

УНИВЕРЗИТЕТ У БЕОГРАДУ
МАТЕМАТИЧКИ ФАКУЛТЕТ



Филип Лазих

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, 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	LLVM компајлерска инфраструктура	3
2.1	LLVM међурепрезентација	4
2.2	LLVM компајлер	6
2.3	Предности LLVM-а	7
3	Оптимизација целовитог програма	8
3.1	Оптимизација током линковања	11
3.2	Оптимизација целовитог програма без подршке линкера	13
3.3	Инлајновање функција	14
3.4	Елиминација мртвог кода	19
3.5	Девиртуализација	23
4	ThinLTO	29
5	Закључак	34
	Литература	35

Глава 1

Увод

Компајлерске оптимизације трансформишу ко́д тако да се програм брже извршава и користи мање меморије док при томе задржава семантичку екивалентност. Обично, оптимизације се извршавају у контексту једног објектног фајла, али пошто у оквиру фајла компајлер нема информације о ко́ду који се налази у другим објектним фајловима, многе оптимизације је немогуће урадити зато што компајлер не може бити сигуран у семантичку екивалентност. Главна тема овог рада биће управе решење овог проблема, односно оптимизација целовитог програма у LLVM компајлерској инфраструктури.

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

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

У глави 4 биће 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.

Као део овог рада имплементиран је алат који визуализује разлике између програма који је преведен са и без оптимизације целовитог програма. Алат је

урађен по угледу на compiler explorer[22] и може се наћи на адреси[23].

Глава 2

LLVM компајлерска инфраструктура

LLVM (Low Level Virtual Machine[1]), упркос свом имену LLVM мало тога има са виртуелним машинама, то је 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 front-end-а за било који програмски језик, као и за развој back-end-а 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 подпројеката су:

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

2.1 LLVM међурепрезентација

LLVM међурепрезентација (LLVM IR[2]) базирана 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. биткод формат (.bc) ¹
3. унутар-меморијски формат

Овим се омогућавају ефикасне компајлерске трансформације и анализе, уз могућност визуалне 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 подразумевано ћемо мислити на овај формат, који се може окарактерисати као асемблерски језик независан од специфичне платформе.

Овде видимо две функције у програмском језику C које сабирају 2 броја.

```
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);
}
```

Сада ћемо представити одговајући код у LLVM међурепрезентацији.

```
define i32 @add1(i32 %a, i32 %b) {
```

¹често се назива и бајткод формат

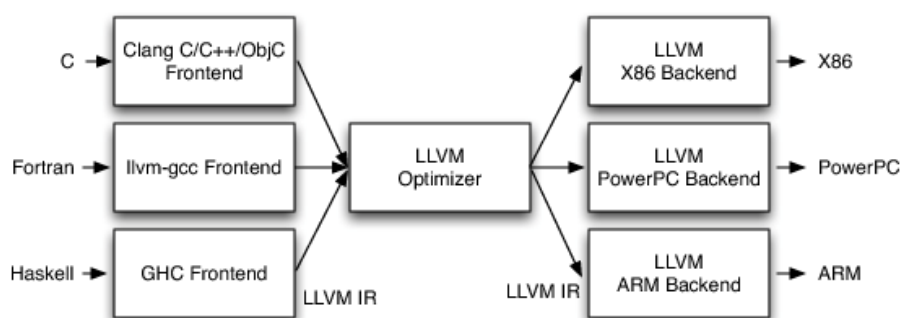
```
entry :  
    %tmp1 = add i32 %a, %b  
    ret i32 %tmp1  
}  
  
define i32 @add2(i32 %a, i32 %b) {  
entry :  
    %tmp1 = icmp eq i32 %a, 0  
    br i1 %tmp1, label %done, label %recurse  
  
recurse :  
    %tmp2 = sub i32 %a, 1  
    %tmp3 = add i32 %b, 1  
    %tmp4 = call i32 @add2(i32 %tmp2, 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, гранање, упоређивање итд. Све ове инструкције су у тро-адресној форми, што значи да могу примити два регистра као улаз и резултат уписати у трећем регистру. Међурепрезентација је строго типизирана (на пример i32 означава тридесет-двобитни целобројни број), док се позив функције означава кључном речи call, а повратна вредност са ret. LLVM не користи фиксан број регистара, већ има бесконачан број променљивих које почињу карактером %. Функције и глобалне променљиве пре свог назива садрже карактер @. Унутар репрезентације постоје и лабеле, тело сваке функције почиње лабелом begin.

2.2 LLVM компајлер

Процес компилације у LLVM инфраструктури започиње у front-end делу који производи међурепрезентацију, која се затим шаље алату за оптимизацију који трансформише кођ кроз велики број оптимизација. Потом се трансформисани кођ преводи у асемблерски кођ на жељеној архитектури, и на крају се асемблерски кођ преводи у машински. Овај процес, наравно поједностављен, можемо видети на слици испод.



Слика 2.1: LLVM процес компилације

Front-end

Front-end је задужен за парсирање, валидацију и проналазак грешака у изворном кођу, затим за превођење парсираног кођа у LLVM међурепрезентацију. Превођење се обично извршава, прво изградњом AST-а[5], а затим и превођењем AST-а у међурепрезентацију. У суштини сваки програмски језик, уколико имплементира front-end који може да изгенерише LLVM међурепрезентацију, може користити алат за оптимизацију или back-end део LLVM-а. Постоји више пројеката који имплементирају LLVM front-end, али најбитнији су:

1. Clang - front-end за C, C++ и Objective C програмске језике
2. DragonEgg - GCC плагин који користи LLVM архитектуру за оптимизацију и генерисање машинског кођа

Алат за оптимизацију

Алат за оптимизацију (eng. optimizer[6]) дизајниран је тако да на улазу прима LLVM међурепрезентацију, изврши оптимизације над међурепрезентацијом и после тога генерише измењену међурепрезентацију, која би требало да се извршава брже. Овај алат је организован у више низова оптимизационих пролаза, тако да је излаз једне оптимизације улаз у другу. Неки од примера оптимизационих пролаза су инлајновање, елиминација мртвог кода, реалокација израза, инваријација петљи итд. Од нивоа оптимизације зависе и оптимизациони пролази који ће бити покренути, на пример, у случају Clang-а, на нивоу -O0 нема оптимизација, док на нивоу -O3 покреће се свих 67 оптимизационих пролаза. Алат за оптимизацију се може покренути командом `opt`.

Back-end

LLVM back-end је фаза у којој се од међурепрезентације, која је улаз за ову фазу, генерише машински код за специфичну архитектуру. Главна компонента back-end-а је генератор кода (eng. LLVM code generator[7]) који користи сличан приступ као алат за оптимизацију, то јест дели генерисање машинског кода на мање пролазе, који имају за циљ генерисање најбољег могућег кода. Неки најбитнији пролази су бирање инструкција, алокација регистара, распоређивање (eng. scheduling). LLVM може генерисати код за велики број архитектура, неки од њих су: x86, ARM, PowerPC, SPARC.

2.3 Предности LLVM-а

LLVM пројекат је бесплатан и његов изворни код је у потпуности доступан, што је навело не само истраживаче са универзитета, већ и велики број компанија да учествују у његовом развоју, тако да данас значајан број људи активно учествује у одржавању и унапређивању овог пројекта. Модуларни дизајн омогућава лако мењање постојећих алата или додавање нових. Захваљујући овом дизајну врло лако је додати нови front-end, back-end или оптимизациони пролаз. Такође, LLVM подржава и:

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

Глава 3

Оптимизација целовитог програма

Обично изворни код 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 код читљивијим, омогућавамо паралелелно компајлирање више фајлова али и избегавамо потребу за компајлирањем целог програма за сваку промену у узворном коду. Овакав приступ има и лошу страну, пошто компајлер преводи фајл по фајл, он нема информације о коду који се налази у другим компилационим јединицама. Зато је немогуће извршити многе оптимизације, због тога што компајлер не може бити сигуран у семантичку еквивалентност. 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]). Сада ћемо на једном малом примеру 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();
```

¹често се назива и јединица транслације

```
    }
}
```

Primer 3.1

Видимо у примеру да функција `do_nothing`, као и што јој име каже, не ради ништа. Уколико овај код преведемо са `-O3` оптимизацијом, без оптимизације целовитог програма, добићемо овај резултат.

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

Видимо да је рачунару било потребно више од једне секунде да изврши програм који не ради ништа. Сада 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
```

Разлика је у времену извршавања је очигледна. Испод имамо приказ LLVM међурепрезентације без и са укљученом оптимизацијом целовитог програма и анализираћемо 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
    }

```

Primer 3.1 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)

```

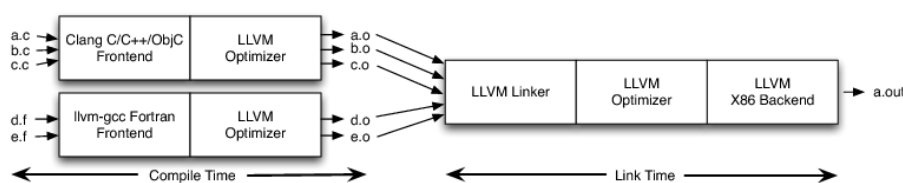
Primer 3.1 sa optimizacijom celovitog programa

У примеру где није укључена оптимизација целовитог програма компајлер не зна како изгледа функција `do_nothing` и он мора милион пута у пељи да је позива. Када је укључена оптимизација целовитог програма компајлер види тело те функције, и види да она не ради ништа, тако да може да оптимизује

не само позивање те функција, односно да је инлајнује, већ може да уклони комплетну петљу, односно елиминисе мртав код. Видимо да је елиминисана и функција `do_nothing` из извршног програма и да је унутар `main` функције остала само повратна вредност, што овај програм само и ради. О инлајновању и елиминацији мртвог кода биће више речи у наставку.

3.1 Оптимизација током линковања

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



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 изврши 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 мора да пронађе све недостајуће симболе у свим објектним фајловима и да упише њихове адресе (такође задатак линкера је да и неким импортованим симбола промени адресу, уколико је компајлер то назначио), то јест да изврши резолуцију и реалокацију симбола. Због ових својстава линкер има круцијалну улогу у оптимизацији целовитог програма, јер има увид у све табеле симбола и алат за оптимизацију може то искористити за оптимизације делова кода који

су му пре били „невидљиви”.

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

1. Читање биткôд фајлова
2. Резолуција симбола
3. Оптимизовање биткод 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 алат под називом libLTO[11]. libLTO је библиотека који је намењена за коришћење од стране линкера. libLTO пружа стабилан интерфејс, тако да је могуће користити LLVM алат за оптимизацију, без потребе за излагањем интерног LLVM кôда. Такође, још једна предност овог алата је то што можемо мењати LLVM LTO кôд независно од линкера, то јесте не морамо за сваку промену кôда мењати и линкер.

Да се вратимо на фазу читања биткôд фајлова, 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 алата 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 изнад, линкер покушава да разреши све симболе помоћу глобалне табеле симбола. Уколико је укључена опција елиминације мртвог кода, која је подразумевано укључена уколико се користи оптимизација

током линковања, линкер чува листу симбола који су коришћени у осталим објектним фајловима, такозвани живи симболи.

Оптимизација биткôд фајлова

У овој фази линкер користи информације из глобалне табеле симбола, и јављује живе симболе алату за оптимизацију `lto_codegen_add_must_preserve_symbol` функцијом. Затим линкер позива алат за отимизацију и генератор кôда над биткôд фајловима функцијом `lto_codegen_compile` чији је резултат објектни фајл који 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.

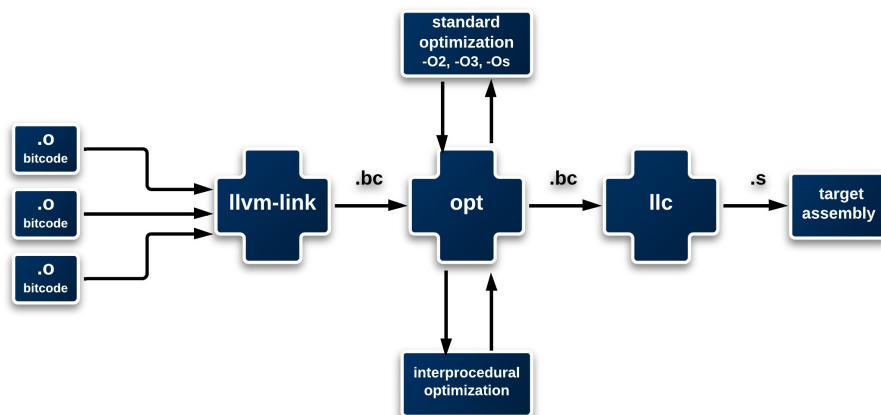
Резолуција симбола након оптимизације

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 сви фајлови су објектни фајлови и линковање се наставља по старом принципу, као да никада нису ни постојали биткôд фајлови.

3.2 Оптимизација целовитог програма без подршке линкера

Приступ отимизације целовитог програма са линкером захтева Gold[12] линкер, који у себи има подршку за `libLTO` библиотеку. 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 перформансе 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.



Слика 3.2: LLVM процес компилације без подршке линкера

3.3 Инлајновање функција

Неписано правило у програмирању је издвајање кода који се понавља у засебне функције. Издвајање кода је корисно зато што на тај начин избацујемо копирање истог кода на више места у програму. На тај начин не само да повећавамо читљивост програма, већ и смањујемо могућност грешака, које су честе при копирању кода. Са друге стране, позиви функција могу бити захтевни што се тиче времена извршања. 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, код функције може бити ван кеша инструкција, што може битно утицати на време извршавања програма. Решење ових проблема је инлајновање функција (eng. function inlining[14]). Инлајновање је замена позива функције у компајлираном коду целокупним телом функције. Поред тога што инлајновање елиминиса трошак позива функције, оно такође омогућава компајлеру да генерише оптималнији код. Пошто је цео код функције уметнут, компајлер може извршити оптимизације у већем блоку, што некада може довести до значајних убрзања. Видели смо да инлајновање функција може бити корисно и намеће се логично питање- када можемо извршити инлајновање? Неке функције можемо одмах елиминисати из списка кандидата за инлајновање, уколико имамо дељену динамичку библиотеку ми немамо информацију о коду функције тако да је не можемо инлајновати. Сличан проблем је са функцијама које се налазе у другим објектним фајловима, али за овај проблем постоји решење, а то је оп-

тимизација целовитог програма. Уколико је оптимизација целовитог програма укључена, компајлер може видети код функција из других објектних фајлова и евенутално их инлајновати. Такође, немогуће је инлајновати функције које садрже инструкције индиректног гранања(eng. indirect branch instructions[15]) јер у том случају, уколико би инлајновали функцију, индиректне гране би нас довеле до неочекиваних инструкција у програму.

Видели смо неке од ситуација у којима је немогуће инлајновати функцију, као и да инлајновање има велики број предности, да ли онда увек инлајновати када је то могуће? Наравно, одговор је не. Поред великог броја предности, инлајновање има и неке мане. Једна од мана је повећање величине извршног фајла, поготово када функције имају велики број инструкција. Видимо да ипак мора да постоји компромис између перформанси програма и величине извршног фајла. Такође, функције са великим бројем инструкција могу да утичу на перформансе, тако што утичу на кеш инструкција, јер велики број инструкција руши локалност референци. На пример уколико инлајнујемо 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 инлајновање оваквих функција. Због свега наведеног за инлајновање користимо хеуристике, преко којих одређујемо да ли неку функцију треба инлајновати 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 функција, где користимо фиксиране границе(eng. threshold) за број иснтрукција, позива итд. и тако одређујемо да ли треба да инлајнујемо 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 перформансе 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 перформансе него статичком анализом. Профајлирањем можемо открити делове кода који се чешће извршавају, и компајлер поклања посебну пажњу оптимизовању и инлајновању тих делова, јер уколико се неке функције скоро никада не позивају, инлајновањем нећемо добити никакве предности у перформансама, само можемо повећати величину извршног фајла. Програмер може у изворном коду сигнализирати компајлеру

да изврши инлајновање атрибутом `always_inline`, на системима Unix, али ни то не гарантује да ће на крају функција заиста бити инлајнована.

Приказаћемо један пример где је могуће извршити инлајновање 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;
}
```

Primer 3.3.1

Сада ћемо видети разлике LLVM међурепрезентације без и са оптимизацијом целовитог програма у примеру 3.3.1.

```
Function Attrs: norecurse uwtable
define i32 @main() local_unnamed_addr #4 !dbg !966 {
    call void @llvm.dbg.value(metadata i32 0,
    metadata !968, metadata !DIEExpression()), !dbg !971
    call void @llvm.dbg.value(metadata i32 0,
    metadata !969, metadata !DIEExpression()), !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 !DIEExpression()), !dbg !987
    %2 = mul nsw i32 %0, %0, !dbg !988
    ret i32 %2, !dbg !989
}
```

Primer 3.3.1 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 !DIEExpression()), !dbg !971
    call void @llvm.dbg.value(metadata i32 0,
        metadata !969, metadata !DIEExpression()), !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
```

Primer 3.3.1 sa optimizacijom celovitog programa

Ако погледамо међурепрезентацију без оптимизације целовитог програма видимо да је она јако слична изворном коду, компајлер не види тело функције `square`, тако да не може неке значајније оптимизације да изврши. Са друге стране, када је укључена оптимизација целовитог програма, компајлер успева да инлајнује функцију. Овим се компајлер не само решава трошкова 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 променљиве `result` може да се израчуна приликом компајлирања, самим тим се решавамо петље и кода око ње. У позиву `std::cout` 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 кода

Елиминација мртвог кода (eng. dead code elimination[16]) је компајлерска оптимизација која елиминише код који не утиче на резултат извршавања програма. Уклањање мртвог кода 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 елиминише локалне променљиве, инлајноване 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. 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.

Сада ћемо видети пример у коме се јасно види предност елиминације мртвог кода са укљученом оптимизацијом целовитог програма.

```
//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 fool();
}
```

Primer 3.4.1

Function Attrs: uwtable

```
declare dereferenceable(272)
%"class.std::basic_ostream"* @_ZSt16__
ostream_insertIcSt11char_traitsIcEERSt13basic_
ostreamIT_T0_ES6_PKS3_1
(%"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_1
(%"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
}

```

Primer 3.4.1 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
}
```

Primer 3.4.1 sa optimizacijom celovitog programa

У примеру без оптимизације целовитог програма, видимо да је компајлер инлајновао 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 инлајновања, није успео да изврши неке веће оптимизације.

Са друге стране, у примеру са оптимизацијом целовитог програма, видимо да је у извршном фајлу остала само `main` функција, која враћа вредност 42. Да бисмо утврдили како се то догодило, проћи ћемо 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, инлајнује је, брише и враћа вредност 42 као повратну вредност функције `main`.

3.5 Девиртуализација

Девиртуализација(eng. devirtualization[17]) је поступак замене виртуелних позива функција директним позивима. Виртуелни позиви функција су неколико пута спорији од директних, што у системима где се перформансе јако вреднују може да буде велики проблем, због тога је пожељно извршити девиртуелизацију када је то могуће. Девиртуализација се најефикасније може извршити уз помоћ оптимизације целовитог програма, али постоје и случајеви у којима је могуће извршити девиртуализацију и без оптимизације целовитог програма. За почетак ћемо се позабавити тим случајевима.

Познат динамички тип објекта

Уколико нам је познат динамички тип објекта при компајлирању, компајлер 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();
}
```

Primer 3.5.1

```
; 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
```

}

Primer 3.5.1 LLVM

Као што видимо у примеру изнад, компајлер успешно девиртуализује и ин-лајнује ове функције јер зна при компајлирању да променљива `b` садржи објекат типа `Base`, док променљива `d` садржи објекат типа `Derived`. Наравно увек када променљива садржи објекат неког типа, а не показивач или референцу, компајлер зна да позив не може бити виртуелан (као у примеру променљиве `d`).

Кључна реч `final`

Искористићемо пример сличан примеру 3.5.1

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

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

int func(Derived *d){
    return d->f();
}
```

Primer 3.5.2

За разлику од примера 3.5.1 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 (eng. `override`).

Унутрашња видљивост

Када кажемо да нека променљива или функција има унутрашњу видљивост (eng. internal linkage[18]) то значи да је она видљива само унутар своје јединице транслације. Што значи да уколико имамо структуру, или класу, која има унутрашњу видљивост, компајлер може девиртуализовати позив функције јер је сигуран да она неће моћи бити преправљена. Видимо да је овај принцип сличан као додавање кључне речи `final`. Један једноставан начин да наша класа добије унутрашњу видљивост јесте смештање исте у безимени простор имена (eng. unnamed namespace[19]). Овај принцип приказан је у примеру 3.5.3.

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

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

Primer 3.5.3

Девиртуализација помоћу оптимизације целовитог програма

Видели смо 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 виртуелних 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. Погледајмо следећи пример, који има само 1 изворни фајл.

```
#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);
}
```

Primer 3.5.4

```
; 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
}

```

Primer 3.5.4 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
}

```

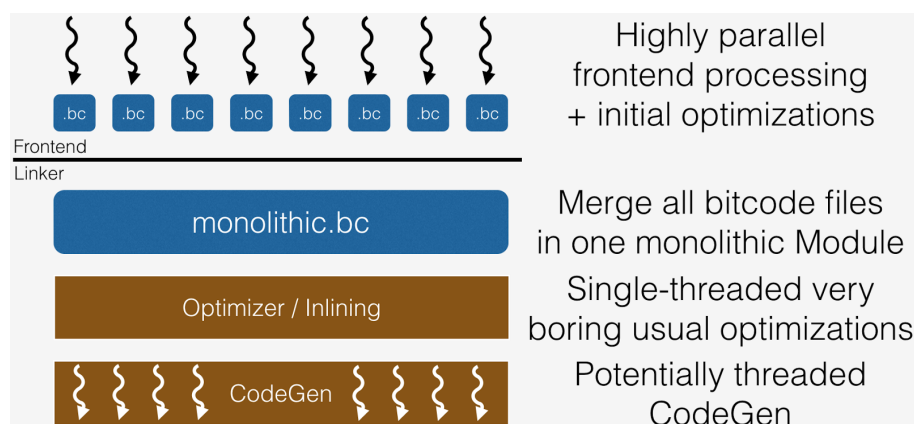
Primer 3.5.4 sa optimizacijom celovitog programa

У примеру без оптимизације целовитог програма видимо инструкције за узимање вредности виртуелног показивача и позивање виртуелне 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 транслације 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 инлајнује позив функције.

Глава 4

ThinLTO

У претходном поглављу видели смо како оптимизација целовитог програма може значајно побољшати перформансе нашег програма. Такође, видели смо како је имплементирана стандардна оптимизација током линковања, укратко линкер добија биткод фајлове, уместо објектних фајлова, затим се ти биткод фајлови спајају у један и над тим фајлом се врше све оптимизације.



Слика 4.1: Стандардни процес оптимизације током линковања

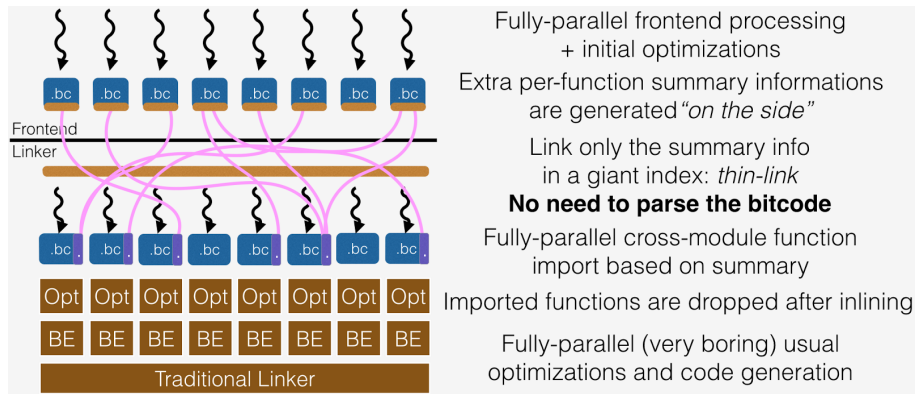
Стандардни приступ има неколико мана. Први проблем је то што губимо предности паралелног компајлирања, која постоји када није активна оптимизација целовитог програма. Као што видимо на слици 4.1 постоји паралелно превођење изворних фајлова у биткод фајлове али због спајања свих фајлова у један велики биткод фајл, ту предност касније губимо зато што све оптимизације над тим фајлом морају да се раде без могућности паралелизације. 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 на време превођења. Још један велики проблем овог приступа је меморијско заузеће. Због тога што сада у меморији морају да се налазе све међурепрезентације, 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 перформансе при превођењу као када није укључена оптимизација целовитог програма, док задржава већину оптимизација и самим тим перформанси извршног фајла као регуларна оптимизација целовитог програма. При ThinLTO оптимизацији уместо учитавања биткода фајлова и спајања у један, већ за сваку јединицу транслације и сваку функцију или глобал у њој, чува кратак резиме за анализу у кораку линковања. Кључна оптимизација коју ThinLTO омогућава је убацивање само оних функција које су потребне конкретном биткоду фајлу и које ће бити инлајноване у том биткоду фајлу. И тај поступак радимо за сваки биткод фајл, што значи да нема спајања, већ се и даље поступак извршава паралелно. ThinLTO процес оптимизације целовитог програма је подељен на 3 фазе:

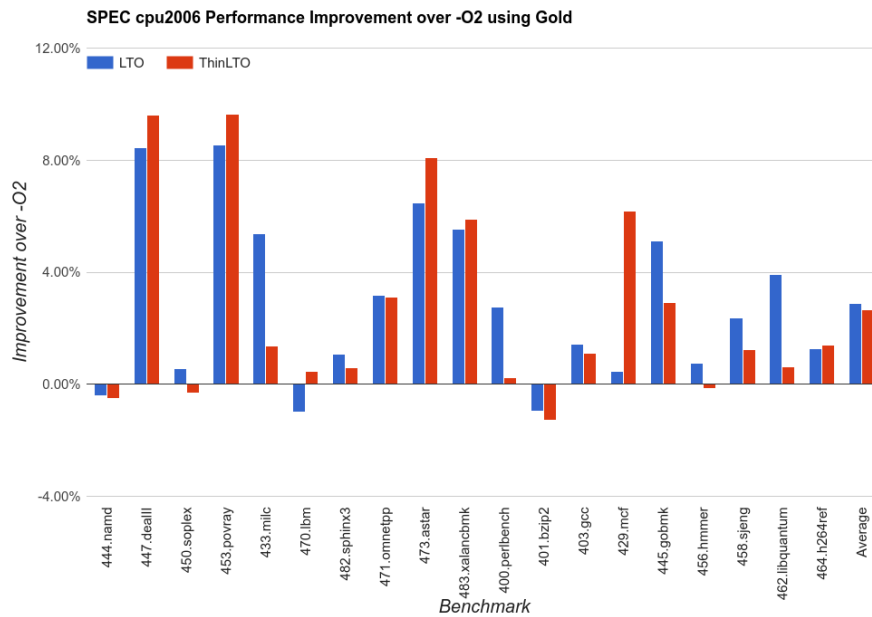
1. Превођење - генеришу се међурепрезентације као и случају стандардног процеса оптимизације целовитог програма, са тим што сада имамо и резиме уз сваку међурепрезентацију
2. Линковање - линкер комбинује резиме из прошлог корака и врши анализу
3. Бекенд - паралелна оптимизација и генерисање кода

Кључни део ThinLTO оптимизације се дешава 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 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 контроле тока(eng. call graph). Ове информације омогућавају креирање комплетног графа током фазе линковања. ThinLTO је једноставно активирати,



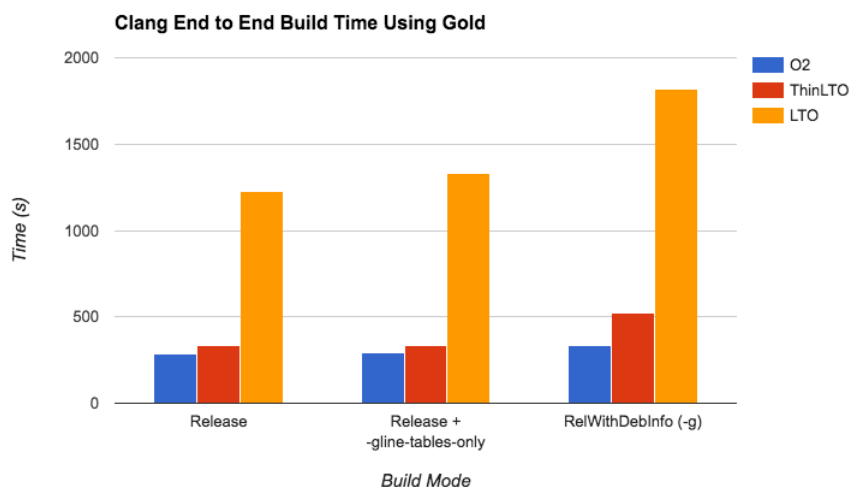
Слика 4.2: ThinLTO процес оптимизације

само је потребно додати `-flto=thin` у командној линији приликом компајлирања. Сада ћемо приказати разлику између перформанси, меморијских захтева као и времена компајлирања између оптимизације током линковања и ThinLTO-а. Сlike су преузете са чланка ThinLTO: Scalable and Incremental LTO[21].



Слика 4.3: Разлика у перформансама између ThinLTO и регуларне оптимизације током линковања

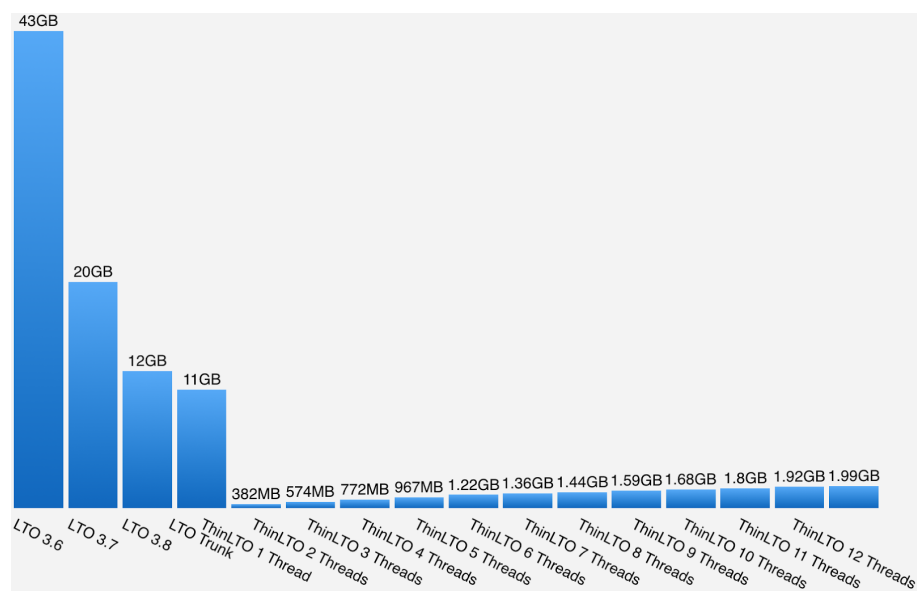
Као што видимо на слици 4.3 у просеку стандардна оптимизација даје за нијансу боље резултате, али постоје ситуације када ThinLTO надмашује стандардну оптимизацију.



Слика 4.4: Разлика у времену превођења програма између ThinLTO и регуларне оптимизације током линковања

На слици 4.4 видимо да је време компајлирања програма са ThinLTO оптимизацијом 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.

Јасно се види на слици 4.5 да је меморијска потрошња регуларне оптимизације током линковања далеко већа од меморијске потрошње ThinLTO-а. Такође, видимо да због огромне потрошње меморије за регуларну оптимизацију током линковања морамо да имам далеко јачи хардвер него за ThinLTO-а, да би уопште превели програм.



Слика 4.5: Разлика у меморијским захтевима између ThinLTO и регуларне оптимизације током линковања

Глава 5

Закључак

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/flipl41/master>