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
}
```

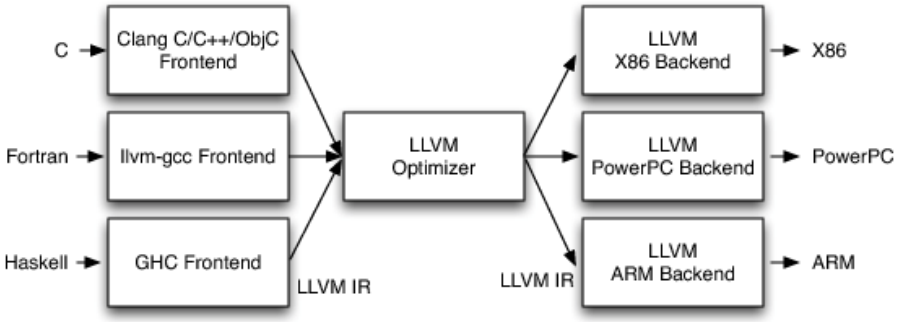
LLVM међурепрезентација је асемблерски формат сличан апстрактном RISC[4] скупу инструкција, са додатним структурама вишег нивоа.

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

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