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
}
```

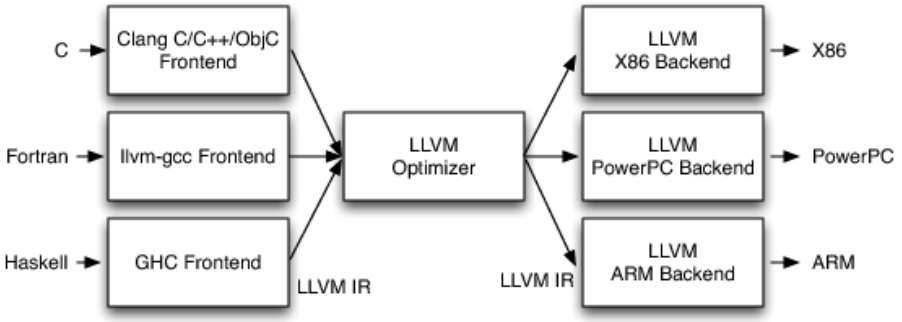
LLVM међурепрезентација је асемблерски формат сличан апстрактном RISC[4] скупу инструкција, са додатним структурама вишег нивоа.

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();
    }
```

```
}
```

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. Једноставно без активне оптимизације целовитог програма, алат за оптимизацију није могао бити сигуран шта функција do_nothing заправо ради, и програм је позивао исту милион пута у петљи, док је са активном оптимизацијом могао да види кôд функције и да оптимизује петљу.

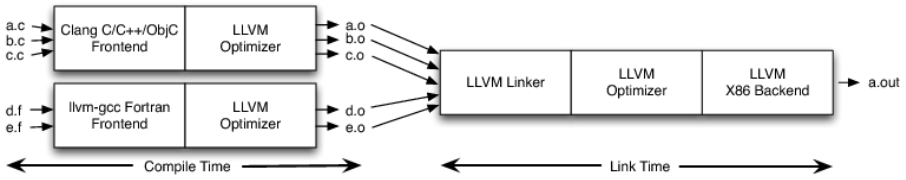
//TODO dodaj LLVM IR

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.

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



Слика 3.1: LLVM процес компилације са подршком линкера

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

омугућава 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

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

//TODO nacrtaj grafik

Глава 4

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