UNIVERZITET U BEOGRADU
MATEMATIČKI FAKULTET

Vladimir Vuksanović

UNAPREĐENJE INFRASTRUKTURE LLVM ČUVANJEM ORIGINALNE LOKACIJE PRI DEBAGOVANJU IZDVOJENOG KODA

master rad

Beograd, 2023.

Mentor:

dr Milena Vujošević Janičić, vanredni profesor
Univerzitet u Beogradu, Matematički fakultet

Članovi komisije:

dr Filip Marić, redovan profesor
Univerzitet u Beogradu, Matematički fakultet

dr Mirko Spasić, docent
Univerzitet u Beogradu, Matematički fakultet

Datum odbrane: _____

Majci i sestri

Naslov master rada: Unapređenje infrastrukture LLVM čuvanjem originalne lokacije pri debagovanju izdvojenog koda

Rezime: Prilikom kompilacije programi se optimizuju radi postizanja boljih performansi u vidu vremena izvršavanja ili memorijskog zauzeća. Dok su optimizacije pogodne za normalno izvršavanje, problem može da nastane prilikom pokretanja optimizovanog programa pomoću debagera. Debager se oslanja na informacije za debagovanje koje kompilator generiše pored mašinskog koda, ali te informacije je teško održati što se više krajnji kôd razlikuje od početnog. Ovaj problem se susreće prilikom optimizacije izdvajanja koda u kompilatoru LLVM. Izdvajanje koda je memorijska optimizacija koja pronalazi segmente koda koji se ponavljaju i premešta ih u jednu funkciju pritom zamenjujući sve segmente pozivom nove funkcije. Posledica manje količine koda je što nema dovoljno prostora za zapis lokacija u izvornom kodu koje čine bitan deo informacija za debagovanje. Lokacije se zapisuju unutar tabele linija i može da postoji po jedna za svaku instrukciju programa. Smanjenjem broja instrukcija prilikom izdvajanja koda, izgubi se mesto za skladištenje odgovarajućih lokacija. Ovaj rad ima cilj da popravi to ponašanje i sačuva što je više moguće originalnih lokacija izdvojenog koda i da ih upotrebi za poboljšanje procesa debagovanja. Poboljšanja debagera obuhvataju podršku za ispisivanje lokacija izdvojenih instrukcija, koračanje po njima i postavljanje tačaka prekida na njih. Rešenje je implementirano kao nadogradnja alata iz infrastrukture LLVM, ali se ideja može primeniti na druge kompilatore i debagere.

Ključne reči: izdvajanje koda, kompilator, debager, informacije za debagovanje, projekat LLVM, kompilator LLVM, debager LLDB

Sadržaj

Glava 1

Uvod

Viši programski jezici (eng. *high-level programming languages*) su drastično promenili način na koji se pravi softver. Programeri više ne moraju da budu detaljno upoznati sa detaljima hardvera već pišu programe nezavisno od arhitekture. Apstrakcija koju nude viši jezici omogućava programerima da razmišljaju na prirodniji način, koristeći napredne konstrukte tog jezika. Kako računari ne mogu direktno da izvrše kôd na višem jeziku, potrebno je prevesti ga u mašinski kôd za ciljnu arhitekturu.

Postoje dva pristupa prevođenju viših jezika: *interpretiranje* i *kompiliranje*. Interpretiranje podrazumeva postojanje odvojenog programa, *interpretatora*, koji kao ulaz prima program napisan na višem programskom jeziku i ulazne podatke i u toku izvršavanja prevodi prosleđeni program naredbu po naredbu. Alternativni pristup je prevesti izvorni kôd u izvršni fajl pre njegovog pokretanja. Tako dobijen program može da se izvršava samostalno bez dodatnih alata. Za taj posao zadužen je *kompilator*. Pored glavnog zadatka da prevede program napisan na višem jeziku u sematički ekvivalentan program za neku mašinu, drugi bitan zadatak kompilatora je da usput transformiše taj program tako da ima bolje performanse. Neke od prednosti ovog pristupa su brže izvršavanje, manji trošak memorije i prepoznavanje jezičkih grešaka pre izvršavanja.

Napisan i preveden program može da ima propuste u svojoj implementaciji. Zbog toga, bitan deo razvoja softvera je rešavanje grešaka u napisanom programu. Jedan od najtežih delova tog procesa je pronaći deo koda koji izaziva grešku. Taj korak je znatno olakšan korišćenjem specijalizovanih alata, *debagera*. Oni uz pomoć podrške operativnog sistema preuzimaju kontrolu nad programom u kome se traži greška što im omogućava da upravljaju njegovim izvršavanjem i čitaju njegovu memoriju.

To omogućava interaktivno izvršavanje programa, pregled vrednosti promenljivih i izraza kao i brojne naprednije mogućnosti.

Za ugodan rad debagera potrebna je pomoć od strane kompilatora da sačuva podatke o izvornom kodu prilikom kompilacije nekog programa. Ti podaci se nazivaju informacije za debagovanje. Značajan deo tih informacija čine lokacije za debagovanje koje povezuju delove izvršnog fajla sa odgovarajućum delovima izvornog koda na osnovu kojih su generisani.

Prevođenje programa sa optimizacijama predstavlja problem održavanju informacija za debagovanje. Što se program više razlikuje od izvornog koda postaje teže, ako ne i nemoguće, ažurirati informacije za debagovanje tako da i dalje opisuju izvorni kôd.

Prethodni problem utiče na implementaciju optimizacije izdvajanja koda u kompilatoru LLVM [16]. Izdvajanje koda je kompilatorska optimizacija čiji cilj je smanjenje količine memorije potrebne za program tako što više puta ponovljen kôd pomera u zasebnu funkciju i briše duplikate [18]. Smanjenje ukupnog broja instrukcija u programu onemogućava skladištenje lokacija na podrazumevan način zato što bi jednoj instrukciji odgovaralo više različitih lokacija, a to trenutno nije podržano. Posledice nedostatka lokacija prilikom debagovanja je što neke osnovne operacije mogu imati neočekivane rezultate. Između ostalog, koračanje po programu potpuno preskače instrukcije bez lokacije i nije moguće postaviti tačku prekida na njih.

U ovoj tezi biće predloženo rešenje za čuvanje originalnih lokacija u izvornom kodu za instrukcije izmeštene optimizacijom izdvajanja koda. Rešenje uključuje skladištenje lokacija na drugo mesto u programu gde će u vreme izvršavanja biti moguće odrediti korektnu lokaciju. Na osnovu tih lokacija biće poboljšan rad debagera na izdvojenom kodu. Rešenje je implementirano kao nadogradnja kompilatora LLVM i debagera LLDB [15].

Ostatak rada je organizovan na sledeći način. Poglavlje 2 opisuje kompilatore i njihovu strukturu, sa posebnom pažnjom posvećenom optimizacijama izvornog koda koje kompilatori izvršavaju. Takođe su detaljno opisane faze kompilacije u okviru kompilatora LLVM za koji će biti implementirano rešenje. U poglavlju 3 se govori o rešavanju grešaka u softveru pomoću debagera, informacijama za debagovanje koje su neophodne za to i formatu DWARF za predstavljanje tih informacija. Na kraju poglavlja opisan je debager LLDB kao i glavni delovi njegove implementacije. Poglavlje 4 uvodi optimizaciju izdvajanja koda, opisuje njenu implementaciju u kompilatoru LLVM i detaljnije predstavlja problem koji je tema ovog rada. Imple-

mentacija rešenja podeljena je na dva dela u okviru poglavlja 5. Prvi deo se bavi implementacijom na strani kompilatora gde je cilj očuvati informacije za debagovanje, a drugi deo implementacijom u debageru gde se sačuvane informacije koriste za poboljšanje podrške za izdvojen kôd. Rezultati implementiranih poboljšanja, kao i testovi ispravnosti predstavljeni su u poglavlju 6. Na kraju, u poglavlju 7 iznet je zaključak rada.

Glava 2

Kompilatori

Računari izvršavaju programe koji su u obliku mašinskog koda. Kako je nepogodno za ljude da čitaju i pišu takav kôd bilo je potrebno da se on na neki način apstrahuje. Prva apstrakcija nastala je u vidu asemblerskog koda. On je uveo mnemonike za mašinske instrukcije što je znatno poboljšalo čitljivost koda. Kako računari još uvek izvršavaju samo mašinski kôd, nastali su posebni alati, asembleri, koji vrše jednostavne zamene mnemonika i operanada sa odgovarajućim mašinskim kodom.

Programiranje direktno asemblerskim jezikom je veoma izazovan posao koji zahteva precizno poznavanje računara za koji se prevodi program. Potreba za pisanjem kompleksnijeg softvera dovela je do nastanka viših programskih jezika. Ti jezici omogućavaju programerima da koriste složenije koncepte i već poznate algoritme i strukture podataka bez potrebe da sami implementiraju sve iz početka. Prevođenje viših jezika je znatno izazovnije od jednostavne tekstualne zamene koju rade asembleri. Alati za prevođenje koda višeg programskog jezika nazivaju se kompilatori.

Zadaci kompilatora su višestruki. Glavni zadatak je da prevede izvorni kôd u sematički ekvivalentan mašinski kôd. Pored toga, proces kompilacije treba da se brzo izvršava kako bi veliki programi mogli da se prevedu u razumnoj količini vremena. Uz to, neophodno svojstvo je da i konačni program bude što efikasniji. Na kraju, kompilator treba da prilikom prevođenja, ukoliko mu je zatraženo, sačuva informacije o izvornom kodu i poveže delove konačnog programa sa odgovarajućim delovima tog koda.

2.1 Struktura modernih kompilatora

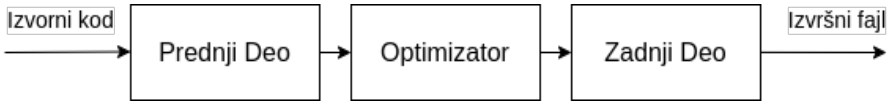
Kompilatori su veoma kompleksni softverski sistemi. Kako postoje već dugo vremena i čine ključne alate za razvoj softvera, oni su tema brojnih istraživanja. Kao rezulat tih istraživanja dobijeno je dosta informacija o dobrim načinima za njihovo struktuiranje i implementaciju. Moderni kompilatori se obično sastoje iz tri dela [7]:

Prednji deo — Analizira i prevodi izvorni kôd na neku vrstu interne reprezentacije. Pritom vrši leksičku, sintaksičku i semantičku analizu nad kodom. Ukoliko korisnički kôd nije ispravan, kompilacija se završava ovde.

Optimizator — Optimizuje internu reprezentaciju kako bi se postigao neki cilj. Može da radi u jednom ili više prolaza.

Zadnji deo — Prevodi validan kôd u obliku interne reprezentacije do programa koji može da se izvrši na ciljnoj mašini.

Samo prednji i zadnji deo su potrebni za generisanje mašinskog koda. Optimizator je opcioni deo nastao iz potrebe da dobijeni kôd ima dobre preformanse. Svaki dobar kompilator ga ipak sadrži, i on čini jedan od najvećih delova kompilatora. Odnos ovih delova je prikazan na slici 2.1. U svakom trenutku program je enkodiran u neku vrstu interne reprezentacije. Ta reprezentacija postaje sve bliža mašinskom kodu u svakom koraku. Ovakva struktura omogućava lako dodavanje podrške za druge programske jezike kao i za nove arhitekture.



Slika 2.1: Grafički prikaz strukture modernog kompilatora

2.2 Kompilatorske optimizacije

Kompilatori osim prevođenja izvornog koda do mašinskih instrukcija takođe mogu da ga modifikuju sa ciljem boljih performansi sve dok se optimizovani kôd ponaša isto kao originalni. Performanse mogu da se odnose na vreme izvršavanja programa, memorijski prostor potreban prilikom izvršavanja ili memorijski prostor potreban za skladištenje izvršnog fajla. U zavisnosti od faze kompilacije u kojoj se rade, optimizacije mogu biti nezavisne ili zavisne od ciljne arhitekture. Nezavisne optimizacije

rade za sve ciljne arhitekture i oslanjaju se na smanjivanje ukupnog broja operacija koje treba izvršiti. Zavisne optimizacije iskorišćavaju detalje arhitekture kako bi podigli performanse. To se ogleda u korišćenju instrukcija koje rade više operacija u isto vreme, imaju kraći zapis ili rade brže od alternativa. Čest primer na arhitekturi `x86_64` je zamena instrukcije `mov eax, 0` sa instrukcijom `xor eax, eax`. Prva instrukcija direktno postavlja vrednost registra `eax` na 0. Druga instrukcija radi isto to, ali konstantu 0 dobija kao rezultat binarne ekskluzivne disjunkcije čija su oba operanda vrednost registra `eax`. Obe operacije imaju isti ishod, ali druga zauzima manje memorije i u nekim slučajevima se brže izvršava.

Optimizacije se u odnosu na opseg koda koji obrađuju mogu podeliti na:

Lokalne optimizacije — Rade na nivou jednog osnovnog bloka (eng. *basic block*). Ovo su najjednostavnije optimizacije zato što ne obrađuju instrukcije skoka.

Globalne (intraproceduralne) optimizacije — Operišu na nivou pojedinačnih funkcija. Većina lokalnih optimizacija se može modifikovati da radi globalno.

Interproceduralne optimizacije — Rade na nivou čitavog programa. One istovremeno obrađuju više funkcija, potencijalno iz različitih kompilacionih jedinica.

Primeri za svaku od prethodnih grupa redom su:

Sažimanje konstanti (eng. *constant folding*) — U programskom kodu se često javljaju konstantni matematički izrazi bez promenljivih. Oni mogu da budu direkno napisani u izvornom kodu, dobijeni pretprocesiranjem ili drugim optimizacijama itd. Kako izraz ne sadrži promenljive, nema razloga da se on izračunava za vreme izvršavanja svaki put, već se to radi u vreme kompilacije.

Globalna numeracija vrednosti (eng. *global value numbering*) — Neki izrazi ili delovi izraza se mogu pojaviti na više mesta u kratkom rasponu u okviru funkcije. Ukoliko su izračunati jednom, nije potrebno računati ih ponovo ispočetka već je moguće obezbediti da prethodni rezultat ostane dostupan u memoriji dok nije ponovo potreban.

Umetanje koda (eng. *inlining*) — Pozivi funkcija su skupa operacija zato što menjaju kontrolu toka izvršavanja i zahtevaju dodatne instrukcije za čuvanje i učitavanje vrednosti registara. Zbog toga je poželjno zameniti pozive kratkih

funkcija sa njihovom definicijom. Time se izbegava cena poziva o trošku veće količine memorije za čuvanje programa.

Postoji nekoliko nivoa optimizacije koji se mogu podesiti u prilikom poziva kompilatora. Izbor nivoa određuje koje optimizacije treba da se izvrše. Način odabira i nivoi optimizacije su drugačiji na različitim kompilatorima. Na kompilatorima `gcc` i `clang` korisnik može da izabere nivo tako što prosledi zastavicu `-O` iza koje se javlja neki od karaktera 0, 1, 2, 3 ili s[1]. `O0` je podrazumevani nivo ukoliko nijedan drugi nije naveden i označava da se kôd uopšte ne optimizuje. Ovaj režim se koristi za pravljenje debag verzija programa. Više reči o tome će biti u poglavlju 3.1. `O1` i `O2` redom uključuju sve veći broj optimizacija podrazumevano da one drastično ne povećavaju memorijsku složenost programa. `O3` ja najveći nivo optimizacije vremena izvršavanja. On podrazumeva sve što se koristi u `O2`, ali uključuje i optimizacije koje mogu da znatno povećaju memorijsko zauzeće programa. Isporučene verzije programa se često kompiliraju na ovaj način. Sa druge strane `Os` nivo takođe podrazumeva sve optimizacije iz `O2` ali dodatno i optimizacije koje smanjuju memorijsko zauzeće na uštrb vremena izvršavanja. Ovaj nivo se koristi za uređaje sa ugrađenim računarom (eng. *embedded devices*) koji imaju malu količinu memorije.

2.3 Kompilator LLVM

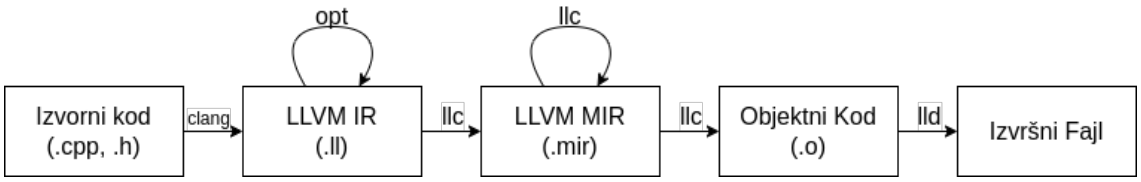
Projekat LLVM započet je 2000. godine na Univerzitetu Ilinois od strane Krisa Latnera. Ubrzo se projektu pridružio i njegov mentor Vikram Adve. Cilj projekta je bio proučavanje tehnika kompiliranja u SSA obliku (eng. *Static Single Assignment*) koje podržavaju statičku i dinamičku kompilaciju proizvoljnih programskih jezika. Inicijalno, naziv LLVM je bio akronim za „virtuelna mašina niskog nivoa" (eng. *Low Level Virtual Machine*). Od tada akronim se više ne koristi, ali je ime ostalo nepromenjeno. Danas, projekat pokriva veliki broj biblioteka i alata koji se koriste za komercijalne i projekte otvorenog koda [16]. Svaki deo projekta je dizajniran kao biblioteka tako da se može ponovo upotrebiti za implementiranje drugih alata. Celokupan izvorni kôd je javno dostupan na servisu GitHub. Prednost pristupu otvorenog koda je da svako ko želi da poboljša kôd ili ispravi neku grešku može to da učini. Velika zajednica se formirala oko projekta što je znatno doprinelo njegovoj popularnosti. Mnoge firme koriste svoje verzije kompilatora LLVM bilo za podršku neke arhitekture ili kao osnovu za novi programski jezik.

[1] Ovo nije potpuna lista. Postoje nivoi koji nisu navedeni.

Glavnu komponentu projekta LLVM čini kolekcija biblioteka u okviru istoimenog potprojekta. One implementiraju optimizator i zadnji deo kompilatora i koriste se od strane većine ostalih alata. Drugi bitan potprojekat je clang. On implementira prednji deo kompilatora za jezike C, C++ i Objective-C. LLVM implementira i svoj debager koji se zove LLDB. Više reči o njemu će biti u poglavlju 3.3. Postoji još veliki broj biblioteka i pomoćnih alata koji nisu pomenuti zato što nisu predmet ovog rada.

Kompilator LLVM je relativno nov u odnosu na druge popularne kompilatore i prati moderniji dizajn. Za razliku od kompilatora GCC [11], koji je napisan u jeziku C i ima monolitnu strukturu, LLVM uživa u pogodnostima koje nudi jezik C++ pritom koristeći modularnu arhitekturu. Dok je najveći deo projekta napisan u programskom jeziku C++, postoji interfejs za povezivanje sa jezicima C i Python.

Implementacija kompilatora LLVM prati opštu strukturu kompilatora prikazanu u poglavlju 2.1 [4]. Korake kompilacije izvršavaju različiti alati. Odnos različitih reprezentacija koje program ima u toku kompilacije i alata koji konvertuju iz jedne reprezantacije u drugu je prikazan na slici 2.2. U nastavku će biti opisani redom prednji, srednji i zadnji deo kompilatora LLVM sa fokusom na poslednja dva kako su oni bitniji za ovaj rad.



Slika 2.2: Grafički prikaz reprezentacija koda prilikom kompilacije kompilatorom LLVM i alata koji se koriste za konverziju između njih

Prednji deo

Projekat LLVM sadrži više različitih prednjih delova kompilatora. Druge organizacije i firme takođe proizvode i održavaju prednje delove kompilatora LLVM za svoje jezike. Iz tog razloga, LLVM podržava veliki broj programskih jezika koji uključuju C, C++, Objectve-C, Fortran, Haskell, Swift i druge. Prednji deo za programske jezike C i C++ koji su fokus ovog rada naziva se Clang.

Postoji nekoliko značenja termina Clang. Može se smatrati da je to prednji deo kompilatora, kao u skladu sa prethodnim korišćenjem izraza, zatim se može podrazumevati da je to alat koji upravlja celim procesom kompilacije ili može da bude biblioteka koja implementira funkcionalnost prednjeg dela kompilatora. U okviru rada se podrazumeva prva interpretacija.

Funkcija ovog dela kompilatora je da izvorni kôd, u ovom slučaju napisan u jezicima C ili C++ prevede u LLVM međureprezentaciju [23]. Usput se izvršavaju sve provere ispravnosti izvornog koda i prikazuju se greške i upozorenja korisniku. Program se detaljno analizira iz nekoliko koraka i pritom dolazi do transformacija izvornog koda.

Prvo se prepoznaju osnovne jedinice gramatike programskog jezika, odnosno lekseme. Lekseme se pretvaraju u različite tipove tokena u zavisnosti od toga šta označavaju. Na primer za ključne reči programskog jezika postoje različite vrste tokena, dok svi identifikatori imaju isti tip tokena. Tokeni osim tipa imaju i informacije o opsegu koji zauzimaju u izvornom kodu pomoću kog se može pristupiti njihovom tekstu.

Nad nizom tokena se radi sintaksna analiza ili parsiranje u skladu sa pravilima gramatike jezika. Ta pravila se zadaju korišćenjem kontekstno-slobodne gramatike (eng. *context-free grammar*). Primenom pravila gradi se stablo parsiranja (eng. *parse tree*). Svaki unutrašnji čvor tog stabla predstavlja jednu operaciju, a njegova deca predstavljaju argumente te operacije. Parsiranje utvrđuje da li program ima ispravnu strukturu, ali ne proverava da li taj program ima smisao. To je zadatak semantičke analize.

Program može da ima dobru strukturu, ali da nema validno značenje. Prolaskom kroz stablo parsiranja se prikupljaju informacije o funkcijama, promenljivima i drugim objektima i smeštaju se u tabelu simbola. Korišćenjem te tabele proverava se ispravnost stabla parsiranja. Primer provere koja se vrši na ovom nivou je da li su promenljive deklarisane pre njihove upotrebe. Rezultat svih ovih analiza je program u obliku apstraktnog sintaksnog stabla (eng. *Abstract Syntax Tree, AST*).

U biblioteci Clang parsiranje i semantička analiza su usko povezani, a lekser se poziva po potrebi od strane parsera [32]. Bitan korak koji se izvršava pre leksiranja je pretprocesiranje u okviru kog se obrađuju direktive poput `#include` i `#define`. Prilikom primene pravila gramatike parser zahteva tokene od leksera i delegira posao kreiranja sintaksnog stabla semantičkom analizatoru. Ukoliko se pronađe greška u bilo kojoj analizi, program se ne moze kompilirati i poruka o razlogu se vraća ko-

risniku. U suprotnom, obilaskom apstraktnog sintaksnog stabla generiše se LLVM međukod. Komanda za prevođenje programa do LLVM međukoda je sledeća:

```
clang -S -emit-llvm -o output.ll input.c
```

Ukoliko se program uspešno prevede do međukoda, onda je izvorni kôd validan za jezik u kome je napisan. Još uvek je moguće da dođe do greške prilikom povezivanja zbog nepostojećih definicija nekih funkcija, klasa ili simbola, ali će kompilacija do asemblerskog ili objektnog koda biti uspešna. Naravno, činjenica da se program kompilira ne znači da on radi ono što je programer zamislio niti da će se program uvek uspešno završiti.

Optimizator

Prednji deo prevodi izvorni kôd u LLVM mašinski nezavisan međukod. U terminologiji se koristi skraćenica IR (eng. *Intermediate Representation*). Izbor reprezentacije međukoda je veoma bitan kada kompilator treba da podrži i više programskih jezika i više ciljnih arhitektura. On treba da je na dovoljno visokom nivou da ne zavisi od arhitekture i u isto vreme na dovoljno niskom nivou da ne zavisi od programskog jezika [30].

LLVM međukod je u SSA obliku [29]. To znači da je svakoj promenljivoj vrednost dodeljena samo jednom i da svakoj upotrebi promenljive prethodi njena definicija. Ovaj oblik je izabran zato što je pogodan za izvršavanje optimizacija.

U listingu 2 je dat primer LLVM međukoda za program prikazan u listingu 1. Kompletan pregled jezika LLVM IR dostupan je na internetu [13].

```c
#include <stdio.h>

int main() {
  int x;
  scanf("%d", &x);
  if (x % 2 == 0)
    printf("Hello, World!");
  return 0;
}
```

Listing 1: C program napravljen za istraživanje strukture LLVM međukoda

Svaki fajl sa LLVM međukodom čuva informacije o jednom modulu. Na početku fajla se nalazi ime fajla u kojem se nalazi izvorni kôd, informacije o načinu zapisa

```
; ModuleID = 'helloworld.c'
source_filename = "helloworld.c"
target datalayout =
↪  "e-m:e-p270:32:32-p271:32:32-p272:64:64-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-unknown-linux-gnu"

@.str = private unnamed_addr constant [3 x i8] c"%d\00", align 1
@.str.1 = private unnamed_addr constant [14 x i8] c"Hello, World!\00", align 1

; Function Attrs: noinline nounwind optnone uwtable
define dso_local i32 @main() #0 {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  store i32 0, ptr %1, align 4
  %3 = call i32 (ptr, ...) @__isoc99_scanf(ptr noundef @.str, ptr noundef %2)
  %4 = load i32, ptr %2, align 4
  %5 = srem i32 %4, 2
  %6 = icmp eq i32 %5, 0
  br i1 %6, label %7, label %9

7:                                                ; preds = %0
  %8 = call i32 (ptr, ...) @printf(ptr noundef @.str.1)
  br label %9

9:                                                ; preds = %7, %0
  ret i32 0
}

declare dso_local i32 @__isoc99_scanf(ptr noundef, ...) #1

declare dso_local i32 @printf(ptr noundef, ...) #1

...

!llvm.module.flags = !{!0, !1, !2, !3, !4}
!llvm.ident = !{!5}

!0 = !{i32 1, !"wchar_size", i32 4}
!1 = !{i32 7, !"PIC Level", i32 2}
!2 = !{i32 7, !"PIE Level", i32 2}
!3 = !{i32 7, !"uwtable", i32 2}
!4 = !{i32 7, !"frame-pointer", i32 2}
!5 = !{!"clang version 16.0.0"}
```

Listing 2: Primer LLVM međukoda

podataka i naziv ciljne arhitekture. U nastavku su globalne promenljive i funkcije. One se prepoznaju po tome što im imena imaju prefiks '@'.

Svaka funkcija se sastoji od niza osnovnih blokova, a svaki blok od niza instrukcija. Osnovni blok je jedinica čije instrukcije se uvek linearno izvršavaju. Ako se

uđe u osnovni blok on će se uvek izvršiti do kraja [1]. Dakle instrukcije grananja smeju i moraju da se pojave na kraju bloka. Granice osnovnih blokova su ozačene kao labele u jeziku C. Takođe, početak funkcije započinje prvi osnovni blok i kraj funkcije završava poslednji blok.

U okviru funkcije lokalne promenljive su zamenjene virtuelnim registrima. Identifikatori virtuelnih registara počinju karakterom '%'. Zbog SSA oblika, svakom od njih se može dodeliti vrednost tačno jednom. Smatra se da tih registara ima neograničen broj.

Instrukcije se sastoje od naziva (`alloca`, `call`, `icmp`, ...), tipa (`i32`, `float`, `label`, ...) i operanada. Ukoliko instrukcija vraća vrednost, on se smešta u novi virtuelni registar. Dodatne informacije o instrukcijama i globalnim objektima se čuvaju u vidu metapodataka. Oni se mogu prepoznati po tome što počinju karakterom '!'. Između ostalog, oni se koriste za informacije za debagovanje. Više reči o primeni metapodataka će biti u poglavlju 3.1.

Optimizacije su implementirane u vidu prolaza (eng. *pass*) koji obrađuju modul, funkciju ili petlju. Prolazi nasleđuju apstraktnu klasu `PassInfoMixin` i implementiraju funkciju `run` sa argumentima za odgovarajuću jedinicu obrade. Oni se dele na analize i transformacije. Analize prikupljaju podatke, dok transformacije koriste te podatke kako bi izmenili kôd. Transformacija zahteva različite vrste analiza i može da ih poništi ako više ne važe posle promene koda.

Optimizovanje programa se radi inkrementalno. Program dobijen kao izlaz iz jedne optimizacije postaje ulaz sledećoj optimizaciji. Dakle konačan program zavisi od redosleda izvršavanja optimizacija. Loš redosled može da rezultuje znatno lošijim performansama izvršnog fajla. Svaki nivo optimizacije propisuje koje optimizacije treba da budu primenjene i kojim redosledom.

Za primenu optimizacija na LLVM IR fajlu se koristi alat `opt`. On prima imena optimizacija koje treba da pokrene i fajl sa ispravnim IR programom koga optimizuje. Takođe moguće je pokretanje korisnički napisanih prolaza učitanih kao dinamičke biblioteke [27]. Primer korišćenja alata `opt` za primenu optimizacija `mem2reg` i `instnamer` izgleda ovako:

```
opt -passes="mem2reg,instnamer" -o output.ll input.ll
```

Zadnji deo

Pre generisanja mašinskog koda, postoji još jedna, mašinski zavisna, interna reprezentacija. U terminologiji projekta LLVM, ona se naziva MIR (eng. *Machine Intermediate Representation*). Ona je dosta bliža ciljnoj arhitekturi i sastoji se od konkretnih instrukcija za tu arhitekturu. MIR kôd se zapisuje u formatu YAML [2]. Odlike funkcija poput imena, mesta poziva itd. su opisane atributima formata YAML. Tela funkcija se takođe nalaze u okviru atributa, serijalizovani u obliku niske. Struktura instrukcija je slična kao za IR kôd. Najveće razlike su u tome što su intrukcije vezane za konkretnu arhitekturu i virtuelni registri su zamenjeni pravim registrima. Kompletan pregled jezika dostupan je u okviru dokumentacije projekta LLVM [14].

Veliki i kompleksan korak u prevođenju koda je prevođenje sa mašinski nezavisne na mašinski zavisnu reprezentaciju. Taj korak se naziva izbor instrukcija (eng. *Instruction Selection*). Postoje tri implementacije izbora instrukcija [26]:

`SelectionDAG` — Podrazumevana implementacija izbora instrukcija na većini arhitektura. Koristi grafovsku reprezantaciju programa i algoritme za uparivanje čvorova i različite vrste obilaska grafa [23].

`GlobalIsel` — Nova implementacija izbora instrukcija sa modularnim dizajnom, poboljšanim performansama i mogućnosti da optimizuje veći deo kôda. Trenutno nije završena implementacija, ali se planira da u budućnosti zameni `SelectionDAG`.

`FastIsel` — Pristup izboru instrukcija koji radi veoma brzo, ali generiše neoptimizovan kôd. Ne podržava spuštanje svih instrukcija i u tim situacijama se oslanja na `SelectionDAG`.

Ovaj rad se fokusira na implementaciju `SelectionDAG` zbog svoje stabilnosti i zato što je podrazumevana za arhitekturu `x86_64`.

Implementacija `SelectionDAG` je dobla ime po načinu reprezentacije koda za vreme izbora instrukcija. Naime, odgovarajuća klasa se takođe zove `SelectionDAG`. Osnovni blokovi programa su predstavljeni usmerenim acikličkim grafovima (eng. *Directed Acyclic Graphs*). Čvorovi tog grafa su instrukcije koje imaju tip `SDNode`, a grane su različite zavisnosti koje postoje između instrukcija. Postoje dva posebna čvora: ulazni čvor i koren. Ulazni čvor obeležava početak osnovnog bloka i služi samo

za postavljanje veza. Sa druge strane, koren označava kraj bloka. Na kraju obrade je vezan za poslednju instrukciju osnovnog bloka.

Konstrukcija grafa se radi prolaskom kroz LLVM međukod uz pomoć klase `SelectionDAGBuilder` koja implementira obrazac „posetilac" (eng. *visitor pattern*) [17]. Za svaku instrukciju se kreira čvor na osnovu njenog koda operacije. Konstrukcija čvora se radi direktno u klasi `SelectionDAG`. Veze se kreiraju kada neki čvor koristi izlaznu vrednost drugog čvora. Tip te vrednosti određuje tip veze.

Prethodno konstruisan graf nije pogodan za spuštanje instrukcija i nad njim se vrše dodatne transformacije. On se dodatno optimizuje zamenom grupe povezanih čvorova sa jednostavnijim grupama uz pomoć algoritama za uparivanje stabala. Te optimizacije mogu da budu i mašinski nezavisne i zavisne. Graf takođe može da sadrži nepodržane tipove i operacije za ciljnu arhitekturu. Njih je potrebno zameniti odgovarajućim podržanim varijantama. Ove transformacije se zovu legalizacija tipova i operacija. Način legalizacije zavisi od arhitekture i implementiran je u odgovarajućim `TargetLowering` klasama. Prolazi optimizacije i legalizacije se rade nekoliko puta u određenom redosledu i rezultuju grafom spremnim za spuštanje instrukcija.

Proces spuštanja instrukcija kada su sve potrebne pripreme izvršene je prilično jednostavan. Kao i prilikom optimizacija, algoritam se zasniva na uparivanju čvorova. Čvorovi svake `SelectionDAG` strukture traže svoj pandan za ciljnu arhitekturu. Arhitektura definiše ta uparivanja u svojoj podklasi klase `SelectionDAGISel`. Ona definiše funkciju `Select` koja prima objekat tipa `SDNode` i vraća čvor sa mašinski zavisnim kodom operacije. U ovom koraku se ne zamenjuju baš svi čvorovi već neki prolaze u sledeću fazu.

Poslednji trag LLVM međukoda koji je opstao u programu, a da je potrebno da se eliminiše, su virtuelni registri. U toku izvršavanja program će imati ograničen broj registara i ne može svaki od njih da se koristi za svaku operaciju. Dodeljivanje registara je korak kada se virtuelni registri zamenjuju konkretnim registrima za ciljnu arhitekturu. Pre dolaska u ovu fazu, već su dodeljeni registri koji su implicitno definisani od strane instrukcija ili konvencije poziva funkcije. Za dalje dodeljivanje registara se koriste različite heuristike pošto je optimalan algoritam NP-kompletan [3]. Jedna od njih je gramziva (eng. *greedy*) strategija koja daje prvenstvo dodeljivanju registara promenljivama sa dužim životnim vekom. Kada nije moguće zadržati sve potrebne promenljive u registrima, dodaju se nove instrukcije za čuvanje i učitavanje tih vrednosti u radnu memoriju.

Dobijeni objekat `SelectionDAG` predstavlja instrukcije potpuno ispravnog MIR programa. Ostalo je još da se one rasporede u linearni redosled uz poštovanje zavisnosti između čvorova. Ovo rezultuje kodom u MIR obliku.

Kao kod mašinski nezavisnog međukoda i na ovom nivou postoje optimizacioni prolazi. Optimizacije na ovom nivou su vezane za ciljnu arhitekturu. Svi mašinski zavisni prolazi nasleđuju klasu `MachineFunctionPass` i implementiraju funkciju `runOnMachineFunction` koja se poziva za svaku funkciju u izvornom kodu. Način izvršavanja prolaza je sličan kao za mašinski nezavisan međukod. Oni se izvršavaju iterativno u unapred zadatom redosledu.

Poslednji prolaz na MIR kodu je `AsmPrinter` koji emituje asemblerski ili objektni kôd za ciljnu arhitekturu. Na ovom mestu se završava zadatak kompilatora. Dalje, ukoliko je emitovan asemblerski kôd on se prevodi do objektnog koristeći asembler. Objektni kôd se povezuje sa sistemskim i korisničkim bibliotekama i drugim objektnim kodom pomoću povezivača (eng. *linker*) i dobija se izvršni fajl.

Alat koji obavlja posao zadnjeg dela kompilatora je `llc`. Njemu se opciono zadaje vrsta izlaza (asemblerski ili objektni kôd) i nivo optimizacije. Bitan parametar koji `llc` prihvata je ciljna arhitektura. Ona se zadaje argumentom `-mtriple` i vrednosti u vidu niske koja sadrži arhitekturu, proizvođača, operativni sistem i okruženje. Komanda za prevođenje mašinski nezavisnog međukoda do asemblerskog fajla za arhitekturu `x86_64` i operativni sistem Linux je sledeća:

```
llc -filetype=asm -mtriple="x86_64-unknown-linux-gnu" -o output.s
↪  input.ll
```

Glava 3

Debageri

Prilikom programiranja bilo kog većeg projekta, neminovno je da će doći do propusta. Propusti se odnose na bilo kakvo ponašanje programa koje nije u skladu sa namerom programera. U zavisnosti od komponente programa u kojoj se propust ispoljava, posledice mogu da budu više ili manje ozbiljne. Na primer, greška u korisničkom interfejsu uglavnom ima dosta manje posledice od greške u sistemu obrade novca u banci. Svakako, svi primećeni propusti trebaju biti popravljeni pre objavljivanja nove verzije softvera.

Proces reprodukovanja, analiziranja i popravljanja grešaka u programu naziva se *debagovanje*. Jedan od osnovnih načina je debagovanje štampanjem (eng. *print debugging*). Ono podrazumeva izmenu izvornog koda tako da ispisuje poruke i vrednosti promenljivih u glavnim koracima dela koda za koje programer smatra da su razlog za postojanje greške. Za greške u jednostavnim programima ovo je sasvim prihvatljiv način za njihovo rešavanje. Jedna od bitnijih mana tog pristupa je potreba za ponovnom kompilacijom celog programa svaki put kada se dodaje ili menja ispis. Kompilacija velikih projekata može da dugo traje, tako da se ovaj pristup ne isplati uvek. Alternativa je korišćenje posebnog alata namenjenog za ovu situaciju.

Debager je softverski alat koji olakšava proces debagovanja. On pruža interaktivan vid izvršavanja programa gde korisnik zadaje komande za izvršavanje programa i proveru vrednosti promenljivih. Umesto izmene izvornog koda, debager direktno ispituje i modifikuje program dok se izvršava. On uz pomoć operativnog sistema dobija potpunu kontrolu nad izvršavanjem programa. To mu daje pristup za čitanje i pisanje cele memorije programa dok se izvršava, uključujući i instrukcije koje će se izvršiti. Na taj način debager može i da kontroliše tok izvršavanja tog programa. Neke od funkcionalnosti debagera su:

Izračunavanje vrednosti izraza — Kada je proces zaustavljen na nekom mestu u kodu, korisnik može da proveri vrednosti promenljivih u tom trenutku. Neki debageri osim toga podržavaju i izračunavanje proizvoljnih izraza koji uključuju promenljive i funkcije programa koji se debaguje.

Izvršavanje korak po korak — Ovo predstavlja najčešći i najjednostavniji vid kretanja kroz program. Omogućava kontrolisano izvršavanje samo određenog dela koda. Korak može da označava jednu liniju izvornog koda, funkciju, instrukciju...

Postavljanje tačaka prekida — Ukoliko programer već naslućuje koji deo koda izaziva problem, on ne želi da izvršava korak po korak program dok se ne dostigne taj deo. Umesto toga može da postavi tačku prekida (eng. *breakpoint*) na to mesto i pusti program da se sam izvršava. Svaki put kada se dostigne tačka prekida proces će biti zaustavljen.

Praćenje vrednosti promenljivih — Neke greške su izazvane neočekivanom promenom vrednosti neke promenljive. Pogodan način za otkrivanje trenutka kada se to desi je postavljanjem tačke prekida nad podatkom (eng. *watchpoint*). Promenom vrednosti na toj adresi će se zaustaviti proces i kontrola se vratiti korisniku.

Pregled stek okvira (eng. *stack frame*) — Pronalazak mesta gde je greška ispoljena nije uvek dovoljno da se odredi gde ona nastaje. Taj deo koda se možda poziva sa više mesta i potrebno je odrediti koji od njih je izazvao problem. Pregledom stek okvira korisnik dobija uvid u istoriju poziva funkcija od početne funkcije.

Izvršavanje programa unazad — Nekada korisnik uoči problem tek nekoliko koraka nakon što je on nastao. Napredni debageri imaju podršku za poništavanje izvršenih instrukcija. Programski brojač (eng. *program counter*) se pomera unazad i sva izmenjena memorija se vraća na prethodno stanje.

Najpoznatiji debageri za jezike C i C++ su *GDB* [12], *LLDB* [15] i *Microsoft Visual Studio Debugger* [25].

Tradicionalan način korišćenja debagera je putem komandne linije. Sve operacije se izvršavaju zadavanjem komande u tekstualnom obliku u formatu koji debager propisuje. Upotrebom komande za pokretanje dela ili celog programa korisnik više

ne može da zadaje nove komande sve dok se prethodna komanda ne izvrši do kraja. Problem sa ovim vidom upotrebe je što korisnik mora da napamet zna strukturu komandi ili da otvara dokumentaciju. Drugi način za upotrebu debagera je pomoću integrisanih razvojnih okruženja (eng. *integrated development environment*). Mnoga okruženja ugrađuju debager u svoj grafički korisnički interfejs (eng. *graphical user interface*). Postojanje grafičkih elemenata olakšava posao korisniku zato što ne mora da memoriše sve komande. Dovoljno je da pritisne odgovarajuće dugme ili na neki način izmeni neki drugi grafički element. Ovo je najpogodniji način za nove korisnike da nauče da koriste debager. Postoji i sredina između prethodna dva načina upotrebe u vidu tekstualnog korisničkog interfejsa (eng. *textual user interface*). Umesto odvojenog prozora, i dalje se koristi komanda linija, ali je ona napravljena tako da liči na grafičko okruženje. Kako ne postoji podrška za kursor, komande se zadaju prečicama na tastaturi.

U nastavku će prvo biti objašnjen koncept informacija za debagovanje, a zatim će biti opisan format DWARF koji služi za zapisivanje tih informacija. Na kraju poglavlja je predstavljen debager LLDB i neki ključni delovi njegove implementacije.

3.1 Informacije za debagovanje

Za uspešno debagovanje programa na nivou izvornog koda (eng. source level debugging) potrebna je podrška kompilatora da sačuva informacije o tom kodu do izvršnog fajla. Pored mašinskog koda, kompilator takođe, ako mu je zatraženo, generiše i dodatne informacije koje se koriste za debagovanje. Između ostalog, one uključuju mapiranja mašinskih instrukcija u lokacije u izvornom kodu, imena funkcija i promenjivih itd. Sve te informacije zauzimaju dodatnu memoriju u krajnjem izvršnom fajlu i produžavaju vreme kompilacije. Iz tog razloga, one se podrazmevano ne generišu, već je potrebno eksplicitno navesti kompilatoru da treba to da uradi. Kompilator LLVM generiše ove podatke dodavanjem zastavice `-g` u okviru komande za kompilaciju.

Većina informacija za debagovanje nastaju u prednjem delu kompilatora, pošto je on zadužen za analiziranja izvornog koda, i prosleđuju se kroz ostale faze kompilacije gde se usput dopunjuju novim podacima kada oni postanu dostupni (npr. lokacije promenljivih za vreme izvršavanja). Očuvanje informacija za debagovanje nailazi na problem kada se uključe kompilatorske optimizacije. Prilikom primene optimizacija, informacije za debagovanje i dalje treba da reflektuju izvorni kôd, a ne njegovu

optimizovanu verziju. Što se više krajnji program razlikuje od izvornog koda postaje teže očuvati informacije, a nekada je to čak potpuno nemoguće. Ovo je jedan od najvećih izazova pisanja optimizacija. Jednostavno rešenje je isključiti kompilatorske optimizacije prilikom debagovanja i ono može da bude pogodno za neke korisnike. Ipak, postoje situacije kada je potrebno prevesti program sa optimizacijama. Na primer, ovo može biti slučaj kada da se program kompilira za računar sa malom količinom memorije gde ne može da se kompletno smesti ukoliko nije optimizovan za memoriju. Drugi problem prevođenja bez optimizacija je što se program onda može razlikovati od verzije koja se isporučuje klijentu i moguće je da se bag ispoljava samo u toj verziji. Iz prethodno navedenih razloga se posebna pažnja prilikom pisanja optmizacija daje održavanju informacija za debagovanje.

Ove informacije se nalaze u posebnim delovima izvršnog fajla odvojenim od svega ostalog. Izvršni fajl je podeljen u nekoliko delova koji se nazivaju segmenti. Najbitniji među njima su tekst segment gde se nalaze instrukcije programa, segment inicijalizovanih podataka i segment neinicijalizovanih podataka. Pored njih mogu da postoje i drugi segmenti, na primer za podatke o razmotavanju steka u slučaju greške. Informacije za debagovanje ne modifikuju nijedan od prethodnih segmenata već definišu nove kako ne bi uticali na postojeće podatke. Broj i naziv novih segmenata zavisi od formata za zapisivanje informacija za debagovanje koji se koristi. Najpopularniji među njima su format CodeView za *Windows* sisteme i format DWARF za sisteme tipa UNIX.

Postojanje standardizovanih formata ima veliki značaj. Ono omogućava korišćenje debagera nezavisno od kompilatora kojim je program preveden. Bitno je samo da oba koriste isti format. Ukoliko bi svaki kompilator pisao informacije za debagovanje u nekom vlasničkom (eng. proprietery) formatu, on bi mogao da se koristi samo sa odgovarajućim debagerom. Zbog standardizovanih formata, program preveden na primer kompilatorom GCC može da se debaguje debagerom LLDB. Iako koriste iste formate, ne mora da znači da će bilo koja dva debagera moći u potpunosti da iskoriste sve dostupne informacije za debagovanje ili da će ih korisiti na iste načine.

3.2 Format DWARF

DWARF [8] je standardizovan format za zapisivanje informacija za debagovanje. Originalno je osmišljen da radi uz format ELF [5] koji se koristi za biblioteke, izvršni i objektni kôd, ali podržava i druge formate. Postoji podrška za programske jezike C,

C++, Ada, Fortran i COBOL, ali je omogućeno proširenje i za druge jezike. Ovaj format se koristi na UNIX sistemima što uključuje *Linux* distribucije i *macOS*. Postojalo je nekoliko revizija standarda do sada, a trenutno je aktuelna verzija 5 koja će biti podrazumevana u nastavku rada.

Sve informacije za debagovanje su raspoređene u nove sekcije izvršnog fajla koje počinju prefiksom `.debug_`. Neke od najbitnijih su [9]:

`.debug_info` — Glavna sekcija za debagovanje. Sadrži skoro sve informacije o entitetima iz izvornog koda.

`.debug_line` — Tabela adresa instrukcija u izvršnom fajlu i odgovarajućih lokacija u izvornom kodu. Takođe sadrži dodatne podatke o granicama prologa i epiloga funkcija.

`.debug_frame` — Podaci o načinu razmotavanja stek okvira.

`.debug_loclists` — Lokacije skladištenja promenljivih za vreme izvršavanja. Za jednu promenljivu može da postoji više lokacija na primer ako se promeni registar u kome je smeštena.

`.debug_abbrev` — Skraćenice za zapisivanje informacija u `.debug_info` sekciji. Koristi se za kompresovanje podataka.

`.debug_str` — Tabela niski koje se koriste u `.debug_info` sekciji.

Podaci u sekciji `.debug_info` su predstavljeni pomoću *osnovnih jedinica* (eng. *Debug Information Entry, DIE*) organizovanih u drvoliku strukturu. Svaka jedinica ima etiketu (eng. *tag*) koja označava njen tip, dodatne informacije predstavljene atributima i može da ima druge jedinice kao decu. Etikete i atributi imaju numeričku vrednost koja služi za zapisivanje u računaru i ime u obliku niske koje služi čoveku za lako čitanje podataka. Standard propisuje validne etikete i atribute koje moraju da postoje za te etikete. Ideja je da svaki entitet programskog jezika bude opisan na ovaj način. U listingu 3 prikazan je primer opisa jedne promenljive. Značenja prikazanih atributa su objašnjena kasnije u odeljku.

Etiketa označava neki entitet izvornog koda ili koncept programskog jezika. Naziv etikete ima prefiks `DW_TAG_` posle kog može da sledi oznaka proizvođača ako je u pitanju proširenje. Neke od najbitnijih etiketa su:

`DW_TAG_compile_unit` — Predstavlja jednu kompilacionu jedinicu. Kao decu sadrži prostore imena (eng. *namespace*), funkcije, globalne promenljive itd.

```
DW_TAG_variable
    DW_AT_location  (DW_OP_fbreg -8)
    DW_AT_name      ("x")
    DW_AT_decl_file ("/home/vladimir/Documents/example.c")
    DW_AT_decl_line (2)
    DW_AT_type      (0x00000052 "int")
```

Listing 3: Primer DWARF jedinice koja opisuje promenljivu

`DW_TAG_subprogram` — Predstavlja jednu funkciju. Sadrži podatke o imenu, parametrima i opsegu funkcije u izvršnom fajlu, između ostalog.

`DW_TAG_call_site` — Označava mesto poziva funkcije. Sadrži referencu na jedinicu `DW_TAG_subprogram` za pozvanu funkciju i adresu povratka iz poziva.

`DW_TAG_variable` — Opisuje jednu promenljivu. Ima informacije o njenom imenu, lokaciji u izvršnom fajlu i mestu deklaracije u izvršnom kodu.

`DW_TAG_base_type` — Pruža informacije o osnovnim tipovima uključujući njihov način zapisa i veličinu. Postoje slične etikete za tipove nizova i niski.

Atributi opisuju svojstva objekata označenih etiketom uz koju stoje. Isti atribut može imati više interpretacija u zavisnosti od etikete, ali jedan atribut se ne može ponavljati više puta u osnovnoj jedinici. Redosled atributa unutar jedinice nije definisan i ne treba se oslanjati da će biti konzistentan. Slično kao za etikete, atribut počinje niskom `DW_AT_` posle koje može da sledi oznaka proizvođača ako je u pitanju proširenje. Neki od najbitnijih atributa su:

`DW_AT_name` — Može da opisuje ime funkcije, promenljive, labele itd.

`DW_AT_low_pc`, `DW_AT_high_pc` — Oba atributa zajedno opisuju opseg adresa. To može da bude opseg kompilacione jedinice, funkcije, postojanja promenljive itd. Ukoliko ne postoje informacije o obe granice, mogu se koristiti i odvojeno.

`DW_AT_decl_file` — Fajl izvornog koda u kome je deklarisan objekat.

`DW_AT_decl_line` — Red u izvornom kodu na kome je deklarisan objekat.

`DW_AT_decl_col` — Kolona u izvornom kodu na kojoj je deklarisan objekat.

`DW_AT_type` — Može da označava različite stvari u zavisnosti od konteksta. Najčešće je u pitanju tip koji ima neka promenljiva ili povratna vrednost funkcije.

Bitna odlika formata DWARF je njegova proširivost. Standard pruža način da se format dopuni novim etiketama i atributima od strane proizvođača kompilatora. Razni alati mogu zatim da koriste te podatke kako bi poboljšali svoju funkcionalnost. Sa druge strane, postojeći alati koji ne razumeju te podatke mogu samo da ih preskoče. Na taj način se održava kompatibilnost sa postojećim alatima. Slično važi za alate koji su napravljeni za starije verzije standarda. Bitna stvar pilikom dodavanja etiketa i atributa je da im se dodele numeričke vrednosti koje nisu već deo standarda i poželjno bi bilo da se takođe ne koriste od strane drugih proizvođača kompilatora. Raspon dozvoljenih vrednosti je dovoljno veliki da ne dođe do preklapanja vrednosti. Kompilatori često koriste samo neki mali opseg za svoje potrebe.

3.3 Debager LLDB

LLDB je debager koji se razvija kao deo projekta LLVM. Napravljen je kao skup ponovo upotrebljivih komponenti koje koriste biblioteke iz projekta LLVM kao što su parser izraza biblioteke clang ili disasembler biblioteke LLVM [15]. Podržava programske jezike C, C++ i Objective-C i dostupan na svim popularnim operativnim sistemima: *Windows*, *macOS* i *Linux*. LLDB je podrazumevani debager u integrisanom razvojnom okruženju *Apple Xcode* [15] i podržan je od strane drugih okruženja uključujući *JetBrains CLion* [21], *QtCreator* [6], *Eclipse* [10] i *VSCode* [24].

Primarni vid upotrebe alata LLDB je putem komandne linije. Korisnik upravlja načinom izvršavanja debagovanog procesa tako što zadaje komande debageru. Te komande su uniformnog oblika. Prvo se zadaje objekat nad kojim treba izvršiti neku operaciju. To može biti proces, nit, tačka prekida itd. Posle toga sledi operacija koja će biti izvršena nad tim objektom. Raspoložive operacije zavise od objekta. U slučaju niti to mogu biti izvršavanja koraka, nastavak izvršavanja programa itd. Na kraju komande se nalaze opcije i argumenti ukoliko su potrebni. Na primer, za postavljanje tačke prekida može se zadati ime funkcije na koju je treba postaviti. Najčešće korišćene komande imaju definisane skraćenice radi lakše upotrebe. Tako se izvršavanje jednog koraka bez ulaska u funkcije može učiniti komandom `thread step-over`, skraćenicom `next` ili samo `n`.

Izvorni kôd projekta LLDB je dostupan u okviru repozitorijuma projekta LLVM, u direktorijumu `lldb`. Glavni deo debagera je implementiran kao biblioteka `liblldb` tako da se mogu napraviti novi alati bazirani na njoj. Sama aplikacija LLDB je jedan od tih alata. Njena implementacija se nalazi u folderu `tools/Driver`. Tu se

samo obrađuju argumenti komandne linije i vrše pripreme za izvršavanje dok je debagovanje delegirano biblioteci.

Debager mora da bude sposoban da predstavi stanje programa koji se debaguje. Klase zadužene za to se nalaze u direktorijumu `Target`. Neke od najbitnijih su:

`Target` — Interfejs za kontrolu izvršavanja ciljnog programa. Zadužen je za kreiranje i modifikovanje tačaka prekida, obaveštavanje objekata kada nastane neki događaj (eng. *event*) i upravljanje procesima.

`Process` — Predstavlja proces koji se debaguje. Sadrži listu niti tog procesa i prosleđuje im događaje koje dobija od klase `Target`.

`Thread` — Jedna nit procesa koji se debaguje. Određuje način izvršavanja i sadrži listu stek okvira.

`StackFrame` — Stek okvir koji pripada jednoj niti procesa koji se debaguje. Stek okviri jedne niti su numerisani tako da najnoviji okvir ima indeks 0, njegov roditelj ima indeks 1, itd. Kroz instancu okvira je moguće pristupiti promenljivama koje su dostupne u tom trenutku izvršavanja.

Ovi podaci nisu ažurni u toku izvršavanja procesa već se računaju po potrebi kada se debagovan proces zaustavi.

Informacije za debagovanje su integrisane pomoću klasa u direktorijumu `Symbol`. Učitavanje i pristup tim informacijama je ostvaren kroz interfejs `SymbolFile`. Podržani formati za debagovanje su implementirani u vidu priključaka (eng. *plug-ins*) koji implementiraju taj interfejs.

Mnoge operacije debagera zahtevaju informacije o simbolu na nekom mestu u programu. U izvornom kodu te informacije se nazivaju kontekst i skladište se u objektima klase `SymbolContext`. On sadrži podatke o ciljnom programu, modulu, kompilacionoj jedinici, funkciji, bloku koda, lokaciji u izvornom fajlu, simbolu i globalnoj promenljivoj. Ne moraju uvek da budu prisutni svi ti podaci, a nekada nisu ni svi dostupni. Kontekst može da se popunjava selektivno, samo podacima koji su potrebni za neku operaciju.

Određivanje konteksta se radi na osnovu informacija za debagovanje. Klasa `SymbolContextScope` opisuje objekte koji su sposobni da odrede svoj kontekst. Nju nasleđuje skoro svaka klasa koja je sadržana u kontekstu. Izuzeci su klasa `StackFrame` koja je ne nasleđuje, ali još uvek može da odredi kontekst na drugi način i klasa `LineEntry` koji ne podržava tu operaciju. Zaključivanje konteksta na

osnovu klase `LineEntry` je moguće u okviru klase `CompileUnit` koja opisuje kompilacionu jedinicu programa.

Najčešći načini dobijanja konteksta su preko adrese instrukcije u izvršnom fajlu ili preko lokacije u izvornom kodu. Drugi način se svakako svodi na prvi određivanjem adrese na osnovu lokacije. Adresa koja se najviše koristi za određivanje konteksta je vrednost programskog brojača koji pokazuje na instrukciju koja se trenutno izvršava. Dobijen kontekst služi za izvršavanje skoro svih ranije opisanih mogućnosti debagera.

Glava 4

Optimizacija izdvajanjem koda

Izdvajanje koda (eng. *outlining*) je interproceduralna kompilatorska optimizacija koja smanjuje količinu memorijskog prostora koji zauzima program, ali pritom potencijalno povećava njegovo vreme izvršavanja. Ona pronalazi segmente koda (uzastopne nizove instrukcija) koji se ponavljaju u programu, izdvaja ih u zasebnu funkciju i menja pojavljivanja tog segmenta sa pozivom ka novoj funkciji. U nekim slučajevima moguće je da se ubrza i vreme potrebno za kompilaciju programa [18].

Ova optimizacija je posebno korisna za uređaje sa malom količinom memorije, najčešće uređajima sa ugrađenim računarom poput pametnih satova, mp3 plejera ili urađaja zasnovanih na razvojnom sistemu Arduino. Vreme izvršavanja programa može da bude oštećeno ako se izdvoji deo koda koji se baš često izvršava (eng. *hot-spot*). To je zato što je poziv funkcije skupa instrukcija. Pre izvršavanja poziva, stek mora da bude pripremljen za poziv i parametri premešteni na odgovarajuća mesta koja propisuje arhitektura. Sve to dodaje trošak koji nije prethodno postojao. Izdvajanjem koda koji se retko izvršava, sa druge strane, moguće je da dođe do poboljšanja u vremenu izvršavanja zbog bolje organizacije keš memorije. Nažalost, u vreme kompilacije je teško odrediti koliko puta će se izvršiti segment koda, time ni razlika u vremenu izvršavanja programa nije lako predvidiva.

U listingu 4 je prikazan primer C koda na koji može da se primeni ova optimizacija. Redovi 11-14 i 16-19 predstavljaju dobre kandidate za izdvajanje. Vrše se iste operacije samo sa drugačijom promeljivom. Ukoliko se ovaj primer prevede do asemblerskog koda komandama prikazanim u listingu 5 tako što se pokrene samo optimizacija izdvajanja koda, rezultujući kôd je prikazan u listingu 6. Može se primetiti da je konstruisana nova funkcija `outlined_ir_func_0`. Ona sadrži sve instrukcije iz izdvojenih redova, a ti redovi su zamenjeni pozivom te nove funkcije.

```c
int global;

__attribute__((noinline)) int foo(int a, int b) {
  return a / b * 2;
}

int main() {
  int x = 0;
  int y = 1;

  int c1 = x + y;
  c1--;
  global += 2;
  foo(c1, global);

  int c2 = x + y;
  c2--;
  global += 2;
  foo(c2, global);

  return c1;
}
```

Listing 4: Primer C programa koji je dobar kandidat za izdvajanje koda

```
$ clang -Xclang -disable-llvm-passes -g -O3 -S -emit-llvm outline.c -o
↪  outline.ll
$ opt -passes="iroutliner" -ir-outlining-no-cost=true -S outline.ll -o
↪  outline_opt.ll
$ llc -filetype=asm outline_opt.ll -o outline.s
```

Listing 5: Instrukcije za prevođenje programa do asemblerskog koda sa uključenim izdvajanjem instrukcija

Ova optimizacija predstavlja suprotan proces poznatijoj optimizaciji, umetanju koda. Umetanje koda podrazumeva zamenu poziva funkcije sa instrukcijama iz tela te funkcije. Najčešće se primenjuje na kratke i jednostavne funkcije, koje se pozivaju na više mesta. Time se poništava cena poziva funkcije, što poboljšava vreme izvršavanja, ali pritom pošto su neki delovi koda uvišestručeni dolazi do povećanja memorijskog zauzeća koda. Iako imaju suprotne efekte, obe optimizacije je moguće koristiti i zajedno, na primer za delimično umetanje [35]. Ova ideja koristi izdvajanje koda kako bi iz funkcije namenjene za umetanje bili izdvojeni „hladni” delovi koda i time se smanjile negativne posledice umetanja. U tom kontekstu izdvajanje koda ne

```
; ModuleID = 'outline.ll'
source_filename = "outline.c"
target datalayout =
↪  "e-m:e-p270:32:32-p271:32:32-p272:64:64-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-unknown-linux-gnu"

...

define dso_local i32 @main() #2 !dbg !30 {
  ...
  call void @outlined_ir_func_0(ptr %2, ptr %3, ptr %4), !dbg !44
  ...
  call void @outlined_ir_func_0(ptr %2, ptr %3, ptr %5), !dbg !47
  ...
}

...

define internal void @outlined_ir_func_0(ptr %0, ptr %1, ptr %2) #4 !dbg !51 {
newFuncRoot:
  br label %_to_outline

_to_outline:                                      ; preds = %newFuncRoot
  %3 = load i32, ptr %0, align 4, !tbaa !19
  %4 = load i32, ptr %1, align 4, !tbaa !19
  %5 = add nsw i32 %3, %4
  store i32 %5, ptr %2, align 4, !tbaa !19
  %6 = load i32, ptr %2, align 4, !tbaa !19
  %7 = add nsw i32 %6, -1
  store i32 %7, ptr %2, align 4, !tbaa !19
  %8 = load i32, ptr @global, align 4, !tbaa !19
  %9 = add nsw i32 %8, 2
  store i32 %9, ptr @global, align 4, !tbaa !19
  %10 = load i32, ptr %2, align 4, !tbaa !19
  %11 = load i32, ptr @global, align 4, !tbaa !19
  %12 = call i32 @foo(i32 noundef %10, i32 noundef %11), !dbg !54
  br label %_after_outline.exitStub

_after_outline.exitStub:                          ; preds = %_to_outline
  ret void
}
```

Listing 6: Isečci iz asemblerskog koda koji sadrži funkciju dobijenu izdvajanjem koda

traži sekvence koje se ponavljaju već koristi heuristike da pronađe kôd koji se retko izvršava.

Izdvajanje koda nije zamena za definisanje novih funkcija u izvornom kodu koje implementiraju operacije koje se često ponavljaju, već je dopuna koja radi dodatno optimizovanje za memoriju na nivou bližem mašinskom kodu. Ukoliko kompilator

podržava ovu optimizaciju, njegov zadatak je da izabere delove koda koje je poželjno izdvojiti tako da se postigne što veća ušteda memorije. Ova optimizacija nije podrazumevano uključena, već programer može da je uključi na podržanim kompilatorima najčešće kompilacijom programa sa optimizacijama za memoriju umesto za vreme.

4.1 Implementacija u kompilatoru LLVM

Kompilator LLVM, kao jedan od najpopularnijih kompilatora za jezike C i C++ ima podršku za veliki broj optimizacija uključujući i izdvajanje koda. Postoje dve implementacije izdvajanja koda koje se primenjuju u različitim fazama kompilacije:

1. `MachineOutliner` — na mašinski zavisnom međukodu (MIR) [28]
2. `IROutliner` — na mašinski nezavisnom međukodu (IR) [22]

Obe su implementirane kao optimizacioni prolazi koji rade na nivou modula tako da omoguće izdvajanje koda iz različitih funkcija. Ideja je da obe implementacije komplementiraju jedna drugu i daju bolje rezultate nego ukoliko bi se koristila samo jedna od njih. Svaka od njih ima svoje prednosti i mane. Na mašinski zavisnom međukodu su dostupne informacije o ciljnoj arhitekturi čime je moguće dosta bolje proceniti cene izdvajanja instrukcija. Bolji model cena instrukcija bitno utiče na određivanje delova koda koji će biti izdvojeni. Problem na ovom nivou je što je u tom trenutku već izvršena alokacija registara. Kandidati za izdvajanje su ograničeni samo na sekvence koje koriste iste registre, iako možda postoji više sekvenci koje vrše isto izračunavanje samo sa drugim registrima. Sa druge strane, na mašinski nezavisnom međukodu se koriste virtuelni registri. Oni ne moraju da se poklapaju između sekvenci tako da je moguće izdvojiti mnogo veći deo koda. Međutim ovde se sada javlja problem procene dobitka memorije zato što se bez konkretnih instrukcija za ciljnu arhitekturu ne može odrediti njihova tačna veličina. Zbog korišćenja procena za cene instrukcija u obe implementacije je moguće da rezultat izdvajanja koda ima veću veličinu nego što bi imao bez te optimizacije, ali to je redak slučaj.

Obe implementacije se zasnivaju na istoj ideji, samo su adaptirane za međukod koji optimizuju. U nastavku će prvo biti objašnjen opšti algoritam, a zatim specifičnosti obe implementacije.

Opšti algoritam za izdvajanje koda

Opšti algoritam za izdvajanje koda se može podeliti na dva osnovna koraka. Prvo se traže svi ponovljeni delovi koda koje bi bilo poželjno izdvojiti. Svaka instanca koja se ponavlja više puta se naziva kandidat za izdvajanje, a sve instance sa istim instrukcijama čine grupu kandidata. Zatim se za neke od tih grupa kandidata koje su izabrane heuristikom generiše po jedna nova funkcija na osnovu instrukcija iz jednog od kandidata iz te grupe i svi kandidati se zamenjuju pozivom te funkcije. Oba koraka sadrže delove koji zavise od međukoda na kom se primenjuju. Oni su opisani u narednim odeljcima.

Algoritam za odabir kandidata se zasniva na ideji da se program može tretirati kao niz numeričkih vrednosti gde se svakoj instrukciji dodeljuje jedna vrednost. Instrukcije se dele u tri grupe:

- Legalne instrukcije su one koje je dozvoljeno izdvojiti. Tu spada većina instrukcija: aritmetičke, logičke...
- Ilegalne instrukcije su one čije izdvajanje je zabranjeno zato što bi moglo da promeni semantiku programa ili zato što zahteva dodatne provere koje nisu podržane. To uključuje, na primer, pozive funkcija ili instrukcije vezane za obradu grešaka.
- Nevidljive instrukcije ne utiču na izdvajanje, ovo su primarno instrukcije koje se koriste za debagovanje.

Pojedinačne vrednosti dodeljene instrukcijama se nadovezuju za svaki osnovni blok i svaki od njih se terminira jedinstvenom vrednosti kako bi se sprečilo izdvajanje koda preko granice blokova. Nadovezivanjem tih reprezentacija osnovnih blokova se dobija reprezentacija celog programa. Nad takvom reprezentacijom programa sada mogu da se primene algoritmi za traženje ponavljajućih podniski (eng. *longest repeated substring*). Implementirani algoritam koristi *sufiksno stablo* (eng. *suffix tree*) [36] za rešavanje tog problema. Ono može efikasno da se sagradi u linearnoj vremenskoj složenosti u odnosu na broj instrukcija u programu pomoću Ukonenovog algoritma [34]. Broj čvorova u ovom stablu je linearno zavisan od broja instrukcija, što znači da je obilazak ovog stabla takođe linearne složenosti.

Svaki unutrašnji čvor osim korena označava segment koji se ponavlja, to jest jednu grupu kandidata. Konkatenacijom vrednosti grana od korena do tog čvora se dobija ponovljen segment, a svi listovi koji su potomci tog čvora označavaju

mesta početka svakog kandidata. Dakle, traženje svih ponavljajućih podniski se može obaviti jednostavnim obilaskom stabla.

Kada se odrede kandidati prelazi se u fazu izdvajanja koda. U petlji se iterativno bira grupa kandidata za izdvajanje koja bi napravila najveći procenjeni doprinos za uštedu memorije. Način računanja doprinosa zavisi od implementacije. U oba slučaja se dobija kao procena razlike u veličini programa ukoliko se grupa ne izdvoji i ako se izdvoji. Za odabranu grupu se generiše nova funkcija, a svi kandidati se zamenjuju pozivom te funkcije. Pritom se pazi da se održi semantika programa na tom međukodu. Detalji ovog postupka takođe variraju u zavisnosti od implementacije. Posle svakog odabira kandidata, ta grupa kao i svi drugi kandidati koji se preklapaju sa njom se brišu iz sufiksnog stabla i postupak se ponavlja sve dok postoje grupe sa pozitivnim doprinosom. Ukoliko ih više nema, prolaz izdvajanja koda je završen.

Specifičnosti implementacije izdvajanja koda na mašinski zavisnom međukodu

Izdvajanje koda na mašinski zavisnom međukodu je implementirano u klasi `MachineOutliner`. Ovo je jednostavnija od dve implementacije. Potrebno je dopuniti nekoliko delova opšteg algoritma. Za kreiranje reprezentacije pogodne za pretragu podniski potrebno je definisati algoritam za heširanje instrukcija. Izbor kandidata za izdvajanje zahteva definisanje metrike za procenu dobitka i neophodan je algoritam za izdvajanje tih kandidata u funkcije. U nastavku će biti objašnjen svaki od ovih delova.

Heširanje instrukcija na ovom nivou je jednostavno. Poređenje se vrši po jednakosti, dakle instrukcija i operandi moraju da budu identični. To predstavlja jedno od najvećih ograničenja na ovom nivou. Ukoliko dve sekvence vrše isto izračunavanje ali njihovim operandima su dodeljeni različiti registri, neće biti moguće izdvojiti taj kôd. Klasifikaciju instrukcija na ovom nivou radi odgovarajuća instanca klase `TargetInstrInfo` za ciljnu arhitekturu.

Procena dobitka memorije se računa kao razlika između veličine programa bez i sa uključenom optimizacijom. Kako se ona radi na nivou jedne grupe kandidata, dovoljno je posmatrati samo instrukcije koje pripadaju toj grupi. Veličina grupe kadidata bez optimizacije ne mora da se proceni, jer je već dostupna kao proizvod veličine jedne grupe i ukupnog broja grupa. Veličina sa primenjenom optimizacijom zahteva više računa. Na kraju optimizacije će postojati jedan kandidat iz grupe

koji predstavlja telo izdvojene funkcije. Njegova veličina se sabira sa procenama cena svih poziva izdvojene funkcije i cenama dodatnih instrukcija za kreiranje novog okvira za funkciju. Sve to ukupno čini procenu sa uključenom optimizacijom. Cene instrukcija na ciljnoj arhitekturi kao i procene poziva se dobijaju kroz interfejs `TargetInstrInfo`.

Postupak kreiranja izdvojene funkcije na ovom nivou je jednostavan. Generiše se nova funkcija sa imenom `OUTLINED_FUNCTION_X` gde je `X` redni broj izdvojene funkcije. Kopiraju se instrukcije proizvoljnog kandidata i menjaju se informacije za debagovanje da referišu na novu funkciju. Svi kandidati se zamenjuju pozivom ka generisanoj funkciji bez potrebe za dodavanje bilo kakvih novih instrukcija.

Specifičnosti implementacije izdvajanja koda na mašinski nezavisnom međukodu

Izdvajanje koda na mašinski nezavisnom međukodu radi prolaz koji se zove `IROutliner`. Ova implementacija je znatno komplikovanija nego mašinski zavisna varijanta, ali pruža više mogućnosti za izdvajanje. U nastavku će, kao i za prethodnu implementaciju, biti objašnjeni algoritam za heširanje instrukcija, metrika za procenu dobitka i način generisanja izdvojenog koda.

Za određivanje kandidata se koristi pomoćna analiza, `IRSimilarityIdentifier`. Umesto poređenja celih instrukcija, poredi se samo tip instrukcije i operanda i dodatni parametri za neke instrukcije (npr. ime pozvane funkcije za instrukciju poziva ili tip predikta za instrukciju poređenja). Operandi ne utiču na heš vrednost instrukcije zato što se oni mogu izvući kao argumenti izdvojene funkcije. Zbog toga je moguće izdvajanje dosta više koda.

Dobitak se računa slično kao na mašinski zavisnom nivou, kao razlika bez i sa uključenom optimizacijom. Kako je ovaj međukod dalji od krajnjeg koda, procena postaje teža zato što se oslanja na manje precizne podatke. Cena bez optimizacije se ovaj put računa kao suma procena veličina svih kandidata koji u ovom slučaju mogu da se razlikuju. Sa druge strane, cena sa optimizacijom sadrži prosečnu veličinu kandidata i procene veličina svih instrukcija za smeštanje i učitavanje potrebnih argumenta kao i za stvaranje nove funkcije. Procene cena instrukcija nakon transformacije na ciljnu arhitekturu se dobija kroz interfejs `TargetTransformInfo`.

Usled postojanja argumenata u izdvojenoj funkciji, način izdvajanja postaje komplikovaniji. Kandidati se izdvajaju iz dva pokušaja. Prvi put kada se izdvoje

određuju se svi argumenti koje će funkcija imati. Svi potrebni argumenti su poznati tek nakon prvog prolaska tako da se drugi put izdvajanje vrši uz zamenu argumenata. Jedan od izdvojenih kandidata se proglašava za konačnu izdvojenu funkciju i njemu se dodeljuje ime `outlined_ir_func_x` gde je `x` redni broj izdvojene funkcije. Informacije iz svih izdvojenih kandidata se spajaju sa izabranom funkcijom i zatim brišu. Na kraju, svaki kandidat se zamenjuje pozivom ka funkciji prilikom čega se dodaju i instrukcije za čuvanje i učitavanje argumenata.

4.2 Izdvajanje koda i debag lokacije

Debag lokacije predstavljaju mesta u izvornom kodu koja odgovaraju instrukciji u izvršnom fajlu. Sastoje se od fajla, reda i kolone. U idealnom slučaju, svaki izraz i naredba u izvornom kodu bi trebalo da ima odgovarajuću lokaciju, ali to nije slučaj pogotovu u optimizovanom kodu. Veliki broj optimizacija menja kôd na način koji može da naruši tačnost debag lokacija. Neki od najčešćih slučajeva su brisanje nepotrebnih instrukcija ili pomeranja instrukcija van tela petlje. Kompilatori se trude da održe što veću količinu debag lokacija, ali nekada to ili nije moguće ili nema smisla posle izvršavanja optimizacije.

U nastavku se podrazumeva da se koristi format DWARF za čuvanje informacija za debagovanje. Lokacije za debagovanje se interno čuvaju u tabeli linija u `.debug_line` sekciji izvršnog fajla. Ta tabela preslikava adresu iz tekst (kôd) segmenta izvršnog fajla na odgovarajuću lokaciju (fajl, red, kolona) u izvornom kodu i neke dodatne informacije o toj lokaciji. Format DWARF propisuje način enkodiranja tabele linija u vidu konačnog automata, ali postoje alati koji prikazuju podatke u obliku tabele. Listing 7 prikazuje upotrebu nekoliko alata za dekodiranje tabele linija, a u listingu 8 se nalazi primer jedne tabele.

```
$ llvm-dwarfdump --debug-line [file]
$ readelf -wL [file]
$ objdump -WL [file]
```

Listing 7: Komande za ispis tabele linija koristeći alate llvm-dwarfdump, readelf i objdump

Debageri prikazuju lokaciju iz tabele koja je vezana za prvu adresu koja je manja ili jednaka adresi instrukcije na kojoj je program zaustavljen. Na taj način ukoliko trenutna instrukcija nema svoju debag lokaciju, prikazaće se lokacija prve prethodne

```
Address            Line   Column File   ISA Discriminator Flags
------------------ ------ ------ ------ --- ------------- -------------
0x0000000000401110      1      0      1   0             0  is_stmt
0x000000000040111b      2      6      1   0             0  is_stmt prologue_end
0x0000000000401122      3      3      1   0             0  is_stmt
0x000000000040112b      4      2      1   0             0  is_stmt
0x000000000040112f      4      2      1   0             0  is_stmt end_sequence
```

Listing 8: Tabela linija ispisana pomoću alata llvm-dwarfdump

instrukcije koja ima lokaciju. Standard propisuje i specijalne debag lokacije koje se poznaju po tome sto imaju red 0. Ove lokacije znače da je kôd kompilatorski generisan i da ne postoji odgovarajuća lokacija u izvornom kodu. Prilikom izvršavanja programa red po red (koristeći komandu `step` u debageru LLDB) preskaču se sve instrukcije sa istim redom kao prethodna ili sa redom 0. Na instrukciju koja nema dodeljenu lokaciju je moguće stati izvršavanjem programa instrukciju po instrukciju, postavljanjem tačke prekida na adresu te instrukcije ili usled zaustavljanja programa zbog izuzetka koji se desio u toj instrukciji.

Prilikom izdvajanja koda jedna instrukcija ima više mogućih debag lokacija u zavisnosti od mesta poziva izdvojene funkcije. Ovo predstavlja više problema:

- U vreme izvršavanja programa nije moguće odrediti koja od ovih lokacija je korektna bez dodatnih informacija
- Format DWARF ne podržava postojanje više različitih lokacija za istu adresu [8]

Radi održavanja korektnosti, prilikom izdvajanja koda brišu se lokacije svih izdvojenih instrukcija. Kao posledica toga mogu se javiti nepogodnosti prilikom debagovanja:

- Izvršavanje korak po korak pravi velike skokove kada dođe do izdvojenog koda, bez bilo kakvog obaveštenja korisniku
- Postavljanje tačaka prekida na izdvojenu instrukciju precizirajući njen red nije moguće

Primer debagovanja sa izdvojenim kodom je prikazan u listingu 9. Redovi 11-14 i 16-19 su izdvojeni u funkciju. Kada debager koračanjem naredbu po naredbu dođe do nekog od tih regiona, u sledećem koraku ga celog preskoči. Slično, pokušaj postavljanja tačke prekida ima neočekivan ishod. U primeru prikazanom u listingu

```
$ lldb outline
(lldb) breakpoint set --line 11
(lldb) run
    8           int x = 0;
    9           int y = 1;
    10
 -> 11          int c1 = x + y;
    12          c1--;
    13          global += 2;
    14          foo(c1, global);
(lldb) next
    13          global += 2;
    14          foo(c1, global);
    15
 -> 16          int c2 = x + y;
    17          c2--;
    18          global += 2;
    19          foo(c2, global);
(lldb) next
    18          global += 2;
    19          foo(c2, global);
    20
 -> 21          return 0;
    22    }
```

Listing 9: Proces debagovanja programa sa izvojenim kodom. Izdvojeni delovi koda su preskočeni prilikom izvršavanja korak po korak.

10 pokušano je postavljanje tačke prekida na liniju 13, ali kako je ona izdvojena i nema lokaciju, tačka je postavljena tek na liniju 16. Ovaj rad pokušava da reši

```
$ lldb outline
(lldb) breakpoint set --line 13
Breakpoint 1: where = outline`main + 87 at outline.c:16:11, address =
↪  0x0000000000001197
(lldb) run
    13          global += 2;
    14          foo(c1, global);
    15
 -> 16          int c2 = x + y;
    17          c2--;
    18          global += 2;
    19          foo(c2, global);
```

Listing 10: Postavljanje tačke prekida na izdvojenu naredbu.

prethodno opisane probleme.

Glava 5

Implementacija rešenja

Implementacija rešenja nadograđuje verziju 16 projekta LLVM, što je najnovija stabilna vezija u vreme pisanja rada. Dodatno se koristi osnova za prepoznavanje izdvojenih instrukcija implementirana u [33]. Celokupan izvorni kôd je dostupan na servisu GitHub na adresi `https://github.com/vvuksanovic/llvm-project/tree/outline-debug-info`. Komande za preuzimanje i kompilaciju projekta su prikazane u listingu 11.

```
$ git clone https://github.com/vvuksanovic/llvm-project.git
$ cd llvm-project
$ git checkout outline-debug-info
$ mkdir build && cd build
$ cmake -G Ninja -DCMAKE_ENABLE_PROJECTS="clang;lldb"
↪  -DCMAKE_BUILD_TYPE=Release -DLLVM_ENABLE_ASSERTIONS=ON ../llvm
$ ninja
```

Listing 11: Komande za preuzimanje i prevođenje kompilatora LLVM

Rešenje se sastoji iz dva dela. Prvo je potrebno u okviru kompilatora generisati potrebne informacije za debagovanje, a zatim ih iz debagera iskoristiti za poboljšanje procesa debagovanja. U nastavku će redom biti opisana oba dela implementacije.

5.1 Implementacija u kompilatoru

Zadatak ovog dela je da prilikom primene optimizacije izdvajanja funkcija zapamti originalne lokacije u izvornom kodu, prenese ih kroz sve faze kompilacije i ispiše ih u sekciju za debagovanje izvršnog fajla. Potrebne informacije se generišu u okviru implementacija optimizacije, u prolazima `IROutliner` i `MachineOutliner`.

Kako tabela linija ne podržava duplirane vrednosti adrese, ona se ne može iskoristiti za čuvanje vrednosti. Umesto toga, ideja je da se debag lokacije sačuvaju u sekciji `.debug_info` umesto u `.debug_line`. Verzija 5 standarda DWARF je uvela novu etiketu, `DW_TAG_call_site`, koja bi trebalo da sadrži podatke o mestu poziva funkcije [19]. Trenutno se već koristi za pružanje informacija o vrednostima parametara prilikom poziva funkcije [20] i da li je poziv repni ili repni rekuzivan. Ovo je pogodno mesto za dodavanje informacija o izdvojenim instrukcijama zato što su one baš vezane za mesto poziva izdvojene funkcije.

Implementacija trenutno podržava čuvanje podataka prilikom izdvajanja koda na mašinski nezavisnoj međureprezentaciji. Kako to uključuje spuštanje kroz mašinski zavisnu reprezentaciju, jednostavno je dodati podršku za izdvajanje koda i na toj reprezentaciji. Za izbor instrukcija se koristi implementacija `SelectionDAG`, tako da će čuvanje podataka raditi za sve podržane arhitekture. Glavni fokus ove implementacije je `x86_64` arhitektura, ali količina koda specifičnog za tu arhitekturu je minimalna.

Debag lokacije moraju da budu prenesene od optimizacionog prolaza `IROutliner`, kroz izbor instrukcija do mašinski zavisnog međukoda i konačno do asemblerskog fajla u okviru debag sekcije. Svaka od navedenih stavki je detaljnije objašnjena u nastavku.

Čuvanje debag lokacija prilikom optimizacije

Kao što je objašnjeno u poglavlju 4.2, optimizacioni prolaz `IROutliner` prilikom izdvajanja instrukcija briše sve podatke o njihovim lokacijama u izvornom kodu. Kako izdvajanje funkcija ostavlja kôd sa manje instrukcija nego sto je bilo pre toga, nije moguće smestiti potrebne informacije u tabelu linija. Jednoj instrukciji bi u tom slučaju moralo da odgovara više mogućih lokacija, što nije dozvoljeno. Kao prvi korak očuvanju ovih podataka ovaj rad uvodi novu unutrašnju (eng. *intrinsic*) funkciju za debagovanje, `llvm.dbg.outlined`, čiji poziv se generiše odmah posle poziva ka izdvojenoj funkciji i dodatno je opisuje odgovarajućom debag lokacijom izdvojene instrukcije.

Unutrašnje funkcije su funkcije koje su ugrađene u sami kompilator takve da on može da ih implementira na optimalan način [30]. One predstavljaju mehanizam proširenja LLVM jezika takav da ne zahteva modifikaciju svih prolaza kada se nešto doda u jezik. Imena svih unutrašnjih funkcija imaju prefiks „`llvm.`” koji će u nastavku biti izostavljen radi čitljivosti. Posebnu klasu unutrašnjih funkcija čine funkcije za

debagovanje. U ovu grupu spadaju `dbg.declare`, `dbg.value` i `dbg.addr` za praćenje promenljivih i `dbg.label` za obeležavanje labela u izvornom kodu. One ne utiču na krajnji izvršni kôd nego se koriste za popunjavanje informacija za debagovanje.

Kao pomoć za identifikovanje izdvojenih instrukcija, uveden je i novi metapodatak `outline_id`. Taj metapodatak će se nalaziti uz svaku izdvojenu instrukciju kao i uz instrukciju poziva izdvojene funkcije. On nema nikakve dodatne podatke nego je umesto toga deklarisan kao jedinstven (eng. *distinct*). To znači da će se za svaku upotrebu kreirati nova instanca umesto da se ponovo koristi postojeća kao što je podrazumevano.

Sada kada postoji način za referisanje instrukcija, moguće je definisati deklaraciju funkcije `dbg.outlined`. Deklaracija ima dva parametra, oba su metapodaci koji pokazuju na `outline_id` identifikatore redom za odgovarajuću izdvojenu instrukciju i instrukciju poziva uz koju stoji. Ovi parametri ne mogu da budu zaključeni implicitno na osnovu mesta na kom se nalaze zbog potencijalnih optimizacija koje mogu da se izvrše kao i promene redosleda instrukcija (npr. u izdvojenoj funkciji). Dodatno, instrukcija sadrži i debag lokaciju koju preuzima od izdvojene instrukcije koju zamenjuje.

Za svaku izdvojenu instrukciju koja ima debag lokaciju se generiše odgovarajuća instrukcija `dbg.outlined` i dodaje se posle instrukcije poziva izdvojene funkcije. Isečak LLVM međukoda programa nakon izdvajanja funkcije je prikazan u listingu 12.

Funkcije generisane izdvajanjem koda je potrebno na neki način razlikovati od drugih funkcija. Trenutno se već u mašinski nezavisnom međukodu dodaje zastavica (eng. *flag*) `DIFlagArtificial`, ali to nije dovoljno da se razlikuje od drugog kompilatorski generisanog koda. Iz tog razloga, dodata je nova vrsta zastavice, `DIFlagOutlined`, koja označava samo izdvojene funkcije. U kasnijoj fazi kompilacije ona će se prevesti u DWARF atribut `DW_AT_LLVM_outlined`. Nova zastavica se dodaje u trenutku generisanja izdvojene funkcije u klasi `IROutliner`, a stara zastavica `DIFlagArtificial` je zadržana radi održanja kompatibilnosti unazad. Pristup ovoj zastavici je omogućen dodavanjem metode `isOutlined` u okviru klase `DISubprogram` koja predstavlja metapodatak sa podacima o funkciji. Ovaj deo implementacije je preuzet iz [33].

```
define dso_local i32 @main() #2 !dbg !30 {
   ...
   call void @outlined_ir_func_0(ptr %x, ptr %y, ptr %c1), !dbg !44,
   ↪  !outline_id !45
   call void @llvm.dbg.outlined(metadata !46, metadata !45), !dbg !44
   call void @llvm.dbg.outlined(metadata !47, metadata !45), !dbg !48
   call void @llvm.dbg.outlined(metadata !49, metadata !45), !dbg !50
   call void @llvm.dbg.outlined(metadata !51, metadata !45), !dbg !43
   call void @llvm.dbg.outlined(metadata !52, metadata !45), !dbg !53
   call void @llvm.dbg.outlined(metadata !54, metadata !45), !dbg !53
   call void @llvm.dbg.outlined(metadata !55, metadata !45), !dbg !53
   call void @llvm.dbg.outlined(metadata !56, metadata !45), !dbg !57
   call void @llvm.dbg.outlined(metadata !58, metadata !45), !dbg !57
   call void @llvm.dbg.outlined(metadata !59, metadata !45), !dbg !57
   call void @llvm.dbg.outlined(metadata !60, metadata !45), !dbg !61
   call void @llvm.dbg.outlined(metadata !62, metadata !45), !dbg !63
   call void @llvm.dbg.outlined(metadata !64, metadata !45), !dbg !65
   ...
}
...
define internal void @outlined_ir_func_0(ptr %0, ptr %1, ptr %2) {
   ...
   %3 = load i32, ptr %0, !outline_id !46
   %4 = load i32, ptr %1, !outline_id !47
   %add = add nsw i32 %3, %4, !outline_id !49
   store i32 %add, ptr %2, !outline_id !51
   %5 = load i32, ptr %2, !outline_id !52
   %dec = add nsw i32 %5, -1, !outline_id !54
   store i32 %dec, ptr %2, !outline_id !55
   %6 = load i32, ptr @global, !outline_id !56
   %add1 = add nsw i32 %6, 2, !outline_id !58
   store i32 %add1, ptr @global, !outline_id !59
   %7 = load i32, ptr %2, !outline_id !60
   %8 = load i32, ptr @global, !outline_id !62
   %call = call i32 @foo(i32 noundef %7, i32 noundef %8), !outline_id !64
   ...
}
```

Listing 12: Primer LLVM međukoda nakon optimizacije izdvajanja koda sa novom instrukcijom i metapodatkom.

Čuvanje debag lokacija prilikom izbora instrukcija

U prethodnoj sekciji je opisana implementacija koja omogućava da potrebne informacije za debagovanje postanu dostupne posle izvršavanja optimizacija na mašinski nezavisnom međukodu. Naredni korak je sprovesti te informacije do sledećeg nivoa, odnosno mašinski zavisnog međukoda. Proces konverzije iz mašinski nezavisnog do mašinski zavisnog međukoda se naziva izbor instrukcija. Postoji više implementacija izbora instrukcija, ali u ovom radu je trenutno podržana samo implementacija `SelectionDAG`.

Od uvedenih podataka za potrebe ovog rada, za instrukciju `dbg.outlined` i vezu između metapodatka `outline_id` i instrukcija za koje su vezane potrebno je implementirati dalje prevođenje. Svi metapodaci ostaju u istom obliku u ovoj fazi kompilacije. Na kraju izbora instrukcija sve instrukcije `dbg.outlined` će biti zamenjene novom MIR instrukcijom `DBG_OUTLINED` sa istim parametrima i debag lokacijom.

Klasa `SelectionDAG` tretira instrukcije za debagovanje drugačije od ostalih instrukcija. One se takođe pretvaraju u čvorove usmerenog acikličnog grafa ali umesto da se ubace u taj graf, one su ostavljene po strani u odvojenim strukturama podataka. Zahvaljujući ovome te instrukcije će ostati neizmenjene prilikom izvršavanja većine faza izbora instrukcija. Po ugledu na druge instrukcije za debagovanje, uveden je tip čvora `SDDbgOutlined` i dodato polje u klasi `SelectionDAG` za njihovo skladištenje. Kako se gradi objekat tipa `SelectionDAG`, sve instrukcije `dbg.outlined` se pretvaraju u instance klase `SDDbgOutlined` i čuvaju u novom polju. Na kraju izbora instrukcija, kada je potrebno poređati instrukcije svaki objekat tipa `SDDbgOutlined` sadrži redni broj koji je dobio prilikom konstrukcije i na osnovu njega se instrukcije `DBG_OUTLINED` emituju na očekivanom mestu u kodu.

Na sličan način kao za čvorove instrukcija za debagovanje, tako se i podaci o metapodatku `outline_id` čuvaju odvojeno od grafa. Konkretno, čuva se u novom polju u okviru promenljive `SDCallSiteDbgInfo`. To je mapa koja preslikava objekat klase `SDNode` u objekat koji sadrži dodatne informacije o čvorovima. Prilikom emitovanja mašinskih instrukcija proverava se postojanje ovog podatka za svaku instrukciju i ukoliko postoji dodeljuje se instrukciji.

Prethodnim izmenama omogućeno je prenošenje svih neophodnih informacija na ovom nivou. Isečak LLVM mašinski zavisnog međukoda programa nakon izdvajanja funkcije je prikazan u listingu 13.

Najveća količina informacija se izgubi prilikom izbora instrukcija. Identifikatori

```
name: main
...
body:
   ...
   CALL64pcrel32 @outlined_ir_func_0, outline-id !45, debug-location !44
   ADJCALLSTACKUP64 0, 0, debug-location !44
   DBG_OUTLINED !46, !45, debug-location !44
   DBG_OUTLINED !47, !45, debug-location !48
   DBG_OUTLINED !49, !45, debug-location !50
   DBG_OUTLINED !51, !45, debug-location !43
   DBG_OUTLINED !52, !45, debug-location !53
   DBG_OUTLINED !54, !45, debug-location !53
   DBG_OUTLINED !55, !45, debug-location !53
   DBG_OUTLINED !56, !45, debug-location !57
   DBG_OUTLINED !58, !45, debug-location !57
   DBG_OUTLINED !59, !45, debug-location !57
   DBG_OUTLINED !60, !45, debug-location !61
   DBG_OUTLINED !62, !45, debug-location !63
   DBG_OUTLINED !64, !45, debug-location !65
   ...
```

Listing 13: Isečak LLVM mašinski zavisnog međukoda dobijen izborom instrukcija koda iz listinga 12.

koji stoje uz instrukcije se mogu izgubiti ako se instrukcija zameni drugom bez kopiranja njenih dodatnih podataka. Na primer, prilikom spuštanja mašinski nezavisne instrukcije poziva funkcije na mašinski zavisan nivo za arhitekturu X86 se pravi potpuno novi čvor. Kako se metapodatak `outline_id` povezuje sa instrukcijom na osnovu čvora klase `SelectionDAG` koji je predstavlja, zamenom tog čvora će se izgubiti ta veza osim ako se ona eksplicitno ne sačuva. Pomenuti primer je popravljen u implementaciji, ali još uvek postoje drugi slučajevi kada će se ovaj problem ispoljiti.

Ispisivanje debag lokacija prilikom emitovanja asemblerskog koda

Poslednja faza kompilacije koju radi direktno kompilator je emitovanje asemblerskog koda. U okviru nje se mašinske instrukcije dobijene izborom instrukcija pretvaraju u odgovarajuće asemblerske instrukcije za tu arhitekturu. U isto vreme se generišu i informacije za debagovanje u izabranom formatu. Sledeće faze kompilacije rade asembler i linker koji će bez problema preneti sve informacije redom do objektnog, odnosno izvršnog fajla. Dakle ovo je poslednji korak koji je potrebno

izmeniti za potrebe ovog rada.

Svi podaci o izdvojenim instrukcijama treba da se pretvore u informacije za debagovanje. Uključivanje informacija za debagovanje ne sme da menja krajnji kôd tako da nijedna instrukcija neće biti dodata ni izmenjena u tekst segmentu programa. Ovaj deo implementacije podržava samo ispisivanje u formatu DWARF. U odeljku 3.2 bilo je reči o mogućnosti proširenja tog formata. Što se tiče kompilatora LLVM, on uvodi svoja proširenja koja se mogu prepoznati po tome što njihovi nazivi sadrže podnisku `LLVM`. Implementacija ovog rada se oslanja na nove vrste etiketa i atributa koji će biti opisani u nastavku.

Kao što je pomenuto na početku poglavlja, novi podaci će biti smešteni u okviru postojeće jedinice tipa `DW_TAG_call_site` koja opisuje poziv funkcije. Za svaku instrukciju `DBG_OUTLINED` u odgovarajućoj jedinici mesta poziva funkcije biće dodata jedinica sa novouvedenom etiketom `DW_TAG_LLVM_outlined_ref`. Ta jedinica sadrži četiri atributa:

- `DW_AT_decl_file` — Identifikator fajla lokacije u izvornom kodu
- `DW_AT_decl_line` — Broj reda u izvornom kodu
- `DW_AT_decl_column` — Broj kolone u izvornom kodu
- `DW_AT_low_pc` — Adresa instrukcije u tekst segmentu programa. U ovom trenutku je predstavljena labelom ispred te instrukcije i biće zamenjena pravom adresom u fazi povezivanja.

Informacije za debagovanje se u kompilatoru LLVM generišu funkciju po funkciju. Međutim, po prirodi ovog zadatka uvek će postojati izdvojena funkcija koju je potrebno obraditi pre nego što je moguće obraditi funkcije koje je pozivaju. LLVM ne pruža mogućnost zadavanja ovakvih zavisnosti, tako da je rešenje pomeriti većinu poslova da se izvrše na samom kraju obrade modula. Obradom svake funkcije se rade potrebne pripreme i čuvaju informacije koje će se na kraju spojiti u konačnu reprezentaciju. Glavna klasa zadužena za upravljanje informacijama za debagovanje na ovom nivou je `DebugHandlerBase`. To je bazna klasa koja radi opšte obrađivanje informacija zajedničko za sve formate za debagovanje. Za sve poslove vezane za konkretan format ona poziva virtuelne funkcije koje implementira odgovarajuća podklasa za taj format. Ovaj rad se fokusira na implementaciju `DwarfDebug`.

Prvi korak prema ispisivanju informacija za debagovanje je prikupljanje instrukcija `DBG_OUTLINED` iz prethodne faze kompilacije. Na početku obrade funkcije instan-

ca klase `DebugHandlerBase` prolazi kroz kôd i pamti sva pojavljivanja unutrašnjih instrukcija za debagovanje. Ta funkcionalnost je proširena da isto to uradi i za instrukcije `DBG_OUTLINED`. Radi lakšeg budućeg pristupa, one se grupišu po njihovom drugom argumentu koji predstavlja metapodatak `outline_id` instrukcije poziva. Svi prikupljeni podaci se smeštaju u polje klase `DebugHandlerBase` tako da je vidljivo i podklasama. Druga stvar koja se radi takođe na početku funkcije je zahtevanje za kreiranje labela ispred izdvojenih instrukcija koje imaju metapodatak `outline_id`. Te labele se dodaju tek prilikom obrađivanja instrukcije ispred koje treba da se smeste, tako da sve potrebne labele postoje tek na kraju obrađivanja cele funkcije.

Celokupnu dalju obradu informacija radi instanca klase `DwarfDebug`. Na kraju obrade svake od funkcija iz korisničkog koda pozvana je funkcija `endFunctionImpl`. Ona prikuplja i emituje sve informacije za debagovanje koje su nastale prilikom obrade te funkcije. Od njih su relevantne:

- DWARF jedinice koje opisuju mesta poziva funkcija.
- Metapodaci `outline_id` koji stoje uz instrukcije poziva za svaku funkciju nastalu izdvajanjem koda.
- Labele postavljene ispred svake izdvojene instrukcije zajedno sa njenim metapodatkom `outline_id`
- Sve ključne informacije iz instrukcija `DBG_OUTLINED` u novoj reprezentaciji specifičnoj za format DWARF, umesto za klasu `MachineInstr`.

Na kraju obrade modula su spremni svi neophodni podaci iz svih funkcija korisničkog koda. Spajanjem tih podataka na osnovu zajedničkih vrednosti metapodatka `outline_id` lako je u postojeće jedinice dodati sve potrebne informacije za debagovanje.

Funkcije za kreiranje novih i modifikovanje postojećih DWARF jedinica nalaze se u klasi `DwarfCompileUnit`. Tu su dodate funkcije za generisanje jedinica sa etiketom `DW_TAG_LLVM_outlined_ref` sa ranije opisanim poljima i dodata je pomoćna funkcija za njihovo ubacivanje u jedinicu tipa `DW_TAG_call_site`. Uz to, toj jedinici je dodat atribut `DW_AT_LLVM_outlined` koji označava da je to poziv izdvojene funkcije. Isti atribut se dodaje i jedinici te funkcije, a ovde je dodat kako bi se kasnije brže odredilo da li se poziva izdvojena funkcija ili ne.

Za potrebe emitovanja instrukcija u tekst segment, u funkciju `emitFunctionBody` klase `AsmPrinter` je dodat slučaj za ispisivanje instrukcije `DBG_OUTLINED` sa pra-

znom implementacijom. Promenom ove implementacije da ispiše komentar može se potvrditi da su instrukcije `DBG_OUTLINED` još uvek prisutne i da se nalaze na očekivanom mestu u kodu.

Prethodnim modifikacijama omogućeno je prevođenje do asemblerskog fajla. Asembler nema predstavu o formatu ili informacijama za debagovanje i ne postoje posebne asemblerske direktive za njih. Umesto toga, sekcije za debagovanje se u asemblerskom kodu popunjavaju literalima koji imaju redosled i veličinu propisane formatom. Ti literali imaju smisla samo kada se parsiraju u istom formatu u kom su zapisani. Jedan alat koji to može da uradi je `llvm-dwarfdump`. Listing 14 prikazuje deo novih etiketa parsiranih iz objektnog fajla pomoću alata `llvm-dwarfdump`. Kako bi imao podršku za prikazivanje ovih etiketa potrebno je koristiti verziju alata koja je sagrađena uz ovaj projekat.

```
0x00000093: DW_TAG_call_site
               DW_AT_call_origin    (0x000000dc)
               DW_AT_call_return_pc (0x0000000000001179)
               DW_AT_LLVM_outlined  (true)

0x00000099: DW_TAG_LLVM_outlined_ref
               DW_AT_decl_file      ("test.c")
               DW_AT_decl_line      (11)
               DW_AT_decl_column    (0x0b)
               DW_AT_low_pc         (0x0000000000001194)

0x0000009e: DW_TAG_LLVM_outlined_ref
               DW_AT_decl_file      ("test.c")
               DW_AT_decl_line      (11)
               DW_AT_decl_column    (0x0f)
               DW_AT_low_pc         (0x0000000000001196)
```

Listing 14: Isečak DWARF stabla koji prikazuje etiketu poziva funkcije i njenu decu sa podacima o izdvojenim instrukcijama

5.2 Implementacija u debageru

Programi prevedeni kompilatorom sa izmenama iz prethodnog poglavlja uz neophodne opcije će sadržati debag informacije o izdvojenom kodu. Pošto implementacija predstavlja novo proširenje standarda DWARF, nijedan postojeći debager

nema podršku za interpretiranje tih informacija. U ovom poglavlju će biti predstavljena implementirana podrška za uvedeno proširenje u debageru `LLDB`. Prvo će biti omogućeno učitavanje novih podataka, a zatim rešeni najveći nedostaci trenutne podrške za debagovanje izdvojenih instrukcija. To uključuje ispisivanje tačnih mesta u izvornom kodu kada se proces zaustavi, podršku za izvršavanje korak po korak i postavljanje tačaka prekida.

Učitavanje podataka

Za rešavanje opisanih problema potreban je način za mapiranje adrese izvršnog fajla na lokaciju u izvornom kodu i obrnuto. Obično oba ova mapiranja radi tabela linija, ali za izdvojene instrukcije ta dva smera su nezavisno implementirana na različite načine zato što su im potrebni različiti podaci od kojih nisu svi dostupni u vreme kompilacije.

Interfejs za učitavanje i pristup informacijama za debagovanje u debageru LLDB je ostvaren kroz klasu `SymbolFile`. On pruža funkcije za parsiranje podataka iz različitih sekcija za debagovanje i njihovo dohvatanje u reprezentaciji nezavisnoj od formata u kome su napisane. Većina podataka se ne učitava odmah po pokretanju debagera već tek kada su ti podaci neophodni. Tako na primer dohvatanjem podataka za jedinicu kompilacije kroz funkciju `GetCompileUnitAtIndex` se izvršava parsiranje te jedinice ukoliko već nije učitana. Za tabelu linija, odgovarajuća funkcija za učitavanje se intuitivno zove `ParseLineTable`, ali joj se pristupa pomoću funkcije `GetLineTable` klase `CompileUnit`.

Podržani formati informacija za debagovanje su implementirani kao priključci. Svaki od njih implementira interfejs `SymbolFile` ili preciznije apstraktnu klasu `SymbolFileCommon`. U slučaju formata DWARF, implementacija se nalazi u klasi `SymbolFileDWARF`.

Mapiranje lokacije u izvornom kodu na adresu izdvojene instrukcije se može uraditi na isti način kao što radi tabela linija za obične instrukcije. Zbog toga se ovaj deo implementacije ugleda na nju. Uvedena je nova tabela linija specijalno za lokacije izdvojenih instrukcija. Podaci korišćeni u toj tabeli se dobijaju obilaskom čitavog DWARF stabla i parsiranjem atributa svih jedinica tipa `DW_TAG_LLVM_outlined_ref`. Nakon učitavanja, nova tabela linija se predaje odgovarajućem objektu klase `CompileUnit` preko koga može da joj se pristupi iz skoro celog koda.

Dohvatanje adrese instrukcije za prosleđenu lokaciju se koristi u okviru određivanja konteksta. U poglavlju 3.3 je rečeno da je kontekst opisan kroz instancu klase `SymbolContext` koja te podatke dobija iz informacija za debagovanje. Kontekst između ostalog podrazumeva jedinicu kompilacije, funkciju i lokaciju u izvornom kodu. Bitno je primetiti da lokacija dobijena iz konteksta ne mora da se poklapa sa prosleđenom lokacijom, već može da se dobije najbliža lokacija za koju postoje informacije.

Za određivanje konteksta na osnovu lokacije zadužena je klasa `CompileUnit`, a funkcija koja to obavlja je `ResolveSymbolContext`. Ona radi tako što pokušava da odredi adresu instrukcije koja ima lokaciju najbližu prosleđenoj. Pored postojeće provere najbliže lokacije iz tabele linija, dodata je ista provera za izdvojenu tabelu linija. Ukoliko u obe tabele postoje lokacije, u kontekst se stavlja ona koja je bliža prosleđenoj. Ako samo jedna od tabela ima validan rezulat, koristiće se on. Ostatak konteksta se određuje preko adrese koja odgovara izabranoj lokaciji. Ovo omogućava svima koji koriste ovaj način dohvatanja konteksta mogu da dobiju informacije o izdvojenim instrukcijama. Upotreba ove izmene za rešenje konkretnog problema će biti prikazana kasnije.

Mapiranje u drugom smeru, sa adrese izdvojene instrukcije na lokaciju u izvornom kodu nije jedinstveno. Činjenica da je instrukcija izdvojena znači da se poziva na bar dva različita mesta u izvornom kodu. Zbog toga jednoj adresi će uvek odgovarati bar dve lokacije. Kako bi se dobila tačna lokacija potrebno je uzeti obzir i kontekst izvršavanja programa. Taj kontekst je predstavljen stek okvirom funkcije koja je pozvala izdvojenu funkciju. Preciznije, predstavljen je adresom povratka (eng. *return address*) na funkciju kojoj odgovara taj stek okvir. Adresa povratka je adresa sledeće instrukcije koja će biti izvršena nakon povratka iz funkcije.

Podaci o mapiranju u ovom smeru su smešteni u instancama klase `CallEdge`. Neka su funkcije čvorovi usmerenog grafa i neka grana između dva čvora postoji za svaki poziv funkcije u smeru od pozivaoca do pozvane funkcije. Može postojati više grana između dve funkcije ukoliko se ista funkcija poziva više puta. Klasom `CallEdge` predstavlja se jedna od tih grana. Ona je dopunjena da sadrži adrese i lokacije svih izdvojenih instrukcija prilikom tog poziva. Prethodno pomenuta klasa `SymbolFile` je zadužena za popunjavanje tih podataka kada su potrebni. Ta implementacija je smeštena u funkciji `CollectCallEdges` i zasniva se na parsiranju jedinice poziva funkcije. Svim instancama klase `CallEdge` je moguće pristupiti kroz objekat tipa `Function` pozivaoca, ili može da se pristupi konkretnoj instanci na osnovu adrese

povratka.

Poslednji podatak koji je potrebno učitati je atribut `DW_AT_LLVM_outlined` koji obeležava izdvojene funkcije. To je implementirano u okviru postojeće implementacije za učitavanje podataka o funkciji u klasi `DWARFASTParserClang`. Pristup informaciji da li je funkcija dobijena izdvajanjem koda je omogućen kroz metod `IsOutlined` dostupan nad objektima klase `Function` koja opisuje jednu funkciju iz koda koji se debaguje.

Ispisivanje ispravnih linija izvornog koda

Prilikom debagovanja korisnik može da kontroliše tok izvršavanja programa. On zadaje komande koje govore debageru koje delove koda da izvrši. Svaki put kada se kontrola vrati korisniku, debager ispiše mesto gde se program zaustavio (i možda nekoliko okolnih linija). Na osnovu toga korisnik saznaje informacije o trenutnom stanju programa i bira šta će sledeće da izvrši.

U prisustvu dobrih debag lokacija prethodni proces funkcioniše očekivano. Međutim u slučaju izvojenih instrukcija koje nemaju vrednost u tabeli linija, lokacija prikazana u debageru će biti netačna. Prikazana lokacija će pripadati prvoj ranijoj instrukciji koja ima lokaciju. Kako se izdvojene funkcije dodaju na adresama iza svih korisničkih funkcija, prva ranija instrukcija može da pripada nekoj drugoj funkciji, ili čak može da joj odgovara lokacija koja označava kompilatorski generisan kôd. Korisnik ne dobija nikakvo obaveštenje o tome šta se desilo, već samo može da vidi da je program napravio veliki skok na kraj fajla iako to nije slučaj.

Prvi izazov koji je potrebno rešiti kako bi imalo smisla rešavati bilo koji drugi problem je kako odrediti tačnu lokaciju u izvornom kodu kada se debager zaustavi u izdvojenoj funkciji. Bez toga iako se uspešno implementira bilo koji način zaustavljanja na izdvojene instrukcije, debager bi prikazivao netačne lokacije što samo otežava debagovanje. Trenutno jedini načini da se zastane na izdvojenoj instrukciji je da se program izvršava instrukciju po instrukciju ili da se postavi tačka prekida po adresi. Oba načina se mogu koristiti za proveru ispravnosti implementacije koja sledi.

Ispisivanje statusne poruke, koja između ostalog uključuje i okolne redove gde je program zaustavljen radi funkcija `GetStatus` klase `StackFrame`. Taj deo koda je izmenjen na sledeći način. Prvo se proverava da li trenutni stek okvir pripada izdvojenoj funkciji. Podatak koji to opisuje je dostupan kroz metod `IsOutlined` klase `Function`, a ona se dobija na osnovu konteksta tog stek okvira. Ukoliko stek

okvir ne pripada izdvojenoj funkciji, određivanje linije se svodi na postojeću implementaciju. U suprotnom se koristi nova implementacija zasnovana na sličnoj ideji kao određivanje vrednosti promenljivih prilikom ulaska u funkciju [31].

Ovaj problem se svodi na određivanje linije u izvornom kodu koristeći adresu instrukcije na kojoj je proces zaustavljen. Podrška za mapiranje u tom smeru je opisana u prethodnoj sekciji. Jedino što je potrebno je proslediti kontekst kako bi se odredila ispravna instanca klase `CallEdge`. Klasa `Function` već ima implementiran mehanizam za pronalaženje te instance na osnovu adrese povratka iz funkcije. Ta adresa se postavlja na vrh steka u trenutku poziva funkcije i lako se dohvata povratkom u rediteljski stek okvir koji pripada pozivaocu. Konačna lokacija je dobijena iz instance klase `CallEdge` iz tog roditeljskog stek okvira pomoću metode `Function::GetCallEdgeForReturnAddress` sa pronađenom adresom povratka kao parametrom. Uz lokaciju, u status je dodata i poruka da je proces zaustavljen u izdvojenom kodu.

Izvršavanje korak po korak

Jedna od najkorišćenijih tehnika debagovanja je izvršavanje korak po korak. Ona omogućava korisniku da interaktivno prati i kontroliše izvršavanje programa. Između dva koraka korisnik može da pogleda vrednosti relevantnih promenljivih i proveri da li se one poklapaju sa očekivanim vrednostima. Debager LLDB nudi nekoliko komandi za koračanje:

- `step-over` — Izvršava sve naredbe do sledećeg reda u izvornom kodu preskačući pozvane funkcije
- `step-into` — Izvršava sve naredbe do sledećeg reda u izvornom kodu pritom ulazeći u pozvane funkcije
- `step-inst` — Izvršava jednu instrukciju izvršnog fajla pritom ulazeći u pozvane funkcije
- `step-inst-over` — Izvršava jednu instrukciju izvršnog fajla bez ulaska u pozvane funkcije

U postojećoj implementaciji jedino komandama `step-inst` i `step-inst-over` je moguće stati na izdvojenu instrukciju. Druge komande, kako se oslanjaju na broj

reda u izvornom kodu, će ih uvek preskočiti. Cilj ovog odeljka je prepraviti i njih da rade u skladu sa korisnikovim očekivanjima.

Tokom sesije debagovanja, korsnik zadaje komande debageru kako bi kontrolisao tok izvršavanja programa. Celokupna logika o tome koliko program treba da se izvršava i kada treba da se zaustvi implementirana je pomoću planova niti (eng. *thread plan*). Više različitih planova može da bude aktivno u isto vreme, zbog čega se oni skladište u stek strukturi podataka. U nastavku se smatra da taj stek raste naviše.

Svaka nit ima svoj stek planova. Kada se program zaustavi zbog nekog događaja svi planovi za tu nit se proveravaju redom od vrha do dna steka. Traži se prvi plan koji može da objasni događaj koji je izazvao zaustavljanje pozivanjem funkcije `DoPlanExplainsStop` svakog plana. Taj plan se onda upita da li treba da se vrati kontrola korisniku funkcijom `ShouldStop`. Ovde se ažurira logika ukoliko je potrebno da se program zaustavi na nekom sledećem mestu. Posle izvršavanja plana proverava se da li se on završio pozivom funkcije `MischiefManaged`. Ukoliko jeste, on i svi planovi iznad njega u steku su uklonjeni ukoliko prijave da više nisu relevantni. Uvek postoji bar jedan plan na steku. To je instanca `ThreadPlanBase` i on se aktivira kada ni jedan drugi plan ne objašnjava događaj kao na primer za tačke prekida napravljene od strane korisnika.

Postoji više implementacija planova niti koji imaju različite kriterijume za zaustavljanje. Fokus ovog rada su implementacije u klasama `ThreadPlanStepInRange` i `ThreadPlanStepOverRange` koje se koriste redom za rad komandi `step-into` i `step-over`. Oni dele zajedničku nadklasu `ThreadPlanStepRange`. Oba plana rade tako što postavljaju tačke prekida na sledeću intrukciju grananja sve dok program ne izađe iz opsega koji mu je zadat u vreme kreiranja. Taj opseg obuhvata sve instrukcije koje pipadaju istom redu u izvornom kodu. Razlog zašto nije moguće odmah postaviti tačku prekida direktno na konačnu lokaciju je zbog mogućih grananja u programu. Razlika između pomenutih planova je što komanda `step-into` prilikom zaustavljanja na instrukciji poziva funkcije dodaje novi plan za ulazak u funkciju, dok komanda `stop-over` to ne radi.

Ideja rešenja je da se osim postojeće tačke prekida uvede još jedna koja se uvek postavlja na sledeću izdvojenu instrukciju. Nadklasa `ThreadPlanStepRange` je dopunjena funkcijama za postavljanje te nove tačke prekida koja se dobija na isti način kao u poglavlju 5.2. Posebno u slučaju komande `step-over` sledeća instrukcija grananja se traži uz bezuslovno preskakanje instrukcija poziva. Taj postupak je izmenjen dodavanjem uslova da pozvana funkcija nije nastala izdvajanjem koda.

Ukoliko jeste, očekivano ponašanje je da debager ipak korača unutar te funkcije kako to odgovara izvornom kodu programa. U suprotnom poziv instrukcije se ignoriše kao i ranije.

Poslednje što je potrebno učiniti je izmeniti konkretne klase planova da dodaju i proveravaju tačku prekida na izdvojenoj instrukciji. Razlikuju se dva slučaja u kojima je potrebno dodati ciljnu tačku prekida. Ulaskom u izdvojenu funkciju potrebno je postaviti sledeći cilj da bude prva izdvojena instrukcija sa lokacijom. Ukoliko je program već zaustavljen usred izdvojene funkcije onda cilj postaje prva sledeća izdvojena instrukcija sa lokacijom u narednom redu. Prethodni slučajevi se lako razlikuju poređenjem odnosa između trenutnog stek okvira i okvira iz trenutka kreiranja plana. Kako bi se plan uopšte aktivirao prilikom dostizanja cilja, ta provera je dodata u funkciju `DoPlanExplainsStop`. Obrada samog događaja se radi na početku funkcije `ShouldStop` gde se odmah vraća kontrola korisniku i plan se završava.

Postavljanje tačaka prekida

Tačka prekida je mehanizam u debageru za pauziranje izvršavanja procesa i vraćanje kontrole korisniku kada proces dostigne mesto u kodu gde je ta tačka postavljena. Debager LLDB podržava nekoliko načina za postavljanje tačaka prekida:

- pomoću adrese u izvršnom fajlu — Najjednostavniji vid postavljanja tačke prekida. Postavlja se tačno na prosleđenu adresu.
- pomoću imena funkcije — U izvornom kodu se pronalaze sve funkcije sa prosleđenim imenom i za svaku od njih dodaje se nova tačka prekida.
- pomoću imena fajla i reda — Prosleđuje se naziv fajla i red u tom fajlu. Na osnovu informacija za debagovanje se tačka prekida postavlja na prvu instrukciju kojoj odgovara prosleđeni red ili prvi sledeći red ukoliko za traženi red ne postoje informacije.
- pomoću regularnog izraza — Izvorni kôd se pretražuje regularnim izrazom i svako poklapanje dodaje novu tačku prekida.

Svi ovi načini se svode na postavljanje preko jedne ili više adresa.

Za potrebe razlikovanja, svakoj napravljenoj tački prekida dodeljuje se jedinstveni identifikator u formatu `X.Y` gde `X` označava identifikator grupe, a `Y` označava

identifikator lokacije u toj grupi. Sve tačke prekida nastale istom komandom imaju istu vrednost `X` i različite vrednosti `Y`. Ne može da se desi da dve tačke prekida iz iste grupe imaju istu lokaciju. Ovaj identifikator se koristi za izmenu svojstava tačaka prekida i prikazuje se u okviru poruke korisniku kada se program zaustavi usled dostizanja jedne od njih. Primer takve poruke je prikazan u listingu 15.

```
* thread #1, name = 'test', stop reason = breakpoint 1.1
```

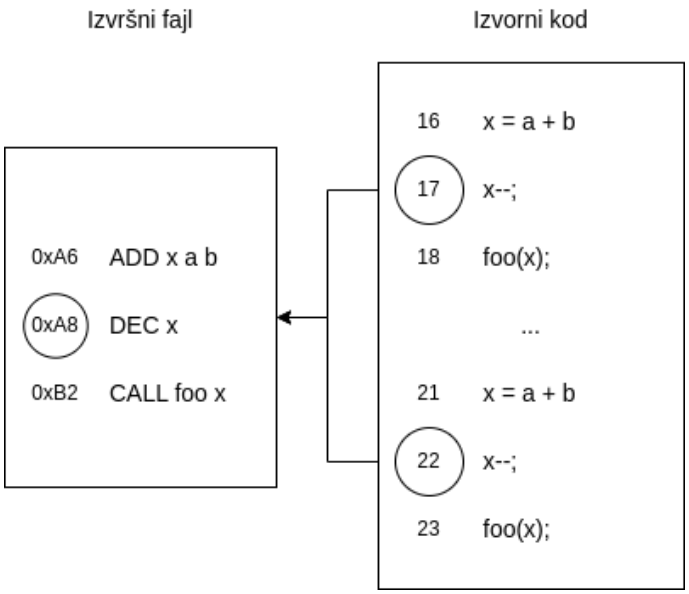
Listing 15: Primer poruke koju prikazuje debager LLDB prilikom zaustavljanja na tački prekida

Prva dva od prethodno opisanih načina rade očekivano za izdvojene instrukcije dok druga dva ne funkcionišu. Ono što izdvaja druga dva načina je što oni moraju da odrede adresu na osnovu lokacije u izvornom kodu. Tačno je da se isti ovaj proces koristi i za postavljanje tačke prekida po imenu funkcije ali kako izdvojeni kôd ne može da sadrži deklaraciju funkcije, dotični problem ne važi u ovom slučaju.

Potreban mehanizam za rešenje ovog problema je već ostvaren dodatnom tabelom linija za izdvojene instrukcije. Ona omogućuje mapiranje lokacija u izvornom kodu na adrese što je baš potrebno u ovom slučaju. Međutim i sa tim se javlja drugi problem. Sada prilikom debagovanja dolazi do neočekivanih zastajkivanja na tačke prekida koje korisnik nije postavio. Radi jednostavnosti, nastavak teksta se odnosi na slučaj zadavanja fajla i reda. Problem je u tome što odgovarajuća adresa pripada izdvojenoj funkciji koja se poziva iz više mesta.

Ilustracija ovog problema je prikazana na slici 5.1. Ukoliko korisnik postavi tačku prekida na red 17, do prekida će doći i prilikom izvršavanja linije 22 i obrnuto. Korisnik ne treba da bude svestan postojanja izdvojene funkcije, jer on debaguje samo na osnovu izvornog koda. Za njega, očekivani ishod treba da bude da se program zaustavlja samo na redu koji je on izabrao.

Rešenje problema nije moguće realizovati u vreme postavljanja tačke prekida, već se provera ispravnosti lokacije radi nakon izazivanja prekida, ali pre povratka kontrole korisniku. Dok klasa `Breakpoint` poseduje sve podatke o grupi tačaka prekida koje su nastale istom komandom, informacije o svakoj pojedinačnoj instanci se nalaze u okviru klase `BreakpointLocation`. Ona je dopunjena poljem koje predstavlja opcionu lokaciju u izvornom kodu na osnovu koje je napravljena i poljem logičkog tipa koje obeležava da li je instrukcija izdvojena. Oba dodata polja dobijaju vrednost prilikom kreiranja tačke prekida na osnovu podataka iz tabela linija.



Slika 5.1: Vizuelni prikaz odnosa izdvojenog koda u izvornom kodu i izvršnom fajl

Dolaskom na tačku prekida u toku izvršavanja radi se provera da li treba vratiti kontrolu korisniku kroz funkciju `ShouldStop` klase `BreakpointLocation`. U ovom trenutku je potrebno proveriti da li se lokacija sačuvana u polju poklapa sa lokacijom koja odgovara trenutnom stanju procesa. Način dobijanja trenutne lokacije je isti kao i u prethodna dva poglavlja, uz pomoć trenutne adrese i konteksta. Ukoliko se lokacije poklapaju, proces će se zaustaviti, a u suprotnom će se prekid ignorisati. Posebna pažnja je data da se ova provera radi pre bilo koje druge operacije koja može da izmeni stanje tačke prekida, na primer uvećanja njenog brojača aktivacije.

Glava 6

Rezultati

Modifikovanu verziju kompilatora i debagera je potrebno detaljno testirati u različitim mogućim situacijama kako bi se postarali da se prvenstveno ne smanjuje funkcionalnost u odnosu na nemodifikovanu verziju, a zatim i da se zapravo unapređuje debagovanje izdvojenog koda. Kako je velika količina koda izmenjena, velike su šanse da je došlo do propusta u implementaciji koji nisu detektovani testiranjem na jednostavnim primerima u toku prvobitnog pisanja koda. To će biti provereno korišćenjem testova.

Projekat LLVM već uključuje veliki broj regresionih testova koji pokrivaju skoro svaku podržanu funkcionalnost. Ti testovi se pokreću koristeći pomoćni alat `lit` (skraćeno za *LLVM Integrated Tester*) i sastoje se od fajlova koji sadrže kôd koji se testira i ugrađene direktive u vidu komentara. Svaki test sadrži bar jednu direktivu za izvršavanje komande. Ona je označena niskom `RUN:` posle koje sledi komanda koja će biti izvršena. Ta komanda određuje rezultat testa. Ukoliko se ona izvrši uspešno (sa rezultatom 0) onda je test uspešan, u suprotnom je neuspešan. Komanda se uglavnom sastoji od pokretanja nekog od alata infrastrukture LLVM čiji rezultat se prenosi u alat `FileCheck` koji omogućava proveru u odnosu na očekivane vrednosti koje su zapisane pomoću `CHECK` direktiva u test fajlu. Ukoliko se za kompilaciju projekta LLVM koristio alat `Ninja`, pokretanje regresionih testova se radi komandom `ninja check-llvm`.

Pokrenuti su svi testovi uključeni u LLVM projekat. Većina testova je uspešno izvršena bez problema. To je bilo i očekivano zato što veliki broj testova i ne radi sa uključenim izdvajanjem koda. Mali deo testova je neuspešno izvršen zato što proverava specifičan redosled informacija za debagovanje koje su izmenjene modifikovanim kompilatorom. U tim situacijama, odgovarajući testovi su izmenjeni tako

da očekuju pojavljivanje novih etiketa i atributa. Poslednji, poseban slučaj je činio test koji sadrži istovremeno izdvojen i umetnut kôd. Ova situacija nije bila predviđena prilikom pisanja implementacije. Dotični test se prevremeno zaustavljao zbog neispunjene tvrdnje da poziv funkcije koja može da bude umetnuta unutar funkcije sa informacijama za debagovanje mora da ima lokaciju. Za taj slučaj je dodata posebna podrška. Pozivima koji ispunjavaju opisani uslov je postavljena veštačka debag lokacija. Posle ispravke poslednjeg testa nije uočeno više problema i svi testovi su se uspešno izvršavali.

Sledeće je potrebno isprobati da li implementacija uspešno rešava probleme koje je namenjena da reši. Demonstracija će biti prikazana na istom primeru koji je predstavljen ranije u listingu 4. Ponašanje debagera na izdvojenom kodu pre modifikacija je opisano u poglavlju 4.2.

Kompilacijom sa modifikovanim kompilatorom i proverom informacija za debagovanje izvršnog fajla se može se primetiti da su nove etikete generisane na mestima poziva funkcija i da se adrese poklapaju sa adresama instrukcija u izdvojenoj funkciji. Pokretanjem debagera na generisanom fajlu i koračanjem po programu dobija se ispis prikazan na slici 16. Ukoliko se ovaj ispis uporedi sa ispisom pre modifikacija primećuje se da se program zaustavlja na 3 od 4 izdvojene instrukcije što prethodno nije bio slučaj. Takođe, svaki put kad se zastane ispisuje se poruka koja govori da se proces nalazi u izdvojenoj funkciji. Četvrta naredba, `c1--;`, ne izaziva prekid izvršavanja programa. Istraživanjem je otkriveno da se odgovarajuća instrukcija koja je u početku bila instrukcija sabiranja sa konstantom -1, u toku izbora instrukcija zamenila sa instrukcijom dekrementiranja za 1. Zbog te zamene izgubljen je metapodatak `outline_id` koji je ta instrukcija originalno imala, stoga nije bilo moguće pronaći adresu te instrukcije prilikom emitovanja asemblerskog koda. Ovo je još jedna instanca problema već opisanog u poglavlju 5.1. Za njegovo rešavanje potrebno je ili ručno popraviti sve slučajeve kada se ovo dešava ili osmisliti mehanizam za automatsko prenošenje metapodataka `outline_id` kada se zameni čvor.

Još jedna implementirana funkcionalnost koju treba proveriti je postavljanje tačaka prekida. U prethodnom primeru je izgledalo kao da se uspešno postavila tačka prekida na liniji 11 koja je izdvojena. Ono što se zapravo desilo je da se tačka postavila na instrukciju poziva izdvojene funkcije koja je dobila lokaciju prve izdvojene instrukcije. Tek sledećim korakom se zapravo došlo do izdvojene instrukcije. Umesto te naredbe na redu 11, tačka će biti postavljena na red 13 kako bi se izbegla prethodna situacija. Ispis debagera je prikazan na slici 17. Komanda za postavljanje

```
$ lldb outline
(lldb) breakpoint set --line 11
(lldb) run
    9           int y = 1;
    10
 -> 11          int c1 = x + y;
    12          c1--;
    13          global += 2;
(lldb) next
    9           int y = 1;
    10
 -> 11          int c1 = x + y;
    12          c1--;
    13          global += 2;
Note: this function is outlined.
(lldb) next
    11          int c1 = x + y;
    12          c1--;
 -> 13          global += 2;
    14          foo(c1, global);
    15
Note: this function is outlined.
(lldb) next
    12          c1--;
    13          global += 2;
 -> 14          foo(c1, global);
    15
    16          int c2 = x + y;
Note: this function is outlined.
```

Listing 16: Koračanje po izdvojenom kodu sa izmenjenim debagerom

tačke prekida je uspešno odredila adresu koja odgovara zadatoj liniji i pokretanjem programa, proces je zaustavljen na toj adresi.

Novi regresioni testovi su dodati za kompilator i debager po ugledu na prethodni primer. U kompilatoru se testira uspešno generisanje informacija za debagovanje u svakom koraku kompilacije. Debager se testira pokretanjem nad programom sa izdvojenim kodom i automatskim zadavanjem komandi koje proveraju ispravnost lokacije, da li se program zaustvio na očekivanim mestima i da li je uspešno postavljena tačka prekida.

```
$ lldb outline
(lldb) breakpoint set --line 13
Breakpoint 1: where = outline`outlined_ir_func_0 + 11 at outline.c,
↪  address = 0x00005555555551a0
(lldb) run
    10
    11                  int c1 = x + y;
    12                  c1--;
 -> 13                  global += 2;
    14                  foo(c1, global);
    15
    16          int c2 = x + y;
Note: this function is outlined.
```

Listing 17: Postavljanje tačke prekida na izdvojenu naredbu sa izmenjenim debagerom

Glava 7

Zaključak

U okviru ovog rada su opisani proces kompilacije sa akcentom na optimizacijama koda i postupak otklanjanja grešaka u softveru pomoću debagera. Detaljno je objašnjena uloga informacija za debagovanje i problem održavanja tih informacija prilikom izvršavanja kompilatorskih optimizacija. Za potrebe kompilacije i debagovanja korišćeni su alati infrastrukture LLVM, konkretno kompilator LLVM i debager LLDB. Glavni predmet rada je optimizacija izdvajanja koda koja štedi memoriju programa tako što pomera ponovljene delove koda u zasebnu funkciju i uklanja duplikate. To smanjenje krajnjeg koda predstavlja problem očuvanju informacija za debagovanje i dovodi do problema prilikom upotrebe debagera.

Doprinos ove teze je poboljšanje očuvanja informacija za debagovanje prilikom optimizacije izdvajanja koda u okviru kompilatora LLVM. One su sačuvane prilikom primene optimizacije, prenete kroz sve faze kompilacije i upisane u sekciju za debagovanje izvršnog fajla. Na osnovu toga poboljšan je i debager LLDB tako što mu je dodata mogućnost da iskoristi te informacije kako bi se ostvarilo bolje iskustvo debagovanja. Dodata je podrška za ispisivanje ispravnih lokacija iz izvornog koda, koračanje i postavljanje tačaka prekida na izdvojenom kodu.

Postoji nekoliko pravaca daljeg razvoja. Predloženo rešenje je moguće implementirati u drugim kompilatorima i debagerima kao što su *GCC* i *GDB*. Što više alata podržava ovo rešenje, veći broj korisnika će uživati u pogodnostima koje ono pruža. Pored toga, moguća su poboljšanja ove implementacije u okviru infrastrukture LLVM. U poglavlju 5.1 je pomenut problem sa održavanjem metapodatka `outline_id` prilikom izbora instrukcija. Njegovim rešavanjem, bilo u opštem ili pojedinačnim slučajevima, dodatno bi se smanjila količina preskočenih naredbi prilikom koračanja u debageru. Zatim, trenutno postoji podrška za čuvanje debag lokacija

samo iz prolaza `IROutliner`. Kako se već svakako te informacije spuštaju do mašinski zavisnog međukoda na putu do izvršnog fajla, moguće je dodati podršku i za prolaz `MachineOutliner`. Pored toga, trenutna implementacija funkcioniše samo za implementaciju pristupa `SelectionDAG` za izbor instrukcija. Iako je to dovoljno za sada, u budućnosti kada implementacija `GlobalIsel` bude prešla u širu upotrebu biće potrebno podržati i nju. Na kraju, jedna posledica nastala opisanim izmenama je što pregled stek okvira dok je proces zaustavljen u izdvojenoj funkciji prikazuje stek okvir i te izdvojene funkcije. Kako ona ne postoji u izvornom kodu, korisniku ne treba da bude prikazana prilikom debagovanja. Suprotna stvar se dešava kada je proces zaustavljen u umetnutom kodu, gde se konstruiše veštački stek okvir koji ne postoji u toku izvršavanja, ali korisnik očekuje da bude prisutan.

Bibliografija

[1] Alfred V. Aho, Monica S. Lam, Ravo Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 2nd edition edition, 2006.

[2] O. Ben-Kiki, C. Evans, and I. döt Net. YAML Ain't Markup Language (YAML™) version 1.2, 2009. on-line at: `https://yaml.org/spec/1.2.1/`.

[3] Florent Bouchez, Alain Darte, Christophe Guillon, and Fabrice Rastello. Register allocation: What does the np-completeness proof of chaitin et al. really prove? or revisiting register allocation: Why and how. In George Almási, Călin Caşcaval, and Peng Wu, editors, *Languages and Compilers for Parallel Computing*, pages 283–298, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[4] Amy Brown and Greg Wilson. *The Architecture of Open Source Applications: Elegance, Evolution, and a Few Fearless Hacks*. The Achrictecture of Open Source Applications. Creative Commons, 2011.

[5] TIS Committee. Executable and Linking Format (ELF) Specification. Technical report, 1995. Version 1.2.

[6] The Qt Company. Qt Creator. on-line at: `https://github.com/qt-creator/qt-creator`.

[7] K.D. Cooper and L. Torczon. *Engineering a Compiler*. Elsevier Science, 2022.

[8] DWARF Debugging Information Format Committee. *DWARF Debugging Information Format Version 5*.

[9] Michael J. Eager. Introduction to the DWARF Debugging Format, 2012.

[10] Eclipse Foundation. Eclipse IDE. on-line at: `https://eclipseide.org/`.

[11] Free Software Foundation. GNU gcc, 2013. on-line at: `http://gcc.gnu.org/`.

[12] Free Software Foundation. GDB: The GNU Project Debugger, 2023. on-line at: `https://www.sourceware.org/gdb/`.

[13] The LLVM Foundation. LLVM Language Reference Manual. on-line at: `https://llvm.org/docs/LangRef.html`.

[14] The LLVM Foundation. Machine IR (MIR) Format Reference Manual. on-line at: `https://www.llvm.org/docs/MIRLangRef.html`.

[15] The LLVM Foundation. The LLDB Debugger. on-line at: `https://lldb.llvm.org/`.

[16] The LLVM Foundation. The LLVM Compiler Infrastructure. on-line at: `https://llvm.org/`.

[17] E. Gamma, R. Helm, R.E. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley professional computing series. Addison-Wesley, 1995.

[18] Dick Grune, Kees van Reeuwijk, Henri E. Bal, Ceriel J.H. Jacobs, and Koen Langendoen. *Modern Compiler Design*. Springer Publishing Company, Incorporated, 2nd edition, 2012.

[19] Jakub Jelinek. Representation of call sites in the debugging information, 2010.

[20] Jakub Jelinek. Tracking of values passed as arguments to functions in debug information, 2010.

[21] JetBrains. CLion. on-line at: `https://www.jetbrains.com/clion/`.

[22] Andrew Litteken. Framework for Finding and Using Similarity at the IR Level, 2020. on-line at: `https://lists.llvm.org/pipermail/llvm-dev/2020-September/144779.html`.

[23] Bruno Cardoso Lopes and Rafael Auler. *Getting Started with LLVM Core Libraries*. Packt Publishing, 2014.

[24] Microsoft. Visual Studio Code. on-line at: `https://code.visualstudio.com/`.

[25] Microsoft. First look at the Visual Studio Debugger, 2022. on-line at: `https://learn.microsoft.com/en-us/visualstudio/debugger/debugger-feature-tour`.

[26] K. Nacke. *Learn LLVM 12: A Beginner's Guide to Learning LLVM Compiler Tools and Core Libraries with C++*. Packt Publishing, Limited, 2021.

[27] Mayur Pandey and Suyog Sarda. *LLVM cookbook: Over 80 engaging recipes that will help you build a compiler frontend, optimizer, and code generator using LLVM*. PACKT, 2017.

[28] Jessica Paquette. Interprocedural MIR-level outlining pass, 2016. on-line at: `https://lists.llvm.org/pipermail/llvm-dev/2016-August/104170.html`.

[29] F. Rastello and F.B. Tichadou. *SSA-based Compiler Design*. Springer International Publishing, 2022.

[30] S. Sarda and M. Pandey. *LLVM Essentials*. Community experience distilled. Packt Publishing, 2015.

[31] Ananth Sowda, Djordje Todorovic, Nikola Prica, and Ivan Baev. Improving llvm debuginfo to recover optimized-out function parameters, 2019.

[32] Anastasia Stulova and Sven van Haastregt. An overview of clang. Technical report, Arm, Cambridge, UK, 2019.

[33] Vojislav Tomašević. Unapređenje infrastrukture llvm dodavanjem informacija za otklanjanje grešaka prilikom autlajning optimizacije. Master's thesis, Elektrotehnički fakultet, Univerzitet u Beogradu, 2022.

[34] E. Ukkonen. *On-line Construction of Suffix-trees*. Series of publications / University of Helsinki, Department of Computer Science. A. University of Helsinki, Department of Computer Science, 1993.

[35] Peng Zhao and José Nelson Amaral. Function outlining and partial inlining. *17th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD'05)*, pages 101–108, 2005.

[36] Miodrag Živković and Vesna Marinković. Konstrukcija i analiza algoritama 2. Materijal sa predavanja, 2022.

Biografija autora

Vladimir Vuksanović je rođen 2. januara 1999. godine u Užicu. Tamo je završio osnovnu školu i prirodno-matematički smer Užičke gimnazije. Posle toga je upisao informatički smer na Matematičkom fakultetu 2017. godine. Osnovne studije je završio za 4 godine sa prosekom 9,82. Nagrađen je kao jedan od najuspešnijih studenata generacije od strane katedre za računarstvo i informatiku i dobitnik je nagrade „Nedeljko Parezanović". Iste godine je upisao master studije na Matematičkom fakultetu takođe na smeru Informatika. Pred kraj studija je radio praksu u kompaniji Sirmija posle čega se tamo zaposlio kao softverski inženjer. Tamo radi na programiranju sistemskog softvera, primarno na projektima vezanim za infrastrukturu LLVM.