УНИВЕРЗИТЕТ У БЕОГРАДУ
МАТЕМАТИЧКИ ФАКУЛТЕТ

Филип Лазић

ОПТИМИЗАЦИЈА ЦЕЛОВИТОГ ПРОГРАМА НА КОМПАЈЛЕРСКОЈ ИНФРАСТРУКТУРИ LLVM

мастер рад

Београд, 2021.

Ментор:

др Иван Чукић, редован професор
Универзитет у Београду, Математички факултет

Чланови комисије:

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

др Саша Малков, доцент
Универзитет у Београду, Математички факултет

Датум одбране: 15. јануар 2016.

Ами, тати и деди

Садржај

Глава 1

Увод

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)
3. унутар-меморијски формат

Овим се омогућавају ефикасне компајлерске транформације 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 независтан 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 кôд у LLVM међурепрезентацији.

```
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
}
```

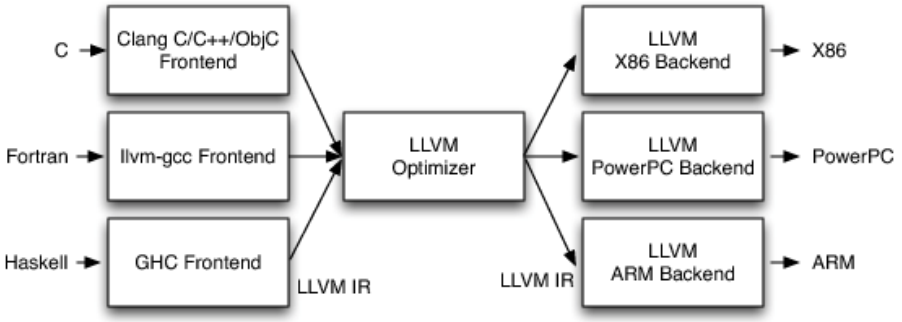
LLVM међурепрезентација је асемблерски формат сличан апстрактном RISC[4] скупу инструкција, са додатним структурама вишег нивоа.

Као што видимо у овом примеру, међурепрезентација подржава линеарне секвенце једноставних инструкција као што су сабирање, одузимање, 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 тридесетдвобитни 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 кôд кроз велики број оптимизација. Потом се трансфор-

мисани 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 испод.



Слика 2.1: LLVM процес компилације

Frond-end

Frond-end је задужен за парсирање, валидацију и проналазак грешака у изворном кôду, затим за превођење парсираног кôда 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 и генерисање машинског кода

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 оптимизациних

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-a, 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 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. 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-a

LLVM пројекат је бесплатан и његов изворни кôд 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. Clang-ов алат за статичку анализу кôда (eng. static code analyzer[9]) - који служи за проналазак могућих грешака у коду
3. оптимизацију током линковања(LTO[10])

Очекује се да LLVM у потпуности замени GCC у блиској будућности.

Глава 3

Оптимизација целовитог програма

Обично изворни кôд програма делимо у више посебних фајлова(eng. source code). Компајлер чита фајл по фајл и за сваки генерише њему одговарајући објектни фајл, то јест сваком фајлу 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 објектним фајловима. Због тога је немогуће извршити многе оптимизације, због тога што компајлер не може бити сигуран у семантичку еквивалентност. Овај проблем се може решити уз помоћ линкера, приступом познатијим као оптимизација током линковања(LTO) или спајањем свих фајлова у један и извршавањем оптимизација на једном великом фајлу.
Сада ћемо на једном малом примену показати због чека оптимизација целовитог програма може бити корисна.

```
//a.h
void do_nothing();
```

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

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

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

```
}
```

Primer 3.1

Видимо у примеру да функција do_nothing, као и што јој име каже, не ради ништа. Уколико овај кôд 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 са 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 оптмизацијом 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 !DIEx
  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 !DIE
  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 !DIE
  %5 = icmp eq i32 %4, 1000000000, !dbg !23
  br i1 %5, label %1, label %2, !dbg !17, !llvm.loop !24
}

; Function Attrs: nounwind readnone speculatable
declare void @llvm.dbg.value(metadata, metadata, metadata) #1

; 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

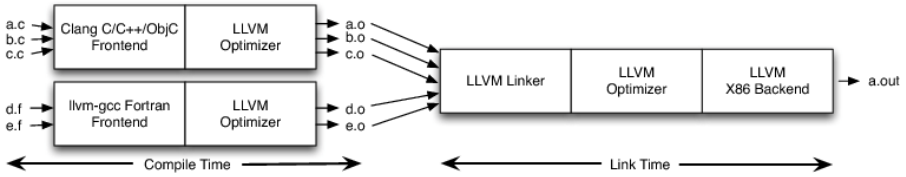
```
; Function Attrs: norecurse nounwind readnone uwtable
define dso_local i32 @main() local_unnamed_addr #0 !dbg !9 {
  call void @llvm.dbg.value(metadata i32 0, metadata !14, metadata !DIEx
  ret i32 0, !dbg !17
}

; Function Attrs: nounwind readnone speculatable
declare void @llvm.dbg.value(metadata, metadata, metadata) #1
```

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 елиминише мртав кôд. 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 кôд биће више речи у наставку.

3.1 Ортимизација током линковања

Захваљујући модуларном дизајну LLVM-а и чињеници да можемо компајлирати део кôда, сачувати резултати и наставити компилацију касније без губитака информација слику 2.1 можемо проширити са линкером и оптимизацијама током овог процеса.



Слика 3.1: LLVM процес компилације са подршком линкера

У наставку ћемо објаснити због чека 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 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 јесте 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 кôда 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

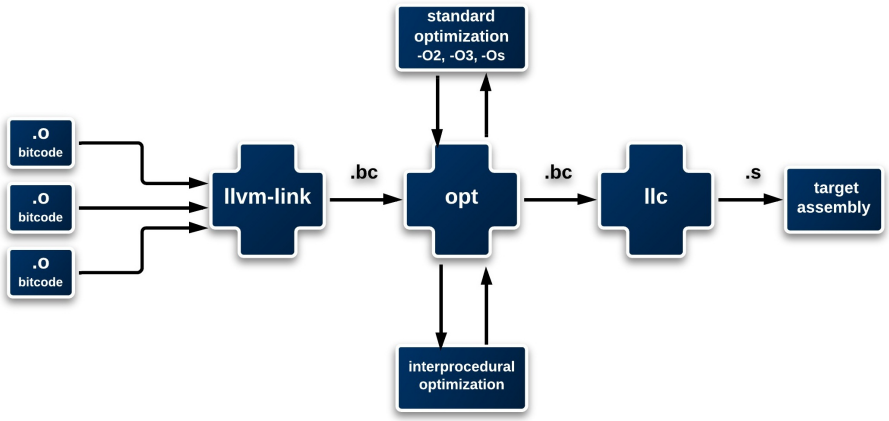
У овој фази линкер користи информације из глобалне табеле симбола, и пријављује живе симболе алату за оптимизацију lto_codegen_add_must_preserve_symbol функцијом. Затим линкер позива алат за отимизацију и генератор кôда над биткôд фајловима функцијом lto_codegen_compile чији је резулат објектни

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

Сада линкер чита оптимизоване објектне фајлове и ажурира табелу симбола уколико има неких промена. На примера 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 Отпимизација целовитог програма без подршке линкера



Слика 3.2: LLVM процес компилације 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].
Са овим приступом добијамо исте перфомансе као са приступом где имамо по-

дршку линкера, са тим што овај приступ неће радити уколико сви фајлови нису биткôд фајлови, односно не ради са објектним фајловима.

3.3 Инлајновање функција

Неписано правило 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 извршања. Када се функција позове долази до креирања новог стек фрејма, 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 инлајновање функција(eng. function inlining[14]), простим речима инлајновање је уметање целовитог кôда функције уместо позива функције. Ово изгледа као добра предност инлајновања, али још већа предност је то што сада компајлер може да генерише оптималнији кôд. Пошто је цео кôд 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, уколико имамо дељену библиотеку ми немамо информацију о кôду 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 оптимизација целовитог програма. Уколико је оптимизација целовитог програма укључена, компајлер може видети кôд 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 перфомански програма и величине извршног фајла. Такође, функције са великим бројем инструкције могу да утичу на перфомансе, тако што утичу на кеш инструкција, јер велики број инструкција руши локалност референци. 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 перфомансе, 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 инлајновању тих делова, јер уколико се неке функције скоро никада не позивају, инлајновањем нећемо добити никакве предности у перфомансама, само можемо повећати величину извршног фајла. Уколико смо сигурни да ће неки део кôда да се често извршава, то можемо назначити компајлеру(у C++ програмском језику) атрибутом [[likely]], без потребе за профајлирањем. Програмер може у извнорном 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.
Сада ћемо приказати 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 !DIE
  call void @llvm.dbg.value(metadata i32 0, metadata !969, metadata !DIE
  br label %3, !dbg !973

; <label>:1:                                      ; preds = %3
  %2 = tail call dereferenceable(272) %"class.std::basic_ostream"* @_ZNS
  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 !DI
  call void @llvm.dbg.value(metadata i32 %4, metadata !969, metadata !DI
  %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 !DI
```

```
  call void @llvm.dbg.value(metadata i32 %7, metadata !968, metadata !DI
  %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(%"cl

; 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 !DI
  %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 !DIE
  call void @llvm.dbg.value(metadata i32 0, metadata !969, metadata !DIE
  %1 = tail call dereferenceable(272) %"class.std::basic_ostream"* @_ZNS
  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(%"cl
```

Primer 3.3.1 sa optimizacijom celovitog programa

Ако погледамо међурепрезентацију 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, 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 кôда 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 кôда

Елиминација мртвог кôда(eng. dead code elimination[16]) је компајлерска оптимизација која елиминише кôд који не утиче на резултат извршавања програма. Уклањања 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, 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.

3.5 Девиртуализација

Девиртуализација(eng. devirtualization[17]) је поступак замене виртуелних позива функција директним позивима. Виртуелни позиви функција су неко-

лико пута спорији од директних, што у системима где се перфомансе јако вреднују може да буде велики проблем, због тога је понекад неопходно извршити девиртуелизацију. 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.

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

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

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

Primer 3.5.1

//todo llvm primer

Као што видимо у примеру изнад, компајлер успешно девиртуализује 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 ће моћи да девиртуализује тај позив(као у примеру променљиве d)

Кључна реч final

Искористићемо пример сличан примеру 3.5.1

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

struct Derived : public 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 би 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 фунцкиција(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, комапјлер 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 : public Base{
    int f() override {return 2;}
};
}
```

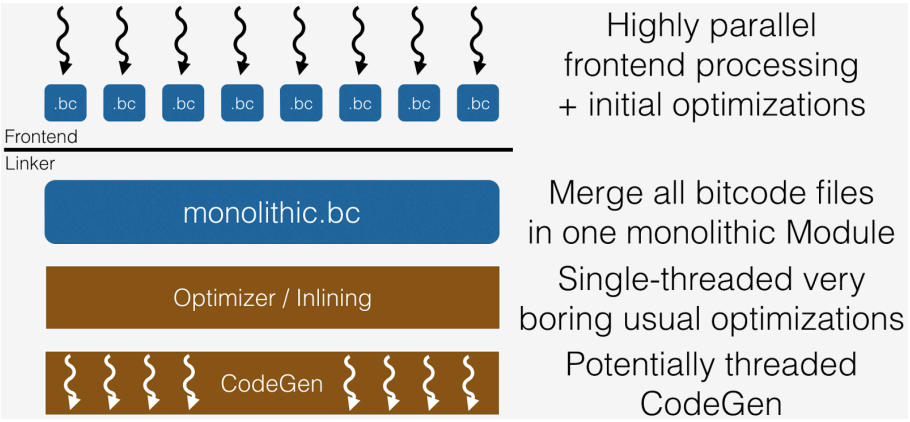
Primer 3.5.3

Девиртуализација помоћу оптимизације целовитог програма

Глава 4

ThinLTO

У претходном програму видели смо како оптимизација целовитог програма може значајно побољшати перфомансе нашег програма. Такође, видели смо како је имплементирана стандардна оптимизација током линковања, укратко линкер добија биткôд фајлове, уместо објектних фајлова, затим се ти биткôд фајлове 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 биткôд фајлове али због спајања свих фајлова у један велики биткôд фајл, ту предност касније губимо 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 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].

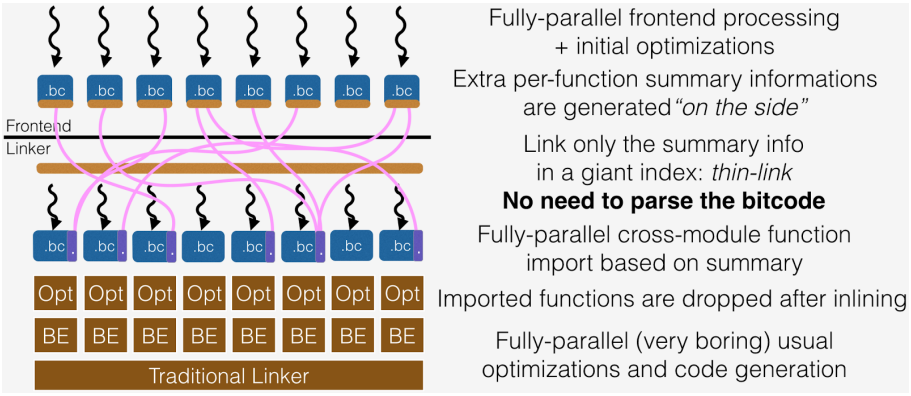
ThinLTO је нови приступ који омогућава сличне перфомансе при превођењу као када није укључена оптимизација целовитог програма, док задржава већину оптимизација и самим тим перфомански извршног фајла као регуларна оптимизација целовитог програма. При ThinLTO оптимизацији уместо учитавања биткôд фајлова 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 биткôд фајлу и које ће бити инлајноване у том биткôд фајлу. И тај поступак радимо за сваки биткôд 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 међурепрезензације као и 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. Бекенд - Паралелна оптимизација и генерисање кôда

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, само је потребно додати -flto=thin у командној линији приликом компајлирања.



Слика 4.2: ThinLTO процес оптимизације

//todo bencmarci

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] Clang Static Code Analyzer – https://clang-analyzer.llvm.org/

[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