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: _____

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

Rezime:

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

Sadržaj

Glava 1

Uvod

Glava 2

Kompajleri

2.1 Informacije za debagovanje

2.2 Kompajlerske optimizacije

2.3 Format DWARF

2.4 Kompajler LLVM

Glava 3

Debageri

3.1 Debager LLDB

Glava 4

Optimizacija izdvajanjem koda

Izdvajanje koda (eng. *outlining*) je kompajlerska 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.

Ova optimizacija je posebno korisna za uređaje sa malom količinom memorije, najčešće ugrađenim (eng. *embedded*) uređajima 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 bas često izvršava (eng. *hot-spot*). Poziv funkcije je skupa instrukcija. Osim samog poziva, stek mora da bude pripremljen za poziv i parametri premešteni na odgovarajuća mesta. 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 1 je prikazan primer C koda. Redovi 11-14 i 16-19 predstavljaju dobre kandidate za izdvajanje. Vrše se iste operacije samo sa drugačijom promeljivom. Ukoliko se ovaj kod prevede do asemblerskog koda komandama prikazanim u listingu 2 tako da se pokrene samo optimizacija izdvajanja koda, rezultujući kod je prikazan u listingu 3. Može se primetiti da je kostruisana nova funkcija `outlined_ir_func_0`. Ona sadrži sve instrukcije iz izdvojenih redova, a ti redovi su zamenjeni pozivom ka toj funkciji.

```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 1: Primer C programa koji je dobar kandidat za izdvajanje koda

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

Listing 2: Instrukcije za prevođenje koda sa uključenim izdvajanjem instrukcija

Ova optimizacija prestavlja suprotan proces poznatijoj optimizaciji, umetanju koda (eng. *inlining*). Umetanje koda podrazumeva zamenu poziva ka funkciji sa telom 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 duplirani dolazi do povećanja memorijskog zauzeća koda.

Iako imaju suprotne efekte, obe optimizacije je moguće koristiti i zajedno na primer za delimično umetanje. 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 traži sekvence koje se ponavljaju već koristi heuristike da pronađe kod koji se retko izvršava.

U jezicima C i C++ postoji ključna reč `inline` koja nagoveštava kompajleru da

```
1  main:                                   # @main
2  .Lfunc_begin1:
3      .loc    0 11 0 is_stmt 1             # outline.c:11:0
4      .cfi_startproc
5  ...
6      movq    %rbx, %rdi
7      movq    %r14, %rsi
8      callq    outlined_ir_func_0
9  ...
10     movq    %rbx, %rdi
11     movq    %r14, %rsi
12     callq    outlined_ir_func_0
13 ...
14   .type    outlined_ir_func_0,@function    # -- Begin function outlined_ir_func_0
15 outlined_ir_func_0:                      # @outlined_ir_func_0
16 .Lfunc_begin2:
17     .loc    0 0 0 is_stmt 1              # outline.c:0:0
18     .cfi_startproc
19 ...
```

Listing 3: Isečci iz asemblerskog koda koji sadrži funkciju dobijenu izdvajanjem koda

primeni umetanje koda na označenu funkciju. Kompajler nije primoran da primeni optimizaciju, već je slobodan da izabere da li će funkciju umetnuti ili ostaviti poziv. Za razliku od toga, za izdvajanje koda ne postoji sličan mehanizam. Ukoliko kompajler podržava ovu optimizaciju, na njemu je da izabere koji delovi koda će biti izdvojeni. Ova optimizacija nije podrazumevano uključena, već programer može da je uključi na podržanim kompajlerima najčešće kompilacijom programa sa opcijom `-Os` (ili `-Oz` za clang) koja optimizuje program za memoriju umesto za vreme.

4.1 Implementacija u kompajleru LLVM

Kompajler LLVM, kao jedan od najpopularnijih kompajlera za jezik C++ ima podršku za veliki broj optimizacija. Jedna od tih optimizacija je i izdvajanje koda. Postoje dve implementacije izdvajanja koda u okviru kompajlera LLVM:

1. `MachineOutliner` - na mašinski-zavisnom međukodu (MIR) [5]
2. `IROutliner` - na mašinski-nezavisnom međukodu (IR) [4]

Obe su implementirane kao prolazi (eng. *pass*) 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 ako bi se koristila samo

jedna od njih. Oba mesta implementacije imaju 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 konkretne ciljne arhitekture ne može odrediti veličina instrukcija. Zbog korišćenja procena za cene instrukcija u obe implementacija je moguće da rezultat izdvajanja koda ima veću veličinu od početnog koda.

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 sadrži dva osnovna koraka: odabir kandidata i izdvajanje koda.

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, ilegalne i nevidljive. Legalne instrukcije su one koje je dozvoljeno izdvojiti. Tu spada većina instrukcija: aritmetičke, logičke... Ilegalne instrukcije su one čije izdvajanje je zabranjeno kao što su pozivi 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 ponavljajucih podniski (eng. *longest repeated substring*). Implementirani algoritam korsiti sufiksno stablo. Sufiksno stablo je struktura podataka koja omogu-

ćava efikasno izvršavanje nekih algoritama nad niskama. Formalno, sufiksno stablo niske S dužine n je korensko stablo za koje važi:

- stablo ima tačno n listova koji su numerisani od 1 do n
- osim korena, svaki unutrašnji čvor ima bar 2 deteta
- svaka grana je označena nepraznom podniskom niske S
- nikoje dve grane koje polaze iz istog čvora nemaju zajednički prefiks
- niska dobijena konkatenacijom oznaka svih grana na putu od korena do lista numerisanog brojem i je sufiks niske S koji počinje od i-tog karaktera

Ovo stablo se sagradi u linearnoj složenosti u odnosu na broj karaktera niske pomoću Ukonenovog algoritma [7]. Broj čvorova u ovom stablu je linearno zavisan od broj karaktera, što znači da je obilazak ovog stabla takođe linearne složenosti. Traženje svih ponavljajućih podniski se može obaviti jednostavnim obilaskom stabla, takođe linearne složenosti.

Kada se odrede kandidati prelazi se u fazu izdvajanja koda. U petlji se odabira grupa kandidata za izdvajanje sa najvećim izračunatim doprinosom. 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 koda na tom međukodu. 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 koje sa pozitivnim doprinosom. Ukoliko ih više nema, prolaz izdvajanja koda je završen.

Specifičnosti na mašinski-zavisnom međukodu

Izdvajanje koda na mašinski-zavisnom međukodu je implementirano u klasi `MachineOutliner`. Ovo je jednostavnija od dve implementacije.

Heširanje instrukcija na ovom nivou je jednostavno. Poređenje se vrši po jednakosti. Dakle instrukcija i operandi moraju da budu identični. Ovo je 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 kod. Još uvek važi klasifikacija instrukcija, tako da instrukcije klasifikovane kao ilegalne nije moguće izdvojiti. Klasifikaciju instrukcija radi `TargetInstrInfo`.

Procena dobitka se računa po formuli: Cene instrukcija na ciljnoj arhitekturi se dobijaju kroz interfejs `TargetInstrInfo`.

Grupe se sortiraju po benefitu. Izbacuju se svi kandidati koji se preklapaju sa prethodno izdvojenim kodom. Generiše se nova funckija sa imenom `OUTLINED_FUNCTION_X` gde je `X` redni broj izdvojene funkcije. Kopiraju se instrukcije proizvoljnog kandidata, generišu se informacije za debagovanje. Svi kandidati se zamenjuju pozivom ka generisanoj funkciji.

Specifičnosti na mašinski-nezavisnom međukodu

Izdvajanje koda na mašinski-nezavisnom međukodu radi prolaz koji se zove `IROutliner`.

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 ili tip predikta za instrukciju poređenja). Operandi ne utiču na heš vrednost instrukcije zato što se oni mogu izvući kao argumenti izdvojene funckije. Ova razlika omogućava izdvajanje dosta više koda.

Dobitak se računa po narednoj formuli: Procene cena instrukcija nakon transformacije na ciljnu arhitekturu se dobija kroz interfejs `TargetTransformInfo`.

Kandidati se izdvajaju iz dva pokušaja. Prvi put kada se izdvoje određuju se svi argumenti koje je potrebno izdvojiti. Drugi put se izdvajanje vrši uz zamenu argumenata. Jedan on izdvojenih kandidata se proglašava za konačnu izdvojenu funkciju. 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. Svaki kandidat je zamenjen pozivom ka funkciji prilikom čega se dodaju i instrukcije za čuvanje i učitavanje argumenata.

4.2 Izdvajanje koda i debag lokacije

Debag lokacije predstavljaju pozicije u izvornom kodu koje odgovaraju instrukciji u izvršnom fajlu. Sastoje se od fajla, reda i kolone. U idealnom slučaju, svaka instruckija u izvornom kodu bi trebalo da ima odgovarajuću lokaciju, ali to nije slučaj pogotovu u optimizovanom kodu. Veliki broj optimizacija menja kod na način koji može da naruši tačnost debag lokacija. Neki od najčešćih slučajeva su brisanje

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

Listing 4: Komande za ispis tabele linija koristeći alate llvm-dwarfdump, readelf i objdump

```
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 5: Tabela linija ispisana pomoću alata llvm-dwarfdump

nepotrebnih instrukcija ili pomeranja instruckija van tela petlje. Kompajleri 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 (kod) 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 razumljive od strane čoveka. Listing 4.2 prikazuje upotrebu nekoliko alata za dekodiranje tabele linija, a u listingu 4.2 se nalazi primer jedne tabele.

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 instrukcije koja ima lokaciju. Standard propisuje i specijalne debag lokacije koje se poznaju po tome sto imaju red 0. Ove lokacije znače da je kod kompajlerski 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 funckije. 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 [1]

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 4.2.

```
$ lldb outline
(lldb) breakpoint set 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    }
```

Glava 5

Implementacija rešenja

Implementacija rešenja nadograđuje verziju 16 projekta LLVM, što je najnovija stabilna vezija u vreme pisanja rada. Celokupan izvorni kod je dostupan na servisu GitHub na adresi `https://github.com/vvuksanovic/llvm-project/tree/outline-debug-info`.

Rešenje se sastoji iz dva dela. Prvo je potrebno u okviru kompajlera 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 kompajleru

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 nije u opticaju za čuvanje vrednosti. Ideja je da se debag lokacije sačuvaju u `.debug_info` sekciji 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 [2]. Trenutno se već koristi za pružanje informacija o vrednostima parametara prilikom poziva funkcije [3] 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.

Trenutna implementacija ima mekoliko bitnih ograničenja. Prvo, podržano je čuvanje samo podataka sa mašinski nezavisne međureprezentacije, ali se lako može

proširiti da podržava i mašinski zavisnu međureprezentaciju. Takođe, `SelectionDAG` je jedina podržana implementacija izbora instrukcija. Ona je izabrana zato što je podrazumevana za veliki broj arhitektura uključujući `x86_64`.

Dakle, debag lokacije moraju da budu prenesene od `IROutliner` optimizacionog prolaza, 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 kod sa manje instrukcija nego sto je bilo pre toga, ne postoji dovoljno mesta da se smeste potrebne informacije. Kao prvi korak očuvanju ovih podataka ovaj rad uvodi novu intrinzičku 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.

Intrinzičke funkcije su funkcije koje su ugradjene u sami kompajler takve da on može da ih implementira na najoptimizovaniji način [6]. One predstavljaju mehanizam proširenja LLVM jezika takav da ne zahteva modifikaciju svih prolaza kada se nešto doda u jezik. Imena svih intrinzičkih funkcija počinju prefiksom „`llvm.`", on će u nastavku biti ometen. Posebnu klasu intrinzičkih 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 kod 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. 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 `dbg.outlined` funkcije. Deklaracija ima dva parametra, oba su metapodaci koji pokazuju na `outline_id` identifikatore redom za odgovarajuci 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 pruzima od izdvojene instrukcije koju zamenjuje.

Kreiranje nove intrinzičke funkcije za debagovanje je veoma jednostavno i radi se na deklarativan način. Jedino što je potrebno uraditi je dodati deklaraciju u `Intrinsics.td` fajlu sa nazivom, u parametrima. Za lakše upravljanje tom funkcijom i njenim parametrima napravljen je novi tip instrukcije `DbgOutlinedInst` koja nasleđuje `DbgInfoIntrinsic` klasu. Ta instrukcija služi kao omotač i koristi se kao alias za instrukciju poziva `dbg.outlined` funkcije. Instanciranje ove instrukcije se radi uz pomoć nove funkcije u klasi `DIBuilder`. Metapodatak `outline_id` se registruje u definicionom fajlu `FixedMetadataKinds.def`. Implementacija se nalazi u klasi `DebugInfoMetadata` i nju je potrebno registrovati u `Metadata.def`. Dodatno, uvedena je podrška za čitanje i pisanje LLVM IR fajlova kao i za njihovu proveru ispravnosti.

Za svaku izdvojenu instrukciju koja ima debag lokaciju se generiše odgovarajuća `dbg.outlined` instrukcija i dodaje se posle instrukcije poziva. Isečak LLVM međukoda programa nakon izdvajanja funkcije je prikazan u listingu 5.1.

Funkcije generisane izdvajanjem koda je potrebno na neki način razlikovati od drugih funkcija. Trenutno se već u mašinski nezavisnom međukodu dodaje zastavica (flag) `DIFlagArtificial` ali to nije dovoljno da se razlikuje od drugog kompajlerski 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 `isOutlined` metode u okviru `DISubprogram` klase koja predstavlja metapodatak sa podacima o funkciji.

Čuvanje debag lokacija prilikom izbora instrukcija

Na kraju prethodne glave su potrebne informacije za debagovanje postale 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 zavisnog međukoda u infrastrukturi LLVM se naziva izbor instrukcija. Postoji vise implementacija izbora instrukcija, ali u ovom radu je trenutno podržan samo `SelectionDAG`.

```
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
   ...
}
```

Slika 5.1: Primer LLVM medjukoda nakon optimizacije izdvajanja koda sa novom instrukcijom i metapodatkom.

SelectionDAG prevodi LLVM međukod u reprezentaciju u obliku usmerenog aciklicnog grafa, a svaka instrkcija postaje čvor u tom grafu. Klasa koja predstavlja instrukciju je `SDNode` i oni se konstruišu u okviru klase `SelectionDAGBuilder`. Nad sagrađenim grafom se rade legalizacija tipova i operacija (pretvaranje tipova i operacija koje nisu podržane na ciljnoj arhitekturi u one koje jesu), izbor instrukcija (zamena mašinski nezavisnih instrukcija sa instrukcijama za konkretnu arhitekturu), određivanje redosleda instrukcija, dodeljivanje registara i optimizacije. Krajnji rezultat izbora instrukcija je program čije su instrukcije prestavljene klasom `MachineInstruction`.

Od uvedenih podataka za potrebe ovog rada, za `dbg.outlined` i vezu između `DIOutlineId` i instrukcija za koje su vezane potrebno implementirati dalje prevođenje. Svi metapodaci ostaju u istom obliku u ovoj fazi kompilacije.

Na kraju selekcije instrukcija sve `dbg.outlined` instrukcije će biti zamenjene ekvivalentnom MIR instrukcijom `DBG_OUTLINED` sa istim parametrima i debag lokacijom. Instrukcija `DBG_OUTLINED` je registrovana u definicionim fajlovima `Target.td` i `TargetOpcodes.def`. SelectionDAG tretira instrukcije za debagovanje drugačije od ostalih instrukcija. One se takođe pretvaraju čvorove usmerenog aciklicnog grafa ali umesto da se ubace u usmeren aciklicni 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 cvora `SDDbgOutlined` i dodato polje u klasi `SelectionDAG` za njihovo skladistenje. Kada se gradi SelectionDAG, sve `dbg.outlined` instrukcije se pretvaraju u `SDDbgOutlined` i čuvaju u novom polju. Na kraju izbora instrukcija, kada je potrebno poređati instrukcije svaki `SDDbgOutlined` sadrži redni broj koji je dobio prilikom konstrukcije i na osnovu njega se `DBG_OUTLINED` emituju na očekivanom mestu u kodu.

Na sličan način kao za čvorove instrukcija za debagovanje, tako se i podaci o `outline_id` metapodatku instrukcija čuva odvojeno od grafa. Konkretno, čuva se u novom polju u okviru promenljive `SDCallSiteDbgInfo`. To je mapa koja preslikava `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.

Najveća količina informacija se izgubi prilikom izbora instrukcija. Identifikatori koji stoje iz instrukcije se mogu izgubiti ako se instrukcija zameni drugom bez kopirnanja njenih dodatnih podataka. Na primer, prilikom spustanja mašinski nezavisne

```
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
   ...
```

Slika 5.2:

instrukcije poziva funkcije na mašinski zavisan nivo za arhitekturu X86 pravi potpuno novi čvor. Zato što se `outline_id` čuva u mapi ciji je ključ `SDNode`, zamenom nekog cvora će se izgubiti ta veza osim ako se taj podatak eksplicitno prenese. Pomenuti primer je popravljen u implementaciji, ali još uvek postoji veliki broj slučajeva kada će ovaj problem ispoljiti.

Isečak LLVM mašinski zavišnog međukoda programa nakon izdvajanja funkcije je prikazan u listingu 5.1.

Ispisivanje debag lokacija prilikom emitovanja asemblerskog koda

```
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)
```

Slika 5.3:

5.2 Implementacija u debageru

Učitavanje podataka

Ispisivanje ispravnih linija izvornog koda

Izvršavanje korak po korak

Postavljanje tačaka prekida

Glava 6

Rezultati

Glava 7

Zaključak

Bibliografija

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

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

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

[4] Andrew Litteken. Framework for finding and using similarity at the ir level, 2020.

[5] Jessica Paquette. Interprocedural mir-level outlining pass, 2016.

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

[7] 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.

Biografija autora