UNIVERZITET U BEOGRADU
MATEMATIČKI FAKULTET

Nikola B. Prica

PODRŠKA ZA PROFAJLIRANJE SOFTVERA UREĐAJA SA UGRAĐENIM RAČUNAROM

master rad

Beograd, 2018.

Mentor:

dr Milena Vujošević Janičić, docent
Univerzitet u Beogradu, Matematički fakultet

Članovi komisije:

prof. dr Filip Marić, vandredni profesor
Univerzitet u Beogradu, Matematički fakultet

doc. dr Jelena Graovac, docent
Univerzitet u Beogradu, Matematički fakultet

Datum odbrane: ____________________

Naslov master rada: Podrška za profajliranje softvera uređaja sa ugrađenim računarom

Rezime: Testiranje softvera predstavlja neizostavan deo njegovog razvoja. Za potrebe testiranja mogu se koristiti podaci o broju izvršavanja delova programa koji se dobijaju tehnikom profajliranja. Profajliranje u osnovi predstavlja dodavanje dodatnog koda u softver kako bi se dobile frekvencije izvršavanja funkcije ili bloka koda. Osim frekfencija mogu se dobiti i drugi podaci kao što je procenat utrošenog vremena u bloku koda, ili pak podatak o alokaciji memorije itd. Podaci dobijeni profajliranjem predstavljaju profil programa. Profil programa može služiti za otkrivanje i opzimizaciju najčešće pozivanih funkcija i blokova koda. Prilikom testiranja određenih ulaznih podataka, profil nam može dati informaciju koji delovi programa su se izvršili i na osnovu toga možemo zaključiti da je taj deo programa istestiran.

Jedan od načina profajliranja, koji će detaljnije biti opisan u ovom radu, jeste kompajlerski zasnovano profajliranje gde se dodatan kôd koji broji frekvencije izvršavanja dodaje prilikom kompajliranja samog programa. Problem koji postoji u trenutnom pristupu za dobijanje profila programa i njegovih deljenih biblioteka kompajliranih kompajlerskom insfrasktrukturom LLVM je nedostatak jednostavnog načina za ispisivanje njihovih profila. Razlog ovog problema je taj što se sam program i njegove deljene biblioteke trenutnim pristupom posmatraju kao odvojeni entiteti. Cilj rada je unapređenje kompajlerskog profajliranja pružanjem jedinstvenog funkcijskog poziva za ispisivanje podataka profajliranja svih profajliranih entiteta sa motivacijom boljeg prilagođavanja ograničenim resursima uređaja sa ugrađenim računarom. Dodatno će biti prikazana i transformacija internog formata profajliranja kompajlerske infrastrukture LLVM tako da ona može da obezbedi ekvivalentne podatke kao i kompajler GCC.

Zbog posebnosti uređaja sa ugrađenim računarom, rezultati rada će biti primenjeni na softveru operativnog sistema *Android-NOUGHAT* za MIPS uređaj sa ugrađenim računarom. Softver na takvim uređajima uglavnom imaju neodređeno vreme izvršavanja te su s toga rezultati ovog rada od velikog značaja za takve uređaje. Kako uređaji sa ugrađenim računarom uglavnom zahtevaju memorijsku efikasnost s jedne strane, a s druge strane, profajliranje u velikoj meri povećava veličinu softvera, to dodatno povećava potrebu za testiranjem na ovakvim uređajima.

Ključne reči: profiajliranje, Clang, LLVM, uređaji sa ugrađenim računarom, embeded sistemi, compiler-rt, clang_rt, compiler_rt

Sadržaj

1 Uvod **1**

2 Sistemi sa ugrađenim računarom **3**

2.1 Šta su sistemi sa ugrađenim računarom? 3

2.2 Razvoj namenskih uređaja 5

2.3 Karakteristike namenskih sistema 6

2.4 Ograničenja namenskih sistema 7

3 Projekat LLVM **10**

3.1 Struktura i organizacija LLVM-a 11

3.2 Prednji deo kompajlera 12

3.3 Srednja reprezentacija 13

3.4 Zadnji deo kompajlera 14

4 Profajliranje **17**

4.1 Instrumentalizacija 18

4.2 Profajliranje ivica i blokova 19

4.3 Izračunavanje broja izvršavanja ivica 19

4.4 Profajliranje uzimanjem uzorka 21

4.5 Algoritam uzorkovanja 22

4.6 Minimizacija uticaja na performanse 24

5 Zajednički format izlaza profajliranja kompajlera Clang i GCC **25**

5.1 Mapirajući pokrivač u kompajleru Clang 26

5.2 Proširivanje formata mapirajućeg pokrivača informacijama o grananju . . 31

5.3 Primer generisanja novog formata 34

6 Unapređenje biblioteke za ispisivanje rezultata profajliranja **36**

6.1 Biblioteka clang_rt.profile 37

6.2 Generisanje dodatnog koda u IR prilikom profajliranja 38

6.3 Funkcijski poziv za ispisivanje svih podataka profajliranja 40

6.4 Testiranje 43

6.5 Dalji rad 45

7 Zaključak **46**

Literatura **48**

Glava 1

Uvod

Oblasti primene i tipovi uređaja sa ugrađenim računarom (televizori, modemi, ruteri, automobili sa sistemom za autonomnu vožnju itd.) se iz godene u godinu šire. Među njima su veoma popularni uređaji koji koriste arhitekturu MIPS [23]. Softver na ovim uređajima mora biti pouzdan, a zbog ograničenosti hardvera samih uređaja, postavlja se i zahtev za visokom vremenskom i prostornom efikasnošću koda. Da bi se ostvarili zahtevi za pouzdanošću i efikasnošću aplikacija za uređaje sa ugrađenim rečunarom, potrebno je prikupiti i analizirati podatke o radu uređaja. Statistički podaci koji su od interesa i koji se razmatraju u ovom radu su podaci o broju izvršavanja linija izvornog koda. Takvi podaci su profili programa. Oni omogućavaju uvid u pokrivenost koda prilikom testiranja, čime se može proceniti kvalitet testova, a time indirektno i kvalitet softvera. Povod za ovaj rad je nastao iz potrebe za dobijanjem statističkih podataka o softveru uređaja sa ugrađenim računarom za čiju izgradnju se koristi kompajler projekta LLVM [24].

Clang, kompajler projekta LLVM, sve više je zastupljen pa samim tim zajednica koja na njemu radi je sve veća. On, kao i kompajler GCC [5], podržava pravljenje pofila programa prilikom njegovog izvršavanja. Kako se program i njegove deljene biblioteke posmatraju kao odvojeni entiteti, kompajlerska infrastruktura LLVM i kompajler GCC ispisuju profil entiteta u okviru koga je funkcija za ispisivanje pozvana. Ispisivanje profila svih entiteta se izvršava tek na kraju rada programa kao deo njegove finalizacije. Cilj ovog rada jeste proširivanje kompajlerske infrastrukture LLVM pružanjem efikasnog jedinstvenog funkcijskog poziva koji se može pozvati u bilom kom trenutku izvršavanja programa radi ispisivanja trenutnih profila programa i njegovih deljenih biblioteka.

Cilj poglavlja 2 je da uvede u sisteme sa ugrađenim računarom, da prikaže neke njihove osnovne karakteristike, ograničenja i njihov značaj. Pregled osnovnih komponenti i načela projekta LLVM opisani su u poglavlju 3. U narednom poglavlju 4 prikazane su neke od osnovnih tehnika profajliranja kao i neki od algoritama koji se koriste kako bi se iz programa mogao izvući njegov profil.

Da bi rezulati prikupljanja profila programa kompajlerom Clang pružli iste informacije kao rezultati profajliranja kompajlerom GCC implementirana je transformacija internog

formata profila kompajlera Clang. Cilj ove transformacije je obezbeđivanje uniformnog formata kako bi se iskoristili već postojeći alati za obradu formata profila kompajlera GCC. Implementacija ove transformacije prikazana je u poglavlju 5.

U okviru rada u poglavlju 6 biće prikazano unapređenje *profile* biblioteke koja je sastavni deo rantajm biblioteka podprojekta *compiler-rt* koje koristi Clang. Kako je memorijski faktor presudan prilikom profajliranja ugrađenih uređaja zbog njihove memorijske ograničenosti, dodatno će biti upoređena količina memorije potrebna za profajiranje kompajlerom Clang na operativnom sistemu *Android-NOUGHAT* koji će se izvršavati na MIPS uređaju.

Implementacije koje su opisane u poglavlju 5 i 6, i koje su osnovni doprinos ovog rada, mogu se preuzeti komandom '*git clone https://gitlab.com/prica/PatchesAndInstallation.git*' u vidu zakrpa (eng. patch). Takođe, ovom komandom se preuzima i datoteka README.txt u kojoj je opisano kako primeniti ove zakrpe na odgovarajuću verziju projekta LLVM. Kroz rad će biti pokazano da je ovim implementacijama ostvareno unapređenje kompajlerskog profajliranja i da je kompajlersko profajliranje na ovaj način bolje prilagođeno ograničenim resursima uređaja sa ugrađenim računarom.

Glava 2

Sistemi sa ugrađenim računarom

Tomas Vatson (eng. *Tomas Watson*), nekadašnji direktor kompanije IBM, 1943. godine rekao je: „Mislim da postoji svetsko tržiste za možda pet računara" [17]. Međutim, danas smo okruženi računarima, srećemo ih svuda oko nas – na poslu, na ulici, u našim domovima... Većina ljudi pojam računara izjednačava sa kućnim desktop računarima, laptop računarima ili superračunarima, tj. sa pojmom računara opštih namena (eng. *general-purpouse computer*). Međutim, veliki broj uređaja sadrži u sebi ugrađene računare (eng. *embedded devices*) koji su napravljeni da obavljaju jasan skup poslova ili čak jedan specifičan posao. Postojanje procesora i softvera na nekim od njih može biti veoma neprimetno. Uređaji sa ugrađenim računarom zajedno za softverom koji se na njima izvršava čine sistem specifične namene ili ugrađeni sistem (eng. *embedded system*). Neki primeri ovih sistema su mikrotalasne pećnice, liftovi, digitalni satovi, televizori, konzole za igrice, štampači, skeneri, digitalne kamere, razni uređaji za seizmološka merenja ... Oni mogu biti i deo nekog većeg sistema, kao što je, na primer, sistem za video nadzor u pametnoj kući.

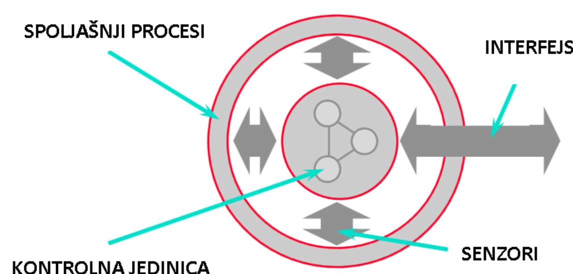
2.1 Šta su sistemi sa ugrađenim računarom?

Za razliku od računara opštih namena, koje možemo koristiti za slušanje muzike, gledanje filmova, pisanje različitih dokumenata, pretraživanje interneta i još mnogo više, uređaji sa ugrađenim računarom napravljeni su tako da rešavaju specifičan skup poslova. Najjednostavniji primer namenskog sistema predstavlja lift. Pomoću table za izbor sprata pristupamo sistemu i biramo željenu akciju. Sistem obrađuje unos, proverava da li je ukupna težina osoba u liftu ispod maksimalne i ukoliko jeste prelazi na određeni sprat, a ukoliko nije daje signal o prekoračenju težine.

Sistem predstavlja funkcionalnu celinu sastavljenu od više komponenti koje poštuju određen skup pravila. Može se posmatrati i kao način organizovanja više poslova po specifičnom planu [4]. Svaki sistem postoji u nekom okruženju – ima nekakve granice,

ima ulaze i izlaze; obuhvata neke procese; može imati podsisteme, kontrolne mehanizme... Povezivanje više sistema u kompleksniji sistem može mu davati neka svojstva koja ne potiču od komponenti već su posledica povezivanja u celinu.

Namenski sistemi (ili sistemi sa ugrađenim računarom) predstaljaju kombinaciju hardvera, softvera i još nekih delova, mehaničkih ili električnih, dizajniranih tako da obavljaju predodređen skup funkcija. Namenski sistem je okružen spoljašnjim procesima koji pripadaju njegovom domenu. Preko različitih vrsta fizičkih uređaja ili senzora sistem dobija informacije o relevantnim spoljašnjim procesima. Sami uređaji za prikupljanje podataka mogu predstavljati sistem za sebe koji će ovako prikupljene podatke obraditi i proslediti za dalju obradu kontrolnoj ili upravljačkoj jedinici. Ona je zadužena za upravljanje procesima sistema. Procesor kao centralna hardverska komponenta i operativni sistem kao softverska čine centralni deo svakog namenskog sistema. Važan deo sistema je i deo koji šalje signale kontrolnoj jedinici i koji je zadužen za prikazivanje trenutnog stanja sistema a koji se naziva korisnički interfejs [3]. On omogućava korisniku da kontroliše neke aspekte sistema. Osnovne komponente sistema prikazane su na slici 2.1.



Slika 2.1: Uopšteni prikaz sistema specifičnih namena

Prema [3] na osnovu prethodno navedenog namenski sistem možemo dekomponovati na pet logičih entiteta:

1. Upravljačka jedinica
2. Kontrolisani fizički uređaji (eng. *the controlled physical device*)
3. Korisnički interfejs
4. Domen ili okruženje
5. Korisnik

Važno je napomenuti da se računari opštih namena mogu povezati sa velikim brojem namenskih sistema. Tastatura i miš su najjednostavniji primer takvog povezivanja. Modem takođe predstavlja sistem specifične namene koji služi za primanje i slanje digitalnog signala preko analogne telefonske linije.

2.2 Razvoj namenskih uređaja

Prvi prepoznatljiv namenski uređaj potiče iz 1960ih godina i to je „Apollo guidance computer", razvijen od strane Čarls Drapera (eng. *Charles Draper*) i njegovog tima. Kasnije je ovaj uređaj našao svoju primenu u različitim industrijama kao što su vojna, vazdušno-kosmička, automobilska, a takođe i u medicini [22].

Pojava mikroprocesora pružila je široko polje za razvoj namenskih sistema. Japanska kompanija Busicom 1969. godine je zatražila od INTEL-a više različitih skupova integrisanih kola, svaki za različitu vrstu digitrona. INTEL-ov odgovor je bio prvi mikroprocesor 4004. Procesor je bio dizajniran da čita skup instrukcija sa spoljšnjeg memorijskog čipa i izvršava ih. Popunjavanjem memorije odgovarajućim redosledom instrukcija postizala se različita funkcionalnost za svaku vrstu digitrona [17].

Nedugo nakon pojave mikroprocesora već je postojala potreba za mikrokontrolerima. Mikrokontroleri, u blagim crtama rečeno, za razliku od mikroprocesora, uključuju RAM, ROM, ulazno izlazne uređaje i u nekim slučajevima neke druge periferne uređaje na jednom čipu, dok mikroprocesor mora biti povezan sa nekim oblikom magistrale podataka kako bi slao i dobijao podatke. Mikrokontroleri generalno obavljaju manji skup operacija od mikroprocesora ali su zato i jeftiniji za proizvodnju [**micr**]. Kao takvi oni su izuzetno povoljni za razvoj namenskih uređaja.

Naredne decenije upotreba mikroprocesora i mikrokontorlera se ravnomerno povećavala. Prvobitni namenski sistemi su bili za kontrolu semafora i kontrolu aerodromskog saobraćaja. Osamdesetih i devedesetih godina prošlog veka namenski sistemi su već našli veliku primenu u kuhinjama (mikrotalasne peći), dnevnim sobama (televizori, zvučnici, daljinski upravljači) i poslovima (faks mašine, štampači, čitači kartica) [22].

Sledeće generacije nameskih sistema, pored sistema zasnovanih na mikroprocesorima i mikrokontrolerima, razvijaju se na naprednijim integrisanim kolima poznatim kao sistemi na čipu (eng. system on a chip - SoC). Sistemi na čipu po svojim karakteristikama veoma podsećaju na mikrokontrolere s tim što su za razliku od njih mnogo programabilno fleksibilniji i hardverski složeniji a samim tim pružaju i bolje performanse. Klasičan primer sistema na čipu predstavljaju mobilni telefoni.

Broj namenskih sistema se veoma brzo povećava. Idući u korak sa razvojem interneta postoji potreba za povezvanjem različitih uređaja preko mreže. Tehnologija, koja je još u razvoju, „Internet Stvari" se bavi kontrolisanjem različitih personalnih uređaja preko mreže. Jedan slučaj primene ove tehnologije je uključivanje klime preko mobilnog telefona.

Perspektivnih sistema specifičnih namena u razvoju ima mnogo, kao i prostora za razvoj novih. Kontrolisanje saobraćaja informisanjem vozača o najboljim rutama tako da se izbegne gužva, inteligentni erbeg sistemi koji prepoznaju da li je na sedištu odrasla ili mlađa osoba, kontrolisanje potrošnje energije kućnih uređaja od strane udaljenog

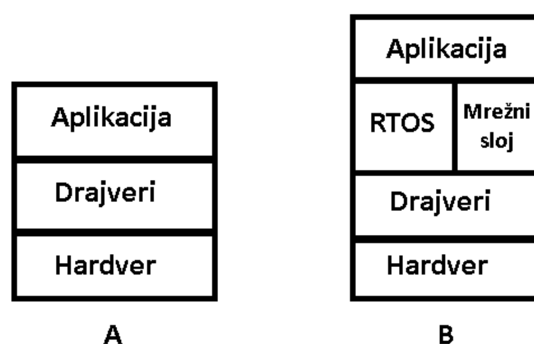
centralnog računara, predstavljaju samo neke od potencijalnih sistema koji imaju veliki potencijal [17].

2.3 Karakteristike namenskih sistema

Namenski sistemi imaju nekoliko zajedničkih karakteristika. Imaju veća hardverska ograničenja od računara opštih namena. Namenski sistemi se prave da budu jeftini, efikasni, ekonomični (po pitanju potrošnje energije i zauzimanja memorijskog prostora na samom uređaju) i kompaktni. Oni su reaktivni sistemi koji zahtevaju odzivnost na promene u okruženju. Rad sistema treba da bude u koraku sa okolinom. Njegova pouzdanost zavisi od ove osobine. Na primer ukoliko se promeni temperatura sobe klima uređaj mora očitati ovaj podatak i prikazati ga korisniku [25]. Softver se koristi za postizanje fleksibilnosti sistema, da može da se menja vremenom kako bi odgovorio novim potrebama korisnika, a hardver treba da garantuje performanse.

Veoma bitan deo nameskih sitema, pored procesora, softvera, ulaznih i izlaznih komponenti, je memorija koja se koristi za smeštanje izvršnog koda i za baratanje podacima za vreme rada sistema. ROM (eng. read-only memory) i RAM (eng. random-access memory) memorija su sastavni deo svakog namenskog uređaja. Obično je memorija vrlo ograničena jer uređaji teže da budu što manji i lakši. Ako je potrebna samo mala količina memorije one zajedno mogu biti deo procesorskog čipa [17]. Ukoliko uređaj zahteva veću količinu memorije obično se to ostvaruje korišćenjem SD kartice.

Osim memorije i procesora ostale hardverske komponente namenskih uređaja su jedinstvene po svom načinu funkcionisanja, u zavisnosti od namene i proizvođača. Svaki od ovih delova zahteva poseban softver koji preslikava njegove mogućnosti i funkcije kako bi se one pozvale u nekom od viših slojeva sistema. Takav softver piše proizvođač hardverske komponente i on predstavlja *drajver* za tu komponentu. Oni omogućavaju upotrebu komponente bez potrebe za poznavanjem njenih detalja [17].



Slika 2.2: Osnovna podela sistema specifičnih namena na osnovu njihove složenosti

Namenski sistemi se u zavisnosti od kompleksnosti sistema ili arhitekture kernela mogu podeliti na različite načine. Na slici 2.2 je prikazana najopštija podela na (A) jednostavne i (B) složenije. Kompleksniji sistemi zahtevaju efikasnije upravljanje resursima sistema. Primećujemo da je hardverski i drajverski blok zajednički jer oni predstavljaju osnovu svakog sistema. Složeniji sistemi uključuju više softverskih slojeva sa zaokruženim skupom funkcionalnosti predstavljenih interfejsom prema višim slojevima. U takvim sistemima uvek postoji sloj operativnog sistema koji radi u realnom vremenu (eng. *real-time operating systems*) (u daljem tekstu RTOS). RTOS je operativni sistem namenjen uređajima koji treba da obrade podatke u realnom vremenu, bez kašnjenja. Pored njega dotatnu pomoć aplikativnom delu pruža i mrežni sloj. On omogućava lakše i efikasnije povezivanje sa drugim uređajima sa kojim je sistem umrežen [17].

Efikasnost aplikativnog sloja zavisi od funkcija koje mu pružaju niži slojevi, tj. od kompletne arhitekture i organizacije namenskog sistema jer postoji sprega između svakog sloja. Određene aplikacije na namenskim uređajima imaju predviđeno vreme za koje neka određena operacija mora tačno da se izvrši. Recimo, predugo izračunavanje neke funkcionalnosti na sistemu za kontrolu leta može ugroziti živote putnika. Ovakvi sistemi, koji ukoliko ne ispune operaciju u očekivanom vremenu imaju katastrofalne posledice, su „kritični sistemi koji rade u realnom vremenu". Što su ozbiljnije posledice to su ti sistemi kritičniji. Oni zahtevaju visoku pouzdanost i efikasnost sistema [17].

2.4 Ograničenja namenskih sistema

Proizvodnja svakog namenskog sistema ima posebne uslove koji su uglavnom međusobno zavisni. Tako, na primer, ako je potrebno da proizvodnja košta manje od 1000 dinara po komadu moramo koristiti jeftiniji procesorski čip i zadovoljiti se manjom pouzdanošću uređaja. Potrebno je dizajnirati sistem tako da može da zadovolji ciljnu funkcionalnost, ali ujedno i da optimizuje veliki broj metrika koje zadovoljavaju zadate uslove. Metrike predstavljalju neku od merljivih odlika sistema, a neke od njih su [25, 17]:

1. **Procesorska snaga**: Meri se u broju instrukcija koje procesor može da izvrši u jednoj sekundi. Pored toga potrebno je uzeti u obzir širinu procesorskog registra koji se kreće u rasponu od 8 do 64 bita.

2. **Memorija**: Veličina memorije (ROM i RAM) dovoljne da sačuva programe i potrebne podatke koji se koriste pri radu. Gornja granica količine memorije zavisi od veličine procesorskog registra (16-bitni adresni registar može da adresira 64KB memorije).

3. **Fizička veličina uređaja**: Predstavlja veličinu uređaja u centrimetrima i njegovu težinu u gramima. Teži se što manjim i kompaktnijim uređajima.

4. **Potrošnja energije**: Za uređaje koji koriste bateriju ovo svojstvo je izuzetno bitno. Manja potrošnja energije povlači za sobom manje zagrevanje uređaja, lakšu bateriju, jednostavniji mehanizam rada i manju veličinu uređaja.

5. **Cena proizvodnje jednog primerka**: Odgovara ceni hardverskih komponenti potrebnih da se ceo sistem sklopi.

6. **Cena razvoja**: Cena potrebna da se dizajnira odgovarajući hardver i razvije softver za njega. Kada se ovaj proces jednom završi moguće je praviti kopije namenskog sistema po ceni proizvodnje jednog primerka.

7. **Broj primeraka**: Očekivana količina uređaja sa kojom će se izaći na tržište. Odnos između cene razvoja i proizvodnje jednog primerka direktno utiče na očekivani broj proizvedenih i prodatih primeraka.

8. **Fleksibilnost**: Mogućnost promene funkcionalnosti uređaja bez velike cene razvoja za novi sistem.

9. **Trajnost**: Očekivani životni vek korišćenja sistema.

10. **Pouzdanost**: Predstavlja odliku sistema da uspešno obavlja zadatu funkciju, pod određenim uslovima, u datom vremenskom intervalu [18].

Uređaji sa ugrađenim računarom imaju izuzetno uska ograničenja za većinu navedenih odlika sistema. Često se pribegava manjoj količini memorije zbog smanjenja troškova proizvodnje i zbog čestog zahteva da dimenzije i težina uređaja budu što manje da bi uređaj bio što praktičniji. Memorijska ograničenja zahtevaju da softver koji treba da se izvršava na uređaju bude što efikasniji. Jedan od način optimizacije softvera je korišćenje rezultata profajliranja (eng. *profiling*) kako bi se otkrilo koje delove koda treba bolje da optimizujemo. Ako profajliranje ne dovede do željenih rezultata vrši se štelovanje softvera, ukoliko je to moguće, tj. vrši se menjanje konstantnih parametara koji oslikavaju okruženje i ograničenja u kojima softver postoji. Optimizacija se takođe može vršiti i prosleđivanjem različitih parametara komapjleru prilikom izgradnje softvera. Ukoliko ni to nije dovoljno mora se menjati celokupna arhitektura sistema.

Pouzdanost softvera predstavlja jednu od najbitnijih metrika namenskih sistema. Da bi se što manje grešaka i neočekivanih situacija javilo u praksi neophodno je podvrgnuti softver skupu testova, koji mogu testirati određene module ili rad sistema u celini, tako da oni pokriju što je moguće više slučajeva upotrebe i time učine sistem pouzdanijim. Prilikom testiranja postavlja se pitanje da li je naš skup testova dovoljno dobar ili ga treba proširiti. I u ovom slučaju možemo koristiti profajlere. Rezultati profajliranja nose informaciju koliko puta se izvršila određena instrukcija ili funkcija. Takvi podaci mogu poslužiti kao heuristika za povećanje pokrivenosti koda softvera zadatim skupom

testova. Što je veći deo softvera istestirano to je on pouzdaniji. Danas postoje sistemi koji primenom ove heuristike automatski unapređuju skup testova [13].

Glava 3

Projekat LLVM

LLVM je kompajlerska infrastruktura koja je inicijalno razvijena od strane Vikrama Adve i Krisa Latnera na Univerzitetu Ilinois 2000. godine. Ona predstavlja kolekciju konfigurabilnog kompajlera[1] i softverskih alata koji se mogu ulančavati ili posebno koristi radi ostvarivanja određene funkcionalnosti [24]. LLVM je mladi kompajler u poređenju sa najpopularnijim kompajlerom otvorenog koda GCC (GNU Compiler Collection) [5], koji je pokrenut 1985. a objavljen 1987. godine. LLVM je kompletno napisan u programskom jeziku C++ i veoma je modularan. Ovaj projekat se brzo razvija jer danas postoji velika zajednica koja ga unapređuje svakodnevno [19].

Prvobitna zamisao prilikom izgradnje LLVM-a je da se stvori „kompajlerska infrastruktura koja će podržavati transparentnu analizu i transformaciju proizvoljnog programa za vreme njegovog izvršavanja, ujedno obezbeđujući dovoljno informacija za novu transformaciju programa za vreme kompajliranja, linkovanja, izvršavanja i vreme između njegova dva pokretanja" [11]. Kasnije se ovom skupu funkcionalnosti dodaje i transformacija prilikom instalacije programa. Ono što razlikuje LLVM od drugih kompajlerskih sistema i šta omogućava navedene osobine je njegova srednja reprezentacija (eng. *intermediate representation - IR*). Upravo je ona centralni deo LLVM-a. Mogućnost njenog zapisivanja na disku je ključna za optimizaciju programa tokom čitavog njegovog postojanja. Kako je srednja reprezentacija arhitekturalno zavisna, prenosivost programa samo preko srednje reprezentacije povlači da LLVM treba da radi kao virtuelna mašina. Međutim razvojem projekta ideja o srednjoj reprezentaciji koja se čuva na disku je ostala prvenstveno kako bi pružila osnove za stvaranje optimizacija tokom faze linkovanja, i vremenom je pridavan sve manji značaj prvobitnoj ideji o optimizaciji tokom čitavog života programa. Zbog toga je projekat LLVM izgubio svoj prvobitni cilj da postane virtuelna mašina i odbacila je akronim LLVM (eng. *Low Level Virtual Machine - LLVM*) „virtuelne mašine niskog nivoa" ali je zadržao ime LLVM iz istorijskih razloga [14].

[1] Pod konfigurabilnim kompajlerom se misli da se već postojeći mehanizmi mogu iskoristiti za prevođenje novog jezika.

3.1 Struktura i organizacija LLVM-a

Projekat LLVM u sebi okuplja desetak drugih projekata pod svojim imenom. Svi oni su pod licencom GPL (eng. General Public License). Neki od njih nisu potrebni kompajleru za njegov rad ali predstavljaju skup korisnih alata i biblioteka. Neki od najprepoznatljivijih podprojekata projekta LLVM su [24]:

Clang - Prednji deo kompajlera kompajlera koji prevodi C-olike jezike u srednju reprezentaciju. On je najprepoznatljiviji podprojekat projekta LLVM. Deo je paketnih sistema za skidanje programa na većini BSD (eng. Berkeley Software Distribution) i GNU/Linux distribucija. Često se Clang spominje kao kompajler ne uzimajući u obzir da je on samo prednji deo kompajlera i da se ispod njega nalazi moćan optimizator LLVM core.

LLVM core - Predstavlja jezgro projekta LLVM. To je zadnji deo kompajlera koji vrši optimizaciju koda počevši od srednje reprezentacije koja se prevodi u asembler ili mašinski jezik. Da bi primenili optimizacije napisane u okviru LLVM-ovog zadnjeg dela dovoljno je napisati samo prednji deo za neki novi jezik.

DragonEgg - Dodatak za GCC koji omogućava pravljenje srednje reprezentacije za druge više programske jezike koji su podržani GCC-om (npr. Ada i Fortran). Dodatno on zamenjuje GCC optimizator sa LLVM jezgrom. Radi za verzije GCC-a veće od 4.5 i samo za x86 i ARM procesorske arhitekture.

libc++ i libc++ ABI - Optimizovana implementacija standardne C++ biblioteke kao i potpuna podrška za c++11. Jedan od razloga za pisanje nove biblioteke libc++ je taj što se tokom razvoja standardne biblioteke (pre postojanja ovog projekta) i njenog korišćenja došlo do novih ideja o implementaciji standardnih kolekcija (eng. *containers*) koje su zahtevale promenu ABI-ja kao i fudamentalne izmene u njihovoj implementaciji. Takođe, razvoj libstc++ je tesno vezan sa razvojem G++ jer je za igradnju novijih verzija libstc++ potrebna približno vodeća verzija G++. Ovo su samo neki od mnogobrojnih razloga za pokretanje ovog projekta [12].

compiler-rt - Skup biblioteka koje se koriste tokom rada programa (eng. *runtime - rt*). One imaju za cilj da obezbede efikasnu implementaciju operacija niskog nivoa koje nisu podržane hardverskim instrukcijama. Pored toga, sadrži i biblioteke koje pružaju podršku sanitajzerima (eng. *sanitizer*) za adrese, niti, memoriju i protok podataka (eng. *data flow*). Sanitajzeri su alati koji pomoću instrumentalizacije pokušavaju da otkriju složenije nepravilnosti u radu programa.

lld - Linker zasnovan na bibliotekama projekat LLVM. Cilj ovog projekta je da se napravi linker koji će biti brži i bolji od postojećeg sistemskog linkera ld koji se trenutno

koristi kao linker u projektu LLVM. Ovaj projekat je i dalje u razvoju. Ukoliko ispuni očekivanja lld će postati zvaničan linker projekta LLVM.

lldb - Debager zasnovan na bibliotekama projekta LLVM. Pruža samo opciju za debagovanje programa koji imaju istu arhitekturu kao i debager. Dosta je brži i memorijski je efikasniji prilikom učitavanja od GDB-a.

Clang, LLVM core, lld učestvuju u izgradnji programa kao posebni i nezavisni delovi. Oni mogu biti ulančani pomoću kompajlera Clang kako bi se ceo posao automatizivao a mogu se i posebno pozivati [14]. Linkovanje biblioteka je takođe automatizovano. Zastavice u komandnoj liniji za kompajliranje naznačavaju koje bibliotekte Clang treba da linkuje. Dobar i pažljiv dizajn ovih komponenti je neophodan kako bi se sve one povezale i efikasno iskoristile napisan kod. Ono što omogućava njihovo lakše povezivanje i maksimalnu iskorišćenost već napisanog koda je LLVM-ova filozofija da se svaka celina, funkcionalnost ili hijerarhija klasa predstavi bibliotekom tako da sav kôd može biti ponovo upotrebljen.

3.2 Prednji deo kompajlera

Alat koji prevodi programski jezik u srednju reprezentaciju predstavlja prednji deo kompajlera. On treba da obuhvati leksičku, sintaksnu i semantičku analizu koda, da ga dopuni u zavisnosti od izabranih opcija i na kraju proizvode srednju reprezentaciju. Tri zvanična prednja dela LLVM-a su llvm-gcc, DragonEgg i Clang.

U prvoj verziji LLVM-a za generisanje srednje reprezentacije se koristila modifikovana verzija GCC-a zvana llvm-gcc. Da bi se koristio ovaj alat bilo je neophodno skinuti i izgraditi ceo modifikovan gcc. Ovo je bilo vremenski zahtevno i komplikovano. DragonEgg projekat je nastao kako bi se izbegla potreba za izgradnjom celog GCC. Umesto toga DragonEgg predstavlja dodatak koji se uključuje u čist GCC. Kako ovaj projekat predstavlja poseban deo potrebno je samo njega održavati i izgraditi [14].

Zvanični prednji deo LLVM-a je Clang koji predstavlja prednji deo za C-olike jezike C\C++\Objective-C. Sve kompajlerske opcije koje postoje u GCC-u su potpuno podržane i u kompajleru Clang. Clang je dizajniran tako da bude memorijski efikasan, da ispisuje jasne, pouzdane i korisne poruke o greškama i upozorenjima. Takođe on pruža čist API[2], tako da se on može koristiti u drugim projektima (najčešće kao sintaksni i semantički analizator u okviru nekog inteligentnog alata za pisanje koda) [19].

[2]Kao što smo već napomenuli ceo sistem LLVM-a je organizovan kao skup biblioteka, pa tako imamo i libclang koju treba da se uključi ukoliko želimo da korisimo API kompajlera Clang.

3.3 Srednja reprezentacija

Srednja reprezentacija projekta LLVM je kičma koja spaja prednji i zadnji deo kompajlera, omogućavajući LLVM-u da obrađuje više programskih jezika i da generiše mašinski kôd za više platformi. Nad srednjom reprezentacijom se izvršava najveći broj platformski nezavisnih optimizacija [14].

Napomenuli smo da je srednja reprezentacija projekta LLVM ono što razlikuje ovaj sistem od drugih kompajlerskih sistema. Naime svaki kompajlerski sistem poseduje svoju srednju reprezentaciju ali se svi oni razlikuju po tome koliko je ta reprezentacija bliska orginalnom kodu višeg programskog jezika, odnosno koliko su instrukcije koje se upotrebljavaju bliske procesorskim instrukcijama za koji se program kompajlira. Viši nivo reprezentacije omogućava da se bolje uoče i optimizuju originalne namere koda, dok sa druge strane što je reprezentacija bliže hardverskim instrkucijama to se program bolje prilagođava ciljanoj arhitekturi. Niži nivoi pružaju više informacija, ujedno ostavljaju više prostora za optimizacije, ali samim tim je teže mapirati/preslikati delove reprezentacije na delove izvornog koda [14].

Kako infrastruktura LLVM predstavlja kompajler za više arhitektura tako postoji potreba za generisanjem mašinskih kodova za različite arhitekture. Ono što olakšava ovaj posao je njegova zajednička srednja reprezentacija za svaku arhitekturu. Ovakva reprezentacija omogućava izvršavanje optimizacija koje su zajedničke za sve arhitekture. Ona je na dovoljno niskom nivou da može da predstavi bilo koji viši jezik, pruži informacije za optimizacije, ujedno ne favorizujući ni jednu konkretnu arhitekturu. Iako srednja reprezentacija ima neke arhitekturalno zavisne delove njen veliki deo je zajednički i ona predstavlja tačku za koju različiti zadnji delovi kompajlera imaju isto razumevanje. Na osnovu nje zadnji deo kompajlera će napraviti svoju reprezentaciju koda, na kojoj će se izvršiti arhitekturalno zavisne optimizacije i generisati mašinski kôd za ciljanu arhitekturu [14].

Instrukcijski skup i memorijski model srednje reprezentacije je malo složeniji od asemblera. Instrukcije su slične RISC[3] ali sa ključnim dodatnim informacijama koje omogućavaju efektivne analize. Tipovi koji se koriste su na dovoljno niskom nivou da mogu implementirati bilo koji tip ili operaciju nekog jezika, ujedno ostavljajući informacije o njima u svim fazama optimizacije. Najvažnije jezičke osobine srednje reprezentacije su [14]:

1. Poseduje svojstvo jednog statičkog dodeljivanja (eng. *Static Single Assignment - SSA*) [**SSA**] koje omogućava lakšu i efikasniju optimizaciju koda. Ovo svojstvo označava da se prilikom korišćenja operatora dodele proračunata vrednost sa desne strane uvek dodeljuje novoj promenljivoj. Promenljiva čija vrednost zavisi od nekog uslova je predstavljena posebnom ϕ - „phi” instrukcijom [**SSA**].

[3] Reduced Instruction Set Computing.

2. Instrukcije za obradu podataka su troadresne. Imaju dva argumenta i rezultat stavljaju na treću lokaciju.

3. Poseduje neograničen broj registara. Posao optimizacije je da najbolje raspodeli registre tako da oni budu što efikasnije iskorišćeni.

Ove osobine reprezentacije možemo uočiti iz primera %foo = add i32 %0, %1. Ovaj primer će sabrati vrednosti lokalnih vrednosti %0 i %1 i smestiti ih u lokalnu promenljivu %foo. Lokalne promenljive su analogne registrima u asembleru i mogu imati bilo koje ime koje počinje sa %.

Način na koji se čuva srednja reprezentacija je od izuzetnog značaja, kako iz ugla LLVM-a tako i sa akademskog ugla. Ona može biti čuvana u memoriji pomoću struktura i klasa koje je predstavljaju, na disku u vidu enkodirane bitkod[4] datoteke, na disku u ljudski čitljivom obliku. Sva tri načina su ekvivalentna. Mogućnost zapisa na disku u ljudski čitljivom obliku je od akademskog i praktičnog značaja jer on predstavlja osnovu za izučavanje i primenjivanje optimizacija tokom svih faza postojanja programa (faze izgradnje, linkovanja, instalacije, rada i između dva pokretanja programa).

Sadržaj jedne LLVM datoteke predstavlja modul. Modul obično predstavlja jednu prevedenu datoteku izvornog koda i on je struktura najvišeg nivoa. On sadrži sekvencu funkcija, koje sadrže sekvencu bazičnih blokova koji sadrže sekvence instrukcija. Pored toga on sadrži i informacije o tipovima podataka ciljne arhitekture, globalnim promenljivama, prototipe eksternih funkcija kao i definicije deklarisanih struktura [14].

3.4 Zadnji deo kompajlera

Rezultat obrade izvornog koda prednjim delom kompajlera je srednja reprezentacija. Ona se predaje zadnjem delu kompajlera koji ima za krajnji cilj da je prevede u asemblerski kôd ili mašinski kôd za ciljanu arhitekturu. Najpre se vrše optimizacije nad srednjom reprezentacijom. Sve one su arhitekturalno nezavisne i zajedničke su za sve arhitekture i izvršavaju se u vidu prolaza (eng. *pass*). Prolaz se izvršava nad srednjom reprezentacijom, obrađuje je, analizira, prepoznaje moguće optimizacije i na kraju je menja u cilju stvaranja optimizovanijeg kôda. Prolaz može zavisiti od nekog drugog prolaza, radi prikupljanja podataka i stvaranja heuristike kako bi se odredili dalji tokovi optimizacije. Takođe on može zahtevati od drugog prolaza da mu pripremi srednju reprezentaciju za drugi prolaz [14].

Nakon optimizacije srednje reprezentacije, potrebno ju je prevesti u mašinske instrukcije. Arhitekturalno nezavisni interfejs za generisanje mašinskog koda dodaje sloj ap-

[4] Ovaj naziv je nastao kao pandan javinom bajtkodu jer LLVM-ova srednja reprezentacija na nižem nivou od Javine reprezentacije.

strakcije koji pomaže prilikom ove transformacije. Svaka arhitektura[5] zahteva postojanje generičke klase koja treba da implementira i specijalizuje ovaj interfejs. Tokom procesa ove transformacije a i nakon njenog završetka vrše se optimizacije koje su specifične za arhitekturu za koju se kôd kompajlira [14].

Postupak prevođenja srednje reprezentacije na konkretnu arhitekturu započinje superprolazom[6] selekcije instrukcija (eng. *instruction selection*). On prevodi srednju reprezentaciju prvo u SelectionDAG, direktni aciklični graf (DAG), u kome svaki čvor najpre predstavlja instrukciju srednje reprezentacije, a grane redosled instrukcija. Bilo kojim DAG-om moguće je predstaviti izvršavanje bilo kog bloka instrukcija. Svaki čvor grafa se propušta kroz legalizacije tipova i instrukcija kao i kroz proces spuštanja (eng. *lowering*). Legalizacija obezbeđuje da su svi tipovi i sve instrukcije koje se koriste podržane od strane hardvera za koji se prevodi (ako arhitektura podržava samo 32-bitno sabiranje a nama je potrebno 64-bitno ova operacija se razbija na dve manje), dok proces spuštanja određen skup čvorova (kao na primer simbol za povratak iz funkcije) odmah prevodi u čvorove koji predstavljaju mašinske instrukcije.

Preostali čvorovi koji predstavljaju instrukcije srednje reprezentacije se mapiraju u čvorove mašinskih instrukcija. Mapiranje se najčeće vrši na osnovu kôda koji se automatski genersiše pomoću llvm-tablegen alata projekta LLVM. Ovaj alat generiše kôd iz datoteke sa ekstenzijom .td koja označava „opis ciljne arhitekture" (eng. *target description - td*). Kako što mu ekstenzija kaže, ova datoteka sadrzi deklarativne slogove za jednu arhitekturu koji mogu opisivati uzorke za mapiranje instrukcija srednje reprezentacije u mašinske instrukcije, funkcije za kodiranje/dekodiranje instrukcija, arhitekturalne registre i njihove osobine itd. Nakon propuštanja .td datoteka kroz llvm-tablegen alat dobijaju se .inc datoteke koje se uključuju u kôd sa #include direktivom. Međutim uzorci za mapiranje ne opisuju neke složene instrukcije. Za takve instukcije potrebno je napisati metodu koja ju je opisuje. Na kraju ovog prolaza imamo graf gde je svaki čvor arhitekturalno zavisan i zapisuje instrukciju direktno podržanu od strane ciljanog procesora [21, 14].

Sledeći korak je planiranje redosleda instrukcija (eng. *instruction scheduling*) ili još poznat kao raspoređivanje instrukcija pre alokacije registara (eng. *Pre-register Allocation Scheduling*) [14]. Kako računari izvršavaju linearan niz instrukcija, potrebno je prethodno stanje gde su instrukcije zapisane u grafu prevesti u lineran niz troadresnih instrukcija. Čvorovi iz grafa se najpre topološki sortiraju. Dobijeni raspored instrukcija može prouzrokovati usporavanje i potrebno ga je optimizovati kako bi se maksimalno iskoritio paralelizam na nivou instrukcija. Ovakvo sortiranje je svojstveno svakoj arhitekturi jer svaka ima svoj skup registara kao i instrukcija koje se mogu paralelno izvršavati a samim

[5]Trenutno podržane arhitekture su AArch64, AMDGPU, ARM, AVR, BPF, Hexagon, Lanai, MSP430, MIPS, NVPTX, PowerPC, RISCV, SPARC, SystemZ, X86, i XCore.

[6]Prolaz koji u sebi sadrži više manjih prolaza.

tim i različite heuristike prilikom raspoređivanja instrukcija. Kada se raspoređivanje završi čvorovi se emituju u novi niz troadresnih instrukcija (class MachineInstr) a DAG se uništava. Elementi novonastalog niza pored samih instrukcija i njihovih adresa sadrži i dodatne informacije kao što su tip instrukcije, da li su mu operandi u memoriji ili registru itd. Dodatne informacije koje nose ovi objekti su neophodni za neke naredne faze optimizacije i analize [21].

Trenutno stanje instrukcija zadovoljava svojstvo jednog statičkog dodeljivanja i dalje zavisi od neograničenog broja registara. Zadatak sledećeg superprolaza dodeljivanja registara (eng. *Register Allocation*) je da razbije pomenuto svojstvo i da dodeli virtuelne registre fizičkim, čiji broj je konačan i različit za svaku arhitekturu. Neki registri mogu već biti dodeljeni u prethodnom prolazu prilikom pravljenja instrukcijskih poziva čije operacije očekuju da rezultat bude upisan u određeni registar ili zbog potreba binarnog aplikativnog interfejsa (eng. *Aplication Binary Interface - ABI*). Postoji više algoritama za dodeljivanje registara. Cilj svakog algoritma je da ograničen broj fizičkih registara što efikasnije dodeli i da se utroši što manje memorijskih lokacija. Prilikom rešavanja ovog problema analiziraju se intervali vrednosti, kao i cene rasipanja promenljivih na memorijske lokacije. Interval vrednosti je predstavljen krajnjom i početnom tačkom postojanja proizvedene vrednosti. Vrednost može biti čuvana na privremenoj lokaciji dok se ponovo ne upotrebi, odnosno uništi. Nakon što su dodeljeni registri pokreće se raspoređivanje instrukcija posle alokacije registara (eng. *Post-register Allocation Scheduling*). Informacije o tipovima registara se koriste kako bi dodatno unapredio raspored instrukcija [21, 14].

Na kraju kada imamo mašinske instrukcije, sledeći korak je da ih zapišemo na zeljenu lokaciju u odgovarajućem formatu. LLVM ih može zapisati direktno u memoriju pomoću JIT (Just-in-Time), ili preko *Machine Code* (MC) apstraktnog radnog sloja (eng. *framework*) (ARS). Na MC ARS su zasvnovani asemblerski i disasemblerski alati projekta LLVM za čitanje i tumačenje binarnih datoteka. Pre postojanja MC ARS-a LLVM je proizvodio asemblersku datoteku i zavisio je od spoljašnjeg alata da ga prevede u objektnu datoteku. Sada ovaj ARS omogućava upisivanje kako u asemblersku datoteku tako i u objektu.

Proces optimizacije i pravljenje binarne datoteke nisu od značaja za rad koliko Clang, srednja reprezentacija i podprojekat compiler-rt, to jest compiler-rt biblioteka za profajliranje. Clang tokom pravljenja srednje reprezentacije, ukoliko je prilikom kompajliranja postavljena zastavica za profajliranje, ubacuje dodatan kôd u srednju reprezentaciju koji predstavlja brojače. Detaljniji opis će biti prikazan u 6 dok će u narednom delu će biti prikazana koncept profajliranja i tehnika koja je primenjena u LLVM-u.

Glava 4

Profajliranje

Proces optimizacije programa predstavlja neizostavan deo razvoja softvera. Da bi se optimizacija uspešno sprovela, neophodno je korišćenje pomoćnih alata koji olakšavaju razumevanje i poboljšavanje koda. Postoji veliki broj različitih pomoćnih alata koji su razvijeni u te svrhe. Pokretanjem programa nekim od ovih alata dobijamo dodatne informacije pomoću kojih stičemo bolju sliku o funkcionisanju programa koji analiziramo kao i o njegovim nedostacima.

U zavisnosti od toga da li podatke o programu dobijamo izvršavajući ga ili ne, analizu koda možemo podeliti na dinamičku i statičku. Statička analiza [1] predstavlja analiziranje programa bez njegovog izvršavanja. Najčešća upotreba statičke analize je prilikom kompilacije i njena uloga je da pomogne kompajleru prilikom donošenja nekih odluka. Podaci dobijeni ovim pristupom ne zavise od ulaza, mogu da budu neprecizni i često predstavljaju samo predviđeno ponašanje programa. Budući da je za dalje razumevanje teme rada od izuzetnog značaja shvatanje dinamičke analize, u daljem tekstu ćemo detaljnije opisati njen način funkcionisanja.

Profajliranje predstavlja vid dinamičke analize koda čiji je rezultat skup podataka dobijen izvršavanjem programa sa određenim ulaznim podacima u određenom trenutku. Profajliranje se može posmatrati i kao ubacivanje dodatnih instrukcija u program kako bi se prikupili podaci o programu za vreme njegovog izvršavanja. Drugačije rečeno, pravi se profil posmatranog programa. Dobijeni podaci nam mogu pomoći u otkrivanju „uskog grla" (to jest dela koda koji se često izvršava), curenja memorije koje treba popraviti, određivanju pokrivenosti koda datim ulazima, proširivanju skupa testova i još mnogih drugih problema.

Merenja se najčešće odnose na broj izvšavanja instrukcija određenog dela koda ili vreme provedeno u njemu. Za neka merenja neophodna je hardverska podrška (na primer, za merenje broja promašaja u keš memoriji ili za merenje utrošenog vremena nastalo zbog čekanja protočne obrade (eng. *pipline stall*) neke instrukcije).

Važno je napomenuti da tokom programa upravljaju konkretni ulazi i da različiti ulazi daju različite rezultate profajliranja. Samim tim, konkretnoj klasi ulaza odgovaraju slični

rezultati. Možemo zaključiti da treba pažljivo birati skup ulaza tako da oni pokrivaju određene delove koda.

4.1 Instrumentalizacija

Ubacivanja novog koda u već napisan program naziva se instrumentalizacija (eng. *instrumentation*). Istrumentalizacija se može izvršiti u različitim fazama. Takođe ona se može podeliti i na osnovu toga kako se ona ubacuje u program. Nju može izvršiti sam programer (manuelno dodavanjem dodatnih linija na željena mesta u kodu ili automatski pomoću alata koji pravi verziju izvornog fajla sa dodatim kodom za brojače), kompajler, linker, može se ubaciti u već iskompajlirani program interpretacijom izvršne datoteke [2, 9] ili čak za vreme izvršavanja programa [20].

Tri najvažnije osobine koje dobra instrumentacija treba da zadovolji su:

1. Prikuplja samo potrebne podatke.

2. Ne utiče na funkcionalnost programa.

3. Ne usporava previše rad programa.

Prvi uslov je veoma bitan. Zahtev za previše podataka dodatno usporava program i samu njihovu obradu, dok premalo informacija može biti beznačajno. Drugi uslov ističe da ukoliko dodata instrumentalizacija utiče na funkcionalnost programa onda prikupljeni podaci neće oslikavati pravi način njegovog rada. Poslednji uslov zavisi od tipa aplikacije i njega možemo kontrolisati u zavisnosti od granulacije, to jest od delova programa koji se instrumentalizuju.

Na osnovu mesta gde se ubacuje dodatni kôd imamo tri vrste profajliranja:

1. Profajliranje putanje (eng. *path profiling*).

2. Profajliranje ivica (eng. *edge profiling*).

3. Profajliranje bloka (eng. *basic–block profiling*).

Profajliranje putanja ima velike primene prilikom poboljšavanja performansi programa, kompilaciji programa zasnovanoj na podacima dobijenim profajliranjem, proveri pokrivenosti koda testom itd. Ukoliko je čitalac zainteresovan može potražiti dalje reference i upoznati sa osnovana profajiranja putanja u drugim radovima [26, 7].

U narednom delu ovog poglavlja biće prikazan algoritam za profajliranje ivica i blokova jer je on potreban kako bi se bolje razumela pozadina rada. Takođe će biti prikazana i dva načina za smanjivanje troškova instrumenalizacije dvema tehnikama uzimanja uzorka.

4.2 Profajliranje ivica i blokova

U ovom odeljku pažnja će biti usmerena na problem dobijanja statističkih podataka o blokovima i ivicama programa koji su se izvršili. Blokovi mogu biti funkcije[1] ili deo koda u kome se ne nalaze instrukcije grananja ili skokova. Ivica predstavlja instrukciju granaja ili skoka kojom se prebacuje tok izvršavanja programa iz jednog bloka u drugi. Ona povezuje dva bloka. Da bi brojali koliko puta se izvšilo određen blok moramo da ubacimo kôd koji će vršiti uvecanje brojača.

Na početku se nećemo fokusirati na to kako kôd brojača radi, ali još jednom ističemo da on uvodi određene troškove. Najjednostavniji način bi bio da za svaki blok ili ivicu ubacimo brojač. Tada bismo imali neophodne podatke, ali bi oni zauzeli previše računarskih resursa.

Algoritam optimalnog instrumentalizovanja ivica koji ćemo prikazati ima minimalni broj brojača umetnut u kôd i on se koristi za instrumentalizaciju u okviru projekta LLVM. On računa broj izvršavanja svakog bloka pomoću brojača ivica tako što sumira sve ivice koje ulaze u blok. Ovo rešenje prvi je teoretski uveo Donald Knut (eng. *Donald E. Knuth*) u [10] gde je on pokazao da je broj umetnutih brojača zaista minimalan. Knutovo rešenje može istrumentalizovati isti broj ivica ali na različite načine. Bal (eng. *Tomas Ball*) i Larus (eng. *James R. Larus*) su prikazali [8] način kako da se proceni koji skup ivica je najoptimalniji. Koristili su jednostavnu statičku analizu programa kako bi instrumentalizovali one ivice za koje se predviđa da će se najmanje puta izvršiti. Prikaz algoritma je preuzet iz [19].

4.3 Izračunavanje broja izvršavanja ivica

Prva stvar koja treba da se napravi je graf kontrole toka (eng. *control flow graph*), u kome svaki čvor predstavlja blok instrukcija, a grana naredbu skoka ili grananja. Najviša tačka predstavlja početak profajlirane procedure. Iz ovakvo dobijenog grafa potrebno je izbaciti sve ivice koje prave ciklus tako da dobijemo razapinjuće stablo (eng. *spanning tree*). Ivicama koje ne pripadaju dobijenom stablu treba dodati brojač. Broj izvršavanja ivica koje ne sadrže brojač se može izračunati drugim delom algoritma koji je opisan u narednom pasusu.

Standardni algoritam računanja razapinjućeg stabla može vratiti različita stabla u zavisnosti od redosleda obrade grana. Kako bisom odabali optimalno razapinjuće stablo (ono stablo koje sadrži najviše ivica iz početnog grafa), na osnovu postojećih informacija, vrši se statička analiza grafa kontrole toka. Ovaj postupak predstavlja prvi deo algoritma 1. Broj instrumentalizovanih ivica odgovara minimalanom broju ciklusa u grafu kontrole toka.

[1] Ako funkcije posmatramo kao blokove, možemo napraviti profil i pomoću alata gprof [6].

Algoritam 2 prikazuje drugi deo algoritma koji se primenjuje nakon izvršavanja programa kako bi se izračunao broj izvšavanja neinstrumentalizovanih ivica. Postupak se sastoji od računanja broja izvršavanja nekog od listova razapinjućeg stabla pomoću već izračunatih susednih ivica. Nakon toga se taj list izbacuje iz stabla. Ova procedura se ponavlja dokle god ima listova u stablu, tj. dok broj izvršavanja svake ivice ne bude izračunat. Drugi način izračunavanja broja izvršavanja neinstrumentalizovanih ivica preko matrica dat je u [26].

Algoritam 1 OptimalnoUbacivanjeIvica(P) $\rightarrow P_i$

```
array of counters C in P
index := 0
for each function f in program P do
    // izračunavanje razapinjućeg stabla za f
    ST := ∅
    for each edge e in function f do
        if adding e to ST does not create a cycle in ST then
            ST := {e} ∪ ST
        end if
    end for
    // dodavanje brojača u P
    for each edge e in function f do
        if e ∉ ST then
            add code to P that initialises C[index] with 0
            add code to P such that {C[index]++} is executed when e is traversed
        else
            add code to P that initialises C[index] with -1
        end if
        index++
    end for
    add code to P that writes counter array to file at end of execution of P
end for
```

Algoritam 2 IzračunavanjePreostalihIvica($P, Profile$) $\rightarrow W$

read array C from $Profile$
$index := 0$
for each function f in program P **do**
  // *učitavanje podataka dobijenih profajliranjem*
  for each edge e in function f **do**
    $w_e = C[index]$
    $index{+}{+}$
    // *ako za ivicu ne postoji brojač, dodati je u otvoren skup*
    if $w_e == -1$ **then**
      $O := \{e\} \cup O$
      w_e := -1
    end if
  end for
  // *preračunati brojače za ivice iz otvorenog skupa*
  while $|O| > 0$ **do**
    for each $e \in O$ **do**
      if either end of e has no adjecent edges in O **then**
        calculate w_e weights of adjecent edges
      end if
    end for
  end while
end for

Kako bi se upotpunio algoritam i obezbedili njegovo korektno funkcionisanje potrebno je prilikom izgradnje grafa kontrole toka dodati virtualne ivice i blokove. Oni ne postoje u sistemu ali ih je potrebno dodati kako bi se obezbedilo da graf kontrole toka ima ciklus i u slučaju kada funkcija sadrži samo jedan blok. U tu svrhu dodaju se dve virtuelne tačke i nekoliko virtuelnih ivica. Jedna virtuelna ivica povezuje početnu virtuelnu tačku i početak. Ostale virtuelne ivice treba da povezuju izlazne blokove funkcije sa izlaznom virtuelnom tačkom. Na kraju treba napraviti još jednu virtuelnu ivicu od početne virtuelne tačke do izlazne virtuelne tačke kako bi se pokrio slučaj kada funkcija ima jedan blok i kako bi se obezbedilo da uvek postoji ciklus [19].

Broj ivica u razapinjućem stablu je jednak $|v| - 1$, gde $|v|$ označava broj čvorova u grafu. Kako naš graf ima dodatne dve virtuelne tačke pravi broj ivica maksimalnog razapinjućeg stabla je $|v_v| = |v| + 1$. Uzimajući da je ukupan broj ivica $|e|$, uključujući i virtuelne, jednak, dobijamo da je ukupan broj instrumentalizovanih ivica $|e_i| = |e_v| - |v_v|$.

4.4 Profajliranje uzimanjem uzorka

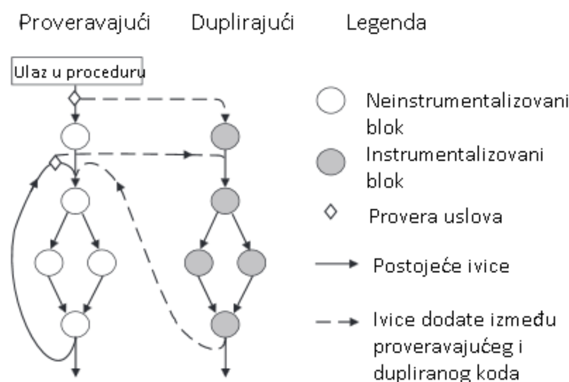
Postizanje pravog poretka između vremena utrošenog prilikom profajliranja i same dobiti na osnovu dobijenih profila je od izuzetne važnosti. Profajliranjem se u prvim koracima dobijaju veliki rezultati, a kako se sve više optimizuje program to ga je teže

unaprediti, a zahtevi za profajliranjem postaju sve veći. Zbog toga je potrebno smanjiti usporavanje programa instrumentalizacijom. Jedan od efikasnih načina koji znatno smanjuje opterećivanje programa je uzimanjem uzorka (eng. *sampling*). Važno je napomenuti da ova tehnika ne određuje način instrumentalizacije, već samo kako smanjiti njene troškove.

Uzorkovanje (eng. *sample based profiling*) posmatra deo izvršavajućeg programa. Ono uzima „slike" programa u određenim vremenskim intervalima i od njih pravi profil programa. Rezultat ovakvog pristupa je znatno manje opterećenje programa, ali kao posledicu ima smanjenu tačnost dobijenih podataka. Za dobro izabrane intervale merenja nepreciznost ove metode nije toliko velika (ispod 10%) da može da utiče na loše tumačenje podataka. Međutim, ukoliko vremenski intervali nisu dobro odabrani može se desiti da se ne zabeleži određeni događaj. Zbog tog razloga ona se najčešće koristi za pronalaženje najfrekventnijih događaja. Algoritam koji ćemo dati predstavlja jedan od osnovnih algoritama za uzorkovanje i on je detaljnije objašnjen u radu [16]. Ovaj pristup predstavlja osnovu i ima svoje mane koje su predočili Martin Hirzel (eng. *Martin Hirzel*) i Trišul Čilambi (eng. *Trishul Chilimbi*) i dali predlog njegovog poboljšanja [15].

4.5 Algoritam uzorkovanja

U okviru algoritma uzorkovanja, pored osnovnog koda pravimo njegov duplikat koji će sadržati instrumentalizovani kôd i njega ćemo nazvati duplirani kod. Orginalni kôd ćemo nazvati proveravajući zato što ćemo u njemu ispitivati uslov koji, ukoliko je ispunjen, treba kontrolisano da pređe u duplirani kod. Ovim uslovom kontrolišemo koliko vremena će se izvršavati svaki od ova dva koda. Kada izvršavanje pređe u duplirani kôd ono tu ostaje ograničeno vreme a zatim se vraća u proveravajući. Opisani postupak prikazan je slikom 4.1.



Slika 4.1: Ilustracija toka izvršavanja proveravajućeg i duplirajućeg koda

Da bi se obezbedila tačnost dobijenih uzoraka oni moraju biti uzimani u statistički

značajnim trenucima. Ovaj trenutak se može inicirati hardverski, putem operativnog sistema ili softverski. Kako se ne može računati na postojanje istog hardvera za sve sisteme prikazaćemo softversko rešenje.

```
if (globalniBrojac <= 0) {
  napraviUzorak();
  globalniBrojac = inicijalnaVrednost;
}
globalniBrojac--;
```

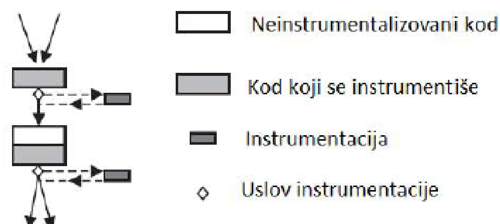
Listing 4.1: Uzorkovanje zasnovano na brojačkoj promenljivoj

Kod sa listinga 4.1 predstavlja uzorkovanje zasnovano na brojanju koje broji koliko puta se određeni događaj izvršio i ukoliko dođe do zadate granice vrši se uzorkovanje. Kôd sa listinga 4.1 može ubaciti kompajler u duplirani kôd i kao rezultat toga promenljiva *globalniBrojač* će najverovatnije biti u nekom od registara procesora.

Za višenitne aplikacije nesinhronizovani brojač može predstavljati problem. Međutim, još veći problem bi predstavljalo sinhronizovano korišćenje ove promenljive jer bi ono dodatno usporilo program. Ako bi ovu promenljivu ostavili nesnihronizovanu izgubili bismo samo na preciznosti brojača koji okida događaj uzorkovanja, ali to je u ovom slučaju zanemarljivo jer će se uzorkovanje izvršavati u dovoljno sličnim intervalima. Prednost ovakvog pristupa je što imamo mogućnost velike frekvencije uzorkovanja.

Prikazani algoritam povećava potrebnu memoriju kao i vreme kompilacije, za koje ne mora da znači da je dvostruko povećano. Neki sistemi instrumentalizuju samo najkorišćenije metode proveravajući ih sve zajedno ili svaku posebno. Ako je prostor ograničen, broj metoda koje se ispituju je proporcijalno manji.

Postoje dve proširene verzije ovog algoritma koje ne prave celu kopiju koda već samo onih delova koji su vezani za instrumentalizaciju. Jedno rešenje zadržava isti broj provera uslova prelaska, brišući blokove koji nemaju uticaj na instrumentalizovane blokove, dok drugo, predstavljeno slikom 4.2, dodaje samo instrumentalizovan kôd uz dodavanje potrebnih provera uslova ograđujući sve operacije instrumentalizacije. Mana drugog pristupa je što postoji veliki broj proveravanja uslova. Kombinacijom ove dve varijante, u zavisnosti od cilja profajliranja, moguće je optimizovati uzorkovanje.



Slika 4.2: Neduplirano uzorkovanje.

4.6 Minimizacija uticaja na performanse

Iako postoje mnogi automatizovani metodi koji se koriste za automatsku intrumentalizaciju programa, postoje određene situacije gde ih ne možemo olako koristiti. Na primer, sistemi u realnom vremenu imaju vrlo stroga vremenska ograničenja, koja ako se prekrše mogu izazvati štetu ili čak ugroziti ljudske živote. Drugi problem predstavljaju i memorijska ograničenja uređaja. Dodatni kôd za instrumentalizaciju i njeno rukovanje može uvećati program tako da on ne može da stane na uređaj.

Kompajlerski zasnovano profajliranje[2] pruža brz, jednostavan i efikasan način profajliranja koda. U ovom delu je priložen algoritam instrumentalizacije sa minimalnim brojem brojača koji je implementiran kao deo podrške za analizu programa u okviru kompajlera Clang, čije će unapređeno rešenje za upravljanje prikupljenim podacima biti prikazano u poglavlju 6. Poglavlje 5 treba da prikaže nadogranju formata profajliranja kompajlera Clang kako bi iz njega mogli da izvedemo ljudski čitljiv format koji koristi GCC. Sama nadogranja kompajlera Clang u cilju stvaranja zajedničkog formata kompajlera Clang i GCC dodaje informaciju o broju izvršavanja svake grane instrukcije grananja.

[2]Instrumentalizaciju vrši kompajler.

Glava 5

Zajednički format izlaza profajliranja kompajlera Clang i GCC

Rezultat profajliranja se zapisuje u datoteku. Posebnim alatima se iz ovako nastalih datoteka izvlače informacije koje se predstavljaju u obliku koji je pogodniji za ljude. Za objedinjavanje podataka profajliranja kompajlera Clang i GCC izabran je ljudski čitljiv format koji koristi kompajler GCC. Razlog za ovakav izbor je taj što za njega već postoje alati za lepsi ispis (gcov, lcov). Takođe, postoje softveri koji mogu koristiti oba kompajlera, i GCC i Clang, za prevođenje pa otuda i još jedan dodatan razlog za stvaranje jedinstvenog formata. Ekstenzija ovog formata je *„.info”*. Datoteka u ovom formatu se dobija na osnovu „.gcda” i „.gcno” datoteka pomoću alata *geninfo*.

Tekstualni format za zapisivanje sirovih podataka profajliranja koji koristi kompajler GCC, započinje sa imenom testa koje sa zadaje sa opcijom „-t <ime testa> ” prilikom generisanja datotke alatom *geninfo*.

TN: <ime testa>

Za svaki izvorni fajl koji je instrumentalizovan postoji deo koji najpre navodi njegovu putanju a zatim podatke koji opisuju kako se izvršio navedeni fajl.

SF:<apsolutna putanja do izvornog fajla>

Nakon ovoga se zapisuje broj početne linije svake pronađene funkcije u navedenom izvornom kodu.

FN:<broj početne linije funkcije>,<ime funkcije>

Zatim sledi lista broja izvršavanja prethodno navedenih instrumentalizovanih funkcija.

FNDA:<broj izvršavanja>,<ime funkcije>

Ovu listu prati sumirani izveštaj za funkcije. Najpre ide broj pronađenih a zatim broj izvršenih funkcija.

FNF:<broj pronađenih funkcija>

FNH:<broj izvršenih funkcija>

Informacije o grananju su navedene tako da svakoj izvršenoj grani odgovara broj linije gde je grananje nastalo i broj izvršavanja te grane.

BRDA:<broj linije grananja>,<broj bloka>,<broj grane>,<broj izvršavanja>

Broj bloka i broj grane su interni identifikatori u okviru kompajlera. Kao i za funkcije tako i za grananja postoje sumarni podaci.

BRF:<broj pronađenih grana>

BRH:<broj izvršenih grana>

Posle informacija o grananjima sledi lista koja prikazuje koliko puta se izvršila svaka linija koda pokrivena instrumentalizacijom.

DA:<broj linije>,<broj izvršavanja linije>

I na kraju idu sumarni podaci o ukupnom broju linija koje su pokrivene instrumentalizacijom i broju linija koje su se bar jednom izvršile.

LH:<broj linija koje su se bar jednom izvršile>

LF:<broj linija pokrivenih instrumentalizacijom>

Na kraju se podaci vezani za jedan izvršni fajl završavaju sa sledećom oznakom:

end_of_record

Primer ispisa u ovom formatu za izvorni kôd sa listinga 5.1 kompajliran kompajlerom GCC dat je slikom 5.7.

Podaci o profilu programa koji je kompajliran i profajliran kompajlerom Clang ne sadrze sve informacije koje su potrebne za ispisivanje „.info" formata, tj. nedostaju informacije o grananjima. U ovom poglavlju će biti prikazano dodavanje lokacije grananja u interni format profajliranja kompajlera Clang tako da se na osnovu tako proširenog formata može dobiti datoteka u „.info" formatu.

5.1 Mapirajući pokrivač u kompajleru Clang

Kompajler Clang pomoću opcije *-fprofile-instr-generate* instrumentalizuje izvorni kod. Pored ove opcije kompajler Clang poseduje opciju *-fcoverage-mapping* koja uključuje dodatne informacije zapisane u formatu mapirajućeg pokrivača (eng. *coverage mapping format*) u objektnu datoteku. Pomoću mapirajućeg pokrivača alati *llvm-cov* i *llvm-profdata* projekta LLVM mapiraju blokove koda sa njihovim brojačima. Da bi se dobili podaci o granananju neophodno je ubaciti dodatni podatak u ovaj format tako da ne narušimo postojeću funkcionalnost. Podatak koji će biti ubačen u ovaj format je broj linije na kojoj se nalazi instrukcija grananja odakle je blok nastao.

Mapirajući regioni

Termin koji se koristi u LLVM dokumentaciji za opisivanje bloka instrukcija i njihovog brojača je mapirajući region. On predstavlja gradivnu jednicu pokrivajućeg formata koja

opisuje svaku funkciju. Jedan mapirajući region određen je identifikacijom datoteke kojoj pripada, opsegom koda kome odgovara, brojačem mapirajućeg regiona i njegovom vrstom. Postoji nekoliko vrsta regiona.

1. Kodovski region (eng. *code region*) odgovaraju delovima izvornog koda i imaju informacije o mapirajućim brojačima. Oni čine većinu mapirajućih regiona. Na slici 5.1 su prikazana tri kodovska regiona.

```
int main(int argc, const char *argv[]) {      // Code Region from 1:40 to 9:2

  if (argc > 1) {                             // Code Region from 3:17 to 5:4
    printf("%s\n", argv[1]);
  } else {                                    // Code Region from 5:10 to 7:4
    printf("\n");
  }
  return 0;
}
```

Slika 5.1: Kodovski regioni.

2. Preskočeni regioni (eng. *skipped region*) su regioni preskočeni predprocesorskom obradom kompajlera. Oni ne sadrže podatke za mapirajuće brojače. Slika 5.2 prikazuje preskočeni region u okviru kodovskog regiona.

```
int main() {                  // Code Region from 1:12 to 6:2
#ifdef DEBUG                  // Skipped Region from 2:1 to 4:2
  printf("Hello world");
#endif
  return 0;
}
```

Slika 5.2: Primer preskočenog regiona.

3. Proširujući regioni (eng. *expansion region*) označavaju delove koda koji treba da se zamene ekspanzijom makroa. Oni poseduju dodatnu informaciju – prošireni identifikator datoteke. Identifikacioni broj datoteke je celobrojna vrednost, tačnije indeks datoteke, koja nam daje informaciju u kojoj izvornoj datoteci ili virtuelnoj datoteci ekspanzije makroa je određeni region lociran. Virtuelna datoteka ekspanzije makroa nam omogućava da razlikujemo istoimeni makro koji se poziva na različitim mestima u kodu, kao na primeru 5.4. Na primeru postoji jedan stvarni indeks datoteke i dva virtualna za svaku ekspanziju makroa. U svakoj virtuelnoj datoteci je smešten po jedan kodovski region. Na slici 5.3 je primer proširujućeg regiona gde je svetlo zelenom bojom prikazan proširujući region a sa svetlo plavom kodovski region u koji se on proširuje. Delovi koda označeni žutom i narandžastom bojom su takođe kodovski regioni.

Mapirajući brojač može biti predstavljen referencom na postojeći brojač ili kao aritmetički izraz nad drugim mapirajućim brojačima ili drugim izrazima. Detaljnije objašnjenje će biti dato kroz primer.

```
int func(int x) {
  #define MAX(x,y) ((x) > (y)? (x) : (y))
  return MAX(x, 42);                        // Expansion Region from 3:10 to 3:13
}
```

Slika 5.3: Primer proširujućeg regiona.

```
void func(const char *str) {     // Code Region from 1:28 to 6:2 with file id 0
  #define PUT printf("%s\n", str) // 2 Code Regions from 2:15 to 2:34 with file ids 1 and 2
  if(*str)
    PUT;                          // Expansion Region from 4:5 to 4:8 with file id 0 that expands a macro with file id 1
  PUT;                            // Expansion Region from 5:3 to 5:6 with file id 0 that expands a macro with file id 2
}
```

Slika 5.4: Primer ekspanzija makroa u različite virtuelne datoteke.

Podaci mapirajućeg pokrivača u srednjoj reprezentaciji projekta LLVM

Srednja reprezentacija, koja sadrži instrumentalizovani kôd, dobija se kada se prilikom kompajliranja, pored zastavica navedenih u delu 5.1, dodaju zastavice *„–S –emit-llvm”*. Rezultat izgradnje izvornog koda sa ovim zastavicama je datoteka LLVM srednje reprezentacije sa *„.ll”* ekstenzijom. Mapirajući pokrivač se dodaje u zaglavlje srednje reprezentacija preko promenljive __llvm_coverage_mapping. Na slici 5.5 je prikazana vrednost ove promenljive izvučena iz srednje reprezentacije za izvorni kôd sa listinga 5.1.

```
1  int foo(int a){
2    return a + 3;
3  }
4
5  int main(){
6    int i = 10;
7    int a = 3;
8    while(i){
9      a++;
10     if(i%2)
11       a = foo(a);
12     i--;
13   }
14   return 0;
15 }
```

Listing 5.1: Primer C datoteke

Promenljiva mapirajućeg pokrivača ima tri strukture u sebi podeljene kao na slici 5.5

1. Zaglavlje mapirajućeg pokrivača

2. Niz podataka o funkcijama

3. Kodirane podatke predstavljene nizom bajtova (sa faktorom poravnanja 8)

```
@__llvm_coverage_mapping = internal constant { { i32, i32, i32, i32 },
                                               [2 x <{ i8*, i32, i32, i64 }>],
                                               [88 x i8] }
  {
   { i32, i32, i32, i32 } ; Zaglavlje mapirajućeg pokrivača
     { i32 2,    ; Broj funkcija
       i32 43,   ; Dužina niske putanja koje pripadaju prevodećem modulu
       i32 45,   ; Dužina kodiranih podataka mapirajuće pokrivača
       i32 0     ; Verzija mapirajućeg pokrivača
     },
   [2 x <{ i8*, i32, i32, i64 }>] ; Podaci o funkcijama
     [
       <{ i8*, i32, i32, i64 }> ; Podaci o prvoj funkciji
         <{
            i8* getelementptr inbounds ([3 x i8], [3 x i8]* @__profn_foo, i32 0, i32 0), ; Pokazivač na ime funkcije
            i32 3,   ; Dužina imena funkcije
            i32 9,   ; Dužina kodinarinih podataka mapirajućeg pokrivača ove funkcije
            i64 0    ; Strukturni heš funkcije
         }>,
      <{ i8*, i32, i32, i64 }>
         <{
            i8* getelementptr inbounds ([4 x i8], [4 x i8]* @__profn_main, i32 0, i32 0), ; Pokazivač na ime funkcije
            i32 4,   ; Dužina imena funkcije
            i32 31,  ; Dužina kodinarinih podataka mapirajućeg pokrivača ove funkcije
            i64 138  ; Strukturni heš funkcije
         }>
     ],
   [88 x i8] c"..." ; Kodirani podaci
  }, section "__llvm_covmap", align 8
```

Slika 5.5: Deo srednje reprezentacije sa promenljivom mapirajućeg pokrivača.

Podaci zapisani u zaglavlju mapirajućeg pokrivača i u slogovima podataka o funkcijama služe da se dekodiraju mapirajući regioni koji su zapisani u nisci kodiranih podataka. Format u kome su zapisani kodirani podaci je sledeći:

[Putanje izvornih kodova, podaciFunkcije0, podaciFunkcije1, . . . , poravnanje]

Radi boljeg razumevanja kodirani podaci su razloženi. Zbog obimnosti kodirani podaci su na slici 5.5 predstavljeni sa „. . . ” ali se sada prikazuju u potpunosti razložene prema prethodnom formatu na četri celine:

[\01\29/home/coverageMappingSimpleExample/main.c]
[\01\00\00\01\01\01\0F\02\02]
[\01\00\01\01\05\05\01\05\0B\0A\02\03\03 \09\00\0A\05\00\0B\05\04\05\02\08\00\0B\09\01\07\00\11]
[\00\00\00\00\00\00]

Prva linija je kodirana putanja izvorne datoteke dužinom zadatom u zaglavlju ove strukture. Prvi bajt označava indeks datoteke a drugi dužinu niske koja označava putanju do izvornog koda.

Ostatak predstavlja niz bajtova koji odgovaraju dvema funkcijma, `foo` i `main`, i poravnanju cele niske kodiranih podataka na 8 bajtova. Sa slike 5.5 se vidi da je mapirajući pokrivač za prvu funkciju iskodiran sa 9 bajtova a za drugu sa 31. Svaka od funkcija je zadata sledećim formatom:

[mapiranje izvršnih datoteka, brojački izrazi, mapirajući regioni]

Da bi se prikazala malo složenija situacija potrebno je preskočiti prvih 9 bajtova koji odgovaraju prvoj funkciji `foo`. Prvi bajt, nakon preskakanja, u ovom nizu je broj izvršnih datoteka u koje će se mapirati izrazi iz ove funkcije, a nakon njega sledi toliko indeksa datoteka. Ukoliko se u funkciji koriste makroi svaki od njih se mapira u virtuelnu datoteku koja predstavlja njegovo proširenje, a indeks datoteke svakog proširenja je jedinstven. U našem primeru prvi broj u trećem redu je „\01". On ozačava da imamo 1 indeks datoteke i njegova oznaka je sledeći bajt „\00".

Brojački izrazi su kodirani tako da prvi broj označava njihov broj a nakon toga se nalazi toliko parova indeksa mapirajućih brojača jer oni služe kao ulaz u binarni operator sabiranja ili oduzimanja u zavisnosti od reference koja pokazuje na njih. U okviru ovih izraza takođe se može naći mapirajući brojač koji se dobija izrazom neka druga dva izraza. Na datom primeru, nakon podatka za mapiranje izvršnih datoteka, posmatra se sledeći podatak „\01" koji govori da ima jedan izraz a on je dat sa naredna dva bajta „\01\05" koja označavaju da je potrebno izvšiti određeni izraz nad mapirajučim brojačima sa indeksom „\01" i „\05".

Mapirajući brojači su kodirani tako da se mogu dobiti neophodne informacije o pravom broju izvršavanja regiona. Vrednost najniža dva bita označava vrstu mapirajućeg brojača:

- 0 - Nula brojač (region se nije izvršio ni jednom)
- 1 - Referencirajući (pokazuje na stvarni instrumentizujući brojač)
- 2 - Oduzimajući izraz (vednost se dobija oduzimanjem dva stvarna brojača)
- 3 - Sabirajući izraz (vrednost se dobija sabiranjem dva stvarna brojača)

Ostali deo mapirajućeg brojača sadrži identifikaciju stvarnog brojača ukoliko se radi o referencirajućem mapiranju, a ukoliko se radi o nekom od izraza onda ostatak označava indeks u brojačkiom nizu izraza u kome se nalazi par identifikatora instrumentalizovanih brojača.

Ako su najniža dva bita zaglavlja nule onda se radi o pseudo brojaču a inače o mapirajućem brojaču. Pseudo brojač može označavati proširujući region, nula region (označen sa „\00") ili preskočen region (označen sa „\04"). U slučaju da je treći najniži bit postavljen na 1 i ako je neki od viših bitova različit od nule radi se o proširujućem regionu i tada ostatak označava identifikator datoteke u koji se ovaj region proširuje.

Podaci koji slede nakon brojačkih izraza su podaci o mapirajućim regionima. Najpre ide njihov broj (u našem primeru je to „\05") a zatim toliko ponavljanja podataka o regionima. Svaki region se sastoji od zaglavlja koje određuje vrstu brojača i četri podatka koja određuju njegovu poziciju: udaljenost linije od prethodnoh regiona (za početni region

se uzima udaljenost od početka datoteke); broj kolone početka regiona; broj linija do sledećeg regiona; broj krajnje kolone.

Brojač prvog regiona u primeru, nakon što je pročitan broj mapirajućih regiona, je „\01" i to je referencirajući brojač na stvarni instrumentalizovani brojač za ovu funkciju sa indeksom 0. Nakon ovog podatka sledi da je početak ovog regiona na udaljenosti „\05" od prethodnog (u ovom slučaju to je od kraja prethodne funkcije) u koloni „\0B", dužine „\0A" sa završetkom u koloni „\02". Slično se čitaju preostali podaci.

5.2 Proširivanje formata mapirajućeg pokrivača informacijama o grananju

Broj izvršavanja grane nastale iz instrukcije grananja zapravo predstavlja broj izvršavanja bloka koda do koga ta grana vodi. Stoga, kako mapirajući regioni opisuju blokove koda i njegove brojače, i kako je uglavnom svaki blok koda posledica nekog grananja, potrebno je proširiti mapirajuće regione informacijom o lokaciji grananja iz koje je taj region nastao. Mapirajući regioni su interno predstavljeni klasom `CounterMappingRegion`. Nju je potrebno proširiti novim poljem `unsigned BranchingStart` koje treba da označi lokaciju instrukcije grananja iz koje je region nastao[1]. Tako bismo u okviru mapirajućeg regiona imali lokaciju instrukcije grananja i broj izvršavanja te grane što odgovara nedostajućoj labeli BRDA „info" formata.

Kôd za generisanje mapirajućeg pokrivača se nalazi u datoteci lib/CodeGen/CoverageMappingGen.cpp podprojekta Clang. Postupak se satoji u obradi svake funkcije posebno tako što se redom posećuju njene instrukcije. Prilikom obrade se obraća pažnja na sve naredbe jezika i svaki od njih ima posebnu funkciju koja ga obrađuje. Svaka naredba jezika označava početak jednog i kraj prethodnog regiona. Kako je cilj da se dodaju informacije o početku grananja regiona od posebnog interesa su naredbe grananja kao što su **while, if, for, do {} while, switch** i operator **„?"**.

Kako preskačuće regione ne razmatramo za dodavanje podataka o grananju jer se ne izvršavaju, a proširujući regioni se preslikavaju u kodovske regione, potrebno je posmatrati samo kodovske regione. Oni su predstavljeni klasom `SourceMappingRegion`. Nju takođe treba proširiti dodatnom informacijom o lokaciji instrukcije grananja. Za razliku od objekata klase `CounterMappingRegion`, objekti klase `SourceMappingRegion` ne moraju u svakom trenutku sadržati granice regiona a i brojač ovakvih regiona je mapirajući brojač.

Princip koji je primenjen za obradu bilo koje naredbe grananja je isti, stoga će rad i izmene biti prikazane na listingu 5.2 koji obrađuje **while** naredbu. Obrade osta-

[1]Ukoliko mapirajući region nije posledica nekog grananja ili je taj region preskačući ova vrednost se postavlja na 0.

lih i preskačućih regiona treba samo ispraviti tako da se slažu sa proširenjem klase `CounterMappingRegion` i `SourceMappingRegion`. Postupak pravljenja regiona se sastoji u dodavanju novog neobrađenog regiona na stek `RegionStack` a zatim obilaska njegovih podregiona. Kada se završi obrada podregiona sa vrha steka se skida dodati region i dodaje se u vektor svih regiona `SourceRegions`.

Na vrhu steka može biti i region koji je u tom trenutku određen samo brojačem. Dodavanje takvog regiona se može videti u poslednjoj liniji iz listinga 5.2. Zbog takvih regiona prilikom obrade svake naredbe potrebno je najpre pozvati funkciju `extendRegion()` koja postavlja početnu lokaciju regiona sa vrha steka na lokaciju koja je zadata kao argument funkcije. Dodatno ova funkija treba da pripremi slučaj kada prosleđena lokacija prenosi obradu u drugi fajl ili ekspanziju makroa. Sledeći poziv ove funkcije se poziva za slučaj da je telo while naredbe ekspanzija makroa.

Glavni posao treba da izvrši funkcija `propagateCounter()`. Ona treba da izvrši obradu tela while petlje. Najpre se pravi novi aktivni region koji se dodaje na stek `RegionStack`. Granica regiona određena je telom petlje a brojač tog regiona je zadat kao argument ove funkcije. Nakon dodavanja regiona na stek `RegionStack` se obilazi telo petlje. Kada se obrada završi skidaju se svi aktivni regioni nastali tokom te obrade do nivoa pre pozivanja funkcije `propagateCounter()`. Novonastali regioni koji nisu skinuti su kodovski regioni koji su nastali ekspanzijom makroa. Svaki skinuti region se dodaje u vektorsku promenljivu `SourceRegions`.

Slično obradi tela petlje se obrađuje i region koji predstavlja uslov petlje. Poziv funkcije `adjustForOutOfOrderTraversal()` treba da postavi promenljivu `MostRecentLocation` koja se koristi za koordinaciju između kodovskih regiona iz različitih datoteka (pa samim tim i ekspanzijama makroa). Ona označava poslednju posećenu lokaciju aktivnih regiona sa steka.

```
void VisitWhileStmt(const WhileStmt *S) {
  extendRegion(S);

  Counter ParentCount = getRegion().getCounter();
  Counter BodyCount = getRegionCounter(S);

  // Handle the body first so that we can get the backedge count
  BreakContinueStack.push_back(BreakContinue());
  extendRegion(S->getBody());
  Counter BackedgeCount = propagateCounts(BodyCount, S->getBody());
  BreakContinue BC = BreakContinueStack.pop_back_val();

  // Go back to handle the condition.
  Counter CondCount =
          addCounters(ParentCount, BackedgeCount, BC.ContinueCount);
  propagateCounts(CondCount, S->getCond());
  adjustForOutOfOrderTraversal(getEnd(S));

  Counter OutCount =
          addCounters(BC.BreakCount, substractCounters(CondCount, BodyCount));
  if(OutCount != ParentCount)
    pushRegion(OutCount);
}
```

Listing 5.2: Neizmenjena obrada while naredbe

Prilikom obrada instrukcija grananja potrebno je pozivu funkcije `propagateCounter()` proslediti dodanu informaciju o početku linije instrukcije grananja regiona koji se stavlja na stek ovom funkcijom. Ova izmena je prikazana na listingu 5.3. Ukoliko se instrukcija grananja nalazi u kodovskom regionu koji je nastao ekspanznijom makroa kao početak grananja će se uzeti lokacija odakle je taj makro pozvan. Na ovakav izbor uticala je činjenica da GCC *„.info”* format radi na ovaj način. Da bi obezbedile potpune informacije, tj da bi svaka instrukcija grananja uvek imala podatke o bar dve grane, potrebno je izbaciti poslednji uslov **OutCount != ParentCount** i uvek dodavati još jedan mapirajući region sa grananjem koje odgovara trenutno obrađujućoj naredbi . Poseban slučaj predstavlja kada u kodu postoji naredba **if** bez pratećeg **else**. Tada se dodaje novi region kome se početak i kraj poklapaju, ovako se pravi region dužine 0, a njegov brojač je oduzimanje mapirajućih brojača roditeljskog regiona i regiona u okviru tela **if** naredbe. Ovim izmenama se ne dodaje stvarni brojač već povećavamo dužinu niske pokrivajućih podataka za dodatni region.

```
void VisitWhileStmt(const WhileStmt *S) {
  extendRegion(S);

  Counter ParentCount = getRegion().getCounter();
  Counter BodyCount = getRegionCounter(S);
  SourceLocation WhileLoc = S->getWhileLoc();

  if(WhileLoc.isMacroID())
    WhileLoc = getIncludeOrExpansionLoc(WhileLoc);

  // Handle the body first so that we can get the backedge count
  BreakContinueStack.push_back(BreakContinue());
  extendRegion(S->getBody());
  Counter BackedgeCount = propagateCounts(BodyCount, S->getBody(), WhileLoc);
  BreakContinue BC = BreakContinueStack.pop_back_val();

  // Go back to handle the condition.
  Counter CondCount =
          addCounters(ParentCount, BackedgeCount, BC.ContinueCount);
  propagateCounts(CondCount, S->getCond());
  adjustForOutOfOrderTraversal(getEnd(S));

  Counter OutCount =
          addCounters(BC.BreakCount, substractCounters(CondCount, BodyCount));
  if(OutCount != ParentCount)
    pushRegion(OutCount, None, None, WhileLoc);
}
```

Listing 5.3: Obrada while naredbe sa ubacivanjem informacije o grananju.

Kada se prođe kroz sve funkcije svi regioni se nalaze u vektoru `SourceRegions`. Iteracijom kroz ovaj vektor se na osnovu objekta klase `SourceMappingRegion` pravi objekat klase `CounterMappingRegion` i takav objekat se dodaje u vektorsku promenljivu `MappingRegions`. U ovu promenljivu se dodaju i preskačući i proširujući regioni. Kada se vektor `MappingRegions` popuni on se prosleđuje klasi `CoverageMappingWritter` koja treba da generiše kodirane podatke. U njoj je potrebno dodati upisivanje kodiranih informacija o grananju. Da bi se štedelo na veličini podatka informacija o grananju je kodirana kao udaljenost od početne linije regiona. Međutim, potrebno je obezbediti bit najniže vrednosti da bi se prepoznala situacija kada se radi o **do {} while** petlji jer u

ovom slučaju broj linije grananja je veći od broja linije početka regiona. Na sličan način potrebno je izmeniti i klasu `CoverageMappingReader` da učitava pokrivajuće podatke.

5.3 Primer generisanja novog formata

Za prikazivanje novih podataka iskorišćena je modifikovana verzija *llvm-cov* alata projekta LLVM kome je dodata dodatna opcija `geninfo` za generisanje podataka u GCC-ovom *„.info”* formatu. Implementacija ovog alata neće biti prikazana u ovom radu. Postupak dobijanja *„.info”* formata za primer 5.1 prikazan je na slici 5.6.

```
~/GeninfoTest ls
main.c
~/GeninfoTest clang -fprofile-instr-generate -fcoverage-mapping main.c -o a.out
~/GeninfoTest ./a.out
~/GeninfoTest llvm-profdata merge default.profraw -o default.profdata
~/GeninfoTest llvm-cov geninfo -show-branching  ./a.out -instr-profile=default.profdata
~/GeninfoTest cat default.info
TN:
SF:/home/rtrk/GeninfoTest/main.c
FN:1,foo
FN:5,main
FNDA:5,foo
FNDA:1,main
FNF:2
FNH:2
BRDA:8,0,0,10
BRDA:8,0,1,1
BRDA:10,0,0,5
BRDA:10,0,1,5
BRF:4
BRH:4
DA:1,5
DA:2,5
DA:5,1
DA:6,1
DA:7,1
DA:8,11
DA:9,10
DA:10,10
DA:11,5
DA:12,10
DA:14,1
LF:11
LH:11
end_of_record
```

Slika 5.6: Primer dobijanja i izgleda „info” datoteke modifikovanim alatima projekta LLVM.

Datoteka **default.profraw** nastaje kao rezultat izvršavanja profajliranog programa. Ona se ispisuje na kraju izvršavanja programa. Ime ove datoteke predstavlja uobičajenu vrednost. Uobičajena vrednost imena datoteke se može promeniti postavljanjem parametra okruženja (eng. *environment variable*) LLVM_PROFILE_FILE. Alat llvm-profile spaja više profraw datoteka u jedanu koja je namenjena *llvm-cov* alatu da je čita.

Ukoliko želimo da iz istog izvornog koda dobijemo GCC-ov *„.info”* koristi se postupak sa slike 5.7.

```
~/GcovInfoTest ls
main.c
~/GcovInfoTest gcc -fprofile-arcs -ftest-coverage main.c
~/GcovInfoTest ls
a.out  main.c  main.gcno
~/GcovInfoTest ./a.out
~/GcovInfoTest ls
a.out  main.c  main.gcda  main.gcno
~/GcovInfoTest geninfo .
Found gcov version: 5.4.0
Scanning . for .gcda files ...
Found 1 data files in .
Processing main.gcda
Finished .info-file creation
~/GcovInfoTest ls
a.out  main.c  main.gcda  main.gcda.info  main.gcno
~/GcovInfoTest cat main.gcda.info
TN:
SF:/home/rtrk/GcovInfoTest/main.c
FN:1,foo
FN:5,main
FNDA:5,foo
FNDA:1,main
FNF:2
FNH:2
BRDA:8,1,0,10
BRDA:8,1,1,1
BRDA:10,0,0,5
BRDA:10,0,1,5
BRF:4
BRH:4
DA:1,5
DA:2,5
DA:5,1
DA:6,1
DA:7,1
DA:8,11
DA:9,10
DA:10,10
DA:11,5
DA:12,10
DA:14,1
LF:11
LH:11
end_of_record
```

Slika 5.7: Primer dobijanja i izgleda „info” datoteke sa standarnim alatima

Glava 6

Unapređenje biblioteke za ispisivanje rezultata profajliranja

Compiler-rt je podprojekat projekta LLVM u kome se nalazi biblioteka za profajliranje na kojoj su rađena unapređenja. Ovaj projekat, pored biblioteke za profajliranje, obezbeđuje biblioteku za efikasnu implementaciju rutina za generisanje koda niskog nivoa kao i drugih funkcija koje se koriste kada na ciljanoj arhitekturi ne postoji kratka sekvenca instrukcija neophodna da se implementira određena IR operacija. Tako, na primer, 32-bitne arhitekture nemaju instrukcijsku podršku za 64-bitno deljenje. *Compiler-rt* obezbeđuje ovu funkcionalnost posebno optimizovanu za svaku podržanu arhitekturu. Ona predstavlja ekvivalent GCC-ovoj biblioteci libgcc. Dodatno ona sadrži sanitajzer za memoriju, adrese, niti i protok podataka [24].

Biblioteka koja sadrži funkcije za ispisivanje rezultata profajliranja je *clang_rt.profile* i napisana je u jeziku C (osim jedne kratke datoteke koja je napisan u C++). Ova biblioteka podržava ispisivanje dva tipa formata profajliranja. Ona, pored formata kompajlera Clang za ispivanje podataka profajliranja, pruža i podršku za GCDA format ispisa koji koristi kompajler GCC. Format GCDA podrazumeva pravljenje nove datoteke sa imenom izvorne datoteke i sufiksom „.gcno” za svaki instrumentalizovani izvorni kôd. Kada se izvrši program, rezultati profajliranja se ispisuju u datoteku sa imenom instrumentalizovane izvorne datoteke i sufiksom „.gcda”. Alat `geninfo` na osnovu „.gcda” i „.gcno” fajlova proizvodi „.info” datoteku čiji je format opisan u poglavlju 5. GCDA format zapisa se poklapao sa starim formatom zapisa iz GCC-a, ali se u novijim verzijama GCC-a ovaj format izmenio i sada postoje različite verzije ovog formata. Implementacija podrške GCDA formatu se nalazi samo u jednoj datoteci ove biblioteke koja je potpuno nezavisna od ostalih datoteka. Izmene koje će biti prikazane odnosiće se na format ispisa kompajlera Clang tako da će ova datoteka biti zanemarena.

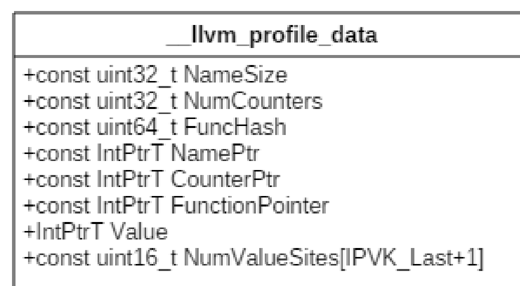
Biblioteka *clang_rt.profile* je statička i potrebno ju je uključiti u svaki kompilacioni

modul[1] koji želimo da profajliramo. Svaka ovako ulinkovana *clang_rt.profile* biblioteka radi potpuno nezavisno. Funkcijski poziv za ispisivanje prikupljenih podataka tokom rada programa bi ispisao samo podatke za modul u kome je funkcija pozvana. Unapređenje *clang_rt.profile* biblioteke zasniva se na stvaranju veze između podataka iz različitih modula sa ciljem da se obezbedi jedan funkcijski poziv koji će ispisati podatke iz svih profajliranih modula.

Prva ideja o načinu implementacije ove veze je da se umesto uvezivanja više statičkih nezavisnih biblioteka napravi jedna dinamička koja bi objedinila sve potrebne podatke i kojom bi se uštedelo na memoriji računara. Tokom implementacije dinamičke biblioteke pokazale su se neke njene mane a samim tim i prednosti korišćenja statičke. Stoga će u radu biti prikazana tri načina implementacije unapređenja: kroz dinamičku biblioteku, kroz statičku i kao mešavina statičke i dinamičke biblioteke.

6.1 Biblioteka clang_rt.profile

Centralna struktura *clang_rt.profile* biblioteke je `__llvm_profile_data` prikazana na slici 6.1. Ona čuva pokazivače na podatke koji su dodati od strane kompajlera u svaki instrumentalizovani modul. Pokazivači ove strukture se odnose samo na podatke koji su vezani za samo jednu funkciju. Polje CounerPtr je pokazivač na niz brojača dužine NumCounters za funkciju čije ime duzine NameSize je zapisano na lokaciji NamePtr. FuncHash predstavlja heš funkcije koji koriste alati projekta LLVM kako bi razlikovali istoimene funkcije iz različitih modula, dok se polje FunctionPointer koristi samo prilikom testiranja. Poslednja dva polja se koriste prilikom praćenja vrednosti određene promenljive. Vrednost promenljive se prati kako bi se na osnovu njenog ponašanja mogle izvršiti određene optimizacije. Najčeće se prati vrednost invarijante petlje.



Slika 6.1: Struktura __llvm_profile_data.

Sve funkcije ove biblioteke služe za upravljanje strukturom sa slike 6.1. Zaglavlje `InstrProfiling.h` pruža interfejs korisniku ukoliko želi da manipuliše prikupljenim po-

[1] Pod kompilacionim modulom se podrazumeva program ili bilo koja dinamička biblioteka.

dacima nezavisno od načina koji kompajler implementira. Globalne pokazivačke promenljive u okviru biblioteke kao i funkcije definisane u zaglavlju `InstrProfiling.h` su definisani sa atributom *„hidden”*. Ovaj atribut sugeriše linkeru način na koji treba da poveže ovako naznačene simbole, to jest on treba da povezuje ovakve simbole samo na nivou modula i oni neće biti vidljivi iz drugih. Ponašanje svakog poziva ovakve funkcije je određeno modulom u kome je pozvana i *„hidden”* podacima koji mu pripadaju.

Ukoliko korisnik želi da isključi inicijalizaciju biblioteke u određenom profajliranom modulu potrebno je da u njemu definiše promenljivu `__llvm_profile_runtime` tipa `int`. Na ovaj način će se zamaskirati istoimena promenljiva iz datoteke `InstrProfilingRuntime.cc` koju kompajler referencira (u narednom delu će biti objašnjen i način na koji se referencira) kako bi se pokrenula inicijalizacija klase `RegisterRuntime`. Ova klasa u svom konstruktoru vrši poziv dve funkcije biblioteke. Funkcija `__llvm_profile_initialize_file()` treba da napravi datoteku u koji će se upisati podaci ukoliko ona ne postoji, dok `__llvm_profile_register_write_file_atexit()` treba pozivom funkcije `atexit` da obezbedi pozivanje funkcije `__llvm_profile_write_file()` na kraju izvršavanja programa. Ova funkcija je „hidden” i ona ispisuje podatke iz niza podataka tipa *__llvm_profile_data* koji odgovaraju jednom modulu. To znači da se ispisivanje podataka svih instrumentalizovanih modula izvršava tek na kraju programa tako što se poziva funkcija ispisivanja za svaki modul posebno. Takav mehanizam nam ne pruža mogućnost ispisivanja svih podataka iz svakog instrumentalizovanog modula u proizvoljnom trenutku što predstavlja problem za programe koji imaju dugo ili pak neograničeno vreme izvršavanja.

6.2 Generisanje dodatnog koda u IR prilikom profajliranja

Prednji deo kompajlera Clang proizvodi srednju reprezentaciju. Njena modifikacija u zadnjem delu kompajlera se odvija kroz više prolaza, od kojih svaki vrši specifičnu modifikaciju samog IR-a. Za profajliranje su interesantni prolazi `PGOInstrumentationGen`, `PGOInstrumentationUse` i `InstrPrfoiling`. Oni se izršavaju u navedenom redosledu.

Ukoliko smo uključili zastavicu za profajliranje u njega će algoritmom Knuta [10], koji smo opisali algoritmom 1, tokom prolaza `PGOInstrumentationGen`, biti označena mesta u srednjoj reprezentaciji na koja treba dodati brojače. Broj izvršavanja preostalih ivca se izračunava algoritmom 2 u narednom prolazu `PGOInstrumentationUse`. On je takođe zadužen za izračunavanje formule po kojoj se izračunava broj izvršavanja blokova pomoću broja izvršavanja ivica.

Sledeći prolaz `InstrPrfoiling` na označena mesta iz prolaza `PGOInstrumentationGen` postavlja brojače. Ubacivanje novog koda u medjureprezentaciju ranim fazama menja znacajno realno ponasanje koda (npr. neki brojac ce zauzeti registar i neka promen-

ljiva ce morati da se prebaci u memoriju i stoga ce instrumentalizovani program raditi drugacije nego originalni). Tokom ovog prolaza ujedno se inicijalizuju potrebeni podaci koji čuvaju informacije o profajliranju u predodređenoj sekciji podataka. U zavisnosti od operativnog sistem na odgovarajući način se vrši povezivanje kompajlerski alociranog niza podataka za jedan modul i promenljivih iz biblioteke koje pokazuju na podatke[2] sa slike 6.1 . Takođe se dodaju i pozivi iz clang_rt.profile koji treba da izvrše inicijalizaciju ove biblioteke kao i registrovanje poziva funkcije za ispisivanje prikupljenih podataka na kraju izvršavanja programa.

Od interesa za ovaj rad neophodno je razumeti kako prolaz `InstrProfiling` uključuje pozive iz statičke biblioteke clang_rt.profile pa ćemo stoga samo taj deo detaljnije prikazati. Pozivi iz posmatrane biblioteke dodaju se na kraju prolaza pomoću tri funkcije:

- **emitRegistration ()** – Linuks, FriBSD i Epl operativni sistemi mogu da dohvate podatke koji su predstavljeni strukturuom sa slike 6.1 tako sto će pomoću *„hidden”* pokazivačke promenljivih iz biblioteke ciljati[3] promenljivu, tj. deo sekcije podataka u kojoj se ona nalazi, tako da, ukoliko se program kompajlira za navedene operativne sisteme, funkcija `emitRegistration()` neće ništa promeniti već će se odmah vratiti. Ukoliko se radi o nekom drugom operativnom sistemu ovaj poziv pravi funkciju `__llvm_profile_register_funsctions (void)` u kojoj će se za svaku profajliranu funkciju iz odgovarajućeg modula pozvati funkcija `__llvm_profile_register_function (__llvm_profile_data *Data)` iz compiler_rt.profile biblioteke. Ona treba da postavi *„hidden”* promenljive tako da one pokazuju na odgovarajuće segmente memorije tako što će svakim pozivom proširivati deo memorije na koji one pokazuju. Argument ove funkcije je struktura koja sadrži informacije za odgovarajuću instrumentalizovanu funkciju ili makro (ova struktura će detaljnije biti opisana u sledećem delu) i koji pokazuju na jedan deo memorije alociran za podatke modula iz kog je funkcija pozvana.

- **emitRuntimeHook ()** – Ukoliko se ne radi o Linuks operativnom sistemu ova funkcija treba da pokrene inicijalizaciju biblioteke. Pozivi za inicijalizaciju biblioteke se nalaze u datoteci `InstrProfilingRuntime.cc` u obliku konstruktora klase `RegisterRuntime` definisanog u ovoj datoteci. Pored definicije ove klase u ovoj datoteci postoji i deklaracija jednog objekta ove klase kao i deklaracija *„hidden”* promenljive `__llvm_profile_runtime`. Poziv `emitRuntimeHook()` treba da napravi novu funkciju `__llvm_profile_runtime_user()` koja treba da vrati promenljivu `__llvm_profile_runtime` i tako iskoristi ovu promenljivu iz biblioteke što će dalje naterati linker da uključi i pravljenje objekta klase `RegisterRuntime`. Ako je ciljni

[2]Množina se pominje zato što se čuva pokazivač na celu strukturu, pokazivač na niz brojača i pokazivač na niz karaktera koji čuva imena funkcija.

[3]Za Likunks, FriBSD i Epl operativne sisteme moguće je postaviti podatke za profajliranje u posebno rezervisane segmente memorije.

operativni sistem programa koji se prevodi Linuks onda se ova funkcija zamenjuje dodatnim parametrom `-u__llvm_profile_runtime` prilikom linkovanja programa koji ima isti efekat kao prethodna funkcija.

- **emitInitialization ()** – Ukoliko postoji funkcija `__llvm_profile_register_functions()`, napravljena u pozivu `emitRegistration()`, i funkcija iz *clang_rt.profile* biblioteke za promenu imena uobičajene datoteke (u koji se ispisuju rezultati profajliranja) onda `emitInitialization()` funkcija treba da obezbedi njihovo pozivanje prilikom inicijalizacije programa. Funkcija za promenu uobičajenog imena datoteke će biti promenjena ukoliko je postavljena promenljiva okruženja (eng. *environment variable*) `LLVM_PROFILE_FILE`.

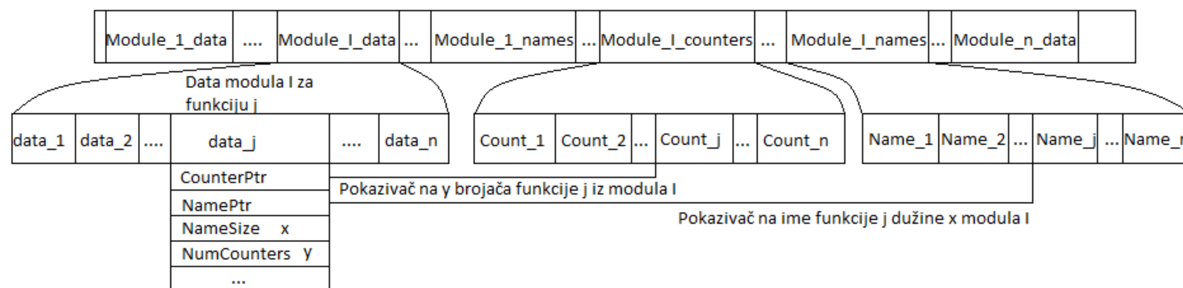
6.3 Funkcijski poziv za ispisivanje svih podataka profajliranja

Kao što je rečeno, problem predstavlja način na koji se ispisuju podaci, to jest korišćenje funkcije atexit kako bi se na kraju posebno iz svakog modula pozvala funkcija za ispisivanje prikupljenih podataka modula iz koga je pozvana. Da bi prevazišli ovo ograničenje potreno je napraviti jednu funkciju koja će ispisivati podatke iz svih modula. Da bi jedna funkcija ispisala sve podatke koristiće se **globalna lista** koja će čuvati pokazivače na podatke iz svakog modula. Element liste je predstavljen strukturom `__llvm_profile_module_data` prikazanom na slici 6.2. U zavisosti od načina kompajliranja ove biblioteke, to jest da li je biblioteka statička, ili je podeljena na dva dela kao statička i dinamička ili je samo dinamička, lista će se popunjavati na različite načine i listom će se upravljati na različite načine. Izbor jednog od ova tri načina povezivanja je moguć preko kompilacione linije. Za čistu dinamičku potrebno je dodati *-fprofile-dynamic*, za statiču i dinamičku *-fprofile-static-dynamic*, dok za čistu statičku nije potrebno navesti ni jedan dodatni argument.

__llvm_profile_module_data
+const __llvm_profile_data *DataFirst +const __llvm_profile_data *DataLast +const char *NameFirst +const char *NameLast +uint64_t *ConutersFirst +uint64_t *CountersLast +struct __llvm_profile_module_data *Next

Slika 6.2: Struktura __llvm_profile_module_data.

Struktura sa slike 6.2 je nastala na osnovu načina na koji su se ispisivali podaci iz strukture `__llvm_profile_data`. Naime, pokazivači iz strukture `__llvm_profile_data`, `NamePtr` i `CounterPtr` predstavljaju pokazivače na odgovarajući deo kontinualne memorije koji se nalazi u sekciji namenjenoj za podatke profajliranja jednog modula. Jedan deo te kontinualne memorije odgovara jednoj funkciji. Isto tako postoji i parče kontinualne memorije i za podatke tipa `__llvm_profile_data`. Na slici 6.3 je prikazan primer rasporeda ovih segmenata memorije za više instrumentalizovanih modulua.



Slika 6.3: Raspored podataka profajliranja u memoriji.

Polja strukture `__llvm_profile_module_data` pokazuju na početak i kraj kontinualnih segmenata memorije jednog modula. Ova struktura treba da zameni upotrebu *„hidden"* promenljivih `DataFirst, DataLast, NameFirst, NameLast, CountersFirst` i *CountersLast* koje su pokazivale na početak i kraj ovih segmenat. Ove *„hidden"* promenljive su vezane za modul, ali ako njihove vrednosti postavimo da budu polja odgovarajućih elementa strukture koja se ulančava u globalnu listu onda ih možemo ispisati iz bilo kog modula. Za Linuks, FriBSD i Epl operativne sisteme se koriste drugačiji nazivi, ali broj promenljivih, njihovi tipovi i značenje su isti. Imena *„hidden"* promenljivih koja smo naveli se koriste za operativne sisteme koji su različiti od navedenih, tj za ostale operativne sisteme kao na primer Windows, NetBSD, OpenBSD, DragonFly itd. U zavisnosti od operativnog sistema se bira jedana od tri datoteke u kojoj su definisane promenljive (Linuks i FriBSD imaju zajedničku, jedna za Epl i poslednja za ostale sisteme). Sve tri datoteke definišu istoimene *„hidden"* funkcije koje vraćaju odgovarajuće promenljive.

Dinamička biblioteka

Ukoliko se radi o potpuno dinamičkoj biblioteci potrebno je u okviru nje obezbediti uniformnu registraciju podataka iz svih modula. U ovom slučaju ne možemo se osloniti na dohvatanje podataka iz određene sekcije iz memorije jer je sada ova biblioteka poseban modul i ona je nezavisna od drugih instrumentalizovanih modula i to predstavlja manu ovog pristupa. Kao deo rešenja će se iskoristiti već postojeći mehanizam registrovanja

podataka pomoću poziva funkcije `__llvm_profile_register_function()` za svaku instrumentalizovanu funkciju. Ovo podrazumeva isključivanje mehanizama „linker magic" koji se koristi za platforme različite od Linuksa i Darwina u funkciji `emitRegistration()` koja je opisana u 6.2 (u llvm verziji 3.6 za Linuks se koristio ovaj mehanizam). Dodatno je potrebno izmeniti ovu funkciju kako bi ona na što efikasniji način čuvala podatke u listi.

Podaci za profajliranje jednog modula su alocirani u jednom kontinualnom delu memorije. Jedan deo tog fragmenta memorije predstavljaja podatke za jednu instrumentalizovanu funkciju koji se prosleđuje kroz argument funkcije `__llvm_profile_register_function()`. Kako se ne može garantovati da se podaci za funkcije dodaju u redosledu u kom su zapisani u kontinualnom delu memorije, za efikasnije čuvanje podataka je implementiran algoritam sličan sortiranju sa ubacivanjem (eng. *insertion sort*). Cilj algoritma koji je opisan u sledećem pasusu je da jedan element liste označava kontinualni deo memorije podataka profajliranja.

Neka je element A, veličine d bajtova, zadat sa svojom početnom adresom *početnaAdr(A)* i krajnjom *krajnjaAdr(A) = početnaAdr(A) + d*. Tada proveravamo elemente liste $lstI$ redom i ukoliko je *krajnjaAdr(A) == početnaAdr(lstI)* onda proširujemo početak elementa $lstI$, ili ukoliko je *krajnjaAdr(lstI) == početnaAdr(A)* onda proširujemo kraj elementa $lstI$. Ako je element proširen treba dodatno proveriti da li ga je moguće pripojiti prethodnom ili narednom elementu liste. Ukoliko ne možemo da proširimo element treba napraviti novi element liste i umetnuti ga na odgovarajuće mesto u zavisnosti od njegove adrese. Kako sve šablonske funkcije iz C++ jezika dele iste brojače a samim tim i isti deo koda, a kompajler svaku vidi kao posebnu pa se za svaku od njih poziva __llvm_profile_register_function() onda moramo dodati još jedan dodatan uslov da ukoliko je *početnaAdr(A) >= početnaAdr(lstI) && krajnjaAdr(A) <= krajnjaAdr(lstI)*, to jest ako je element A već sadržan u elementu $lstI$, onda treba završiti sa ubacivanjem elementa A jer je on već ubačen. Složenost ovog algoritma ne zavisi samo od broja funkcija za koje se ubacuju podaci, već i od broja instrumentalizovanih modula jer će podaci za funkcije jednog modula biti grupisani, stoga ako imamo k modula i n funkcija onda imamo složenost sortiranja ubacivanjem $\mathcal{O}(n * k)$.

Prilikom dodavanja novih elemenata listi neophodno je koristiti funkciju standardne biblioteke `malloc`. Korišćenje ove funkcije dovodi do padanja jednog testa iz skupa testova za profile biblioteku. Cilj svih biblioteka iz compiler-rt je da budu nezavisna od standardne biblioteke, međutim profajl biblioteka kao i sam podprojekat compiler-rt i dalje koriste standardnu biblioteku, pa je stoga ovaj test zanemaren.

Statička biblioteka

Statička biblioteka popunjava listu dosta brže za Linuks i Epl operativne siteme i ne zahteva dodatnu alokaciju memorije. Ovo rešenje predstavlja jednostavnu nadogradnju prethodne neizmenjene verzije statičke biblioteke. U odnosu na dinamičku, umesto alokacije novih elemenata i proširivanja elemenata liste, svaki instrumentalizovani modul će sadržati statičku promenljivu koja predstavlja jedan element liste. Povezivanje elemenat iz svih modula se obavlja prilikom inicijalizacije programa pomoću funkcije `__llvm_profile_module_register()`. Ona treba da polja strukture `__llvm_profile_module_data` poveže sa odgovarajućim pokazivačima pomoću funkcija `__llvm_profile_[begin|end]_[data|names|counters]()`, i da takav element doda u listu. Poziv ove funkcije se vrši pomoću konstruktora nove klase `RegisterModuleLink`, na sličan način kao što se obavlja inicijalizacija biblioteke pomoću klase `RegisterRuntime`. Navedenu funkciju smo mogli dodati u konstruktor `RegisterRuntime` klase ali ukoliko korisnik želi sam da izvrši inicijalizaciju morao bi da pozove i ovu funkciju. Bolje rešenje bi bilo da sam kompajler doda poziv ove funkcije prilikom inicijalizacije svakog modula ali je zbog fleksibilnosti[4] implementirano povezivanje elemenata kroz konstruktor klase `RegisterRuntime`.

Statička i dinamička biblioteka

Verzija rešenja gde imamo statičku i dinamičku bilioteku je jako slična statičkoj. Naime, ceo kôd je isti, samo što je podeljen na statički deo koji je zadužen za popunjavanje polja elementa liste i na dinamički koji je zadužen za povezivanje elemenata liste i upravljanje podacima. Prednost ovog pristupa u odnosu na kompletno statičku je ušteda memorije koja se ostvaruje zahvaljujući zajedničkom delu za upravljanjem podacima. Ako imamo n instrumentalizovanih modula, a razlika između količine memorije statičke biblioteke iz prethodnog dela i iz ovog je d, onda imamo memorijsku uštedu od $(n-1)*d$. Razlika u odnosu na čistu dinamičku biblioteku je taj što ona mora da pozove `__llvm_profile_register_function()` za svaku instrumentalizovanu funkciju i u slučaju kada se izvršava na Linuks ili Epl operativnim sistemima.

6.4 Testiranje

Primena prethodno navedene tri biblioteke je istestirana na *Android-NOUGHAT* operativnom sistemu za MIPS CI20 uređaj. Za potrebe testiranja i upoređivanja izvučeni su podaci o veličini programa i deljenih biblioteka komandom `size` za MIPS uređaje. Oni

[4] Pod fleksibilnosti se misli da ovaj pristup ne zahteva nikakve promene u kompajleru već se poziv ove funkcije može uključiti prilikom procesa linkovanja dodavanjem -u__llvm_profile_module_link u link liniju.

Tabela 6.1: Ukupno uvećanje i prosečno procentualno povećanje po entitetu

	Statička	Dinamička	Kombinacija
Povećanje u MB	553	674	529
Prosečno povećanje u %	136%	33%	48%
Povećanje biblioteke u MB	402	478	390
Prosečno povećanje bilioteke u %	153%	35%	53%
Povećanje programa u MB	150	196	139
Prosečno povećanje programa u %	116%	31%	42%

su kompajlirani kompajlerom Clang bez instrumentalizacije i sa instrumentalizacijom na tri načina u zavisnosti od primenjenih zastavica prilikom kompajliranja koje su date u poglavlju 6.3.

Biblioteke za profajliranje su dodatno izmenjene tako da pored postojeće funkcionalnosti prave jednu nit sa beskonačnom petljom u kojoj će u fiksnom vremenskom intervalu pozivati funkciju za ispisivanje prikupljenih podataka.

Prilikom izgradnje sistema bilo je potrebno izmeniti sistem za izgradnju (eng. build system) softvera tako da on instrumentalizuje binarne datoteke koje se kompajliraju za ciljanu MIPS platformu. U sistemu za izgradnju, pored zastavica za način instrumentalizacije bilo je potrebno dodati i zastavicu „-mxgot" koja treba da poveća veličinu globalne tabele rastojanja (eng. global offset table). Potreba za povećanjem ove tabele nastaje zbog upotrebe velikog broja brojača za instrumentalizaciju. Takođe izostavljena je instrumentalizacija statičkih izvršnih datoteka jer se oni mogu instrumentalizovati samo sa statičkim pristupom. Da bi takve datoteke mogli instrumentalizovati sa ostale dve verzije pristupa bilo bi potrebno napraviti statičke verzije biblioteka iz dinamičkog i kombinovanog pristupa.

Prilikom testiranja izvršena je instrumentalizacija 556 entiteta različitih veličina, od toga 293 su dinamičke biblioteke a 263 su programi. Rezultati iz table 6.1 pokazuju da instrumentalizacija dinamičkom bibliotekom koristi najviše memorije iako ona u proseku povećava svaku biblioteku za 33%. Razlog za ovakve rezultate je taj što dinamički pristup zahteva pravljenje `__llvm_profile_module_data` objekta za svaku funkciju kako bi se oni prosledili kao argument funkcije `__llvm_profile_register_function()`. Kada je broj funkcija entiteta dovoljno veliki dinamički pristup zahteva veću količinu memorije od preostala dva pristupa. Sa druge strane rezultati iz tabele 6.1 prikazuju da se najveća memorijska ušteda dobija korišćenjem kombinovanog pristupa.

U tabeli 6.1 takođe stoji da je prosečno uvećanje za kombinovani pristup 48%. Ovaj podatak, kao i podatak o uvećanju statičkim pristupom od 136% može da zavara jer je 180 instrumentalizovanih entiteta manje od veličine samih biblioteka. Prosečno zauzeće biblioteka iz sva tri pristupa je 88KB te je stoga u tabeli 6.2 prikazano procentualno uvećanje za entite čija je veličina do 88KB i čija je veličina preko te granice. Na osnovu prikaza iz table 6.2 može se zaključiti da visok procenat za statičke biblioteke iz table

Tabela 6.2: Procenat uvećanja entiteta za objekte sa zadatnom granicom od 88K

	Statička	Dinamička	Kombinacija
Ispod 88K	338%	34%	85%
Iznad 88K	40%	33%	30%

6.1 potiče od uvećanja entita manjih od prosečne veličine biblioteka. Takođe, uvećanje kombinovanog pristupa od 85% za entitete manje od 88K opravdava prosečno uvećanje od 42% svih entiteta. Isto tako, kako su biblioteke u većini slučajeva manje od programa, procenat prosečnog povećanja biblioteka, 153% za statički pristup i 53% za kombinovani, je očekivan. Dodatno može se videti da dinamički pristup daje srazmerna uvećanja za sve entitete.

Poređenje veličina izvršnih datoteka izgrađenih kompajlerom Clang i kompajlerom GCC prevazilazi okvire ovog rada pa s toga ono nije urađeno. Softver *Android-NOUGHAT* operativnog sistema je podešen tako da se za njegovu izgradnju koristi samo kompajler Clang. S toga ne postoji jednostavan način da se u sistemu za izgradnju na jednostavan način promeni kompajler. Pojedini moduli, ukoliko nisu u prevelikoj sprezi sa kompajlerom Clang, mogu biti izgrađeni kompajlerom GCC minimalnim izmenama njihovih mejkfajlova (eng. makefile), ali prilikom njihove instrumentalizacije došlo bi do raznih problema. Jedan od uočenih problema je problem premale globalne tabele rastojanja GCC-ove libgcov.a statičke biblioteke za ispsisvanje prikupljenih podataka. Ova biblioteka se u sistemu izgradnje softvera koristi kao već izgrađena (eng. prebuilt) biblioteka.

6.5 Dalji rad

Prenošenje (eng. porting) ove implementacije na noviju verziju svakako donosi novu funkcionalnost ali sa sobom nosi i određene izazove. Novije verzije profile bibliotke podprojekta compiler-rt su dodatno poboljšale memorijsku efikasnost biblioteke unapređenim načinom za efikasnije čuvanje niski imena funkcija. Takođe, na novijim verzijama je dodato novo polje u __llvm_profile_data strukturu koje služi za praćenje vrednosti (eng. porting)promenljivih što dodatno podiže složenost prenošenja na novu verziju.

Glava 7

Zaključak

U ovom radu ukratko je dat uvid u osnove uređaja sa ugrađenim računarom. Opisani su njihovi osnovni logički entiteti, karakteristike i njihova moguća ograničenja. Pored toga prikazana je struktura i organizacija kompajlerske infrastrukture LLVM, kao i njena osnovna načela i ideje. Isto tako, prikazana je osnovna podela profajliranja na osnovu granulacije, kao i osnovni algoritmi koji se koriste prilikom instrumentalizacije programa. Jedan od prikazanih algoritama je i algoritam optimalne instrumentalizacije ivica koji uključuje i izračunavanje broja izvršavanja blokova koda. Ovaj algoritam se primenjuje prilikom profajliranja kompajlerom Clang.

Da bi se izjednačila količina informacija profajliranja dobijena kompajlerom Clang sa količinom informacija koju daje kompajler GCC, dat je opis mapirajućeg pokrivača kompajlera Clang, koji se koristi kao njegov interni format profajliranja, kao i implementacija proširenja ovog formata. Važnost postojanja zajedničkog formata dolazi do izražaja na projektima koji se razvijaju duži niz godina, koji su u prošlosti koristili kompajler GCC, a usled sve veće popularnosti kompajlera Clang, postepeno prelaze na njega. Tom prilikom se mogu iskoristi već postojeći alati napisani za obradu formata profajliranja kompajlera GCC.

U sklopu rada implementirano je proširenje rantajm biblioteke `profile` podprojekta compiler-rt projekta LLVM. Implementacija pruža jedinstven funkcijski poziv za ispisivanje svih podataka profajlianja iz logički odvojenih entiteta pravljenjem liste podataka profajliranja svih modula. Pružanje ovakve podrške je urađeno na tri načina. Jedan način je pravljenje dinamičke biblioteke, koja se ponaša kao jedinstveni centralni deo i koja popunjava i upravlja listom svih podataka profajliranja. Ovaj pristup očekuje da kompajler obezbedi poziv funkcije za registraciju podataka profajliranja svake instrumentalizovane funkcije. Takva funkcija popunjava globalnu listu podataka profajliranja. Drugi način se sastoji od dodavanja statičke bibliotke u svaki instrumentalizovani modul. Tako dodata biblioteka dodaje jedan element, koji predstavlja sve funkcije modula, u globalnu listu podataka prilikom inicijalizacije samog modula. Poslednji način predstavlja kombinaciju prethodna dva. On se sastoji od statičke biblioteke zadužene za popunjavanje globalne

liste i dinamičke biblioteke koja je zadužena za upravljanjem podacima iz iste. Sva tri pristupa su istestirana na operativnom sistemu *Android-NOUGHAT* za MIPS uređaj sa ugrađenim računarom.

Postojanje jednog funkcijskog poziva za ispisivanje svih prikupljenih podataka u velikoj meri olakšava prikupljanje podataka profajliranja. Jedan od jednostavnijih načina za dodavanje ovakve podrške bi bilo pravljenje biblioteke koja bi na određeni signal iz komandne linije pozivala ovu funkciju. Međutim, profajliranje zahteva osetnu količinu dodatne memorije, a upravo tim resusursom su uređaji sa ugrađenim rečunarom u većini slučajeva izuzetno ograničeni. Rezultati rada pokazuju da kombinovani pristup profajliranja zahteva manje memorije nego statički i dinamički pristup. Ponašanje kombinovanog pristupa se pokazalo najbolje u praksi, iako dinamički pristup daje konstantno uvećanje entiteta različitih veličina. Uzimajući u obzir i bržu inicijalizaciju kombinovanog pristupa, jasno je da je on u prednosti u odnosu na dinamički pristup. Prednost kombinovanog pristupa u odnosu na statički se ogleda u njegovom manjem prosečnom uvećanju biblioteka različitih veličina koje proističe iz načina korišćenja statičkog i dinamičkog dela.

Literatura

[1] MAGNUS ÅGREN. „Static Code Analysis For Embedded Systems”. Master teza. Sweden: Department of Computer Science i Engineering Chalmers University of Technology, 2009.

[2] Alan Eustance Amitabh Srivastava. “ATOM: A System for Building Customized Program Analysis Tools”. English. In: *ACM SIGPLAN Conference on Programming Language Design and Implementation* 26 (1994), pp. 196–205.

[3] M. Broy. „FemSys’97”. U: Apr. 1997.

[4] *Embedded System TutorialsPoint*. `http://aiweb.techfak.uni-bielefeld.de/content/bworld-robot-control-software/`. [Online; accessed 10-May-2017]. 2015.

[5] *GNU Compiler Collection (GCC)*. `https://gcc.gnu.org/`. Accessed: 2017-05-12.

[6] *GPROF Tutorial – How to use Linux GNU GCC Profiling Tool*. `http://www.thegeekstuff.com/2012/08/gprof-tutorial`. Accessed: 2017-05-12.

[7] Thomas Ball James R.Larus. “Efficient Path Profiling”. English. In: *MICRO 29 Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture* (1996), pp. 46–57.

[8] Thomas Ball James R.Larus. “Optimally profiling and tracing programs”. English. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 16 (1994), pp. 1319–1360.

[9] Thomas Ball James R.Larus. “Rewriting Executable Files to Measure Program Behavior”. English. In: *Software—Practice & Experience* (1994), pp. 197–218.

[10] Donald E. Knuth i Francis R. Stevenson. „Optimal measurement points for program frequency counts”. U: *BIT Numerical Mathematics* 13.3 (Sept. 1973), str. 313–322.

[11] Chris Lattner i Vikram Adve. „Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO’04)”. U: Palo Alto, California, Mar. 2004.

[12] *„libc++” C++ Standard Library*. :https://libcxx.llvm.org/. Accessed: 2017-06-29.

[13] *libFuzzer – a library for coverage-guided fuzz testing.* `http://llvm.org/docs/LibFuzzer.html`. Accessed: 2017-05-12.

[14] Bruno Cardoso Lopes i Rafael Auler. *Getting Started with LLVM Core Libraries.* Livery Place 35 Livery Street Birmingham B3 2PB, UK.: Packt Publishing, 2014.

[15] Trishul Chilimbi Martin Hirzel. "Bursty Tracing: A Framework for Low-Overhead Temporal Profling". English. In: *Workshop on Feedback-Directed and Dynamic Optimizations (FDDO)* 4 (2001), pp. 33–37.

[16] Barbara G. Ryder Matthew Arnold. "A Framework for Reducing the Cost of Instrumented Code". English. In: *ACM SIGPLAN Notices* 36 (1994), pp. 168–179.

[17] Anthony Massa Michael Barr. *Programming Embedded Systems, Second Edition with C and GNU Development Tools.* English. Ed. by O'Reilly. 2006.

[18] Rifat M.Ramović. *Pouzdanost sistema elektronskih, telekomunkikacionih i informacionih.* Katedra za Mikroelektroniku i tehničku fiziku, 2005.

[19] Andreas Neustifter. „Efficient Profiling in the LLVM Compiler Infrastructuren". Master teza. Vienna: Vienna University of Technology, 2010.

[20] Julian Seward Nicholas Nethercote. "Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation". English. In: *ACM SIGPLAN Notices - Proceedings of the 2007 PLDI conference* 42 (2007), pp. 89–100.

[21] Mayur Pandey i Suyog Sarda. *LLVM Cookbook.* Livery Place 35 Livery Street Birmingham B3 2PB, UK.: Packt Publishing, 2015.

[22] Sriram Neelakandan P.Raghavan Amol Lad. *Embedded Linux System Design And Development.* English. 2006.

[23] Dominic Sweetman. *See MIPS Run.* 2006.

[24] *The LLVM Compiler Infrastructure.* `http://llvm.org/`. Accessed: 2017-05-12.

[25] Frank Vahid and Tony Givargis. *Embedded System Design:A Unified Hardware/-Software Approach.* English. 1999.

[26] Fernando Sánchez Villaamil and Ivica Bogosavljević. "Efficient Instrumentation for the Purpose of Profiling and Debugging". English. In: (2008).

Biografija autora

Nikola Prica (*Valjevo, 18. decembar 1992.*) Rođen sam u Valjevu. Završio sam Valjevsku Gimnaziju 2011. godine i iste godine upisao Matematički fakultet u Beogradu. 2014. godine sam završio osnovne studije na Matematičkom fakultetu sa prosečnom ocenom 8.56 i upisao master na istom. Položio sam ispite master studija 2016. godine sa prosečnom 8.25. Od jula 2015. godine pa do sad radim kao inženjer u RT-RK. Radio sam u timu za uspostavljanje instrumentalizacije CISCO Polaris softvera za CISCO modeme i rutere. Trenutno sam u timu koji se bavi održavanjem i unapređivanjem GDB softvera za debagovanje kao i unapređivanjem kompajlerske infrastrukture LLVM sa ciljem poboljšanja dostupnosti debag informacija prilikom kompajlerske optimizacije softvera.