritmus alias analýzy a integrujte jej do LLVM.
5. Na vybrané sadě testů porovnejte implementaci s existujícími algoritmy v LLVM z hlediska rychlosti překladu a výkonu výsledné aplikace.
6. Zhodnoťte dosažené výsledky a navrhněte možná budoucí vylepšení.

Literatura:

- Yannis Smaragdakis and George Balatsouras. 2015. Pointer Analysis. Found. Trends Program. Lang. 2, 1 (April 2015), 1-69.
- Appel, Andrew W. (1998). Modern Compiler Implementation in ML. Cambridge, UK: Cambridge University Press
- L. Andersen. Program Analysis and Specialization for the C Programming Language. PhD thesis, University of Copenhagen, DIKU, DIKU report 94/19, 1994.
- Steensgaard, Bjarne (1996), "Points-to analysis in almost linear time", Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '96), New York, NY, USA: ACM, pp. 32-41
- http://llvm.org/docs/AliasAnalysis.html

Pro udělení zápočtu za první semestr je požadováno:

- Body 1 až 3.

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

Technická zpráva bakalářské práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap (20 až 30% celkového rozsahu technické zprávy).

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

Vedoucí: **Hruška Tomáš, prof. Ing., CSc.**, UIFS FIT VUT
Konzultant: Šnobl Pavel, Ing., CODASIP
Datum zadání: 1. listopadu 2017
Datum odevzdání: 16. května 2018

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

L.S.

doc. Dr. Ing. Dušan Kolář
vedoucí ústavu

Abstrakt

Táto práca sa venuje problematike alias analýzy a možnostiam jej vylepšenia v LLVM frameworku. Cieľom tejto práce je zlepšiť jej presnosť, čoho bolo dosiahnuté rozšírením implementácie Andersenovho algoritmu o citlivosť na položky štruktúr. Vysvetlené sú pojmy súvisiace s alias analýzou a je popísaný princíp algoritmov alias analýzy. Predstavený je koncept LLVM frameworku, popísané sú aktuálne implementované algoritmy alias analýzy. Porovnanie týchto algoritmov bolo vykonané z pohľadu ich princípu fungovania, vlastností a obmedzení. Implementácia citlivosti na položky štruktúr bola vyskúšaná na sade programov, ktorými sa testujú prekladače. Bol preskúmaný jej vplyv na rýchlosť prekladu programov a ich výkonnosť. Získané výsledky preukazujú zvýšenie presnosti alias analýzy v LLVM frameworku.

Abstract

This thesis is dedicated to the problem of alias analysis and possibilities of its improvement in the LLVM framework. The goal of this thesis is to improve the accuracy, which was achieved by extending the existing implementation of Andresen algorithm to be field sensitive. The terms related to alias analysis and algorithms of the alias analysis available in LLVM are explained. These algorithms are compared according to their base idea, features, and limitations. The implementation of the field sensitivity has been tested using compiler test suites. Its impact on program compilation speed and performance has been analyzed. The measured results show an increase in the accuracy of alias analysis in the LLVM framework.

Kľúčové slová

alias analýza, analýza ukazovateľov, optimalizácie, ukazovateľ, štruktúra, položka, statická analýza, LLVM, Clang

Keywords

alias analysis, pointer analysis, optimizations, pointers, structure, field, static analysis, LLVM, Clang

Citácia

BOLVANSKÝ, Dávid. *Alias analýza v překladači jazyka C*. Brno, 2018. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce prof. Ing. Tomáš Hruška, CSc.

Alias analýza v překladači jazyka C

Prehlásenie

Prehlasujem, že som túto bakalársku prácu vypracoval samostatne pod vedením pána prof. Ing. Tomáša Hrušku, CSc. Ďalšie informácie mi poskytol Ing. Pavel Šnobl. Uviedol som všetky literárne pramene a publikácie, z ktorých som čerpal.

.....
Dávid Bolvanský
13. mája 2018

Poďakovanie

Rád by som vyslovil poďakovanie vedúcemu mojej bakalárskej práce prof. Ing. Tomášovi Hruškovi, CSc. a môjmu odbornému konzultantovi Ing. Pavlovi Šnoblovi za pomoc, ochotu a venovaný čas pri písaní tejto práce. Ďakujem vývojárom LLVM za technické rady týkajúce sa LLVM frameworku a spoločnosti Codasip za poskytnuté nástroje, ktoré boli použité v tejto práci.

Obsah

Kapitola 1

Úvod

Typickým znakom niektorých imperatívnych programovacích jazykov je dátový typ *ukazovateľ*. Ukazovatele sú dôležitejšou časťou jazyka C/C++, ale aj zradnou. Ukazovateľ umožňuje čítanie a zápis do ľubovoľného miesta v pamäti programu. Reprezentuje adresu objektu a nesie so sebou súčasne informáciu o dátovom type objektu, ktorý sa na tejto adrese nachádza. Má široké využitie v podobe predávania argumentov funkcií odkazom, adresovania dynamicky alokovanej pamäte, nepriameho adresovania premenných. Ďalej je možné v programoch používať ukazovatele na ukazovatele, ukazovatele na funkcie, a tiež meniť typ ukazovateľov. Pomocou ukazovateľov je možné nepriamo zmeniť obsah pamäte, čo predstavuje komplikácie nielen pre prekladače, ale aj pre nástroje, ktoré analyzujú programy používajúce ukazovatele.

Mnohé prekladače vykonávajú rôzne optimalizácie nad zdrojovými kódmi. Bez informácií o mieste, ktoré môže byť nepriamo zmenené pomocou ukazovateľa, prekladač nedokáže vykonávať rôzne optimalizácie týkajúce sa predovšetkým prístupov do pamäti. Súčasťou moderných prekladačov je alias analýza, ktorá tieto informácie poskytuje. Informácie získané alias analýzou umožňujú vykonávanie optimalizácií prístupov do pamäti a ich eliminácií či zmeny poradia, ale taktiež môže ísť o automatickú paralelizáciu kódu a rôzne iné transformácie programu.

Alias analýza, taktiež známa ako analýza ukazovateľov či *points-to* analýza, sa zaraďuje medzi statické analýzy programov. Alias analýza zisťuje, či dva ukazovatele môžu ukazovať na ten istý objekt v pamäti. Pre tento účel existuje niekoľko algoritmov. Súčasťou práce je popis princípov týchto algoritmov spolu s jednoduchými príkladmi, ktoré ozrejmujú ich činnosť v praxi.

Bakalárska práca sa zameriava na alias analýzu v LLVM a na možnosti zlepšenia jej aktuálne suboptimálneho stavu. Časť práce sa venuje analýze implementovaných algoritmov a ich porovnaniu. Následne sú analyzované jej nedostatky a obmedzenia. Na základe získaných poznatkov je navrhnuté a implementované rozšírenie Andersenovho algoritmu v LLVM o citlivosť na položky štruktúr za účelom zvýšenia presnosti výsledkov alias analýzy. Skúma sa vplyv tohto rozšírenia na rýchlosť prekladu programov a ich výkonnosť na mnohých testovaných programoch a architektúrach procesorov.

Kapitola 2

Alias analýza

Alias analýza je v súčasnej dobe aktívnou témou vo výskume [17, 2, 11]. Pre analýzu programov, ktoré používajú ukazovatele, je potrebná znalosť princípu ich fungovania. Ukazovateľ je premenná, ktorá obsahuje adresu na miesto v pamäti, kde je uložený nejaký objekt. Objektom rozumieme pomenovanú oblasť v pamäti. Analýza ukazovateľov si kladie za cieľ staticky určiť možné hodnoty ukazovateľa za behu programu. Bez týchto znalostí dochádza ku konzervatívnym odhadom ohľadom prístupov do pamäti, čo ovplyvňuje efektivitu a presnosť analýz optimalizátora založených na týchto informáciách [9].

Vo všeobecnosti je problém, ktorý sa alias analýza snaží vyriešiť, nerozhodnuteľný [6]. Z tohto dôvodu bolo publikovaných niekoľko aproximačných algoritmov, ktoré poskytujú kompromis medzi efektívnosťou analýzy a presnosťou počítačového riešenia.

Alias analýza je jedna zo základných typov statickej analýzy programov. Prebieha v prekladači počas zostavovania programu do binárnej podoby. Statická analýza je založená na skúmaní zdrojových kódov, prípadne objektových súborov. Cieľom alias analýzy je vypočítať množiny objektov v programe, na ktoré môže ukazovateľ alebo výraz, ktorého je súčasťou, ukazovať. Ak dva ukazovatele (napr. *p1*, *p2*) odkazujú na to isté miesto v pamäti, vtedy hovoríme, že *p1* je aliasom *p2*. Alias analýza tradične odpovedá na otázku, či dva ukazovatele ukazujú na to isté miesto v pamäti niekoľkými možnými výstupmi. Výstupy alias analýzy sú detailnejšie popísané v podkapitole 2.6.

Existuje mnoho rôznych algoritmov alias analýzy a mnoho spôsobov ich klasifikácie. Môžeme ich klasifikovať nielen podľa citlivosti na tok programu, kontext, položky štruktúr, ale aj podľa toho či sú založené na zjednocovaní alebo podmnožinách. Podrobná klasifikácia je v podkapitole 2.4.

2.1 Vznik aliasov

Aliasy môže vzniknúť nasledujúcimi spôsobmi:

- Ukazovateľmi

```
int *p, x;
p = &x;
```

  Aliasy: **p* a *x*

- Predaním hodnotou ukazovateľa

```
void foo(int *a, int *b) { ... }

int main(void) {
        int x = 0;
        foo(&x, &x);
}
```

  Aliasy: **a* a **b*

- Indexovaním poľa

```
int i, j;
int arr[32];
i = j;
```

  Aliasy: *arr[i]* a *arr[j]*

- Štruktúrou typu *union*

```
union {
        int i;
        float f;
} u;
```

  Aliasy: *u.i* a *u.f*

2.2 Formulácia problematiky alias analýzy

Cieľom alias analýzy je vypočítať aliasové vzťahy medzi dvoma ukazovateľmi. Ak máme dva ukazovatele p a q, alias analýza rozhoduje o tom, či môžu byť aliasmi alebo nie, tzv. *may analysis*. V posledných dvoch desaťročiach bolo navrhnutých niekoľko prístupov na riešenie problému. Vhodným prístupom je formulovať problém alias analýzy ako problém dosiahnuteľnosti pre bezkontextový jazyk (*CFL reachability problem*) [26].

2.2.1 Graf programových výrazov

Interná reprezentácia programu vytvorená prekladačom je transformovaná do grafovej štruktúry *Program Expression Graph (PEG)*, ktorá obsahuje programové výrazy. Transformácia prebieha podľa nasledovných pravidiel:

- každý ukazovateľ sa stane uzlom v grafe
- pre každú inštrukciu priradenia $x = y$ sa pridá hrana z uzlu y do uzlu x s hranou s popisom P do grafu (P značí, že išlo o priradenie)
- pre každú inštrukciu načítania $x = load\ y$ sa pridá hrana z uzlu y do uzlu x s hranou s popisom N do grafu (N značí, že išlo o načítanie)
- pre každú inštrukciu uloženia *store x, y* sa pridá hrana z uzlu y do uzlu x s hranou s popisom N do grafu
- pre každú P hranu z uzlu x do uzlu y sa pridá ďalšia hrana z uzlu y do uzlu x s popisom $\overline{P}$
- pre každú N hranu z uzlu x do uzlu y sa pridá ďalšia hrana z uzlu y do uzlu x s popisom $\overline{N}$

To, ako sú transformované inštrukcie pre návrat z funkcií a volanie funkcií, závisí od citlivosti analýzy na kontext. Pri kontextovo necitlivej schéme sú zavedené ďalšie pravidlá:

- pridá sa P hrana z každého aktuálneho parametra do každého formálneho parametra
- pridá sa P hrana z každej návratovej hodiny do ľavej strany každého miesta, kde došlo k volaniu funkcie

Precíznejšia schéma je popísaná v podkapitole 5.5.

2.2.2 Bezkontextová gramatika alias analýzy

Vznik pamäťových aliasov je možné popísať nasledujúcou bezkontextovou gramatikou [27]:

$$\begin{aligned} \mathbf{M} &::= \overline{\mathbf{N}} \quad \mathbf{V} \quad \mathbf{N} \\ \mathbf{V} &::= \overline{\mathbf{F}} \quad \mathbf{M}? \quad \mathbf{F} \\ \mathbf{F} &::= (\mathbf{P} \quad \mathbf{M}?)^* \\ \overline{\mathbf{F}} &::= (\mathbf{M}? \quad \overline{\mathbf{P}})^* \end{aligned}$$

Obrázok 2.1: Bezkontextová gramatika pre alias analýzu [27]

Terminály v tejto gramatike P, $\overline{P}$, N, $\overline{N}$ zodpovedajú popisom uzlov v PEG. Neterminály majú nasledujúci význam:

- ***M***: pamäťové aliasy
  Dva ukazovatele sú pamäťové, aliasy, ak označujú rovnaké miesto v pamäti.
- ***V***: hodnotové aliasy
  Dva ukazovatele sú hodnotové aliasy, ak majú rovnakú hodnotu ukazovateľa.
- ***F***: toky hodnôt
- $\overline{\boldsymbol{F}}$: obrátené toky hodnôt

Pravidlá je možné potom interpretovať nasledovne:

- $\mathbf{M} ::= \overline{\mathbf{N}} \quad \mathbf{V} \quad \mathbf{N}$
  Dva umiestnania v pamäti $\star e1$ a $\star e2$ sú pamäťovými aliasmi, $M(\star e1, \star e2)$, ak ich adresy majú rovnakú hodnotu: $V(\star e1, \star e2)$. Z toho dôvodu cesta z $\star e1$ do $\star e2$ sa skladá z opačnej hrany načítania $\overline{N}(\star e1, \star e2)$, hrany hodnotového aliasu $V(\star\ e1, \star e2)$ a hrany načítania $N(\star e1, \star e2)$.
- $\mathbf{V} ::= \overline{\mathbf{F}} \quad \mathbf{M}? \quad \mathbf{F}$
  Dva výrazy $e1$ a $e2$ sú hodnotovými aliasmi, ak existujú dva výrazy $e'1$ a $e'2$, ktoré sú pamäťovými aliasmi, $M(e'1, e'2)$, a ktorých tok hodnôt do $e1$, resp. $e2$, je: $F(e'1, e'2)$ a $\overline{F}(e'1, e'2)$.
- $\mathbf{F} ::= (\mathbf{P} \quad \mathbf{M}?)^*$
  Tok hodnôt je daný sekvenciami priradení a pamäťovými aliasmi.
- $\overline{\mathbf{F}} ::= (\mathbf{M}? \quad \overline{\mathbf{P}})^*$
  Tok hodnôt v opačnom smere.

2.2.3 Problém dosiahnuteľnosti pre bezkontextový jazyk

Po vytvorení PEG sa problém zisťovania, či dva ukazovatele x a y sú aliasmi, stáva problémom vyhľadávania, či existuje cesta v PEG z uzlu viažúceho sa ku ukazovateľu y do uzlu viažúceho sa k ukazovateľu x, a zároveň platí podmienka, že reťazec vytváraný spájaním popisov hrán na ceste je generovaný gramatikou uvedenou na obrázku 2.2.2.

2.3 Reprezentácie aliasov

Existuje niekoľko spôsobov reprezentácie aliasov, medzi ktoré patria:

1. *Points-to* páry, kde prvý prvok sa odkazuje na druhý
2. Dvojice odkazujúce sa na rovnaké miesto v pamäti
3. Zhodné (ekvivalentné) množiny

Príklad v jazyku C:

```
int x = 0;
int *p, *q;
p = &x;
q = p;
```

Použitím vyššie spomenutých spôsobov je možné aliasy v tomto príklade ($*p$ a x, $*q$ a x, $*p$ a $*q$) reprezentovať nasledovne:

1. (p → x), (q → x)
2. (*p, x), (*q, x), (*p, *q)
3. {*p, *q, x}

2.4 Klasifikácia prístupov k alias analýze

Aktuálne prístupy k alias analýze je možné kategorizovať pomocou niekoľkých základných kritérií [9]. Každé z týchto kritérií ma dopad na presnosť a zložitosť algoritmov alias analýzy a ovplyvňuje, ako sa modeluje spracovávaný program.

Citlivosť na tok programu (*flow sensitivity*): Klasifikácia podľa toho, či sa informácie o toku riadenia programu používajú počas analýzy. Absenciou týchto informácií dochádza ku konzervatívnemu výsledku.

Tokovo citlivá analýza využíva informácie o toku programu a počíta riešenie pre každý bod v programe. Má vyššiu presnosť ako tokovo necitlivá analýza, no je menej rýchlejšia.

U tokovo necitlivej analýze sa počíta jedno riešenie pre celý program alebo pre každú funkciu. Tokovo necitlivé analýzy sú analýzy založené buď na zhodnosti, kde sa považujú priradenia ako obojsmerné a typicky sa používa dátová štruktúra „disjuktná množina“, alebo na základe podmnožín, kde sa vníma priradenie ako jednosmerný tok hodnôt.

Nasledujúci príklad poukazuje na citlivosť na tok riadenia programu:

```
int main(void) {
        int a = 0, b = 0;
        int *p1, *p2;
        p1 = &a;
        p1 = &b;
        p2 = p1;
}
```

	Tokovo citlivá	Tokovo necitlivá
points-to množina	{p1 → {a, b}, p2 → {b}}	{p1 → {a, b}, p2 → {a, b}}

Tabuľka 2.1: Výsledky alias analýzy

Citlivosť na kontext (*context sensitivity*): Klasifikácia podľa toho, či sa kontext volania funkcie (*call site*) berie do úvahy pri analýze funkcie. Kontext volania funkcie je miesto, odkiaľ je funkcia volaná. Toto miesto je potrebné analyzovať samostatne, keďže hodnoty parametrov funkcie v jednom mieste jej volania nemajú žiadnu súvislosť s návratovými hodnotami v druhom mieste jej volania. Analýza, ktorá rozlišuje kontexty volania funkcie, sa označuje ako kontextovo citlivá analýza. Kontextovo necitlivá analýza neuvažuje kontexty volania funkcia a je menej presná.

Nasledujúci príklad poukazuje na citlivosť na kontext volania funkcie:

```
#include <assert.h>

int* inc(int *x) {
    *x += 1;
    return x;
}
void foo() {
    int a = 1;
    int *b = inc(&a);
    assert(a == 2);
}
void bar() {
    int c = 2;
    int *d = inc(&c);
    assert(c == 3);
}

int main(void) {
    foo();
    bar();
}
```

	Kontextovo citlivá	Kontextovo necitlivá
points-to množina	{b → {a}, d → {c}}	{b → {a, c}, d → {a, c}}

Tabuľka 2.2: Výsledky alias analýzy

Citlivosť na položky štruktúr (*field sensitivity*): Klasifikácia podľa toho, či sa jednotlivé položky štruktúr počas analýzy. Pri analýze necitlivej na položky štruktúr sa jednotlivé položky nerozlišujú a obsah celej štruktúry je modelovaný ako jeden objekt v pamäti. Ak sa však tieto položky rozlišujú a modelujú osobitne, ide o analýzu citlivú na položky štruktúr. Väčšina súčasných algoritmov je položkovo necitlivá.

Nasledujúci príklad poukazuje na citlivosť na položky štruktúry:

```
typedef struct { int *p1; int *p2; } example_t;
int main(void) {
    example_t a;
    int *c;
    int d, e, f;
    a.p1 = &d;
    a.p2 = &f;
    c = a.f1;
}
```

	Položkovo citlivá	Položkovo necitlivá
points-to množina	{c → {d}}	{c → {d, f}}

Tabuľka 2.3: Výsledky alias analýzy

2.5 Intraprocedurálna a interprocedurálna analýza

Podľa toho, či sa alias analýza sleduje tok informácií v rámci celého programu, alebo pre každú funkciu osobitne, rozlišujeme intraprocedurálnu a interprocedurálnu analýzu.

Intraprocedurálna analýza

Intraprocedurálna analýza skúma tok riadenia a dát v telách funkcií, no neuvažuje interakciu medzi funkciami [14]. Nerieši predávanie ukazovateľov ako argumentov alebo funkcie vracajúce ukazovatele. Je jednoduchšia na implementáciu, nenáročná na výkon, ale zároveň je aj menej presná.

Interprocedurálna analýza

Interprocedurálna analýza skúma interakcie medzi funkciami, dochádza k sledovaniu toku riadenia a dát naprieč volaniami funkcií [10]. Poskytuje presnejšie výsledky za cenu náročnejších výpočtov.

2.5.1 Prístupy interprocedurálnej analýzy

Vo všeobecnosti existujú dva odlišné prístupy na vykonávanie interprocedurálnej analýzy, a to prístup zdola nahor a prístup zhora nadol.

Prístup zdola nahor

Prístup zdola nahor (prístup založený na vyhodnocovaní funkcií) analyzuje funkciu až po dokončení analýzy všetkých jej volaní. Prístup zdola nahor je vo všeobecnosti presnejší ako prístup zhora nadol, keďže pri analýze volaní funkcie je už možné tieto volania zaradiť do známeho kontextu.

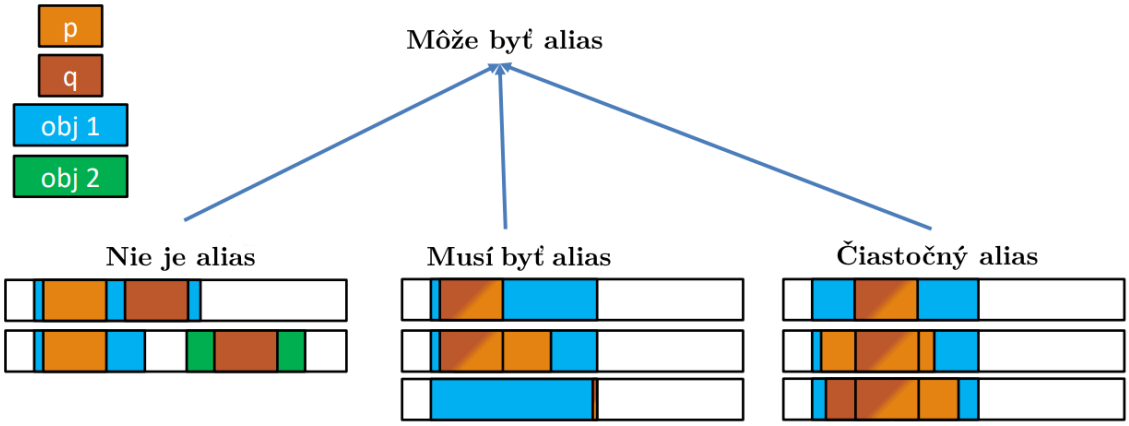
Prístup zhora nadol

Prístup zhora nadol analyzuje funkcie predtým ako dokončí analýzu všetkých jej volaní. Pri prístupe zhora nadol funkcie musia byť preskúmané nezávisle a mimo kontextu. Nie sú dostupné informácie o stave pamäti, argumentoch, globálnych hodnotách, atď. Na vykonanie analýzy je preto potrebné urobiť konzervatívne odhady týkajúce sa týchto externých stavov. Výmenou za možnú stratu presnosti získame lepšiu rozšíriteľnosť; každá funkcia sa spracováva iba raz a získané výsledky sú znovu použiteľné na všetkých miestach, kde dochádza k volaniu funkcie.

2.6 Výstupy alias analýzy

Alias analýza odpovedá na otázku, či dva ukazovatele ukazujú na to isté miesto v pamäti. Tento výstup nemusí byť binárny, a často sú rozlišované nasledujúce prípady:

- **Nie je alias (*No Alias*)**
  Dva ukazovatele neodkazujú na rovnaký objekt v pamäti.
- **Môže byť alias (*May Alias*)**
  Dva ukazovatele môžu odkazovať na rovnaký objekt v pamäti.
- **Čiastočný alias (*Partial Alias*)**
  Dva objekty v pamäti, ktoré sa prekrývajú a nemusia začínať na rovnakej adrese.
- **Musí byť alias (*Must Alias*)**
  Dva ukazovatele zaručene odkazujú na rovnaký objekt v pamäti.



Obrázok 2.2: Výstupy alias analýzy[1]

Na obrázku 2.2 sú zobrazené možné umiestnenia dvoch objektov v pamäti spolu s prípadmi, ako na ne môžu ukazovatele ukazovať. Ku každému prípadu je uvedený výstup alias analýzy.

[1] http://llvm.org/devmtg/2018-04/slides/Lopes-Sbirlea-Pointers,%20Alias%20and%20ModRef%20Analyses.pdf

Kapitola 3

Algoritmy alias analýzy

Táto kapitola popisuje niektoré z najznámejších algoritmov alias analýzy. Princíp týchto algoritmov je vysvetlený na jednoduchých príkladoch.

3.1 Andersenov algoritmus

Andersenov algoritmus bol predstavený v roku 1994. Ide o interprocedurálnu, tokovo a kontextovo necitlivú analýzu, ktorá je založená na riešení systému obmedzení podmnožín (*subset constraints*) [1]. Obmedzenia sú obmedzujúce podmienky, ktoré vyplývajú z operácií týkajúcich sa priradenia ukazovateľov. Cieľom algoritmu je vytvorenie *points-to (pts)* množiny pre každý ukazovateľ v programe.

Andersenov algoritmus alias analýzy vytvára *points-to* graf [1]. Uzly v grafe reprezentujú abstraktné pamäťové umiestnenia. Hrany grafu reprezentujú *points-to* informácie. Napríklad hrana z uzlu x do y značí, že x môže ukazovať na y [5].

Systém obmedzení je definovaný ako množina obmedzení nad typmi ukazovateľov. Pre získanie riešenia je potrebné prevádzať substitúcie premenných, ktoré sú ukazovateľmi na množiny abstraktných umiestnení, kým nie sú všetky obmedzenia splnené. Analýza založená na obmedzeniach funguje v dvoch fázach. V prvej fáze sa vytvára systém obmedzení, ktorý reprezentuje závislosti medzi ukazovateľmi a abstraktnými umiestneniami.

Typ obmedzenia	Priradenie	Obmedzenie	Význam
základné	a = &b	a $\subseteq$ {b}	b $\in$ pts(a)
jednoduché	a = b	a $\subseteq$ b	pts(a) $\in$ pts(a)
komplexné	a = *b	a $\subseteq$ *b	$\forall$v $\in$ pts(b): pts(a) $\subseteq$ pts(v)
komplexné	*a = b	*a $\subseteq$ b	$\forall$v $\in$ pts(a): pts(v) $\subseteq$ pts(b)

Tabuľka 3.1: Priradenia ukazovateľov a ich vzťah s obmedzeniami

V druhej fáze dochádza k riešeniu tohto systému obmedzení pomocou iteratívneho postupu vyhľadávania riešenia.

Nasledujúci príklad ukazuje princíp Andersenovho algoritmu.

```
int main(void) {
    int *a, *b, *c, *d,
    int x, y;
    a = &x;
    b = &a;
    c = &y;
    d = c;
    *b = a;
}
```

Príkazy v zdrojovom texte sú transformované do obmedzení:

Priradenie	Obmedzenie
a = &x	x ∈ pts(a)
b = &a	pts(a) ∈ pts(b)
c = &y	y ∈ pts(c)
d = c	pts(c) ∈ pts(d)
*b = d	∀v ∈ pts(d): pts(v) ⊆ pts(b)

Tabuľka 3.2: Priradenia ukazovateľov a ich vzťah s obmedzeniami

Následne sú získané obmedzenia iteratívne šírené, kým nie sú všetky obmedzenia splnené. Výsledné *points-to* množiny: {a → {x}}, {c → {y}}, {d → {x, y}}.

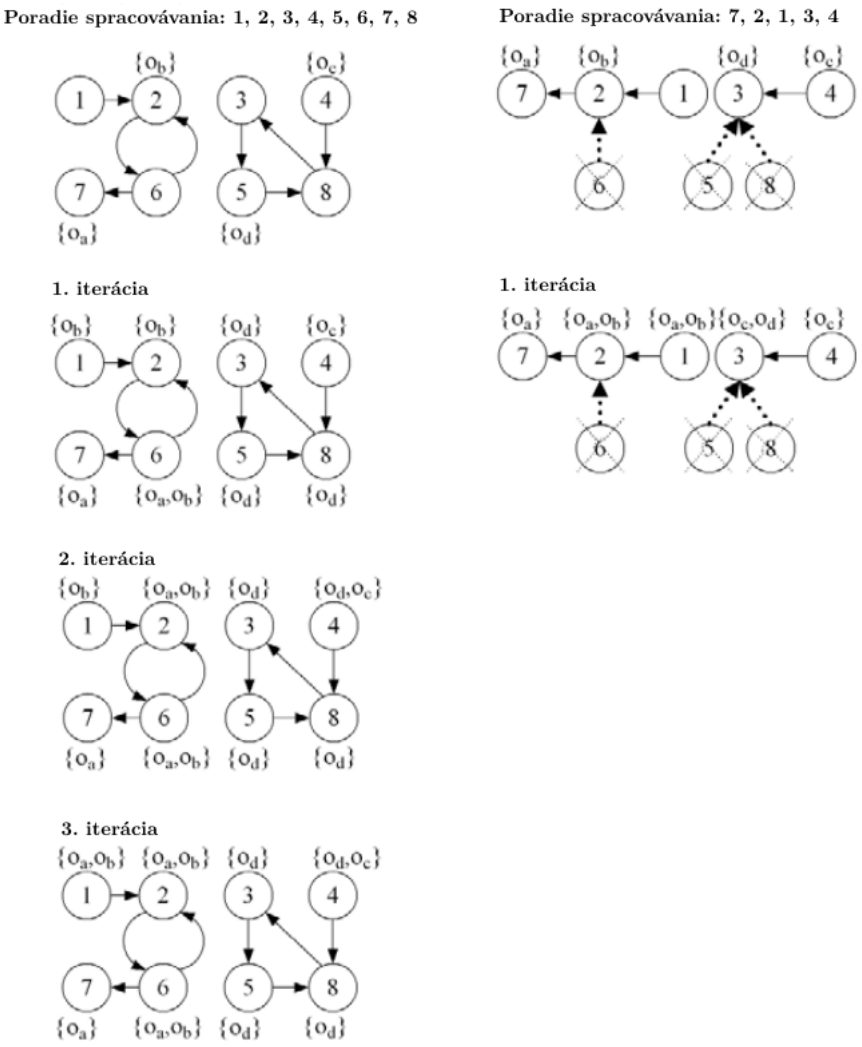
3.1.1 Optimalizácie výpočtu

Andersenov algoritmus má v najhorších prípadoch časovú zložitosť $\mathcal{O}(n^3)$ [21]. Procesorový čas je použitý hlavne na iteratívny proces vytváraniapoints-tomnožín pre každú premennú typu ukazovateľ. Optimalizácie výpočtu je možné dosiahnuť pomocou optimalizačných metód. Medzi tieto optimalizačné metódy patrí eliminácia cyklov (*cycle elimination*) a vytvorenie sumárov funkcií (*method summarization*) [20].

3.1.2 Eliminácia cyklov

Redukcia počtu uzlov pomocou eliminácie cyklov je hlavným prístupom na riešenie problému s efektívnosťou Andersenovho algoritmu. Niektoré cykly je možné odhaliť staticky, iné za behu pri pridávaní hrán.

Základom je myšlienka, že ak všetky uzly v cykle hrán majú rovnaké *points-to* množiny, tak je možné ich zlúčenie do jedného uzla [20]. Ak zlúčime všetky cykly v hlavnom *points-to* grafe, uzly a hrany vytvoria orientovaný acyklický graf. Následne môžeme topologicky zoradiť tento acyklický graf. Výpočet tranzitívneho uzáveru v topologickom poradí je oveľa efektívnejší, pretože je potrebná iba jedna iterácia prechodu nad uzlami.



Obrázok 3.1: Topologické zoradenie a eliminácia cyklov [20]

Obrázok 3.1 poukazuje na účinnosť procesu topologického zoradenia a procesu eliminácií cyklov. Na ľavej strane je pôvodný *points-to* graf s ôsmimi uzlami. Vedľa každého uzla je zobrazená jeho počiatočná *points-to* množina. Ak dôjde k výpočtu *points-to* množín uzlov v poradí od 1 po 8, algoritmus získa výsledky po troch iteráciách. Osem uzlov je spracovávaných v každej iterácií.

Points-to graf po topologickom zoradení a procese eliminácií cyklov je zobrazený na pravej strane. Uzol č. 2 a uzol č. 6 tvoria cyklus, a preto sú zlúčené do jedného uzlu. Uzol č. 2 sa stáva reprezentantom týchto dvoch uzlov s *points-to* množinou $\{o_b\}$. Cyklus tvorený uzlami č. 3, č. 5 a č. 8 je taktiež zrušený. Uzly sú zlúčené do jedného uzlu, uzlu č. 3, ako reprezentanta všetkých troch uzlov s *points-to* množinou $\{o_d\}$.

Uzly sú taktiež zoradené v poradí 7, 2, 1, 3, 4. Pri spracovaní uzlov v tomto poradí je potrebná len jedna iterácia. V tejto iterácii prebieha spracovanie piatich uzlov.

Vytvorenie sumárov funkcií

Namiesto práce priamo nad celým zdrojovým kódom algoritmus najskôr vytvorí sumáre každej funkcie v programe a následne za pomoci týchto sumárov sa vypočítajú *points-to* informácie. Tento prístup má dve výhody. Prvá spočíva v tom, že interprocedulárny algoritmus nemusí analyzovať zdrojový kód opakovane, čím sa zvyšuje efektívnosť a škálovateľnosť analýzy. Druhou výhodou je, že vytvorené sumáre môžu byť predspracované za účelom redukcie veľkosti hlavného *points-to* grafu, ktorý bude zostavený v interprocedurálnej fáze.

3.2 Steensgaardov algoritmus

Predstavený v roku 1996 ako algoritmus interprocedulárnej, tokovo a kontextovo necitlivej alias analýzy s takmer lineárnou časovou zložitosťou $\mathcal{O}(n \cdot \alpha(n, n))$, kde α je inverzná Ackermannova funkcia [22]. Podobne ako Andersenov algoritmus je založený na riešení systému obmedzení. Namiesto obmedzení podmnožín ale používa obmedzenia ekvivalencie (*equality constraints*). Je menej precíznejší ako Andersenov algoritmus, no výpočtovo je rýchlejší. V praxi sa ukázalo, že alias analýza programov s miliónmi riadkov kódu, ktorá prebehla pomocou Steensgardovho algoritmu, trvala pod jednu minútu [21].

Typ obmedzenia	Priradenie	Obmedzenie	Význam
základné	a = &b	a $\subseteq$ {b}	b $\in$ pts(a)
jednoduché	a = b	a = b	pts(a) = pts(a)
komplexné	a = *b	a = *b	$\forall$v $\in$ pts(b): pts(a) = pts(v)
komplexné	*a = b	*a = b	$\forall$v $\in$ pts(a): pts(v) = pts(b)

Tabuľka 3.3: Priradenia ukazovateľov a ich vzťah s obmedzeniami

Algoritmus je založený na neštandardnom typovom systéme. Typ odvodený pre každú premennú reprezentuje množinu umiestení, na ktoré môže premenná ukazovať. Typ odvodený pre premennú typu ukazovateľ na funkciu reprezentuje množinu funkcii, na ktoré môže premenná ukazovať a zahŕňa typ tejto funkcie. Výsledky sú ekvivalentné s alias analýzami, ktoré sú tokovo necitlivé, a kde sa predpokladá, že vzťahy medzi aliasmi sú reflexívne a tranzitívne.

Cieľom algoritmu je otypovanie zdrojového textu programu, t.j. každá premenná je opísaná odlišnými typovými premennými. Typová premenná sa skladá z rýchlej, tzv. *union-find* štruktúry s priradenou informáciou o type. Samotné otypovanie má byť minimálne. Každá typová premenná by mala popisovať čo najmenej umiestnení v pamäti. V ďalšej fáze sa spracuje každý príkaz len raz. Dochádza k zlučovaniu typových premenných tak, aby boli splnené pravidlá otypovania pre jednotlivé výrazy. Premenné sú zlučované stanovením ich spoločného najmenšieho horného uzáveru a zlúčením ich cieľových typových premenných.

Princíp tohto algoritmu je znázornený na nasledovnom príklade:

```
int main(void) {
    int a,b,c,d;
    int *p;

    p = &c;
    p = &a;
    a = &b;
    *p = &d;
}
```

V prvej fáze sú všetky premenné reprezentované samostatnou typovou premennou (ekvivalenčnou triedou). Následne sa začnú spracovávať jednotlivé príkazy v kóde.

1. po spracovaní prvého výrazu priradenia ekvivalečná trieda p ukazuje na triedu c
2. druhý príkaz spôsobí zlúčenie ekvivalenčnej triedy c a a
3. po spracovaní tretieho príkazu ekvivalenčná trieda a ukazuje na triedu zodpovedajúcu premennej b
4. po spracovaní posledného príkazu dôjde k zlúčeniu tried b a d:

Výsledné *points-to* množiny: {p → {a, c}}, {c → {b, d}}, {a → {b, d}}.

3.3 Typovo založená alias analýza

Ak analýza pri určovaní aliasov používa informácie o typovej kompatibilite, ide o typovo založenú alias analýzu [4]. Je založená na vlastnosti zo štandardu jazyka C/C++, že dve premenné odlišných typov nemôžu byť aliasmi. Každé umiestnenie v pamäti má typ a je možné pristúpiť k pamäti len za pomoci správneho typu (prekladač predpokladá že dva prístupy s odlišným typmi sa nealiasujú). Toto pravidlo umožňuje prekladaču zmeniť poradie operácii, keď je jasné podľa typov, že sa pristupuje k odlišným objektom.

Nasledujúci príklad poukazuje na optimalizáciu kódu pomocou typovo založenej alias analýze:

```
int i;
void foo(double *d) {
    i = 10;
    *d = 5.0;
    i = i + 40;
}
```

Zmena hodnoty **d* nemôže zároveň zmeniť aj hodnotu *i*, keďže ide o rôzne typy. Kompilátor teda môže presunúť uloženie hodnoty do *i* za uloženie hodnoty do **d*.

Optimalizovaná funkcia vyzerá nasledovne:

```
int i;
void foo(double *d) {
    *d = 5.0;
    i = 50;
}
```

Pravidlá typovo založenej alias analýzy hovoria len o prístupe k objektom, nehovoria nič o pretypovaní ukazovateľov (*pointer casting*). Ak sú splnené pravidlá pre pretypovanie ukazovateľov, tak je možné pretypovať ukazovatele medzi odlišnými typmi.

Nasledujúci príklad poukazuje na možnosti pretypovania ukazovateľov medzi odlišnými typmi:

```
int i;
float *f = (float *)&i;
```

V tom prípade však prístup k **f* nie je definovaným chovaním, keďže objekt na ktorý ukazuje je typu *int* a podľa štandardu jazyka C/C++ nie je dovolené pristupovať k nemu ako k typu *float*.

3.4 Analýza dátových štruktúr

Alias analýza založená na analýze dátových štruktúr (*Data Structure Analysis, DSA*) bola predstavená v roku 2017 Chrisom Lattnerom [13]. Podobne ako Steensgardov algoritmus je založená na princípe zjednocovania. Je tokovo necitlivá, no je citlivá na kontext a položky štruktúr, za čo bola považovaná za príliš náročnú a nevhodnú pre praktické použitie. DSA ponúka rýchlu kontextovo a položkovo citlivú alias analýzu vylúčením citlivosti na kontext v rámci silne súvislých komponent v grafe volaní funkcií.

DSA používa rozšírenie Tarjanovho algoritmu, ktorý slúži na vyhľadávanie silne súvislých komponentov [23] za účelom postupnej konštrukcie grafu volaní funkcií počas analýzy bez potreby iterácie, a to aj v prípade výskytu ukazovateľov na funkcie či výskytu rekurzívnych funkcií.

Citlivosť na kontext a položky štruktúr je dosiahnutá klonovaním hromady. Klonovanie hromady vytvára abstraktný popis pre každú dátovú štruktúru alebo funkciu. Pomocou tohto popisu je možné modelovať odlišné inštancie dátových štruktúr, ktoré boli vytvorené na rôznych miestach v programe.

DSA je vykonávaná v troch fázach: (1) lokálna analýza, (2) analýza zdola nahor, (3) analýza zhora nadol. V prvej fáze sa vytvára graf lokálnych dátových štruktúr pre každú funkciu. Obsahuje všetky objekty v pamäti, ktoré sú prístupné v rámci funkcie. V ďalšej fáze prebieha analýza zdola nahor, kde sa vkladajú informácie o volanej funkcie do grafu dátových štruktúr volajúcej funkcie. V poslednej fáze analýza zhora nadol doplní nekompletné informácie o argumentoch zlúčením grafov dátových štruktúr volajúcej a volanej funkcie.

V definícii grafu dátových štruktúr sa predpokladá, že vstupné programy majú jednoduchý typový systém so štrukturálnou ekvivalenciou. V typovom systéme majú celočíselné a reálne typy preddefinované veľkosti. Nachádzajú sa tu štyri odvodené typy, medzi ktoré patria ukazovatele, štruktúry, polia a funkcie. Každý uzol grafu reprezentuje množinu objektov v pamäti. Všetky objekty v pamäti, na ktoré môže premenná ukazovať, sú reprezentované jedným uzlom v grafe.

Podpora citlivosti na kontext a položky štruktúr výrazne zvyšuje presnosť alias analýzy. V porovnávaniach [13] sa ukázalo, že presnosť DSA môže byť na úrovni Andersenovho algoritmu alias analýzy.

Kapitola 4

LLVM framework

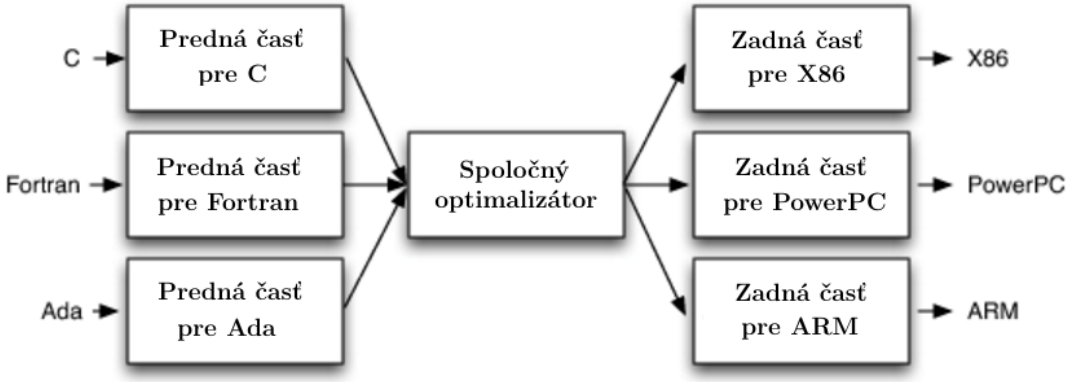
LLVM (*Low Level Virtual Machine*) je aplikačný rámec (framework) napísaný v jazyku C++, ktorý obsahuje kolekciu modulárnych a znovu použiteľných prekladových a nástrojových technológií (*toolchains*) slúžiacich na tvorbu prekladačov [15]. Pri jeho navrhovaní sa kládol dôraz na podporu pokročilých analýz a transformácií programov [12]. Poskytuje vysokoúrovňové informácie pre transformácie za dobu prekladu a zostavovania. Framework je licencovaný pod NCSA licenciou, ktorá je založená na MIT a BSD licenciách. LLVM framework a reprezentácia kódu (LLVM IR) spolu vytvárajú kombináciu dôležitých schopností, ktoré sú dôležité pre praktické dlhotrvajúce analýzy a transformácie programov.

4.1 Architektúra LLVM

LLVM sa skladá z troch hlavných častí: predná časť (*front end*), optimalizátor, zadná časť (*back end*).

Obrázok 4.1: Komponenty trojfázového prekladača [12]

Výhoda tohto návrhu sa naplno prejaví, ak je cieľom rozšíriť prekladač o podporu ďalších zdrojových jazykov alebo cieľových architektúr. Ak prekladač používa spoločnú reprezentáciu kódu v jeho optimalizátore, potom môže byť predná časť napísaná pre akýkoľvek jazyk, ktorý je možné transformovať do tejto spoločnej reprezentácie kódu. Taktiež zadná časť môže byť napísaná pre akúkoľvek architektúru, čo je znázornené na obrázku 4.2.



Obrázok 4.2: Architektúra LLVM frameworku [12]

4.2 Reprezentácia kódu

LLVM IR predstavuje vnútornú reprezentáciu kódu v LLVM[1], ktorá je považovaná za najdôležitejšiu časť návrhu LLVM. Ide o formu, ktorou predná časť prekladača reprezentuje zdrojový kód [15]. Bola navrhnutá s mnohými špecifickými cieľmi, vrátane podpory ľahkých optimalizácií za behu, intraprocedurálnych optimalizácií, celkovej analýzy programov a agresívnych reštrukturalizačných transformácií. Najdôležitejším aspektom tohto návrhu je, že LLVM IR forma je definovaná ako jazyk s dobre definovanou sémantikou. Vnútorná reprezentácia kódu je silne typovaná s jednoduchým typovým systémom, napríklad *i32* je 32-bitové celé číslo, *i32** je ukazovateľ na 32-bitové celé číslo. Inštrukcie majú trojadresnú formu, čo znamená, že majú určitý počet vstupov a výsledok majú v inom registri.

Program *Hello world* v jazyku C:

```c
#include <stdio.h>

int main(void) {
    puts("Hello world!");
}
```

Vnútorná reprezentácia tohto kódu v LLVM IR:

```
@.str = private unnamed_addr constant [13 x i8] c"Hello world!\00", align 1

define i32 @main() {
       %call = call i32 @puts(i8* getelementptr inbounds ([13 x i8],
              [13 x i8]* @.str, i32 0, i32 0))
       ret i32 0
}

declare i32 @puts(i8*)
```

LLVM IR forma musí byť jednoduchá na vygenerovanie prednou časťou prekladača, no zároveň dostatočne popisná na to, aby umožnila dôležité optimalizácie.

[1] https://llvm.org/docs/LangRef.html

4.3 Predná časť

Predná časť spracováva zdrojový kód a kontroluje ho na prítomnosť chýb. Zostavuje jazykovo špecifický abstraktný syntaktický strom (AST), ktorý reprezentuje vstupný kód. Abstraktný syntaktický strom je voliteľne transformovaný do novej reprezentácie vhodnej pre optimalizátor.

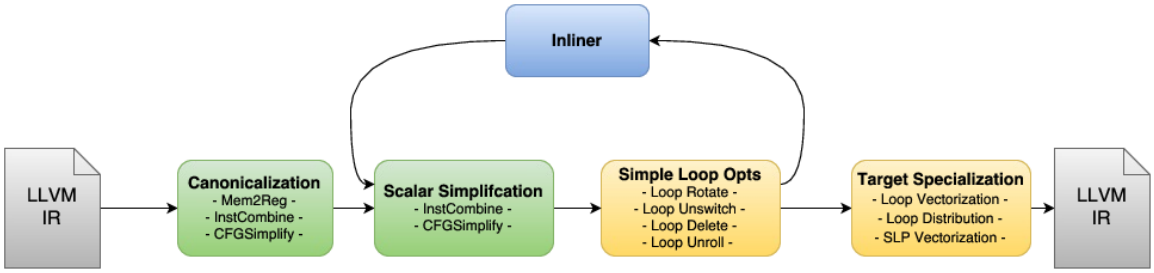
4.3.1 Clang

Clang[2] je prednou časťou prekladača pre programovacie jazyky C, C++, Objective C/C++, OpenCL C. Je súčasťou LLVM od verzie 2.6. Vďaka znovu použiteľným nástrojom umožňuje zníženie závislosti vývojárov na prekladači GCC[3], ktorý môže byť plne nahradený týmito nástrojmi. Clang je podobne ako LLVM framework licencovaný pod licenciou NCSA, ktorá je kompatibilná s komerčnými produktami.

Clang si udržiava viac informácií počas procesu prekladu než GCC a zachováva si celkovú formu pôvodného kódu. Cieľom je jednoduchšie mapovanie chýb späť na pôvodný zdroj. Chybové hlásenia Clangu sú zamerané na to, aby boli detailnejšie a špecifickejšie. Ponúka strojovú formu chybových hlásení určenú pre integrované vývojové prostredia. Modulárny dizajn prekladača umožňuje indexáciu zdrojového textu a kontrolu syntaxe. Derivačný strom je vhodnejší na podporu automatickej refaktorizácie kódu, keďže priamo reprezentuje pôvodný zdrojový kód.

4.4 Optimalizátor

Optimalizátor je zodpovedný za vykonávanie širokej škály transformácií znázornených na obrázku 4.3 pre zrýchlenie a zvýšenie výkonnosti prekladaného programu. Optimalizátor je zvyčajne nezávislý od jazyka a cieľovej architektúry. Vstupom aj výstupom optimalizátora je interná reprezentácia kódu — LLVM IR.



Obrázok 4.3: Prechody optimalizátora[4]

[2] https://clang.llvm.org/
[3] https://gcc.gnu.org/
[4] https://llvm.org/docs/Passes.html

Činnosť priechodov:

- ***Canonicalization***
  Úpravy LLVM IR so zameraním na skalárne optimalizácie.
- ***Scalar Simplification***
  Prevod inštrukcií a toku riadenia na optimálnejšie formy.
- ***Simple Loop Opts***
  Optimalizácie cyklov.
- ***Inliner***
  Vkladanie tiel funkcií do tiel iných funkcií.
- ***Target Specialization***
  Optimalizácie špecifické pre cieľovú architektúru.

Optimalizácie sa implementujú ako prechody (*passes*), ktoré prechádzajú určitou časťou programu, aby zhromaždili informácie alebo transformovali program. Prechody v LLVM je možné rozdeliť do troch kategórií [18]:

- **Analytické prechody**
  Zhromažďujú informácie, ktoré môžu používať iné prechody. Môžu byť použité na účely ladenia alebo vizualizácie programov. Alias analýza sa zaraďuje práve do tejto kategórie.
- **Transformačné prechody**
  Môžu používať alebo zneplatniť analytické prechody. Menia samotný program (eliminácia mŕtveho kódu, šírenie konštánt, rozbalenie cyklov, atď).
- **Nástrojové prechody**
  Obsahujú pomocné akcie, akými sú napríklad získanie základných blokov z modulu alebo zobrazenie grafu toku riadenia programu (CFG).

Optimalizácie v LLVM sú väčšinou rýchlejšie, využívajú menej pamäte, a kvalitou kódu sú porovnateľné prekladaču GCC [3].

4.4.1 Ukážka práce optimalizátora

Vstupom optimalizátora je vnútorná reprezentácia kódu z prednej časti:

```
@.str = private unnamed_addr constant [4 x i8] c"%d\0A\00", align 1
define dso_local i32 @main() #0 {
       %retval = alloca i32, align 4
       %i = alloca i32, align 4
       store i32 0, i32* %retval, align 4
       store i32 0, i32* %i, align 4
       br label %for.cond
for.cond: ; preds = %for.inc, %entry
       %0 = load i32, i32* %i, align 4
       %cmp = icmp slt i32 %0, 5
       br i1 %cmp, label %for.body, label %for.end
for.body: ; preds = %for.cond
       %1 = load i32, i32* %i, align 4
       %call = call i32 (i8*, ...) @printf(i8* getelementptr
              inbounds ([4 x i8], [4 x i8]* @.str, i32 0, i32 0), i32 %1)
       br label %for.inc
for.inc: ; preds = %for.body
       %2 = load i32, i32* %i, align 4
       %inc = add nsw i32 %2, 1
       store i32 %inc, i32* %i, align 4
       br label %for.cond
for.end: ; preds = %for.cond
       %3 = load i32, i32* %retval, align 4
       ret i32 %3
}
```

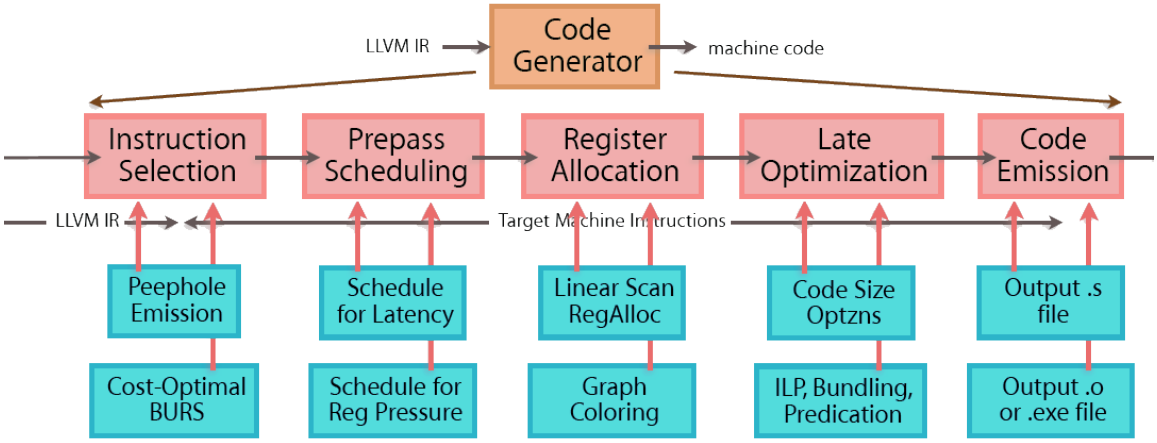
Po použití optimalizačnej úrovne O3 na tento neoptimalizovaný LLVM IR kód je výstupom optimalizátora nasledovný LLVM IR kód:

```
define dso_local i32 @main() local_unnamed_addr #0 {
       %call = tail call i32 (i8*, ...) @printf(i8* getelementptr
              inbounds ([4 x i8], [4 x i8]* @.str, i64 0, i64 0), i32 0)
       %call.1 = tail call i32 (i8*, ...) @printf(i8* getelementptr
              inbounds ([4 x i8], [4 x i8]* @.str, i64 0, i64 0), i32 1)
       %call.2 = tail call i32 (i8*, ...) @printf(i8* getelementptr
              inbounds ([4 x i8], [4 x i8]* @.str, i64 0, i64 0), i32 2)
       %call.3 = tail call i32 (i8*, ...) @printf(i8* getelementptr
              inbounds ([4 x i8], [4 x i8]* @.str, i64 0, i64 0), i32 3)
       %call.4 = tail call i32 (i8*, ...) @printf(i8* getelementptr
              inbounds ([4 x i8], [4 x i8]* @.str, i64 0, i64 0), i32 4)
       ret i32 0
}
```

V tom prípade optimalizátor zistil, že je možné a výhodné cyklus rozvinúť (*loop unrolling*) a túto transformáciu následne vykonal.

4.5 Zadná časť

Zadná časť, tiež známa ako generátor kódu, transformuje LLVM IR z výstupu optimalizátora na cieľovú inštrukčnú sadu. Generátor kódu je zodpovedný za vygenerovanie čo najlepšieho strojového kódu pre danú cieľovú architektúru. Je jednou z najzložitejších častí LLVM.



Obrázok 4.4: Schéma generátora kódu[5]

Generovanie kódu sa skladá z niekoľkých fáz:

- ***Instruction Selection***
  Výber inštrukcií podľa vzorov alebo ohodnotenia.
- ***Prepass Scheduling***
  Plánovanie kódu, zmena poradia inštrukcií.
- ***Register Allocation***
  Priradenie registrov k premenným v programe.
- ***Late Optimalization***
  Neskoršie optimalizácie na zníženie veľkosti kódu, optimalizácie podľa predikátov.
- ***Code Emission***
  Výpis strojového kódu do súboru a tvorba binárneho spustiteľného súboru.

LLVM od verzie 3.4 podporuje mnoho inštrukčných sád: ARM, Qualcomm Hexagon, MIPS, Nvidia Parallel Thread Execution, PowerPC, AMD TeraScale, AMD Graphics Core Next , SPARC, z/Architecture, x86, x86-64 a XCore. Niektoré funkcie nie sú dostupné na niektorých platformách. Väčšina z nich je však dostupná pre x86, x86-64, z/Architecture, ARM a PowerPC [6].

[5] https://blog.csdn.net/dashuniuniu/article/details/50385528
[6] http://llvm.org/docs/CodeGenerator.html#target-feature-matrix

Kapitola 5

Alias analýza v LLVM

Kapitola sa zameria na samotnú implementáciu algoritmov alias analýzy v LLVM. Obsahuje ich porovnanie z pohľadu presnosti výsledkov a rýchlosti ich získavania. Poskytuje podrobnosti o ich implementácii v LLVM. Taktiež popisuje spôsob implementácie interprocedurálnej analýzy v LLVM. V závere kapitoly sú uvedené aktuálne problémy alias analýzy v LLVM spolu s možnosťami, ako by bolo možné tieto problémy vyriešiť v budúcnosti.

5.1 Dostupné algoritmy

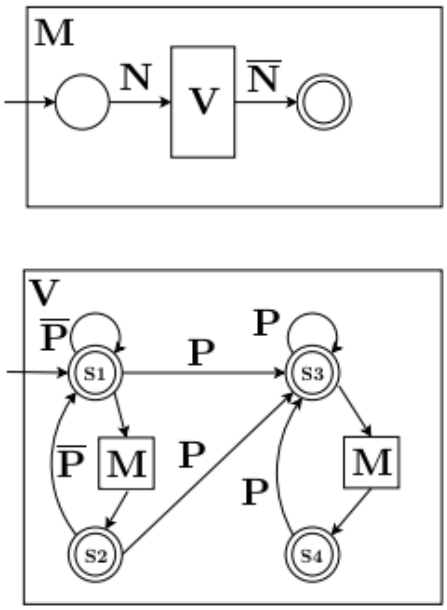
V LLVM sú implementované dva *cfl-aa* prechody, ktoré vykonávajú alias analýzu založenú na probléme dosiahnuteľnosti pre bezkontextový jazyk: *cfl-steens-aa* a *cfl-anders-aa*. V princípe su skoro identické, líšia sa v tom, ako prebieha výpočet dosiahnuteľnosti pre bezkontextový jazyk. Hlavný dôvod rozdelenia na *cfl-steens-aa* a *cfl-anders-aa* je ten, že každý algoritmus ma svoj špecifický pomer efektívnosti voči presnosti.

Oba *cfl-aa* prechody analyzujú každú funkciu oddelene. V rámci analýzy funkcie sa kompletné informácie o aliasoch vypočítajú dopredu. Následné zisťovanie, či dva ukazovatele sú aliasmi, môže byť zodpovedané veľmi rýchlo. Analýza naprieč hraniciam funkcií využíva stratégiu založenú na získavaní výstupov na dopyty (*queries*). Táto stratégia je tiež známa ako *demand-driven* stratégia.

Procedúra zaisťujúca súlad s špecifikáciami popísanými v podkapitole 2.2 je implementovaná v triede `CFLGraphBuilder`, ktorá sa nachádza v *lib/Analysis/CFLGraph.h*. Oba *cfl-aa* prechody sú založené na vytváraní PEG štruktúry (v LLVM ako `CFLGraph`).

5.1.1 Alias analýza založená na Andersenovom algoritme

Prechod *cfl-anders-aa* si na rozdiel od *cfl-steens-aa* volí iný kompromis, kde presnosť je dôležitejšia pred rýchlosťou analýzy. V *cfl-anders-aa* sa používa pôvodná gramatika 2.2.2, nie sú vytvárané žiadne odhady. Dôvodom implementácie *cfl-anders-aa* je aj fakt, že aj keď je teoretická výpočtová náročnosť dosiahnuteľnosti pre bezkontextový jazyk kubická, v praxi takmer nedochádza k dosiahnutiu najhoršieho prípadu [21]. Taktiež so správnou množinou implementovaných optimalizácií [7] dochádza k rýchlejšiu výpočtu a zároveň je presnosť zachovaná [8].



Obrázok 5.1: Hierarchický stavový automat [27]

Algoritmus použitý pre *cfl-anders-aa* vychádza z už existujúcej práce [27]. Základnou myšlienkou algoritmu je výpočet tradičného tranzitívneho uzáveru za účelom šírenia informácií o dosiahnuteľnosti pre bezkontextový jazyk. K zastaveniu šírenia dochádza, ak sa zistí, že popisy hrán nevytvárajú reťazec generovaný bezkontextovou gramatikou. Gramatika je transformovaná do hierarchického stavového automatu. Rozhodnúť o šírení dosiahnuteľnosti pozdĺž hrany je možné pomocou kontroly, či popis na hrane môže spôsobiť presun z jedného platného stavu do druhého.

Implementácia Andersenovho algoritmu

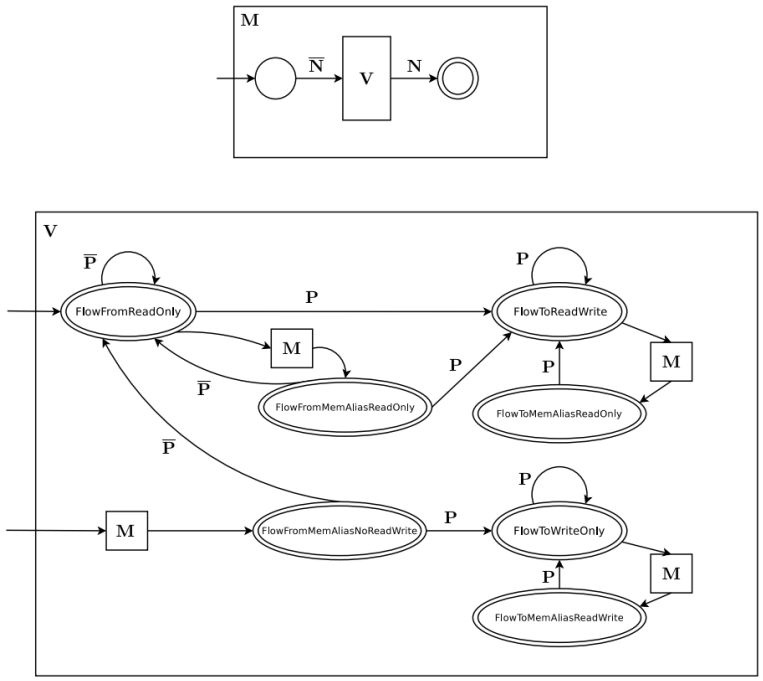
Algoritmus implementovaný v LLVM je založený na schéme rekurzívneho stavového automatu predstaveného v [27]. Algoritmus je založený na zjednocovaní a používa techniku vytvárania sumárov funkcií. Hlavnou myšlienku je rozšíriť algoritmus tradičného tranzitívneho uzáveru o vykonávanie CFL porovnávania (*CFL matching*). Namiesto zaznamenávania či hodnota X je dosiahnuteľná z hodnoty Y, dochádza k sledovaniu, či hodnota X je dosiahnuteľná z hodnoty Y v stave Z. Stav udáva aktuálnu pozíciu v procese CFL porovnávania. U CFL porovnávania sa výberovo rozbalí tranzitívny uzáver zahodením hrán, ktoré nie sú rozpoznané stavovým automatom. V aktuálnej implementácii[1] sú dve odlišnosti oproti pôvodnému algoritmu.

Prvou odlišnosťou je, že algoritmus počíta všetky alias páry po vytvorení CFL grafu, no autori práce hovoria o výpočte na vyžiadanie. CFL graf je pomocná dátová štruktúra používaná CFL založenou alias analýzou na popis tokovo necitlivých chovaní súvisiacich s ukazovateľmi. Hlavný zmysel tohto grafu je abstrakcia faktov a ich transformácia do formy, ktorá je ľahko spracovateľná CFL analýzami.

[1] https://github.com/llvm-mirror/llvm/blob/release_60/lib/Analysis/CFLAndersAliasAnalysis.cpp

Algoritmus bol oproti [27] zjednodušený a kvôli zvýšenej komplexnosti kódu a vyšších pamäťových požiadavkách. V prípade potreby v budúcnosti je podľa vývojárov ale možný prechod na tento algoritmus.

Druhou odlišnosťou je fakt, že autori práce zavádzajú stavový automat, ktorý nerozlišuje medzi čítaním a zápisom hodnôt. Napríklad, ak hodnota Y je dosiahnuteľná z hodnoty X v stave *S3*, môže to byť prípad že X sa zapisuje do Y alebo aj že tretia hodnota Z sa zapisuje do X aj Y. Za účelom odlíšenia týchto prípadov (ktoré sú dôležité pri vytváraní sumárov funkcií) došlo v implementácií k zdvojeniu niektorých stavov v navrhovanom stavovom automate. Na obrázku 5.2 je zobrazený upravený stavový automat, ktorý bol implementovaný v LLVM. Zdvojenie stavov nemení množinu reťazcov, ktorú automat prijíma. Tento spôsob implementácie poskytuje informácie o tom, z ktorej hodnoty sa číta a do ktorej sa zapisuje.



Obrázok 5.2: Hierarchický stavový automat pre *cfl-anders-aa* [3]

5.1.2 Alias analýza založená na Steensgardovom algoritme

Prechod *cfl-steens-aa* využíva jednoduché pozorovanie na drastické zlepšenie najhoršieho prípadu časovej zložitosti. Ak v gramatike pre alias analýzu prezentovanej v podkapitole 2.2.2 dôjde k ignorovaniu neterminálov P a $\overline{P}$, potom nová gramatika v podstate popisuje jazyk *Dyck* (jazyk, ktorý rozpoznáva len vyvážené zátvorky). Ak je *Dyck* základným bezkontextovým jazykom, dosiahnuteľnosť pre bezkontextový jazyk môže byť vypočítaná v lineárnom (ak graf je strom) alebo lineárne logaritmickom (ak graf je všeobecnejší než strom) čase so správnym výberom dátovej štruktúry [25].

Spôsob, akým *cfl-steens-aa* ignoruje neterminály P a $\overline{P}$ v pôvodnej gramatike, spočíva v zlúčení uzlov v `CFLGraph`. Ak dva uzly v grafe sú prepojené P hranou alebo $\overline{P}$ hranou, tak sú následne zlúčené do jedného uzla. Tento proces sa opakuje až kým nezostane žiadna P alebo $\overline{P}$ hrana. Následne sa použije algoritmus so zložitosťou $\mathcal{O}(n)$ [25] na výpočet do-

siahnuteľnosti pre bezkontextový jazyk *Dyck* pre zjednodušený graf. Algoritmus nakoniec zoskupí ukazovatele do ekvivalentných množín. To, či dva ukazovatele môžu byť aliasmi sa dá následne zistiť podľa toho, či sú v rovnakej ekvivalentnej množine.

Tento trik so zlučovaním uzlov v podstate odhaduje neterminál V a neterminál M v pôvodnej gramatike ich tranzitívnym uzáverom. Keďže dochádza k aproximácií dosiahnuteľnosti pre bezkontextový jazyk podľa pôvodnej gramatiky 2.2.2, potenciálnym výstupom analýzy je, že dva ukazovatele môžu byť aliasmi, no podľa pôvodnej gramatiky aliasmi nie sú. Z hľadiska analýzy presnosti bolo preukázané [27], že odhad neterminálov V a M ich tranzitívnym uzáverom vedie k analýze ekvivalentnej Steensgardovmu algoritmu založenému na zjednocovaní.

Implementácia Steensgardovho algoritmu

Steensgardov algoritmus je algoritmus alias analýzy, ktorý v implementovanej verzii[2] používa techniku vytvárania sumárov funkcií. Použitý algoritmus je spojením algoritmu opísaného v [27] a algoritmu z [25]. Princíp tohto novo zloženého algoritmu hovorí o vytváraní grafu použití premennej, kde každý uzol je umiestnenie v pamäti a každá hrana je akcia vykonaná nad týmto umiestnením. Medzi akcie sa zaraduje dereferencia, referencia a priradenie. Dve premenné sú považované za aliasy, ak je možné dosiahnuť uzol jednej hodnoty z uzlu inej hodnoty. Jazyk, ktorý je vytváraný spojením všetkých popisov hrán (akcií), je generovaný bezkontextovou gramatikou.

Pretože tento algoritmus vyžaduje prehľadávanie grafu pri každom dopyte, sa za pomoci algoritmu uvedeného v [25] tento graf transformuje do množín premenných, ktoré sa môžu byť aliasmi v približne $n \cdot log(n)$ čase, kde n je počet premenných. Cieľom transformácie je získať výstupy na dopyty v konštantnom čase. Presnosť tejto analýzy je približne rovnaká ako jednoúrovňový kontextovo citlivý Steensgardov algoritmus alias analýzy.

5.1.3 Typovo založená alias analýza

V LLVM je taktiež implementovaná typovo založená alias analýza[3] založená na metadátach. Keďže v LLVM pamäť nemá typy, samotný typový systém v LLVM nie je vhodný pre TBAA. Riešením tohto problému je pridávanie metadát do vnútornej reprezentácie kódu. Metadáta slúžia na opis typového systému vysokoúrovňového jazyka. Pomocou metadát je možné implementovať klasickú typovo založenú alias analýzu pre C/C++. Taktiež umožňujú implementáciu vlastnej TBAA aj pre iné jazyky.

LLVM podporuje dva typy formátov metadát: skalárna TBAA a TBAA založená na cestách v rámci štruktúr (struct-path aware). Uprednostňovaná je TBAA založená na cestách v rámci štruktúr a v budúcnosti môže byť podľa poznámok vývojárov LLVM v implementácii TBAA podpora pre skalárnu TBAA odstránená.

5.2 Porovnanie dostupných algoritmov

Implementácie Andersenovho a Steensgardovho algoritmu použité v LLVM (popísané v podkapitole 5.1) boli porovnávané na množine testovacích programov určených na vyhodnocovanie/porovnávanie algoritmov na alias analýzu programov s ukazovateľmi. Tieto programy

[2] https://github.com/llvm-mirror/llvm/blob/release_60/lib/Analysis/CFLSteensAliasAnalysis.cpp

[3] https://github.com/llvm-mirror/llvm/blob/release_60/lib/Analysis/TypeBasedAliasAnalysis.cpp

obsahujú pokročilú prácu s ukazovateľmi a aritmetikou nad nimi. Porovnávanie implementácií prebiehalo s LLVM/Clang vo verzii 6.0 na architektúre x86-64.

Pre každý kód bolo potrebné najskôr vygenerovať jeho vnútornú reprezentáciu (LLVM IR), ktorá je vstupom do optimalizátora. Generovanie LLVM IR bolo vykonané za pomoci nasledovného príkazu:

```
clang -S -emit-llvm source.c
```

Výstupom je LLVM IR súbor s príponou *.ll*. Na tento súbor s LLVM IR je následne aplikovaný LLVM optimalizátor — *opt*. Aby sme získali skutočné výsledky pre daný algoritmus, je potrebné vypnúť základnú alias analýzu v LLVM — *basicaa* — pomocou prepínača *-disable-basicaa*. Pre zobrazenie štatistík o výsledkoch alias analýzy sa používa prepínač *–aa-eval*. Prepínač *-time-passes* zas umožňuje zobraziť informácie o čase strávenom v jednotlivých prechodoch. Za účelom merania je taktiež potrebné použiť prepínač *–disable-output* pre vypnutie binárneho výstupu optimalizátora. Posledný prepínač rozhoduje, ktorý algoritmus alias analýzy sa aplikuje na LLVM IR. Pre Andersenov algoritmus je to prepínač *-cfl-anders-aa* a pre Steensgardov algoritmus *-cfl-steens-aa*.

Za účelom získania výsledku pre daný algoritmus alias analýzy boli použité nasledovné príkazy:

```
opt --disable-basicaa --aa-eval -cfl-anders-aa --disable-output out.ll
opt --disable-basicaa --aa-eval -cfl-steens-aa --disable-output out.ll
```

Typický formát štatistík o výsledkoch alias analýzy vyzerá nasledovne:

```
===== Alias Analysis Evaluator Report =====
1381 Total Alias Queries Performed
937 no alias responses (67.8%)
444 may alias responses (32.1%)
0 partial alias responses (0.0%)
0 must alias responses (0.0%)
Alias Analysis Evaluator Pointer Alias Summary: 67%/32%/0%/0%
1636 Total ModRef Queries Performed
0 no mod/ref responses (0.0%)
0 mod responses (0.0%)
0 ref responses (0.0%)
1636 mod & ref responses (100.0%)
Alias Analysis Evaluator Mod/Ref Summary: 0%/0%/0%/100%
```

Vo výstupe je prehľadne uvedený počet skúmaných párov a ďalej sú uvedené možné výstupy alias analýzy (viď podkapitola 2.6) a počet týchto výstupov pre analyzovaný LLVM IR kód. Informácie z tohoto výstupu boli následné použité pre porovnanie implementácií algoritmov alias analýzy v LLVM.

Samotné meranie sa vykonávalo na množine programov obsahujúcich rôzne použitie ukazovateľov. Jednalo sa o prevažne programy, ktoré implementujú rôzne známe algoritmy v jazyku C: CRC32, FFT, MD5, SHA3, ARC4, ABM4. V tejto množine programov sa taktiež nachádzali kódy súvisiace s implementáciou matíc a operácií nad nimi a jednosmerne viazaného zoznamu, ktorý je často používanou dátovou štruktúrou vo viacerých programoch.

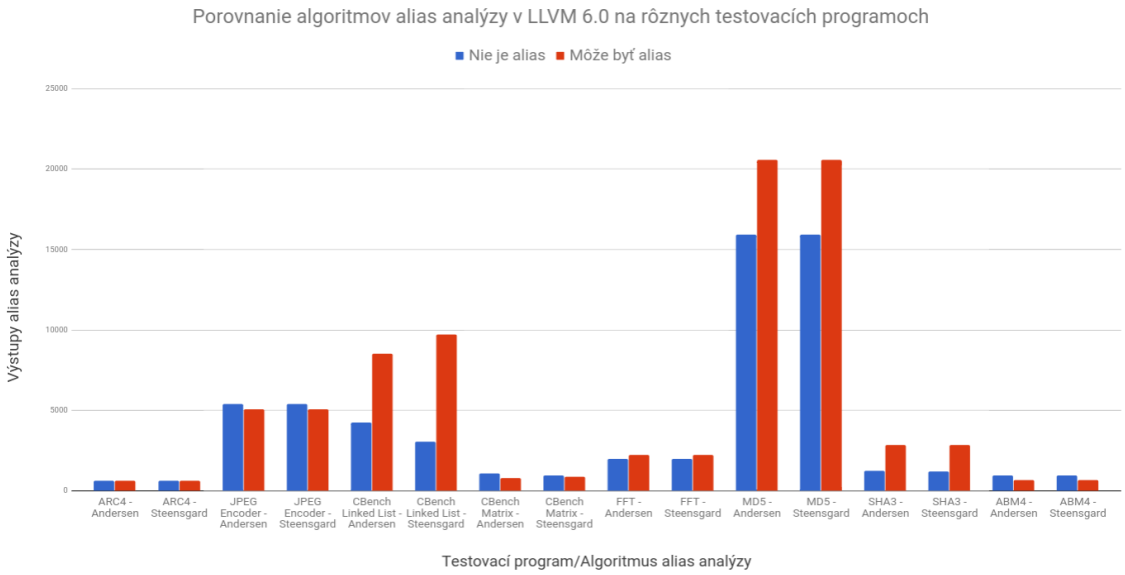
Program	Nie je alias	Môže byť alias
ARC4	622	621
JPEG Encoder	5390	5050
CBench Linked List	4245	8536
CBench Matrix	1073	781
FFT	1975	2211
MD5	15937	20582
SHA3	1225	2850
ABM4	964	651

Tabuľka 5.1: Výsledky Andersenovho algoritmu alias analýzy

Program	Nie je alias	Môže byť alias
ARC4	620	623
JPEG Encoder	5384	5056
CBench Linked List	3068	9713
CBench Matrix	974	880
FFT	1975	2211
MD5	15937	20582
SHA3	1217	2858
ABM4	964	651

Tabuľka 5.2: Výsledky Steensgardovho algoritmu alias analýzy

Tabuľky 5.1 a 5.2 obsahujú výsledky meraní pre jednotlivé algoritmy. U každého programu u uvedený počet *nie je alias* a *môže byť alias* výstupov.



Obrázok 5.3: Porovnanie implementácie algoritmov alias analýzy v LLVM

Výsledky oboch algoritmov sú za účelom jednoduchšieho porovnávania reprezentované grafom na obrázku 5.3 obsahujúcim výsledky (počty *nie je alias* a *môže byť alias* výstupov) porovnávaných algoritmov na testovaných programoch.

Podľa grafu je možné rozhodnúť, že výstupom Andersenovho algoritmu sú presnejšie výsledky (viac jasných *nie je alias* výstupov) v porovnaní so Steensgardovým algoritmom (viac špekulatívnych *môže byť alias* výstupov). Tento poznatok z merania potvrdzuje teoretické predpoklady o presnosti porovnávaných algoritmov. Ako je možné vidieť u programov, ktoré značne využívajú ukazovatele, rozdiely medzi algoritmami môžeme vo všeobecnosti označiť za takmer zanedbateľné s ohľadom na väčší rozsah kódu. Keďže meranie prebiehalo na menších zdrojových súboroch, oba algoritmy poskytli výsledky približne za rovnakú dobu, prípadne bola zaznamenaná kratšia doba u Steensgardovho algoritmu. Vykonané porovnávanie potvrdzuje poznatok z praxe tykajúci sa výhodného pomeru presnosti k dobe trvania analýzy práve u Steensgardovho algoritmu.

5.3 Transformácia LLVM IR do PEG

Určité aspekty v LLVM IR nemožno správne zakódovať do grafu programových výrazov (PEG), v LLVM tzv. `CFLGraph`:

- LLVM umožňuje pretypovania medzi ukazovateľmi a celými číslami, ale `CFLGraph` obsahuje len ukazovatele ako uzly.
- LLVM modul môže obsahovať globálne premenné, ktoré nepatria do žiadnej funkcie, a preto nie sú modelované v `CFLGraph`.
- LLVM modul môže obsahovať deklarované, ale nie definované funkcie. `CFLGraph` pre len deklarované funkcie nemá možnosť zistiť ich vedľajšie efekty, a preto o nich musí uvažovať ako o funkciách, ktoré môžu zapisovať čokoľvek na akékoľvek miesto v pamäti, ku ktorému majú prístup.

Pre riešenie vyššie uvedených vlastností jazyka má každý uzol v `CFLGraph` viaceré atribúty, prípadne nemusí mať žiadny. Atribúty slúžia ako únikové cesty z algoritmu pre dosiahnuteľnosť pre bezkontextový jazyk. Ak pri analýze vzťahu medzi ukazovateľmi p a q je jeden z nich označený jedným alebo viacerými atribútmi, alias analýza preskočí výpočet dosiahnuteľnosti pre bezkontextový jazyk a vráti výsledok podľa tabuľky 5.4. Atribúty majú tiež vlastnosť, že sú šírené nadol: ak je ukazovateľ p označený určitými atribútmi, potom všetky ukazovatele, na ktoré tranzitívne ukazuje ukazovateľ p, sú označené rovnakou množinou atribútov.

	Bez atribútov	**AttrEscaped**	**AttrGlobal**	**AttrArgument**	**AttrCaller**	**AttrUnknown**
Bez atribútov	CFL výpočet	CFL výpočet	nie je alias	nie je alias	nie je alias	nie je alias
AttrEscaped	CFL výpočet	CFL výpočet	nie je alias	nie je alias	môže byť alias	môže byť alias
AttrGlobal	nie je alias	nie je alias	môže byť alias	môže byť alias	môže byť alias	môže byť alias
AttrArgument	nie je alias	nie je alias	môže byť alias	môže byť alias	môže byť alias	môže byť alias
AttrCaller	nie je alias	môže byť alias	môže byť alias	môže byť alias	môže byť alias	môže byť alias
AttrUnknown	nie je alias	môže byť alias	môže byť alias	môže byť alias	môže byť alias	môže byť alias

Obrázok 5.4: Pravidlá pre spracovanie atribútov [3]

V tabuľke 5.4 riadky predstavujú atribúty jedného ukazovateľa, stĺpce atribúty iného ukazovateľa. Každá bunka predstavuje odpoveď alias analýzy podľa atribútov dvoch ukazovateľov. Ak má každý ukazovateľ viac ako jeden atribút, kombinácie, ktoré vedú k *môže byť alias* výstupu, majú prioritu nad tými, ktoré vedú k *nie je alias* výstupu.

Existuje päť rôznych atribútov: `AttrEscaped`, `AttrCaller`, `AttrGlobal`, `AttrArgument` a `AttrUnknown`. Ich významy sú nasledovné:

- Ukazovatele, ktoré sú alokované lokálne, ale sú priradené do ďalšej hodnoty, ktorá nie je modelovaná analýzou, sú označené atribútom `AttrEscaped`. Patria sem ukazovatele, ktoré sú pretypované na celé číslo, ale aj ukazovatele, ktoré sú argumentami volania funkcie, o ktorej nie sú žiadne informácie.
- Formálne parametre funkcií sú označené atribútom `AttrArgument`. Hodnoty, na ktoré ukazujú tieto parametre sú označené atribútom `AttrCaller`.
- Globálne hodnoty sú označené atribútom `AttrGlobal`.
- Všetky hodnoty, ktoré pochádzajú zo zdroja, o ktorom nie sú žiadne informácie, sú označené atribútom `AttrUnknown`.

5.4 Inštrukcie súvisiace s alias analýzou

Táto podkapitola popisuje LLVM IR inštrukcie, ktoré pracujú s ukazovateľmi.

Inštrukcia Store

Inštrukcia `Store` sa používa na zápis hodnoty do pamäti. Syntax inštrukcie je:

```
store <ty> <value>, <ty>* <pointer>
```

Zdrojový kód v jazyku C:

```
int x;
int * p;
p = &x;
```

LLVM IR pre tento kód:

```
%x = alloca i32, align 4
%p = alloca i32 *, align 8
store i32 * %x, i32 ** %p, align 8
```

Na treťom riadku LLVM IR kódu dochádza k zápisu adresy x do ukazovateľa p. Ukazovateľ p ukazuje na x. Inštrukcia `Store` je ekvivalentná obmedzeniu $*a = b$.

Inštrukcia Load

Inštrukcia `Load` sa používa na načítanie hodnoty z pamäti. Syntax inštrukcie je:

```
<result> = load <ty>, <ty>* <pointer>
```

Zdrojový kód v jazyku C:

```
int *p;
*p = 0;
```

LLVM IR pre tento kód:

```
%p = alloca i32 *, align 8
%0 = load i32 ** %p, align 8
store i32 0, i32 * %0, align 4
```

Na druhom riadku LLVM IR kódu sa do %0 uloží adresa pamäte, kam ukazuje ukazovateľ p. Na treťom riadku dochádza k zápisu konštantnej hodnoty 0 do %0. Označenie %0 v tomto príklade je konceptuálne ekvivalentné s dereferenciou ukazovateľa p v zdrojovom jazyku. Inštrukciu `Load` sa považuje za jednoúrovňovú dereferenciu a je ekvivalentná obmedzeniu $a = *b$.

Inštrukcia Call

Inštrukcia `Call` sa používa na volanie funkcií. Syntax inštrukcie je:

```
<result> = call <ty> <fnptrval>(<function args>)
```

Identifikátor *result* je návratová hodnota funkcie typu *ty*, *fnptrval* je identifikátor volanej funkcie a *function args* je zoznam argumentov, s ktorými je funkcia volaná. Argumenty sa v jazyku C predávajú hodnotou. Tento spôsob predávania argumentov je ekvivalentný obmedzeniu $a = b$.

Inštrukcia GetElementPtr

Inštrukcia `GetElementPtr` sa používa na získanie adresy položky v agregovanej dátovej štruktúre. Vykonáva len výpočet adresy a nepristupuje k pamäti. Syntax inštrukcie je:

```
<result> = getelementptr inbounds <ty>* <ptrval>{, <ty> <idx>}*
```

Prvý identifikátor *ty* definuje typ agregovanej dátovej štruktúry *ptrval*. Nasledujúci zoznam *ty idx* označuje indexy požadovaných položiek. Vypočítaná adresa sa ukladá do *result*. Ak sa rozlišuje rozdiel medzi *result* a *ptrval*, je táto inštrukcia ekvivalentná obmedzeniu $a = b$ a dochádza k zlúčeniu *result* s *ptrval*.

Inštrukcia PHI

LLVM IR používa SSA (static single assignment) formu na reprezentáciu premenných. SSA forma je založená na predpoklade, že premenné sú priradené v presnom jednom mieste v programe [16] a je dosiahnutá rozdelením existujúcich premenných na mnohé variácie.

Zdrojový kód v jazyku C:

```
x = 1;
x = 2;
y = x;
```

SSA forma pre tento kód:

```
x1 = 1;
x2 = 2;
y1 = x2;
```

SSA forma umožňuje prekladačom vykonávať mnoho druhov optimalizácií (šírenie konštánt, eliminácia mŕtveho kódu).

`PHI` uzol má v LLVM IR nasledovnú syntax:

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

Identifikátor *result* je nová SSA premenná, *ty* je typ premennej a *[val0, label0]* je zoznam SSA premien s označeniami, ktoré zodpovedajú riadiacim tokom každej SSA premennej. Inštrukcia `PHI` je ekvivalentná obmedzeniu $a = b$. Dochádza k zlúčeniu všetkých SSA premenných s *result*.

Inštrukcia BitCast

Inštrukcia `BitCast` sa používa na zmenu typu premennej na iný typ bez toho, aby zmenila hodnotu tejto premennej. Syntax inštrukcie je:

```
<result> = bitcast <ty> <value> to <ty2>
```

Identifikátor *result* je premenná nového typu, *ty* je pôvodný typ, *value* je premenná na pretypovanie a *ty2* je nový typ. Inštrukcia `BitCast` je ekvivalentná obmedzeniu $a = b$. Dochádza k zlúčeniu *value* s *result*.

Inštrukcia IntToPtr

Inštrukcia `IntToPtr` sa taktiež používa na pretypovanie, konkrétne ide o pretypovanie celého čísla na ukazovateľ. Syntax inštrukcie je:

```
<result> = inttoptr <ty> <value> to <ty2>
```

Identifikátor *result* je nový ukazovateľ, *ty* je pôvodný typ, *value* je celé číslo a *ty2* je nový typ. Inštrukcia `IntToPtr` je ekvivalentná obmedzeniu $a = b$. Dochádza k zlúčeniu *value* s *result*.

Inštrukcia Select

Inštrukcia `Select` sa používa na výber hodnoty na základe podmienky bez vetvenia. Syntax inštrukcie je:

```
<result> = select selty <cond>, <ty> <val1>, <ty> <val2>
```

Ak je podmienka *cond* pravdivá, do *result* sa uloží hodnota *val1*, inak *val2*. Keďže výsledok podmienky je počas statickej analýzy neznámy, inštrukcia je ekvivalentná obmedzeniu $a = b$. Dochádza k zlúčeniu *result*, *val1* a *val2*.

5.5 Interprocedurálna analýza

Oba *cfl-aa* prechody používajú prístup zdola nahor pri analýze naprieč hraniciam funkcií Každá funkcia sa najprv analyzuje nezávisle a intraprocedurálne. Ukladajú sa externe viditeľné efekty do sumáru danej funkcie. Formát sumáru funkcie má nasledujúcu [3]:

$$
\begin{array}{rcl}
\mathit{FunctionSummary} & ::= & \mathit{ExternallyVisibleEffect}^{*} \\
\mathit{ExternallyVisibleEffect} & ::= & \mathit{InterfaceValue} = \mathit{InterfaceValue}' \\
 & | & \mathit{InterfaceValue} \leftarrow \mathit{Attribute} \\
\mathit{InterfaceValue} & ::= & \textbf{ReturnValue} \\
 & | & \textbf{Parameter}(i) \\
 & | & *\mathit{InterfaceValue}' \\
\mathit{Attribute} & ::= & \textbf{AttrEscaped} \\
 & | & \textbf{AttrGlobal} \\
 & | & \textbf{AttrUnknown}
\end{array}
$$

Obrázok 5.5: Syntax sumáru funkcie

Sumár sa vytvára na každom mieste, kde dochádza k volaniu funkcie. *ReturnValue* sa nahrádza pravou stranou miesta volania funkcie, *Parameter(i)* sa nahrádza i-tým argumentom v mieste volania funkcie. Návratová hodnota a parametre sa označujú správnou množinou atribútov. Nakoniec dochádza k začleneniu vytvoreného hodnotenia funkcie do grafu programových výrazov (PEG) volajúcej funkcie. Interprocedurálna časť analýzy je prevádzaná vo fáze tvorby grafu programových výrazov. Výpočet problému dosiahnuteľnosti pre bezkontextový jazyk neberie do úvahy sumáre funkcií, a preto nie je potrebné vykonať žiadne zmeny pre fungovanie interprocedurálnej analýzy.

5.6 Problémy dostupných algoritmov

Plne funkčný prechod alias analýzy, tak ako aj iné analytické prechody, vyžadujú množstvo vynaloženej práce a prechody *cfl-steens-aa* a *cfl-anders-aa* nie sú výnimkami. V tejto podkapitole sú uvedené aktuálne problémy alias analýzy v LLVM [3] spolu s možnými riešeniami týchto problémov.

5.6.1 Pomalý výpočet cfl-anders-aa

Prechod *cfl-anders-aa* je od *cfl-steens-aa* pomalší a v niektorých prípadoch sa neskončí ani v akceptovateľnej dobe. Dôvodom je fakt, že na rozdiel *cfl-steens-aa*, kde na tomto prechode pracovalo mnoho ľudí, bol prechod *cfl-anders-aa* implementovaný a spravovaný jednou osobou [3]. Je teda menej odladený na chyby a suboptimálny.

Ako kontextovo citlivá analýza založená na zlučovaní má *cfl-anders-aa* vyššie výpočtové nároky, a preto je vhodné použiť rôzne formy optimalizácií [7, 8] a výpočty na vyžiadanie [27], ktoré zvyšujú rozšíriteľnosť. V súčasnosti žiadne z týchto optimalizácií nie sú implementované, keďže v skorších fázach implementácie bol kladený dôraz na správnosť výsledkov a implementácie než na rýchlosť výpočtu. Časom by bolo vhodné pridať tieto optimalizácie do aktuálnej implementácie prechodu *cfl-anders-aa*.

5.6.2 Chýbajúca citlivosť na položky u cfl-aa

V súčasnosti oba *cfl-aa* prechody spracovávajú štruktúry ako veľké bloky dát a nerozlišujú medzi položkami v rámci rovnakej štruktúry. Zvyčajne však platí, že ak štruktúra obsahuje viac ako jednu položku typu ukazovateľ, tieto položky majú tendenciu ukazovať na rôzne miesta v pamäti. Takéto spracovanie štruktúr spôsobuje značnú stratu presnosti, čo môže byť jeden z dôvodov, prečo je *cfl-aa* často prekonaná prechodom *basicaa* z hľadiska presnosti vo vyhodnocovaní. Pre pridanie citlivosti na položky je potrebné rozšíriť `CFLGraph` o informáciu obsahujúcu posun (offset) položky v rámci štruktúry a upraviť logiku šírenia u dosiahnuteľnosti pre bezkontextový jazyk.

Citlivosť na položky štruktúr by bolo vhodnejšie implementovať do *cfl-anders-aa* než do *cfl-steens-aa* a to nielen z dôvodu samotného princípu *cfl-steens-aa*, ktorý je založený na zjednocovaní, ale aj z dôvodu, že je vhodnejšie zvýšiť presnosť výsledkov u *cfl-anders-aa*, keďže *cfl-anders-aa* podáva presnejšie výsledky ako *cfl-steens-aa*.

5.6.3 Nepresná interprocedurálna analýza pre rekurzívne volania funkcií

V podkapitole 5.5 bolo spomenuté, že oba *cfl-aa* prechody analyzujú funkcie v poradí zdola nahor: volané funkcie sú analyzované pred volajúcou funkciou. Ak v programe existujú rekurzívne volania (silne súvislé komponenty v grafe volaní), volajúca funkcia môže byť volanou funkciou. V takomto prípade sú potrebné ďalšie pravidlá pre definovanie poradia analýzy funkcií.

Súčasné riešenie spočíva v prerušení rekurzívneho cyklu. Funkcia v rekurzívnej slučke je konzervatívne považovaná za externe definovanú funkciu (môže zapisovať čokoľvek na akékoľvek miesto pamäte, ku ktorému má prístup). Výber funkcie závisí od toho, ktorá funkcia sa analyzuje ako prvá. Aktuálne implementovaný prístup je logicky platný, ale môže byť nepresný, pretože neexistuje žiadna záruka, že kritické funkcie nebudú vyhodnotené konzervatívne. Presnejším prístupom je začať analýzu s predpokladom, že všetky funkcie nemajú vedľajšie efekty (side effects). Následne po analýze tela funkcie, kde by mohli byť objavené vedľajšie efekty, by sa tieto vedľajšie efekty šírili späť volajúcim funkciám. Ak by táto propagácia pridala viac vedľajších efektov volajúcej funkcií, došlo by k ďalšej propagácií na funkcie, ktoré volajú túto danú funkciu. Celý proces by sa opakoval až kým by sa nedosiahol pevný bod.

Táto stratégia založená na pevnom bode môže byť výpočtovo náročnejšia, pretože už neexistuje záruka, že každá funkcia bude analyzovaná iba raz. Ak sa v grafe volaní funkcií nachádzajú cykly, funkcie v tomto grafe môžu byť analyzované viac ako raz kvôli šíreniu informácií o vedľajších efektoch. Otázkou zostáva, či zvýšenie presnosti stojí sa zvýšenie výpočtovej náročnosti.

5.6.4 Chýbajúca podpora modref dopytov u cfl-aa

Rozhranie `AliasAnalysis` v LLVM obsahuje množstvo API funkcií. Prechody *cfl-anders-aa* a *cfl-steens-aa* podporujú len `alias()` dopyty. Avšak metóda `alias()` nie je jediný spôsob, ako je možné komunikovať s `AliasAnalysis`. Metóda `getModRefInfo()` poskytuje informácie o tom, či dané miesto volania funkcie (alebo daná volaná funkcia) môže čítať alebo zapisovať do určitých miest v pamäti. Táto informácia je veľmi dôležitá pre ďalšie optimalizačné prechody, ktoré sa týkajú inštrukcií pre volanie alebo invokáciu (spustenie) funkcií. Bez `getModRefInfo()` by tie prechody nepresunuli alebo nevymazali nadbytočné invokácie funkcií, napriek tomu, že alias analýza by vedela, že invokácia je bez vedľajších efektov.

Pre *cfl-aa* analýzy existovala naivná podpora pre *modref*. Metóda `getModRefInfo()` prechádzala vytvoreniu sumárov funkcií a podľa toho odvodzovali *modref* informácie. Podpora bola neskôr odstránená kvôli zisteniu, že použitie sumárov funkcií na odvodenie *modref* informácií nebolo logicky správne. Sumáre pre alias analýzu obsahovali len hodnoty typu ukazovateľ, no logicky správna *modref* analýza musí tiež pracovať s čítaniami a zápismi hodnôt, ktoré nie sú typu ukazovateľ. Ak niektorá funkcia pristupuje k hodnote, ktorá nie je ukazovateľom, skutočnosť sa neodrazí vo výsledku alias analýzy, ale mala by byť zachytená *modref* analýzou.

Modref analýza môže používať výsledky alias analýzy na zisťovanie miest v pamäti, na ktoré sa pristupuje. Koncepčne však ide o odlišnú analýzu, ktorá musí brať do úvahy širší rozsah hodnôt, a preto vyžaduje komplexnejšiu implementáciu v porovnaní s vyššie spomenutou naivnou implementáciou.

5.6.5 Spolupráca s basicaa

Prechod *basicaa* je prekvapivo efektívny a presný, čo bolo preukázané viacerými meraniami [3]. Je založený na *demand-driven* stratégii (vykonávanie výpočtov na vyžiadanie), spracováva širší výber hodnôt a dokonca má schopnosť rozkladať GEP (`GetElementPtr`) inštrukcie. Všetky tieto vlastnosti jasne naznačujú, že *basicaa* nie je niečo, čo sa dá ľahko nahradiť. Slovo „basic“ v *basicaa* neznamená „naivný“ alebo „elementárny“, znamená skôr „nevyhnutný“ či „nepostrádateľný“.

Ak *basicaa* tvrdí, že dva ukazovatele nie sú aliasmi, *cfl-aa* prechody už nemusia vykonávať výpočtovo náročné analýzy, aby prišli s rovnakým rozhodnutím o vzťahu dvoch ukazovateľov ako *basicaa*. Do budúcna by bolo zaujímavé vypracovať podrobnejšiu analýzu predností a slabín *basicaa*, ktorá by poskytla lepšie poznatky o tom, aké aspekty *cfl-aa* by sa mali vylepšiť, a zároveň aby tieto vylepšenia znovu neimplementovali to, čo už vykonáva *cfl-aa*.

5.6.6 Reakcia cfl-aa na zmenu tela funkcie

V tom prípade sa jedná skôr o obmedzenie *cfl-aa* prechodov ako o obmedzenie celej alias analýzy v LLVM. Pri zapnutí *cfl-anders-aa* alebo *cfl-steens-aa* použitím prepínača *-use-cfl-a* pri optimalizačnej úrovni väčšej ako nula dochádza k tomu, že správca prechodov vloží *cfl-anders-aa* alebo *cfl-steens-aa* na začiatok postupnosti behu prechodov (*pipeline*). Raz spustí *cfl-aa* prechod a už ho nikdy znova nespustí.

Vyhodnotenia funkcií vygenerované *cfl-aa* sa rýchlo stávajú zastaranými po behu niekoľkých transformačných prechodov. Tieto transformačné prechody môžu potenciálne pridávať alebo odstrániť hodnoty z tiel funkcií. Pre zachovanie logickej správnosti musí alias analýza konzervatívne vrátiť odpoveď *nie je alias* ak niektoré z hodnôt z dopytu na preskúmanie alias analýzou nie sú nájdené v sumári funkcie. Vo výsledku len transformačné prechody naplánované v na začiatku postupnosti behu optimalizačných prechodov profitujú z výsledkov *cfl-aa*. Neskôr naplánované prechody vôbec nepocítia prítomnosť *cfl-aa*.

Riešením tohto obmedzenia by mohlo byť riešenie vo forme periodického spustenia *cfl-aa* správcom prechodov po určitom počte behov transformačných prechodov. Prípadne povoliť transformačným prechodom explicitne zneplatniť sumáre funkcií, ktoré boli vygenerované prechodom *cfl-aa*. Takto zneplatnené sumáre funkcií by boli nahradené novými sumármi, ktoré by alias analýza znovu automaticky vygenerovala.

Kapitola 6

Citlivosť na položky štruktúr

V podkapitole 5.6 je uvedených viacero problémov v súčasnej implementácii alias analýzy v LLVM. Po zvážení týchto problémov som sa rozhodol, že implementačnou časťou tejto práce bude rozšírenie súčasnej implementácie Andersenovho algoritmu v LLVM o citlivosť na položky štruktúr. Pridanie citlivosti na položky štruktúr môže viesť k nezanedbateľnému zlepšeniu presnosti alias analýzy v LLVM. Implementované riešenie je založené metóde pracujúcej s posunmi (*offsetmi*) položiek v štruktúrach. Ako je uvedené v podkapitole 6.1, ktorá analyzuje možné metódy pridania citlivosti na položky do základného modelu, táto metóda neobsahuje problémy vznikajúce pri použití metódy založenej na názvoch položiek v štruktúrach.

6.1 Rozšírenie základného modelu

V tejto podkapitole sú uvedené možné metódy pridania citlivosti na položky štruktúr do základného modelu spolu s ich výhodami a nevýhodami.

6.1.1 Metóda založená na posunoch v štruktúrach

Prvá metóda je založená na posunoch položiek v štruktúrach [19]. Hlavnou myšlienkou tejto metódy je, že ak sú premenné v obmedzeniach (contraint variables) identifikované celými číslami, je možné sa na takúto premennú odkazovať pomocou posunu od druhej premennej. Základný model je rozšírený o nasledujúce tvary: $p \supseteq (\star(q+k)$ / $(\star(p+k) \supseteq q$ / $(\star(p+k) \supseteq q$, kde k je ľubovoľná konštanta a $\star(p+k)$ znamená: pridanie $Sol(p)$ (množina riešení, solutions set pre p) do dočasnej množiny, pridanie k ku každému prvku a dereferencovanie. Pri $k = 0$ sú tieto tvary ekvivalentné komplexným obmedzeniam v pôvodnom jazyku obmedzení.

Princíp tejto metódy na príklade:

```
typedef struct { int *f1; int *f2; } aggr_t;
aggr_t a, *b;
int *p, **q, c;
b = &a;
b->f2 = &c;
p = b->f2;
```

Indexy v bloku: $idx(a.f1) = 0, idx(a.f2) = 1, idx(b) = 2, dx(p) = 3, idx(q) = 4, idx(c) = 5$.

Points-to množiny:

- $b \supseteq a.f1$
- $\star(b+1) \supseteq t,\ t \supseteq c$
- $p \supseteq \star(b+1)$

Adresa a je modelovaná ako adresa jej prvej položky. Týmto spôsobom môžu byť položky a prístupné cez b bez problémov.

Pre analýzu jazyka C je potrebné reprezentovať aj konštrukcie typu $q = \&((*b).f2)$. Táto reprezentácia je problémom, keďže má dôjsť k načítaniu indexu *a.f2* do *Sol(q)*, ale neexistuje na to žiadny spôsob. Riešením je rozšíriť jazyk obmedzenie o povolenie reprezentácie $q \supseteq b+1$, kde sa načíta *Sol(b)* do dočasnej množiny, pridá sa 1 ku každému prvku. Dočasná množina sa zlúči do *Sol(q)*. Novo zavedené tvary je možné reprezentovať zmenou grafu obmedzení na graf, ktorý bude obsahovať ohodnotenia. Z $p \supseteq q+k$ sa stane $q \xrightarrow{k} p$.

6.1.2 Metóda založená na názvoch položiek v štruktúrach

Ďalšia metóda je založená na samotných názvoch položiek v štruktúrach [24]. Princíp tejto metódy je vysvetlený na nasledovnom kóde [19]:

```
typedef struct { int *f1; int *f2; } my_struct_t;
my_struct_t a, *b;
int *p, c;
a.f1 = &c;
b = &a; b
p = b->f2;
```

Points-to množiny:

- $a.f1 \supseteq c$
- $b \supseteq a$
- $p \supseteq (\star b) \parallel f2$

Operátor $\parallel$ je v tom prípade spojenie reťazcov, kde $a \parallel b \rightarrow a{\cdot}b$ a $(\star a) \parallel b \rightarrow c{\cdot}b$, ak $a \supseteq c$. Konštrukcia $p \supseteq (\star b) \parallel x$ vlastne nahrádza $p \supseteq \star a(b+k)$ z vyššie opísanej metódy založenej na posuvoch. Aj keď sa zdá, že rozdiel medzi týmito metódami je malý, tento prístup skrýva hneď niekoľko nevýhod.

```
typedef struct { int *f1; int *f2; } ty_1;
typedef struct { int *f3; int *f4; } ty_2;
ty_1 a;
ty_2 b;
void *c;
int d;
b.f3 = &d;
c = &b;
a = (struct ty_1) *c;
```

Tento kód je platným kódom jazyka C, no posledný príkaz spôsobí problém tejto metóde. Je to kvôli tomu, že typ a rozhoduje o tom, ktoré položky sú zahrnuté pri priradení.

Problémom je, že premenné *b.f1* a *b.f2* (vyplývajúce z *(⋆c) || f1 and (⋆c) || f2)* neexistujú, keďže *b* má iný, ale kompatibilný typ s *a*. Pre vyriešenie tohto problému boli vytvorené tri nové funkcie [24]: `normalise`, `lookup` a `resolve`. Ich účelom je vyriešiť problém rôznych názvov reprezentujúcich rovnaké miesto (ako napríklad *b.f1* a *b.f3*). Pri použití metódy založenej na posunoch tento problém nevzniká.

6.2 Implementácia citlivosti na položky štruktúr

Táto podkapitola popisuje postup implementácie citlivosti na položky štruktúr v prechode *cfl-anders-aa*. Ako je už uvedené v podkapitole 6.1.1, implementácia je založená na metóde pracujúcej s posuvmi položiek v rámci štruktúr. Kód rozšírenia je napísaný v jazyku C++11 a spĺňuje vývojárske pravidlá pre LLVM projekt[1]. Vývoj rozšírenia prebiehal s kódmi LLVM vo vezii 6.0[2]. Táto verzia LLVM bola oficiálne vydaná v marci 2018.

6.2.1 Zavedenie typu FieldOffset v AliasAnalysisSummary

Bol pridaný dátový typ `FieldOffset` založený na type `int64_t`. Tento dátový typ bude následne použitý na modelovanie hodnoty posunu položky v rámci štruktúry. Taktiež bola vytvorená konštanta `UnknownOffset`, ktorá má maximálnu hodnotu typu `int64_t` a reprezentuje neznámu (nekonštantnú) hodnotu posunu. Ďalej boli implementované pomocné funkcie na negáciu, sčítanie a odčítanie dvoch premenných typu `FieldOffset`, ktoré sú následne použité v rozšírení jadra algoritmu. Ak je jeden z argumentov týchto funkcií hodnota `UnknownOffset`, výsledok funkcie je tiež hodnota `UnknownOffset`.

6.2.2 Podpora posunov položiek v CFLGraph

Trieda `Edge`, ktorá reprezentuje hranu v `CFLGraph`, bola rozšírená o položku `Offset`, ktorá je dátového typu `FieldOffset`. Táto hodnota reprezentuje samotných posun položky v rámci štruktúry. Pri analýze inštrukcií s konštantnými výrazmi v LLVM IR kóde sa vo funkcii `visitConstantExpr` v `CFLGraph` skúma inštrukcia `GetElementPtr`, ktorej súčasťou je práve posun v štruktúre. Inštrukcia `GetElementPtr` je podrobnejšie popísaná v podkapitole 5.4. Vo funkcii `visitGEP` sa táto inštrukcia podrobne rozkladá a analyzuje. Pomocou funkcie `accumulateConstantOffset` sa z argumentu typu `GEPOperator` získava konštantná hodnota posunu položky v štruktúre. Následne sa pomocou funkcie `addAssignEdge` pridá hrana priradenia medzi *ptrval* a *result*, ktorá je ohodnotená hodnotou tohto posunu. Ak je posun nekonštantná hodnota, ako posun sa nastaví konštanta `UnknownOffset`. U uzlov, ktoré nesúvisia so štruktúrami, má hrana medzi týmito uzlami hodnotu 0. Pre zjednodušenie práce s funkciami `addEdge` a `addAssignEdge` je argument pre posuv predvolene nastavený na hodnotu 0.

6.2.3 Úprava jadra algoritmu

Tieto úpravy sa týkali priamo Andersenovho algoritmu v LLVM, ktorý je implementovaný v súbore *CFLAndersAliasAnalysis.cpp*. Pomocné triedy `ValueSummary` a `WorkListItem` sú rozšírené o položku `Offset` typu `FieldOffset`, ktorá reprezentuje posun. Informácie o posunoch sú propagované do `ReachabilitySet` (množiny dosiahnuteľnosti, neterminál

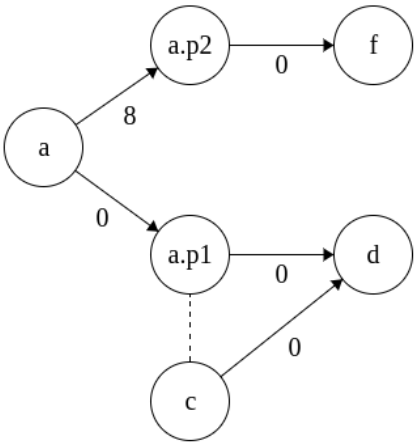
[1] https://llvm.org/docs/CodingStandards.html
[2] https://github.com/llvm-mirror/llvm/tree/release_60

V v pôvodnej gramatike 2.2.2). Metóda `insert` pre vkladanie hrán do `ReachabilitySet` bola rozšírená o parameter, ktorý umožní predávanie informácie o posune. V miestach, kde sa táto funkcia volá, je rozšírené volanie funkcie o argument s posuvom položky v rámci štruktúry. Posuny sú taktiež propagované do vektora `ExtRelations` obsahujúceho záznamy o externých vzťahoch. Propagácia pamäťových aliasov (`MemAliasPropagatate`) je spúšťaná len pri nulovom posune alebo pri posune s hodnotou `UnknownOffset`. Kľúčovou zmenou, je že funkcia `propagate` implementuje novú znalosť, že ak existuje hrana priradenia z X + *posuv* do Y, to znamená že miesto *(Y - posuv)* je dosiahnuteľné z X v stave `FlowToWriteOnly` a miesto *(X + posuv)* je dosiahnuteľné z Y v stave `FlowFromReadOnly`.

Princíp rozšírenia o citlivosti na položky štruktúr je nasledovný:

```
typedef struct { int *p1; int *p2; } example_t;
int main(void) {
    example_t a;
    int *c;
    int d,e,f;

    a.p1 = &d;
    a.p2 = &f;
    c = a.f1;
}
```



Obrázok 6.1: Grafová štruktúra PEG pre vyššie uvedený kód v jazyku C

Alias analýza s citlivosťou na položky štruktúr vie odhaliť, že ukazovateľ c môže ukazovať len na miesto v pamäti, kde je uložená hodnota premennej d.

	Bez citlivosti na položky štruktúr	S citlivosťou na položky štruktúr
points-to množina	{c → {d}}	{c → {d, f}}

Tabuľka 6.1: Výsledky alias analýzy

Testy implementácie a prekladača

Nová zostava prekladača s implementovanou citlivosťou na položky štruktúr bola podrobená testom, či zmeny nespôsobili problémy pri samotnom prekladaní programov. Implementácia bola overená na rôznych testoch: testy od Codasip (*Codasip compiler tests*) a následne aj na sade testov, ktorou sa testuje samotný prekladač jeho vývojármi (*LLVM/Clang test suite*). Testy nepreukázali žiadne regresívne chovanie po aplikácii úprav. Ďalším overením stability implementácie bolo samotné testovanie výkonnosti preložených programov v SPEC2006, ktoré prebehlo bez problémov.

6.3 Podpora citlivosti na položky štruktúr v praxi

Príklad v jazyku C pre ukážku, ako citlivosť na položky štruktúr alebo polí ovplyvní výsledky prechodu *cfl-anders-aa*:

```
int a[3];
int *a4 = a[0];
int *a8 = a[1];
int *b = a;
int *b4 = b[1];
```

LLVM IR pre tento kód je:

```
%a = alloca [3 x i32], align 4
%a4 = getelementptr inbounds [3 x i32], [3 x i32]* %a, i32 0, i32 1
%a8 = getelementptr inbounds [3 x i32], [3 x i32]* %a, i32 0, i32 2
%b = bitcast [3 x i32]* %a to i32*
%b4 = getelementptr inbounds i32, i32* %b, i32 1
```

Po spustení prechodu *cfl-anders-aa* (aktuálna implementácia v LLVM) sú výsledky nasledovné:

```
MayAlias: [3 x i32]* %a, i32* %a4
MayAlias: [3 x i32]* %a, i32* %a8
MayAlias: i32* %a4, i32* %a8
MayAlias: [3 x i32]* %a, i32* %b
MayAlias: i32* %a4, i32* %b
MayAlias: i32* %a8, i32* %b
MayAlias: [3 x i32]* %a, i32* %b4
MayAlias: i32* %a4, i32* %b4
MayAlias: i32* %a8, i32* %b4
MayAlias: i32* %b, i32* %b4
```

Alias analýza odpovedala na všetky dopyty, či dva ukazovatele sú aliasmi konzervatívne, výstupom *môže byť alias*.

Po pridaní citlivosti na položky štruktúr do *cfl-anders-aa* sú výsledky prechodu:

```
MayAlias: [3 x i32]* %a, i32* %a4
MayAlias: [3 x i32]* %a, i32* %a8
NoAlias: i32* %a4, i32* %a8
MayAlias: [3 x i32]* %a, i32* %b
NoAlias: i32* %a4, i32* %b
NoAlias: i32* %a8, i32* %b
MayAlias: [3 x i32]* %a, i32* %b4
MayAlias: i32* %a4, i32* %b4
NoAlias: i32* %a8, i32* %b4
NoAlias: i32* %b, i32* %b4
```

Alias analýza vďaka podpore citlivosti na položky vedela rozhodnúť, že napríklad ukazovatele *a4* a *a8* nie sú aliasmi. Je to z dôvodu, že každá položka bola modelovaná jednotlivo, narozdiel od doterajšej implementácie, kde sú všetky položky modelované ako jeden ukazovateľ (jeden uzol v grafe).

Kapitola 7

Vplyv rozšírenia na rýchlosť prekladu a výkonnosť programov

Dokončená implementácia rozšírenia prechodu *cfl-anders-aa* o citlivosť na položky štruktúr bola podrobená rôznym analýzam a meraniam. Boli zisťované vplyvy tohto rozšírenia na rýchlosť prekladu programov a na výkonnosť preložených programov.

7.1 Analýza rýchlosti prekladu

Na sade testovacích programov, na ktorých som v podkapitole 5.2 už vykonával porovnanie algoritmov alias analýzy, som vykonal ďalšie merania. Cieľom bolo zistiť, ako pridaná citlivosť na položky ovplyvnila dobu prekladu týchto programov. Optimalizačná úroveň bola nastavená na O3. Meranie bolo vykonávané na architektúre x86-64 s prekladačom Clang vo verzii 6.0. Merania doby prekladu boli vykonávané pomocou nástroja *time*. Uvedené hodnoty v milisekundách sú priemery z desiatich nameraných hodnôt dôb trvania prekladu.

Program	Bez citlivosti na položky (v ms)	S citlivosťou na položky (v ms)	nárast (v %)
bitcnt	55	56	1,02
coremark	516	519	1,01
conv2d	106	107	1,01
crc	61	61	0
dhrystone	123	124	1,01
fft	142	145	1,02
fir	97	97	0
md5	256	259	1,01
rc4	86	88	1,02
sha3	191	193	1,01
SIMLIB	6 964	7 096	1,01
dmckeee	6 751	6 829	1,01

Tabuľka 7.1: Porovnanie doby trvania prekladu

Ako je možné vidieť v tabuľke 7.1, meraním som potvrdil očakávané zvýšenie časovej náročnosti. Pridaná citlivosť na položky štruktúr podľa nameraných výsledkov zvýšila dobu potrebnú na preklad programov približne o 1 percento.

7.2 Analýza výkonnosti programov

Táto podkapitola popisuje vykonané merania výkonnosti preložených programov a analýzu nameraných výsledkov, kde sa porovnával pôvodný prechod *cfl-anders-aa* s verziou tohto prechodu rozšírenou o citlivosť na položky štruktúr.

7.2.1 Merania na x86-64 architektúre

Merania výkonnosti programov x86-64 architektúre bolo vykonané pomocou SPEC CPU 2006[1], konkrétne s SPECint 2006[2]. Optimalizačná úroveň bola nastavená na O3. Počet spustení testovacej sady programov bol ponechaný na predvolenej hodnote 3.

SPEC

SPEC (*Standard Performance Evaluation Corporation*) je nezisková organizácia vytvorená na zriadenie, udržiavanie a schvaľovanie štandardizovaných referenčných kritérií a nástrojov na vyhodnocovanie výkonnosti a energetickej efektívnosti najnovšej generácie počítačových systémov.

SPEC CPU 2006

SPEC CPU 2006 obsahuje sadu programov (*benchmarks*), na ktorých sa vykonávajú testy výkonnosti procesorov. Skladá sa z SPECint a SPECfp častí, ktoré testujú výkonnosť procesora pri celočíselných výpočtoch, resp. výpočtoch s reálnymi číslami. SPECint obsahuje programy v C, C++, SPECfp v C, C++ a Fortrane. Pre jazyk Fortran existuje prekladač Flang[3], ktorý je založený na LLVM, no zatiaľ je experimentálny a v aktívnom vývoji. Z tohoto dôvodu som sa rozhodol, že merania budú zamerané len na C/C++ programy v SPECint. Po skončení testu je výsledný čas porovnaný s referenčným časom, následne sa získa pomer referenčného času ku nameranému. Tento pomer sa stáva výsledkom pre daný test.

Program	Jazyk	Oblasť
400.perlbench	C	programovací jazyk
401.bzip2	C	kompresia
403.gcc	C	prekladač C
429.mcf	C	kombinované optimalizácie
445.gobmk	C	umelá inteligencia: hra Go
456.hmmer	C	hľadanie sekvencie génov
458.sjeng	C	umelá inteligencia: šachy
462.libquantum	C	fyzikálne a kvantové výpočty
464.h264ref	C	kompresia videa
471.omnetpp	C++	diskrétna simulácia
473.astar	C++	algoritmy hľadania ciest
483.xalancbmk	C++	spracovanie XML

Tabuľka 7.2: Programy v SPECint 2006

[1] https://www.spec.org/cpu2006/
[2] https://www.spec.org/cpu2006/CINT2006/
[3] https://github.com/flang-compiler/flang

Výsledky meraní

Hlavným výsledkom SPECint je SPECint_base2006, čo je hodnota definujúca priemerný pomer zo všetkých testov. Ak sa program vykonal rýchlejšie, v porovnaní voči referenčnému času to má dôsledok v podobe zvýšenia pomeru týchto dvoch časov.

Program	Sekundy	Pomer	Sekundy	Pomer	Sekundy	Pomer
400.perlbench	276	35,4	263	37,2	297	32,9
401.bzip2	290	24,7	364	36,5	403	24,0
403.gcc	250	32,2	229	35,1	246	32,7
429.mcf	350	26,0	270	33,6	295	30,9
445.gobmk	378	27,8	377	27,8	394	26,6
456.hmmmer	337	27,7	337	27,7	350	26,7
459.sjeng	398	30,4	395	30 6	412	29,4
462.libquantum	282	72,7	286	72,5	309	67,0
464.h264ref	404	54,8	434	51,0	437	50,6
471.omnetpp	330	18,9	376	16,6	371	16,8
473.astar	314	22,3	364	19,3	353	19,9
483.xalancbmk	177	39,1	190	36,3	216	31,9

Tabuľka 7.3: Výsledky jednotlivých programov — *cfl-anders-aa* bez citlivosti na položky

Program	Sekundy	Pomer	Sekundy	Pomer	Sekundy	Pomer
400.perlbench	281	34,8	259	37,8	259	37,8
401.bzip2	363	26,6	363	26,5	364	26,5
403.gcc	229	35,2	224	36,0	225	35,8
429.mcf	282	32,3	262	34,8	263	34,7
445.gobmk	377	27,7	377	27,7	378	27,8
456.hmmmer	336	27,8	335	27,8	335	27,8
459.sjeng	390	31,0	388	31,2	387	31,2
462.libquantum	280	74,1	276	75,1	278	74,6
464.h264ref	402	55,1	400	55,3	401	55,2
471.omnetpp	310	20,2	330	18,9	310	20,2
473.astar	307	22,9	309	22,7	312	22,5
483.xalancbmk	174	39,7	176	39,3	176	39,3

Tabuľka 7.4: Výsledky jednotlivých programov — *cfl-anders-aa* s citlivosťou na položky

	Bez citlivosti na položky	S citlivosťou na položky
SPECint_base2006	31,4	33,8

Tabuľka 7.5: Výsledky SPECint_base2006

Výsledky v tabuľke 7.5 poukazujú na to, že verzia prechodu *cfl-anders-aa* s citlivosťou na položky dosiahla vyššiu hodnotu SPECint_base2006. Tento výsledok značí, že testované programy prebehli rýchlejšie (presné výsledné hodnoty su uvedené v tabuľkách 7.3 a 7.4). Meranie preukázalo, že došlo k zvýšeniu výkonnosti programov a to práve kvôli rozšíreniu *cfl-anders-aa* o citlivosť na položky štruktúr.

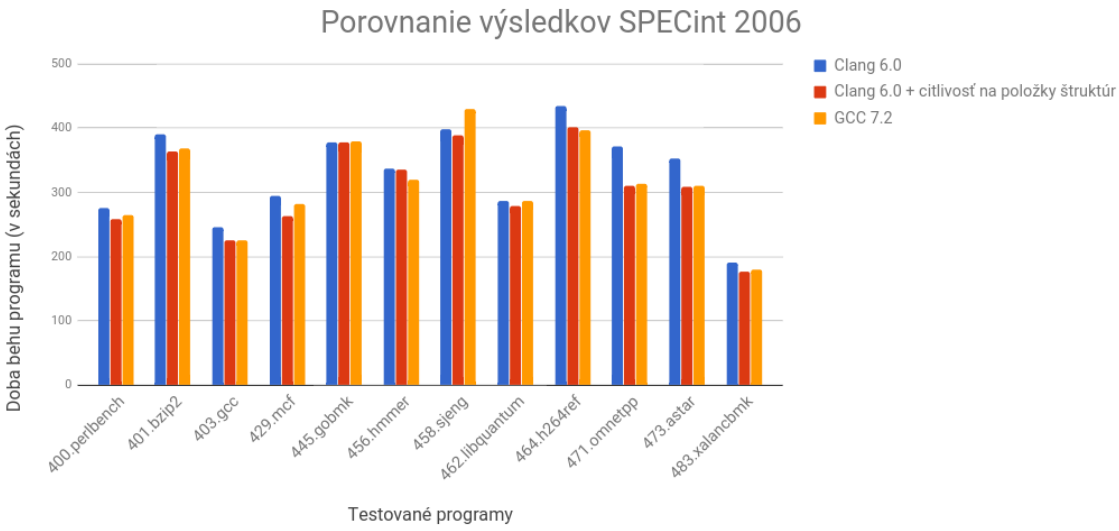
Porovnanie s GCC

Alias analýza v prekladači GCC je citlivá na položky štruktúr[4]. Pre porovnanie optimalizácií prekladača Clang voči prekladaču GCC boli zmerané výsledky SPECint 2006 aj pre GCC 7.2. Výsledná hodnota SPECint_base2006 pre GCC 7.2 je 33,2.

Program	Sekundy	Pomer	Sekundy	Pomer	Sekundy	Pomer
400.perlbench	279	35,0	365	36,9	363	37,1
401.bzip2	377	25,6	368	26,2	368	26,2
403.gcc	226	35,6	224	35,9	225	35,8
429.mcf	288	31,7	282	32,3	279	32,7
445.gobmk	380	27,6	380	27,6	380	27,6
456.hmmmer	319	29,3	319	27,3	318	29,3
459.sjeng	432	28,0	430	28,1	429	28,2
462.libquantum	292	71,0	286	72,4	284	72,9
464.h264ref	397	55,7	398	55,6	396	55,9
471.omnetpp	318	19,6	313	20,0	307	20,4
473.astar	311	22,6	310	22,6	310	22,6
483.xalancbmk	184	37,4	177	38,9	179	38,5

Tabuľka 7.6: Výsledky jednotlivých programov — GCC 7.2

[4] https://www.airs.com/dnovillo/200711-GCC-Internals/200711-GCC-Internals-5-alias.pdf



Obrázok 7.1: Porovnanie nameraných výsledkov SPECint 2006

Graf na obrázku 7.1 zobrazuje najrýchlejšie časy behu programov zo všetkých testovaných prípadov. Z grafu je možné vidieť znateľný časový rozdiel — nárast výkonnosti programov, ktorý priniesla práve implementovaná citlivosť na položky štruktúr. Clang s alias analýzou citlivou na položky štruktúr vo väčšine prípadov dominoval aj nad GCC. Dosiahol SPECint_base2006 hodnotu 33,8, zatiaľ čo GCC 7.2 dosiahol výsledku 33,2.

7.2.2 Merania na ďalších architektúrach

Merania výkonnosti programov boli taktiež vykonané na ďalších architektúrach — *µRISC*, *µVliw*, *Helium*, *Berkelium*. Vplyv rozšírenia *cfl-anders-aa* o citlivosť na položky štruktúr bol analyzovaný na preložených programoch bežiacich na týchto architektúrach procesorov. Meranie bolo vykonané pomocou vývojárskeho prostredia od spoločnosti Codasip. Meranie spočívalo v zistení počtu vykonaných hodinových cyklov.

Codasip Studio

Codasip Studio sa používa na:

- prototypovanie procesorov pre špecifické aplikačné domény
- rýchly prieskum dizajnov
- vývoj vlastných rozšírení pomocou jazyka CodAL slúžiaceho na popis architektúr

Codasip Studio generuje špecifické SDK pre danú architektúru procesorov spolu s:

- skriptami na testy a syntézu
- prekladačom a simulátormi
- pokročilými nástrojmi na profilovanie a ladenie

Za účelom merania som pomocou tohto prostredia vygeneroval SDK pre vyššie spomenuté architektúry, prekladač a simulátor na inštrukčnej úrovni. Následne som pomocou tohto prekladača vygeneroval LLVM IR kód, ktorý bol následne optimalizovaný optimalizátorom, ktorý obsahoval moje rozšírenie *cfl-anders-aa* o citlivosť na položky štruktúr. Optimalizovaný LLVM IR kód bol následne preložený do binárnej formy vygenerovaným Codasip prekladačom.

Počet vykonaných hodinových cyklov bol získaný pomocou simulátora na inštrukčnej úrovni a jeho prepínača *–dump-clock-cycles* nasledovným príkazom:

```
ia-isimulator -r program_binary --dump-clock-cycles result.txt
```

Testované architektúry procesorov

Prvým testovaným procesorom bol procesor *Codasip µRISC*. Tento procesor je navrhnutý najmä pre výukové účely, ale obsahuje kompletnú minimálnu inštrukčnú sadu potrebnú pre automatické zostavenie prekladača jazyka C. Ďalším testovaným procesorom je *Codasip µVliw*, ktorý umožňuje inštrukčný paralelizmus (vykonávanie niekoľko nezávislých inštrukcií súbežne) bez toho, aby procesor musel vyhodnocovať časovú nadväznosť jednotlivých operácií a možné kolízie. Paralelizmus je dosiahnutý zlúčením vzájomne nezávislých operácií v jeden celok. Tretím testovaným procesorom je *Codix Helium*, ktorý je ideálny pre využitia, kde je požadovaná čo najväčšia výkonnosť pri zachovaní nízkej spotreby. Vďaka rozsiahlej inštrukčnej sade a rôzne dlhým inštrukciám ponúka *Codix Helium* nielen o dosť lepší výkon ako ostatné procesory v tejto triede, ale aj kompaktnú veľkosť kódu. Posledným testovaným procesorom bol *Codix Berkelium*, ktorý ponúka kompatibilitu s inštrukčnou sadou RISC-V s možnosťami ostatných architektúr.

Výsledky meraní

Nameranými hodnotami boli počty vykonaných hodinových cyklov u jednotlivých programov. Nižší počet hodinových cyklov, ktorý reprezentuje vyššiu výkonnosť programu, je v tabuľke zvýraznený tučným písmom.

Program	Bez citlivosti na položky (cykly)	S citlivosťou na položky (cykly)
conv2d	30162	**27359**
coremark	5350983	**5248486**
dhrystone	**8240492**	8250492
fft	**236039**	236879
fir	100365	**79490**
md5	**1051573**	1055224

Tabuľka 7.7: Výsledky merania pre architektúru *Codasip µRISC*

Program	Bez citlivosti na položky (cykly)	S citlivosťou na položky (cykly)
conv2d	18976	**18173**
coremark	3960966	**3826227**
dhrystone	**5870335**	5880335
fft	131739	**129939**
fir	54518	**48823**
md5	**629450**	631165

Tabuľka 7.8: Výsledky merania pre architektúru *Codasip µVliw*

Program	Bez citlivosti na položky (cykly)	S citlivosťou na položky (cykly)
conv2d	27714	**25916**
coremark	4200910	**4061390**
dhrystone	**6590484**	6600484
fft	**213355**	217265
fir	98640	**77745**
md5	**978048**	981845

Tabuľka 7.9: Výsledky merania pre architektúru *Codix Helium*

Program	Bez citlivosti na položky (cykly)	S citlivosťou na položky (cykly)
conv2d	27663	**27359**
coremark	4326735	**4255755**
dhrystone	**6000475**	6010475
fft	**222355**	226270
fir	99375	**78485**
md5	**980604**	983896

Tabuľka 7.10: Výsledky merania pre architektúru *Codix Berkelium*

Z výsledkov je možné odvodiť, že k viditeľnému zlepšeniu výkonnosti došlo v programoch *conv2d*, *coremark* a *fir*. Programy *conv2d* a *fir* pracujú s rozsiahlym poľom konštantných čísel a maticami, program *coremark* implementuje jednosmerne viazaný zoznam a funkcie pracujúce s ním. Tento charakter programov poukazuje na fakt, že citlivosť na položky štruktúr sa naplno prejavila v podobe lepších výsledkov alias analýzy, z ktorých profitovali ostatné optimalizačné prechody.

Mierne zvýšenie počtu hodinových taktov u programov *dhrystone* a *md5* mohlo nastať aj napriek lepším výsledkom alias analýzy, keďže ostatné prechody tieto výsledky preberajú a s nimi pracujú. Taktiež záleží aj na kvalite použitých heuristík v týchto prechodoch.

Pre podrobnejšiu analýzu práce optimalizátora som získal LLVM IR po každom optimalizačnom prechode pomocou prepínača *-print-after-all*. Získané dáta som porovnal pre zistenie rozdielov v LLVM IR. Zvýšil sa počet vygenerovaných inštrukcií v prechode *Simplify the CFG*, ktoré ale boli následne eliminované v prechode *Combine redundant instructions*. Prechody súvisiace s optimalizáciou cyklov *Rotate Loops* a *Induction Variable Simplification* taktiež vygenerovali väčší počet inštrukcií.

Ďalej som porovnal výsledky pre preložené programy pomocou vygenerovaného Codasip prekladača v kombinácii s optimalizátorom z LLVM 6.0, ktorý obsahoval prechod *cfl-anders-aa* bez citlivosti na položky štruktúr. Získané výsledky boli na úrovni výsledkov optimalizátora s prechodom *cfl-anders-aa* rozšíreným o citlivosť na položky štruktúr. Keďže sú prekladače od spoločnosti Codasip v súčasnosti založené na LLVM 5.0, rozdiel vo výsledkoch a mierne zhoršenie v niektorých prípadoch súvisí so zmenami medzi verziami 5.0 a 6.0 a nie s implementovaným rozšírením. Tieto zmeny medzi verziami by bolo vhodné do budúcna analyzovať a zistiť, čo spôsobilo horšie výsledky v testovaných prípadoch.

7.3 Zhodnotenie a ďalšie možnosti vylepšenia

Prechod *cfl-anders-aa* v LLVM bol rozšírený o citlivosť na položky štruktúr. Bolo vykonaných niekoľko meraní pre analýzu vplyvu tohto rozšírenia na rýchlosť prekladu testovaných programov a ich výkonnosť. Získané výsledky preukázali značný nárast výkonnosti programov za cenu zanedbateľného zvýšenia času prekladu (okolo jedného percenta). Rozšírenie prechodu *cfl-anders-aa* o citlivosť na položky štruktúr vylepšilo schopnosti alias analýzy v LLVM a zvýšilo presnosť jej výsledkov. Toto rozšírenie preto považujem za vhodne zvolenú možnosť vylepšenia alias analýzy v LLVM, ktorá priniesla viditeľný nárast výkonnosti preložených programov.

V podkapitole 5.6 som uviedol hneď niekoľko problémov alias analýzy v LLVM, ktoré stoja za preskúmanie a zhodnotenie, či vynaložené úsilie do implementácie bude adekvátne získaným výsledkom. Za menej náročné vylepšenie sa javí spresnenie alias analýzy pre analýzu rekurzívnych funkcií. Za najviac prospešné vylepšenie alias analýzy v LLVM považujem vyladenie samotnej implementácie *cfl-anders-aa* a s ním spojené zrýchlenie výpočtov pomocou optimalizácií pre tento algoritmus. Za ďalšie vhodné vylepšenie sa dá považovať zavedenie reakcie *cfl-aa* na zmenu tela funkcie spôsobenú optimalizačnými a transformačnými prechodmi.

Kapitola 8

Záver

Cieľom tejto práce bolo preskúmať stav alias analýzy v LLVM a zvýšiť presnosť jej výsledkov. Po preštudovaní problematiky alias analýzy bolo vykonané porovnanie aktuálne implementovaných algoritmov v LLVM. Prebehla analýza problémov a obmedzení súčasnej implementácie alias analýzy v LLVM. Na základe týchto zistení som sa rozhodol implementovať rozšírenie Andersenovho algoritmu v LLVM o citlivosť na položky štruktúr.

Citlivosť na položky štruktúr znamená, že každá položka v štruktúre je modelovaná osobitne, ako samostatný objekt, z pohľadu alias analýzy. Implementované rozšírenie je založené na metóde, ktorá rozširuje základný model alias analýzy o ohodnotené hrany, kde je použitá informácia o posune položky v rámci štruktúry.

Implementácia bola následne podrobená rozsiahlej sade testov určených na overenie funkčnosti prekladača. Dôležitou časťou práce bola analýza vplyvu rozšírenia na rýchlosť prekladu a výkonnosť preložených programov. Podrobné merania boli vykonané na x86-64 architektúre s Clang/LLVM 6.0. Merania odhalili zvýšenie času potrebného na preklad programu o jedno percento. Výkonnosť programov zaznamenala významný nárast. Rozšírenie o citlivosť na položky štruktúr prinieslo presnejšie výsledky alias analýzy, ktoré následne umožnili optimalizačným prechodom vykonávať lepšie a výhodnejšie transformácie a optimalizácie, ktoré sa prejavili v efektivite a rýchlosti testovaných programov. Vďaka tomuto rozšíreniu výsledné programy preložené prekladačom Clang s implementovaným rozšírením výrazne prekonali pôvodnú verziu prekladača bez citlivosti na položky štruktúr. Prekladač Clang s týmto rozšírením bol porovnaný aj voči prekladaču GCC. Výkonnosť preložených programov dosiahla nielen úroveň výkonnosti programov, ktoré boli preložené prekladačom GCC, ale ich aj mierne prekonala.

Výsledky boli získané meraním aj na rôznych architektúrach procesorov od spoločnosti Codasip za pomoci prostredia Codasip Studio. U týchto meraní bol taktiež v niektorých prípadoch potvrdený značný nárast výkonnosti testovaných programov.

V práci boli spomenuté aj ďalšie obmedzenia alias analýzy, ktoré by bolo vhodné preskúmať. Z pohľadu možného zvýšenia presnosti výsledkov alias analýzy sa ako veľmi zaujímavé možnosti javia rôzne optimalizácie pre samotné jadro Andersenovho algoritmu. Ďalšou možnosťou je implementácia reakcie alias analýzy na zmeny v telách funkcií po priebežných optimalizačných prechodoch a následné prepočítanie jej výsledkov za účelom ich spresnenia.

Literatúra

[1] Andersen, L.; Institut, K. U. D.: *Program Analysis and Specialization for the C Programming Language*. DIKU rapport, Datalogisk Institut, 1994.

[2] Chatterjee, K.; Choudhary, B.; Pavlogiannis, A.: Optimal Dyck Reachability for Data-dependence and Alias Analysis. *Proc. ACM Program. Lang.*, ročník 2, č. POPL, December 2017: s. 30:1–30:30, ISSN 2475-1421, doi:10.1145/3158118.
Dostupné z: http://doi.acm.org/10.1145/3158118

[3] Chen, J.: CFL Alias Analysis [online]. August 2016 [cit. 2017-11-29].
Dostupné z: https://github.com/grievejia/GSoC2016/blob/master/writeup.pdf

[4] Diwan, A.; McKinley, K. S.; Moss, J. E. B.: Type-based Alias Analysis. *SIGPLAN Not.*, ročník 33, č. 5, Máj 1998: s. 106–117, ISSN 0362-1340, doi:10.1145/277652.277670.
Dostupné z: http://doi.acm.org/10.1145/277652.277670

[5] Fähndrich, M.; Foster, J. S.; Su, Z.; aj.: Partial Online Cycle Elimination in Inclusion Constraint Graphs. *SIGPLAN Not.*, ročník 33, č. 5, Máj 1998: s. 85–96, ISSN 0362-1340, doi:10.1145/277652.277667.
Dostupné z: http://doi.acm.org/10.1145/277652.277667

[6] Ghiya, R.; Lavery, D.; Sehr, D.: On the Importance of Points-to Analysis and Other Memory Disambiguation Methods for C Programs. *SIGPLAN Not.*, ročník 36, č. 5, Máj 2001: s. 47–58, ISSN 0362-1340, doi:10.1145/378795.378806.
Dostupné z: http://doi.acm.org/10.1145/378795.378806

[7] Hardekopf, B.; Lin, C.: Exploiting Pointer and Location Equivalence to Optimize Pointer Analysis. In *Proceedings of the 14th International Conference on Static Analysis*, SAS'07, Berlin, Heidelberg: Springer-Verlag, 2007, ISBN 3-540-74060-0, 978-3-540-74060-5, s. 265–280.
Dostupné z: http://dl.acm.org/citation.cfm?id=2391451.2391470

[8] Hardekopf, B.; Lin, C.: Flow-sensitive Pointer Analysis for Millions of Lines of Code. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '11, Washington, DC, USA: IEEE Computer Society, 2011, ISBN 978-1-61284-356-8, s. 289–298.
Dostupné z: http://dl.acm.org/citation.cfm?id=2190025.2190075

[9] Hind, M.: Pointer Analysis: Haven'T We Solved This Problem Yet? In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '01, New York, NY, USA: ACM, 2001, ISBN

1-58113-413-4, s. 54–61, doi:10.1145/379605.379665.
Dostupné z: http://doi.acm.org/10.1145/379605.379665

[10] Hind, M.; Burke, M.; Carini, P.; aj.: Interprocedural Pointer Alias Analysis. *ACM Trans. Program. Lang. Syst.*, ročník 21, č. 4, Júl 1999: s. 848–894, ISSN 0164-0925, doi:10.1145/325478.325519.
Dostupné z: http://doi.acm.org/10.1145/325478.325519

[11] Kastrinis, G.; Balatsouras, G.; Ferles, K.; aj.: An Efficient Data Structure for Must-alias Analysis. In *Proceedings of the 27th International Conference on Compiler Construction*, CC 2018, New York, NY, USA: ACM, 2018, ISBN 978-1-4503-5644-2, s. 48–58, doi:10.1145/3178372.3179519.
Dostupné z: http://doi.acm.org/10.1145/3178372.3179519

[12] Lattner, C.; Adve, V.: LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, Washington, DC, USA: IEEE Computer Society, 2004, ISBN 0-7695-2102-9, s. 75–86.
Dostupné z: http://dl.acm.org/citation.cfm?id=977395.977673

[13] Lattner, C.; Lenharth, A.; Adve, V.: Making Context-sensitive Points-to Analysis with Heap Cloning Practical for the Real World. *SIGPLAN Not.*, ročník 42, č. 6, Jún 2007: s. 278–289, ISSN 0362-1340, doi:10.1145/1273442.1250766.
Dostupné z: http://doi.acm.org/10.1145/1273442.1250766

[14] Lin, S.-H.: *Alias Analysis in LLVM [online]*. Diplomová práca, Lehigh University, 5 2015 [cit. 2017-12-18].
Dostupné z: http://www.cse.psu.edu/~gxt29/paper/ShengHsiuLin_thesis.pdf

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

[16] McConnell, C.; Johnson, R. E.: Using Static Single Assignment Form in a Code Optimizer. *ACM Lett. Program. Lang. Syst.*, ročník 1, č. 2, Jún 1992: s. 152–160, ISSN 1057-4514, doi:10.1145/151333.151368.
Dostupné z: http://doi.acm.org/10.1145/151333.151368

[17] Paisante, V.; Maalej, M.; Barbosa, L.; aj.: Symbolic Range Analysis of Pointers. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, CGO '16, New York, NY, USA: ACM, 2016, ISBN 978-1-4503-3778-6, s. 171–181, doi:10.1145/2854038.2854050.
Dostupné z: http://doi.acm.org/10.1145/2854038.2854050

[18] Pandey, M.; Sarda, S.: *LLVM Cookbook*. Packt Publishing, 2015, ISBN 178528598X, 9781785285981.

[19] Pearce, D. J.; Kelly, P. H.; Hankin, C.: Efficient Field-sensitive Pointer Analysis of C. *ACM Trans. Program. Lang. Syst.*, ročník 30, č. 1, November 2007, ISSN 0164-0925, doi:10.1145/1290520.1290524.
Dostupné z: http://doi.acm.org/10.1145/1290520.1290524

[20] Qian, L.; Jianhua, Z.; Xuandong, L.: Optimize Context-sensitive Andersen-style Points-to Analysis by Method Summarization and Cycle-elimination. In *Proceedings of the 4th International Conference on Leveraging Applications of Formal Methods, Verification, and Validation - Volume Part I*, ISoLA'10, Berlin, Heidelberg: Springer-Verlag, 2010, ISBN 3-642-16557-5, 978-3-642-16557-3, s. 564–578.
Dostupné z: http://dl.acm.org/citation.cfm?id=1939281.1939334

[21] Sridharan, M.; Fink, S. J.: The Complexity of Andersen's Analysis in Practice. In *Proceedings of the 16th International Symposium on Static Analysis*, SAS '09, Berlin, Heidelberg: Springer-Verlag, 2009, ISBN 978-3-642-03236-3, s. 205–221, doi:10.1007/978-3-642-03237-0_15.
Dostupné z: http://dx.doi.org/10.1007/978-3-642-03237-0_15

[22] Steensgaard, B.: Points-to Analysis in Almost Linear Time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '96, New York, NY, USA: ACM, 1996, ISBN 0-89791-769-3, s. 32–41, doi:10.1145/237721.237727.
Dostupné z: http://doi.acm.org/10.1145/237721.237727

[23] Tarjan, R.: Depth-First Search and Linear Graph Algorithms. *SIAM Journal on Computing*, ročník 1, č. 2, 1972: s. 146–160, doi:10.1137/0201010.
Dostupné z: https://doi.org/10.1137/0201010

[24] Yong, S. H.; Horwitz, S.; Reps, T.: Pointer Analysis for Programs with Structures and Casting. *SIGPLAN Not.*, ročník 34, č. 5, Máj 1999: s. 91–103, ISSN 0362-1340, doi:10.1145/301631.301647.
Dostupné z: http://doi.acm.org/10.1145/301631.301647

[25] Zhang, Q.; Lyu, M. R.; Yuan, H.; aj.: Fast Algorithms for Dyck-CFL-reachability with Applications to Alias Analysis. *SIGPLAN Not.*, ročník 48, č. 6, Jún 2013: s. 435–446, ISSN 0362-1340, doi:10.1145/2499370.2462159.
Dostupné z: http://doi.acm.org/10.1145/2499370.2462159

[26] Zhang, Q.; Xiao, X.; Zhang, C.; aj.: Efficient Subcubic Alias Analysis for C. *SIGPLAN Not.*, ročník 49, č. 10, Október 2014: s. 829–845, ISSN 0362-1340, doi:10.1145/2714064.2660213.
Dostupné z: http://doi.acm.org/10.1145/2714064.2660213

[27] Zheng, X.; Rugina, R.: Demand-driven Alias Analysis for C. *SIGPLAN Not.*, ročník 43, č. 1, Január 2008: s. 197–208, ISSN 0362-1340, doi:10.1145/1328897.1328464.
Dostupné z: http://doi.acm.org/10.1145/1328897.1328464

Príloha A

Obsah CD

Na priloženom médiu v adresári *src* sú dostupné upravené súbory, v ktorých je implementované rozšírenie Andersenovho algoritmu o citlivosť na položky štruktúr a súbor *manual.txt* s návodom, ako použiť tieto súbory. V adresári *doc* sa nachádza PDF verzia technickej správy spolu s jej zdrojovými súbormi v jazyku LaTeX:

```
CD
├── src
│   ├── CFLGraph.h
│   ├── AliasAnalysisSummary.h
│   ├── CFLAndersAliasAnalysis.cpp
│   └── manual.txt
└── doc
    ├── BP.pdf
    └── zdrojové súbory technickej správy
```

Príloha B

Návod na použitie

1. Stiahnuť zdrojové súbory LLVM a Clang 6.0 podľa návodu na stránkach LLVM projektu[1]
2. Zdrojové súbory z priečinka *src* uloženého na priloženom CD skopírovať do adresára *lib/Analysis* a prepísať nimi pôvodné súbory
3. Zostaviť prekladač Clang podľa návodu na stránkach LLVM projektu[2]

[1] https://llvm.org/docs/GettingStarted.html
[2] https://clang.llvm.org/get_started.html