UNIVERZITET U BEOGRADU
MATEMATIČKI FAKULTET

Petar Tešić

AUTOMATSKA DETEKCIJA I OPTIMIZACIJA ALGORITMA CRC U OKVIRU KOMPAJLERSKE INFRASTRUKTURE LLVM

master rad

Beograd, 2024.

Mentor:

dr Milena Vujošević Janičić, vanredni profesor
Univerzitet u Beogradu, Matematički fakultet

Članovi komisije:

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

dr Mirko Spasić, docent
Univerzitet u Beogradu, Matematički fakultet

Datum odbrane: Septembar, 2024.

Zahvaljujem se svojoj porodici, prijateljima, Kaći, profesorki Mileni i svim kolegama iz kompanije SYRMIA na nesebičnoj pomoći i podršci

Naslov master rada: Automatska detekcija i optimizacija algoritma CRC u okviru kompajlerske infrastrukture LLVM

Rezime: U savremenoj industriji razvoja softvera, efikasnost i optimizacija koda predstavljaju ključne aspekte u postizanju visokih performansi računarskih sistema. Ovaj master rad istražuje inovativan pristup prevođenju algoritma CRC (eng. *Cyclic Redundancy Check*) korišćenjem kompilatorske infrastrukture LLVM. Algoritam CRC detektuje potencijalne promene u podacima nastalim usled transfera kroz različite medijume (žičane mreže, bežične mreže ili optičke kablove) i ima široku primenu u digitalnoj komunikaciji, gde se koristi za proveru integriteta podataka. Zbog svoje učestale primene važno je koristiti optimizovane verzije ovog algoritma.

Cilj ovog rada je unapređenje infrastrukture LLVM u kontekstu prevođenja algoritma CRC, i na taj način ostvarivanje boljih performansi programa koji koriste algoritam CRC i LLVM kao svoj kompilator. Osnovna ideja rešenja predstavljenog u radu zasniva se na detekciji neoptimizovane verzije algoritma CRC na nivou LLVM međureprezentacije i zamenjivanju funkcionalno ekvivalentnom optimizovanom verzijom. Unapređenje je vidljivo na različitim procesorskim arhitekturama, specijalno i na procesorskoj arhitekturi RISC-V. Rezultati dobijeni testiranjem predstavljenog rešenja pokazuju značajno poboljšanje performansi algoritma CRC prevedenog LLVM kompilatorom.

Ključne reči: kompilatorska infrastruktura LLVM, algoritam CRC, arhitektura RISC-V, LLVM međureprezentacija, LLVM mašinski zavisan međukod, domenski specifičan jezik TableGen, LLVM optimizacioni prolaz

Sadržaj

Glava 1

Uvod

Programski prevodioci prevode programe iz viših programskih jezika u mašinski kôd i time omogućavaju izvršavanje softvera na različitim arhitekturama računara. Zahvaljujući optimizacijama koje vrše prevodioci dobija se kvalitetniji izvršivi kôd, odnosno kôd koji se izvršava brže, koji zauzima manju količinu memorije ili koji ima bolju energetsku efikasnost.

Veliki broj softverskih kompanija ulaže značajna finansijska sredstva u razvoj programskih prevodilaca, odnosno u razvoj kompilatorskih infrastruktura, sa ciljem da softver koji oni proizvode ima kvalitetan izvršivi kôd. Na primer, kompanije *Apple* [38], *Google* [35], *Nvidia* [11] ulažu u razvoj kompilatorske infrastrukture LLVM [26], kojoj je ovaj rad posvećen.

LLVM je jedan od najpopularnijih prevodilaca za programske jezike kao što su *C*, *C++*, *Rust* i *Swift*. U ovom radu biće predstavljene optimizacije implementirane u okviru kompilatora LLVM koje treba da omoguće efikasniji rezultat prevođenja algoritma CRC [39]. Optimizacije su sprovedene na nivou LLVM međureprezentacije. Jedna od optimizacija ima za cilj upotrebu na RISC-V arhitekturi [43], dok je drugu moguće koristiti i na drugim arhitekturama.

Rad se sastoji od šest poglavlja. U poglavlju 2 se govori o procesu prevođenja programa, metodologijama konstrukcije kompilatora i o kompilatorskoj infrastrukturi LLVM. Predstavljena je istorija samog projekta, arhitektura projekta, delovi kompilatora i međureprezentacija koju LLVM koristi. Dodatno, opisana je i infrastruktura za testiranje. Poglavlje 3 je posvećeno algoritmu CRC i problemu njegovog prepoznavanja. Predstavljen je koncept detektovanja grešaka u podacima, zajedno sa trenutnim načinima korekcije i detekcije grešaka. Poglavlje 4 je posvećeno procesorskim arhitekturama, sa akcentom na procesorsku arhitekturu

RISC-V jer je baš za nju predložena jedna optimizacija u okviru ovog rada. U poglavlju 5 su predstavljena dva rešenja problema detektovanja algoritma CRC i njegovog optimizovanja. Na početku poglavlja su opisani ideja i postupak zajednički za oba rešenja, a u nastavku su redom nalaze detalji implementacije svakog od predloženih rešenja. Poslednja sekcija u poglavlju 5 opisuje rezultate eksperimentalne evaluacije predloženih rešenja. Rezultati pokazuju za čak 35% bolju vremensku efikasnost programa nad kojim je pokrenuta jedna od predloženih implementacija. U poglavlju 6 je sumirano sve što je predstavljeno u radu i ostavljene su smernice i sugestije čitaocima rada za dalje istraživanje na temu unapređenja LLVM i drugih kompilatora, a i na temu kompilatorskih optimizacija.

Glava 2

Kompilatori i projekat LLVM

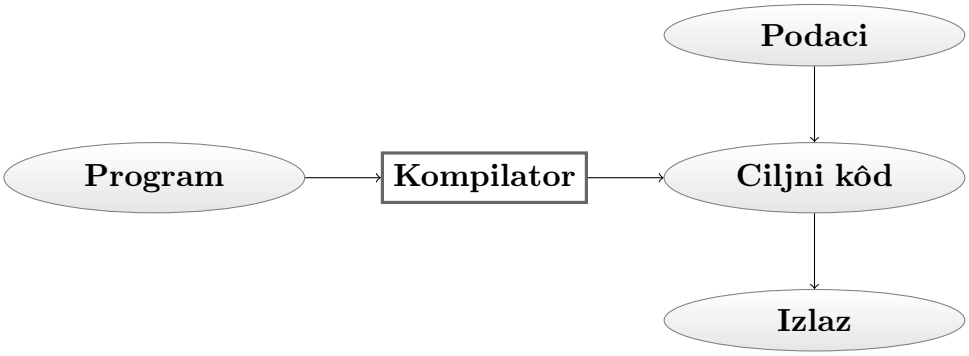
Izvršavanje programa omogućavaju programski prevodioci koji prevode program iz višeg programskog jezika u kôd koji računar može da izvrši. Za samo izvršavanje programa, posebno je važna efikasnost koja u velikoj meri zavisi od optimizacija koje programski prevodioci sprovode u toku prevođenja programa.

2.1 Osnovne vrste prevodilaca

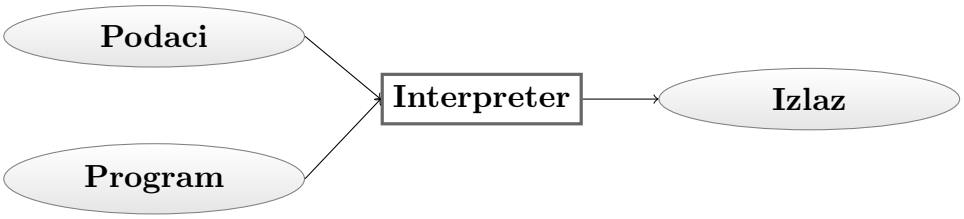
Programski prevodioci mogu vršti kompilaciju programa ili njegovu interpretaciju. Za kompilaciju programa koriste se kompilatori, a za interpretaciju interpreteri.

Kompilatori iliti kompajleri predstavljaju veoma kompleksan softver sposoban da proizvoljan kôd napisan u nekom višem programskom jeziku konvertuje u kvalitetan mašinski kôd koji će se potom izvršavati na računaru. Proces kompilacije podrazumeva razdvojenost procesa prevođenja i procesa izvršavanja programa. Kompilacijom se kreira izvršiva datoteka koju je potom moguće pokretati, to jest, izvršavati na računaru proizvoljan broj puta. Za svaku promenu u izvornom kodu potrebno je ponoviti proces kompilacije. Neki od najpoznatijih kompilatora otvorenog koda danas su GCC [15], LLVM, RUSTC [14] i GraalVM [37]. Primeri programskih jezika za čije prevođenje je potrebno koristiti kompilaciju su C, C++, Rust i Go. Skica procesa kompilacije prikazana je na slici 2.1.

Prilikom procesa interpretacije faze prevođenja i izvršavanja programa su isprepletane. To znači da se interpretacijom svaka naredba prvo prevodi, a potom se izvršava. Bilo koja greška u programu biće otkrivena tek kada se prevođenjem i izvršavanjem dođe do linije u kojoj se ona nalazi. Skica procesa interpretacije



Slika 2.1: Proces kompilacije i izvršavanja



Slika 2.2: Proces interpretacije i izvršavanja

prikazana je na slici 2.2. U nastavku rada svako spominjanje procesa prevođenja odnosiće se isključivo na kompilaciju.

2.2 Faze prevođenja programa

U okviru procesa prevođenja programa postoje više različitih faza [2]:

- leksička analiza,
- sintaksička analiza,
- semantička analiza,
- generisanje međukoda,
- optimizacija međukoda i
- generisanje i optimizacija mašinskog koda.

Bitan korak koji se izvršava pre faze leksičke analize, ali koji ne smatramo posebnom fazom prevođenja, jeste pretprocesiranje. U okviru pretprocesiranja se obrađuju direktive poput `#include` i `#define` direktiva u programskom jeziku `C`.

Korak koji se izvršava nakon prevođenja, a koji takođe ne smatramo posebnom fazom prevođenja, jeste linkovanje. Linkovanjem se kreira jedinstvena izvšiva datoteka od jednog ili više objektnih modula. Objektni moduli mogu nastati ili kompilacijom izvornog koda ili sadržati mašinski kôd i podatke standardne ili neke nestandardne biblioteke.

U fazi leksičke analize, za koju je odgovaran deo programskog prevodioca koji zovemo leksički analizator ili lekser, ustanovljava se da li su u izvornom kodu iskorišćene samo dopustive lekseme iliti tokeni. Pod dopustivim leksemama podrazumevamo reči napisane u izvornom kodu koje mogu biti prepoznate konačnim automatima kreiranim na osnovu odgovarajućih regularnih izraza. Regularni izrazi imaju za cilj da opišu imena promenljivih, ključne reči, operatore, separatore i tako dalje.

Ukoliko je program koji prevodimo sačinjen isključivo od dopustivih leksema, u tom slučaju niz prepoznatih leksema se prosleđuje delu programskog prevodioca zaduženom za sintaksičku analizu, sintaksičkom analizatoru iliti parseru. U protivnom, u slučaju detektovanja nedopustive lekseme, proces prevođenja se zaustavlja i programski prevodilac nam saopštava poruku o leksičkoj greški.

U narednoj fazi, fazi sintaksičke analize, proverava se da li su prepoznate lekseme složene u skladu sa gramatikom programskog jezika u kome je izvorni kôd napisan. Pravila slaganja leksema odnosno formiranja rečeničnih konstrukcija se zadaju korišćenjem kontekstno-slobodne gramatike (eng. *context-free grammar*).

Prolaskom kroz naredbe programa i korišćenjem potisnih automata (kreiranim na osnovu gramatike) proverava se da li su instrukcije u skladu sa gramatikom programskog jezika. Ukoliko jesu, na osnovu njih se gradi stablo parsiranja (eng. *parse tree*).

Stablo parsiranja je struktura nalik stablu koja predstavlja sintaksičku strukturu programa. Sastoji se od čvorova, od kojih svaki odgovara elementu programskog jezika u kome je program napisan. Unutrašnji čvorovi stabla parsiranja su predstavljeni neterminalima (ključnim rečima, operatorima, ...), dok su listovi stabla parsiranja predstavljeni terminalima (konstantama, imenima promenljivih, ...). Stablom parsiranja se ne apstrahuju detalji sintakse programskog jezika. Time ova struktura predstavlja detaljniji prikaz izvornog koda i često je korisna za debagovanje.

Prolaskom kroz stablo parsiranja se prikupljaju informacije o funkcijama, promenljivama i drugim objektima i smeštaju se u tabelu simbola. Korišćenjem tabele

simbola proverava se struktura stabla parsiranja. Na osnovu stabla parsiranja se potom kreira apstraktno sintaksno stablo programa (eng. *Abstract Syntax Tree - AST*) [27].

Apstraktno sintaksno stablo je u suštini pojednostavljena verzija stabla parsiranja. Čvorovi apstraktnog sintaksnog stabla su apstrahovani od detalja sintakse programskog jezika, što ovu strukturu čini sažetijom i lakšom za čitanje od stabla parsiranja. Apstraktnim sintaksnim stablom je, takođe, olakšano i zaključivanje o strukturi koda.

U slučaju upotrebe nedozvoljene rečenične konstrukcije, proces prevođenja se zaustavlja i prevodilac nam saopštava poruku o sintaksnoj greški. Ukoliko do takve greške nije došlo, započinje se faza semantičke analize u kojoj se proverava značenje ispravnim rečeničnim konstrukcijama. Provera semantičke ispravnosti programa podrazumeva više nezavisnih provera poput provere tipova, provere labela, provere kontrole toka programa. Na primer, proverom tipova se proverava da li svaka operacija u programu podržana sistemom tipova jezika u kome je program napisan.

Tokom faze generisanja međukoda se od izvornog koda generiše kôd nezavisan od programskog jezika koji nazivamo međukodom iliti međureprezentacijom. Međureprezentacija služi kao most između visokog nivoa apstrakcije izvornog koda i niskog nivoa apstrakcije instrukcija ciljne arhitekture. Nakon generisanja međukoda sledi faza njegove optimizacije. Faza generisanja mašinskog koda i njegova optimizacija je poslednja faza prevođenja u kojoj se od optimizovanog međukoda dobija krajnji mašinski kôd za ciljnu procesorsku arhitekturu. Rešenja opisana u ovom radu predstavljaju optimizacije koje se odvijaju upravo u fazama optimizacije međukoda i optimizacije mašinskog koda.

2.3 Bitni delovi kompilatora

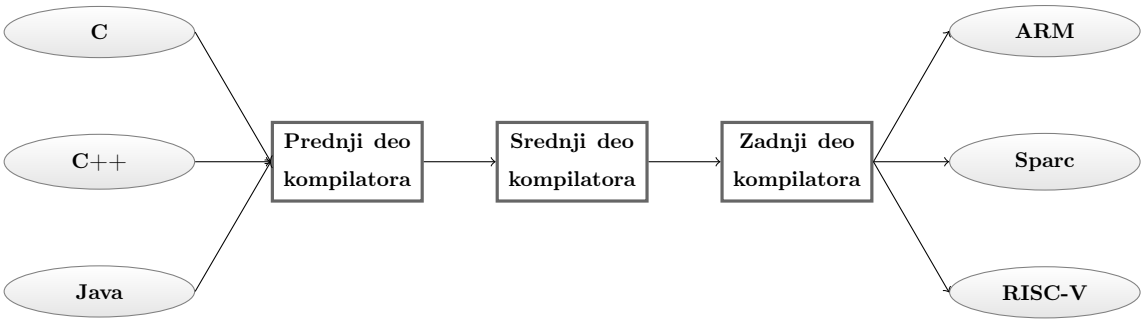
U arhitekturi većine današnjih kompilatora moguće je uočiti tri bitna dela. U pitanju su prednji deo [3], srednji deo [4] i zadnji deo [5] kompilatora.

Prednji deo kompilatora (eng. *front end*) je zavisan od višeg programskog jezika kojeg prevodi. Odgovornost prednjeg dela kompilatora jeste obavljanje leksičke, sintaksičke i semantičke analize. Kao rezultat rada prednjeg dela kompilatora generiše se međukod odnosno međureprezentacija (eng. *intermediate representation*) izvornog koda koja se prosleđuje srednjem delu kompilatora.

Srednji deo kompilatora (eng. *middle end*) je nezavisan od programskog jezika i ciljne arhitekture. Njegova odgovornost jeste sprovođenje optimizacija na međukodu (međureprezentaciji) kako bi se unapredile njegove performanse i kako bi se od istog dobio kvalitetan mašinski kôd na samom kraju kompilacije. Optimizacije srednjeg dela kompilatora su nezavisne od procesorske arhitekture za koju je krajnji kôd namenjen.

Da bi srednji deo kompilatora izvršio kvalitetnu optimizaciju, najpre vrši analizu koda. Analiza obuhvata prikupljanje informacija o programu na osnovu generisane međureprezentacije. Postoje razne vrste analiza, kao što su analiza toka podataka [13], analiza zavisnosti [40], analiza aliasa [21], analiza pointera [28] i druge. Precizne analize su preduslov za kvalitetnu optimizaciju.

Korišćenjem rezultata analize, pokreće se odgovarajuća optimizacija. Pod optimizacijom podrazumevamo transformisanje međureprezentacije u funkcionalno ekvivalentan, ali brži ili manji oblik. Izbor optimizacija koje će biti izvršene zavisi od želja korisnika i argumenata koji se zadaju prilikom pokretanja kompilatora.



Slika 2.3: Arhitektura modernih kompilatora

Zadnji deo kompilatora (eng. *back end*) zavisi od ciljne arhitekture i zadužen je za generisanje koda za ciljnu arhitekturu. Generisanje koda podrazumeva transliranje transformisanog međukoda u izlazni jezik, obično mašinski jezik računarskog sistema. Ovo uključuje odluke vezane za resurse i skladištenje podataka, kao na primer koje promenljive će biti u registrima a koje u memoriji, i izbor i određivanje redosleda instrukcija zajedno sa odgovarajućim tipovima adresiranja. Takođe, potrebno je generisati i podatke za dabagovanje kako bi se omogućilo debagovanje izvornog programa.

Zadnji deo kompilatora je odgovoran i za sprovođenje mašinski zavisnih optimizacija, odnosno optimizacija koje zavise od detalja arhitekture procesora na

kojem će se program izvršavati. Mašinski zavisne optimizacije obuhvataju, na primer, prepisivanje kraćih nizova instrukcije u odgovarajuće efikasnije instrukcije.

Za generisanje mašinskog koda potrebni su samo prednji i zadnji deo kompilatora. Optimizator odnosno srednji deo kompilatora je opcioni i nastao je iz potrebe da dobijeni kôd ima dobre performanse. Svaki dobar kompilator ga sadrži i on predstavlja njegovu najveću komponentu.

2.4 Kompilatorske optimizacije

Kompilator sprovodi optimizacije sa ciljem povećanja performansi krajnjeg izvršivog koda. Uslov koji svaka optimizacija mora da zadovolji jeste da se optimizovani kôd ponaša semantički ekvivalentno originalnom kodu. Performanse mogu da se odnose na vreme izvršavanja programa, memorijski prostor potreban prilikom njegovog izvršavanja, memorijski prostor potreban za skladištenje izvršive datoteke ili na energetsku efikasnost programa.

Kompilatorske optimizacije mogu biti nezavisne ili zavisne od ciljne arhitekture. Nezavisne optimizacije rade za sve ciljne arhitekture i najčešće smanjuju ukupan broj operacija koje treba izvršiti. Zavisne optimizacije koriste detalje arhitekture kako bi povećale performanse. To podrazumeva korišćenje instrukcija koje obavljaju više operacija u isto vreme, imaju kraći zapis ili rade brže od odgovarajućih instrukcija u izvornom kodu. Na primer, na arhitekturi `x86-64` se često instrukcija `mov eax, 0` zamenjuje instrukcijom `xor eax, eax`.

Optimizacije se u odnosu na opseg koda koji obrađuju mogu podeliti na: lokalne optimizacije, globalne optimizacije i međuproceduralne optimizacije. Lokalne optimizacije rade na nivou jednog osnovnog bloka (eng. *basic block*) i predstavljaju najjednostavnije optimizacije. Globalne optimizacije operišu na nivou pojedinačnih funkcija. Većina lokalnih optimizacija se može modifikovati da radi globalno. Međuproceduralne optimizacije rade na nivou čitavog programa. One istovremeno obrađuju više funkcija od kojih neke mogu biti i u različitim jedinicama prevođenja.

Neke od popularnih optimizacija su umetanje koda (eng. *inlining*), eliminacija mrtvog koda (eng. *dead code elimination*), sažimanje konstanti (eng. *constant folding*), propagiranje konstanti (eng. *constant propagation*), razmotavanje petlji (eng. *loop unrolling*) i neke vrste automatske paralelizacije. U nastavku će samo neke od navedenih optimizacija biti opisane.

Umetanje koda (eng. *inlining*) se koristi jer su pozivi funkcija skupa operacija zbog toga što menjaju kontrolu toka izvršavanja i zahtevaju dodatne instrukcije za čuvanje i učitavanje vrednosti registara. Iz tog razloga je poželjno pozive kratkih funkcija zameniti njihovom definicijom. Time se izbegava cena poziva o trošku veće količine memorije za čuvanje programa.

Razmotavanje petlji (eng. *loop unrolling*) podrazumeva da se telo petlje ponovi više puta u jednoj iteraciji i da se broj iteracija samim tim isti broj puta smanji. Izvršavanje petlji sa kratkim telom dovodi do velikog broja skokova u kratkom vremenskom intervalu. Skokovi su skupa instrukcija i njihovim smanjenjem moguće je znatno povećati performanse programa. Ovo je moguće samo u posebnim situacijama kada kompilator ima dovoljno informacija o načinu izvršavanja petlje.

Eliminacija mrtvog koda (eng. *dead code elimination*) podrazumeva uklanjanje dela programa za koga je statičkom analizom utvrđeno da se nikada neće izvršiti. Brisanjem takvog koda može se uštedeti memorija i ubrzati proces kompilacije. Takođe, ovom tehnikom je, zbog bolje organizacije keš memorije, moguće ubrzati i vreme izvršavanja programa.

Postoji više različitih nivoa optimizacija koji se mogu pokrenuti prilikom poziva kompilatora. Izbor nivoa optimizacije i efekat optimizacija su drugačiji na različitim kompilatorima. Na primer, kompilatori `gcc` i `Clang` omogućavaju korisniku da izabere nivo optimizacije tako što kompilatoru prosledi opciju `-O` iza koje sledi neki od karaktera `0`, `1`, `2`, `3` ili `s`.

Opcijom `-O0` se naglašava da kôd nije potrebno optimizovati. Ova opcija je podrazumevana ukoliko nijedan druga opcija nije navedena. Opcije `-O1` i `-O2` uključuju redom sve veći broj optimizacija, ali ne povećevaju drastično veličinu memorije koju program koristi prilikom izvršavanja. Opcija `-O3` predstavlja najveći nivo optimizacije posvećen smanjenju vremena izvršavanja programa. Sa druge strane *-Os* nivo podrazumeva optimizacije koje smanjuju memorijsko zauzeće, ali povećavaju vreme izvršavanja.

Svaki od opisanih nivoa optimizacija ima svoju konkretnu primenu. Na primer, nivo -O0 se koristi za pravljenje `debug` verzija programa. Isporučene verzije programa se često kompiliraju navođenjem opcije -O3, dok se nivo -Os koristi za uređaje sa ugrađenim računarom (eng. *embedded devices*).

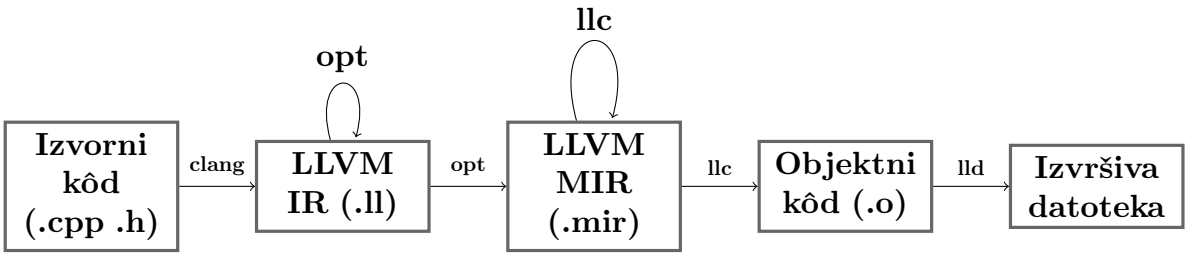
2.5 Kompilatorska infrastruktura LLVM

Projekat LLVM [26] je započet kao istraživački projekat 2000. godine na Univerzitetu Ilinois od strane Krisa Latnera, čoveka koga je ovaj projekat i proslavio, a koji danas radi na razvoju programskog jezika Mojo [31]. Cilj projekta LLVM je bio proučavanje tehnika statičkog i dinamičkog prevođenja proizvoljnih programskih jezika korišćenjem kompiliranja u SSA obliku (eng. *Static Single Assignment*). Naziv LLVM je predstavljao akronim za „virtuelna mašina niskog nivoa" (eng. *Low Level Virtual Machine*). Međutim, isti akronim se više ne koristi, ali je ime projekta ostalo nepromenjeno. Danas, projekat sadrži veliki broj biblioteka i alata koji se koriste kako u komercijalne svrhe tako i u svrhe razvoja projekata otvorenog koda. Svaki deo projekta je dizajniran kao biblioteka tako da se može ponovo upotrebiti za implementiranje drugih alata. Celokupan izvorni kôd je javno dostupan na servisu GitHub i oko njega je formirana velika zajednica ljudi koji rade na različitim delovima projekta i svakodnevno ga unapređuju. Veliki broj kompanija koristi svoje verzije kompilatora LLVM bilo u celosti bilo samo neke njegove delove (prednji, srednji ili zadnji deo kimpilatora) za podršku neke procesorkse arhitekture ili kao osnovu za novi programski jezik.

O popularnosti projekta LLVM govori i činjenica da je 2012. godine projekat osvojio nagradu *ACM Software System Award* [30] (nagrada se dodeljuje jednom softverskom sistemu godišnje, počevši od 1983. godine), kao i da se dva puta u toku svake godine organizuje *LLVM Developers Meeting* i *LLVM WorkShop* u različitim gradovima Evrope i Amerike. Licenca koda u okviru LLVM projekta je "Apache 2.0 License with LLVM exceptions". Licenciranje se menjalo tokom razvoja projekta, ali je projekat uvek bio otvorenog koda.

LLVM se sastoji od velikog broja manjih potprojekata koji predstavljaju zasebne i funkcionalne celine. Neki od primarnih potprojekata su *LLVM Core Libraries* [22], *Clang* [17], *MLIR* [45], *OpenMP* [34], *polly* [1], *klee* [44], *LLD* [20]. Svi potprojekti zajedno čine potpun programski prevodilac koji pored svog prednjeg, srednjeg i zadnjeg dela poseduje optimizatore, asemblere, linkere i druge alate koji pružaju podršku tokom čitavog procesa prevođenja. LLVM implementira i svoj debager koji se zove LLDB [25].

Kompilator LLVM je relativno nov u odnosu na druge popularne kompilatore i prati moderniji dizajn. Za razliku od kompilatora GCC, koji je napisan u jeziku C i ima monolitnu strukturu, LLVM uživa u pogodnostima koje nudi jezik



Slika 2.4: Prikaz promena internih reprezentacija programa prilikom prevođenja LLVM kompilatorom

C++ pritom koristeći modularnu arhitekturu. Najveći deo projekta je napisan u programskom jeziku C++, ali postoji i interfejs za povezivanje sa jezicima C i Python. Implementacija kompilatora LLVM prati opštu strukturu kompilatora prikazanu u poglavlju 2.3. Korake kompilacije izvršavaju različiti alati. Odnos različitih reprezentacija programa u toku kompilacije i alata koji ga konvertuju iz jedne reprezantacije u drugu je prikazan na slici 2.4.

U nastavku će redom biti opisani prednji, srednji i zadnji deo kompilatora LLVM sa najvećim fokusom na zadnji deo kompilatora LLVM. Jedno od rešenja koje će u poglavlju 5 biti opisano zahteva dobro poznavanje rada zadnjeg dela kompilatora LLVM, pa je zato ovom delu posvećeno najviše pažnje.

Prednji deo LLVM-a

Prednji deo LLVM kompilatora je zadužen za pokretanje leksičke, sintaksičke i semantičke analize, kreiranje apstraktnog sintaksnog stabla i na kraju generisanje LLVM međukoda. Projekat LLVM podržava više različitih prednjih delova kompilatora. Mnoge kompanije takođe razvijaju i održavaju prednje delove kompilatora LLVM za svoje programske jezike. Zahvaljujući tome, LLVM podržava veliki broj programskih jezika kao što su C, C++, Objectve-C, Fortran, Haskell, Swift i drugi.

Prednji deo kompilatora prilagođen jezicima C i C++ koji su korišćeni u ovom radu jeste Clang. Termin Clang može imati više značenja. Clang može predstavljati prednji deo kompilatora, a takođe može predstavljati i čitav kompilator odnosno alat koji upravlja čitavim procesom kompilacije. U okviru rada, termin Clang će od sada predstavljati prednji deo LLVM kompilatora.

Funkcija Clang-a je, dakle, da izvorni kôd napisan u jezicima C ili C++ prevede u LLVM međureprezentaciju. Da bi se takvo prevođenje izvelo potrebno je

program prvo detaljno izanalizirati, a potom započeti transformacije izvornog koda. U okviru Clang-a sintaksička i semantička analiza su usko povezane. Ukoliko se pronađe greška u bilo kojoj analizi, proces prevođenja se zaustavlja i poruka o razlogu se saopštava korisniku. U suprotnom, obilaskom apstraktnog sintaksnog stabla generiše se LLVM međukod. Ukoliko se program uspešno prevede do međukoda, to znači da je izvorni kôd ispravan za jezik u kome je napisan. Komanda za prevođenje izvornog C programa u LLVM međureprezentaciju (međukod) je sledeća:

```
clang -S -emit-llvm input.c -o output.ll
```

Srednji deo LLVM-a

Srednji deo LLVM kompilatora zadužen je za sprovođenje mašinski nezavisnih optimizacija nad LLVM međureprezentacijom (međukodom) dobijenom od strane *Clang*-a. LLVM međukod je napisan u SSA obliku pogodnom za izvršavanje optimizacija. Optimizacije srednjeg dela LLVM kompilatora su implementirane u vidu prolaza (eng. *pass*). Prolazi implementiraju funkciju `run` sa argumentima za odgovarajuću jedinicu obrade. Jedinica obrade može biti modul, funkcija ili petlja. Prolazi se dele na analize i transformacije.

Analize prikupljaju podatke, a transformacije te podatke koriste kako bi izmenile kôd. Transformacije zahtevaju različite vrste analiza koje mogu biti poništene nakon promene koda (ukoliko više ne važe). Optimizovanje LLVM međukoda se vrši inkrementalno, tako što međukod dobijen kao rezultat jedne optimizacije postaje ulaz sledećoj optimizaciji. Konačan LLVM međukod zavisi od redosleda izvršavanja optimizacija. Loš redosled može da rezultuje znatno lošijim performansama međukoda, a kasnije i izvršive datoteke.

Redosled izvršavanja optimizacionih prolaza određuje LLVM-ov menadžer prolaza (eng. *LLVM Pass Manager*). Menadžer prolaza je takođe odgovoran za upravljanje rezultatima sprovedenih analiza. Rezultati analize bi trebalo da se dele između različitih prolaza kada god je to moguće, budući da je ponovno izračunavanje analiza skupo. Da bi to uradio, LLVM-ov menadžer prolaza mora keširati rezultate analiza ili ih ponovo izračunati u slučaju da su poništeni transformacijama.

U okviru LLVM kompilatora postoje trenutno dva menadžera prolaza i to stari (eng. *Legacy Pass Manager*) i novi (eng. *New Pass Manager*). Stari menadžer

prolaza se koristio dugi niz godina, ali se zbog efikasnosti prešlo na korišćenje novog menadžera. Međutim, i dalje se radi na tome da se sve funkcionalnosti podržane na starom menadžeru implementiraju i na novom.

Zadatak srednjeg dela LLVM kompilatora obavlja alat *opt*. Alat kao svoje argumente očekuje imena optimizacija koje treba da pokrene i datoteku sa ispravnim LLVM IR kodom koga optimizuje. Primer poziva alata *opt* za pokretanje optimizacionih prolaza `loweratomic` i `instnamer` je:

```
opt input.ll -passes="loweratomic,instnamer" -o output.ll
```

Zadnji deo LLVM-a

Zadnji deo kompilatora je odgovoran za generisanje mašinskog koda za ciljnu arhitekturu, ali i mašinski zavisne optimizacije. Poslednja, mašinski zavisna, reprezentacija kroz koju izvorni kôd prolazi neposredno pre generisanja mašinskog koda se u LLVM terminologiji naziva MIR (eng. *Machine Intermediate Representation*). Instrukcije MIR koda su dosta slične instrukcijama u IR kodu. Međutim, MIR reprezentacija je bliža ciljnoj arhitekturi, te su virtuelni registri zamenjeni pravim fizičkim registrima, a intrukcije MIR koda su stvarno izvršive na konkretnoj arhitekturi. Kompletan pregled jezika dostupan je u okviru dokumentacije projekta LLVM [23].

Prevođenje reprezentacije programa sa mašinski nezavisne na mašinski zavisnu reprezentaciju predstavlja vrlo kompleksan korak koji se naziva izbor instrukcija (eng. *instruction selection*). U okviru LLVM-a postoje tri implementacije izbora instrukcija i to su: `SelectionDag`, `GlobalIsel` i `FastIsel`.

SelectionDAG predstavlja podrazumevanu implementaciju izbora instrukcija za većinu procesorskih arhitektura koja koristi grafovsku reprezantaciju programa i algoritme za uparivanje čvorova i različite vrste obilaska grafa.

GlobalIsel predstavlja novu implementaciju izbora instrukcija sa modularnim dizajnom, poboljšanim performansama i mogućnošću da optimizuje veći deo kôda. Implementacija `GlobalIsel` nije u potpunosti završena, ali postoji plan inženjera okupljenih oko LLVM-a da ovaj izbor instrukcija zameni `SelectionDAG`.

FastIsel je pristup izboru instrukcija koji radi veoma brzo, ali generiše neoptimizovan kôd. Takođe, postoje instrukcije čije spuštanje nije podržano `FastIsel` pristupom.

Rešenja koja će u ovom radu biti predstavljena se oslanjaju na izbor instrukcija `SelectionDAG` zbog njegove stabilnosti i daleko veće korišćenosti u vreme pisanja rada. Pristup `SelectionDAG` je dobio ime po načinu reprezentacije koda za vreme izbora instrukcija. Naime, osnovni blokovi programa bivaju predstavljeni usmerenim acikličkim grafovima (eng. *directed acyclic graphs*). Čvorovi tog grafa su instrukcije koje imaju tip `SDNode`, a grane predstavljaju različite zavisnosti koje postoje između instrukcija. Argumentima instrukcija se dodeljuje tip `SDValue`. Od svih čvorova u grafu posebno se izdvajaju dva i to su ulazni čvor i koren. Ulazni čvor obeležava početak osnovnog bloka i služi samo za postavljanje veza. Sa druge strane, koren označava kraj bloka. Na kraju obrade je vezan za poslednju instrukciju osnovnog bloka.

Pristup `SelectionDAG` izboru instrukcija implementaran je pomoću istoimene klase `SelectionDAG`. Gore pomenuta grafovska reprezentacija programa se gradi prolaskom kroz LLVM međukod uz pomoć klase `SelectionDAGBuilder`. Za svaku instrukciju se kreira čvor na osnovu operacije koju predstavlja. Veze se kreiraju kada neki čvor koristi izlaznu vrednost drugog čvora. Tip te vrednosti određuje tip veze. Ovako konstruisan graf nije pogodan za spuštanje instrukcija, te se nad njim moraju izvršiti određene transformacije.

Graf se optimizuje zamenom grupe povezanih čvorova jednostavnijim grupama korišćenjem algoritama za uparivanje stabala. Optimizacije nad grafom mogu biti i mašinski zavisne i mašinski nezavisne. Takođe, graf može da sadrži nepodržane tipove i operacije za ciljnu arhitekturu. Njih je potrebno zameniti odgovarajućim podržanim tipovima i operacijama i tu transformaciju nazivamo legalizacijom tipova i operacija. Način legalizacije zavisi od arhitekture i implementiran je u odgovarajućim klasama `TargetLowering`, što znači da za svaku arhitekturu postoju odgovarajuća klasa `TargetLowering`. Neki od primera klasa `TargetLowering` su `ArmTargerLowering`, `RISCVIselLowering` i `ARMIselLowering`. Optimizacije i legalizacije se vrše više puta u određenom redosledu i rezultuju grafom spremnim za spuštanje instrukcija.

Proces spuštanja instrukcija je prilično jednostavan i zasniva se na uparivanju čvorova. Čvorovi svake `SelectionDAG` strukture traže svoj pandan za ciljnu

arhitekturu. U ovom koraku se ne zamenjuju baš svi čvorovi već neki prolaze u narednu fazu prevođenja.

Poslednja stvar koju je potrebno eliminisati iz LLVM međukoda su virtuelni registri. Korak u kome se virtuelni registri zamenjuju konkretnim registrima za ciljnu arhitekturu nazivamo dodeljivanjem registara. Neki od fizičkih registara se su već implicitno dodeljeni na osnovu instrukcija. Preostali registri se dodeljuju korišćenjem heuristika, jer je problem NP-kompletan, pa za njega ne možemo koristiti optimalan algoritam. Jedna od često korišćenih heuristika je gramziva (eng. *greedy*) koja prednost prilikom dodeljivanju registara daje promenljivama sa dužim životnim vekom.

Optimizacioni prolazi postoje i na MIR nivou i način njihovog izvršavanja je sličan prolazima na IR nivou odnosno na nivou mašinski nezavisnog međukoda. Međutim, optimizacije na MIR nivou su vezane za ciljnu arhitekturu. Mašinski zavisni prolazi nasleđuju klasu `MachineFunctionPass` i implementiraju funkciju `runOnMachineFunction` koja se poziva za svaku funkciju u izvornom kodu. Prolazi se izvršavaju iterativno u unapred zadatom redosledu.

Poslednji prolaz na MIR kodu u kome se, u zavisnosti od navedenih opcija, emituje asemblerski ili objektni kôd za ciljnu arhitekturu i kojim se završava zadatak kompilatora jeste `AsmPrinter`. Ukoliko je emitovan asemblerski kôd on se prevodi do objektnog koda korišćenjem asemblera, dok se objektni kôd povezuje sa sistemskim i korisničkim bibliotekama i drugim objektnim datotekama pomoću povezivača (eng. *linker*) kako bi se od njega dobila krajnja izvršiva datoteka.

Posao zadnjeg dela LLVM kompilatora obavlja alat `llc`. Alatu se opciono zadaje vrsta izlaza koju treba da generiše i nivo optimizacije. Izlaz alata `llc` može biti asemblerski ili objektni kôd. Još jedan bitan parametar koji se alatu `llc` može proslediti je ciljna arhitektura. Ciljna arhitektura se zadaje argumentom `-mtriple` i vrednošću u vidu niske koja sadrži naziv arhitekture procesora, njegovog proizvođača, operativnog sistema i okruženja. Komanda za prevođenje LLVM međukoda do objektnog koda za arhitekturu `x86_64` i operativni sistem `Linux` je sledeća:

```
llc -filetype=obj -mtriple="x86_64-unknown-linux-gnu" input.ll -o output.o
```

2.6 LLVM međureprezentacija

Da bi kompilator mogao da prevodi više programskih jezika za više ciljnih arhitektura potrebno je da međukod bude na dovoljno apstraktnom nivou da ne

zavisi od procesorske arhitekture, ali i na dovoljno opštem nivou kako ne bi zavisio od programskog jezika u kome je izvorni kôd napisan. LLVM međukod je zapisan u SSA obliku što znači da poštuje svojstvo jedinstvenog statičkog dodeljivanja. Ovo svojstvo omogućava da se svakoj promenljivoj samo jednom može dodeliti vrednost i da svakoj upotrebi promenljive prethodi njena definicija.

Datoteke koje u sebi sadrže LLVM međukod se najčešće obeležavaju ekstenzijom `.ll`. Na početku takvih datoteka se nalazi ime datoteke u kojoj je napisan izvorni kôd (npr. `test.c`), informacije o načinu zapisa podataka i naziv ciljne arhitekture. U nastavku se nalaze globalne promenljive i funkcije čiji nazivi počinju prefiksom '@'.

LLVM međureprezentacija (LLVM IR) je u okviru projekta LLVM implementirana kroz hijerarhiju klasa. Svaka od klasa u toj hijerarhiji predstavlja jedan od sledećih entiteta: modul, funkciju, osnovni blok i instrukciju. U nastavku će svaki od entiteta biti pojedinačno objašnjen.

Moduli su najviši entiteti u hijerarhiji i definisani su sadržajem datoteka u kojima se nalazi LLVM međureprezentacija (LLVM međukod). Funkcije izgrađuju jedan modul i odgovaraju funkcijama napisanim u izvornom kodu. Svaka funkcija se sastoji od niza osnovnih blokova, a svaki blok od niza instrukcija. Lokalne promenljive funkcija su predstavljene virtuelnim registrima. Identifikatori virtuelnih registara počinju karakterom '%'. Zahvaljujući SSA obliku, svakom od virtuelnih registara se vrednost može dodeliti tačno jednom, tako da virtuelnih registara može biti neograničeno mnogo.

Osnovni blokovi (eng. *basic blocks*) predstavljaju niz instrukcija koji se uvek izvršavaju linearno i u celini. Svaka funkcija započinje prvim osnovnim blokom i završava se poslednjim osnovnim blokom. Ukoliko se nekim programskim tokom uđe u osnovni blok on će se uvek izvršiti do kraja. To implicira da se instrukcije grananja pojavljuju isključivo na krajevima osnovnih blokova. Granice osnovnih blokova su označene poput labela u asemblerskom jeziku.

Na kraju, instrukcije izgrađuju osnovne blokove i delom odgovaraju instrukcijama u izvornom kodu. Instrukcije LLVM međukoda mogu biti dvoadresne ili troadresne što podrazumeva da svaka instrukcija ima jedan ili dva argumenta koji predstavljaju operande i dodatni argument koji predstavlja lokaciju za smeštanje rezultata. Instrukcije se sastoje od naziva (`alloca`, `store`, `zext`, ...), tipa (`i8`, `i16`, `float`, ...) i operanada. Dodatne informacije o instrukcijama i globalnim objektima se čuvaju u vidu metapodataka. Metapodaci počinju karakterom '!' i obično se

```c
#include <stdio.h>

int main() {
    int x;
    scanf("%d", &x);
    if (x > 0)
        printf("Welcome to the LLVM IR!");

    return 0;
}
```

Listing 1: Kratak C program na osnovu koga kreiramo program sa LLVM međukodom

koriste kao informacije za debagovanje. Primer LLVM međukoda dobijenog od kratkog programa napisanog u programskom jeziku C prikazan je u listingu 2, dok je program od koga je LLVM međukod dobijen prikazan u listingu 1.

Novitet koji se uvodi na nivou LLVM međureprezentacije, a koji ne postoji na nivou izvornog koda, jeste intrinzička funkcija. Intrinzička funkcija (eng. *intrinsic function*) predstavlja funkciju koja se kroz veći deo prevođenja LLVM kompilatorom (eng. *LLVM pipeline*) može smatrati validnom iako nikakav kôd ne postoji iza nje, ali koja će u određenjom trenutku biti zamanjena blokom koda koji bi trebalo da predstavlja. Da bi neka funkcija mogla biti smatrana intrinzičkom funkcijom, potrebno je istu deklarisati u okviru posebnih datoteka projekta LLVM. To su datoteke sa ekstenzijom `.td` čiji je sadržaj napisan jezikom `TableGen`.

`TableGen` predstavlja domenski specifičan jezik (eng. *domain specific language*) razvijen za potrebe LLVM projekta koji je zadužen za generisanje različitih datoteka ili struktura podataka u situacijama kada je manuelno kreiranje i održavanje istih dosta zahtevno. `TableGen` datoteke pokrivaju značajan procenat celokupnog koda u okviru LLVM projekta. Pozovom alata `cloc` (eng. *count lines of code*) koji je u stanju da prebroji linije u datoteci i kategorizuje ih prema programskom jeziku dobija se da je više od 500 000 linija koda projekta LLVM napisano upravo u sintaksi `TableGen`.

2.7 LLVM infrastruktura za testiranje

Za testiranje promena napravljenih u okviru projekta LLVM kreirana je čitava infrastruktura za testiranje. LLVM infrastruktura za testiranje omogućava kori-

```
; ModuleID = 'test.c'
source_filename = "test.c"
target datalayout =
↪  "e-m:e-p270:32:32-p271:32:32-p272:64:64-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-pc-linux-gnu"

@.str = private unnamed_addr constant [3 x i8] c"%d\00", align 1
@.str.1 = private unnamed_addr constant [24 x i8] c"Welcome to the LLVM IR!\00",
↪  align 1

; Function Attrs: noinline nounwind optnone uwtable
define dso_local i32 @main() #0 {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  store i32 0, i32* %1, align 4
  %3 = call i32 (i8*, ...) @__isoc99_scanf(i8* noundef getelementptr inbounds ([3
  ↪  x i8], [3 x i8]* @.str, i64 0, i64 0), i32* noundef %2)
  %4 = load i32, i32* %2, align 4
  %5 = icmp sgt i32 %4, 0
  br i1 %5, label %6, label %8

6:                                                ; preds = %0
  %7 = call i32 (i8*, ...) @printf(i8* noundef getelementptr inbounds ([24 x i8],
  ↪  [24 x i8]* @.str.1, i64 0, i64 0))
  br label %8

8:                                                ; preds = %6, %0
  ret i32 0
}

declare i32 @__isoc99_scanf(i8* noundef, ...) #1

declare i32 @printf(i8* noundef, ...) #1

attributes #0 = { noinline nounwind optnone uwtable "frame-pointer"="all"
↪  "min-legal-vector-width"="0" "no-trapping-math"="true"
↪  "stack-protector-buffer-size"="8" "target-cpu"="x86-64"
↪  "target-features"="+cx8,+fxsr,+mmx,+sse,+sse2,+x87" "tune-cpu"="generic" }
attributes #1 = { "frame-pointer"="all" "no-trapping-math"="true"
↪  "stack-protector-buffer-size"="8" "target-cpu"="x86-64"
↪  "target-features"="+cx8,+fxsr,+mmx,+sse,+sse2,+x87" "tune-cpu"="generic" }

!llvm.module.flags = !{!0, !1, !2, !3, !4}
!llvm.ident = !{!5}

!0 = !{i32 1, !"wchar_size", i32 4}
!1 = !{i32 7, !"PIC Level", i32 2}
!2 = !{i32 7, !"PIE Level", i32 2}
!3 = !{i32 7, !"uwtable", i32 1}
!4 = !{i32 7, !"frame-pointer", i32 2}
!5 = !{!"Ubuntu clang version 14.0.0-1ubuntu1.1"}
```

Listing 2: Primer LLVM međureprezentacije (LLVM međukoda)

šćenje testova jedinica koda (eng. *unit tests*) kao i čitavih programa za testiranje. Dodatno, posebna vrsta testova su i regresioni testovi koji služe da obezbede da promene napravljene u LLVM projektu neće izazvati defekte u njegovoj funkcionalnosti. Testovi jedinica koda i regresioni testovi su sadržani u LLVM-ovom repozitorijumu. Očekuje se da se u svakom trenutku svaki od ovih testova izvršava sa očekivanim ishodom i preporuka je pokrenuti ih pre i nakon kreiranja promena na projektu.

Testovi jedinica koda su napisani korišćenjem *Google* testova i *Google Mock*-a i nalaze se u direktorijumu `llvm/unittests`. Testovi jedinica koda se koriste za proveru korišćenja biblioteka i za druge generičke strukture podataka. Ukoliko je potrebno testirati transformacije i analize sprovedene na nivou međureprezentacije, odnosno IR nivou, u tom slučaju je bolje koristiti regresione testove.

Čitavi programi za testiranje se nazivaju „LLVM test suite" ili samo „test-suite" i za njih postoji repozitorijum na platformi `GitHub`. Ovi testovi sadrže delove koda koji predstavljaju čitave programe i koji se mogu sastaviti i povezati u samostalan program koji se može izvršiti. Programi se obično pišu u jezicima visokog nivoa kao što su C ili C++. Programi za testiranje se prevode korišćenjem odabranih parametara kompilatora, a zatim se izvršavaju kako bi se uhvatile informacije o njegovom izlazu i vremenu izvršavanja. Izlaz programa za testiranje se poredi sa referentnim izlazom. Za više detalja pogledajte *test-suite* vodič [24].

Regresioni testovi

Regresioni testovi su mali delovi koda koji testiraju specifičnu komponentu LLVM kompilatora ili izazivaju određenu grešku u okviru LLVM-a. Jezik u kome su napisani zavisi od dela LLVM-a koji se tim testom proverava. Kada se pronađe neka greška u okviru projekta LLVM, obično se na osnovu nje kreiraju regresioni testovi koji sadrže taman dovoljno koda da reprodukuju problem. Na primer, regresioni test može biti mali deo LLVM IR koda izdvojen iz neke aplikacije. Regresioni testovi se nalaze u okviru direktorijuma `llvm/test`. Promene opisane u ovom radu biće testirane kreiranjem regresionih testova, a potom pokretanjem alata `llvm-lit` [19] i `Filecheck` [18] nad njima.

Alat `llvm-lit` je deo projekta LLVM. To je alat za izvršavanje LLVM i Clang programa za testiranje, sumiranje njihovih rezultata i saopštavanje potencijalnih grešaka. Alat je dizajniran tako da bude lak za testiranje (u smislu memorijskog zauzeća i minimalističke sintakse) sa što jednostavnijim korisničkim interfejsom. Alat je moguće pokrenuti nad jednim ili više testova navedenih u komandnoj liniji. Testovi mogu biti ili pojedinačne test datoteke ili direktorijumi sa test datotekama. Svaki navedeni test se izvršava, nekada i konkurentno. Kada su svi testovi pokrenuti `llvm-lit` počinje sa štampanjem informacija o broju testova koji su uspešno završili sa radom i onima kod kojih se desila greška.

Alat `Filecheck` čita dve datoteke (jednu sa standardnog ulaza, a jednu navedenu kao argument komandne linije) i koristi jednu kako bi verifikovao drugu. Ovakvo ponašanje je izuzetno korisno za pokretanje skupova testova, kojima se proverava da li rezultat rada nekog alata (na primer, alata `llc`) sadrži neku očekivanu informaciju (na primer, instrukciju `clmul` u asemblerskom kodu koji alat generiše). Deluje da je alat `Filecheck` vrlo sličan alatu `grep`, međutim alat `Filecheck` omogućava daleko efikasnije prepoznavanje više različitih ulaza u jednoj datoteci i izlistavanje prepoznatih delova u specifičnom redosledu. Uputstvo za korišćenje alata `Filecheck` prikazano je u listingu 3. Datoteka `match-filename` predstavlja datoteku koja sadrži obrazac koji treba prepoznati. Datoteka koja se verifkuje se unosi sa standardnog ulaza ukoliko opcija `-input-file` nije upotrebljena.

```
FileCheck match-filename [-check-prefix=XXX] [-strict-whitespace]
```

Listing 3: Uputstvo za pokretanje alata `FileCheck`

Glava 3

Algoritam CRC i problem njegovog prepoznavanja

Sa porastom protoka podataka kroz različite mrežne kanale, greške u podacima odnosno narušavanje njihovog integriteta je postalo sve učestalije. Usled unutrašnjih ili spoljašnjih smetnji, podaci koji se prenose često postaju korumpirani ili oštećeni. To dovodi do gubitka podataka. Jedna od najpoznatijih metoda za otkrivanje oštećenja u podacima je ciklična provera redundansi (eng. *Cyclic Redundancy Check - CRC*), odnosno algoritam CRC.

Primene algoritma CRC obuhvataju različite aspekte digitalne komunikacije. Na primer, u mrežnim protokolima kao što je `Ethernet` [7] algoritam CRC se koristi za otkrivanje grešaka prilikom prenosa podataka. U bežičnoj komunikaciji, na primer tehnologiji `Bluetooth` [42], algoritam CRC se primenjuje kako bi se osigurala tačnost i pouzdanost prenosa podataka između uređaja. Takođe, u sistemu za čuvanje podataka kao što je `RAID` (eng. *Redundant Array of Independent Disks*) [46], CRC može otkriti i ispraviti greške na nivou diska.

3.1 Algoritam CRC

Ciklična provera redundansi predstavlja tehniku koja omogućava da se putem dodatnih bitova detektuju promene u podacima. Ova tehnika se u velikoj meri koristi u digitalnim mrežama i u uređajima za skladištenje. Podaci koji ulaze u ove sisteme se proširuju dodatnim bitovima (dodatni bitovi se nadovezuju na bitove podataka). Dodatne bitove nazivamo kontrolnim bitovima (eng. *checksum*) i oni se dobijaju kao ostatak pri deljenju bitova podataka sa odabranim polinomom. De-

ljenje se izvodi u binarnoj osnovi. Po prijemu podataka, izračunavanje se ponavlja (deljenje bitova podataka odabranim polinomom) i ukoliko se dobijena vrednost ne poklapa se očekivanom vrednošću, zaključuje se da je došlo do narušavanja integriteta podataka i mogu se pokrenuti akcije za ispravljanje oštećenih delova podataka.

Algoritam CRC se tako naziva jer su bitovi kojima se podaci proširuju redundantni (njima se podaci samo proširuju), a sam algoritam se zasniva na cikličnim kodovima [47]. Algoritam CRC je popularan jer se može implementirati u samom hardveru, lako se matematički analizira i naročito dobro detektuje greške nastale usled šuma, a unutar transportnih kanala.

Specifikacija CRC koda zahteva i definiciju polinoma generatora. Polinom generator postaje delilac u polinomskom deljenju, a poruka se uzima kao deljenik. Količnik pri deljenju se odbacuje, a bitovi ostatka se uzimaju kao kontrolni bitovi. Bitno je naglasiti da se koeficijenti polinoma računaju na osnovu aritmetike u konačnom polju. U praksi, sve često korišćene verzije algortima CRC koriste konačno polje sa dva elementa (odnosno Galoavu grupu GF(2)).

Najjednostavniji metod za detekciju grešaka u podacima jeste bit parnosti, koji predstavlja kontrolni CRC bit dužine jedan. Bit parnosti koristi polinom $x + 1$ (dva člana) i obično se označava kao `CRC-1`. Ovaj kontrolni bit ima vrednost jedan ukoliko je broj jedinica u bitovima podataka paran. U suprotnom, ovaj kontrolni bit ima vrednost 0. Primalac poruke proširene bitom parnosti proverava da li vrednost bita parnosti odgovara parnosti bitova poruke. Ukoliko dođe do promene u podacima koja se ne odrazi na parnost bitova, ta promena neće biti detektovana ovom metodom.

Sledeći jednostavan metod za detekciju grešaka jeste izračunavanje sume bitova podataka. Bitovi sume se nadovezuju na bitove podataka (odnosno dodeljuju podacima u vidu kontrolnih bitova) i podaci se u tom obliku šalju. Primalac poruke (odnosno podatka) može po prijemu poruke izračunati njenu sumu bitova i tu vrednost uporediti sa vrednošću predstavljenom kontrolnim bitovima. Ukoliko se ove dve sume razlikuju to bi trebalo da signalizira da je usled transfera podatka došlo do narušavanja njegovog integriteta. U suprotnom, ukoliko su sume identične, to bi značilo da do narušavanja integriteta podataka nije došlo. Međutim, ukoliko su bitovi podataka promenjeni, ali tako da je suma njihovih bitova ostala ista, upoređivanjem sume bitova poruke i kontrolnih bitovima dolazi se do zaključka da se sa primljenom porukom ništa nije desilo iako zapravo jeste. Takođe, ukoliko je

neki od kontrolnih bitova promenjen, dolazi se do zaključka da je integritet poruke narušen iako zapravo nije.

Navedeni primeri govore o tome da je izbor polinoma generatora zapravo najvažniji deo implementacije algoritma CRC. Polinom mora biti izabran tako da se maksimizuju mogućnosti otkrivanja grešaka uz minimizovanje ukupnih verovatnoća kolizije. Pod kolizijom se podrazumevaju opisane situacije u kojima promena u podacima nije detektovana (a desila se) i situacije u kojima se promena signalizira iako se nije stvarno desila. Najvažniji atribut polinoma jeste njegov najveći stepen zbog njegovog direktnog uticaja na dužinu kontrolnih bitova.

Kontrolni bitovi dužine n nazivaju se n-bitnim CRC-om. Za dato n moguće je generisati više različitih kontrolnih bitova (svaki za različiti polinom). Polinom sa najvišim stepenom n ima ukupno $n+1$ članova, samim tim i bitovsku reprezentaciju dužine $n+1$. Najčešće korišćeni polinomi su polinomi dužine 9 bita (`CRC-8`), 17 bita (`CRC-16`), 33 bita (`CRC-32`) i 65 bita (`CRC-64`). Na primer, često korišćeni polinom u implementaciji algoritma `CRC-8` jeste $x^8 + x^7 + x^6 + x^4 + x^2 + 1$, dok je u slučaju algoritma `CRC-16` to polinom $x^{16} + x^{15} + x^2 + 1$. Različite konfiguracije polinoma generatora omogućavaju prilagođavanje algoritma CRC specifičnim zahtevima sistema.

Pojednostavljen primer računanja CRC kontrolnih bitova na strani pošiljaoca i provere integriteta podataka na strani primaoca poruke prikazan je u listinzima 4 i 5. U primeru se 14-bitna poruka proširuje sa tri 0 bita (inicijalnim kontrolnim bitovima), a kao polinom generator koristi se polinom x^3+x+1. U listingu 4 prikazano je računanje kontrolnih bitova na strani pošiljaoca poruke. Kontrolni bitovi se računaju kao ostatak pri deljenju bitova poruke sa bitovima polinoma. Deljenje se sprovodi primenom ekskluzivne disjunkcije (operacije XOR) i pomeranja bitova polinoma za po jedan bit udesno u svakoj iteraciji. Ostatak dobijen na kraju deljenja predstavlja kontrolne bitove kojima će poruka biti produžena i u tom obliku poslata. U listingu 5 prikazana je provera ispravnosti podataka na strani primaoca poruke. Ispravnost primljene poruke se proverava ponovnim izvođenjem deljenja, ovog puta nad bitovima primljene poruke (bitovi poruke i kontrolni bitovi dobijeni iz prethodnog deljenja). Ukoliko je ostatak pri deljenju jednak 0 (odnosno jednak inicijalnim kontrolnim bitovima) zaključuje se da do promene podataka nije došlo. U suprotnom se zaključuje da je do promene podataka došlo.

Prikazani algoritam može biti prilično neefikasan i u najgorem slučaju obrađivati bit po bit. Za veće ulazne podatke, to bi bilo prilično sporo. Međutim, moguće

```
11010011101100 000 <--- bitovi podataka prošireni sa tri inicijalna
↪  kontrolna bita
1011               <--- delilac (polinom generator x^3 + x + 1)
01100011101100 000 <--- rezultat
 1011              <--- delilac ...
00111011101100 000
  1011
00010111101100 000
   1011
00000001101100 000 <--- deljenje se nastavlja od prvog sledećeg bita 1
       1011
00000000110100 000
        1011
00000000011000 000
         1011
00000000001110 000
          1011
00000000000101 000
           101 1
-----------------
00000000000000 100 <--- ostatak (poslednja 3 bita).
                   Deljenje se zaustavlja pošto je količnik postao 0.
```

Listing 4: Računanje kontrolnih bitova na strani pošiljaoca

```
11010011101100 100 <--- bitovi podataka prošireni kontrolnim bitovima
1011               <--- delilac (polinom generator x^3 + x + 1)
01100011101100 100 <--- rezultat
 1011              <--- delilac ...
00111011101100 100

......

00000000001110 100
          1011
00000000000101 100
           101 1
------------------
00000000000000 000 <--- ostatak
```

Listing 5: Provera integriteta podataka na strani primaoca poruke

je unapred izračunati rezultate deljenja za svaku moguću vrednost bloka podataka koji se deli polinomom i sačuvati ih u pomoćni niz ili tabelu. Sačuvane rezultate bi potom bilo moguće koristiti čime bi se ubrzala obrada. U slučaju polinoma stepena

8 blok podataka koji se deli polinomom je dužine 8 bita (jednog bajta). To znači da bi pomoćni niz čuvao ukupno 256 vrednosti (za svaku vrednost bajta čuvao bi se po jedan rezultat deljenja). Verzije algoritma CRC koje koriste pomoćne nizove odnosno tabele nazivaju se *table-based* verzije algoritma CRC.

3.2 Problem prepoznavanja algoritma CRC

Problem koji se u ovom radu rešava jeste prepoznavanje (detektovanje) algoritma CRC i njegovo optimizovanje. Pod prepoznavanjem se podrazumeva pronalaženje njegove implementacije u izvornom kodu nekog programa. Da bi prepoznavanje algoritma CRC bilo izvodiljivo potrebno je da njegova implementacija bude unapred poznata. Prepoznavanje se može izvršiti prolaskom kroz kôd programa (krećući se od prve ka poslednjoj instrukciji ili u obrnutom smeru). Ovim postupkom se vrši upoređivanje redosleda instrukcija programa sa redosledom instrukcija algoritma CRC. Takođe se proverava da li svaka od instrukcija programa sadrži argumente istog tipa i iste vrednosti kao odgovarajuća instrukcija u algoritmu CRC. Različit redosled nezavisnih instrukcija programa ne bi trebalo da utiče na rezultat prepoznavanja. Korišćenje različitih, funkcionalno ekvivalentnih instrukcija u programu takođe ne bi trebalo da utiče na rezultat prepoznavanja. Na kraju ovakvog postupka, ukoliko ni u jednoj od spomenutih provera nije bilo razlike između očekivanih i dobijenih rezultata, može se zaključiti da je algoritam prepoznat (odnosno detektovan). Tek tada se mogu pokrenuti akcije za njegovo optimizovanje.

Problem sa prepoznavanjem algoritma CRC jeste taj što postoji veliki broj njegovih implementacija, pa je za svaku od njih potrebno napraviti poseban šablon za prepoznavanje (eng. *pattern matcher*). Pod šablonom za prepoznavanje se podrazumevaju sve provere koje se nad jednim programom vrše (prolazak kroz njegove instrukcije, provera njihovog redosleda, vrednosti njihovih operanada i druge) kako bi se ustanovilo da li on u sebi sadrži implementaciju nekog algoritma. Razvijanje svakog od šablona zahteva dosta vremena i truda kako bi se napravio šablon sposoban da prepozna što veći broj modifikacija iste verzije algoritma. Sa druge strane, prepoznavanje jednostavnijih algoritama ili aritmetičkih izraza (proširenog oblika kvadrata binoma, kuba binoma, izraza za računanje rešenja kvadratne jednačine) je daleko jednostavnije i zahteva pokrivanje znatno manjeg broja slučajeva [1].

[1] U okviru `GitHub` repozitorijuma koji sadrži sav materijal ovog rada nalazi se i implementacija

Kako se usled porasta broja informacija koje se prosleđuju putem interneta, povećava upotreba algoritma CRC i drugih metoda za proveru integriteta podataka tako raste potreba da svaka od korišćenih metoda bude što efikasnija. Jedan od načina da se obezbedi efikasnost ovih algoritama jeste upravo uvođenje dodatnih provera u prevodiocima kojima bi se ovi algoritmi detektovali i potom optimizovali.

LLVM optimizacionog prolaza pod nazivom `expression-optimizer`. Svrha tog optimizacionog prolaza jeste detektovanje proširenog oblika kvadrata binoma ($a^2 + 2ab + b^2$) i zamena istog odgovarajućim skraćenim zapisom ($(a + b)^2$).

Glava 4

Procesorske arhitekture i arhitektura RISC-V

Softver komunicira sa hardverom korišćenjem skupa instrukcija (eng. *Instruction Set Architecture - ISA*). Neki od primera instrukcija koji mogu biti sadržane u tom skupu su `add`, `sub` i `mul`. Te instrukcije predstavljaju operacije koji je procesor u stanju da izvrši. Programi se obično sastoje od miliona instrukcija koje se potom velikom brzinom izvršavaju na procesoru.

4.1 Procesorske arhitekture RISC i CISC

U savremenoj softverskoj industriji postoje dva pristupa razvoju skupa instrukcija za procesorke arhitekture. U pitanju su pristup zasnovan na redukovanom skupu instrukcija koji rezultuje RISC arhitekturama procesora i pristup zasnovan na kompleksnom skupu instrukcija koji rezultuje CISC arhitekturama procesora.

RISC (eng. *Reduced Instruction Set Computer*) je naziv za arhitekture čiji se skup instrukcija sastoji iz relativno malog broja instrukcija koje obavljaju samo jednostavne operacije. Najčešće podržane aritmetičke i logičke instrukcije su instrukcije sabiranja i oduzimanja, instrukcije množenja, instrukcije pomeranja i bitovskih operacija. Složenije aritmetičke operacije se moraju implementirati softverski svođenjem na jednostavnije. Za instrukcije ove arhitekture je karakteristično da imaju uniforman format, kao i jednostavne načine adresiranja operanada. RISC arhitekture se najčešće realizuju kao `load-store` arhitekture. To znači da se podržane instrukcije mogu pode-

liti u dve kategorije: one koje vrše pristup memoriji i one koje obavljaju aritmetičko-logičke operacije. Za pristup memoriji postoje posebne instrukcije koje podatke iz memorije upisuju u registre, kao i one koje vrednosti registara upisuju u memoriju. Sve ostale instrukcije rade isključivo sa registarskim (ili neposrednim) operandima. Tipični predstavnici RISC arhitekture su ARM [36], MIPS [41] i RISC-V [9] procesori.

CISC (eng. *Complex Instruction Set Computer*) je naziv za arhitekture čiji skup instrukcija može sadržati i instrukcije koje obavljaju prilično složene operacije poput sinusa, kosinusa, korena, logaritma i sličnih. Ove operacije zahtevaju izračunavanje u velikom broju ciklusa unutar samog procesora, svođenjem na jednostavnije koje aritmetičko-logička jedinica direktno podržava. Zbog toga je organizacija ovakvih procesora mnogo složenija. Za razliku od RISC procesora, CISC procesori podržavaju različite formate instrukcija kao i veći broj načina adresiranja operanada. To znači da se, u zavisnosti od tipa operanada, ista instrukcija može pojaviti u više oblika. Aritmetičke istrukcije CISC procesora imaju mogućnost da koriste podatke direktno iz memorije, tj. instrukcije mogu imati i registarske i memorijske operande. Tipični predstavnici CISC arhitekture su Intel procesori [10].

U slučaju RISC arhitektura programer je primoran da pomoću jednostavnih operacija koje procesor podržava realizuje komplikovane algoritme, što njegove programe čini dužim. Kod CISC arhitektura su, sa druge strane, programi obično kraći, jer za mnoge složene operacije postoje instrukcije koje procesor direktno podržava. Aritmetičke instrukcije CISC arhitektura mogu pored registarskih imati i memorijske operande. To omogućava da se podaci iz memorije mogu direktno koristiti. CISC arhitekture su bile veoma popularne u periodu kada je kapacitet memorija bio mali. U to vreme je postojao veliki jaz između brzine procesora i brzine memorije, pa su memorijski transferi bili skupi. Zbog toga je bilo potrebno imati što manji broj instrukcija u programu (odnosno što kraće programe) koje bi obavljale složene operacije.

Sa razvojem tehnologije pomenuti problemi se postepeno gube, te RISC arhitekture postaju popularnije. Memorije imaju veći kapacitet, memorijski transferi postaju brži zahvaljujući efikasnim keš memorijama, a programski prevodioci napreduju u vidu optimizacije koda. Time mogućnosti hardvera više ne zaostaju za mogućnostima softvera. Takođe, moderni procesori imaju sve veći broj registara,

što je neophodno u realizaciji `load-store` arhitektura odnosno RISC procesora. Jednostavnost RISC skupova instrukcija omogućava jednostavnu implementaciju procesora, a to izvršavanje instrukcija čini veoma brzim. Sa druge strane, dizajn CISC procesora je dosta kompleksniji, pa se čak i jednostavne instrukcije na njima izvršavaju sporije nego na RISC procesorima. Budući da se u programima jednostavne instrukcije daleko više koriste, efikasnost RISC arhitektura dolazi do izražaja u većini praktičnih primena.

4.2 Arhitektura RISC-V

Projekat RISC-V je u maju 2010. godine započeo profesor Krste Asanović sa nekolicinom svojih studenata u laboratoriji za paralelno izračunavanje na Univerzitetu Berkli. U ovoj laboratoriji je takođe razvijen i `Chisel` [8], jezik za konstrukciju hardvera, koji je korišćen za dizajn mnogih RISC-V procesora.

Projektom RISC-V upravlja neprofitna organizacija `RISC-V International` sa trenutnim sedištem u Švajcarskoj [32]. Danas postoje članovi u preko 70 zemalja koji doprinose i sarađuju na definisanju RISC-V specifikacija. Projekat je otvorenog koda i koristi licencu `Berkley Software Distribution` (skraćeno `BSD`). Prva publikacija u kojoj je opisan skup instrukcija RISC-V je objavljena 2011. godine [6].

RISC-V ISA [33] predstavlja skup instrukcija otvorenog koda koja pruža osnovu za dizajn RISC-V procesora. Ovaj skup instrukcija predstavlja modularan i proširiv skup koji se može prilagoditi specifičnim aplikacijama i slučajevima upotrebe. Poput većine RISC arhitektura i skup instrukcija RISC-V je osmišljen kao `load-store` arhitektura. Njegove instrukcije u pokretnom zarezu koriste standard IEEE 754. Značajne karakteristike skupa instrukcija RISC-V su: lokacija bitova instrukcija izabrana tako da pojednostave upotrebu multipleksora u procesoru, dizajn koji je arhitektonski neutralan i fiksna lokacija za znakovni bit konstanti kojom se ubrzava postupak proširenja znaka.

Osnovni skup instrukcija RISC-V sadrži instrukcije fiksne dužine od 32 bita, a takođe podržava i 16-bitne instrukcije koje koriste ugrađeni sistemi, neki personalni računari, superkompjuteri sa vektorskim procesorima i drugi računarski sistemi. Instrukcije se navode u redosledu `little-endian`[1]. Specifikacija skupa

[1] `Little-endian` je sistem uređenja bajtova koji se koristi u arhitekturi računara za definisanje rasporeda višebajtnih vrednosti podataka u memoriji. U ovom sistemu bajt koji nosi bitove naj-

instrukcija RISC-V definiše 32-bitne i 64-bitne verzije adresnog prostora. Specifikacija uključuje i opis verzije 128-bitnog adresnog prostora, kao ekstrapolaciju 32-bitnih i 64-bitnih adresnih prostora.

Dizajn procesora zahteva stručnost u oblasti elektronske digitalne logike, kompilatora i operativnih sistema. Da bi pokrili troškove timova koji se bave dizajnom procesora, komercijalni prodavci intelektualne svojine procesora, kao što su ARM i MIPS, naplaćuju autorske naknade za korišćenje njihovih dizajna i patenata. Oni takođe često zahtevaju sporazume o neotkrivanju detalja o prednostima njihovog dizajna. U mnogim slučajevima, oni ne opisuju razloge svojih izbora u dizajnu.

Projekat RISC-V je započet sa ciljem da se napravi praktičan skup instrukcija koji će biti otvorenog koda, upotrebljiv u akademske svrhe i primenljiv u bilo kom dizajnu hardvera ili softvera bez nadoknadi. Takođe, razlozi za svaku odluku o dizajnu projekta su navedeni, barem u širem smislu. Autori RISC-V su akademici koji imaju značajno iskustvo u dizajnu računarskih sistema, a skup instrukcija RISC-V je i nastao iz serije akademskih projekata na temu dizajna računara. Jednostavnost skupa instrukcija RISC-V omogućava korišćenje softvera za kontrolu istraživačkih mašina, čime se podstiče upotreba RISC-V u akademske svrhe. Skup instrukcija promenljive dužine pruža prostor za ekstenzije sa različitim primenama (za studentske vežbe, za istraživanje).

Skup instrukcija RISC-V ima modularan dizajn, koji se sastoji od alternativnih osnovnih delova, sa dodatnim opcionim ekstenzijama. Osnova skupa instrukcija RISC-V i njene ekstenzije razvijeni su u zajedničkom naporu između industrije, istraživačke zajednice i obrazovnih institucija. Osnova skupa instrukcija određuje instrukcije (i njihovo kodiranje), kontrolu toka (eng. `control flow`), registre i njihove veličine, memoriju i adresiranje, kao i druge elemente. Sama osnova skupa instrukcija RISC-V dovoljna je za implementaciju pojednostavljenog računara opšte namene, sa punom softverskom podrškom (uključujući i kompilator opšte namene). Standardne ekstenzije su specifikovane tako da rade sa svim standardnim osnovama i to bez ikakvog konflikta. Mnogi RISC-V računari bi mogli da implementiraju ekstenzije sa kompresovanim instrukcijama kojima bi smanjili potrošnju energije, veličinu koda i upotrebu memorije.

manje težine se postavlja na najnižu memorijsku adresu. Sa druge strane, `big-endian` je sistem uređenja u kome se bitovi najveće težine smeštaju na najniže memorijske adrese.

Ekstenzije arhitekture RISC-V

U novembru 2021. su u skup instrukcija RISC-V uvedene sledeće ekstenzije: Zba, Zbb, Zbc, Zbs [16]. Ekstenzije Zba, Zbb i Zbs predstavljaju proširenja standardnih instrukcija koje rade sa celim brojevima. Ekstenzija Zbb nudi instrukcije za brojanje vodećih ili završnih 0 bitova ili svih bitova koji imaju vrednost 1. Ekstenzija Zbs omogućava postavljanje, dohvatanje, brisanje i prebacivanje pojedinačnih bitova u registru prema njihovom indeksu.

Ekstenzija Zbc sadrži instrukcije za „množenje bez prenosa" kojim se vrši množenje binarnih polinoma u Galoavom polju GF(2). U pitanju su instrukcije `clmul`, `clmulh`, `clmulr`. Instrukcija `clmul` (eng. `carry-less multiplication`) izvršava operaciju množenja bez generisanja i propagiranja prenosa. Instrukcija prihvata dva celobrojna argumenta iste širine i vraća ceo broj sa dvostrukom širinom argumenata. Mehanički je ostvarena kao množenje korišćenjem višestruke ekskluzivne disjunkcije (operacije `XOR`) umesto višestrukog sabiranja (operacije `ADD`), dok matematički predstavlja množenje dva binarna polinoma. Slično, instrukcija `clmulh` računa donju polovinu proizvoda bez prenosa, a instrukcija `clmulr` radi isto što i instrukcija `clmul` samo nad obrnutim redosledom bitova njenih argumenata (`clmul(A, B) = rev(clmul(rev(A), rev(B)))`).

Neke od primena proizvoda bez prenosa odnosno operacije `clmul` mogu biti kriptografija, heširanje, Mortonovo kodiranje, računanje Grejovog koda, implementacija algoritma CRC i druge. Slične instrukcije su podržane i u `x86`, `SPARC` i drugim arhitekturama. Glavne prednosti korišćenja instrukcije `clmul` u implementaciji algoritma CRC su: poboljšane performanse, značajno smanjena potrošnja. Nedostatak jeste to što je instrukcija primeljiva samo nad 32-bitnim CRC vrednostima.

Glava 5

Implementacija i evaluacija rešenja

U ovom poglavlju biće predstavljena rešenja problema prepoznavanja neoptimizovane verzije algoritma CRC i njegovog zamenjivanja optimizovanom verzijom algoritma. Implementacije rešenja su urađene na verziji 18 projekta LLVM i javno su dostupne na platformi *GitHub* na sledećem linku: `https://github.com/PosteruOle/master_thesis`. Komande za preuzimanje i kompilaciju projekta LLVM navedene su u listingu 6.

```
$ git clone git@github.com:PosteruOle/llvm-project.git
$ cd llvm-project
$ git checkout riscv_crc
$ mkdir build && cd build
$ cmake -G Ninja -DCMAKE_BUILD_TYPE=Release -DLLVM_ENABLE_ASSERTIONS=ON
↪  ../llvm
$ ninja
```

Listing 6: Komande za preuzimanje i prevođenje kompilatora LLVM

Implementacije su urađene na paru algoritama CRC (paru koji čine neoptimizovana i odgovarajuća optimizovana verzija), a isti postupak je moguće primeniti i na drugim parovima funkcionalno ekvivalentnih algoritama. Neoptimizovana verzija algoritma CRC je prikazana u listingu 7, dok je optimizovana verzija algoritma CRC prikazana u listingu 8. Pozivom prednjeg dela LLVM kompilatora odnosno alata Clang nad neoptimizovanom i optimizovanom verzijom algoritma CRC dobijaju se odgovarajuće LLVM međureprezentacije prikazane u listinzima 9 i 10.

```
unsigned short crcu8(unsigned char data, unsigned short crc) {
    unsigned char i = 0, x16 = 0, carry = 0;
    for (i = 0; i < 8; i++) {
       x16 = (unsigned char)((data & 1) ^ ((unsigned char)crc & 1));
       data >>= 1;
       if (x16 == 1) {
         crc ^= 0x4002;
         carry = 1;
       } else {
         carry = 0;
       }
       crc >>= 1;
       if (carry)
         crc |= 0x8000;
       else
         crc &= 0x7fff;
    }
    return crc;
}
```

Listing 7: Primer neoptimizovanog algoritma CRC

```
unsigned short crcu8_optimized(unsigned char data, unsigned short _crc)  {
    unsigned char i = 0, x16 = 0, carry = 0;
    long crc = _crc;
    crc ^= data;
    for (i = 0; i < 8; i++) {
      x16 = (unsigned char)crc & 1;
      data >>= 1;
      crc >>= 1;
      crc ^= (x16 & 1) ? 0xa001 : 0; // Conditional XOR
    }
    return crc;
}
```

Listing 8: Primer optimizovanog algoritma CRC

Obe verzije algoritma generišu kontrolne bitove za 8-bitni podatak na osnovu inicijalnih kontrolnih bitova koji se prosleđuju kao argument funkcije. Funkcija `crc8` (listing 7) prihvata ulazni podatak i početnu 16-bitnu vrednost CRC-a, zatim obrađuje svaki bit ulaznog podatka, primenjujući polinom predstavljen bitovima broja `0x4002` i na kraju vraća ažuriranu CRC vrednost. Ažurirana vrednost se potom može koristiti za detekciju grešaka u podacima.

Funkcija `crcu8_optimized` (listing 8) donosi nekoliko poboljšanja u odnosu na originalnu `crcu8` funkciju. Nad inicijalnom vrednošću CRC se vrši bitovska ekskluzivna disjunkcija (operacija `XOR`) sa bitovima ulaznog podatka pre početka petlje, čime se pojednostavljuje obrada. Optimizovana verzija koristi tip `long` za promenu CRC vrednosti, omogućavajući direktne bitovske operacije bez potrebe za dodatnim promenljivama za čuvanje rezultata. Optimizovana verzija algoritma CRC dodatno uklanja nepotrebne operacije prisutne u neoptimizovanoj verziji. Na primer, neoptimizovana verzija koristi promenljivu `carry` za postavljanje bitova najveće težine, dok optimizovana verzija koristi direktnu uslovnu ekskluzivnu disjunkciju čime se gubi potreba za dodatnom promenljivom. Sve ove izmene čine optimizovanu funkciju bržom i efikasnijom, a pritom se zadržava ista funkcionalnost kao kod originalne funkcije.

Implementacije su dodate u projekat LLVM kao optimizacioni prolaz na nivou međureprezentacije. Naziv optimizacionog prolaza je `crc-recognition` i njegova implementacija se nalazi u datotekama `RecognizingCRC.cpp` i `RecognizingCRC.h`. Implementacije su prvobitno bile deo optimizacionog prolaza pod imenom `aggressive-instcombine` opisanog u datotekama `AggressiveInstCombine.cpp` i `AggressiveInstCombine.h`, ali su zbog preglednosti pomerene u zaseban optimizacioni prolaz.

Ideja rešenja

Rešenja koja će u nastavku biti predstavljena podrazumevaju prepoznavanje algoritma na nivou LLVM međureprezentacije, zatim uklanjanje prepoznatih instrukcija po uspešnom prepoznavanju i na kraju generisanje optimizovanog međukoda ili asemblerskog koda. Pretpostavka je da ukoliko neki program koristi algoritam CRC, taj kôd će najverovatnije biti izdvojen u posebnu funkciju kako bi se više puta koristio i kako ne bi bio repliciran. Iz tog razloga svaka od implementacija prolazi kroz sve funkcije definisane u okviru modula odnosno `.ll` datoteke

```
; ModuleID = 'syrmia_crc_unoptimized.c'
source_filename = "syrmia_crc_unoptimized.c"
target datalayout = "e-m:e-p270:32:32-p271:32:32-p272:64:64-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-pc-linux-gnu"

@.str = private unnamed_addr constant [13 x i8] c"report = %u\0A\00", align 1

; Function Attrs: noinline nounwind uwtable
define dso_local zeroext i16 @crcu8(i8 zeroext %0, i16 zeroext %1) #0 {
  %3 = alloca i8, align 1
  %4 = alloca i16, align 2
  %5 = alloca i8, align 1
  %6 = alloca i8, align 1
  %7 = alloca i8, align 1
  store i8 %0, i8* %3, align 1
  store i16 %1, i16* %4, align 2
  store i8 0, i8* %5, align 1
  store i8 0, i8* %6, align 1
  store i8 0, i8* %7, align 1
  store i8 0, i8* %5, align 1
  br label %8

8:                                                ; preds = %53, %2
  %9 = load i8, i8* %5, align 1
  %10 = zext i8 %9 to i32
  %11 = icmp slt i32 %10, 8
  br i1 %11, label %12, label %56

12:                                               ; preds = %8
  %13 = load i8, i8* %3, align 1
  %14 = zext i8 %13 to i32
  %15 = and i32 %14, 1
  %16 = load i16, i16* %4, align 2
  %17 = trunc i16 %16 to i8
  %18 = zext i8 %17 to i32
  %19 = and i32 %18, 1
  %20 = xor i32 %15, %19
  %21 = trunc i32 %20 to i8
  store i8 %21, i8* %6, align 1
  %22 = load i8, i8* %3, align 1
  %23 = zext i8 %22 to i32
  %24 = ashr i32 %23, 1
  %25 = trunc i32 %24 to i8
  store i8 %25, i8* %3, align 1
  %26 = load i8, i8* %6, align 1
  %27 = zext i8 %26 to i32
  %28 = icmp eq i32 %27, 1
  br i1 %28, label %29, label %34

29:                                               ; preds = %12
  %30 = load i16, i16* %4, align 2
  %31 = zext i16 %30 to i32
  %32 = xor i32 %31, 16386
  %33 = trunc i32 %32 to i16
```

Listing 9: Deo LLVM međukoda dobijenog prevođenjem neoptimizovane verzije algoritma CRC

```
; ModuleID = 'syrmia_crc_optimized.c'
source_filename = "syrmia_crc_optimized.c"
target datalayout = "e-m:e-p270:32:32-p271:32:32-p272:64:64-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-pc-linux-gnu"

@.str = private unnamed_addr constant [13 x i8] c"report = %u\0A\00", align 1

; Function Attrs: noinline nounwind uwtable
define dso_local zeroext i16 @crcu8_optimized(i8 noundef zeroext %0, i16 noundef zeroext %1) #0 {
  %3 = alloca i8, align 1
  %4 = alloca i16, align 2
  %5 = alloca i8, align 1
  %6 = alloca i8, align 1
  %7 = alloca i8, align 1
  %8 = alloca i64, align 8
  store i8 %0, i8* %3, align 1
  store i16 %1, i16* %4, align 2
  store i8 0, i8* %5, align 1
  store i8 0, i8* %6, align 1
  store i8 0, i8* %7, align 1
  %9 = load i16, i16* %4, align 2
  %10 = zext i16 %9 to i64
  store i64 %10, i64* %8, align 8
  %11 = load i8, i8* %3, align 1
  %12 = zext i8 %11 to i64
  %13 = load i64, i64* %8, align 8
  %14 = xor i64 %13, %12
  store i64 %14, i64* %8, align 8
  store i8 0, i8* %5, align 1
  br label %15

15:                                               ; preds = %40, %2
  %16 = load i8, i8* %5, align 1
  %17 = zext i8 %16 to i32
  %18 = icmp slt i32 %17, 8
  br i1 %18, label %19, label %43

19:                                               ; preds = %15
  %20 = load i64, i64* %8, align 8
  %21 = trunc i64 %20 to i8
  %22 = zext i8 %21 to i32
  %23 = and i32 %22, 1
  %24 = trunc i32 %23 to i8
  store i8 %24, i8* %6, align 1
  %25 = load i8, i8* %3, align 1
  %26 = zext i8 %25 to i32
  %27 = ashr i32 %26, 1
  %28 = trunc i32 %27 to i8
  store i8 %28, i8* %3, align 1
  %29 = load i64, i64* %8, align 8
  %30 = ashr i64 %29, 1
  store i64 %30, i64* %8, align 8
  %31 = load i8, i8* %6, align 1
  %32 = zext i8 %31 to i32
```

Listing 10: Deo LLVM međukoda dobijenog prevođenjem optimizovane verzije algoritma CRC

koja se obrađuje i proverava da li je u nekoj od njih sadržana neoptimizovana verzija algoritma CRC.

Prepoznavanje algoritma izvršeno je kretanjem unazad kroz instrukcije svake od funckija u modulu koji obrađujemo. Kretanje započinjemo od poslednje instrukcije poslednjeg osnovnog bloka i krećemo se unazad po prethodnim instrukcijama poslednjeg osnovnog bloka. Kretanje zatim nastavljamo po instrukcijama prethodnih osnovnih blokova, sve dok ne dođemo do prve instrukcije u prvom osnovnom bloku[1]. U svakom koraku proveravamo da li smo naišli na instrukciju koju očekujemo i proveramo da li je instrukcija pozvana nad odgovarajućim virtuelnim registrima odnosno operandima. Takođe, ako funkcija na kojoj se trenutno nalazimo kao jedan od svojih operanada sadrži konstantu proveravamo da li je ta konstanta jednaka vrednosti koju očekujemo.

Ukoliko u bilo kom koraku primetimo razliku bilo u instrukciji koju proveravamo bilo u njenim operandima, prekidamo postupak prepoznavanja i signaliziramo da traženi algoritam nije prepoznat. U protivnom, ukoliko ovim postupkom nije došlo do prekida (i ukoliko je i poslednja instrukcija proverena) zaključujemo da smo prepoznali traženi algoritam i da možemo započeti brisanje instrukcija prepoznatog algoritma.

Ovakvim načinom prepoznavanja algoritma se nameće dosta uslova koji moraju biti zadovoljeni kako bi se zaključilo da je algoritam prepoznat. Međutim, postoje provere koje uvode određenu fleksibilnost prilikom prepoznavanja. Prilikom prepoznavanja se zanemaruju nazivi argumenata, lokalnih promenljivih i nazivi osnovnih blokova. Umesto toga prate se virtuelni registri u koje su smešteni prvo argumenti funkcije, a potom i lokalne promenljive. Dodatno, za nezavisne instrukcije istog osnovnog bloka omogućeno je prepoznavanja algoritma za svaki mogući redosled takvih instrukcija.

Brisanje prepoznatih instrukcija takođe izvršavamo kretanjem unazad i uklanjanjem svake od instrukcija na koju naiđemo. Iako je prepoznavanje algoritma moglo biti izvršeno i u suprotnom smeru odnosno kretanjem unapred od prve ka poslednjoj instrukciji, brisanje instrukcija funkcije je ipak bolje obaviti kretanjem unazad, kako bi IR kôd u svakom trenutku bio u ispravnom stanju. Naime, instrukcije bliže poslednjoj instrukciji (poslednjeg osnovnog bloka) zavise od početnih instrukcija (instrukcija bližim prvom osnovnom bloku) i virtuelnih registara inicijalizovanih upravo tim početnim instrukcijama.

[1]Kretanje se izvršava linearno po blokovima nezavisno od grafa kontrole toka

Nakon uspešnog brisanja prepoznatih instrukcija, generišemo instrukcije koje pripadaju međukodu optimizovanog algoritma CRC i to radimo kreiranjem osnovnih bloka i smeštanjem odgovarajućih instrukcija unutar njih. Prepoznavanje i uklanjanje prepoznatih instrukcija su u obe implementacije izvršene kretanjem u nazad, dok je generisanje novih instrukcija izvršeno kretanjem unapred.

5.1 Implementacija na IR nivou

Ova implementacija je u potpunosti sprovedena na nivou međureprezentacije. To znači da je program napisan u programskim jezicima C ili C++ preveden prednjim delom LLVM kompilatora (alatom `Clang`) u datoteku koji sadrži međureprezentaciju početnog programa i da je potom ista `.ll` datoteka izmenjena pozivom srednjeg dela LLVM kompilatora (alata `opt`).

U kontekstu problema prepoznavanja i optimizovanja algoritma CRC, implementacija na IR nivou podrazumeva prevođenje neoptimizovane verzije ovog algoritma i izmenu njegovog međukoda tako da bude identičan međukodu koji se dobija prevođenjem optimizovane verzije algoritma. Izmena međukoda podrazumeva uklanjanje IR instrukcija neoptimizovanog algoritma CRC i generisanje novih IR instrukcija koje pripadaju međukodu optimozovanog algoritma CRC. Time od jedne sintaksno ispravne verzije IR programa dobijamo novi, takođe ispravan, IR program.

Prosleđivanjem izmenjenog međukoda zadnjem delu kompilatora dobija se objektna, asemblerska ili izvršiva datoteka koja efikasnije izvršava sve ono što je neoptimizovanom verzijom algoritma CRC zadato. Funkcije, klase (i njihove metode) i operatori korišćeni za realizaciju ove ideje navedeni su u nastavku:

getPrevNode() — metod koji omogućava kretanje unazad po instrukcijama osnovnog bloka, kretanje unazad po osnovnim blokovima funkcija kao i kretanje unazad po funkcijama nekog modula. Metod je moguće pozvati nad instancama klase `Function`, `BasicBlock` i `Instruction` i u zavisnosti od toga nad instancom koje klase je metod pozvan dobija se drugačiji prethodni čvor. Ako se metod pozove nad objektom koji predstavlja instrukciju dobija se pokazivač na prvu prethodnu instrukciju u okviru istog osnovnog bloka. Ako se metod pozove nad objektom klase `BasicBlock` dobija se pokazivač na prvi prethodni osnovni blok u okviru iste funkcije. Na kraju, kao rezultat

poziva metoda nad instancom klase `Function` dobija se pokazivač na prvu prethodno definisanu funkciju u okviru istog modula.

getParent() — metod vraća pokazivač na entitet u okviru koga se nalazi entitet nad kojim je metod pozvan. Ukoliko se metod pozove nad instancom klase `Instruction`, dobija se pokazivač na instancu klase `BasicBlock` koja predstavlja osnovni blok u kome se nalazi instrukcija nad kojom je pozvan metod. Pozivom metoda nad objektom klase `BasicBlock` kao rezultat se dobija pokazivač na funkciju u okviru koje se taj objekat tog osnovnog bloka nalazi. Metod radi analogno u slučaju njegovog poziva nad instancom klase `Function`.

dyn_cast — operator koji omogućava dinamičku konverziju objekta jednog tipa u objekat drugog tipa (ukoliko su oni u odnosu nasleđivanja). U kontekstu prepoznavanja instrukcija operator `dyn_cast` se koristi za konverziju instance klase `Instruction` u tačno određen tip instrukcije, odnosno u instancu klase koja odgovara tom tipu. U slučaju da je instrukcija nad kojom se vrši konverzija drugog tipa u odnosu na tip naveden u operatoru `dyn_cast` rezultat je vrednost `NULL`.

match() — funkcija koja vrši upoređivanje objekta klase `Instruction` sa obrascem formiranim korišćenjem `m_Value()`, `m_Load()`, `m_Add()`, `m_Xor()` i drugih sličnih funkcija iz `PatternMatch` prostora imena. Funkcija `match()` kao svoj prvi argument prihvata objekat klase `Instruction`, a pomenuti obrazac kao svoj drugi argument. Funkcija proverava da li je objekat koji predstavlja ispravnu IR instrukciju moguće uklopiti u obrazac i vraća `True` u slučaju da je to moguće uraditi, odnosno `False` u suprotnom.

eraseFromParent() — metod koji omogućava brisanje instrukcije iz osnovnog bloka kojem pripada, brisanje osnovnog bloka iz funckije u kojoj se nalazi ili brisanje funkcije iz svog modula. U zavisnosti od toga nad instancom koje klase je metod pozvan izvršiće se odgovarajuće brisanje.

IRBuilder — klasa koja implementira metode kao što su `CreateAlloca()`, `CreateStore()`, `CreateLoad()`, `CreateAdd()`, `CreateMul()`, `CreateXor()` i druge. Ovim metodama je redom moguće kreirati `alloca`, `store`, `load`, `mul`, `xor` i druge IR instrukcije. Konstruktor klase `IRBuilder` kao argumente prihvata pokazivač na IR instruckiju ili pokazivač na osnovni blok. Ovim

argumentima se zadaje pozicija od koje će se kreirati i ubacivati nove IR instrukcije. Instrukcije se podrazumevano ubacuju odmah nakon prosleđene IR instrukcije odnosno odmah nakon poslednje IR instrukcije u prosleđenom osnovnom bloku.

Create() — metoda koju implementira `BasicBlock` klasa. Ovom metodom moguće je kreirati novi osnovni blok.

Za pokretanje prvog implementacionog rešenja potrebno je izvršiti naredbe prikazane u listingu 11. Prvom komandom se program napisan u programskom jeziku C (sadržan u datoteci `syrmia-crc-unoptimized.c`) prevodi u LLVM međureprezentaciju upotrebom prednjeg dela LLVM kompilatora odnosno alata `Clang`. Drugom komandom se pokreće LLVM optimizator, to jest, alat `opt` kojim se započinju analize i transformacije međukoda dobijenog pozivom prethodne komande. Kao rezultat poziva ove komande dobija se optimizovana LLVM međureprezentacija.

```
$ clang -O0 -Xclang -disable-O0-optnone -S -emit-llvm
↪  ../test/syrmia_crc_unoptimized.c -o syrmia_crc_unoptimized.ll
$ ../build/bin/opt -crc-opt -S syrmia_crc_unoptimized.ll
↪  -passes=crc-recognition -o syrmia_crc_unoptimized.ll
```

Listing 11: Pokretanje prvog implementacionog rešenja

Opcijama `-S` i `-emit-llvm` navedenim u prvoj komandi se naglašava da je ulaznu datoteku potrebno prevesti u LLVM međureprezentaciju. Opcija `-disable-O0-optnone` služi da omogući da eventualni poziv LLVM optimizatora nad izlaznom datotekom (datotekom dobijenom pozivom prve komande) može da sprovede transformacije nad LLVM međukodom.

U drugoj naredbi su navedene opcije `-S`, `-passes` i `-crc-opt`. Opcijom `-S` naglašavamo da je izlaznu datoteku potrebno predstaviti LLVM međureprezentacijom, a ne bajtkodom (eng. *bytecode*). Opcijom `-passes` biramo optimizacioni prolaz koji želimo da pokrenemo, tj. `crc-recognition`. Na kraju, opcija `-crc-opt` nije podrazumevana opcija za alat *opt* već je to korisnički definisana opcija. Tom opcijom se u optimizacionom prolazu `crc-recognition` naglašava da je potrebno upotrebiti provere iz implementacije na IR nivou. U listingu 12 možete videti način na koji je moguće uvesti novu opciju koja se kasnije može proslediti prilikom poziva LLVM optimizatoru.

Rezultat implementacije na IR nivou se dobija pokretanjem LLVM optimizatora. Njegovim pokretanjem ulazna `.ll` datoteka (dobijena pozivom prednjeg dela LLVM nad neoptimizovanim algoritmom CRC) će biti prepravljena tako da sadrži međukod optimizovanog algoritma CRC. Pozivom alata `diff` se dobijeni međukod može uporediti sa međukodom optimizovanog algoritma CRC. Upoređivanjem uviđamo da su međukodovi identični. To znači da se sa rezultatom implementacije na IR nivou može nastaviti proces prevođenja i da se od istog može dobiti izvršiva datoteka koja će izvršavati optimizovani algoritam CRC.

```
static cl::opt<bool> UseNaiveCRCOptimization("crc-opt", cl::init(false),
↪  cl::Hidden, cl::desc("running IR level CRC algorithm optimization"));

static cl::opt<bool> UseIntrinsicsCRCOptimization("crc-opt-intrinsic",
↪  cl::init(false), cl::Hidden, cl::desc("running CRC algorithm
↪  optimization with intrinsic function usage"));
```

Listing 12: Uvođenje korisnički definisanih opcija za LLVM optimizator

5.2 Implementacija pomoću intrinzičkih funkcija

Implementacija pomoću intrinzičkih funkcija je slična implementaciji na IR nivou, ali se u njenom slučaju optimizovani kôd generiše u kasnijim fazama prevođenja i na drugačiji način. Još jedna bitna razlika između ove dve implementacije jeste u krajnjoj arhitekturi za koju su namenjene. Implementaciju na IR nivou je moguće prevesti i pokrenuti na proizvoljnoj procesorskoj arhitekturi, dok je u slučaju implementacije pomoću intrinzičkih funkcija to jedino moguće uraditi za RISC-V procesorsku arhitekturu. Ukoliko nije poznato za koju arhitekturu je potrebno optimizovati algoritam CRC preporučuje se korišćenje implementacije na IR nivou, a ukoliko je odabrana arhitektura baš RISC-V u tom slučaju je bolje koristiti implementaciju pomoću intrinzičkih funkcija.

Implementacija pomoću intrinzičkih funkcija takođe počiva na prepoznavanju neoptimizovane verzije algoritma CRC i uklanjanju prepoznatih instrukcija. Međutim, nakon uklanjanja prepoznatih instrukcija ova implementacija predlaže kreiranje poziva intrinzičke funkcije pod nazivom `riscv-crc` i vraćanje njene povratne vrednosti. Argumenti intrinzičke funkcije su identični argumentima funkcije

čiji kôd prepoznajemo i navode se u istom redosledu. Pred sam kraj prevođenja programa intrinzička funkcija se zamenjuje sekvencom mašinskih instrukcija koje pripadaju optimizovanoj verziji algoritma CRC.

Intrinzička funkcija `riscv-crc` je deklarisana u `TableGen` datoteci `IntrinsicsRISCV.td`. Ista deklaracija je mogla biti urađena i u `Intrinsics.td` datoteci, ali je odabrani pristup bolji jer se njime precizira za koju procesorsku arhitekturu je intrinzička funkcija namenjena. Deklaracija funkcije `riscv-crc` prikazana je u listingu 13. Deklaracijom je eksplicitno navedeno kog tipa treba da budu argumenti i povratna vrednost ove intrinzičke funkcije.

```
def int_riscv_crc : DefaultAttrsIntrinsic<[llvm_i16_ty], [llvm_i8_ty,
↪  llvm_i16_ty]>;
  }
```

Listing 13: Deklarisanje intrinzičke funkcije `riscv-crc`

Nakon kreiranja IR instrukcije koja sadrži poziv `riscv-crc` intrinzičke funkcije naredni korak u prevođenju jeste spuštanje te instrukcije u validan `SelectionDag` čvor. Spuštanje ove IR instrukcije je omogućeno kreiranjem promena unutar datoteke `RISCVIselLowering.cpp`. Tim promenama se omogućava kreiranje `SelectionDag` čvora pod nazivom `pseudo_riscv_crc`, a samim tim i propagiranje poziva intrinzičke funkcije kroz kasnije faze prevođenja. Kako `pseudo_riscv_crc SelectionDag` čvor nije podrazumevano podržan u okviru LLVM-a, izvršena je njegova deklaracija u okviru datoteke `RISCVInstrInfo.td`. Deklaracijom se, takođe, omogućava kasnije prevođenje `SelectionDAG` čvora u „pseudo" mašinsku instrukciju.

Pod pseudo mašinskom instrukcijom se podrazumeva instrukcija čiji asemblerski zapis neće biti iskorišćen već će biti zamenjen asemblerskim zapisom druge mašinske instrukcije ili sekvence mašinskih instrukcija. Deklaracija „pseudo" mašinske instrukcije `pseudo_crc` je takođe izvršena u okviru datoteke `RISCVInstrInfo.td` i prikazana je u listingu 14.

```
def PseudoCRC32 : Pseudo<(outs GPR:$dst),
                         (ins GPR:$lhs, GPR:$rhs), [], "pseudo_crc",
                         ↪  "$dst, $lhs, $rhs">;
}
```

Listing 14: Deklaracija „pseudo" mašinske instrukcije `pseudo_crc`

Za pokretanje implementacije pomoću intrinzičkih funkcija potrebno je pored alata `clang` i `opt` (koji se u ovom slučaju pozivaju nad drugim argumentima) pokrenuti i alat `llc` (zadnji deo LLVM kompilatora). Komande su prikazane u listingu 15. Prva komanda je ekvivalentna odgovarajućoj prvoj komandi potrebnoj za pokretanje implementacije na IR nivou (i služi za prevođenje izvornog koda u LLVM međukod). Razlika u odnosu na poziv prethodne prve komande jeste u dvema dodatnim opcijama. U pitanju su opcije `-target` i `-march`. Opcijom `-target` se bira uopštena konfiguracija procesora na kome želimo da se krajnji mašinski kôd izvršava. Opcija `-march` služi za dodavanje specifičnih atributa za već odabrani procesor. Vrednost `rv64izbc` predstavlja posebnu verziju RISC-V procesora širine 64 bita koji nam omogućava korišćenje instrukcije `clmul`.

```
$ clang -O0 -Xclang -disable-O0-optnone -S -emit-llvm -target
↪  riscv64-unknown-linux-gnu -march=rv64izbc
↪  ../test/syrmia_crc_unoptimized.c -o syrmia_crc_unoptimized.ll
$ ../build/bin/opt -crc-opt-intrinsic -S syrmia_crc_unoptimized.ll
↪  -passes=crc-recognition -o syrmia_crc_unoptimized.ll
$ ../build/bin/llc syrmia_crc_unoptimized.ll -print-after-all -debug -o
↪  - &> out.txt
```

Listing 15: Pokretanje implementacionog rešenja zasnovanog na intrinzičkim funkcijama

Drugom komandom se, kao i u slučaju pokretanja implementacije na IR nivou, pokreće LLVM optimizator (alat `opt`) koji sprovodi analize i transformacije nad datotekom dobijenom iz prve komande. Opcijom `-passes` ponovo biramo optimizacioni prolaz `crc-recognition`, ali ovoga puta biramo opciju `-crc-opt-intrinsic` kojom naglašavamo da prepoznati algoritam CRC bude optimizovan korišćenjem intrinzičkih funkcija.

Trećom komandom se poziva zadnji deo LLVM kompilatora (alat `llc`) sa opcijama `-print-after-all` i `-debug`. Opcija `-print-after-all` omogućava ispis stanja posle svakog IR i MIR prolaza, ali ne omogućava ispis međustanja kroz koja prolaze `SelectionDAG` čvorovi. Iz tog razloga je potrebno dodati i opciju `-debug`. Navođenje ovih opcija ne utiče na rezultat rada alata `llc`, već se njima samo stiče uvid u njegov rad i olakšava debagovanje. Pokretanjem treće komande se kreira datoteka `out.txt` u kojoj se nalazi detaljan prikaz rada zadnjeg dela LLVM kompilatora. Na samom kraju datoteke `out.txt` prikazan je asemblerski kôd koji predstavlja krajnji rezultat rada alata `llc`.

Dobijeni asemblerski kôd je identičan asemblerskom kodu navedenom u članku koji je korišćen kao resurs [29]. U pomenutom članku je predstavljena podrška za prevođenje algoritma CRC u okviru kompilatora GCC i njegovo efikasnije izvršavanje na arhitekturi RISC-V, te je ideja bila omogućiti istu podršku i u slučaju LLVM kompilatora. Asemblerski kôd naveden u članku predstavlja implementaciju algoritma CRC koja koristi instrukciju `clmul` i koja bi iz tog razloga trebalo da se efikasno izvršava na arhitekturi RISC-V. Predloženi asemblerski kôd naveden je u listingu 16.

```
li      a4, quotient
li      a5, polynomial
xor     a0, a1, a0
clmul   a0, a0, a4
srli    a0, a0, crc_size
clmul   a0, a0, a5
slli    a0, a0, GET_MODE_BITSIZE (word_mode) - crc_size
srli    a0, a0, GET_MODE_BITSIZE (word_mode) - crc_size
ret
```

Listing 16: Asemblerski kôd optimizovanog algoritma CRC predložen u artiklu korišćenom kao resurs

Dobijeni asemblerski kôd namenjen je za izvršavanje na procesorskoj arhitekturi RISC-V, a mašina na kojoj je proces prevođenja pokrenut pripada CISC arhitekturi procesora. To znači da dobijeni asemberski kôd nije moguće izvršiti na mašini na kojoj je proces prevođenja pokrenut. Ovakav postupak prevođenja programa za mašinu čiji su procesor, operativni sistem i druge specifikacije potencijalno drugačije od mašine na kojoj se program prevodi naziva se kros-kompilacija (eng. *cross compiling*).

5.3 Regresioni testovi

Regresioni testovi kreirani za potrebe testiranja implementacija se mogu pronaći u okviru `regression_tests` direktorijuma unutar `GitHub` repozitorijuma koji sadrži sav materijal ovog rada. Za obe implementacije je kreiran po jedan regresioni test kojim se proverava da li je algoritam CRC uspešno prepoznat i optimizovan (pokretanjem odgovarajuće implementacije).

U svakom od testova su korišćene klauze `CHECK` i `CHECK-NEXT`. Svaka od ovih klauza kao svoj argument prihvata string koji se navodi u istoj liniji u kojoj i

klauza (nakon njenog naziva i znaka ':'). Klauzom `CHECK` se proverava da li je string naveden kao argument klauze sadržan u izlazu nad kojim se vrši pretraga (sadržaju datoteke ili izlazu generisanom od strane nekog alata). Klauzom `CHECK-NEXT` se vrši slična provera. Ovom klauzom se očekuje da string naveden kao argument bude sadržan u liniji koja sledi odmah nakon linije proverene klauzom `CHECK`. To znači da bi klauzulu `CHECK` trebalo koristiti za pronalaženje prve linije u obrascu koji očekujemo. Za pronalaženje svake sledeće linije u obrascu, pod uslovom da se obrazac sastoji od sekvence uzastopnih linija (što sa rezultatima predstavljenih implementacija jeste slučaj), trebalo bi koristiti klauzulu `CHECK-NEXT`. U slučaju obrasca koji čine neuzastopne linije moguće je više puta koristiti klauzu `CHECK` ili koristiti druge klauze koje alat `Filecheck` podržava.

Regresionim testom namenjenim za testiranje implementacije na IR nivou proverava se da li je LLVM međukod dobijen pokretanjem implementacije identičan međukodu optimizovanog algoritma CRC. Test je moguće upotrebiti i nad modifikacijama neoptimizovane verzije algoritma CRC. Modifikacije algoritma su napravljene permutovanjem redosleda nezavisnih instrukcija u okviru istog osnovnog bloka. Neophodno je svaku od modifikacija prevesti prednjim delom LLVM kompilatora i rezultat prevođenja smestiti u datoteku pod nazivom `syrmia_crc_unoptimized.ll`. Nakon toga je moguće pokrenuti test.

Za potrebe testiranja implementacije pomoću intrinzičkih funkcija kreiran je takođe jedan regresioni test prikazan u listingu 17. Testom se proverava da li je asemblerski kôd generisan od strane alata *llc* identičan kodu koji očekujemo (odnosno kodu navedenom u korišćenom resursu). Test je takođe moguće upotrebiti nad različitim modifikacijama originalnog neoptimizovanog algoritma CRC.

5.4 Funkcionalna ispravnost implementacije

Programi korišćeni za proveru fukcionalne ispravnosti implementacije na IR nivou se nalaze unutar direktorijuma `implementation/functional_equivalence` u okviru `GitHub` repozitorijuma koji sadrži sav materijal rada. U pitanju su programi: `generate_inputs.cpp`, `optimized_crc.c` i `unoptimized_crc.c`.

Program `generate_inputs.cpp` služi za generisanje nasumičnih ulaza (argumenata) nad kojima će odgovarajuće verzije algoritma CRC biti pozivane. Program izgenerisane ulaze upisuje u datoteku `inputs.txt` tako da se svaki ulaz (par argumenata) nalazi u posebnoj liniji.

```
; RUN: ../build/bin/llc syrmia_crc_unoptimized.ll -print-after-all -debug
↪  %s 2>&1 | FileCheck %s

; CHECK: crcu8:
crcu8:
; CHECK-NEXT: li      a4, quotient
   li      a4, quotient
; CHECK-NEXT: li      a5, polynomial
   li      a5, polynomial
; CHECK-NEXT: xor     a0, a1, a0
   xor     a0, a1, a0
; CHECK-NEXT: clmul   a0, a0, a4
   clmul   a0, a0, a4
; CHECK-NEXT: srli    a0, a0, crc_size
   srli    a0, a0, crc_size
; CHECK-NEXT: clmul   a0, a0, a5
   clmul   a0, a0, a5
; CHECK-NEXT: slli    a0, a0, GET_MODE_BITSIZE (word_mode) - crc_size
   slli    a0, a0, GET_MODE_BITSIZE (word_mode) - crc_size
; CHECK-NEXT: srli    a0, a0, GET_MODE_BITSIZE (word_mode) - crc_size
   srli    a0, a0, GET_MODE_BITSIZE (word_mode) - crc_size
; CHECK-NEXT: ret
   ret
```

Listing 17: Regresioni test kreiran za proveru implementacije pomoću intrinzičkih funkcija

Program `optimized_crc.c` iščitava sadržaj datoteke `inputs.txt` i nad njima poziva optimizovanu verziju algoritma CRC. Rezultate izvršavanja optimizovane verzije algoritma program upisuje u datoteku čiji je naziv prosleđen kao argument komandne linije. Program `unoptimized_crc.c` takođe iščitava sadržaj datoteke `inputs.txt` i nad njima poziva neoptimizovanu verziju algoritma CRC. Rezultati izvršavanja neoptimizovane verzije algoritma se, takođe, upisuju u datoteku čiji je naziv prosleđen kao argument komandne linije.

Program `optimized_crc.c` se prevodi korišćenjem kompilatora Clang. Program `unoptimized_crc.c` se prevodi na dva načina. Prvi put pomoću kompilatora Clang, a drugi put komandama za pokretanje optimizacije na IR nivou i nakon toga kompilatorom Clang. Drugi način prevođenja programa `unoptimized_crc.c` omogućava kreiranje izvršive datoteke na osnovu međukoda dobijenog pokretanjem implementacije na IR nivou (nad neoptimizovanim algoritmom CRC).

Pokretanjem svih izvršivih datoteka (dobijenih iz odgovarajućih prevođenja)

nastaju tri različite tekstualne datoteke imenovane prema prosleđenim argumentima komandne linije. Datoteke sadrže rezultate odgovarajućih verzija algoritma CRC pokrenutih nad istim argumentima. Korišćenjem alata `diff` za upoređivanje sadržaja svakog para datoteka dobija se da su im sadržaji identični. Time se zaključuje da su korišćene verzije algoritma CRC funkcionalno ekvivalentne, a takođe se zaključuje i da se pokretanjem implementacije na IR nivou nad neoptimizovanom verzijom algoritma zadržava funkcionalna ekvivalentnost.

Postupak testiranja mašinskog koda za arhitekturu RISC-V (dobijenog pokretanjem implementacije pomoću intrinzičkih funckija) potrebno je sprovesti ili na odgovarajućem procesoru ili na nekom od emulatora[2]. Opisana implementacija nažalost nije dala željene rezultate. Njenim pokretanjem ustanovljeno je da generisani asemblerski kôd nije funkcionalno ekvivalentan početnom algoritmu CRC[3]. U skladu sa time, evaluacija efikasnosti rešenja rađena je samo za implementaciju na IR nivou.

5.5 Rezultati evaluacije efikasnosti optimizacije

Eksperimentalna evaluacija efikasnosti optimizacije je urađena na Dell računaru 15. generacije sa 16 gigabajta radne memorije, procesorom Intel Core i7 13. generacije i operativnim sistemom Ubuntu 22.04.4. Ovaj postupak je, kao i provera funkcionalne ispravnosti, izvršen za neoptimizovanu i optimizovanu verziju algoritma CRC koje su prevedene kompilatorom Clang, kao i za neoptimizovanu verziju algoritma CRC nad kojom je pokrenuta implementacija na IR nivou, a zatim nastavljeno prevođenje pomoću kompilatora Clang. Programi korišćeni za poređenje vremena izvršavanja nalaze se unutar direktorijuma `implementation/time_measurement` u okviru `GitHub` repozitorijuma koji sadrži sav materijal rada.

Postupak poređenja je izvršen kroz 4 serije od ukupno 5 ponavljanja. U serijama se redom porede vremena izvršavanje nad ulazima veličine 100 000, 1 000 000, 10 000 000 i 100 000 000. U svakom ponavljanju u okviru iste serije se generišu nasumični ulazi i smeštaju u datoteku tako da se u svakoj liniji nalazi jedan ulaz.

[2]Primer emulatora je QEMU [12]. QEMU omogućava izvršavanje jednog operativnog sistema i programa jedne mašine na drugoj mašini. Često se koristi za virtuelizaciju izvršavanja programa na udaljenoj mašini sa drugačijim specifikacijama u odnosu na mašinu na kojoj je emulator pokrenut.

[3]Možda je to i razlog zašto implementacija opisana u članku koji je služio kao osnova za implementaciju nije integrisana u projekat GCC.

Broj linija u datoteci odgovara veličini ulaza za tu seriju. Nakon generisanja ulaza odgovarajuća verzija algoritma se prevodi jednim od opisanih načina prevođenja, a zatim pokreće nad svakim izgnerisanim ulazom (svakom linijom u datoteci). Ukupno vreme izvršavanja predstavlja vreme potrebno da se odgovarajuća verzija algoritma izvrši za svaki izgenerisani ulaz. Rezultati dobijeni izvođenjem serija prikazani su u tebelama 5.1, 5.2, 5.3 i 5.4.

Vremena izvršavanja neoptimizovane verzije algoritma CRC prevedene kompilatorom Clang su prikazana u koloni „`unoptimized_crc.c`". Vremena izvršavanja optimizovane verzije algoritma prevedene kompilatorom Clang su prikazana u koloni „`optimized_crc.c`". U koloni „`unoptimized_crc.c + optimizacija`" su predstavljena vremena izvršavanja neoptimizovane verzije algoritma CRC nad kojom je pokrenuta optimizacija na IR nivou.

U tabelama je prikazan prosečan broj milisekundi potreban za izvršavanje svake od verzija algoritma (prevedene na odgovarajući način) u okviru odgovarajuće serije. Takođe, prikazan je i procenat bolje efikasnosti prosečne vrednosti rezultata u kolonama „`optimized_crc.c`" i „`unoptimized_crc.c + optimizacija`" nad prosečnom vrednošću rezultata iz kolone „`unoptimized_crc.c`". Na osnovu rezultata iz tabele kreiran je grafik prikazan na slici 5.5.

Rezultati pokazuju da je optimizovana verzija algoritma CRC u proseku za 34.6% vremenski efikasnija u odnosu na neoptimizovanu verziju algoritma. Takođe, pokretanjem optimizacije na IR nivou (nad neoptimizovanom verzijom algoritma CRC) se u proseku postiže gotovo isto poboljšanje od 35.1%. Na grafiku se može videti kako se rezultati prikazani u kolonama ,,`optimized_crc.c`'' i ,,`unoptimized_crc.c + optimizacija`'' poklapaju. Ovakav rezultat je i očekivan budući da je međukod optimizovanog algoritma CRC korišćen u okviru implementacije optimizacije na IR nivou.

Ulaz veličine: 100 000	`unoptimized_crc.c`	`optimized_crc.c`	`unoptimized_crc.c` + optimizacija
Ponavljanje broj 1:	25ms	15ms	15ms
Ponavljanje broj 2:	24ms	15ms	16ms
Ponavljanje broj 3:	25ms	16ms	15ms
Ponavljanje broj 4:	24ms	15ms	15ms
Ponavljanje broj 5:	25ms	15ms	15ms
Prosečno vreme izvršavanja:	**24.6ms**	**15.2ms**	**15.2ms**
Prosečan procenat ubrzanja:		**38.2%**	**38.2%**

Slika 5.1: Poređenje vremena izvršavanja obe verzije algoritma CRC nad ulazom veličine 100 000

Ulaz veličine: 1 000 000	`unoptimized_crc.c`	`optimized_crc.c`	`unoptimized_crc.c` + optimizacija
Ponavljanje broj 1:	194ms	140ms	141ms
Ponavljanje broj 2:	205ms	143ms	142ms
Ponavljanje broj 3:	199ms	141ms	143ms
Ponavljanje broj 4:	195ms	143ms	142ms
Ponavljanje broj 5:	198ms	141ms	141ms
Prosečno vreme izvršavanja:	**198.2ms**	**141.6ms**	**141.8ms**
Prosečan procenat ubrzanja:		**28.6%**	**28.5%**

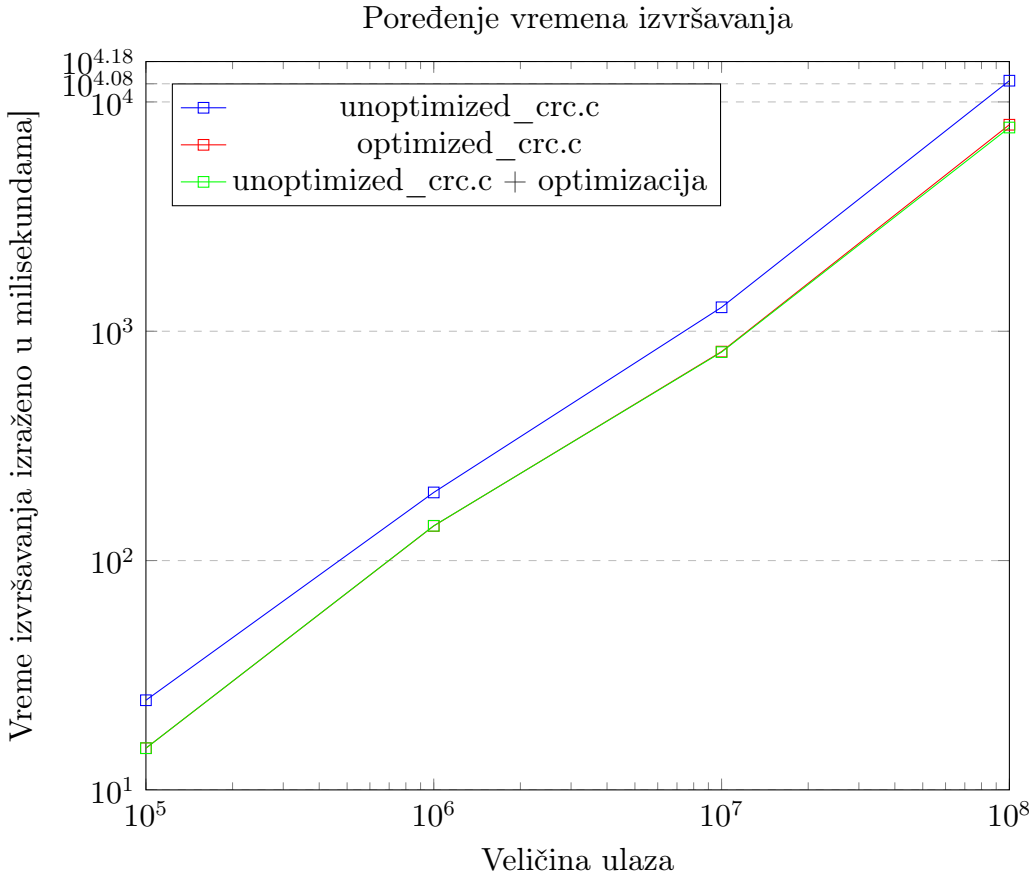
Slika 5.2: Poređenje vremena izvršavanja obe verzije algoritma CRC nad ulazom veličine 1 000 000

Ulaz veličine: 10 000 000	`unoptimized_crc.c`	`optimized_crc.c`	`unoptimized_crc.c` + optimizacija
Ponavljanje broj 1:	1260ms	820ms	818ms
Ponavljanje broj 2:	1243ms	826ms	819ms
Ponavljanje broj 3:	1230ms	812ms	822ms
Ponavljanje broj 4:	1282ms	825ms	800ms
Ponavljanje broj 5:	1345ms	793ms	795ms
Prosečno vreme izvršavanja:	**1272.0ms**	**815.2ms**	**810.8ms**
Prosečan procenat ubrzanja:		**35.9%**	**36.3%**

Slika 5.3: Poređenje vremena izvršavanja obe verzije algoritma CRC nad ulazom veličine 10 000 000

Ulaz veličine: 100 000 000	`unoptimized_crc.c`	`optimized_crc.c`	`unoptimized_crc.c` + optimizacija
Ponavljanje broj 1:	12376ms	7831ms	7656ms
Ponavljanje broj 2:	12303ms	7883ms	7909ms
Ponavljanje broj 3:	12641ms	8049ms	7948ms
Ponavljanje broj 4:	12240ms	8088ms	7553ms
Ponavljanje broj 5:	12319ms	7915ms	7607ms
Prosečno vreme izvršavanja:	**12375.8ms**	**7953.2ms**	**7734.6ms**
Prosečan procenat ubrzanja:		**35.7%**	**37.5%**

Slika 5.4: Poređenje vremena izvršavanja obe verzije algoritma CRC nad ulazom veličine 100 000 000



Slika 5.5: Grafik poređenja vremena izvršavanja neoptimizovane i optimizovane verzije algoritma CRC

Glava 6

Zaključak

Cilj ovog rada je da predstavi jedan pravac unapređenja kompilatorske infrastrukture LLVM i da precizno opiše način na koji je takva unapređenja moguće integrisati u projekat LLVM. Kao ilustracija ove vrste unapređenja odabran je problem detekcije i optimizacije algoritma CRC zbog važnosti i sve učestalije primene ovog algoritma.

Glavni predmet rada je uvođenje novog optimizacionog prolaza u okviru projekta LLVM koji na nivou LLVM međureprezentacije detektuje određenu sekvencu IR instrukcija i zamenjuje je drugom. U radu su predstavljene dve mogućnosti, prva koja sekvencu IR instrukcija zamenjuje drugom sekvencom IR istrukcija i druga koja sekvencu IR isntrukcija zamenjuje pozivom intrinzičke funkcije. Oba rešenja sadržana su u kreiranom optimizacionom prolazu. Ideja predloženih rešenja može biti iskorišćena za prepoznavanje i optimizovanje drugih verzija algoritma CRC kao i potpuno drugih algoritama i aritmetičkih izraza.

Doprinos ove teze se jeste predstavljanje metoda za prepoznavanje nekog algoritma korišćenjem kompilatora LLVM. Svrha prepoznavanja i optimizovanja nekog algoritma jeste da se poveća efikasnost programa dobijenog na samom kraju prevođenja. Efikasnost se odnosi na smanjenje vremena izvršavanja i smanjenje memorijskog zauzeća programa. Pokretanjem implementacije na IR nivou postiže se smanjenje vremena izvršavanja neoptimizovanog algoritma CRC za čak 37%.

Postoji nekoliko pravaca u kojima bi predložena rešenja mogla biti poboljšana. Trenutne implementacije funkcionišu isključivo za pristup odabiru instrukcija ostvarenom kroz `SelectionDAG`. U budućnosti bi trebalo podržati pristup selekcije instrukcija `GlobalIsel` koji se sve više koristi.

Detektovanje i optimizovanje algoritma CRC bi moglo biti omogućeno za veli-

ki broj drugih verzija algoritma primenom pristupa predstavljenog u ovom radu. Postupak prepoznavanja bi mogao biti poboljšan tako da prepoznaje još neke verzije neoptimizovanog algoritma CRC. Predlog implementacije pomoću intrinzičkih funkcija bi mogao biti izmenjen tako da bude podržan i na drugim arhitekturama.

Nadam se da će ovaj rad čitaocima pomoći da se bolje upoznaju sa projektom LLVM, algoritmom CRC i uopšte postupkom optimizovanja nekog algoritma. Takođe, nadam da će rad dati motivaciju i smernice za dalje unapređivanje LLVM i drugih kompilatora kako bi se i neki drugi algoritmi, aritmetički izrazi ili često korišćene programske konstrukcije prepoznavali, optimizovali i kasnije efikasnije izvršavali.

Bibliografija

[1] Polly - LLVM Framework for High-Level Loop and Data-Locality Optimizations. on-line at: `https://polly.llvm.org/`.

[2] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., USA, 2006.

[3] Chris Arnott. How do compilers work 1 — front end. *Medium*, 2017. on-line at: `https://medium.com/@ChrisCanCompute/how-do-compilers-work-1-front-end-5c308b56c44c`.

[4] Chris Arnott. How do compilers work 2 — middle end. *Medium*, 2017. on-line at: `https://medium.com/@ChrisCanCompute/how-do-compilers-work-2-middle-end-c4c8cff80b90`.

[5] Chris Arnott. How do compilers work 3 — back end. *Medium*, 2017. on-line at: `https://medium.com/@ChrisCanCompute/how-do-compilers-work-3-back-end-bcd860b849d9`.

[6] Andrew Waterman Yunsup Lee David A. Patterson Krste Asanovic. The RISC-V Instruction Set Manual, Volume I: Base User-Level ISA. on-line at: `https://people.eecs.berkeley.edu/~krste/papers/EECS-2011-62.pdf`.

[7] Elsevier B.V. Ethernet Protocol - an overview. on-line at: `https://www.sciencedirect.com/topics/computer-science/ethernet-protocol`.

[8] ChipsAlliance. Chisel Software-defined hardware. on-line at: `https://www.chisel-lang.org/`.

[9] Codasip. What is a RISC-V processor? on-line at: `https://codasip.com/glossary/risc-v-processors/`.

[10] Intel Corporation. Intel Processors for PC, Laptops, Servers, and AI. on-line at: `https://www.intel.com/content/www/us/en/products/details/processors.html`.

[11] Nvidia Corporation. CUDA LLVM Compiler, 2024. on-line at: `https://developer.nvidia.com/cuda-llvm-compiler`.

[12] The QEMU Project Developers. GNU QEMU. on-line at: `https://www.qemu.org/`.

[13] Science Direct. Introduction: Data Flow Analysis. on-line at: `https://edurev.in/t/187294/Introduction-Data-Flow-Analysis`.

[14] Free Software Foundation. Rust Compiler Development Guide. on-line at: `https://rustc-dev-guide.rust-lang.org/getting-started.html`.

[15] Free Software Foundation. GNU gcc, 2013. on-line at: `http://gcc.gnu.org/`.

[16] RISC-V Foundation. RISC-V Bit-manipulation A, B, C and S Extensions. on-line at: `https://five-embeddev.com/riscv-bitmanip/1.0.0/bitmanip.html`.

[17] The LLVM Foundation. Clang: a C language family frontend for LLVM. on-line at: `https://clang.llvm.org/`.

[18] The LLVM Foundation. FileCheck - Flexible pattern matching file verifier. on-line at: `https://llvm.org/docs/CommandGuide/FileCheck.html`.

[19] The LLVM Foundation. lit - LLVM Integrated Tester. on-line at: `https://www.llvm.org/docs/CommandGuide/lit.html`.

[20] The LLVM Foundation. LLD - The LLVM Linker. on-line at: `https://lld.llvm.org/`.

[21] The LLVM Foundation. LLVM Alias Analysis Infrastructure. on-line at: `https://llvm.org/docs/AliasAnalysis.html`.

[22] The LLVM Foundation. LLVM Core Libraries. on-line at: `https://llvm.org/docs/doxygen/group__LLVMCCore.html`.

[23] The LLVM Foundation. Machine IR (MIR) Format Reference Manual. on-line at: `https://llvm.org/docs/MIRLangRef.html`.

[24] The LLVM Foundation. test-suite Guide. on-line at: `https://www.llvm.org/docs/TestSuiteGuide.html`.

[25] The LLVM Foundation. The LLDB Debugger. on-line at: `https://lldb.llvm.org/`.

[26] The LLVM Foundation. The LLVM Compiler Infrastructure. on-line at: `https://llvm.org/`.

[27] GeeksforGeeks. Abstract Syntax Tree vs Parse Tree. on-line at: `https://www.geeksforgeeks.org/abstract-syntax-tree-vs-parse-tree/`.

[28] GeeksforGeeks. Pointer-Analysis. on-line at: `https://www.geeksforgeeks.org/pointer-analysis/`.

[29] Mariam Harutyunyan. RISC-V: Added support for CRC. on-line at: `https://www.mail-archive.com/gcc-patches@gcc.gnu.org/msg315381.html`.

[30] ACM Inc. ACM Software System Award, 2024. on-line at: `https://awards.acm.org/software-system`.

[31] Modular Inc. Mojo Programming Language, 2024. on-line at: `https://www.modular.com/max/mojo`.

[32] RISC-V International. History of RISC-V. on-line at: `https://riscv.org/about/history/`.

[33] RISC-V International. RISCV ISA: Specifications. on-line at: `https://riscv.org/technical/specifications/`.

[34] LLVM/OpenMP. Welcome to the documentation of OpenMP in LLVM! on-line at: `https://openmp.llvm.org/`.

[35] Phoronix Media. Google Is Hiring More LLVM/Clang Developers. on-line at: `https://www.phoronix.com/news/Google-More-LLVM-Developers`.

[36] Arm Limited (or its affiliates). ARM: Microprocessor Cores and Processor Technology. on-line at: `https://www.arm.com/products/silicon-ip-cpu`.

[37] Oracle. What is Oracle GraalVM?, 2024. on-line at: `https://www.oracle.com/java/graalvm/what-is-graalvm/`.

[38] Inc. Quiller Media. Apple's other open secret: the LLVM Compiler, 2024. on-line at: `https://appleinsider.com/articles/08/06/20/apples_other_open_secret_the_llvm_complier`.

[39] Simplilearn Solutions. All Rights Reserved. What Is Cyclic Redundancy Check (CRC), and It's Role in Checking Error. on-line at: `https://www.simplilearn.com/tutorials/networking-tutorial/what-is-cyclic-redundancy-check`.

[40] EduRev Education Revolution. Dependence Analysis. on-line at: `https://www.sciencedirect.com/topics/computer-science/dependence-analysis`.

[41] MIPS All rights reserved. MIPS Processor, RISC-V, Innovate Compute. on-line at: `https://mips.com/products/hardware/`.

[42] Samsung. All rights reserved. What is Bluetooth and how do I use it? on-line at: `https://www.samsung.com/uk/support/mobile-devices/what-is-bluetooth/`.

[43] Inc. All Rights Reserved Synopsys. What is RISC-V? on-line at: `https://www.synopsys.com/glossary/what-is-risc-v.html`.

[44] The KLEE Team. KLEE Symbolic Execution Engine. on-line at: `https://klee-se.org/`.

[45] The MLIR Team. MLIR - Multi-Level Intermediate Representation Overview. on-line at: `https://mlir.llvm.org/`.

[46] TechTarget. RAID (redundant array of independent disks). on-line at: `https://www.techtarget.com/searchstorage/definition/RAID`.

[47] Michigan State University. Cyclic Codes. on-line at: `https://users.math.msu.edu/users/halljo/classes/codenotes/Cyclic.pdf`.

Biografija autora

Petar Tešić je rođen 19. jula 1999. godine u Beogradu. Osnovnu školu i opšti smer gimnazije završio je u Ubu. Nakon završene gimnazije upisao je Matematički fakultet u Beogradu na smeru Računarstvo i informatika u okviru studijskog programa Matematika. Osnovne studije završio je u roku sa prosečnom ocenom 9,25. Po završetku osnovnih studija, upisao je master studije na Matematičkom fakultetu na smeru Informatika i postao saradnik u nastavi na Katedri za računarstvo i informatiku. Kao saradnik u nastavi bio je angažovan na sledećim kursevima: Programiranje 2, Uvod u relacione baze podataka, Prevođenje programskih jezika i Računarske mreže. Radio je dva meseca kao praktikant u kompaniji *FIS*. Nakon toga se priključio kompaniji *SYRMIA* kao član *System Software* tima gde je radio na zadatku predstavljenom u ovom radu, a takođe je bio angažovan i na internom projektu *Autocheck*. Od maja 2024. godine radi kao softverski inženjer u kompaniji *Cisco*.