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

1 Uvod **1**

2 Kompajleri **2**

2.1 Primeri korišćenja klasičnih LaTeX elemenata 2

2.2 Informacije za debagovanje 2

2.3 Kompajlerske optimizacije 2

2.4 Format DWARF 2

2.5 Kompajler LLVM 2

3 Debageri **3**

3.1 Debager LLDB 3

4 Optimizacija izdvajanjem koda **4**

4.1 Implementacija u kompajleru LLVM 6

4.2 Izdvajanje koda i debag lokacije 9

5 Implementacija resenja **13**

5.1 Implementacija u kompajleru 13

6 Razrada **14**

6.1 Implementacija u debageru 14

7 Rezultati **15**

8 Zaključak **16**

Bibliografija **17**

Glava 1

Uvod

Glava 2

Kompajleri

2.1 Primeri korišćenja klasičnih LaTeX elemenata

Ovo je rečenica u kojoj se javlja citat [5]. Još jedan citat [2]. Isprobavamo navodnike: „Rekao je da mu se javimo sutra”. U tabeli **??** koja sledi prikazani su rezultati eksperimenta. Ово је пример ћириличког текста који се јавља у латиничком документу. U ovoj rečenici se javlja jedna reč na ћирилици . Iza ove rečenice sledi fusnota.[1]

2.2 Informacije za debagovanje

2.3 Kompajlerske optimizacije

2.4 Format DWARF

2.5 Kompajler LLVM

[1] Ovo je fusnota.

Glava 3

Debageri

3.1 Debager LLDB

Glava 4

Optimizacija izdvajanjem koda

Izdvajanje koda (eng. *outlining*) je kompajlerska optimizacija koja smanjuje kolicinu memorijskog prostora koji zauzima program, ali pritom potencijalno povecava 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 slucajevima moguce je da se ubrza i vreme potrebno za kompilaciju programa.

Ova optimizacija je posebno korisna za uredjaje sa malom kolicinom memorije, najcesce ugradjenim (eng. *embedded*) uredjajima poput pametnih satova, mp3 plejera ili uradjaja zasnovanih na razvojnom sistemu Arduino. Vreme izvrsavanja programa moze da bude osteceno ako se izdvoji deo koda koji se bas cesto izvrsava (eng. *hotspot*). Poziv funkcije je skupa instrukcija. Osim samog poziva, stek mora da bude pripremljen za poziv i parametri premesteni na odgovarajuca mesta. Sve to dodaje trosak koji nije prethodno postojao. Izdvajanjem koda koji se retko izvrsava, sa druge strane, moguce je da dodje do poboljsanja u vremenu izvrsavanja zbog bolje organizacije kes memorije. Nazalost, u vreme kompilacije je tesko odrediti koliko puta ce se izvrsiti segment koda, time ni razlika u vremenu izvrsavanja programa nije lako predvidiva.

U listingu 1 je prikazan primer C koda. Redovi 11-14 i 16-19 predstavljaju dobrie kandidate za izdvajanje. Vrse se iste operacije samo sa drugacijom promeljivom. Ukoliko se ovaj kod prevede do asemblerskog koda komandama prikazanim u listingu 2 da se pokrene samo optimizacija izdvajanja koda, rezultujuci kod prikazan u listingu 3. Moze se primetiti da je kostruisana nova funkcija `outlined_ir_func_0`. Ona sadrzi 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 prevodjenje koda sa ukljucenim izdvajanjem instrukcija

Ova optimizacija prestavlja suprotan proces poznatijoj optimizaciji, umetanju koda (eng. *inlining*). Umetanje koda podrazumeva zamenu poziva ka funkciji sa telom te funkcije. Najcesce se primenjuje na kratke i jednostavne funkcije, koje se pozivaju na vise mesta. Time se ponistava cena poziva funkcije, sto poboljsava vreme izvrsavanja, ali pritom posto su neki delovi koda duplirani dolazi do povecanja memorijskog zauzeca koda.

Iako imaju suprotne efekte, obe optimizacije je moguce koristiti i zajedno na primer za delimicno umetanje. Ova ideja koristi izdvajanje koda kako bi iz funkcije namenjene za umetanje bili izdvojeni "hladniđelovi koda i time se smanjile negativne posledice umetanja. U tom kontekstu izdvajanje koda ne trazi sekvence koje se ponavljaju vec koristi heuristike da pronadje kod koji se retko izvrsava.

U jezicima C i C++ postoji kljucna rec `inline` koja nagovestava kompajleru da

```
main:                                   # @main
.Lfunc_begin1:
    .loc    0 11 0 is_stmt 1                # outline.c:11:0
    .cfi_startproc
...
    movq    %rbx, %rdi
    movq    %r14, %rsi
    callq    outlined_ir_func_0
...
    movq    %rbx, %rdi
    movq    %r14, %rsi
    callq    outlined_ir_func_0
...
  .type    outlined_ir_func_0,@function    # -- Begin function outlined_ir_func_0
outlined_ir_func_0:                     # @outlined_ir_func_0
.Lfunc_begin2:
    .loc    0 0 0 is_stmt 1                 # outline.c:0:0
    .cfi_startproc
...
```

Listing 3: Isecci iz asemblerskog koda koji sadrzi funkciju dobijenu izdvajanjem koda

primeni umetanje koda da oznacenu funkciju. Kompajler nije primoran da primeni optimizaciju, vec je slobodan da izabere da li ce funkciju umetnuti ili ostaviti poziv. Za razliku od toga, za izdvajanje koda ne postoji slican mehanizam. Ukoliko kompajler podrzava ovu optimizaciju, na njemu je da izabere koji delovi koda ce biti izdvojeni. Ova optimizacija nije podrazumevano ukljucena, vec programer može da je uključi na podrzanim kompajlerima najcesce 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 podrsku za veliki broj optimizacija. Jedna od tih optimizacija je i izdvajanje koda. Postoje dve implementacije izdvajanja koda u okviru kompajlera LLVM:

1. `MachineOutliner` - na masinski-zavisnom medjukodu (MIR) [4]
2. `IROutliner` - na masinski-nezavisnom medjukodu (IR) [3]

Obe su implementirane kao prolazi (eng. *pass*) koji rade na nivou modula tako da omoguce izdvajanje koda iz razlicitih 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 masinski-zavisnom medjukodu su dostupne informacije o ciljnoj arhitekturi cime je moguce dosta bolje proceniti cene izdvajanja instrukcija. Bolji model cena instrukcija bitno utice na odredjivanje delova koda koji ce biti izdvojeni. Problem na ovom nivou je sto je u tom trenutku vec izvrsena alokacija registara. Kandidati za izdvjanje su ograniceni samo na sekvence koje koriste iste registre, iako mozda postoji vise sekvenci koje vrse isto izracunavanje samo sa drugim registrima. Sa druge strane, na masinski-nezavisnom medjukodu se koriste virtuelni registri. Oni ne moraju da se poklapaju izmedju sekvenci tako da je moguce izdvojiti mnogo veci deo koda. Medjutim ovde se sada javlja problem procene dobitka memorije zato sto se bez konkretne ciljne arhitekture ne moze odrediti velicina instrukcija. Zbog koriscenja procena za cene instrukcija u obe implementacija je moguce da rezultat izdvajanja koda ima vecu velicinu od pocetnog koda.

Obe implementacije se zasnivaju na istoj ideji, samo su adaptirane za medjukod koji optimizuju. U nastavku ce prvo biti objasnjen opsti algoritam, a zatim specificnosti obe implementacije.

Opsti algoritam za izdvajanje koda

Opsti algoritam za izdvajanje koda sadrzi dva osnovna koraka: odabir kandidata i izdvajanje koda.

Algoritam za odabir kandidata se zasniva na ideji da se program moze tretirati kao niz numerickih 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 vecina instrukcija: aritmeticke, logicke... Ilegalne instrukcije su one cije izdvajanje je zabranjeno kao sto su pozivi funkcija ili instrukcije vezane za obradu gresaka. Nevidljive instrukcije ne uticu na izdvajanje, ovo su primarno instrukcije koje se koriste za debagovanje. Pojedinacne vrednosti dodeljene instrukcijama se nadovezuju za svaki osnovni blok i svaki od njih se terminira jedinstvenom vrednosti kako bi se sprecilo 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 trazenje 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 slozenosti u odnosu na broj karaktera niske pomocu Ukonenovog algoritma [6]. Broj cvorova u ovom stablu je linearno zavisan od broj karaktera, sto znaci da je obilazak ovog stabla takodje linearne slozenosti. Trazenje svih ponavljajucih podniski se moze obaviti jednostavnim obilaskom stabla, takodje linearne slozenosti.

Kada se odrede kandidati prelazi se u fazu izdvajanja koda. U petlji se odabira grupa kandidata za izdvajanje na navecim izracunatim doprinosom. Nacin racunanja doprinosa zavisi od implementacije. U oba slucaja se dobija kao procena razlike u velicini programa ukoliko se grupa ne izdvoji i ako se izdvoji. Za odabranu grupu se generise nova funkcija, a svi kandidati se zamenjuju pozivom te funkcije. Pritom se pazi da se odrzi semantika koda na tom medjukodu. Posle svakog odabira kandidata, ta grupa kao i svi drugi kandidati koji se preklapaju sa njom se brisu iz sufiksnog stabla i postupak se ponavlja sve dok postoje grupe koje sa pozitivnim doprinosom. Ukoliko ih vise nema, prolaz izdvajanja koda je zavrsen.

Specifičnosti na masinski-zavisnom medjukodu

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

Hesiranje instrukcija na ovom nivou je jednostavno. Poredjenje se vrsi po jednakosti. Dakle instrukcija i operandi moraju da budu identicni. Ovo je jedno od najvecih ogranicenja na ovom nivou. Ukoliko dve sekvence vrse isto izracunavanje ali njihovim operandima su dodeljeni razliciti registri, nece biti moguce izdvojiti taj kod. Jos uvek vazi klasifikacija instrukcija, tako da instrukcije klasifikovane kao ilegalne nije moguce izdvojiti. Klasifikaciju instrukcija radi `TargetInstrInfo`.

Procena dobitka se racuna 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. Generise se nova funckija sa imenom `OUTLINED_FUNCTION_X` gde je `X` redni broj izdvojene funkcije. Kopiraju se instrukcije proizvoljnog kandidata, generisu se informacije za debagovanje. Svi kandidati se zamenjuju pozivom ka generisanoj funkciji.

Specifičnosti na masinski-nezavisnom medjukodu

Izdvajanje koda na masinski-nezavisnom medjukodu radi prolaz koji se zove `IROutliner`.

Za odredjivanje kandidata se koristi pomocna analiza, `IRSimilarityIdentifier`. Umesto poredjenja celih instrukcija, poredi se samo tip instrukcije i operanda i dodatni parametri za neke instrukcije (npr. ime pozvane funkcije ili tip predikta za instrukciju poredjenja). Operandi ne uticu na hes vrednost instrukcije zato sto se oni mogu izvuci kao argumenti izdvojene funckije. Ova razlika omogucava izdvajanje dosta vise koda.

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

Kandidati se izdvajaju iz dva pokusaja. Prvi put kada se izdvoje odredjuju se svi argumenti koje je potrebno izdvojiti. Drugi put se izdvajanje vrsi uz zamenu argumenata. Jedan on izdvojenih kandidata se proglasava za konacnu 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 brisu. Svaki kandidat je zamenjen pozivom ka funkciji prilikom cega se dodaju i instrukcije za cuvanje i ucitavanje argumenata.

4.2 Izdvajanje koda i debag lokacije

Debag lokacije predstavljaju pozicije u izvornom kodu koje odgovaraju instrukciji u izvrsnom fajlu. Sastoje se od fajla, reda i kolone. U idealnom slucaju, svaka instruckija u izvornom kodu bi trebalo da ima odgovarajucu lokaciju, ali to nije slucaj pogotovu u optimizovanom kodu. Veliki broj optimizacija menja kod na nacin koji moze da narusi tacnost debag lokacija. Neki od najcescih slucajeva su brisanje

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

Listing 4: Komande za ispis tabele linija koristeci 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 pomocu alata llvm-dwarfdump

nepotrebnih instrukcija ili pomeranja instruckija van tela petlje. Kompajleri se trude da odrze sto vecu kolicinu debag lokacija, ali nekada to ili nije moguce ili nema smisla posle izvrsavanja optimizacije.

U nastavku se podrazumeva da se koristi format DWARF za cuvanje informacija za debagovanje. Lokacije za debagovanje se interno cuvaju u tabeli linija u `.debug_line` sekciji izvrsnog fajla. Ta tabela preslikava adresu iz tekst (kod) segmenta izvrsnog fajla na odgovarajucu lokaciju (fajl, red, kolona) u izvornom kodu i neke dodatne informacije o toj lokaciji. Format DWARF propisuje nacin enkodiranja tabele linija u vidu konacnog automata, ali postoje alati koji prikazuju podatke u obliku tabele razumljive od strane coveka. 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 nacin ukoliko trenutna instrukcija nema svoju debag lokaciju, prikazace 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 znace da je kod kompajlerski generisan i da ne postoji odgovarajuca lokacija u izvornom kodu. Prilikom izvrsavanja programa red po red (koristeci komandu `step` u debageru LLDB) preskacu se sve instrukcije sa Na instrukciju koja nema dodeljenu lokaciju je moguce stati izvrsavanjem programa instrukciju po instrukciju, postavljanjem tacke prekida na adresu te instrukcije ili usled zaustavljanja programa zbog izuzetka koji se desio u toj instrukciji.

Prilikom izdvajanja koda jedna instrukcija ima vise mogucih debag lokacija u zavisnosti od mesta poziva izdvojene funckije. Ovo predstavlja vise problema:

- U vreme izvrsavanja programa nije moguce odrediti koja od ovih lokacija je korektna bez dodatnih informacija
- Format DWARF ne podrzava postojanje vise razlicitih lokacija za istu adresu [1]

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

- Izvrsavanje korak po korak pravi velike skokove kada dodje do izdvojenog koda, bez bilo kakvog obavestanja korisniku
- Postavljanje tacaka prekida na izdvojenu instrukciju precizirajuci njen red nije moguce

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 resenja

Implementacija nadogradjuje verziju 16 projekta LLVM. Celokupan izvorni kod je dostupan na Github servisu na adresi `https://github.com/vvuksanovic/llvm-project/tree/outline-debug-info`

Resenje se sastoji iz dva dela. Prvo je potrebno u okviru kompajlera generisati potrebne informacije za debagovanje, a zatim ih iz debagera iskoristiti za poboljsanje procesa debagovanja. U ovom poglavlju ce biti opisana oba dela implementacije.

5.1 Implementacija u kompajleru

Glava 6

Razrada

Cuvanje debag lokacija prilikom optimizacije

Cuvanje debag lokacija prilikom izbora instrukcija

Ispisivanje debag lokacija

6.1 Implementacija u debageru

Ispisivanje linija koda

Izvrsavanje korak po korak

Postavljanje tacaka prekida

Glava 7

Rezultati

Glava 8

Zaključak

Bibliografija

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

[2] Yuri Gurevich and Saharon Shelah. Expected computation time for Hamiltonian path problem. *SIAM Journal on Computing*, 16:486–502, 1987.

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

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

[5] Petar Petrović and Mika Mikić. Naučni rad. In Miloje Milojević, editor, *Konferencija iz matematike i računarstva*, 2015.

[6] 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