УНИВЕРЗИТЕТ У БЕОГРАДУ
МАТЕМАТИЧКИ ФАКУЛТЕТ

Филип Лазић

ОПТИМИЗАЦИЈА ЦЕЛОВИТОГ ПРОГРАМА НА КОМПАЈЛЕРСКОЈ ИНФРАСТРУКТУРИ LLVM

мастер рад

Београд, 2021.

Ментор:

др Иван Чукић, доцент
Универзитет у Београду, Математички факултет

Чланови комисије:

др Милена Вујошевић Јаничић, ванредни професор
Универзитет у Београду, Математички факултет

др Саша Малков, ванредни професор
Универзитет у Београду, Математички факултет

Датум одбране: 15. јануар 2016.

Садржај

1 Увод **1**

2 Компајлерска инфраструктура LLVM **3**

2.1 Међурепрезентација LLVM 4

2.2 Компајлер LLVM 5

2.3 Предности LLVM-а 8

3 Оптимизација целовитог програма **9**

3.1 Ортимизација током линковања 12

3.2 Оптимизација целовитог програма без подршке линкера 14

3.3 Уметање функција 15

3.4 Елиминација мртвог кôда 19

3.5 Девиртуализација 23

4 ThinLTO **29**

5 Алат за визуализацију промена **34**

6 Закључак **40**

Литература **41**

Глава 1

Увод

Компајлерске оптимизације трансформишу кôд тако да се програм брже извршава или користи мање меморије док при томе задржава семантичку еквивалентност. Обично, оптимизације се извршавају у контексту једног објектног фајла, али пошто у оквиру фајла компајлер нема информације о кôду који се налази у другим објектним фајловима, многе оптимизације је немогуће урадити зато што компајлер не може бити сигуран у семантичку еквивалентност. Главна тема овог рада биће управо решавање овог проблема, односно оптимизација целовитог програма у компајлерској инфраструктури LLVM, као и развој програма за визуализацију промена у међурепрезентацији приликом оптимизације целовитог програма.

У глави 2 овог рада је описана компајлерска инфраструктура LLVM, њене предности као и међурепрезентација LLVM, чијим се трансформацијама и имплементирају компајлерске оптимизације.

У глави 3 је описана оптимизација целовитог програма, која је и главни фокус овог рада. Поред самог описа имплементације оптимизације целовитог програма у компајлерској инфраструктури LLVM, у раду су приказане најважније оптимизације као што су елиминација мртвог кôда, уметање и девиртуализација. Уз сваку оптимизацију су приказани примери који показују разлику унутар међурепрезентације LLVM када је активна оптимизација целовитог програма и када није.

У глави 4 је приказан нови приступ оптимизацији целовитог програма ThinLTO, који умањује утицај оптимизације целовитог програма на време превођења програма, као и на меморијско заузеће без битних губитака у квалитету оптимизација.

У глави 5 је приказан алат развијен као део овог рада, који визуализује разлике између програма који је преведен са и без оптимизације целовитог програма.

Глава 2

Компајлерска инфраструктура LLVM

LLVM [1] сачињава колекција алата (компајлера, асемблера, дибагера, линкера) који су дизајнирани да буду компатибилни са постојећим алатима пре свега на Unix системима. Ови алати се могу користити за развој предњег дела компајлера (eng. front-end) за било који програмски језик, као и за развој задњег дела компајлера (eng.back-end) за сваку компјутерску архитектуру. LLVM је започет као истраживачки пројекат на Универзитету Илиноис са циљем да пружи статичку и динамичку компилацију програмских језика. Данас, LLVM садржи велики број подпројеката који се користе у великом обиму што у продукцијске што у истраживачке сврхе.
Неки од најбитнијих потпројеката су:

1. Језгро LLVM-а које садржи све потребне алате и библиотеке за конверзију међурепрезентације у објектне фајлове
2. Clang – предњи део за програмске језике C, C++ и Objective C
3. libc++ – имплементација стандардне библиотеке C++
4. LLDB – дебагер
5. LLD – линкер

2.1 Међурепрезентација LLVM

Међурепрезентација LLVM (LLVM IR [2]) заснована је на статичкој јединственој форми доделе (SSA [3]). Ова форма захтева да се свакој променљивој вредност додели тачно једном, као и да свака променљива буде дефинисана пре употребе. LLVM међурепрезентација је дизајнирана тако да подржи интерпроцедуралне оптимизације, анализу целог програма, агресивно реструктуирање програма итд. Веома битан аспект LLVM међурепрезентације је то што је она дефинисана као језик са јасно дефинисаном семантиком. Ова међурепрезентација се може користити у три различите форме:

1. текстуални асемблерски формат (.ll)
2. биткôд формат (.bc) [1]
3. унутар-меморијски формат

Ове три форме омогућавају лакши развој, уз могућност визуелне анализе и дебаговања трансформација. Сва три формата су еквивалентна и лако се могу трансформисати један у други без губитка информација. У овом раду највише ћемо се фокусирати на текстуални формат и под међурепрезентацијом ћемо најчешће мислити на овај формат, који се може окарактерисати као асемблерски језик у највећој мери независан од специфичне платформе.

Да бисмо приказали како изгледа текстуални формат LLVM међурепрезентације, превешћемо[2] наредне две функције написане у програмском језику C.

```
unsigned add1(unsigned a, unsigned b) {
  return a+b;
}
// Rekurzivna funkcija za sabiranje 2 broja.
unsigned add2(unsigned a, unsigned b) {
  if (a == 0) return b;
  return add2(a-1, b+1);
}
```

[1] често се назива и бајткôд формат

[2] сви примери у раду преведени су компајлером Clang са нивоом оптимизације O3.

Ове две функције се преводе у наредни кôд.

```
define i32 @add1(i32 %a, i32 %b) {
entry:
  %tmp1 = add i32 %a, %b
  ret i32 %tmp1
}

define i32 @add2(i32 %a, i32 %b) {
entry:
  %tmp1 = icmp eq i32 %a, 0
  br i1 %tmp1, label %done, label %recurse

recurse:
  %tmp2 = sub i32 %a, 1
  %tmp3 = add i32 %b, 1
  %tmp4 = call i32 @add2(i32 %tmp2, i32 %tmp3)
  ret i32 %tmp4

done:
  ret i32 %b
}
```

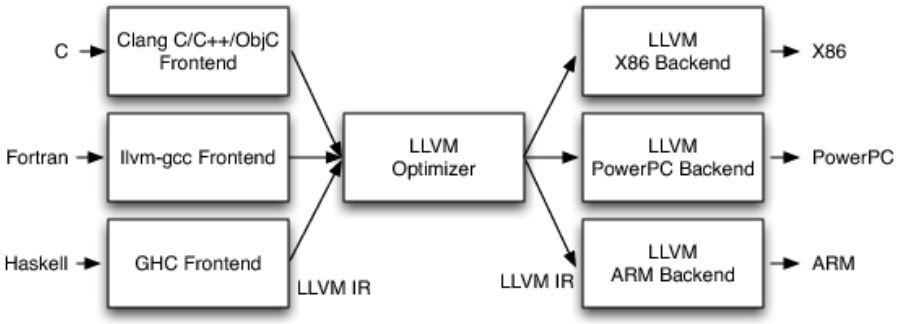
LLVM међурепрезентација је асемблерски формат сличан апстрактном RISC [4] скупу инструкција, са додатним структурама вишег нивоа.

Као што видимо у овом примеру, међурепрезентација подржава линеарне секвенце једноставних инструкција као што су сабирање, одузимање, наредбе условног и безусловног скока, упоређивање итд. Све ове инструкције су у троадресној форми, што значи да могу примити највише два регистра као улаз и резултат, ако постоји, уписати у трећи регистар. Међурепрезентација је строго типизирана (на пример i32 означава тридесетдвобитни целобројни број), док се позив функције означава кључном речи call, а враћање резултата са ret. LLVM не користи фиксан број регистара, већ има неограничен број променљивих које почињу карактером %. Функције и глобалне променљиве пре свог назива садрже карактер @. Међурепрезентација садржи и лабеле.

2.2 Компајлер LLVM

Процес компилације у инфраструктури LLVM започиње у предњем делу који производи међурепрезентацију, која се затим шаље алату за оптимизацију

који трансформише кôд кроз велики број оптимизација. Потом се трансформисани кôд преводи у асемблерски кôд на жељеној архитектури, и на крају се асемблерски кôд преводи у машински. Овај процес, наравно поједностављен, можемо видети на слици 2.1.



Слика 2.1: процес компилације LLVM

Предњи део компајлера

Предњи део компајлера задужен је за парсирање, валидацију и проналазак грешака у изворном кôду, затим за превођење парсираног кôда у међурепрезентацију LLVM. Превођење се обично изводи, прво изградњом апстрактног синтаксног стабла (AST [5]), а затим и превођењем AST-а у међурепрезентацију. У суштини сваки програмски језик, уколико имплементира предњи део који може да изгенерише међурепрезентацију LLVM, може користити алат за оптимизацију или задњи део LLVM-а. Најбитнији пројекат који имплементира предњи део LLVM-а је Clang. Clang је предњи део компајлера за програмске језике C, C++ и Objective C.

Алат за оптимизацију

Алат за оптимизацију (opt [6]) дизајниран је тако да на улазу прима LLVM међурепрезентацију и изврши оптимизације над међурепрезентацијом. Овај алат је организован у више низова оптимизационих пролаза, тако да је излаз једне оптимизације улаз у другу. Неки од примера оптимизационих пролаза су уметање, елиминација мртвог кôда, реалокација израза, размотавање петљи итд. Од нивоа оптимизације зависе и оптимизациони пролази који ће бити

покренути. У наставку ћемо приказати основне нивое оптимизације у случају Clang-a [28][3]

1. O0 – основне оптимизације, корисне због брзине компајлирања и лакшег дебаговања. Једна од опција које се задају алату за оптимизацију на овом нивоу је -always-inline (извршиће се уметање само оних функција које су експлицитно означене са always-inline).

2. O1 – додаје велики број пролаза у односу на ниво 0, неке од опција које се задају алату за оптимизацију су: –strip-dead-prototypes (елиминација декларација функција за које не постоји имплементација), -loop-rotate (ротација петље), -loop-deletion (елиминација петљи чије извршавање не утиче на повратну вредност функције), -loop-unroll (одмотавање петље).

3. O2 – овај ниво садржи све пролазе као ниво O1 уз додате опције: -inline (уметање функције), -gvn (елиминација редундантних инструкција), -constmerge (спаја поновљене глобалне константе у једну заједничку). У овом нивоу се избацује опција -always-inline који ниво O0 додаје.

4. O3 – најећи ниво оптимизације, генерише извршни фајл који се најбрже извршава али по цену времена компилације. Садржи пролазе као ниво O2 уз -argpromotion (функцијама које имају аргументе показивачког типа и уколико се само користи вредност на коју реферише показивач, односно не мења се вредност у меморији, овај пролаз ће заменити показивачки тип са типом на који показује, када је то могуће).

Задњи део компајлера

Задњи део LLVM-a генерише од међурепрезентације машински кôд за специфичну архитектуру. Главна компонента back-end-a је генератор кôда (eng. LLVM code generator [7]) који користи сличан приступ као алат за оптимизацију, то јест дели генерисање машинског кôда на мање пролазе, који имају за циљ генерисање најбољег могућег кôда. Најбитнији пролази су бирање инструкција, алокација регистара, распоређивање (eng. scheduling). LLVM може генерисати код за велики број архитектура, неке од њих су: x86, ARM, PowerPC, SPARC.

[3]Наведени пролази алата за оптимизацију су везани за верзију 6.0 Clang-a

2.3 Предности LLVM-а

Инфраструктура LLVM-а је бесплатна и њен изворни кôд је у потпуности доступан, што је навело не само истраживаче са универзитета, већ и велики број компанија да учествују у развоју, тако да данас значајан број људи активно учествује у одржавању и унапређивању овое инфраструктуре. Модуларни дизајн омогућава лако мењање постојећих алата или додавање нових. Захваљујући овом дизајну врло лако је додати нови предњи део компајлера, задњи део или оптимизациони пролаз. Такође, LLVM подржава и:

1. JIT компилацију [8]
2. оптимизацију током линковања (LTO [10])

Глава 3

Оптимизација целовитог програма

Обично изворни кôд програма делимо у више компилационих јединица (eng. compilation unit) [1]. Компајлер чита фајл по фајл и за сваки генерише њему одговарајући објектни фајл, то јест свакој компилационој јединици одговара један објектни фајл. Овако чинимо наш кôд читљивијим, омогућавамо паралелелно компајлирање више фајлова али и избегавамо потребу за компајлирањем целог програма за сваку промену у узворном кôду. Овакав приступ има и лошу страну, пошто компајлер преводи фајл по фајл, он нема информације о кôду који се налази у другим компилационим јединицама. Због тога што компајлер не види тела функција имплементираних у другим компилационим јединицама, не може у потпуности да сагледа логику и изврши жељене оптимизације. Овај проблем се може решити уз помоћ линкера, оптимизацијом током линковања (LTO) или спајањем свих фајлова у један и извршавањем оптимизација на једном великом фајлу (eng. unity build [9]).

У наредном примеру ћемо показати због чега оптимизација целовитог програма може бити корисна.

```
//a.h                                   //a.cpp
void do_nothing();                      void do_nothing(){}

//main.cpp
#include "a.hpp"

int main(){
    for (int i = 0; i < 1'000'000'000; i++){
        do_nothing();
```

[1] често се назива и јединица превођења

```
    }
}
```

Listing 3.1: Primer pozivanja funkcije bez tela

У листингу 3.1 функција do_nothing има празно тело. Уколико овај кôд преведемо са оптимизацијом -O3, без оптимизације целовитог програма, добићемо резултат приказан у листингу 3.2:

```
clang++  main.cpp a.cpp  -O3
time ./a.out
real    0m1,022s
user    0m1,014s
sys     0m0,000s
```

Listing 3.2: Rezultat bez optimizacije celovitog programa

Може се приметити да је рачунару било потребно више од једне секунде да изврши програм који не ради ништа. У наставку ћемо исте фајлове превести са оптимизацијом целовитог програма:

```
clang++  main.cpp a.cpp  -O3 -flto=full
time ./a.out
real    0m0,003s
user    0m0,003s
sys     0m0,000s
```

Listing 3.3: Rezultat sa optimizacijom celovitog programa

Разлика у времену извршавања је незанемарљива, као што се може видети у листингу 3.3. У листингу 3.4 имамо приказ међурепрезентације LLVM без и са укљученом оптмизацијом целовитог програма и биће анализиране разлике између њих.

```
; Function Attrs: norecurse uwtable
define i32 @main() local_unnamed_addr #0 !dbg !9 {
  call void @llvm.dbg.value(metadata i32 0,
  metadata !14, metadata !DIExpression()), !dbg !16
  br label %2, !dbg !17

; <label>:1:                                      ; preds = %2
  ret i32 0, !dbg !18

; <label>:2:                                      ; preds = %2, %0
  %3 = phi i32 [ 0, %0 ], [ %4, %2 ]
```

```
  call void @llvm.dbg.value(metadata i32 %3,
  metadata !14, metadata !DIExpression()), !dbg !16
  tail call void @_Z10do_nothingv(), !dbg !19
  %4 = add nuw nsw i32 %3, 1, !dbg !22
  call void @llvm.dbg.value(metadata i32 %4,
  metadata !14, metadata !DIExpression()), !dbg !16
  %5 = icmp eq i32 %4, 1000000000, !dbg !23
  br i1 %5, label %1, label %2, !dbg !17, !llvm.loop
}

; Function Attrs: nounwind readnone speculatable
declare void @llvm.dbg.value(metadata, metadata, metadata)

; Function Attrs: norecurse nounwind readnone uwtable
define void @_Z10do_nothingv() local_unnamed_addr #2
 !dbg !26 {
  ret void, !dbg !29
}
```

Listing 3.4: Међуреprezentacija bez optimizacije celovitog programa

```
; Function Attrs: norecurse nounwind readnone uwtable
define dso_local i32 @main() local_unnamed_addr #0
!dbg !9 {
  call void @llvm.dbg.value(metadata i32 0,
  metadata !14, metadata !DIExpression()), !dbg !16
  ret i32 0, !dbg !17
}

; Function Attrs: nounwind readnone speculatable
declare void @llvm.dbg.value(metadata, metadata, metadata)
```

Listing 3.5: Међуреprezentacija sa optimizacijom celovitog programa

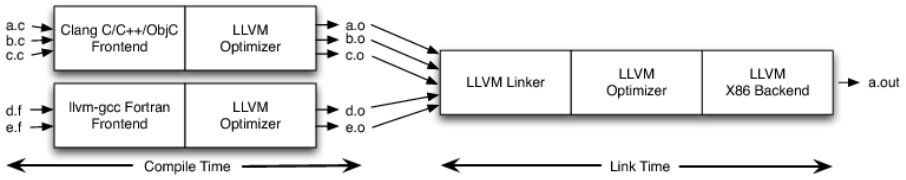
У листингу 3.4, где није укључена оптимизација целовитог програма, компајлер нема информацију како изгледа функција do_nothing и компајлер генерише кôд који је у петљи милион пута позива. У листингу 3.5, када је укључена оптимизација целовитог програма, компајлер има информацију о телу те функције, упознат је са тим да она не ради ништа, тако да може да оптимизује не само позивање те функције, односно да је уметне, већ може да уклони комплетну петљу, јер после уметања функције do_nothing тело петље остаје празно. Видимо да је елиминисана и функција do_nothing из извршног про-

грама и да је унутар функције main остало само враћање резултата. О уметању и елиминацији мртвог кôда биће више речи у наставку.

Оптимизација целовитог програма се такође може користити за детекцију недефинисаног понашања, када компајлер нема увид у цео кôд у јединици превођења и без активне оптимизације целовитог програма компајлер не може програмеру да укаже на грешку, чак и када су сва обавештења укључена (-Wall). У компајлеру gcc је ова подршка имплементирана, што можемо видети у раду *Undefined behavior in theory and practice* [26], тако да се у будућности ова функционалност очекује и у компајлеру clang.

3.1 Ортимизација током линковања

Захваљујући модуларном дизајну LLVM-а и чињеници да можемо компајлирати део кôда, сачувати резултат и наставити компилацију касније без губитака информација слику 2.1 можемо допунити додавањем линкера и оптимизацијама током процеса линковања.



Слика 3.1: LLVM процес компилације са подршком линкера

У наставку ћемо објаснити због чега је линкер користан у оптимизацији целовитог програма.

Главни задатак линкера је да све објектне фајлове споји у један фајл, извршни фајл или дељену библиотеку. Да би испунио овај задатак линкер прво мора да извши реалокацију симбола и резолуцију симбола. Симболи могу бити глобалне променљиве, функције, класе итд. Сваки објектни фајл садржи табелу симбола у којој се налазе сви симболи који могу бити дефинисани у истом објектном фајлу или у неком другом. Уколико симбол није дефинисан унутар објектног фајла он ће у табели симбола бити означен као „extern”, у супротном биће означен као „import”. Да би се успешно превео програм у извршни фајл, линкер прво мора да пронађе све недостајуће симболе у свим објектним фајловима и повеже их са тачно једном дефиницијом, то јест да изврши ре-

золуцију симбола. Затим линкер траба да да упише адресе симбола (такође задатак линкера је да и неким импортованим симболима промени адресу, уколико је компајлер то назначио), односно да изврши реалокацију симбола. Због ових својстава линкер има пресудну улогу у оптимизацији целовитог програма. Линкер има увид у све табеле симбола и алат за оптимизацију може то искористити за оптимизације делова кôда који су му пре били „невидљиви".

У наставку приказаћемо интеракцију између линкера и алата за оптимизацију. Оптимизација током линковања у инфраструктури LLVM садржи четири фазе:

1. Читање биткôд фајлова
2. Резолуција симбола
3. Оптимизовање биткôд фајлова
4. Резолуција симбола након оптимизације

Читање биткôд фајлова

Сви објектни фајлови долазе до линкера, који из њих чита и сакупља информације о симболима, који су присутни у фајловима. Ови фајлови могу бити у форми LLVM биткôд фајлова или стандардних објектних фајлова (eng. native object files). Линкер већ има могућност за третирање објектних фајлова. Да би могао правилно да чита и LLVM биткôд фајлове потребна му је помоћ, а то му омогућава libLTO [11]. libLTO је библиотека који је намењена за коришћење од стране линкера. libLTO има стабилан интерфејс, тако да је могуће користити LLVM алат за оптимизацију, без потребе за излагањем интерног LLVM кôда. Такође, још једна предност ове библиотеке је то што можемо мењати LLVM LTO кôд независно од линкера, то јест не морамо за сваку промену кôда који имплементира оптимизације мењати и линкер.

Уколико линкер добије објектни фајл, он већ зна да чита тај фајл и додаће симболе у глобалну табелу симбола. Уколико је у питању LLVM биткôд фајл, линкер ће позвати функције
lto_module_get_symbol_name и lto_module_get_symbol_attribute libLTO библиотеке да би добио све дефинисане симболе, затим ће те симболе, као у случају стандардног објектног фајла, додати у глобалну табелу симбола.

Резолуција симбола

Као што је већ објашњено у фази читања биткôд фајлова, линкер покушава да разреши све симболе помоћу глобалне табеле симбола. Уколико је укључена опција елиминације мртвог кôда линкер чува листу симбола који су коришћени у осталим објектним фајловима, такозвани живи симболи. Опција елиминације мртвог кôда је подразумевано укључена уколико се користи оптимизација током линковања.

Оптимизација биткôд фајлова

У овој фази линкер користи информације из глобалне табеле симбола, и пријављује живе симболе алату за оптимизацију функцијом lto_codegen_add_must_preserve_symbol. Затим линкер позива алат за отимизацију и генератор кôда над биткôд фајловима функцијом lto_codegen_compile, чији је резулат објектни фајл који је настао спајањем више биткôд фајлова, са примењеним оптимизацијама на њима. Примећујемо да је оптимизације могуће извршити искључиво на биткôд фајловима, то јест објектни фајлови се не оптимизују на овај начин.

Резолуција симбола након оптимизације

У овој фази линкер чита оптимизоване објектне фајлове и ажурира табелу симбола уколико има неких промена. На пример уколико је укључена елиминација мртвог кôда, линкер може да избаци неке симболе из табеле. У овој фази нема више биткôд фајлова, то јест сви фајлови су објектни фајлови. Након оптимизације наставља се са процесом изградње кôда као да оптимизације није ни било.

3.2 Оптимизација целовитог програма без подршке линкера

Приступ отимизације целовитог програма са линкером захтева линкер Gold [12], који у себи има подршку за библиотеку libLTO. На неким системима овај линкер није доступан и ту је немогуће извршити стандардну оптимизацију током линковања. Алтернативни приступ је спајање свих LLVM биткôд фајлова у

један биткôд фајл и извршавање оптимизација над тим фајлом. Ово је могуће захваљујући LLVM алату llvm-link [13].

Овим приступом добијамо исте перфомансе као са приступом где имамо подршку линкера, са тим што овај приступ неће радити уколико сви фајлови нису биткôд фајлови, односно не ради са објектним фајловима.

3.3 Уметање функција

Уобичајена пракса у програмирању је издвајање кôда који се понавља у засебне функције. Издвајање кôда је корисно зато што на тај начин избацујемо копирање истог кôда на више места у програму. На тај начин не само да повећавамо читљивост програма, већ и смањујемо могућност грешака, које су честе при копирању кôда. Са друге стране, позиви функција могу бити захтевни што се тиче времена извршавања. Када се функција позове долази до креирања новог стек оквира, померања показивача инструкција на почетак те функције, чувања тренутног стања позиваоца функције у регистрима и слично. Такође, кôд функције може бити ван кеша инструкција, што може битно утицати на време извршавања програма. Решење ових проблема је уметање функција [14] (eng. function inlining). Уметање је замена позива функције у компајлираном кôду целокупним телом позване функције. Поред тога што уметање елиминише утрошено време позива функције, оно такође омогућава компајлеру да генерише оптималан кôд. Пошто је цео кôд функције уметнут, компајлер може извршити оптимизације у већем блоку, што некада може довести до значајних убрзања.

Показано је да уметање функција може бити корисно и намеће се логично питање – када је могуће извршити уметање? Неке функције можемо одмах елиминисати из списка кандидата за уметање, уколико имамо дељену динамичку библиотеку, компајлер нема информацију о кôду функције тако да је не може уметнути. Сличан проблем је са функцијама које се налазе у другим објектним фајловима.

Видели смо да уметање има велики број предности, али да га није могуће увек урадити, да ли онда увек уметнути функцију када је то могуће? Одговор је не. Поред великог броја предности, уметање има и неке мане. Једна од мана је повећање величине извршног фајла, поготово када функције имају велики број инструкција. Због тога ипак мора постојати компромис између перфоманси

програма и величине извршног фајла.

Функције са великим бројем инструкција могу негативно да утичу на перфомансе, тако што утичу на кеш инструкција, јер велики број инструкција руши локалност референци. На пример уколико уметнему функцију са великим бројем инструкција унутар петље, врло је могуће да тај блок више не стаје у кеш инструкција, и онда при свакој итерацији петље имамо промашај кеша. Због тога се обично избегава уметање оваквих функција. Такође, функције које се скоро никада не позивају, нема смисла уметати. Уметањем таквих функција нећемо добити никакве предности у перфомансама, само можемо повећати величину извршног фајла.

Да би компајлер генерисао оптмималан кôд, користи хеуристике, преко којих одређује да ли неку функцију треба уметнути или не. Битне информације које користе хеуристике су колико функција има инструкција, колико пута се позива у току програма, да ли је функција коришћена у осталим објектним фајловима и слично.

Поред ове статичке анализе сложености функција, где користимо фиксиране границе (eng. threshold) за број иснтрукција, позива итд. и тако одређујемо да ли треба да уметнемо функцију, постоји и динамчка анализа. Динамичка анализа користи информације које се добијају приликом профајлирања програма. На овај начин можемо добити прецизније информације и самим тим боље перфомансе програма после оптимизације, али само у случају да тестно окружење програма симулира реалну ситуацију у којој ће се програм извршавати. У супротном можемо добити лошије перфомансе него статичком анализом. Профајлирањем можемо открити делове кôда који се чешће извршавају, и компајлер поклања посебну пажњу оптимизовању и уметању функција које се налазе у тим деловима.

Програмер може у изворном кôду сигнализирати компајлеру да изврши уметање атрибутом `always_inline`, на системима `Unix`, али ни то не гарантује да ће на крају функција заиста бити уметнута.

У наставку биће приказан један пример где је могуће извршити уметање уколико је укључена оптимизација целовитог програма.

```
//square.hpp
int square(int);

//square.cpp
int square(int a){
```

```
    return a *a;
}

//main.cpp
#include "square.hpp"
#include <iostream>

int main(){
    int result = 0;

    for (int i = 0; i < 100; i++){
        result+= square(i);
    }
    std::cout << result;
}
```

Listing 3.6: Primer umetanja funkcije

У наставку приказаћемо разлике међурепрезентације LLVM без и са оптимизације целовитог програма. Преведен је кôд из листинга 3.6.

```
 Function Attrs: norecurse uwtable
define i32 @main() local_unnamed_addr #4 !dbg !966 {
  call void @llvm.dbg.value(metadata i32 0,
  metadata !968, metadata !DIExpression()), !dbg !971
  call void @llvm.dbg.value(metadata i32 0,
  metadata !969, metadata !DIExpression()), !dbg !972
  br label %3, !dbg !973

; <label>:1:                                      ; preds = %3
  %2 = tail call dereferenceable(272)
  %"class.std::basic_ostream"*
  @_ZNSolsEi(%"class.std::basic_ostream"*
  nonnull @_ZSt4cout, i32 %7), !dbg !974
  ret i32 0, !dbg !975

; <label>:3:                                      ; preds = %3, %0
  %4 = phi i32 [ 0, %0 ], [ %8, %3 ]
  %5 = phi i32 [ 0, %0 ], [ %7, %3 ]
  call void @llvm.dbg.value(metadata i32 %5,
  metadata !968, metadata !DIExpression()), !dbg !971
  call void @llvm.dbg.value(metadata i32 %4,
  metadata !969, metadata !DIExpression()), !dbg !972
  %6 = tail call i32 @_Z6squarei(i32 %4), !dbg !976
```

```
  %7 = add nsw i32 %6, %5, !dbg !979
  %8 = add nuw nsw i32 %4, 1, !dbg !980
  call void @llvm.dbg.value(metadata i32 %8,
  metadata !969, metadata !DIExpression()), !dbg !972
  call void @llvm.dbg.value(metadata i32 %7,
  metadata !968, metadata !DIExpression()), !dbg !971
  %9 = icmp eq i32 %8, 100, !dbg !981
  br i1 %9, label %1, label %3, !dbg !973, !llvm.loop !982
}

; Function Attrs: nounwind readnone speculatable
declare void @llvm.dbg.value(metadata,
metadata, metadata) #5

declare dereferenceable(272)
 %"class.std::basic_ostream"*
 @_ZNSolsEi(%"class.std::basic_ostream"*, i32)
 local_unnamed_addr #1

; Function Attrs: nounwind readnone uwtable
define i32 @_Z6squarei(i32) local_unnamed_addr
 #6 !dbg !984 {
  call void @llvm.dbg.value(metadata i32 %0,
  metadata !986, metadata !DIExpression()), !dbg !987
  %2 = mul nsw i32 %0, %0, !dbg !988
  ret i32 %2, !dbg !989
}
```

Listing 3.7: Међуреprezentacija bez optimizacije celovitog programa

```
; Function Attrs: norecurse uwtable
define dso_local i32 @main() local_unnamed_addr
#4 !dbg !966 {
  call void @llvm.dbg.value(metadata i32 0,
  metadata !968, metadata !DIExpression()), !dbg !971
  call void @llvm.dbg.value(metadata i32 0,
  metadata !969, metadata !DIExpression()), !dbg !972
  %1 = tail call dereferenceable(272)
  %"class.std::basic_ostream"*
  @_ZNSolsEi(%"class.std::basic_ostream"* nonnull
  @_ZSt4cout, i32 328350), !dbg !973
  ret i32 0, !dbg !974
}
```

```
; Function Attrs: nounwind readnone speculatable
declare void @llvm.dbg.value(metadata,
 metadata, metadata) #5

declare dereferenceable(272)
%"class.std::basic_ostream"*
@_ZNSolsEi(%"class.std::basic_ostream"*, i32)
local_unnamed_addr #1
```

Listing 3.8: Међуреprezentacija sa optimizacijom celovitog programa

Ако погледамо међурепрезентацију из листинга 3.7, без оптимизације целовитог програма, видимо да је она јако слична изворном кôду, компајлер не види тело функције `square`, тако да не може неке значајније оптимизације да изврши. Са друге стране у листингу 3.8, када је укључена оптимизација целовитог програма, компајлер успева да уметне функцију. Овим се компајлер не само да генерише кôд који не садржи трошак позива функција, већ омогућава и остале оптимизације. Зато што сада имамо тело функције у блоку петље, компајлер увиђа да се петља извршава константан број пута и да нема потребе стално израчунавати исту приликом покретања програма, већ вредност промењљиве result може да се израчуна у току компилације програма, самим тим се компајлер елиминише петљу и кôд око ње. У позиву функције видимо резултат израчунавања :

```
tail call dereferenceable(272) %"class.std::basic_ostream"*
@_ZNSolsEi(%"class.std::basic_ostream"
* nonnull @_ZSt4cout, i32 328350)
```

Такође, више нема потребе за постојањем функције `square` (више се нигде не користи) и она је избрисана из извршног фајла.

3.4 Елиминација мртвог кôда

Елиминација мртвог кôда [16] (eng. dead code elimination) је компајлерска оптимизација која елиминише кôд који не утиче на резултат извршавања програма. Уклањање мртвог кôда има многе предности: смањује величину извршног програма, побољшава локалност инструкција, уклањањем непотребних инструкција такође повећава брзину извршавања програма. Без укључене

оптимизације целовитог програма компајлер може да елинише локалне променљиве, уметнуте статичке функције као и статичке глобалне променљиве. То ради једноставним праћењем позива свих статичких глобала и сврставањем истих у живе или мртве скупове, у зависности да ли се глобал користи или не. Глобали су идентификатори који имају глобални опсег (eng. scope). То значи да су они видљиви унутар целог програма, то јест свака јединица превођења може видети глобал дефинисан у некој другој јединици превођења. Због тога сме постојати само једна дефиниција глобала. Глобали могу бити функције, променљиве, класе... Све глобале из мртвог скупа, на крају оптимизационих пролаза, можемо елиминисати. Са укљученом оптимизацијом целовитог програма можемо избацити не само статичке глобалне променљиве или функције, него све глобале који се не користе. Овај поступак се извршава током линковања и описан је у секцији 3.1.

Пример елиминације мртвог кôда смо већ видели у листингу 3.8 у којем је компајлер уклонио функцију `square` јер се није користила. Програм у листингу 3.9 приказаће ово на неколико очигледнијих примера:

```
//a.hpp
int foo1(void);
void foo2(void);
void foo4(void);

//a.cpp
#include "a.hpp"

static signed int i = 0;

void foo2(void) {
  i = -1;
}

static int foo3() {
  foo4();
  return 10;
}

int foo1(void) {
  int data = 0;

  if (i < 0)
```

```
    data = foo3();

  data = data + 42;
  return data;
}

//main.cpp
#include <iostream>
#include "a.hpp"

void foo4(void) {
  std::cout << ("Hi\n");
}

int main() {
  return foo1();
}
```

Listing 3.9: Primer eliminacije mrtvog koda

```
 Function Attrs: uwtable

declare dereferenceable(272)
%"class.std::basic_ostream"* @_ZSt16__
ostream_insertIcSt11char_traitsIcEERSt13basic_
ostreamIT_T0_ES6_PKS3_l
(%"class.std::basic_ostream"*
dereferenceable(272), i8*, i64) local_unnamed_addr #1

; Function Attrs: uwtable
define void @_Z4foo4v() local_unnamed_addr #0 !dbg !983 {
  call void @llvm.dbg.value(metadata
  %"class.std::basic_ostream"* @_ZSt4cout,
  metadata !984, metadata !DIExpression()), !dbg !1048
  call void @llvm.dbg.value(metadata i8*
  getelementptr inbounds ([4 x i8], [4 x i8]*
   @.str, i64 0, i64 0),
  metadata !993, metadata !DIExpression()), !dbg !1050
  %1 = tail call dereferenceable(272)
  %"class.std::basic_ostream"*
   @_ZSt16__ostream_insertIcSt11char_
  traitsIcEERSt13basic_ostreamIT_T0_ES6_PKS3_l
  (%"class.std::basic_ostream"* nonnull dereferenceable(272)
```

```
  @_ZSt4cout, i8* nonnull getelementptr
  inbounds ([4 x i8], [4 x i8]* @.str, i64 0, i64 0),
  i64 3), !dbg !1051
  ret void, !dbg !1053
}

; Function Attrs: norecurse uwtable
define i32 @main() local_unnamed_addr #5 !dbg !1054 {
  %1 = tail call i32 @_Z4foo1v(), !dbg !1055
  ret i32 %1, !dbg !1056
}

; Function Attrs: norecurse nounwind uwtable
define void @_Z4foo2v() local_unnamed_addr #6 !dbg !1057 {
  store i1 true, i1* @_ZL1i, align 4
  ret void, !dbg !1058
}

; Function Attrs: uwtable
define i32 @_Z4foo1v() local_unnamed_addr #0 !dbg !1059 {
  call void @llvm.dbg.value(metadata i32 0,
   metadata !1061, metadata !DIExpression()), !dbg !1062
  %1 = load i1, i1* @_ZL1i, align 4
  br i1 %1, label %2, label %3, !dbg !1063

; <label>:2:                                      ; preds = %0
  tail call void @_Z4foo4v(), !dbg !1064
  call void @llvm.dbg.value(metadata i32 10,
  metadata !1061, metadata !DIExpression()), !dbg !1062
  br label %3, !dbg !1068

; <label>:3:                                      ; preds = %2, %0
  %4 = phi i32 [ 52, %2 ], [ 42, %0 ]
  call void @llvm.dbg.value(metadata i32 %4,
  metadata !1061, metadata !DIExpression()), !dbg !1062
  ret i32 %4, !dbg !1069
}
```

Listing 3.10: Међуреprezentacija bez optimizacije celovitog programa

```
; Function Attrs: norecurse nounwind readnone uwtable
define dso_local i32 @main() local_unnamed_addr #4
!dbg !982 {
```

```
  ret i32 42, !dbg !983
}
```

Listing 3.11: Међуреprezentacija sa optimizacijom celovitog programa

У листингу 3.10, без оптимизације целовитог програма, видимо да је компајлер уметнуо функцију `foo3` и она се не налази унутар извршног фајла. Компајлер је успео да елиминише функцију јер је она статичка, али остале нису тако да се оне налазе у извршном фајлу. Видимо да је и кôд међурепрезентације сличан изворном, тако да компајлер поред уметања, није успео да изврши неке веће оптимизације.

Са друге стране, у листингу 3.11, са оптимизацијом целовитог програма, видимо да је у извршном фајлу остала само функција `main`, која враћа вредност 42. Да бисмо утврдили како се то догодило, пролази се кроз цео процес оптимизације овог програма, са укљученом оптимизацијом целовитог програма.

Линкер прво препознаје да се функција `foo2` не користи нигде у програму, шаље ту информацију компајлеру (конкретно алату за оптимизацију) и он је брише. Чим обрише функцију, алат за оптимизацију види да услов `i < 0` никада није испуњен, тако да може да обрише и тај део кôда али и функцију `foo3`, јер се она сада више неће користити. Линкер сада препознаје да се функција `foo4` не користи више тако да се и она брише. На крају остаје само функција `foo1` која увек враћа вредност 42, то алат за оптимизацију препознаје, уметне је, брише и враћа вредност 42 као повратну вредност функције `main`.

3.5 Девиртуализација

Девиртуализација [17] (eng. devirtualization) је поступак замене виртуалних позива функција непосредним позивима. Виртуални позиви функција су спорији од непосредних, што у системима у којима је брзина извршавања програма кључна може да буде велики проблем. Због тога је пожељно извршити девиртуализацију кад год је то могуће. Девиртуализација се најефикасније може извршити приликом оптимизације целовитог програма, али постоје три случаја у којима компајлер може да закључи да је девиртуализација могућа, без увида у целокупан програмски кôд. Прво ће бити описани ти случајеви.

Познат динамички тип објекта

Уколико је компајлеру познат динамички тип објекта при компајлирању, он може да девиртуализује позив функције.

```
#include <iostream>
struct Base{
    virtual int f(){return 1;}
};

struct Derived : Base{
    int f() override {return 2;}
};

int main(){
    Derived d;
    Base * b = new Base();
    std::cout << b->f();
    std::cout << d.f();
}
```

Listing 3.12: Poznat dinamički tip objekta

```
; Function Attrs: norecurse uwtable
define i32 @main() local_unnamed_addr #4 !dbg !964 {
  call void @llvm.dbg.value(metadata %struct.Derived* undef,
   metadata !966, metadata !DIExpression()), !dbg !985
  %1 = tail call dereferenceable(272)
  %"class.std::basic_ostream"* @_ZNSolsEi
  (%"class.std::basic_ostream"* nonnull @_ZSt4cout, i32 1),
  !dbg !986
  call void @llvm.dbg.value(metadata %struct.Derived* undef,
  metadata !966, metadata !DIExpression()), !dbg !985
  %2 = tail call dereferenceable(272)
  %"class.std::basic_ostream"* @_ZNSolsEi
  (%"class.std::basic_ostream"* nonnull @_ZSt4cout,
  i32 2), !dbg !987
  ret i32 0, !dbg !988
}
```

Listing 3.13: Међуреprezentacija programa iz listinga 3.12

У листинг 3.13 види се да компајлер успешно девиртуализује и умеће позив `f->b()` јер има информацију да је променљива `b` иницијализована на `new`

`Base()`. Променљива `d` је типа `Derived`, па ту и не постоји виртуелни позив и поред тога што је чланска функција `f` виртуална. [2]

Кључна реч final

У програмском језику C++ кључна реч `final` означава да функција не може бити преправљена (eng. override) у изведеној класи (уколико се кључна реч `final` налази у потпису функције) или да се из те класе не може извести нова класа (уколико се кључна реч `final` налази у потпису класе).

```
struct Base{
    virtual int f(){return 1;}
};

struct Derived final : Base{
    int f() override {return 2;}
};
int func(Derived *d){
    return d->f();
}
```

Listing 3.14: Ključna reč final

За разлику од примера у листингу 3.12 овде смо структуру `Derived` обележили кључном речју `final`. То говори компајлеру да ни у овој, али ни у било којој другој јединици превођења, не може постојати структура која је изведена из структуре `Derived`. То сазнање омогућава компајлеру да изврши девиртуализацију позива функције `f()`. Примећујемо када не бисмо експлицитно обележили Derived са `final` девиртуализација не би била могућа, јер компајлер не може да зна да ли у некој другој јединици превођења постоји структура изведена из структуре `Derived` и самим тим преправљена функција.

Унутрашња видљивост

Када кажемо да нека променљива или функција има унутрашњу видљивост [18] (eng. internal linkage) то значи да је она видљива само унутар своје јединице транслације. Што значи да уколико имамо структуру, или класу, која има унутрашњу видљивост, комапјлер може девиртуализовати позив функције јер је

[2] Када променљива није показивачког или референцног типа компајлер зна да позив не може бити виртуалан.

сигуран да она неће моћи бити преправљена. Овај принцип је сличан додавању кључне речи `final`. Један једноставан начин да наша класа добије унутрашњу видљивост јесте смештање исте у безимени простор имена [19] (eng. unnamed namespace). Овај принцип приказан је у листингу 3.15.

```
namespace{
struct Base{
    virtual int f(){return 1;}
};

struct Derived : Base{
    int f() override {return 2;}
};
}
```

Listing 3.15: Unutrašnja vidljivost

Девиртуализација помоћу оптимизације целовитог програма

Приказане су ситуације када компајлер без оптимизације целовитог програма може да изврши девиртуализацију. У већини ситуација нећемо наилазити на тако једноставне случајеве, а и често ће дефиниције виртуалних функција бити у другим јединицама транслације. Због тога што компајлер види цео програм, када је укључена оптимизација целовитог програма, он види и функције и класе, тако да може да изврши девиртуализацију агресивније. У наставку биће приказан пример у листингу 3.16, који има само један изворни фајл и његове одговарајуће међурепрезентације у листинзима 3.17 и 3.18.

```
#include <iostream>
struct Base
{
  virtual int f() { return 1; }
};

struct Derived : public Base
{
  int f() override { return 2; }
};

int g(Derived *obj)
```

```
{
  return obj->f();
}

int main()
{
  Derived *obj = new Derived();
  std::cout << g(obj);
}
```

Listing 3.16: Devirtualizacija primer

```
; Function Attrs: uwtable
define i32 @_Z1gP7Derived(%struct.Derived*)
 local_unnamed_addr #0 !dbg !964 {
  call void @llvm.dbg.value(metadata %struct.Derived* %0,
   metadata !985, metadata !DIExpression()), !dbg !986
  %2 = bitcast %struct.Derived* %0 to i32
  (%struct.Derived*)***, !dbg !987
  %3 = load i32 (%struct.Derived*)**,
  i32 (%struct.Derived*)*** %2,
  align 8, !dbg !987, !tbaa !988
  %4 = load i32 (%struct.Derived*)*,
  i32 (%struct.Derived*)** %3,
  align 8, !dbg !987
  %5 = tail call i32 %4(%struct.Derived* %0), !dbg !987
  ret i32 %5, !dbg !991
}

; Function Attrs: nounwind readnone speculatable
declare void @llvm.dbg.value(metadata,
 metadata, metadata) #4

; Function Attrs: norecurse uwtable
define i32 @main() local_unnamed_addr #5
!dbg !992 {
  %1 = tail call dereferenceable(272)
  %"class.std::basic_ostream"*
  @_ZNSolsEi(%"class.std::basic_ostream"* nonnull
  @_ZSt4cout, i32 2), !dbg !995
  ret i32 0, !dbg !996
}
```

Listing 3.17: Међуреprezentacija bez optimizacije celovitog programa

```
; Function Attrs: norecurse uwtable
define dso_local i32 @main() local_unnamed_addr #4 !dbg !964 {
  %1 = tail call dereferenceable(272)
  %"class.std::basic_ostream"*
  @_ZNSolsEi(%"class.std::basic_ostream"* nonnull
   @_ZSt4cout, i32 2), !dbg !984
  ret i32 0, !dbg !985
}
```

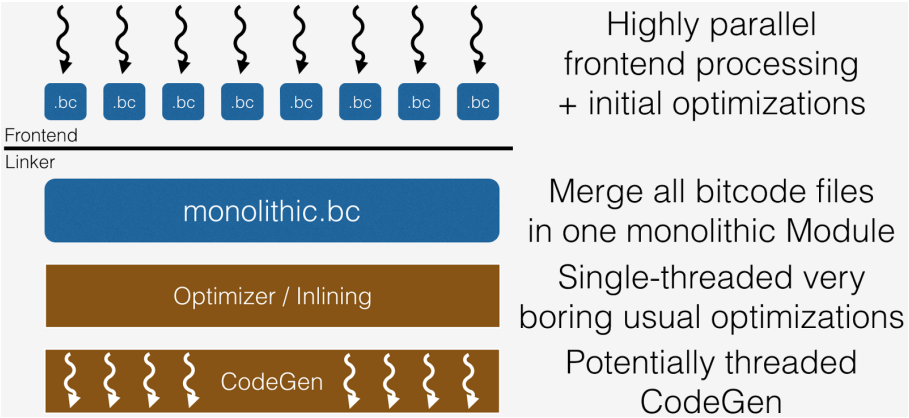
Listing 3.18: Међуреprezentacija sa optimizacijom celovitog programa

У листингу 3.17 постоје инструкције за узимање вредности виртуалног показивача и позивање виртуалне функције, што значи да компајлер није успео да девиртуализује позив иако види тело функције и класе. То се дешава због тога што компајлер не зна да ли у некој другој јединици превођења постоји класа која је изведена из класе `Derived` и због тога не може да изврши оптимизације. Са укљученом оптимизацијом целовитог програма компајлер види да не постоји класа изведена из класе `Derived` и компајлер успешно девиртуализује, а затим и умеће позив функције.

Глава 4

ThinLTO

У претходном поглављу описане су оптимизације које омогућава оптимизација целовитог програма и које могу значајно побољшати перфомансе нашег програма. Такође, видели смо како је имплементирана стандардна оптимизација током линковања, линкер добија биткôд фајлове, уместо објектних фајлова, затим се ти биткôд фајлови спајају у један и над тим фајлом се врше све оптимизације.



Слика 4.1: Стандардни процес оптимизације током линковања. Слика преузета са чланка *„ThinLTO: Scalable and Incremental LTO”* [21]

Стандардни приступ има неколико мана. Први проблем је то што се губи предност паралелног компајлирања, која постоји када није активна оптимизација целовитог програма. На слици 4.1 се види да постоји паралелно превођење изворних фајлова у биткôд фајлове али због спајања свих фајлова у један велики биткôд фајл, та предност се касније губи зато што све оптимизације над

тим фајлом морају да се раде без могућности парелелизације. Због тога компајлирање траје много дуже него без оптимизације целовитог програма. Такође, за сваку промену у било ком изворном фајлу, ми поново морамо испочетка вршити све оптимизације на обједињеном фајлу, што поново изузетно утиче не време превођења. Још један велики проблем овог приступа је то што сада у меморији морају да се налазе међурепрезентације свих компилационих јединица одједном, спојене у једну. Често је немогуће извршити оптимизацију целовитог програма, поготово на машинама које немају велику радну меморију. Приступ за решавање ових проблема је ThinLTO [20].

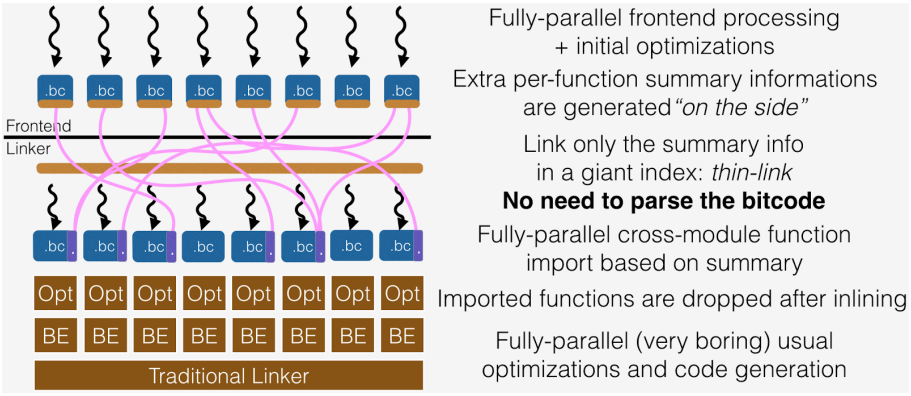
ThinLTO је нови приступ који омогућава сличне перфомансе при превођењу као када није укључена оптимизација целовитог програма, док задржава већину оптимизација и самим тим перфомансе извршног фајла као регуларна оптимизација целовитог програма. При оптимизацији ThinLTO, за разлику од класичне оптимизације целовитог програма, уместо учитавања биткôд фајлова и спајања у један, ThinLTO за сваку јединицу превођења чува кратак резиме за анализу у кораку линковања. Уз резиме чувају се и локације функција за касније инлајновање у друге јединице превођења. Кључна оптимизација коју ThinLTO омогућава је убацивање само оних функција које су потребне конкретном биткôд фајлу и које ће бити уметнуте у том биткôд фајлу. И тај поступак се ради за сваки биткôд фајл, што значи да нема спајања, већ се и даље поступак извршава паралелно. Процес оптимизације целовитог програма ThinLTO подељен је на три фазе:

1. Превођење – генеришу се међурепрезензације као и у случају стандардног процеса оптимизације целовитог програма, са тим што сада имамо и резиме уз сваку међурепрезентацију.

2. Линковање – линкер комбинује резимее из прошлог корака и врши анализу.

3. Задњи део – паралелна оптимизација и генерисање кôда.

Кључни део оптимизације ThinLTO дешава се у првој фази, а то су креирања резимеа. Свака глобална променљива и функција се налазе у резимеу, за ту јединицу транслације. Резиме садржи по једно поље за сваки симбол и у том пољу се налазе подаци који описују тај симбол. На пример, за функцију, у пољу унутар резимеа може да стоји њена видљивост, број инструкција које функција



Слика 4.2: ThinLTO процес оптимизације. Слика преузета са чланка *„ThinLTO: Scalable and Incremental LTO"*

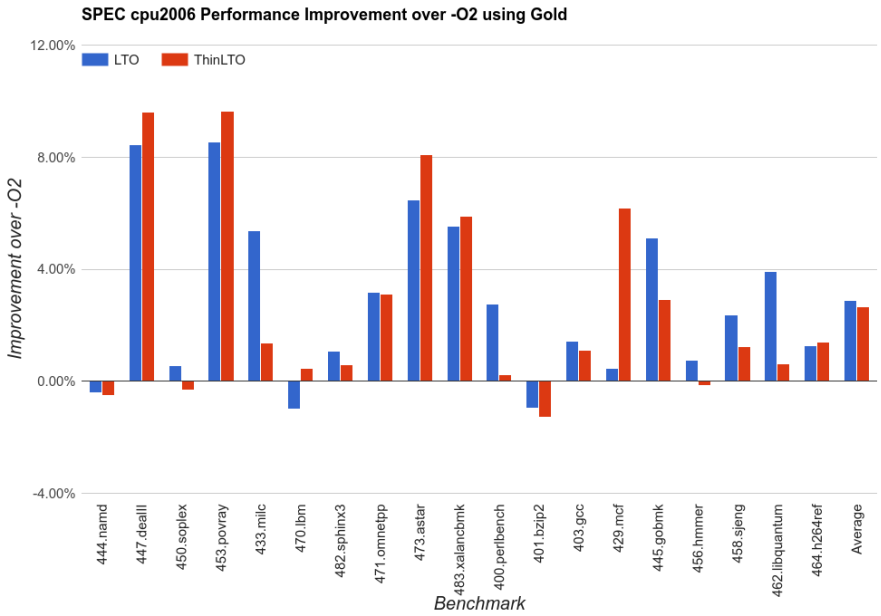
садржи, информације за профајлирање уколико су потребне и слично. Додатно, свака референца према другом симболу (позив друге функције, узимање адресе, приступање глобалу) се записује у резиме и тако се гради граф позива (eng. call graph). Ове информације омогућавају креирање комплетног графа током фазе линковања. ThinLTO је једноставно активирати, само је потребно додати `-flto=thin` у командној линији приликом компајлирања.

У наставку биће приказана разлиау између перформанси, меморијских захтева као и времена компајлирања између оптимизације током линковања и ThinLTO-а.

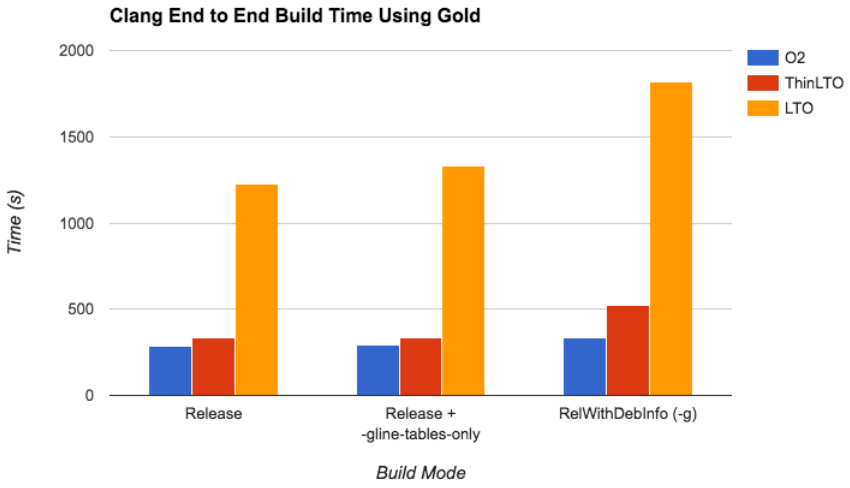
На слици 4.3 показано је да у просеку стандардна оптимизација даје за нијансу боље резултате, али постоје ситуације када ThinLTO надмашује стандардну оптимизацију.

На слици 4.4 види се да је време компајлирања програма са оптимизацијом ThinLTO јако слично времену превођења без оптимизације током линковања. Осетнија разлика између ова два начина компајлирања је приликом компајлирања програма који има информације потребне за дебаговање, али у току су унапређења у овом пољу, па се очекује смањење ове разлике у будућности. Што се тиче регуларне оптимизације током линковања, компајлирање програма је далеко спорије него код ThinLTO-а у сваком измереном случају.

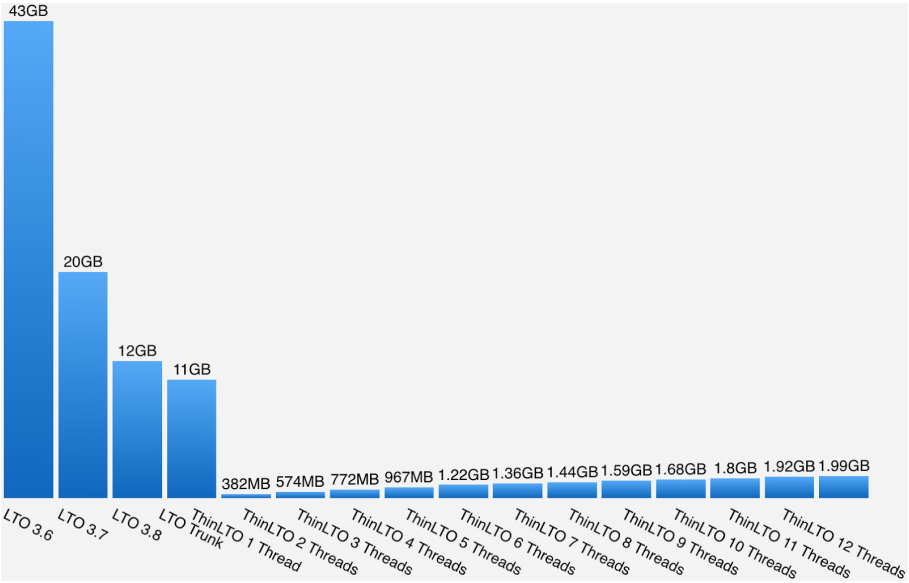
На слици 4.5 се види да је меморијска потрошња регуларне оптимизације током линковања далеко већа од меморијске потрошње ThinLTO-а. Такође, показано је да због великих меморијских захтева за регуларну оптимизацију током линковања морамо да имам јачи хардвер него за ThinLTO-а, да би уопште превели програм.



Слика 4.3: Разлика у перформансама између ThinLTO и регуларне оптимизације током линковања. Слика преузета са чланка *„ThinLTO: Scalable and Incremental LTO”*



Слика 4.4: Разлика у времену превођења програма између ThinLTO и регуларне оптимизације током линковања. Слика преузета са чланка *„ThinLTO: Scalable and Incremental LTO”*



Слика 4.5: Разлика у меморијским захтевима између ThinLTO и регуларне оптимизације током линковања. Слика преузета са чланка *„ThinLTO: Scalable and Incremental LTO”*

Глава 5

Алат за визуализацију промена

Алат имплементиран као део овог рада визуализује промене унутар LLVM међурепрезентације програма преведеног са и без оптимизације целовитог програма. Инспирација за овај алат био је compiler explorer[22]. Идеја иза compiler explorera била је приказивање одговарајућих асемблерских инструкција за сваку линију изворног кôда. Ова функционалност је корисна јер програмер тако има увид у кôд који је компајлер изгенерисао за њега и евентуално може да пронађе неку грешку у изворном кôду која је видљива тек након оптимизација, или да провери, колико је компајлер добро оптимизовао кôд након превођења. Касније, у compiler explorer додата је подршка и за приказ LLVM међурепрезентације јер је она разумљивија од конкретног асемблерског језика и садржи више опција за дебаговање. Compiler explorer приказује промене у контексту једне јединице транслације, што значи да кôд мора да се успешно компајлира, али не мора да се линкује. Са друге стране, алат за визуализацију имплементиран за потребе овог рада, приказује разлике у међурепрезентацијама које одговарају извршним фајловима добијених компилацијом са и без укључене оптимизације целовитог програма. За разлику од compiler explorer-а за коришћење овог алата неопходно је да сви улазни фајлови буду успешно линковани, али због тога он може да прикаже међурепрезентације извршних фајлова. Алат повезује линије изворног кôда са одговарајућим линијама у LLVM међурепрезентацији, одговарајуће линије су приказане истом бојом. Такође, постоји опција приказивања diff-а[24] између међурепрезентација са и без активне оптимизације целовитог програма. Тренутно, подржани су искључиво програми писани у програмском језику C++. Изворни кôд алата може се наћи на адреси[23].

Захтеви за покретање алата:

1. оперативни систем Linux
2. python3
3. python3-tk - за графички интерфејс
4. clang - за превођење програма
5. llvm - због алата llvm-dis[27] који врши конверзију из биткôд формата у читљиви формат
6. wllvm - за добијање међурепрезентације из извршног фајла
7. kompare - за графички приказ diff фајла

Пример покретања алата:

```
python3 -i {putanja_do_foldera_sa_.cpp_fajlovima}
-o {nivo_optimizacije('0', '1', '2', '3', '4',
                      'z', 'g', 'z', 'fast)}
```

Алат је написан у програмском језику Python и садржи две главне функционалности.

Прва функционалност је приказ LLVM међурепрезентације са и без активне оптимизације целовитог програма као и њихово мапирање са линијама у изворном кôду. Мапирање је имплементирано тако једној линији у изворном кôду придружује одговарајуће линије у међурепрезентацијама. Свако мапирање је обојено различитом бојом и обојене су само оне линије у изворном кôду које су се после свих оптимизација превеле у одговарајућу међурепрезентацију (линије које су избачене се не боје).

Друга функционалност је приказивање графичког diff-а унутар алата kompare[25]. Помоћу ове функционалности лако можемо видети који LLVM блокови су склоњени, промењени или додати. Да бисмо повезали линије између више различитих фајлова извршено је енкодирање „име фајла:линија изворног кôда:LLVMIR". Име фајла је обавезно да би направили разлику између истих линија кôда у две различите јединице транслације. При креирању diff фајла занемарени су:

1. имена променљивих – имена променљивих – током компилације имена променљивих које компајлер генерише унутар међурепрезентације нису стабилна, па су изостављана приликом креирања diff фајла;

2. линије кôда које служе за дибаг информације (!dbg број). Бројеви у два фајла не морају бити исти, а утичу на diff (слично као имена променљивих);

3. коментари.

Као што смо рекли, енкодиране линије користимо за креирање diff фајла. Уколико након завршетка алгоритма између неке две линије нема разлике, онда ту енкодирану линију задржавамо у diff фајлу и приказујемо на излазу. У супротном, приказаћемо линије које одговарају неенкодираним верзијама. То јест, заменићемо у фајлу енкодиране линије њиховим правим вредностима у међурепрезентацијама. На слици 5.4 на 55. линији видимо пример енкодирања. Видимо да је та линија из фајла main.cpp и да је то десета линија у том фајлу. Та линија је иста у обе верзије (без активне и оптимизације целовитог програма и са активном оптимизацијом целовитог програма). Док на пример енкодиране линије 59 и 46 нису исте, па су приказане њихове одговарајуће неенкодиране линије.

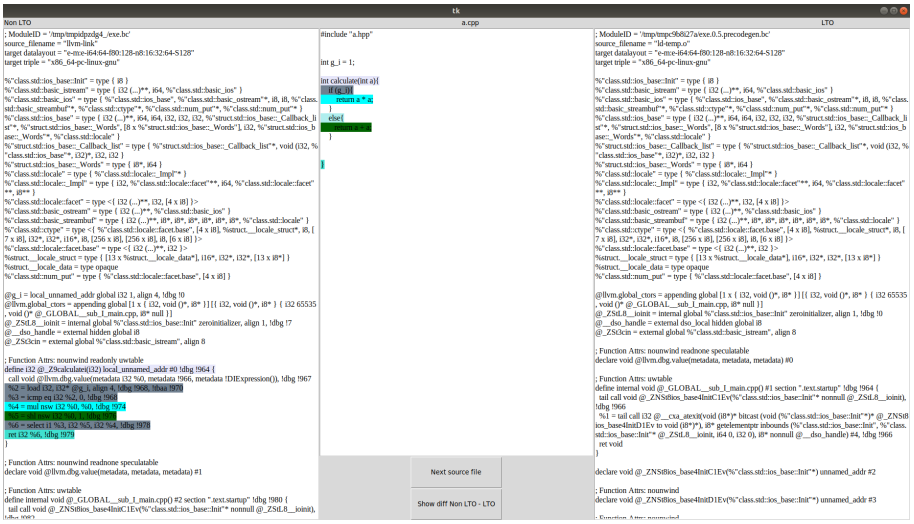
У наставку ћемо приказати алата над следећим фајловима.

```
// a.hpp
int calculate(int num);

//a.cpp
#include "a.hpp"


int g_i = 1;

int calculate(int a){
    if (g_i){
        return a * a;
    }
    else{
       return a + a;
    }

}

// main.cpp
#include "a.hpp"
#include <iostream>
```

```
int main(){
    int n;
    std::cin >> n;

    int result = 0;

    for (int i = 0 ; i < n; i++){
        result += calculate(i);
    }

    return result;
}
```

Primer 5.1

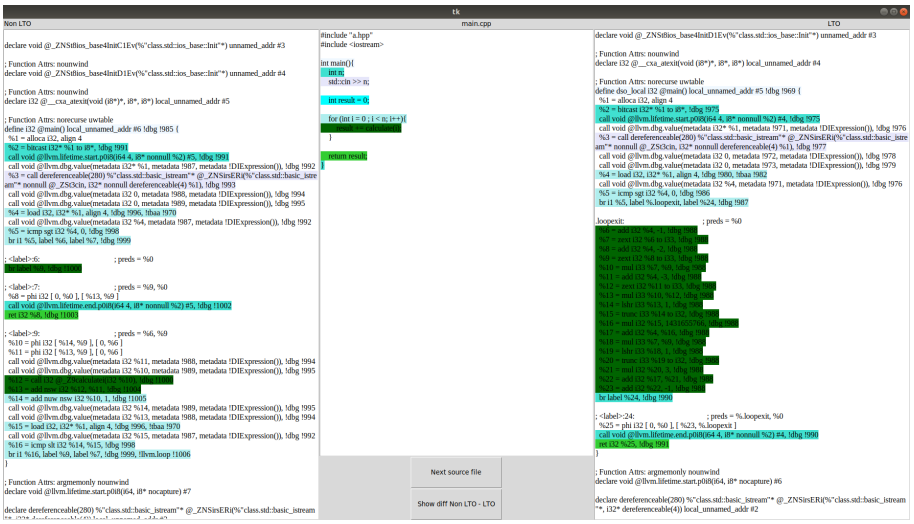
Када покренемо алат, добијамо следећу слику:



Слика 5.1: Први изворни фајл

У средњем прозору видимо да је тренутни изворни фајл, који повезујемо са одговарајућим међурепрезентацијама, фајл a.cpp. Са његове леве стране налази се прозор међурепрезентације где није извршена оптимизација целовитог програма, и обојене истим бојама одговарајуће линије у изворном и LLVM фајлу. Са десне стране се налази прозор са оптимизованом међурепрезентацијом где видимо да ништа није обојено, то је зато што је цео кôд a.cpp фајла оптимизован и не налази се у извршном фајлу.

Кликом на дугме Next source file, приказује се main.cpp фајл и одговарајуће мапирање линија. Наравно, више се не приказују мапирања из претходног



Слика 5.2: Други изворни фајл

изворног фајла, већ само из тренутног.

Кликом на дугме Show diff добијамо приказ разлика између неоптимизоване и оптимизоване верзије, приказане у алату kompare.

Слика 5.3: diff прва слика

У примеру 5.1 видимо да је са активном оптимизацијом целовитог програма, елиминисана глобална променљива g_i као и функција calculate. Затим видимо и разлике у функцији main. Због тога што не види тело функције у верзији без активне оптимизације целовитог програма, компајлер генерише кôд који позива функцију calculate. У верзији са активном оптимизацијом целовитог програма, видимо не само да је функција calculate инлајнована, већ је избрисан

Слика 5.4: diff друга слика

део функције који никада није доступан јер је вредност глобалне променљиве увек 1, а то се може видети тек после процеса линковања. Такође, видимо да је због инлајновања извршена и оптимизација петље, јер алат за оптимизацију схвата шта рачунамо и извршава израчунавање без потребе за петљом.

Глава 6

Закључак

У раду је представљена LLVM компајлерска инфраструктура са посебним акцентом на оптимизацију целовитог програма. Видели смо да оптимизација целовитог програма доноси нека значајна побољшања што се тиче времена извршавања и смањивања величине извршног фајла, али уз спорију компилацију програма и већим заузећем меморије током тог процеса. Ови проблеми су донекле решени новим приступом у оптимизацији целовитог програма ThinLTO, али уз нешто лошије перфомансе добијеног извршног фајла. У будућности LLVM заједница ће активно наставити на решавању проблема код оба приступа.

Као део рада имплементиран је алат који визуализује промене између међурепрезентација преведених са и без активне оптимизације целовитог програма. Алат омогућава програмеру да види које оптимизације је компајлер успео да изврши тек након укључене оптимизације целовитог програма. Ово је посебно корисно у едукативне сврхе као и за програмере који раде на самим компајлерима, да би проверили како оптимизације раде и евентуално пронађу грешку у имплементацијама неких оптимизација.

Литература

[1] LLVM Compiler Infrastructure – https://llvm.org/docs/index.html

[2] LLVM Language Reference Manual – https://llvm.org/docs/LangRef.html

[3] SSA Form – https://en.wikipedia.org/wiki/Static_single_assignment_form

[4] RISC – https://en.wikipedia.org/wiki/Reduced_instruction_set_computer

[5] Abstract Syntax tree – https://en.wikipedia.org/wiki/Abstract_syntax_tree

[6] Optimizer – https://llvm.org/docs/CommandGuide/opt.html

[7] LLVM Code Generator – https://llvm.org/docs/CodeGenerator.html

[8] Just In Time Compilation – https://en.wikipedia.org/wiki/Just-in-time_compilation

[9] Unity build – https://en.wikipedia.org/wiki/Unity_build

[10] Link Time Optimization – https://llvm.org/docs/LinkTimeOptimization.html

[11] libLTO – https://llvm.org/docs/LinkTimeOptimization.html#liblto

[12] Gold linker – https://llvm.org/docs/GoldPlugin.html

[13] llvm-link – https://llvm.org/docs/CommandGuide/llvm-link.html

[14] Function inlining – https://www.cs.cornell.edu/courses/cs6120/2019fa/blog/llvm-function-inlining/

[15] Indirect branch instructions – https://en.wikipedia.org/wiki/Indirect_branch

[16] Dead code elimination – https://en.wikipedia.org/wiki/Dead_code_elimination

[17] Devirtualization – https://blog.llvm.org/2017/03/devirtualization-in-llvm-and-clang.html

[18] Internal linkage – https://www.learncpp.com/cpp-tutorial/internal-linkage/

[19] Unnamed namespace – https://www.ibm.com/docs/en/i/7.3?topic=only-unnamed-namespaces-c

[20] ThinLTO – https://clang.llvm.org/docs/ThinLTO.html

[21] http://blog.llvm.org/2016/06/thinlto-scalable-and-incremental-lto.html

[22] https://godbolt.org/

[23] https://github.com/filipl41/master

[24] Diff – https://en.wikipedia.org/wiki/Diff

[25] Kompare – https://apps.kde.org/kompare/

[26] Undefined behavior in theory and practice – https://queue.acm.org/detail.cfm?id=3468263

[27] – https://llvm.org/docs/CommandGuide/llvm-dis.html

[28] Clang optimization levels – https://stackoverflow.com/questions/15548023/clang-optimization-levels