UNIVERZITET U BEOGRADU
MATEMATIČKI FAKULTET

Marina R. Nikolić

PRIKUPLJANJE I PRIKAZ PODATAKA O IZVRŠAVANJU PROGRAMA

master rad

Beograd, 2018.

Mentor:

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

Članovi komisije:

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

dr Milan Banković, docent
Univerzitet u Beogradu, Matematički fakultet

Datum odbrane: _____

Mentoru za predanost i pomoć, firmi za resurse, porodici i prijateljima za podršku

Naslov master rada: Prikupljanje i prikaz podataka o izvršavanju programa

Rezime: tekst apstrakta rada

Ključne reči: profajliranje, pokrivenost kôda, GCC, GCOV

Sadržaj

Glava 1

Uvod

1. kratak opis o čemu će biti reči u daljem tekstu
2. iako vidim da je popularno po master radovim ada se piše po poglavljima ovde (tipa, u poglavlju X je opisano to i to), ja bih uvod radije sročila kao pričicu koja prati rad
3. ovde bih dodala na samom početku i na samom kraju značaj teme kao takve i naravno značaj mog doprinosa (na kraju zbog efekta)

Glava 2

Analiza kôda, profajliranje i pokrivenost. Implementacija u programskom prevodiocu GCC

Razvoj softvera je znatno širi pojam od pisanja kôda. Obuhvata više, podjednako važnih segmenata, kao što su: planiranje, analiza i usklađivanje sa zahtevima klijenata, testiranje, analiza performansi, optimizacija, održavanje i mnogi drugi. Samo investiranjem u svaki ponaosob, može se proizvesti kvalitetan i dugotrajan softver. Njihova kompleksnost proporcijano raste sa značajem i složenošću krajnjeg proizvoda, iz čega proističe i važnost njihovog olakšavanja. Postoje brojne metodologije i tehnike koje su specijalizovane za vođenje procesa karakterističnih za rane faze razvoja, kao što su planiranja ili analize zahteva. Međutim, ovaj rad će se usresrediti na prikaz onih koje olakšavaju procese kasnog razvoja, pre svega testiranja i optimizacije. Za uspešno sprovođenje tih procesa, važan faktor je odabir tehnika koje će se primenjivati i jedinica kôda kome su oni najneophodniji, a kvalitetan odabir je uslovljen dobrim poznavanjem samog softvera, njegovih karakteristika i ponašanja. Takvu vrstu informacije obezbeđuje analiza programa.

2.1 Analiza programa

Analiza programa predstavlja automatizovani proces analiziranja raznih aspekata softvera, pre svega njegovog ponašanja, u različitim slučajevima upotrebe, u cilju olakšavanja procesa testiranja korektnosti, naročito eksterno nabavljenih delova softvera, procene performansi i optimizacije. Pruža korisne informacije o raspodeli

potrošnje resursa, čvorovima ekstremne potrošnje, potencijalnim kritičnim segmentima izvršavanja, korektnosti toka izvršavanja i slično. Poput projekta veštačke inteligencije, njen krajnji cilj je stvaranje „pametnog prevodioca", koji bi mogao automatski generisati efikasan, a pouzdan kôd. Značaj njenih trenutnih mogućnosti, kao i brzina kojom se unapređuje, ukazuju na veliku verovatnoću ostvarljivosti tog cilja. Analiza programa je veoma širok pojam, koji obuhvata veliki broj vrlo raznovrsnih metoda, ali se može veoma precizno podeliti na dva osnovna tipa. To su statička i dinamička analiza.

Statička analiza programa

Statička analiza programa [15] obuhvata sve metode i tehnike utvrđivanja ponašanja programa, za koje ga nije potrebno izvršiti. Sve procedure se vrše nad izvornim kôdom i, prikupljajući podatke o njegovoj strukturi, generišu korisne informacije o mogućim ishodima njegovog budućeg izvršavanja. Primer su mnogobrojne softverske metrike, koje na osnovu podataka o broju linija, klasa ili metoda, izračunavaju takozvani „statistički kvalitet" softvera.

Njena glavna prednost proističe upravo iz toga, što kôd nije potrebno izvršiti. Ovakvim ograničenjem se često odlikuje razvoj velikih i skupih softverskih sistema, gde se zbog materijalnih mogućnosti ne može vršiti testiranje svih manjih jedinica u realnom okruženju. Kao ilustrativan primer se može posmatrati razvoj softera za automatsko navođenje rakete i jedan manji segment tog razvoja koji predstavlja program za izračunavanje potrošnje goriva prilikom jednje vožnje. Testiranje korektnosti sastavne jedinice te veličine se u najvećoj meri vrši na simulatorima. Lansiranje prave rakete za potrebe ovakvog testiranja je ekonomski neopravdano, iako okruženje koje simulator pruža ne obuhvata sve alternativne slučajeve upotrebe.

Sa druge strane, ukoliko uzmemo u obzir činjenicu da vreme izvršavanja proizvoljnog programa može biti proizvoljno dugo, iz neneophodnosti izvršavanja, može se izvesti još jedna velika prednost statičke analize, a to je brzina. Faktor brzine čini osnovu ocene svakog pristupa.

Iz nenophpodnosti izvršavanja, proističe još jedna velika prednost statičke analize, a to je nepristrasnost. Nezavisnost od ulaznih podataka i okruženja, omogućava efikasnu detekciju graničnih slučajeva.

Osnovne mane softverskih metrika su uzrokovane uskom vezom njihovih tehnika sa statistikom kao naukom i predstavljaju nepreciznost i smanjenu informativnost o praktičnim slučajevima upotrebe. Rezultati nisu eksperimentalne prirode, već prika-

zuju teorijsko predviđanje ponašanja. Zbog toga se ne trebaju smatrati potvrdama ispravnosti ili performansi, već isključivo tretirati kao smernice pri razvoju.

Postoje i određene statičke metode koje su preciznije i mogu garantovati ispravnost kôda koji se razvija, poput simboličkog izvršavanja [16], proveravanja modela [18] ili apstraktne interpretacije [10]. Ove metode simuliraju ponašanje programa uzimajući u obzir i ulazne vrednosti, čime se povećava preciznost i informativnost. Međutim, uticaj realnih parametara okruženja, čija specifikacija nije u potpunosti poznata, se i dalje zasniva na predviđanju i statističkim informacijama o slučajevima upotrebe. Kao primer nedovoljeno potpune specifikacije se mogu posmatrati eksterno nabavljene komponente sa zatvorenim kôdom. Nepoznavanje svih alternativnih tokova upotrebe ili greške u dokumentaciji mogu prouzrokovati slabosti u modelima kreiranim ovim metodama.

Dinamička analiza programa

Dinamička analiza programa [12] obuhvata sve metode i tehnike prikupljanja podataka o programu tokom njegovog izvršavanja i utvrđivanja ponašanja programa na osnovu tih podataka. Procedure uglavnom započinju u fazi prevođenja, ali najvažniji deo se obavlja u toku i nakon izvršavanja. Pored strukture kôda i statičkih podataka, na njen ishod utiču i ulazne vrednosti, kao i parametri okruženja. Testovi jedinica kôda, sistemski testovi i testovi prihvatljivosti koriste isključivo ovaj vid analize programa.

Sve njene glavne prednosti u odnosu na statičku analizu, proističu iz uticaja „realnih parametara". Određene mane softverskih rešenja ispoljavaju se samo u toku rada tog softvera, a mnoge i proističu upravo iz spoljnih faktora ili veze sa njima. Statistički savršen softver koji je nedovoljno primenljiv u praksi, predstavlja jedan od tri osnovna neuspeha prilikom razvoja softvera [6]. Marketinška istraživanja, analize zahteva korisnika i detaljni popisi slučajeva upotrebe se primenjuju u ranim fazama razvoja softvera u cilju zaštite od ove vrste neuspeha. Međutim, pojedini faktori okruženja, poput vrednosti jedne jedinice iz skupa obrade, čiji uticaj se zanemaruje kao dozvoljeno odstupanje, greška zaokruživanja ili usled efekta mase, tzv. „lažne pozitivne ili negativne vrednosti", se ne mogu detektovati metodama koje se baziraju na statistici. Kao ilustrativan primer može se posmatrati testiranje uspešnosti prenosa bitova kroz određeni fizički medijum i sledeći rezultati testiranja: 1 000 000 000 bitova koji su uspešno stigli na destinaciju i 10 izgubljenih bitova. Procenat neuspeha iznosi 0.000001%, što se zaokruživanjem na 5 ili manje decimala

svodi na 0%. Na osnovu ovog podatka, može se zaključiti da je testiranje završeno uspešno, i pritom potpuno zanemariti značaj izgubljenih delova informacije.

Posledice zanemarivanja uticaja pojedinačnih slučajeva obuhvataju brojna prilagođavanja i održavanja u kasnim fazama razvoja, koja se neretko završavaju odustajanjem od razvoja nakon isteka novčanih sredstava ili pronalaska kvalitetnijeg rešenja. Zbog toga su testiranja u realnom okruženju veoma važna, a kako su po svojoj prirodi ograničena resursima, važno je i iz njih ekstrahovati što više informacija za naredne iteracije razvoja. Njih obezbeđuje dinamička analiza programa.

Važna prednost dinamičke analize je i univerzalnost, koja proističe iz činjenice da se sve tehnike primenjuju na izvršnu verziju, bez neophodnog prisustva izvornog kôda. Oblast primene je šira, jer obuhvata i programe sa „zatvorenim" kôdom. Pisanje celokupnog kôda softvera je skupo, kako u ekonomskom, tako i u pogledu utrošenog vremena, zbog čega ne predstavlja dovoljno kompetetivan način proizvodnje. Ovaj princip nije karakteristika samo softverske industrije, već je globalna odlika industrije kao grane privrede. Kao ilustrativan primer se može posmatrati javni prevoz građana i porediti cena jedne autobuske karte u odnosu na cenu goriva i održavanja automobila, na relaciji od nekoliko kilometara. Ukupna cena jednog prevoza se ravnomerno raspoređuje na više putnika, čime je pojedinačna cena po putniku znatno manja. Sa druge strane, prevoznik nema obavezu da proizvod ustupi za tačnu cenu pojedinačnog dela, količnika cene vožnje i broja putnika, iz čega proističe njegova zarada. Postavljanje previsoke cene u cilju maksimalne zarade može prouzrokovati manjak interesovanja za proizvod, usled neisplativosti korisniku, te njen izbor mora biti izbalansiran rezultatima pažljivog proučavanja tržišta. Cena održavanja automobila je dodata u ilustraciju, u cilju naglašavanja troškova održavanja softvera, koje često predstavlja najveći materijalni rashod razvoja. U razvoju softvera, ovaj princip se ogleda u eksternom nabavljanju komponenti, u kom slučaju se često može kupiti samo izvršna verzija. Izvorni kôd predstavlja poslovnu tajnu proizvođača. Procene kvaliteta pre integracije, kao i testiranje kompatibilnosti sa ostatkom softvera, stoga se mogu obaviti jedino dinamičkim pristupom.

Najveća mana ovog pristupa jeste potencijalni osećaj lažne sigurnosti. To je, u određenoj meri, neizostavna stavka svakog testiranja. Neiskusni razvojni timovi se mogu previše osloniti na rezultate analize i time prevideti činjenicu da ona, kao automatizovani proces, ne može garantovati stoprocentnu ispravnost. Alati koji je vrše su takođe softverski proizvodi, i samim tim jednako podložni greškama koliko i kôd koji se njima analizira.

2.2 Profajliranje i pokrivenost kôda

Posebno mesto u tehnikama dinamičke analize ima profajliranje, i njemu će ovaj rad biti u potpunosti posvećen.

Profajliranje

Profajliranje [13, 19] predstavlja prikupljanje raznih podataka iz izvršavanja programa u realnom ili simuliranom okruženju, koji pružaju uvid u tok i performanse rada programa. Obradom ovih podataka, dobijaju se vredne informacije o vremenskim i memorijskim zahtevima programa, složenosti i iskorišćenosti pojedinih delova kôda i slično. Rezultati rada alata za profajliranje predstavljaju korisne smernice za procese testiranja i optimizacije, jer ukazuju na delove kôda kojima su oni najneophodniji.

Ulazne vrednosti i parametri okruženja, zajedno sa kôdom programa, jedinstveno određuju tok izvršavanja. Uočavanje pozitivnih podataka o izvršavanju van predviđenog toka, ili negativnih u njegovoj unutrašnjosti, za unapred određen slučaj upotrebe, je stoga dobar pokazatelj da se u kôdu nalaze greške.

Kao ilustrativan primer mogu se posmatrati program za obradu teksta i slučaj upotrebe koji se sastoji iz tri koraka: učitavanje teksta, podebljavanje jedne reči i sačuvavanje izmena. Predviđen tok izvršavanja obuhvata prolazak kroz pet funkcija: otvaranje željenog fajla, prikazivanje teksta na ekranu, podebljavanje odabrane reči, memorisanje promena i zatvaranje programa. Na osnovu ovog toka, izvodi se teorijski zaključak da se funkcija koja vrši podebljavanje teksta izvršila, dok funkcija koja iskrivljuje tekst nije. Ukoliko eksperimentalni podaci, poreklom iz konkretnog izvršavanja, nisu u skladu sa teorijskom pretpostavkom, već potvrđuju izvršavanje funkcije za iskrivljavanje ili negiraju izvršavanje funkcije za podebljavanje teksta, može se zaključiti da se program ne izvršava pravilno. Efekat ove dve funkcije se može u određenim slučajevima primetiti i na osnovu prikaza na ekranu, međutim izostanak efekta memorisanja izmena gotovo sigurno neće biti uočen u odgovarajućem trenutku.

Profajliranje pruža i dodatnu olakšicu za budući proces „debagovanja", sužavanjem oblasti pretrage. Detekcija memorijski ili vremenski izrazito zahtevnih segmenata, kao i segmenata koji se veoma često izvršavaju usmerava pažnju razvojnog tima na neophodnost optimizacije, pritom takođe obezbeđujući dodatnu informaciju gde je ona i koliko potrebna. Poređenjem performansi različitih verzija kda, može se

izvršiti dobra procena kvaliteta i odabir odgovarajućeg algoritma u ranim fazama, kada je njegova zamena u velikoj meri jeftinija. Smernice koje profajleri daju mogu znatno „očistiti" kôd od nepotrebnih grananja, logički neiskorišćenih promenljivih, „mrtvog kôda" i sličnih propusta. Stoga značajno olakšavaju i proces refaktorisanja kôda. Vrši se alatom koji se naziva profajler i sastoji se od tri usko spregnute faze: instrumentalizacija, prikupljanje i obrada podataka.

Faze profajliranja

Instrumentalizacija [7] kôda predstavlja ubacivanje dodatnih instrukcija u program sa ciljem merenja karakteristika programa. Instrukcije predstavlju kôd inicijalizacija određenih dodatnih struktura za instrumentalizaciju i pravila za njihovo popunjavanje. Dodatne strukture imaju ulogu skladišta za metapodatke, a za popunjavanje je zadužen sam instrumentalizovani program. Time se stvara opterećenje i smanjuju performanse, ali je, iz više razloga, najpouzdanije i najoptimalnije moguće rešenje. Prvenstveno, iz ugla bezbednosti. Neograničen pristup internim podacima jednog programa ne sme imati niko sem njega samog, jer bi se time otvorile brojne mogućnosti za razvoj novog malicioznog softvera koji bi zloupotrebio ovaj bezbednosni propust, bilo napadajući alat za instrumentalizaciju, bilo poruke koje razmenjuje sa instrumentalizovanim programom. Zaštita u vidu šifrovanja bi zahtevala dodatno trošenje resursa, što nije isplativo. Pored bezbednosnog aspekta, bitan faktor je i sinhronizacija. U sistemu sa eksternim alatom, usklađivanje čitanja i pisanja memorijskih segmenata dodeljenih programu bi iziskivalo dodatno trošenje procesorskog vremena i memorije, a i zaključavanje bi povećalo vremensku složenost.

Faza prikupljanja podataka obuhvata: čitanje dodatnih struktura sa metapodacima, njihovo konvertovanje u pogodniji oblik i eksterno skladištenje. Da bi oblik bio pogodan, neophodno je da predstavlja dobar balans između veličine, koja treba biti što manja, i informativnosti, koja treba biti što veća. Ukoliko neki podaci mogu da se izvedu iz ostalih, oni se eliminišu. Lokacija podataka u eksternom skladištu predstavlja memorijski besplatan podatak, koji omogućava dodatnu kompresiju bez gubitka na informativnosti. Dovoljno je upisati vrednost željenog podatka, jer je njegovo značenje precizno određeno redosledom upisa bajtova u eksterno skladiste, odnosno položajem bajtova podatka u odnosu na bajtove specijalnih oznaka za razgraničavanje. Ova faza je takođe poverena samom programu, iz istih razloga kao i instrumentalizacija.

Produkt prve dve faze su sirovi podaci, koji u sebi nose informacije o karakte-

ristikama programa u realnim slučajevima upotrebe, ali kako se podaci prikupljaju samo ako program ima dodatnu funkciju da u toku rada prikuplja i svoje metapodatke, ne može se obezbediti potpuna preciznost informacija. Uticaj se ne može u potpunosti ukloniti, međutim mora biti sveden na granicu prihvatljivosti. Ispravna instrumentalizacija ne sme uticati na funkcionalnost programa.

Poslednja faza predstavlja obradu sirovih podataka do korisne informacije. Krajnji proizvod predstavlja jedan ili više izveštaja u formatu pogodnom prvenstveno za razvojni tim, ne za računar. Osnovne karakteristike izveštaja treba da budu: uniformnost, preglednost, povišena (vraćanje izvedenih podataka) ili snižena informativnost (filtriranje podataka po kategorijama), unija pojedinačnih i statističkih prikaza i slično. Ovu fazu obično obavljaju eksterni alati, jer je potpuno nezavisna od izvršavanja programa i njegove interne memorije. U zavisnosti od toga koje se karakteristike mere i potreba korisnika, krajnji izveštaji variraju od jednorečeničnih ispisa, preko kolekcija fajlova, do interaktivnih aplikacija. Mogu se meriti razne karakteristike, poput na primer memorijskih zahteva ili tragova izvršavanja, ali po informativnosti i mogućnostima kombinovanja sa drugim informacijama, ističe se pokrivenost kôda.

Pokrivenost kôda

Pokrivenost kôda [8, 9, 20, 11, 14, 17] predstavlja „stepen izvršenosti kôda". Izračunava se kao odnos broja izvršenih i neizvršenih linija, blokova, grana ili funkcija i izražava se u procentima. U strogom smislu, pokrivenost kôda je jedan jedini broj, dobijen merenjem nad celim sistemom. Taj broj je sam po sebi veoma informativan. Što je pokrivenost manja, to je verovatnoća da u kôdu postoje ozbiljne greške u logici veća.

Međutim, nakon merenja na celom skupu, poželjno je izvršiti i merenja na manjim segmentima: komponentama, klasama ili funkcijama, kako bi se detektovali propusti globalne informacije. Na primer, ukoliko je stil pisanja kôda takav da se po fajlovima grupišu slični metodi iz različitih klasa, ovakvim pristupom mogu se bolje detektovati slabo ili nimalo korišćene klase, ili objekti koji se prave i uništavaju bez da utiču na ukupnu funkcionalnost. Podaci o izvršavanju konkretnih linija, mogu doprineti pronalasku petlji koje se izvršavaju veliki broj puta, logički neiskorišćenih delova koda ili bespotrebnih grananja koja se svedu na isti krajnji rezultat. Stoga, pokrivenost kôda ne treba shvatati samo u svom najužem smislu, već maksimalno

iskoristiti sve njene mogućnosti. Uzroci neočekivane pokrivenosti mogu biti veoma raznovrsni. U daljem tekstu biće predstavljeno nekoliko primera.

Stariji softver koji se duže vreme održava, neretko sadrži visok procenat kôda iz prethodnih verzija, koji je vremenom izgubio svoju funkcionalnost. Smenom razvojnih timova, naročito u okruženjima koja ne podržavaju detaljno dokumentovanje učinka, često se gube informacije o funkcionalnosti pojedinih delova kôda. Usled nedostatka informacija, novi razvijaoci se često ne odlučuju na eliminisanje ili zamenjivanje delova kôda, već se uglavnom vrši dodavanje. Funkcije ili klase, a neretko i čitave komponente, tako postaju „mrtav kôd", koji otežava procese održavanja i „debagovanja". Kôd ovakvog softvera ima naročito malu pokrivenost.

Važan faktor prilikom razvoja softvera predstavlja i balans između preciznosti i brzine. Preopterećivanje programa ispitivanjem malo verovatnih alternativnih slučajeva, dovodi do slabljenja performansi. Pored toga, suvišna grananja mogu proizvesti ogromne količine mrtvog kôda, od linija pa do čitavih klasa ili komponenti pisanih isključivo za te specijalne slučajeve. To znatno otežava održavanje kôda, debagovanje i refaktorisanje. Mala pokrivenost može biti dobar pokazatelj, a podaci izvršavanja linija odrediti preciznije lokaciju problema.

Gotovo sve današnje sisteme odlikuje konkurentno ili paralelno izvršavanje. Njima se postiže značajan porast efikasnosti, ali i povećava broj potencijalnih problema koji mogu nastati prilikom izvršavanja, poput živih i mrtvih zaključavanja, ili trke za resursima. Ovi problem mogu uzrokovati blokiranje ili prestanak rada celog sistema, a njihovo blagovremeno otkrivanje je veoma teško. Algoritam rada procesora određuje koji će se proces, kada i koliko izvršavati, a programer može jedino implementirati neke vidove zaštite atomičnosti operacija ili nametanja prioriteta procesa. Međutim, i pored zaštitnih mehanizama, dešava se da se nekim procesima ne dodeli vreme na procesoru. Takvi kôdovi imaju izuzetno niske pokrivenosti, a najbolji pokazatelj su pokrivenosti pojedinačnih izvršavanja koje iznose nula procenata. Prilikom rada sa nitima, niske pokrivenosti mogu biti simptom i preopterećenosti.

Najozbiljniji problemi koji uzrokuju malu pokrivenost su „greške u logici". One mogu varirati, od pogrešno definisanih uslova u granama ili petljama do potpuno promašenih algoritama. Neočekivana pokrivenost je dobar pokazatelj da u kôdu ima ovakvih grešaka. Pokrivenost manja od očekivane može, na primer, biti uzrokovana pozivom pogrešnih funkcija, ulaskom u neproduktivnu granu ili prevremenim izlaskom iz programa. Veća pokrivenost od očekivane može biti simptom nepravilnog rada uslova u naredbi grananja, loše konstruisanih provera u kôdu i slično. Kako

uzroci mogu biti veoma raznovrsni, dobro je pored pokrivenosti celog softvera, meriti i pokrivenosti na segmentima. Kombinovanjem svih rezultata, sužava se oblast pretrage i lako locira greška u logici.

Potvrda ispravnosti kôda pre nego što ode u produkciju, najčešće su samo dobri rezultati testiranja. Međutim, na ishod testova ne utiču samo karakteristike softvera koji se testira, već i njihova ispravnost. Testovi se često sami ne testiraju dovoljno dobro, što može dovesti do ozbiljnih posledica. Lažan negativan rezultat može uzrokovati bespotrebnu potrošnju vremena i novca na traženje nepostojeće greške u kôdu. Lažan pozitivan rezultat može imati još i ozbiljnije posledice, čija težina zavisi od važnosti samog softvera. Stoga je veoma korisno primeniti tehniku određivanja pokrivenosti kôda i na testove, a ne samo na primarni softver. Mala pokrivenost je dobar indikator da u kôdu postoje segmenti koji nisu testirani, a koji su samim tim potencijalna opasnost.

Računanjem pojedinačnih pokrivenosti možemo doći i do informacija o često korišćenim segmentima kôda. One umnogome olakšavaju razvojnom timu prilikom donošenja odluka vezanih za vremensku optimizaciju. Kombinovanjem sa podacima za pojedinačne linije koje alociraju memoriju, mogu se pronaći memorijski zahtevni segmenti koji su dobri kandidati za prostornu optimizaciju.

Najsitniji podaci, poput podataka o izvršavanju pojedinih linija ili blokova se mogu koristiti i za refaktorisanje. Uklanjanjanje mrtvog kôda ili razbijanje preopterećenih funkcija, su samo neki od primera refaktorišućih procesa koji su olakšani uz informacije o pokrivenosti kôda, a čije sprovođenje umnogome pospešuje održavanje ili dalji razvoj.

Raznovrsnost gore navedenih primera dokazuje veliki značaj i potencijal pokrivenosti kôda. Stoga će na nju biti u poptunosti skoncentrisan ostatak ovog rada.

Poznatiji prevodioci, poput GCC-a [3], ICC-a [5] i Clang-a [1] u određenoj meri poseduju ugrađenu podršku za testiranje pokrivenosti kôda. Projekat LLVM trenutno prednjači u raznovrsnosti, jer pruža i mogućnost merenja pokrivenosti u toku izvršavanja. Sa druge strane, GCC trenutno podržava samo testiranje pokrivenosti nakon izvršavanja programa, ali ga odlikuju znatno bolje performanse, pre svega u pogledu memorijske zahtevnosti. Prikupljanje i obrada podataka o pokrivenosti kôda u toku izvršavanja programa korišćenjem tehnika GCC-a, kombinuje dobru ideju projekta LLVM i dobre tehnike prevodioca GCC, čime prednjači i u oblasti mogućnosti i u oblasti performansi. U okviru projekta na kome je utemeljen ovaj rad, izvršena je detaljna analiza postojećih mogućnosti u okviru prevodioca GCC i

implementirana je podrška za prikupljanje podataka u toku izvršavanja, kao i novi, unapređeni alat za njihov vizuelni prikaz.

2.3 Postojeća rešenja u okviru GCCa

Programski prevodilac GCC sadrži ugrađenu podršku za određivanje pokrivenosti kôda, integrisanu u statičku biblioteku za prikupljanje podataka po imenu libgcov i alat za vizuelni prikaz podataka GCOV [4, 2].

Metapodaci izvršavanja čuvaju se u deljenoj memoriji programa, u listi posebnih, globalno definisanih struktura tipa `gcov_info`, čija se inicijalizacija ugrađuje u binarni kôd prevođenjem sa posebnim flegovima za instrumentalizaciju: `-fprofile-arcs -ftest-coverage`. Flegovi se navode tokom prevođenja izvornog kôda do objektnog fajla, a simboli koji se njima unose razrešavaju se kasnije u fazi linkovanja.

Pored ubacivanja instrukcija u binarni kôd programa, flegovi za instrumentalizaciju obavljaju još jednu važnu aktivnost, a to je kreiranje dodatnog fajla, odmah pored njemu odgovarajućeg fajla izvornog kôda, sa ekstenzijom gcno (GCov NOtes file). To je relativno mali, binarni fajl, koji sadrži sve neophodne statičke informacije o strukturi izvornog kôda čijim prevođenjem nastaje. Njegova glavna uloga jeste da predstavlja struktuni kostur budućeg finalnog proizvoda alata, koji će se kasnije nadograditi podacima dobijenim dinamički u toku izvršavanja. Format gcno fajla je utvrđen zajedničkim standardom GCC-a i alata GCOV, koji je specijalizovan i za njegovo tumačenje. Veoma je nečitljiv ljudskom oku, što je uzrokovano maksimalnim stepenom kompresije podataka. Korišćenje specijalnih oznaka, korišćenje pozicije kao interpretacije podatka, kao i pažljivo odabrani minimalni skup potrebnih informacija o strukturi, samo su neke od tehnika kompresije korišćenih u cilju maksimalne štednje memorije. Posebno je važno napomenuti da je čuvanje podataka o strukturi u vidu eksternih binarnih fajlova osnovni uzrok boljih memorijskih performansi GCC instrumentalizacije u odnosu na Clang-ovo profajliranje, pomenute na kraju prethodnog poglavlja, jer umanjuje rizik od eksplozije veličine samog programa. Uvećanje izvršnog fajla do veće vrednosti od memorijske količine koja je za njega predviđena na sistemu, može dovesti do nepravilnosti u radu, a kako memorijski zahtevniji instrumentlizovani program se ponaša drugačije od regularnog, rezultati testiranja neće odražavati realno stanje. Naročito, na sistemima sa veoma ograničenim memorijskim prostorom, GCC instrumentalizacija je jedina moguća.

Svaka struktura tipa `gcov_info` iz liste, odgovara tačno jednom instrumentalizovanom objektnom fajlu koji učestvuje u izgradnji programa. Pored osnovnih podataka poput imena fajla ili verzije alata, svaka struktura tipa `gcov_info` sadrži i pokazivač na niz struktura tipa `gcov_fn_info`, u kojima se skladišti po nekoliko posebnih brojača za svaku funkciju tog fajla. Na osnovu vrednosti u njima, može se konstruisati podatak o količini izvršavanja bilo koje jedinice kôda u okviru te funkcije. Tokom rada programa, vrednosti u brojačima se konstantno ažuriraju, i u svakom trenutku odražavaju realno stanje izvršavanja. Ti podaci predstavljaju jezgro informacije o pokrivenosti kôda, ali njih eksterni alat poput GCOV-a ne može direktno koristiti iz više razloga. Prvi razlog je bezbednosne prirode, i velikom merom je obrazložen u prethodnom poglavlju. Eksternim alatima se ni u kom slučaju ne treba omogućiti čitanje internih podataka programa. Detaljno je obrazloženo i pitanje sinhronizacije pisanja i čitanja, koje važi i u ovom slučaju. Naposletku, ovim pristupom bi podaci imali poreklo samo iz jednog izvršavanja, što bespotrebno ograničava mogućnosti alata.

Rešenje koje je trenutno implementirano u GCC, upravo iz tih razloga, sadrži jednog „posrednika" između instrumentalizovanog programa i eksternih alata, a to je libgcov. Biblioteka, po svojoj prirodi, se ugrađuje u program i time postaje deo njega, što joj daje ekskluzivno pravo pristupa njegovoj deljenoj memoriji. Njen osnovni zadatak je ekstrakcija podataka iz strukture `gcov_info` i njihovo konvertovanje u oblik pogodan za obradu eksternim alatom. Statička funkcija `gcov_at_exit` preuzima vrednosti brojača, računa sumarne i statističke podatke i sve zajedno upisuje u posebni binarni fajl sa ekstenzijom gcda (GCov DAta file) u unapred utvrđenom formatu i na unapred utvrđenoj lokaciji. Format gcda fajla je takođe veoma nečitljiv, usled primene sličnih tehnika za kompresiju, kao u slučaju gcno fajla. Generiše se uvek pored objetktnog fajla kome odgovara, u slučaju da fajl sa istim imenom i ekstenzijom već ne postoji na toj lokaciji. U slučaju višestrukog pokretanja programa, vrednosti iz predhodnih izvršavanja već se nalaze u gcda fajlu, te se on samo ažurira, a za sumiranje starih i novih podataka, zadužena je druga funkcija po imenu `__gcov_merge_add_`. Stoga, za zanemarivanje starih podataka, neophodno je premestiti ili ukloniti prethodni gcda fajl pre novog pokretanja.

Na kraju izvršavanja instrumentalizovanog programa, svi podaci potrebni za informisanje razvojnog tima o pokrivenosti njihovog kôda, nalaze se na fajl sistemu i mogu se pakovati, premeštati i skladištiti. To je veoma korisna činjenica, jer pruža nove mogućnosti kombinovanja rezultata različitih merenja. Ukoliko postoji

potreba da se neki test prekine na određeno vreme i započne novi, gcda fajlovi prvog testa se mogu spakovati na drugu lokaciju, čime će se za drugi test generisati novi, i ponovo prebaciti pored objektnih pred nastavak prvog testa. Za ekonomično skladištenje mogu se koristiti i kompresovane arhive ili eksterni memorijski mediji. Međutim, njihova osnovna funkcija je da predstavljaju ulazne parametre za alat GCOV, koji na osnovu njih kreira tekstualni izveštaj, pogodniji za interpretaciju od strane razvojnog tima.

Za generisanje jednog izveštaja, potrebno je alatu GCOV proslediti u vidu argumentata: jedan izvorni fajl, jedan odgovarajući strukturni fajl: gcno i jedan fajl sa vrednostima brojača: gcda. Poseban tekstualni fajl sa ekstenzijom gcov se kreira za svaki instrumentalizovani fajl izvornog kôda. Izveštaj se sastoji od celokupnog sadržaja izvornog kôda, uz dodatak jedne vrednosti ispred svake izvršne linije, koja predstavlja broj puta koliko se ta linija izvršavala. Ukoliko se linija nije izvršila nijednom, ispred nje se stavlja posebna oznaka sastavljana od pet simbola tarabice. Prvih nekoliko linija izveštaja rezervisano je za statističke podatke o imenima fajlova od kojih je kreiran, dok se na standardni izlaz štampa najvažnija vrednost: odnos broja izvršenih linija i ukupnog broja linija, odnosno pokrivenost kôda. Na slici 2.1, prikazan je primer osnovnog GCOV izveštaja, koji se generiše pozivom alata bez dodatnih opcija. Korišćenjem flegova u pozivu alata, izveštaj se može unaprediti i podacima o blokovima, granama, funkcijama i slično.

Čitanje podataka iz deljene memorije programa i njihovo skladištenje u gcda fajlove, odvija se kao poslednja instrukcija programa pre kraja izvršavanja (at_exit). Usled toga, iako analiza GCOV alatom nije striktno vezana za vremenski tok izvršavanja, ne može se vršiti pre kraja programa. Ovo je veliki nedostatak, koji u nekim specifičnim slučajevima može poptuno onemogućiti proveravanje pokrivenosti kôda. Programi kod kojih je na primer vreme rada izuzetno dugo ili su podaci dostupni i/ili korisni samo tokom rada, kao na primer sistemi za rad u realnom vremenu, serveri ili operativni sistemi, ne mogu koristiti statičku instrumentalizaciju na kraju izvršavanja. Ukoliko imaju ograničene memorijske mogućnosti, što je često slučaj na ovakvim sistemima, ne mogu koristiti ni dinamički pristup programskog prevodioca projekta LLVM. Za obezbeđivanje informacija o pokrivenosti kôda ovakvih programa, neophodno je proširiti mogućnosti instrumentalizacije GCC-a na prikupljanje podataka u toku izvršavanja.

Prikaz u vidu pojedinačnih izveštaja za svaki fajl izvornog kôda, takođe poseduje određene mane. Svaki izveštaj se nalazi na posebnoj lokaciji u okviru direktorijuma

```
    -:    0:Source:1.c
    -:    0:Graph:1.gcno
    -:    0:Data:1.gcda
    -:    0:Runs:1
    -:    0:Programs:1
    -:    1:#include <stdio.h>
    -:    2:
    1:    3:void pozdrav(){
    -:    4:
    1:    5:    printf("Dobar dan! Dobrodosli u test program!\n");
    -:    6:
    1:    7:}
    -:    8:
    1:    9:int oprostaj(){
    -:   10:
    1:   11:    printf("Dovidjenja! Ugodan dan!\n");
    -:   12:
    1:   13:}
    -:   14:
    1:   15:int main(){
    -:   16:
    1:   17:    pozdrav();
    -:   18:
    1:   19:    int a = 1;
    1:   20:    int b = 2;
    -:   21:
    1:   22:    if(a==b){
#####:   23:        printf("Netacno: 1=2\n");
    -:   24:    }
    -:   25:    else{
    1:   26:        printf("Tacno: 1!=2\n");
    -:   27:    }
    -:   28:
    1:   29:    oprostaj();
    -:   30:
    1:   31:    return 0;
    -:   32:
    -:   33:}
```

Slika 2.1: osnovni GCOV izveštaj

projekta, što otežava njihov pregled kao celine. Dodatne informacije, poput onih o pokrivenosti pojedinačnih funkcija, koje se dobijaju dodavanjem opcija u poziv alata, kao i vrednost pokrivenosti fajla, se ne nalaze u okviru izveštaja, već samo ispisa na standardni izlaz, što uzrokuje potencijalni gubitak tih informacija. Vrednost pokrivenosti kôda čitavog projekta se ne izračunava, čime je krajnji rezultat oslabljen za još jednu bitnu informaciju. Potreba za prevazilaženjem ovih mana je uticala na formiranje ideje o novom alatu za vizuelni prikaz GCOV statistike, koji je izgrađen u okviru ovog projekta.

Glava 3

Zacetak ideje i trnoviti putevi

3.1 Ideja – dinamicki pristup

1. Uvod u moj projekat
2. Šta je ovde drukčije i bolje
3. Samo teorija, bez detalja kako tačno radi šta

3.2 Razmatrana rešenja

1. Dva puta koja su se predamnom bejaše otvorila – da li napadati GCOV alat ili menjati biblioteku
2. Kako i zašto sam odabrala ovo što sam odabrala
3. Lepa pričica da se pokaže da se ipak ulagalo malo mozga u projekat

Glava 4

Implementacija i analiza

4.1 Implementacija

1. Biblioteka
2. GUI (signali za prikupljanje podataka, generisanje izvestaja)

4.2 Demonstracija i uputstvo za upotrebu

1. primer rada biblioteke I GUI-ja sa slikama
2. dobar moment da se naglasi da rad ima primenu na bilo koji kod
3. ne znam jel smem pominjati digitalnu i ko ga sad koristi

4.3 Performanse

1. da li smo postigli cilj
2. da li možemo isto što i pre, pa i više
3. memorija I bezbednost – test sa Valgrindom
4. složenost – vremenska i prostorna
5. jednostavnost upoterebe
6. ne bi bilo loše ovde pomenuti LLVM i njihovu runtime instrumentalizaciju

4.4 Primena

1. Gde bi sve ovo moglo da radi
2. Ne znam koliko smem odavati na čemu je testirano I na čemu radi
3. Ideja: Ako bi se ovakav jedan alat unapredio I ugradio npr u pejsmejker da signalizira da nešto ne radi kako treba, to što je runtime prikupljanje moglo bi nekome spasiti život

Glava 5

Zaključak

1. Šta je urađeno
2. Koji je značaj toga što je urađeno (gde sad radi – onliko koliko smem da kazem)
3. Šta bi još moglo da se uradi:
    a) Ideja: Ako bi se ovakav jedan alat unapredio I ugradio npr u pejsmejker da signalizira da nešto ne radi kako treba, to što je runtime prikupljanje moglo bi nekome spasiti život
    b) Moze mala komparacija sa LLVMom – tipa da se analizira sta je dobro i da se malo unapredi po ugledu na LLVM

Bibliografija

[1] Clang: a c language family frontend for llvm. https://clang.llvm.org/.

[2] Code coverage using gcov. https://web.archive.org/web/20140409083331/http://xview.net/pape

[3] Gcc, the gnu compiler collection. https://gcc.gnu.org.

[4] Gcov official site. http://gcc.gnu.org/onlinedocs/gcc/Gcov.html.

[5] Intel® parallel studio xe 2018: Getting started with the intel® c++ compiler 18.0 for linux*. https://software.intel.com/en-us/get-started-with-cpp-compiler-18.0-for-linux-parallel-studio-xe-2018.

[6] Razvoj softvera - materijali sa predavanja. http://poincare.matf.bg.ac.rs/ smalkov/files/rs.r290.2018/public/Predavanja/Razvoj

[7] Source code instrumentation overview. https://www.ibm.com/support/knowledgecenter/SSSHU

[8] Paul Ammann and Jeff Offutt. Introduction to software testing. *Cambridge University Press*, 2016.

[9] R Brader, H Hilliker, and A Wills. Unit Testing: Testing the Inside. *Microsoft Developer Guidance*, 2013.

[10] Ozren Demonja, Stefan Maksimović, and Marko Crnobrnja. Apstraktna interpretacija. http://poincare.matf.bg.ac.rs/ milena/msnr/2017/12/09_

[11] A Glower. In pursuit of code quality: Don't be fooled by the coverage report. *IBM Developer Works blog post*, 2006.

[12] V. Gupta. Measurement of Dynamic Metrics Using Dynamic Analysis of Programs. *APPLIED COMPUTING CONFERENCE (ACC '08), Istanbul, Turkey*, 2008.

[13] A. Homescu. Profile-guided automated software diversity. *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO). IEEE Computer Society*, 2013.

[14] B Marick. How to misuse code coverage. *Proceedings of the 16th Interational Conference on Testing Computer Software*, 1999.

[15] F Nielson, H. R. Nielson, and C Hankin. Principles of program analysis. *Springer Science & Business Media*, 2015.

[16] Ljubica Peleksić and Milica Kojičić. Simboličko izvršavanje. http://webmail.matf.bg.ac.rs/ milena/msnr/2016/12/11_VerifikacijaSoftvera_PeleksicKojicic.pdf.

[17] A Piziali. "Code coverage," in Functional verification coverage measurement and analysis. *Springer Science & Business Media*, 2007.

[18] Nikola Vlahovic, Mišić Petar, and Muljaić Aleksandar. Proveravanje modela. http://poincare.matf.bg.ac.rs/ milena/msnr/2017/10/03_ProveravanjeModelaVlahovicMisicMuljaic.pdf.

[19] Milena Vujošević Janičić. Verifikacija softvera. http://www.programskijezici.matf.bg.ac.rs/vs/predavanja/03_dinamicka_analiza/03_dinamicl

[20] L William, B Smith, and S Heckman. Test Coverage with EclEmma. *Technical Report Raleigh*, 2008.

Biografija autora

Marina Nikolić (*Sombor, 17. decembar 1992.*) je ...