УНИВЕРЗИТЕТ У НОВОМ САДУ
ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА У НОВОМ САДУ

Горан Вуков

Један приступ унапређењу подршке за прављење профила у преводиоцу LLVM ради прикупљања података током извршавања програма

МАСТЕР РАД

Нови Сад, 2017

УНИВЕРЗИТЕТ У НОВОМ САДУ ● **ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА**
21000 НОВИ САД, Трг Доситеја Обрадовића 6

КЉУЧНА ДОКУМЕНТАЦИЈСКА ИНФОРМАЦИЈА

Редни број, **РБР**:			
Идентификациони број, **ИБР**:			
Тип документације, **ТД**:		Монографска документација	
Тип записа, **ТЗ**:		Текстуални штампани материјал	
Врста рада, **ВР**:		Дипломски – мастер рад	
Аутор, **АУ**:		Горан Вуков	
Ментор, **МН**:		проф. др. Никола Теслић	
Наслов рада, **НР**:		Један приступ унапређењу подршке за прављење профила у преводиоцу LLVM ради прикупљања података током извршавања програма	
Језик публикације, **ЈП**:		Српски / латиница	
Језик извода, **ЈИ**:		Српски	
Земља публиковања, **ЗП**:		Република Србија	
Уже географско подручје, **УГП**:		Војводина	
Година, **ГО**:		2017	
Издавач, **ИЗ**:		Ауторски репринт	
Место и адреса, **МА**:		Нови Сад; трг Доситеја Обрадовића 6	
Физички опис рада, **ФО**: (поглавља/страна/ цитата/табела/слика/графика/прилога)		8/37/10/0/20/0/0	
Научна област, **НО**:		Електротехника и рачунарство	
Научна дисциплина, **НД**:		Рачунарска техника	
Предметна одредница/Кључне речи, **ПО**:		покривеност кода, LLVM, инструментализација вођена профилисањем	
УДК			
Чува се, **ЧУ**:		У библиотеци Факултета техничких наука, Нови Сад	
Важна напомена, **ВН**:			
Извод, **ИЗ**:		У оквиру овог рада представљено је једно побољшање везано за повећање потребног складишног простора приликом прављења профила у преводиоцу ЛЛВМ ради прикупљања података током извршавања програма повезивањем *clang.rt_profile* библиотеке као дељене. Такође је представљено једно решење чувања података о извршавању кода у произвољном тренутку. Испитивањем решења је постигнуто смањење величине преведеног испитног програмског кода.	
Датум прихватања теме, **ДП**:			
Датум одбране, **ДО**:			
Чланови комисије, **КО**:	Председник:	проф. др. Горан Станојевић	
	Члан:	доц. др. Милан Бјелица	Потпис ментора
	Члан, ментор:	проф. др. Никола Теслић	

UNIVERSITY OF NOVI SAD ● **FACULTY OF TECHNICAL SCIENCES**
21000 NOVI SAD, Trg Dositeja Obradovića 6

KEY WORDS DOCUMENTATION

Accession number, **ANO**:			
Identification number, **INO**:			
Document type, **DT**:		Monographic publication	
Type of record, **TR**:		Textual printed material	
Contents code, **CC**:		Master Thesis	
Author, **AU**:		Goran Vukov	
Mentor, **MN**:		PhD Nikola Teslic	
Title, **TI**:		One approach to improve program execution data profile suppport in LLVM compiler	
Language of text, **LT**:		Serbian	
Language of abstract, **LA**:		Serbian	
Country of publication, **CP**:		Republic of Serbia	
Locality of publication, **LP**:		Vojvodina	
Publication year, **PY**:		2017	
Publisher, **PB**:		Author's reprint	
Publication place, **PP**:		Novi Sad, Dositeja Obradovica sq. 6	
Physical description, **PD**: (chapters/pages/ref./tables/pictures/graphs/appendixes)		8/37/10/0/20/0/0	
Scientific field, **SF**:		Electrical Engineering	
Scientific discipline, **SD**:		Computer Engineering, Engineering of Computer Based Systems	
Subject/Key words, **S/KW**:		LLVM, code coverage, profile guided optimization	
UC			
Holding data, **HD**:		The Library of Faculty of Technical Sciences, Novi Sad, Serbia	
Note, **N**:			
Abstract, **AB**:		This paper present one improvement related to increasement of required space during creation of profile in LLVM compiler for gathering data during program execution by linking shared *clang.rt_profile* library. Paper also present one solution for storing data about code execution in given momment. Testing of the solution has achieved reduction of the test code size used.	
Accepted by the Scientific Board on, **ASB**:			
Defended on, **DE**:			
Defended Board, **DB**:	President:	PhD Goran Stojanovic	
	Member:	PhD Milan Bjelica	Menthor's sign
	Member, Mentor:	PhD Nikola Teslic	

Zahvalnost

Posebno se zahvaljujem Branislavu Kordiću na pruženoj stručnoj i nesebičnoj podršci tokom studiranja.

Takođe se zahvaljujem porodici, prijateljima, devojci i kolegama na pruženoj moralnoj podršci.

SADRŽAJ

SPISAK SLIKA

SKRAĆENICE

API – *Application program interface,* programski interfejs

AST – *Abstract syntax tree,* apstraktno sintaksno stablo

GCC – *GNU Compiler collection,* skup tehnologija za prevođenje GNU

ICC – *Intel Compiler Collection*, skup tehnologija za prevođenje Intel

IR – *Intermediate Representation*, međuprezentacija koda

LLVM – *Low Level Virtual Machine*, skup tehnologija za projektovanje programskih prevodilaca

PGO – *Profile Guided Optimization*, optimizacija vođena profilisanjem

1. Uvod

Alati za analizu pokrivenosti koda danas čine sastavni deo alata za analizu kvaliteta programskog koda. Osnovna namena je da se odredi procenat programskog koda koji je pokriven ispitnim slučajevima (eng. *tests*) i prepoznaju delovi koda koji se često izvršavaju i koje je potrebno optimizovati. Takođe, čuvanjem i upoređivanjem podataka o pokrivenosti različitih verzija istog koda može se ustavnoviti nivo kvaliteta ispitnih slučajeva.

Prvi put značaj pokrivenosti koda se pominje u časopisu Communications of the ACM 1963. godine. Te godine istraživači Joan C. Miller i Clifford J. Maloney su u vidu naučnog rada objavili prvo istraživanje na tu temu. Danas alati i podaci o pokrivenosti koda predstavljaju sastavni deo razvoja svakog programa koda visokog kvaliteta.

LLVM koji je nastao mnogo kasnije od programskog prevodioca GCC (1987. godine) i poseduje delove koji nisu dovoljno razvijeni i koji su trenutno u razvoju. Ovaj rad će upravo prikazati jedno unapređenje vezano za smanjenje skladišnog prostora potrebnog za pokrivenost koda programskog prevodioca LLVM.

U okviru ovog rada biće predstavljen način rada, namena i jedno proširenje alata za analizu pokrivenosti koda u programskom prevodiocu LLVM. Razlog za odabir ovog programskog prevodioca je sve veća zastupljenost u poređenju sa ostalim programskim prevodiocima kao što su ICC, GCC i drugi. Kroz ovaj rad će takođe biti predstavljena struktura i način rada programskog prevodioca LLVM kao i poređenje sa drugim prevodiocima, pre svega sa ICC i GCC.

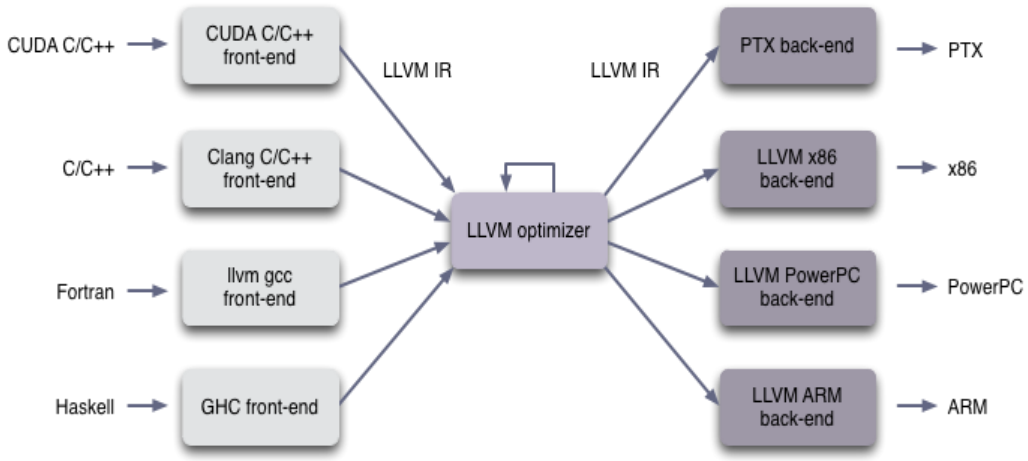
2. Projekat LLVM

LLVM čini lanac alata za prevođenje programskog koda viših nivoa za željenu arhitekturu. Program je otvorenog koda pod licencom „University of Illinois/NCSA Open Source License“. Nastao je 2000. godine na University of Illinois at Urbana–Champaign od strane Chris Lattner i Vikram Adve koji su radili na inovativnoj strukturi za optimizaciju programskih prevodilaca. LLVM doživljava naglu prekretnicu u svom razvoju 2005. godine kada kompanija Apple poziva Chris Lattnera da se pridruži njihovom timu koji će raditi na skupu alata LLVM za potrebe te kompanije. Od tada LLVM postaje sastavni deo alata u kompaniji Apple. Nedugo zatim i ostale velike kompanije kao sto su Sony (PlayStation), Google, Cisco i ostale kreću istim putem.

LLVM je napisan u programskom jezuku C++. Dizajn LLVM-a se bazira na ideji da svi alati budu biblioteke, ostavljajući veoma mali procenat koda koji čini deo nekog alata i koji se ne može koristiti mimo njega. Ovakav dizajn daje korisniku mogućnost da sam definiše korake prevođenja programskog koda i pri tom ima uvid u izlaz svakog od njih. Takođe, ovakav dizajn omogućava dodavanje i razvijanje novih alata.

U ranijim verzijama, projekat LLVM je bio zasnovan samo na zadnjem (eng. *backend*) delu, i koristio je prilagođeni GCC prednji deo za prevodjenje programskog koda viših programskih jezika (C/ C++) u LLVM IR (eng. *IR – intermediate representation*). Zadnji deo LLVM-a prevodi međukod LLVM-a u mašinske instrukcije za željenu arhitekturu. Povezivač LLVM (eng. *linker*) je još uvek u fazi razvoja i trenutno se koristi povezivač GNU za finalno spajanje programskog koda. Dakle, sam dizajn ovog prevodioca ostavlja slobodu kombinovanja više alata, čak i one koji nisu njegov sastavni deo, kako bi se omogućilo kombinovanje njegovih komponenti za specifične potrebe prilikom prevođenja. LLVM ima podršku za prevođenje programskih jezika kao što su: C,C++, C#, Fortran, Python, Objective C. Svaki od nevedenih programskih jezika koristi zaseban prednji deo ovog prevodioca.

U okviru ovog rada spomenućemo nekoliko najbitnijih projekata u okviru LLVM-a. To su: jezgro LLVM, Clang, Compiler-RT, DragonEgg i ispitni slučajevi LLVM.



Slika 1 Struktura programskog prevodioca LLVM

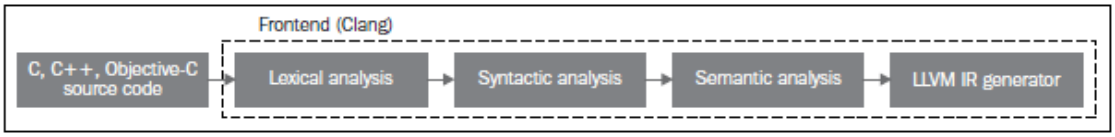
2.1 Clang

Clang je zaseban projekat koji predstavlja prednji deo za prevodilac LLVM za sledeće jezike: C, C++ i Objetive-C. Programski jezik višeg nivoa prevodi na LLVM IR. Napravljen je da koristi zadnji deo LLVM-a i ujedno da bude njegov rukovalac (eng. *driver*), odnosno program koji poziva korisnik i kome se prosleđuju argumenti prevođenja. Uveden je od verzije 2.6. i sa njim predstavlja kompletnu zamenu za GCC. Može se besplatno preuzeti sa intenet stranice prevodioca LLVM. Razvijen je od strane kompanije Apple ali su kasnije i druge kompanije počele da rade na njemu. Clang je dizajniran da čuva više informacija o kodu u toku prevođenja, u poređenju sa prevodiocem GCC, kako bi se lakše pronašla lokacija i bolje opisala priroda greške. Dizajn Clang API (eng. *Application program interface*) veoma liči na GCC i komande njegovog poziva su dosta slične kao i dodatne opcije prilikom prevođenja.

Prednosti Clanga u odnosu na ostale konkurentne prevodioce kao što je GCC se ogleda u smanjenoj potrošnji radne memorije prilikom prevođenja, brže prevođenje izvornog koda i detaljniji opis greške ili upozorenja ukoliko se jave prilikom prevođenja.

2.1.1 Koraci u radu Clang-a

Na Slici 2. su prikazani koraci u radu Clang-a, prednjeg dela prevodioca LLVM. Programski kod višeg nivoa koji se prevodi kada se prosledi Clang-u prolazi kroz nekoliko alata odnosno međukoraka dok se ne prevede u LLVM IR.



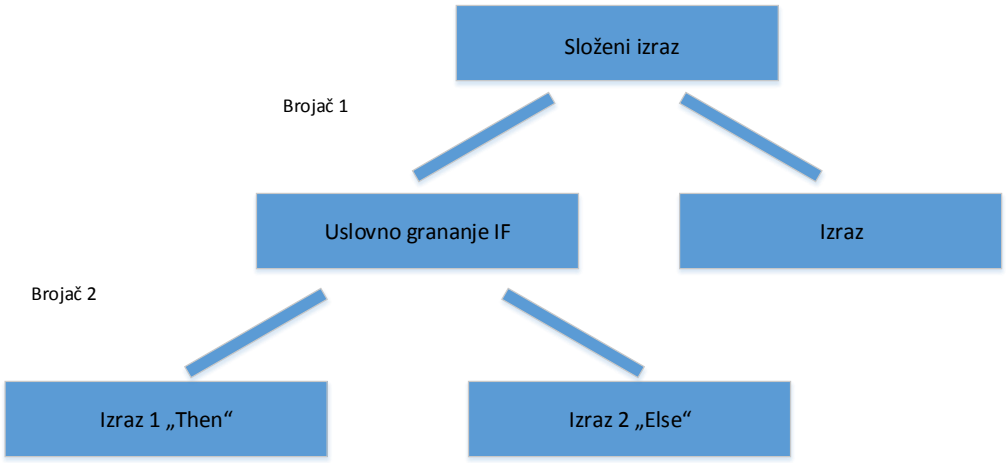
Slika 2 Koraci u radu Clang-a

Kao što je već spomenuto LLVM je organizovan skup biblioteka pa će tako svaki korak prilikom prevođenja predstavljati poziv jedne ili više biblioteka. Za pozivanje Clang-a koristi se sprega biblioteke *libclang*. Prvi korak rada Clang-a predstavlja leksičku analizu koju obavlja biblioteka *libclangLex*. To podrazumeva uključivanje sadržaja datoteka zaglavlja, proširivanje makroa, razrešavanje uslovnog prevođenja i slično.

Sledeći korak ka dobijanju međukoda predstavlja lekser. Lekser obavlja leksičku analizu koda i od njega pravi niz tokena. Svaki token predstavlja karakter ili reč specifičnu za programski jezik koji se prevodi (listu ovih simbola odnosno reči je moguće naći na putanji *llvm/include/clang/Basic/TokenKinds.def*). Upotrebom opcije *-dump-tokens* kao izlaz biće prikazani tokeni za kod koji se prevodi.

Naredni korak u prevođenju je sintaksna analiza uz pomoć biblioteke *libclangParse*. Ova biblioteka parsira kod i grupiše tokene dobijene leksičkom analizom u smislene celine specifične za sitaksu programskog jezika koji se prevodi. Parser ne proverava u potpunosti tačnost ulaznog koda, na primer ako posmatramo primer rečenice na srpskom jeziku, parser neće voditi računa šta znači napisana rečenica ali će proveriti da li je ta rečenica napisana u ispravnom formatu.

Nakon sintaksne analize, semantička analiza vrši određene provere nad predhodno parsiranim kodom. Izlaz iz parsera je predstavljen u obliku AST „*apstraktno sintaksno stablo*“ što je ujedno i ulazni parametar za semantičku proveru. Kod predstavljen u obliku AST predstavlja lakši oblik prezentacije istog za dalje korake u predvođenju. Čvorovi AST-a se predstavljaju sa: deklaracijama, naredbama i tipovima. tj. klasama *Decl*, *Stmt* i *Type*. AST se može ispisati nakon formiranja opcijom *-ast-dump*. Prilikom uključivanja opcija za pokrivenost koda u AST-u se dodaju brojači. Primer je prikazan na Slici 4. Za svaki osnovni blok programskog koda je potrebno imati podatak o izvršenosti, u tom slučaju potrebno je dodati brojače u neke od njih. Kao što je prikazano na Slici 4. vrednost brojača za pojedine osnovne blokove je moguće izračunati na osnovu ostalih brojača. Na ovaj način se štedi memorija i smanjuje broj brojača koji usporavaju instrumentalizovani kod i povećavaju njegovu veličinu. Na Slici 3. se lako može uočiti da se vrednost osnovnog bloka „Izraz 2 Else“ može dobiti oduzimanjem vrednosti brojača jedan i dva.

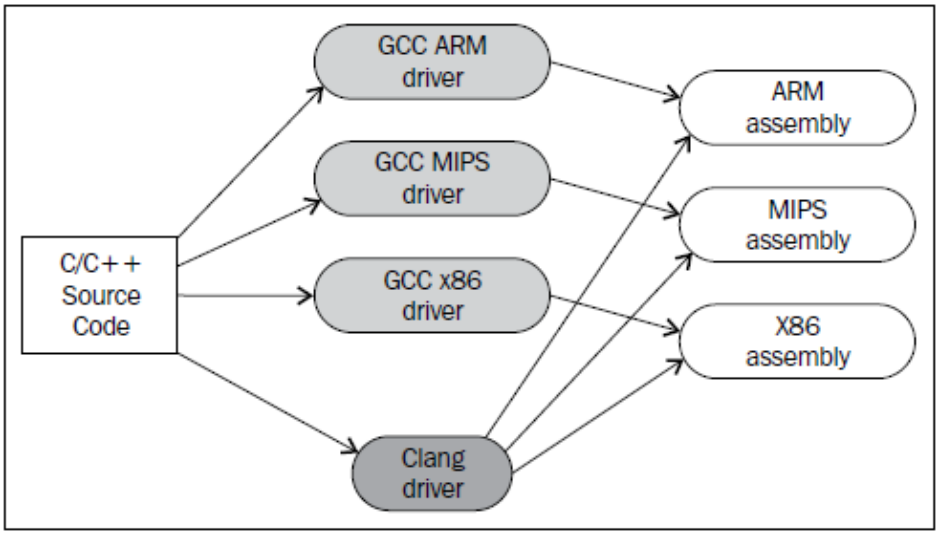


Slika 3 Dodavanje brojača u AST stablo

Na kraju, kada je stablo obrađeno, Clang će kao izlaz AST pretvoriti u LLVM IR. Ovako prezentovan kod je pogodan za generisanje mašinskog koda za željenu ciljnu arhitekturu.

2.2 Jezgro LLVM

Samo jezgro LLVM-a od samog nastanka predstavljalo je projekat koji je zasnovan na istraživanju optimizacija niskog nivoa virtualnih mašina. Pod jezgrom LLVM se podrazumeva zadnji deo ovog programskog prevodioca koji od LLVM IR generiše kod za željenu arhitekturu. LLVM je u osnovi osmišljen da ima podršku za prevođenje koda za više različitih arhitektura (eng. *cross-compiler*). Na primer, specificiranjem opcije *arch=mips64* može se prevesti izvorni kod za MIPS64 arhitekturu. Ovo čini njegovu prednost u odnosu na ostale prevodioce kao što je GCC. Prevodilac GCC takođe može prevoditi kod na više različitih arhitektura ali ga je potrebno prevesti za svaku arhitekturu posebno, dok rukovaoc LLVM omogućava da se to uradi samo specificiranjem opcije komandne linije (Slika 4).



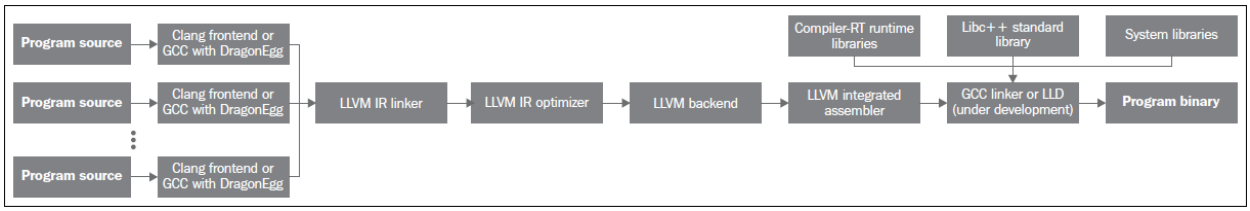
Slika 4 Razlika u prevođenju za ciljne arhitekture između GCC-a i LLVM-a

Jezgro LLVM odnosno njegov zadnji deo predstavlja nezamenljivi korak u prevođenju ukoliko kažemo da je kod preveden sa prevodiocem LLVM. Za prevođenje programskog koda viših programskih jezika kao što su C ili C++ na LLVM IR može se koristi Clang, DragonEgg baziran na prevodiocu GCC, ili čak napisati novi prednji deo. Za krajnje povezivanje prevedenog koda može se koristiti povezivač GNU-GCC umesto povezivača LLVM *llvm-ld* koji je trenutno u fazi razvoja, dok jezgro LLVM-a ostaje nezamenjivo kada je u pitanju prevođenje LLVM IR-a na ciljnu arhitekturu, on čini osnovu ovog programskog prevodioca.

LLVM IR može biti zapisan u binarnom i ljudski-čitljivom (*eng. human readable*) formatu što je veoma korisno prilikom odklanjanja grešaka.

2.3 Kompatibilnost sa alatima GCC

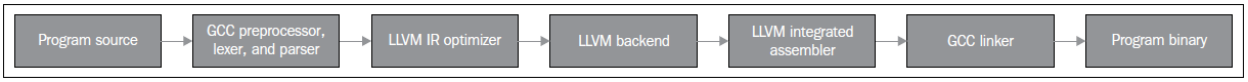
U početnoj fazi projekta LLVM sadašnji prednji deo Clang nije postojao i korišćena je prilagođena verzija GCC, odnosno DragonEGG za generisanje LLVM IR. DragonEgg se i dalje razvija kao zaseban projekat i moguće ga je koristiti i preuzeti sa zvanične internet stranice LLVM. LLVM još uvek koristi povezivač GNU jer je LLVM *llvm-ld* još uvek u razvojnoj fazi, moguće je koristiti ga ali nije toliko uspešan kao povezivač GNU-a prilikom spajanja datoteka sa velikim brojem simbola.



Slika 5 Prikaz toka prevođenja programskog koda sa različitim prednjim delovima

Kao što je prikazano na Slici 5. struktura LLVM ostavlja mogućnost kombinovanja različitih alata kao i uključivanje dodatnih. Sa Slike 5. se možeme zaključiti da je moguće koristiti DragonEgg baziran na prevodiocu GCC za pravljenje LLVM IR, Clang ili neki drugi koji kao izlaz daje LLVM IR. U fazi povezivanja se takođe može uočiti da je ostavljena mogućnost izbora alata, npr. ld (povezivač GNU) ili lld (povezivač LLVM) ili se može napisati novi.

Ukoliko se koristi prilagođeni prednji deo GCC odnosno DragonEgg kao i povezivač GNU, tok prevođenja bi izgledao kao na Slici 6.



Slika 6 Koraci prevođenja sa upotrebom povezivača GNU i prilagođenim prednjim delom GCC

2.4 Skup ispitnih slučajeva LLVM

Ispitni slučajevi LLVM proveravaju ispravnost samog alata ali takođe i performansi, odnosno potrošnju memorije i vreme potrebno za prevođenje koda koji se ispituje. Nakon svake izmene ili proširenja ovog prevodioca potrebno je zadovoljiti kriterijume ispitnih slučajeva i po potrebi dodati nove. Ukoliko nove izmene znatno uspore izvršavanje prevođenja ili povećaju potrošnju radne memorije može se očekivati da neki od ispitnih slučajeva budu neuspešno izvršeni. Ispitni slučajevi se takođe mogu preuzeti zasebno i predstavljaju jedan projekat sa sopstvenom centralnom arhivom (eng. *repository*). Za njihovo korišćenje je potrebno preuzeti izvorni kod na lokaciju *llvm/projects/test-suite* i prevesti ceo LLVM kako bi oni bili uključeni. Ispitivanje se može pokrenuti komandom "*make check-all"*.

Za dodavanje novog ispitnog slučaja potrebno je dodati novu *Makefile* datoteku odnosno recept koji će napraviti novi slučaj ispitivanja. Nova *Makefile* datoteka treba da bude smeštena na putanji "*llvm/projects/test-suite/TEST.<ime_testa>.Makefile"*. U okviru skupa svih ispitnih slučajeva nalazi se i jedan primer na datoj putanji "*llvm/projects/test-suite/TEST.example.Makefile"* koji predstavlja dobar primer kako treba dodati novi ispitni slučaj.

2.5 Projekat compiler-rt

Svakoj ciljnjoj arhitekturi su potrebne dodatne funkcije koje bi simulirale operacije niskog nivoa koje nisu podržane na ciljnoj arhitekturi. Pod ovim podrazumevamo rukovanje 64-bitnim registrima i vrednostima na 32-bitnoj arhitekturi. U tom slučaju potrebno je pozvati posebnu funkciju koja će obaviti (simulirati) aritmetičke operacije nad 64-bitnim vrednostima na 32-bitnoj arhitekturi. Prednji deo programskog prevodioca kada uoči takvu operaciju dodaje poziv posebne funkcije za simulaciju i povezivač će očekivati njenu definiciju prilikom povezivanja. Još jedan primer korišćenja ove biblioteke je upravo i tema ovog rada. Prilikom instrumentalizacije koda, odnosno dodavanja podrške za pokrivenost koda, prednji deo prevodioca će dodati poziv određenih funkcija iz *compiler_rt-a* odnosno *clang.rt_profile* biblioteke prilikom prevođenja i povezivač će očekivati informaciju gde se ti simboli nalaze.

Ukoliko je potrebno koristiti neke od ovih ili drugih funkcionalnosti ove biblioteke, potrebno je preuzeti kod ove biblioteke i dodati ga prilikom prevođenja samog LLVM prevodioca, u suprotnom neće biti moguće uspešno prevođenje instrumentalizovanog koda. Ova biblioteka nije obavezna i predstavlja dodatne opcije prevodioca. Takođe *compiler_rt* (eng. *compiler runtime*) predstavlja odvojen projekat unutar prevodioca LLVM i ima svoju zasebanu centralnu arhivu. Za korišćenje funkcionalnosti *compiler_rt* projekta potrebno je dodati izvorni kod u LLVM radni prostor na lokaciju *llvm/projects/compiler_rt/* a zatim ponovo prevesti ceo LLVM. Najznačajni deo za izradu ovog rada je upravo biblioteka *clang_rt.profile* koja se nalazi na putanji

llvm/projects/compiler-rt/lib/profile/ i sastavni je deo projekta *compiler_rt*. Ova biblioteka pruža podršku u toku izvršavanja instrumentalizovanog koda za prikupljanje podataka o izvršenosti kao i o čuvanju istih nakon završetka rada programa. Kada je instrumentalizacija koda uključena, prednji deo prevodioca će dodati pozive funkcija koji su definisani u ovoj biblioteci, i bez nje je nemoguće pokrenuti instrumentalizovani program.

3. Pokrivenost koda u prevodiocu LLVM

Alati za pokrivenosti koda daju informacije o stepenu izvršenosti koda u nekom od sledecih fomata: XML, TEXT, JSON ili HTML. Ti izveštaji se koriste za unapređivanje kvaliteta koda informacijama koji deo koda se izvršava ispitnim slučajevima (eng. *unity testing*), koji deo koda nije izvršen (nije pokrivnen ispitnim slučajevima), koji deo koda se najučestalije izvršava te ga je potrebno optimizovati.

LLVM poseduje dva pristupa instrumentalizacije programskog koda. U početku nastanka korišćen je format instrumentalizacije GCC do implementacije sopstvenog načina. Format GCC koji je primenjen na LLVM je kompatibilan sa verzijom GCC 4.2 kao i sa još nekim novijim verzijama GCC-a.

3.1 Format instrumentalizacije GCC i pokrivenost koda

Za korišćenje formata GCC za instrumentalizaciju programskog koda potrebno je proslediti odnosno uključiti opcije *-ftest-coverage -fprofile-arcs*. Opcijom *-fprofile-arcs* aktivira se optimizacija vođena profilisanjem koda (eng. *Profile Guided Optimization - PGO*) odnosno biće dodate dodatne instrukcije kao i strukture podataka gde se podaci o pokrivenosti koda smeštaju. Druga opcija *–ftest-coverage* za svaku datoteku C u fazi prevođenja pravi datoteku .gcda sa informacijama koje će alat *llvm-cov gcov* koristiti za spregu informacije o izvršavanju instrumentalizovanog koda dobijenih od PGO i samog izvornog koda kako bi se dodelila prava vrednost brojača svakom delu koda koji je instrumentalizovan.

Kada se izvršavanje instrumentalizovanog koda završi pozivom funkcije *atexit()* napravi se .gcno sa informacijama o izvršavanju instrumentalizovanog koda. Nakon toga je moguće napraviti tekstualni izveštaj pomoću datoteka .gcno i .gcda.

3.2 Format instrumentalizacije LLVM i pokrivenost koda

LLVM poseduje svoj format instrumentalizacije i baziran je na PGO. Uz ovu opciju koristi se još jedna opcija koja dodaje podatke kako bi se prilikom pravljenja izveštaja odredilo kom delu izvornog koda pripada koja vrednost brojača dobijena izvršavanjem instrumentalizovanog koda. LLVM koristi drugačiji pristup smeštanja podataka o mapiranju regiona izvornog koda i brojača koji su dodati u kod uključivanjem opcije PGO. Za razliku od GCC-a, LLVM ne pravi dodatne datoteke (.gcda za GCC), već sve informacije potrebne za generisanje izveštaja čuva u sklopu objektne datoteke. Dakle, za generisanje izveštaja o pokrivenosti koda dovoljno je prevesti željenu datoteku koju je potrebno instrumentalizovati sa dve dodatne opcije *-fprofile-instr-generate* i *-fcoverage-mapping* i pokrenuti prevedeni kod. Nakon završetka izvršavanja koda pravi se datoteka *default.profraw* koja u sebi sadrži podatke o izvršenosti koda, odnosno vrednosti svih brojača. Alat *llvm-cov* koji kao parametar prima datoteku C, datoteku dobijenu izvršavanjem koda i objektnu datoteku a zatim generiše izveštaj u ljudski čitljivom obliku. Dakle alat za generisanje izveštaja će iz objektne datoteke upotrebiti informacije o mapiranju brojača i uz pomoć datoteke o izvšenosti koda svakom delu koda dodeliti odgovarajući brojač odnosno vrednost koliko se puta taj deo koda izvršio.

3.3 Primer prevođenja upotrebom formata LLVM

Da bi se uključila podrška pokrivenosti koda u programskom prevodiocu LLVM potrebno je za datu datoteku koji se prevodi proslediti dve opcije Clang-u – programskom rukovaocu za LLVM: *-fprofile-instr-generate* i *-fcoverage-mapping*.

Prva opcija, odnosno *-fprofile-instr-generate* uključuje PGO i dodaje dodatni kod koji će prikupljati podatke o izvršenosti u postojeći koje će kasnije služiti za optimizaciju. Umetanje dodatnog koda uključivanjem ove opcije nije tema ovog rada, te ćemo to uzeti kao nivo abstrakcije.

Druga opcija tokom prevođenja *-fcoverage-mapping* će umetnuti informacije o mapiranju regiona koji zapravo opisuju koji brojač odgovara kom delu instrumentalizovanog koda. Zapravo ova opcija se oslanja na opciju *-fprofile-instr-generate* i dodaje informacije koje su potrebne za da bi se napravio izveštaj o pokrivenosti koda. Ove informacije mapiraju svaki deo koda odgovarajućem brojaču i one su napravljene da budu nezavisne odnosno one će biti umetnute u kod LLVM IR-a koji je nezavisan od ciljne arhitekture a kasnije će zatim biti prevedene i u objektni fajl. Ove informacije će služiti alatu za pravljenje izveštaja da podatke dobijene izvršavanjem datoteke prevedene sa PGO obradi i pripoji tačno određenom delu koda. Ovaj format mapiranja podataka je napravljen da bude univerzalan i pogodan za prednji deo bilo kog programskog prevodioca, a ne samo LLVM. Za prevođenje željene datoteke C sa uključenom opcijom za pokrivenost koda potrebno je koristiti sledeće opcije:

clang main.c -fprofile-instr-generate -fcoverage-mapping -o main

Programskom rukovaocu LLVM kao opcije komandne linije prosleđuju se ime ulazne datoteke C, ime izlazne izvršne odnosno objektne datoteke i opcije za uključivanje PGO kao i opcije za dodavanje informacija o mapiranju brojača sa odgovarajućim delom koda.

Ovako prevedena izlazna datoteka u sebi sadrži sve što je neophodno za prikupljanje podataka o izvršenosti koda. Da bi se prikupili podaci potrebno je pokrenuti prevedenu izvršnu datoteku:

./main

Nakon što se dati program izvrši u istom direktorijumu napraviće se datoteka *default.profraw* koja sadrži podatke o brojačima kao i regionima koje oni predstavljaju. Nakon toga se poziva alat *llvm-profdata* koji spaja više datoteka o pokrivenosti koda koji su dobijeni izvršavanjem različitih ispitnih slučajeva od strane PGO instrumentalizacije:

llvm-profdata merge -o main.profdata default.profraw

Nakon toga kao izlaz se pravi datoteka koja je pogodna za ulaz u alat *llvm-cov* koji kao svoj izlaz generiše krajnji izveštaj o pokrivenosti koda:

llvm-cov show main -instr-profile=main.profdata main.c

Izlaz iz ovog alata je zbirni tekstualni izvestaj o izvršavanju koda dobijenog izvršavanjem željenih ispitnih slučajeva. Ukoliko je potrebno spojiti informacije o pokrivenosti koda dobijene izvršavanjem različitih ispitnih slučajeva potrebno je koristiti alat *llvm-profdata merge*.

3.4 Mapiranja brojača

Mapiranje brojača funkcioniše na nivou funkcije, odnosno koristi svojstva PGO. Prednji deo prevodioca za svaku funkciju koja je instrumentalizovana dodaje podatke koji mapiraju prevođeni programski kod sa brojačima dodatim prilikom prevođenja sa PGO. Ovi podaci su predstavljeni u obliku niza mapiranih regiona. Jedan mapirani region sadrži podatke o njegovom početku i kraju odnosno broju kolone i reda za njegov početak i kraj. Postoje nekoliko različtih tipova regiona koji se mapiraju.

3.4.1 Regioni koji odgovaraju delovima izvornog koda i njihovim brojačima

Ovi regioni predstavljaju većinu i koristi ih alat za generisanje izveštaja o pokivenosti koda prilikom računanja izvršenosti linija, zatim regiona odnosno linija koje nisu izvšene kao i za računanje statistika za svaku funkciju, na primer koliko linija jedne funkcije se izvršilo u procentima. Slika 7. predstavlja ove regione.

```
int main(int argc, const char *argv[]) {          // Regon koda od 1:40 do 9:2
    if (argc > 1) {                               // Regon koda od 3:17 do 5:4
        printf("%s\n", argv[1]);
    } else {                                      // Regon koda od 5:10 do 7:4
        printf("\n");
    }
    return 0;
}
```

Slika 7 Prikaz regiona koji odgovara jednom delu izvonog koda

Kao što možemo videti na primeru za funkciju sa Slike 7 , regioni su podeljeni na osnovne blokove koda. Svaki blok izdvojen vitičastim zagradama je opisan njegovim početkom i krajem gde početak i kraj predstavlja podatak sačinjen od rednog broja kolone i reda. Prvi region, na primer, predstavlja funkciju *main* i on počinje u prvom redu i četrdesetoj koloni a završava se u devetom redu i drugoj koloni. Unutar ovog regiona odnosno funkcije postoje još dva osnovna bloka koda izdvojena vitičastim zagradama koje takođe predstavljaju regione koje je potrebno mapirati. U ovom slučaju to su uslovna grananja koda koja počinju u trećem odnosno petom redu i sedamnaestoj odnosno desetoj koloni a završavaju u petom osnosno sedmom redu i četvrtoj koloni.

3.4.2 Preskočeni regioni

Preskočeni regioni se koriste za predstavljanje delova koda odnosno regiona koji će biti preskočeni od strane pretprocesiranja Clang. Njima neće biti dodeljivani brojači zato što prednji deo prevodioca zna da oni nikada neće biti izvršeni. Oni se koriste isključivo od strane alata za generisanje izveštaja kako bi taj alat imao informaciju da taj deo koda odnosno region nema izvršenih linija, primer je dat na Slici 8.

```
int main() {                          // Region koda od 1:12 do 6:2
#ifdef DEBUG                          // Preskočeni region koda od 2:1 do 4:2
  printf("Hello world");
#endif
  return 0;
}
```

Slika 8 Primer preskočenih regiona

3.4.3 Regioni proširenja

Regioni proširenja se koriste za predstavljanje C/C++ makroa. Ovi regioni imaju jedan dodatni parametar, jedinstveni identifikacioni broj datoteke (eng. *expanded file id*). Ovaj parametar se koristi od strane alata za generisanje izveštaja kako bi se pronašli mapirani regioni koji su kreirani kao rezultat izvršavanja datog makroa, upoređivanjem njihovog jedinstvenog broja sa

jedinstvenim brojem datoteke koja se trenutno obrađuje prilikom kreiranja izveštaja o pokrivenosti koda. Ovi regioni ne poseduju sopstveni brojač već alat za pravljenje izveštaja uzima broj izvršenosti ovog regiona uzimanjem vrednosti prvog brojača za datu datoteku odnosno njen jedinstveni broj. Primer ovih regiona je dat na Slici 9.

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

Slika 9 Primer regiona ekspanzije odnosno proširenja

3.4.4 Mapiranje brojača PGO

Mapirani brojač predstavlja referencu na postojeći brojač koji je dodat prilikom prevođenja programskog koda sa opcijom PGO. Uključivanjem opcije mapiranja tokom prevođenja prednji deo prevodioca će sačuvati informacije o brojačima za neke od regiona. Broj izvršavanja za region sa takvim brojačem se određuje uzimanjem vrednosti odgovarajućeg brojača dodatim od strane PGO. Neki od regiona ne poseduju sopstveni brojač već se njihova vrednost dobija korišćenjem vrednosti ostalih brojača kako bi se uštedela memorija, dok nekim brojačima prednji deo prevodioca dodeljuje konstantnu vrednost. U prvom slučaju kao sto je predstavljeno na Slici 10. vrednost brojača regiona označenog žutom bojom se računa korišćenjem vrednosti brojača regiona nula i jedan. Na ovaj način se štedi radna memorija i smanjuje veličina samog binarnog izlaza programskog prevodioca.

```
int main(int argc, const char *argv[]) {   // Mapirani brojač prvog regiona je referenca na
brojač dobijen profajliranjem  #0

  if (argc > 1) {                  // Mapirani brojač prvog regiona je referenca na brojač dobijen
profajliranjem  #1
    printf("%s\n", argv[1]);
  } else {                         // Mapirani brojač ovog regiona je izraz (referenca na PGO
brojač #0 – referenca na PGO brojač #1)
    printf(„\n“);
  }
  return 0;
}
```

Slika 10 Primer dodeljivanja brojača regionima

Takođe za brojače nekih regiona prednji deo prevodioca je pre izvršavanja prilikom prevođenja dodelio vrednost, oni predstavljaju delove koda odnosno regione za koje je prednji deo prevodioca predprocesiranjem ustanovio da se nikada neće izvršiti te im je dodelio vrednost nula. Na Slici 11. je predstavljen primer ovih regiona.

```
int main() {
  return 0;
  printf("Hello world!\n");   // Region koda koji se nikada neće izvršiti, njegova vrednost je 0
}
```

Slika 11 Primer regiona koji se nikada neće izvršiti

Mapirani brojači kojima je dodeljena vrednost nula prilikom prevođenja služe kako bi alat za generisanje izveštaja ispravno prikazao vrednosti izvršenosti svih neizvršenih linija za datu datoteku. Bez njih, alat za generisanje izveštaja bi predpostavio da su te linije i regioni izvršeni jer ne poseduje „znanje“ prednjeg dela prevodioca.

3.5 Interpretacija u LLVM IR

Podaci o mapiranju regiona o pokrivenosti koda dodaju se u LLVM IR, predstavljeni u jednu konstantnu globalnu promenljivu odnosno strukturu podataka *__llvm_coverage_mapping* u *__llvm_covmap* sekciji podataka. Za primer sa Slike 12. prikazan je izgled LLVM IR na Slici 13.

```
int foo() {
  return 42;
}
int bar() {
  return 13;
}
```

Slika 12 Primer koda sa dve funkcije za koji je predstavljen LLVM IR

```
@__llvm_coverage_mapping = internal constant { { i32, i32, i32, i32 }, [2 x { i8*, i32, i32 }],
[40 x i8] } {
  { i32, i32, i32, i32 } ; Zaglavlje mapiranih podataka
  {
    i32 2,  ; Broj instrumentalizovanih funkcija
    i32 20, ; Dužina reči koja sadrži kodovanu prevodilačku jedinicu imena datoteka
    i32 20, ; Dužina reči koja sadrži kodovane mapirane podatke
    i32 0,  ; Verzija formata mapiranja podataka
  },
  [2 x { i8*, i32, i32 }] [ ; Podaci za instrumentalizovane funkcije
    { i8*, i32, i32 } { i8* getelementptr inbounds ([3 x i8]* @__llvm_profile_name_foo, i32 0,
i32 0), ; Ime prve funkcije
      i32 3, ; Dužina imena prve funkcije
      i32 9  ; Dužina reči kodovanih mapiranih podataka za datu funkciju
    },
    { i8*, i32, i32 } { i8* getelementptr inbounds ([3 x i8]* @__llvm_profile_name_bar, i32 0,
i32 0), ; Ime druge funkcije
      i32 3, ; Dužina imena druge funkcije
      i32 9  ; Dužina reči kodovanih mapiranih podataka za datu funkciju
    }],
  [40 x i8] c"..." ; Kodovani podaci o pokrivenosti
}, section "__llvm_covmap", align 8
```

Slika 13 Prikaz LLVM IR za primer koda C sa Slike 12

Kao što se može videti sa Slike 13. promenljiva za mapiranje podataka o izvršenosti, koda koji je instrumentalizovan u primeru na Slici 12. je dobijena od strane prednjeg dela prevodioca i sadrži tri osnovna polja:

- Zaglavlje mapiranih podataka
- Lista koja predstavlja zapis instrumentalizovanih datoteka
- Mapirane podatke za instrumentalizovan kod koji su predstavljeni u vidu niza bajtova. (Nule na kraju su dodate kako bi se forsiralo osmobitno poravnanje ovih podataka)

Kao što je prikazano na Slici 13. zaglavlje mapiranih podataka poseduje sledeća polja: broj instrumentalizovanih funkcija, dužinu reči trećeg polja strukture podataka *__llvm_coverage_mapping* koja sadrži kodovanu prevodilačku jedinicu imena funkcije, dužinu reči trećeg polja strukture podataka *__llvm_coverage_mapping* koja sadrži kodovane mapirane podatke i na kraju verziju formata mapiranja podataka koja je u ovom slučaju prva, odnosno verzija je nulta.

Zapis jedne funkcije je struktura koja sadrži tri parametra *{i8*, i32, i32},* ime funkcije, dužinu imena funkcije i dužinu kodiranih mapiranih podataka za tu funkciju.

Kodirani podaci se čuvaju u jednoj reči (eng. *string*) koja sadrži kodirano ime fajla i koja se koristi za prevodilačku jedinicu i kodovane mapirane podatke za svaku funkciju u toj

prevodilačkoj jedinici. Kodovani podatak ima sledeću strukturu: *[ime datoteke, kodovani mapirani podatak za funkciju 1, kodovani mapirani podatak za funkciju 2, ..., popuna memorije]*. Ukoliko je potrebno kodovanim mapiranim podacima će biti dodate nule na kraju kako bi se ispunio prostor do prve veličine koja je deljiva sa osam.

4. Koncept rešenja

U okviru ovog rada biće predstavljeno jedno proširenje za biblioteku *clang.rt_profile* koje će smanjiti veličinu instrumentalizovanog koda u slučaju instrumentalizacije velikog broja komponenti sistema koji se ispituje. Pod komponentama sistema ćemo podrazumevati biblioteke i procese.

Biblioteka *clang.rt_profile* je deo projekta *compiler_rt* (eng. *Compiler runtime*). Unutar ovog projekta se razvijaju biblioteke koje služe kao podrška prilikom izvršavanja programskog koda koji je preveden sa dodatnim opcijama LLVM. U ovom radu su korišćene opcije *-fprofile-instr-generate* i *-fcoverage-mapping* koje aktiviranjem prilikom prevođenja koriste funkcionalnosti biblioteke *clang.rt_profile*. Kada su navedene opcije uključene, prednji deo prevodioca Clang prilikom prevođenja dodaje dodatne instrukcije, brojače i strukture podataka u kod koji se instrumentalizuje. Većina implementacija ovih funkcija se upravo nalazi unutar biblioteke *clang.rt_profile* i bez nje nije moguće izvršiti instrumentalizovani kod.

Ova biblioteka je implementirana da se koristi isključivo kao statička, odnosno nije predviđena da se uvezuje dinamički u instrumentalizovnu komponentu nekog sistema. U slučaju instrumentalizacije celog jednog sistem koji ima mnogo procesa i biblioteka značajno bi se povećala veličina slike koju je potrebno pokrenuti na uređaju. U zavisnosti od veličine skladišne memorije uređaja za koju je ispitivani kod preveden, moguće je susresti se sa problemom kao što je veličina slike koja prevazilazi skladišni kapacitet uređaja.

Nakon pokretanja instrumentalizovanog koda i izvršavanja željenih ispitnih sličajeva za koje je potrebno dobiti podatke o izvršenosti, podaci mogu biti sačuvani na disku jedino nakon završetka rada procesa odnosno pozivom funkcije *atexit()*. U ovom slučaju nemoguće je dobiti željene podatke o pokrivenosti koda za procese koji se nikada ne završe, odnosno koji su stalno prisutni i rade u pozadini. Za procese ovog tipa potrebno je implementirati spregu, odnosno

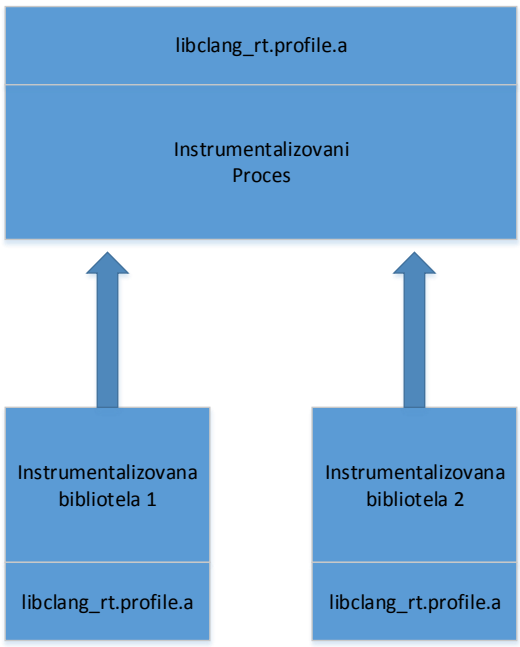
međuprocesnu komunikaciju kako bi se u proizvoljnom trenutku mogli sačuvati podaci o izvršenosti.

Uz opciju za čuvanje podataka o izvršenosti u proizvoljnom trenutku potrebno je i dodati podršku za brisanje vrednosti svih brojača odnosno vraćanje njihovih vrednosti na nulu. Ova opcija je od izrazitog značaja ukoliko je potrebno dobiti podatke o samo jednom ili više ispitnih slučajeva bez uticaja spoljnih faktora na taj deo koda. Na primer, ukoliko je potrebno dobiti odvojene podatke o izvršavanju dva ispitna slučaja potrebno postaviti vrenosti svih brojača na nulu, zatim pokrenuti prvi ispitni slučaj i sačuvati rezultate, ponovo postaviti vrednosti brojača na nulu, pokrenuti sledeći ispitni slučaj i sačuvati rezultate. Spajanje rezultata ova dva ispitna slučaja se jednostavno može učiniti alatom *llvm-prof data merge.*

4.1 Biblioteka *clang.rt_profile* kao statička

Biblioteka *clang_rt.profile* je u osnovi osmišljena da bude povezana u instrumentalizovani proces kao statička i nije je moguće prevesti da radi kao deljena biblioteka bez dodatnih izmena u samoj biblioteci. Za svaku instrumentalizovanu komponentu odnosno proces ili biblioteku potrebno je uvezati po jednu biblioteku koja će za nju prikupljati podatke o pokrivenosti koda. U slučaju instrumentalizacije velikog broja komponenti znatno bi se povećala veličina izvršne datoteke koja je već povećana samom instrumentalizacijom koda. Jedan od načina smanjena veličine izvršnog koda je i prevođenje odnosno povezivanje biblioteke *clang_rt.rofile* kao deljene. U te svrhe potrebno je modifikovati sam izvorni kod te biblioteke.

Na Slici 14. je slikovito prikazan jedan primer uvezivanja statičke biblioteke *clang_rt.profile* gde je instrumentalizovan jedan proces i dve deljene biblioteke.



Slika 14 Prikaz uvezivanja biblioteke libclang_rt.profile.a kao statičke

Prilikom prevođenja koda sa uključenim opcijama za prikupljanje podataka o izvršenosti, prednji deo prevodioca dodaje dodatne pozive funkcija u kod koji se instrumentalizuje. Implementacija tih funkcija se nalaze u biblioteci *clang_rt.profile.* Te funkcije su specifične i ne bi bile funkcionalne ukoliko bi se ova biblioteka prevela i koristila kao deljena jer su predviđene da se koriste samo unutar jednog modula (procesa ili deljene biblioteke). Definisane su atributom *COMPILER_RT_VISIBILITY* što ih čini vidljivim samo u jednom modulu gde su definisane. U slučaju da ovu biblioteku uvežemo kao dinamičku dobili bismo grešku povezivača da simboli biblioteke *clang_rt.profile* nisu vidljivi komponentama instrumentalizovanog koda. Neki od simbola koji su značajni i koji će biti detaljno opisani u opisu rešenja su funkcija *__llvm_profile_register_function(),* promenljiva označenog celobrojnog tipa vrednosti *int __llvm_profile_runtime* kao i osnovna struktura podataka koja se koristi za čuvanje podataka o izvršenosti jedne funkcije *__llvm_profile_data.*

4.2 Čuvanje i brisanje rezultata u proizvoljnom trenutku

Za dobavljanje informacija o pokrivenosti koda u trenutnoj LLVM implementaciji projekta *compiler_rt* odnosno biblioteke *clang_rt.profile* potrebno je da se instrumentalizovani proces završi nakon čega će podaci biti sačuvani na masovnoj memoriji. Ovakva implementacija predstavlja problem za pozadinske procese koji se neprekidno izvršavaju. Oni se nikada ne završavaju i za njih nije moguće dobiti podatke o izvršenosti. Takođe, nemoguće je ponovo postaviti vrednosti brojača na nulu kako bi se prikupili rezultati za drugi ispitni slučaj. Prilikom pokretanja instrumentalizovane komponente koja ima uvezanu ovu biblioteku biće pozvana funkcija *__llvm_profile_register_write_file_atexit* koja će registrovati funkciju koja treba da bude pozvana nakon završetka rada instrumentalizovane komponente *atexit(writeFileWithoutReturn).* Prilikom završetka rada instrumentalizovane komponente data funkcija će napraviti datoteku koja sadrži informacije o pokrivenosti koda.

Za izradu i ispitivanje ovog rada korišćen je operativni sistemi Linux. Za dodavanje opcije čuvanja i postavljanja vrednosti brojača na nulu u proizvoljnom trenutku potrebno je izabrati način međuprocesne komunikacije kako bi što manje uticali na rad sistema i dobili precizne podatke. Pod tim se podrazumeva da operativni sistem može da radi bez narušavanja funkcionalnosti prilikom rukovanja podacima o pokrivenosti.

5. Implementacija

U ovom poglavlju će biti opisano jedno rešenje prevođenja biblioteke *clang_rt.profile* kao dinamičke. Ona je deo projekta *compiler_rt* i samim tim neophodno je promeniti izvorni kod same biblioteke. Potrebno je prilagoditi njen rad kako bi podaci svih instrumentalizovanih komponenti ispitnog sistema bili ispravno sačuvani.

Takođe biće opisano i jedno rešenje čuvanja i uspostavljanja početnih vrednosti brojača u proizvoljnom trenutku.

5.1 Biblioteka *clang_rt.profile* kao deljena

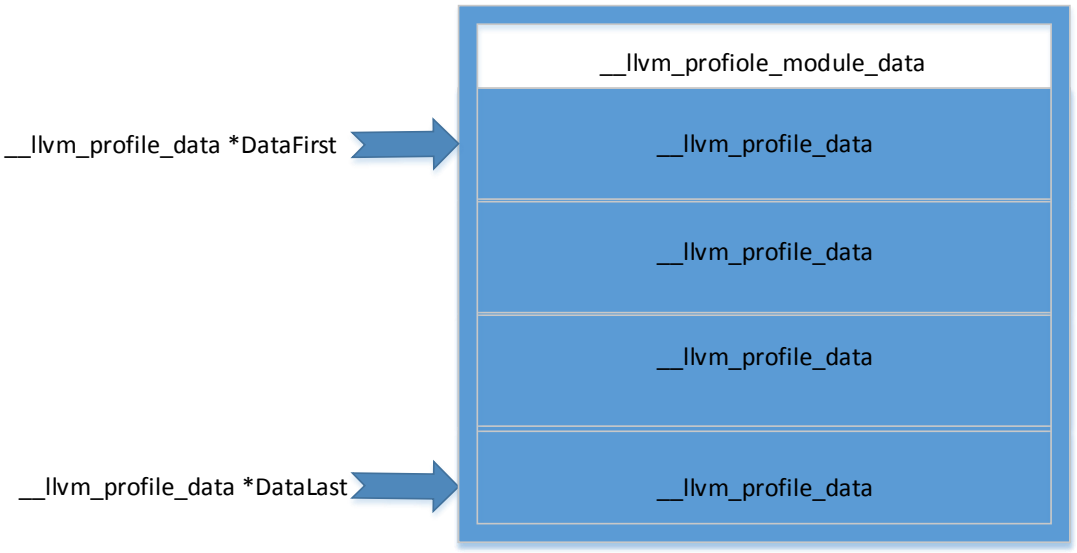
Ukoliko se biblioteka *clang_rt.profile* prevede bez ikakvih izmena kao deljena i pokuša povezati u neku instrumentalizovanu komponentu dobiće se greška povezivača da dva simbola nisu razrešena odnosno da povezivač nije u stanju da nađe njihove definicije: *__llvm_profile_register_function()* i *__llvm_profile_runtime.* Ova funkcija i promenljiva su osmšiljene da rade iskljucivo u slučaju korišćenja ove biblioteke kao statičke, odnosno definisani su sa atributom *COMPILER_RT_VISIBILITY* koji definiše vidljivost ovih simbola samo u okviru jednog modula. Samim tim ostale komponente ne mogu da vide ova dva simbola koja služe za rukovanje podacima o pokrivenosti koda jer su vidljivi samo u okviru same biblioteke *clang_rt.profile*.

Promenljiva *__llvm_profile_runtime* dodata od strane prednjeg dela prevodioca u kod koji se instrumentalizuje, služi kako bi se uključila inicijalizacija jedne komponente instrumentalizovanog sistema u toku njegovog izvršavanja. Proveravanjem vrednosti ove promenljive biće odlučeno da li će data komponenta biti uzeta u obzir prilikom rada instrumentalizovanog sistema. U slučaju da ova promenljiva nije inicijalizovana biće pozvane funkcije koje će obaviti potrebnu inicijalizaciju za datu komponentu kao što je registrovanje

funkcije koja će biti pozvana nakon završetka izvršavanja date komponente - *atexit()* i incijalizacija struktura koje su potrebne za čuvanje informacija o pokrivenosti koda. U slučaju da je u nekoj instrumentalizovanoj komponenti ručno inicijalizovana vrednost ove promenljive na jedan, inicijalizacija će biti preskočena i rezultati o izvršenosti te komponente se neće naći u datoteci koja će biti napravljena nakon rada te komponente.

Pozivi funkcije *__llvm_profile_register_function(__llvm_profile_data *Data)* se dodaju u sve instrumentalizovane funkcije. Njen parametar je pokazivač na strukturu podataka *__llvm_profile_data* koja predstavlja podatke o izvršenosti jedne funkcije. Na ovaj način, prilikom pozivanja instrumentalizovane funkcije biće pozvana i ova funkcija koja će preko pokazivača *__llvm_profile_data* proslediti informaciju gde su tačno sačuvani podaci za tu instrumentalizovanu funkciju. Ova strukura u sebi sadrži podatke o imenu svake funkcije, broju brojača, kao i pokazivače na te brojače.

Najveći deo izmena koje se odnose na dodavanju podrške za rad biblioteke *clang_rt.profile* kao deljene je vezan za samu funkciju *__llvm_profile_register_function* odnosno na način na koji će ova funkcija čuvati podatke o svakoj instrumentalizovanoj funkciji u sebi. Podaci za jednu komponentu su složeni kontinualno u jednom delu memorije od strane prevodioca. Na primer, ukoliko bismo posmatrali te podatke za četri instrumentalizovane funkcije (Slika 15.) iz jedne komponente instrumentalizovanog sistema oni bi bili spojeni jedni do drugih.



Slika 15 Prikaz podataka o pokrivenosti koda za jednu komponentu

Dakle potrebno voditi računa samo o dve vrednosti, odnosno pokazivačima na prvu i poslednju strukturu *__llvm_profile_data* jer su podaci u memoriji kontinualno spojeni i unapred je poznata njihova duzina. Tako se prilikom poziva funkcije *__llvm_profile_register_function (__llvm_profile_data *Data)* proverava da li je vrednost trenutnog pokazivača manja od prvog i potrebno ga je ažurirati ili je veći od poslednjeg i njega je potrebno ažurirati.

Ovakav pristup znatno olaksava zapis podataka o pokrivenosti koda na masovnu memoriju. Pošto je poznata vrednost adrese prve i poslenje strukture kao i dužina strukture zapisa jedne funkcije nakon završetka izvršavanja instrumentalizovane komponente i prilikom poziva funkcije *atexit()* odnosno *__llvm_profile_write_file()* podaci između prve i poslednje adrese će biti preslikani u datoteku na masovnoj memoriji koja će kasnije služiti za generisanje izveštaja o pokrivenosti.

U slučaju dinamičke biblioteke *clang_rt.profile* u samoj implementaciji *__llvm_profile_register_function* funkcije je potrebno obezbediti da se prilikom poziva funkcije za čuvanje podataka svi podaci za sve instrumentalizovane komponente odgovarajuće sačuvaju u datoteku na masovnoj memoriji. Pod tim se podrazumeva da deljena biblioteka ima informacije, odnosno pristup podacima o izvršenosti koda za svaku komponentu instrumentalizovanog sistema.

Pošto su podaci svake pojedinačne instrumentalizovane komponente kontinualno grupisani na različitim delovima memorije potrebno je obezbediti vezu izmedju njih i unaprediti postojeći pristup korišćenjem funkcije *__llvm_profile_register_function(__llvm_profile_data *Data).* Kada se pokrene instrumentalizovani kod, svaka instrumentalizovana funkcija će pozivom funkcije za registraciju i prosleđivanjem pokazivača na strukturu gde se nalaze njeni podaci obezbediti spregu sa bibliotekom *clang_rt.profile* koja je zadužena za rukovanje tim podacima. Deljena biblioteka *clang_rt.profile* će na ovaj način znati gde se u memoriji nalaze podaci za datu funkciju. Prilikom zapisivanja podataka u datoteku ova biblioteka mora da pristupi podacima svih instrumentalizovanih funkcija iz svih modula i zapiše ih u datoteku u odgovarajućem redosledu. Ukoliko se pomešaju podaci alat za generisanje izveštaja alat *llvm-show* će prijaviti grešku da podaci o pokrivenosti nisu ispravni. Zbog toga je od velike važnosti kako će biblioteka *clang_rt.profile* čuvati informacije gde se nalaze podaci svih instrumentalizovanih funkcija, iz svih modula prilikom njihove inicijalizacije, odnosno pozivom funkcije *__llvm_profile_register_function* i prosleđivanjem pokazivača na te podatke *__llvm_profile_data *Data*. Biblioteka *clang_rt.profile* mora da čuva informacije o pokrivenosti u istom redosledu kako su oni zaista zapisani u memoriji kako bi se sačuvali u datoteku u ispravnom formatu.

Pošto je jedini prosleđeni podatak pokazivač na strukturu podataka koja opisuje izvršenost jedne funkcije, iskoristićemo njegovu vrednost za očuvanje pravilnog redosleda svih podataka komponenti u instrumentalizovanom sistemu. Kako bi se svi podaci sačuvali u ispravnom formatu korišćena je jednostruko spregnuta lista Slika 16. i algoritam sortiranja sa umetanjem (eng. *insertion sort*). Na ovaj način podaci će uvek biti sortirani u listi redosledom kojim treba da budu zapisani u datoteku.

Slika 16 Uvezivanje podataka iz različitih modula u jednostruko spregnutu listu

Jedan element ove liste je struktura koja u sebi sadrži pokazivač na jednu strukturu podataka *__llvm_profile_data* i pokazivač na sledeći element u listi. Potrebno je uvek pamtiti pokazivač na prvi element i prolaziti kroz listu dok se ne dobije nevažeća adresa sledećeg elementa odnosno *null* pokazivač. Pošto znamo da su podaci o pokrivenosti jedne komponente kontinualno složeni u memoriji prilikom umetanja novog elementa u listu proveravaćemo vrednost njegove adrese i po principu rada algoritma sortiranja sa umetanjem dodati novi element na odgovarajuću poziciju. Na ovaj način je osigurano da su informacije o adresi svih podaka za svaku instrumentalizovanu komponentu povezani kontinualno baš kao što su sačuvani u memoriji i da će biti zapisani u ispravnom formatu u izlaznu datoteku. Prilikom zapisa ovih podataka u izlaznu datoteku potrebno je samo iterirati kroz listu od prvog do poslednjeg elementa i podaci će biti sačuvani u ispravnom formatu.

5.2 Čuvanje i brisanje rezultata u proizvoljnom trenutku

Za uspostavljanje početnih vrednosti brojača i čuvanje podataka na masovnoj memoriji u proizvoljnom trenutku potrebno je obezbediti međuprocesnu komunikaciju sa instrumentalizovanim procesom i obezbediti spregu koji će omogućiti date funkcionalnosti unutar instrumentalizovanog procesa. Za izradu i ispitivanje ovog rešenja korišćena je platforma *Linux* pa su u obzir uzeta dva vida među procesne komunikacije, signali (eng. *signals*) i utičnica (eng. *unix domain socket*). Prilikom implementacije i ispitivanja prvog načina ustanovljeno je da on nije pogodan u ovom slučaju iz nekoliko razloga. Prvi razlog donosi moguće probleme u radu samog sistema jer je moguće pobuditi neželjenu reakciju u slučaju da je i neki drugi proces obrađuje isti signal. Drugi razlog je dugo vreme obrade signala, zbog snimanja velike količine podataka na masovnu memoriju. Za to vreme funkcija samog uređaja bi bila blokirana što bi uzrokovalo moguće reakcije kao što su ponovno pokretanje uređaja od strane sigurnosnih merača vremena (eng. *watch dog timer*).

Drugi pristup podrazumeva upotrebu utičnica za implementaciju klijent – poslužilac arhitekture. Poslužilac je implementiran u biblioteci koja se uvezuje u svaki instrumentalizovani proces. Klijent je zaseban proces koji se pokreće u željenom trenutku da ostvari komunikaciju sa instrumentalizovanim procesom i pošalje komandu za postavljanje vrednosti brojača na nulu ili komadnu za čuvanje podataka u datoteku na masovnoj memoriji. Poslužilac biblioteka koja je uvezana u instrumentalizovani proces u sebi sadrži funkciju koja se poziva prilikom pokretanja

instrumentalizovane komponente i definisana je atributom *__attribute__((constructor)).* Ona pokreće novu programsku nit (eng. *thread*) koji pravi utičnicu i očekuje poruke klijentske strane. Utičnice prave svi instrumentalizovani procesi na lokaciji */temp/code_coverage/ <ime_procesa_pid>*. Ime utičnice sastoji se od imena procesa i njegovog jedninstvenog identifikacionog broja *PID* ukoliko se neki od procesa pokrene više puta. Pretragom putanje */temp/code_coverage/* klijentski proces nalazi sve instrumentalizovane procese kojima može da pošalje odgovarajuće komande. Nit u poslužiocu prihvata komande i poziva odgovarajuće funkcije iz biblioteke *clang_rt.profile*:

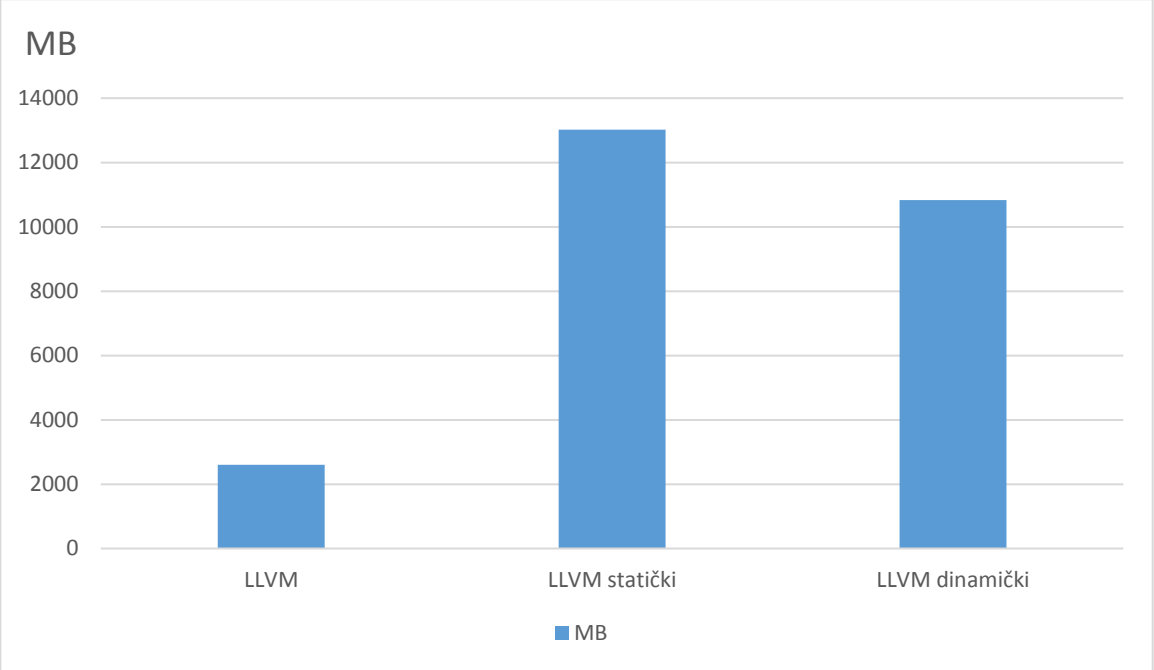
- __llvm_profile_write_file()
- __llvm_profile_reset_counters()

Prva funkcija omogućava funkcionalnost čuvanja podataka o izvršenosti koda na masovnu memoriju u proizvoljnom trenutku i poziva se po prijemu "*coverage_dump*" komade od klijentskog procesa. Po prijemu komande "*coverage_clean*" poziva se funkcija *__llvm_profile_reset_counters()* koja će postaviti vrednost brojača na nulu za dati instrumentalizovani proces.

6. Ispitivanje i rezultati

Sva ispitivanja su izvršena na operativnom sistemu baziranom na Linuksu i sve izmene koje su opisane u ovom radu su napravljene na programu otvorenog koda LLVM. Za potrebe ispitivanja uštede skladišne memorije korišćenjem deljene biblioteke *clang_rt.profile* korišćen je operativni sistem Ubuntu 16.04 i arhitektura x86_64. Takođe je ispitana i ispravnost ponuđenog rešenja upoređivanjem izlazne datoteke dobijene novim i starim pristupom. Za potrebe ispitivanja instrumentalizovan je sam programski prevodilac LLVM.

U prvom ispitnom slučaju prikazanom na Slici 18. odnosno grafiku zavisnosti rešenja i potrebne memorije možemo uočiti potrebnu memoriju za instalaciju programskog prevodioca LLVM u slučaju kada nije instrumentalizovan, prvi stub, kada je instrumentalizovan upotrebom statičkog pristupa *clang_rt.profile* biblioteke, drugi stub, i upotrebom dinamičkog pristupa biblioteke *clang_rt.profile*, treći stub.



Slika 17 Rezultati ispitivanja u slučaju programskog prevodioca LLVM

Sa grafika se uočava da je ostvarena ušteda od 2186 MB u memoriji primenom dinamičkog pristupa u odnosu na statiči što čini uštedu od oko 17% za ovaj ispitni slučaj. Prvi stub sa Slike 17. predstavlja memoriju koja je zauzeta prilikom instalacije prevodioca LLVM 3.8 bez izmena i koji je preveden korišćenjem alata Cmake i sledećim parametrima:

```
cmake -G "Unix Makefiles" -DCMAKE_INSTALL_PREFIX=../install_clean
../llvm
```

Drugi stub predstavlja memoriju potrebnu za instaliranje instrumentalizovanog LLVM 3.8 bez izmena koji je preveden sa prevodiocem LLVM 3.8 koji je korišćen za dobijanje rezultata za prvi stub. Za prevođenje ove verzije je takođe korišćen alat Cmake sa sledećim opcijama:

```
cmake -G "Unix Makefiles" -DCMAKE_INSTALL_PREFIX=../install_intr -
DCMAKE_C_FLAGS="-fprofile-instr-generate -fcoverage-mapping" -
DCMAKE_CXX_FLAGS="-fprofile-instr-generate -fcoverage-mapping"
../llvm
```

Treći stub predstavlja memoriju potrebnu za instaliranje instrumentalizovanog prevodioca LLVM 3.8 bez izmena koji je preveden prevodiocem LLVM 3.8 koji sadrži izmene koje su potrebne za korišćenje biblioteke *clang_rt.profile* kao dinamičke. U ovom slučaju je takođe korišćen alat Cmake sa sledećim opcijama:

```
cmake -G "Unix Makefiles" -DCMAKE_C_FLAGS="-fprofile-instr-generate -
fcoverage-mapping" -DCMAKE_CXX_FLAGS="-fprofile-instr-generate -
fcoverage-mapping" -DCMAKE_CXX_LINK_FLAGS=" -lclang_rt.profile-x86_64
-Wl,-rpath,/media/rtrk/Data/benchmark/llvm_shared/install/lib/clang/
3.8.0/lib/linux/ -L/media/rtrk/Data/benchmark/llvm_shared/install/lib/
```

```
clang/ 3.8.0/lib/linux/" -DCMAKE_C_LINK_FLAGS=" -lclang_rt.profile-
x86_64 -Wl,-rpath,/media/rtrk/Data/benchmark/llvm_shared/install/lib/
clang/ 3.8.0/lib/linux/ -L/media/rtrk/Data/benchmark/llvm_shared/
install/lib/ clang/3.8.0/lib/linux/" -DCMAKE_INSTALL_PREFIX=/media/
rtrk/Data/ benchmark/llvm_instr_with_shared/install2/  ../llvm
```

Prilikom rada instrumentalizovanog prevodioca LLVM 3.8 sa dinamičkim pristupom ispitane su funkcionalnosti za čuvanje ili brisanje podataka o pokrivenosti u proizvoljnom trenutku tako što su pokretanjem klijentskog programa i prosleđivanjem poruke „*coverage_dump*“ odnosno „*coverage_clear*“ čuvani podaci o pokrivenosti u proizvoljnom trenutku kao i brisanje istih. Prilikom pravljenja izveštaja o pokrivenosti sa podacima dobijenim na ovaj način nisu uočene nepravilnosti prilikom rada alata za obradu profilisanih podataka kao ni problemi u samom izveštaju tekstualnog formata. Prilikom pozivanja ovih funkcionalnosti u toku rada instrumentalizovanog programa odnosno LLVM-a nije uočeno da pozivanje istih u toku rada utiče na njegov rad i performanse.

Ispitana je ispravnost dobijenih izlaznih podataka pri dinamičkom pristupu. Za skup prostih primera je izvršena instrumentalizacija sa statičkim i dinamičkim pristupom i pri tome je uočeno da u izlaznim podacima nema razlike. Zatim su za oba slučaja i sve ispitne primere uspešno napravljeni izveštaji o pokrivenosti programskog koda korišćenjem alata *llvm-profdata* i *llvm-cov*. Oba alata su uspešno obradili podatke dobijene izvršavanjem instrumentalizovanog koda bez grešaka i upozorenja što znači da su podaci pri dinamičkom pristupu sačuvani u ispravnom formatu.

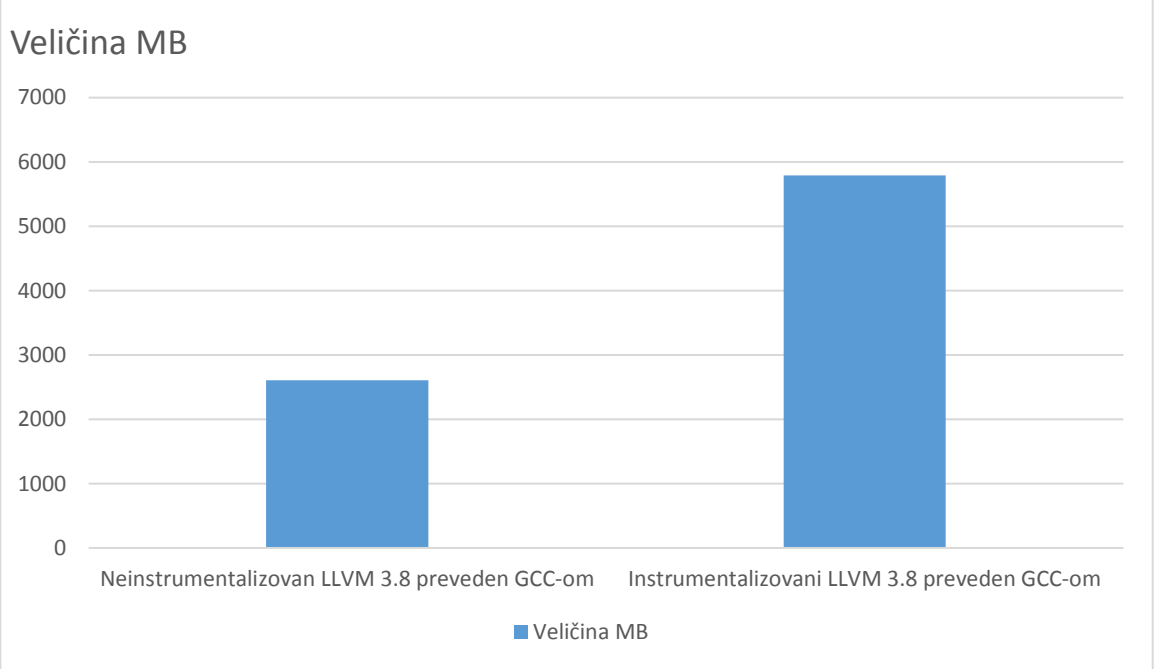
6.1 Poređenje rezultata sa programskim prevodiocem GCC

Kako bi se izmerila uspešnost alata o pokrivenosti programskog koda upoređeni su rezultati sa programskim prevodiocem GCC. Instrumentalizovana je ista verzija LLVM 3.8 koja je korišćena za ispitivanje performansi samog prevodioca LLVM 3.8. Za svrhe upoređivanja rezultata korišćeni su sledeći parametri: veličina instaliranog LLVM 3.8 prevedenog sa programskim prevodiocem GCC, veličina instaliranog LLVM 3.8 sa uključenom opcijom za pokrivenost koda takođe prevedenog sa programskim prevodiocem GCC, vreme koje je potrebno da se isti LLVM 3.8 kod prevede.

Korišćene su sledeće verzije prevodioca: LLVM 3.8.0 verzija *release 8. mart 2016* i GCC 5.4.0 verzija *release 3. jun 2016*. Izabrane su verzije približnih datuma kako bi se dobili što relevantniji rezultati. Prilikom ovog ispitnog slučaja korišćene su *release* verzije oba prevodioca jer verzija *debug* značajno utiče na vreme koje je potrebno da se sa takvom verzijom prevodioca prevede neki drugi programski kod. U slučaju prevodioca LLVM prilikom prevođenja je

eksplicitno navedena opcija *DCMAKE_BUILD_TYPE=Release*, u suprotnom podrazumevana verzija je *debug*.

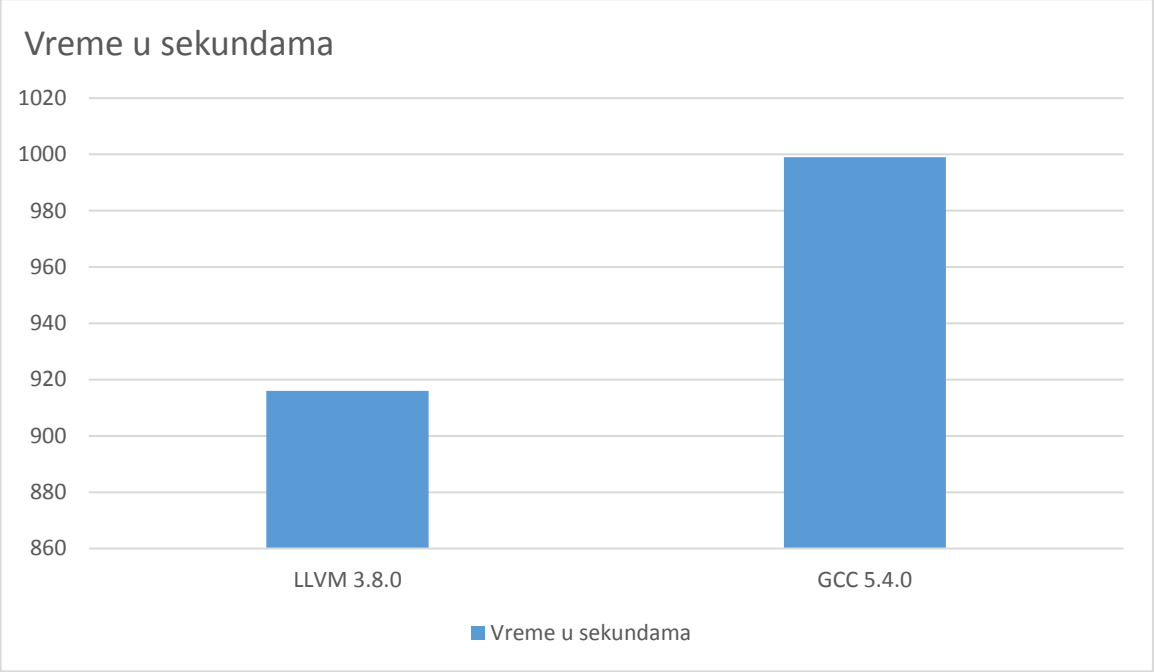
Prva analiza Slika 18. prikazuje zavisnost veličine instaliranog LLVM 3.8 prevedene sa programskim prevodiocem GCC 5.4.0 sa i bez uključene opcije za pokrivenost koda. Iz priložene slike se može zaključiti da je bez instrumentalizacije potrebno 2606 MB da bi se instalirao LLVM 3.8 dok je za instrumentalizovanu verziju potrebno 5792 MB.



Slika 18 Rezultati dobijeni korišćenjem programskog prevodioca GCC 5.4.0

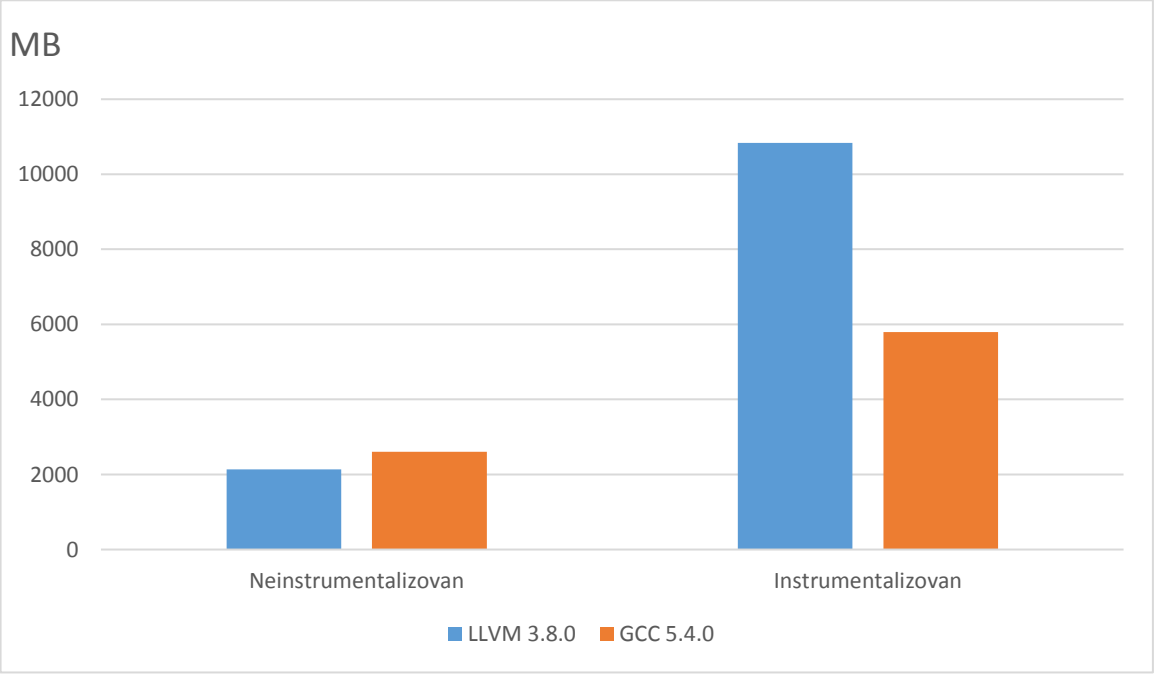
U drugoj analizi izmereno je vreme koje je potrebno da se prevede izvorni kod LLVM 3.8 sa prevodiocem GCC 5.4.0 i LLVM 3.8.0. U ovom slučaju je izmereno vreme samo za slučaj kada se dati programski kod prevodi bez uključene opcije za pokrivenost koda zbog nedostatka radne memorije i korišćenja pomoću (eng. *swap*) memorije što znatno utiče na vreme prevođenja.

Ispitivanje je izvršeno upotrebom procesora Intel i7-4790k sa 16GB radne memorije. U slučaju ispitivanja oba prevodioca vreme potrebno za prevođenje izvornog koda LLVM 3.8 sa uključenom opcijom pokrivenosti koda je premašivalo dvanaest sati zbog nedostatka radne memorije. Na Slici 19. su prikazani rezultati zavisnosti prevodioca i vremena koje je potrebno da se isti kod LLVM 3.8 prevede. Iz priloženog se može tvrditi da LLVM (15 minuta 16 sekundi) odlikuje kraća brzina prevođenja u odnosu sa GCC-om (16 minuta 39 sekundi).



Slika 19 Prikaz zavisnosti prevodioca i vremena potrebnog za prevođenje LLVM 3.8 bez uključene opcije za pokrivenost koda

Iz priloženih rezultata dobijenih ispitivanjem opcije za pokrivenost koda prevodioca GCC možemo potvrditi činjenicu da ovaj alata ima još prostora za poboljšanja kada je u pitanju LLVM. Upoređivanjem rezultata dobijenih poboljšanjem kojim se bavio ovaj rad pokazuju da je alat za pokrivenost koda kod prevodioca GCC i dalje mnogo bolji u odnosu na LLVM kada je u pitanju memorija koja je potrebna za instalaciju prevedenog koda (Slika 20).



Slika 20 Upoređivanje rezultata prevodioca LLVM 3.8.0 i GCC 5.4.0

7. Zaključak

U okviru ovog rada uspešno je realizovano i ispitano jedno unapređenje programskog prevodioca LLVM. Cilj ovog rada je bila ušteda memorije prilikom instrumentalizacije programskog koda za potrebe dobijanja izveštaja o pokrivenosti. Predloženo rešenje nije donelo značajnije poboljšanje, uočeno je da sama instrumentalizacija za potrebe pokrivenosti koda znatno povećava veliličinu objektnih datoteka i da je potrebno dodatno optimizovati celokupan pristup LLVM-a kada je u pitanju instrumentalizacija samog koda korišćenjem opcije PGO (eng. *Profile guided optimization*) i opcije za mapiranje podataka odnosno brojača za relevantan deo koda.

LLVM nudi još jedan pristup za dobijanje informacija o pokrivenosti koda koji je još uvek u fazi razvoja korišćenjem adresnog sanitajzera (eng. *SanitizerCoverage*). Trenutno rešenje nudi informacije da li je neki deo koda izvršen ili nije uz mogućost korišćenja osmobitnih brojača. Prema podacima sa zvanične stranice LLVM ovaj pristup zahteva manje memorije i znatno manje utiče na performance izvršavanja instrumentalizovanog koda od pristupa sa PGO. Mogućnost proširenja adresnog sanitajzera potencijalno može da zameni pristup korišćenjem PGO.

Takođe upoređivanjem načina za dobijanje podataka o pokrivenosti koda sa drugim programskim prevodiocima kao što su GCC i ICC ustanovljeno je da postojeće rešenje LLVM-a koje je korišćeno u ovom radu ima prostora za unapređenje. Prilikom instrumentalizacije za potrebe dobijanja podataka o pokrivenosti koda sa programskim prevodiocem GCC u toku prevođenja za svaki instrumentalizovanu datoteku C biće napravljena istoimena datoteka *gcno* koja sadrži informacije o mapiranju podataka dok će nakon izvršavanja instrumentalizovanog koda biti napravljena datoteka *gcda* koja sadrži informacije o izvršenosti instrumentalizovanih datoteka. Kombinovanjem ove dve datoteke moguće je napraviti izveštaj o izvršenosti koda korišćenjem alata *gcov*.

Programski prevodilac ICC prilikom prevođenja koda sa uključenom opcijom za instrumentalizaciju podatke takođe koristi pristup PGO i u trenutku prevođenja pravi datoteku *spi* gde se čuvaju informacije o mapiranju brojača, takođe nakon izvršavanja datog instrumentalizovanog koda biće izgenerisana datoteka *dyn* koja sadrži informacije o brojačima. Korišćenjem ove dve datoteke moguće je izgenerisati izveštaj u više formata korišćenjem alata *codecov*. Iz date analize ustanovljeno je da LLVM sve podatke o mapiranju brojača nepotrebno čuva u samoj objektnoj datoteci čime se povećava veličina prevedenog programa. Deo ovih informacija je takođe moguće izmestiti u posebnu datotkeku kako bi se uštedelo na memoriji prevedenog programa. Ova ušteta je naročito pogodna prilikom prevođenja na druge arhitekture (eng. *Cross compiling*) i namenske uređaje (eng. *Embedded device*) koji imaju ograničenu veličinu memorije.

8. Literatura

[1] *Getting Started with LLVM Core Libraries*, Bruno Cardoso Lopes, Bruno Cardoso Lopes, August 26, 2014

[2] *PGO and LLVM Status and Current Work*, Bob Wilson, Diego Novillo, Chandler Carruth, 11.2013.

[3] https://releases.llvm.org/3.8.0/docs/CoverageMappingFormat.html

[4] https://clang.llvm.org/docs/SourceBasedCodeCoverage.html

[5] http://releases.llvm.org/3.8.0/tools/docs/SanitizerCoverage.html

[6] https://releases.llvm.org/3.8.0/docs/CommandGuide/llvm-cov.html

[7] https://gcc.gnu.org/onlinedocs/gcc/Gcov-Intro.html

[8] Intel C++ Compiler 16 User and Reference Guides, Intel Software Development Tools 2016

[9] Clang: Much mor than just a C/C++ Compiler, Tilmann Scheller, Samsung Open Source Group, LinuxCon Europe 2016 Berlin, Germany October 2016

[10] AddressSanitizer + Code Coverage, Kostya Serebrayany, Google EuroLLVM 2014