Optimizacija potrebne programske memorije kod automatski generiranih testova za Autosar RTE

Vlašiček, Kristijan

SVEUČILIŠTE JOSIPA JURJA STROSSMAYERA U OSIJEKU
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA

Sveučilišni studij

OPTIMIZACIJA POTREBNE PROGRAMSKE MEMORIJE KOD AUTOMATSKI GENERIRANIH TESTOVA ZA AUTOSAR RTE

Diplomski rad

Kristijan Vlašiček

Osijek, 2020.

Sadržaj:

1. UVOD

Razvojem automobilske industrije povećao se broj elektroničkih komponenti u automobilima, a samim time i potreban broj upravljačkih programa. Kako automobilska industrija teži razvoju autonomnih vozila potrebno je osigurati prostora za još više elektroničkih komponenti, a tako i memorijskog prostora za instalaciju upravljačkih programa. U automobilskoj industriji izgradnja programske podrške temelji se na AUTOSAR standardu. Pri tome su korišteni ugradbeni računalni sustavi koji su unaprijed definirani s obzirom na broj sučelja između komponenata. Ugradbeni računalni sustavi su sustavi s ograničenim resursima (memorije, procesorska moć) i stoga je vrlo važno obratiti pozornost na zauzeće potrebne programske memorije, radne memorije, ali i tijek izvođenja programa zbog brzine izvođenja koja je ograničena procesorom. Kako bi se program mogao instalirati na ugradbeni sustav potrebno je da veličina programskog koda ne prelazi veličinu dostupne programske memorije pa je stoga potrebno optimizirati programski kod. Optimizacija je potrebna i kako bi se osigurao slobodan prostor u programskoj memoriji za potrebe naknadnih nadogradnji programa. Prilikom optimizacije programske memorije potrebno je pripaziti na zadržavanje performansi programskog koda jer se obično radi o sustavima koji rade u stvarnom vremenu, pa je potrebno izbjeći narušavanje sigurnosti ili funkcionalnosti programa.

Rad u okviru ovog diplomskog rada temelji se na referentnom testnom okruženju razvijenom u programskom jeziku Python, koje generira testni C kod za svako sučelje pojedinačno. U referentnom se okruženju C kod ponavlja za svaki test. Zbog memorijskih ograničenja ugradbenih računalnih sustava ovo može predstavljati problem jer za veliki broj sučelja generirani C kod može biti veći od raspoloživih resursa. Stoga je u okviru ovog diplomskog rada potrebno izvršiti analizu načina rada pojedinog testa za AUTOSAR RTE sučelje te na odgovarajući način modificirati C kod na način da se koriste generičke funkcije umjesto posebnih funkcija za svaki pojedinačni test. Nakon osmišljene optimizacije referentnog koda potrebno je modificirati postojeći Python alat kako bi generirao tako optimiziran C kod pri čemu je neophodno da se zadrži identična funkcionalnost referentnog koda.

U diplomskom radu su prvo obrađeni povezani radovi u kojima je pronađena motivacija za pojedine metode optimizacije programske memorije. Zatim je navedena primjena svake od odabranih metoda optimizacije, prvo na odabranoj programskoj komponenti, a zatim na čitavom skupu testnih programa pomoću TEG-a. Na kraju rada prikazani su rezultati optimizacije potrebne programske memorije, te su uspoređene statistike prolaznosti testova na testnoj ploči inicijalnog (referentnog) i optimiziranog programskog koda.

2. PREGLED LITERATURE I POSTOJEĆA RJEŠENJA

Pregledom i istraživanjem literature pronađeno je nekoliko metoda korištenih u svrhu optimizacije veličine programskog koda, odnosno optimizacije programske memorije. Nadalje, u ovom poglavlju su izdvojene knjige i znanstveni radovi koji su motivirali optimizaciju potrebne programske memorije koda razmatranog u sklopu ovog diplomskog rada.

Jedna od najučinkovitijih metoda uštede programske memorije jest pronalazak i uklanjanje dupliciranog programskog koda. Prema [1] i [2], duplicirani kod je česta pojava u velikim sustavima. Razlozi zbog kojih programeri dupliciraju kod su različiti. Jedan od razloga je izrada kopije fragmenata koda što je jednostavnije i brže od pisanja koda ispočetka. Drugi glavni razlog je zbog učinkovitosti programskog koda, kod se brže izvodi kada se ne pozivaju funkcije ili metode. U ovom radu, algoritam detekcije dupliciranog koda je proces u dva koraka. U prvom koraku se izvorni programski kod prevodi u interni format, osnovni znakovni niz (engl. *string*). U drugom koraku se nad dobivenim formatom koda provodi algoritam usporedbe i koristi se usporedba podudaranja znakovnog niza. Ovaj algoritam kao rezultat daje Booleovu istinitu vrijednost (engl. *boolean true*) ako je pronađena identična linija programskog koda, inače vraća Booleovu neistinitu vrijednost (engl. *boolean false*). Ta vrijednost se pohranjuje u matricu uzimajući koordinate koje uspoređeni entiteti imaju u svojim uređenim zbirkama kao koordinate matrice za rezultat usporedbe. Nakon ovog postupka matrica sadrži podudaranja pojedinih linija koda. Problem ovog algoritma je što ne označuje kod koji se razlikuje u jednoj liniji koda (problem kod funkcija), ali taj problem je riješen preko matrice koja bilježi dijagonalne linije i omogućuje toleranciju određene razlike. Nakon toga generira se tekstualni izvještaj koji sadrži uparene nizove pronađene provedbom algoritma za usporedbu, a uz to, naznačene su i točne lokacije dupliciranog programskog koda. Performanse ovog alata dovoljne su za izvorne programske kodove čiji zapis sadrži manje od milijun linija programskog koda (engl. *Million lines of code*, MLOC). Taj alat za detekciju dupliciranog programskog koda pomogao bi u optimizaciji programskog koda diplomskog rada, ali nije poželjno koristiti alate trećih strana zbog sigurnosti i privatnosti dodijeljenog programskog koda u fazi produkcije. Prema principu rada alata za detekciju dupliciranog koda, vizualnom analizom pronađeni su duplicirani nizovi linija programskog koda koji su kasnije optimizirani uvođenjem generičkih funkcija.

Prema radovima [3] i [4] razvijena je nova transformacija programskog koda koja štedi programsku memoriju. Umjesto da se ograničava na standardno uklanjanje točnih duplikata, uveden je pojam strukturne sličnosti funkcija na temelju kojih se omogućava spajanje sličnih

funkcija. Ključna ideja je da se dvije ili više sličnih funkcija spoje u jednu funkciju nadopunjenu odgovarajućim upravljačkim protokom kako bi se riješile razlike, dok se izvorne funkcije mogu ukloniti iz izvornog programskog koda. Gotovo identičan pristup je iskorišten prilikom optimizacije programske memorije programskog koda dobivenog u sklopu izrade diplomskog rada.

U knjizi [5] je navedeno kako je poželjno podijeliti programski kod u različite izvorne datoteke. Te datoteke su datoteke zaglavlja koje sadrže deklaracije funkcija koje trebaju biti dostupne u svim izvornim datotekama (kompilacijskim jedinicama) projekta. Iako je knjiga napisana za C++ programski jezik, datoteke zaglavlja koriste se i u C programskom jeziku, pa tako i u projektu koji je dobiven u sklopu diplomskog rada. Temeljem ovog naputka, neke funkcije su u diplomskom radu izdvojene u datoteke zaglavlja.

Još jedna od često korištenih metoda optimizacije veličine programskog koda jest pronalazak i uklanjanje mrtvog programskog koda. Prema [6], varijabla je „živa“ u programu sve dok se njena vrijednost može naknadno koristiti negdje u programskom kodu, a od trenutka kada se više neće koristiti dalje u kodu smatra se „mrtvom“. Povezana ideja je mrtav (ili beskorisni kod) koji predstavlja programski kod koji računa vrijednosti koje se nikad neće koristiti ili programski kod koji se nikad neće izvesti. Iako programer vjerojatno neće namjerno uvesti mrtvi programski kod, on se može pojaviti kao rezultat naknadnih transformacija i izmjena u programskom kodu. Prema [7], navedeno je da se mrtvi kod često pojavljuje kod automatski generiranih programskih kodova koji naglašavaju ponovnu upotrebu programskog koda. Ovakav slučaj je i kod inicijalnog programskog koda obuhvaćenog diplomskim radom nad kojim je napravljena optimizacija programske memorije. Razlog tome je automatsko generiranje programskog koda, stoga postoji više različitih blokova programskog koda koji se smatraju mrtvim kodom. Nadalje, prema [7], preporučuje se ostavljanje redundantnog programskog (mrtvog) koda u nepromijenjenom izvornom kodu i ostaviti optimizatoru da ga ukloni tijekom prevođenja (engl. *build*). U ovom diplomskom radu mrtvi programski kod je ipak uklonjen, jer se tako dobila ušteda potrebne programske memorije za razliku od slučaja kada se ostavi optimizatoru prevoditelja (engl. *compiler*) da optimizira mrtvi kod.

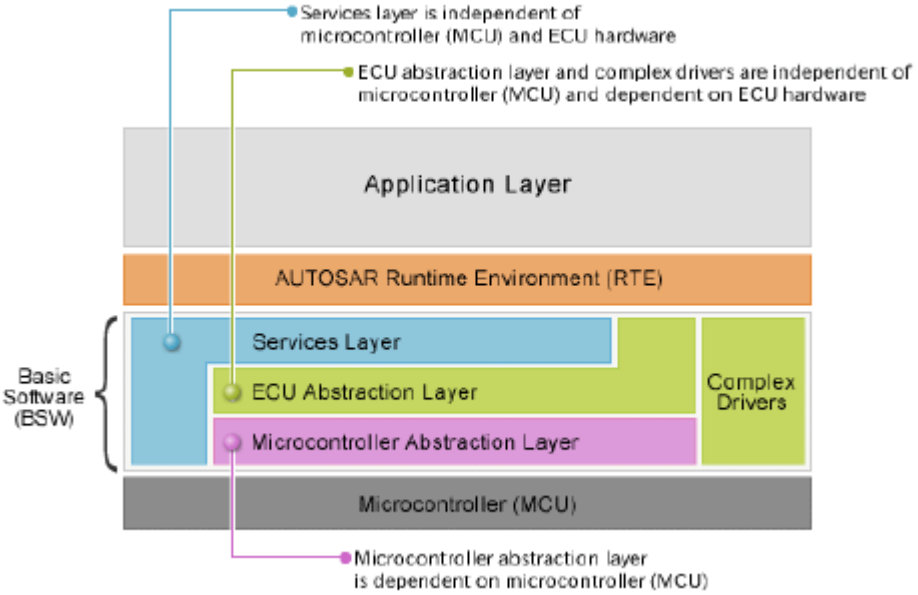
Prema [8], za uklanjanje mrtvog programskog koda u Java programskom jeziku pomoću Eclipse prevoditelja za Javu (engl. *Eclipse compiler for Java*, ECJ) predložen je pristup jedinstvene statičke podjele (engl. *Single Static Assingnment*, SSA). Nakon konverzije programskog koda u SSA, primjenjuje se lanac definicije korištenja (engl. *definition-use*, DU) na SSA, npr. pronalazi

se varijabla čija je definicija dana u programu, ali se ne koristi prilikom izvođenja programa. Rezultat ove strategije jest mrtvi kod, varijabla koja ima definiciju, ali se ne upotrebljava. Algoritam za eliminaciju mrtvog koda napisan je tako da mu je ulazni podatak program u SSA formi, a izlazni podatak mu je program s uklonjenim mrtvim kodom. Algoritam prolazi kroz svaku liniju programskog koda, te ako pronađe da linija programskog koda sadrži Phi funkciju, on ju briše. Phi funkcija se koristi kada se pojavljuje druga definicija iste varijable u čvoru spajanja, npr. y1=phi(y,y), gdje su y i y različite definicije iste varijable, a y1 je nova varijabla. Pri prepoznavanju zareza ili znaka jednakosti, algoritam rastavlja ostatak programskog koda na dijelove kako bi mogao analizirati element po element programskog koda. Zatim ako se u „check“ polju nalazi varijabla čiji je indeks manji od 0, ispisuje se poruka „Varijabla se ne koristi“. Ovaj pristup pridonosi kraćem vremenu prevođenja programskog koda, jer je provedenim algoritmom uklonjen mrtvi kod i time smanjen broj linija programskog koda za prevođenje. Također uklanjanjem mrtvog koda oslobođeni su registri koji su bili rezervirani za varijable koje se nisu koristile tijekom izvođenja programa i time se smanjilo zauzeće programske memorije. Iako je u spomenutom radu korišten algoritam koji pronalazi i uklanja mrtvi kod, nije ga moguće primijeniti na ovaj diplomski rad jer se ne koristi Eclipse prevoditelj, već drugi specifičan za dobiveni sustav. Prema principu rada spomenutog algoritma za eliminaciju mrtvog koda, ručno su uklonjeni dijelovi programskog koda u ovom diplomskom radu.

U radu [9] također je predstavljen model za izdvajanje mrtvog programskog koda iz programa tehnikom rezanja programa kako bi se povećala kvaliteta programskog koda. U predloženom modelu tehnika rezanja razgradnjom korištena je za izdvajanje aktivnog programskog koda i uklanjanje mrtvog koda. Nadalje, tehnika refaktoriranja korištena je za poravnanje programskog koda nakon uklanjanja mrtvog programskog koda. Kao rezultat ovog modela povećana je učinkovitost programa u smislu zauzeća memorije i vremena izvođenja, a osim toga, poboljšana je i preglednost koda nakon uklanjanja mrtvog programskog koda.

2.1. AUTOSAR RTE

Prema [10], AUTOSAR, odnosno *Automotive Open System Architecture*, je skup standarda osnovan od strane globalnog partnerstva proizvođača automobila, dobavljača, pružatelja usluga i kompanija iz automobilske industrije elektronike, poluvodiča i programske podrške. AUTOSAR ima svrhu standardizacije razvoja osnovnih funkcija sustava i funkcionalnih sučelja u automobilskoj industriji.



Slika 2.1. Arhitektura AUTOSAR-a, preuzeto sa [11]

Kako je prikazano na slici 2.1. arhitekturu AUTOSAR-a predstavljaju sljedeća sučelja: mikroupravljač (engl. *Microcontroller*, MCU), osnovna programska podrška (engl. *Basic Software*, BSW) koja ima nekoliko podslojeva, zatim sloj izvršnog okruženja (engl. *Run Time Environment*,RTE) i iznad njega aplikacijski sloj. RTE je jezgra arhitekture AUTOSAR ECU (engl. *Electronic Compute Unit*, ECU) i predstavlja realizaciju sučelja AUTOSAR-ove virtualne funkcionalne sabirnice (engl. *Virtual Functional BUS*, VFB). RTE pruža infrastrukturne usluge koje omogućuju komunikaciju između komponenti programske podrške (SWC-a) AUTOSAR-a. RTE također predstavlja vezu kojom komponente programske podrške pristupaju osnovnim modulima programske podrške (BSW-a), uključujući operacijski sustav (engl. *Operating System*, OS) i komunikacijske usluge. U principu, RTE se može logično podijeliti na komunikaciju između komponenti programske podrške (inter i intra ECU) i organizaciju komponenti programske podrške.

2.2. MotionWise

MotionWise jest platforma razvijena od strane TTTech grupacije u svrhu sigurnog upravljanja radom elektroničkih komponenata u automobilskoj industriji. Gotove programske komponente koje se implementiraju unutar ECU-a automobila nazivaju se *System build*. Prije nego programski kod uđe u produkciju, potrebno ga je testirati. Iz tog razloga je razvijena testna programska podrška, odnosno *Test build*. *Test build* ne ulazi u završni proizvod već je riječ o pojednostavljenom *System build*-u koji se koristi za provjeru sučelja kojima nije moguće pristupiti

konvencionalnim testnim metodama (dijagnostikom, *debugger*-om, itd.). Programski kod koji je potrebno optimizirati u sklopu ovog diplomskog rada jest *Test build* i napisan je na način da se za ponavljajuće blokove koda ne koriste funkcije. Takvi blokovi koda ponavljaju se i po više desetaka puta, a razlikuju se u samo nekoliko varijabli, što predstavlja potencijal za uštedu programske memorije. U programskim komponentama postoji više skupina takvih blokova koji služe za komunikaciju na ploči i obično rade u parovima kao što su *send* i *receive, read* i *write* ili *client* i *server*. Nadalje ovakvi se blokovi koda ponavljaju u svim programskim komponentama što znatno povećava potrošnju programske memorije. Kako se testovi provode na više segmenata, postoje generirana četiri testna programa, a to su *Interface* (ITF), *Canbus* (BUS), *Persistency* (PER) *i Platform Functions* (PFF). Na navedenim programima ponavljaju se već spomenuti parovi blokova programskog koda, čime se još više povećava potrošnja programske memorije, osim na PFF-u gdje ne postoje navedeni blokovi koda. Nadalje, sve te komponente generiraju se za dva *hosta* što dodatno povećava potrošnju programske memorije. *Hostovi* su *SafetyHost* (SH) i *PerformanceHost* (PH). Kako se na ITF testnom programu najviše ponavljaju takvi blokovi koda, tako za taj program postoji i potencijal za najviše uštede programske memorije, stoga se glavni fokus ovog rada sveo na optimizaciju programske memorije na tom testnom programu.

Daljnjom analizom koda, na ITF testnom programu se vidi da postoje sljedeći blokovi koda koji uvijek odrade isti posao, kao da se radi o funkcijama, a oni su označeni kao *mainRteRead, mainRteIRead, mainRteWrite, mainRteIWrite, mainRteIWriteRef, mainRteSend* i *mainRteReceive*. Zaključak je da se za svaki od navedenih blokova može napraviti jedinstvena generička funkcija kojoj bi se kao parametri predali samo oni dijelovi koda koji se razlikuju u odnosu na originalni kod u velikom broju ponavljajućih blokova. Takvim pristupom moglo bi se doći do značajne uštede programske memorije, slično kao što je navedeno u [3]. Dodatnom analizom koda uspostavilo se da postoje generirane funkcije i sučelja koja se u velikom broju slučajeva ne koriste tijekom izvođenja koda, te se korištenjem metode eliminacije mrtvog koda prema [6] može postići dodatna ušteda programske memorije. Također postoji nekoliko takvih sučelja i funkcija koje se ipak koriste, ali iz njih je moguće napraviti generičku funkciju kojom bi se uklonila mnogobrojna ponavljanja posebnih takvih funkcija i tako se postigla dodatna ušteda programske memorije. Sve komponente testova generirane su pomoću generatora testnog okruženja (engl. *Test Environment Generator*, TEG), alata razvijenog u Python-u.

2.3. Test Environment Generator (TEG)

Test Environment Generator (TEG) je Python alat napravljen od strane tvrtke TTTech koji služi za automatsko generiranje testnih okruženja potrebnih za testiranje platformi. Glavni zadatak TEG-a je da na temelju danih podataka ulaznog modela generira testne komponente, slučajeve za stresna testiranja i testne slučajeve. TEG se sastoji od datoteka napisanih u Python-u i CAPL datoteka koje služe za parsiranje i generiranje nekoliko testnih okruženja koja se izvode korištenjem CANoe-a. Postoje sljedeća četiri glavna testna okruženja. RTE sučelja (za internu komunikaciju), CAN (engl. *Controller Area Network*)/*FlexRay*/*Ethernet* (za vanjsku komunikaciju), *Persistency* i prilagođeni testovi na platformi (hardver, dijagnostika, praćenje...). Navedena testna okruženja jednakim redoslijedom podijeljena su kao sljedeći programski moduli: ITF, BUS, PER, PFF. TEG proces radi na takav način da prvo analizira baze podataka i poveže signalne sabirnice:

a) Pronađu se putanje u config.ini datoteci
b) Odrađuje se parsiranje podataka iz arxml-a, dbc-a i Fibex datoteka
c) Konfigurira se DataModel iz korisnički definiranih tablica.

Zatim se radi generiranje testnih komponenti:

a) Iz fragmenata u testne programske komponente (engl. *Software Component,* SWC)
b) Iz testnih podataka (*CommonTypes*) u testne slučajeve: ITF, BUS, PER, PFF
c) Iz testnih podataka (*CommonTypes*) u testni kod: signalne reference, testne uređene n-torke za svaki signal na sabirnici, TEG tipovi potrebni za CANoe i programsku podršku

Izlazni podaci iz TEG-a su popisi testnih slučajeva, C izvorne datoteke (SWC-i), CAPL izvorne datoteke i model testnog objekta.

3. OPTIMIZACIJA PROGRAMSKE MEMORIJE

U nastavku ovog poglavlja opisane su korištene metode optimizacije i način na koji su primijenjene u programskom kodu koji je potrebno optimizirati u sklopu ovog diplomskog rada. Nakon toga opisana je modifikacija postojećeg Python alata u svrhu integracije rješenja.

3.1. Korištene metode optimizacije

Kako bi se iskoristio navedeni potencijal iz drugog poglavlja za uštedu programske memorije korištene su sljedeće metode optimizacije programske memorije:

a) Generičke funkcije

Prilikom analize dobivenog koda, uočeno je da su mnogi dijelovi različitih sučelja slični po sastavu i funkcionalnosti, što omogućuje korištenje funkcija kao zamjenu za glavni dio programskog koda sučelja. Kako bi se uštedjela programska memorija, navedeni dijelovi koda su prilagođeni i predani generičkim funkcijama za potrebe optimizacije. Novonapisane funkcije su omogućile jasniju preglednost koda i njegove funkcionalnosti. Osim smanjenja potrošnje programske memorije, postignuta je modularnost koda, a time i lakše održavanje programskog koda (otkrivanje i ispravak pogrešaka).

b) Izdvajanje ponavljajućeg koda u datoteku zaglavlja

Zbog uvođenja novih funkcija u pojedinu programsku komponentu, dolazi do ponavljanja identičnih funkcija samo s različitim imenom funkcije što predstavlja redundantni kod. Zbog spomenute redundancije omogućena je dodatna optimizacija programske memorije tako da se identične funkcije izdvoje u posebnu datoteku zaglavlja s jednim imenom. Tako se izdvoje sve novonapisane funkcije, a redundantni kod se obriše, te se na taj način postiže dodatna ušteda memorije programskog koda. Osim toga, uočeno je ponavljanje određenih struktura koje je također moguće izdvojiti u datoteku zaglavlja zbog zajedničkog korištenja. Primjenom ove metode u slučaju potrebe za ispravljanjem pogrešaka programskog koda, potrebna je samo izmjena u funkciji. Bez izdvajanja funkcije u datoteku zaglavlja potrebna je izmjena u fragmentu i zatim ponovno generiranje programskih komponenti TEG-om.

c) Mrtvi kod

Osim spomenute transformacije navedenih sučelja u generičke funkcije, analizom rada programskog koda, zaključeno je da neke funkcionalnosti koda nikad nisu korištene.

Pronalaskom tih nekorištenih dijelova koda omogućuje se uklanjanje „mrtvog“ programskog koda, tj. sučelja koja se nikad ne koriste tijekom izvođenja sustava za testiranje.

3.2. Ručno pisani C kod

U svrhu očuvanja funkcionalnosti programskog koda, optimizacija programske memorije započinje modeliranjem programskog koda generirane programske komponente. Navedene metode optimizacije programske memorije primijenjene su na proizvoljno odabranoj programskoj komponenti. Ovakav pristup je odabran kako bi se očuvala funkcionalnost programskog koda te izbjeglo nepotrebno testiranje metoda optimizacije na svim programskim komponentama.

3.2.1. Generičke funkcije

U svrhu provedbe metode uvođenja generičke funkcije, prvo je ručno pisan C kod unutar proizvoljno odabrane programske komponente kako bi se provjerila funkcionalnost čitavog programa uvođenjem takve promjene. Tako je prvo odabran jedan blok koda unutar jedne komponente programske podrške te je od toga bloka napravljena funkcija, ali bez promjena varijabli jer se mijenja samo jedan blok koda u svrhu ispitivanja ispravnosti rada. Tako u kodu na mjesto zamijenjenog bloka dolazi poziv funkcije sa svim potrebnim argumentima, a sama deklaracija i definicija funkcije je napisana iznad glavnog (engl. *main*) programskog koda. Primjer primjene ove metode optimizacije može se vidjeti na slikama 3.1. i 3.2., gdje je na prvoj slici prikazan kod prije primjene metode optimizacije, a na drugoj slici je prikaz koda nakon primjene metode uvođenja generičkih funkcija. U referentnom programskom kodu, blok koda iz kojeg je izdvojena funkcija napisan je u 196 linija koda. U optimiziranom programskom kodu izdvojena funkcija s imenom funkcije i popisom argumenata sastoji se od 206 linija koda, dok je poziv funkcije napisan u 58 linija koda, te je poziv funkcije osnova uštede ove metode.

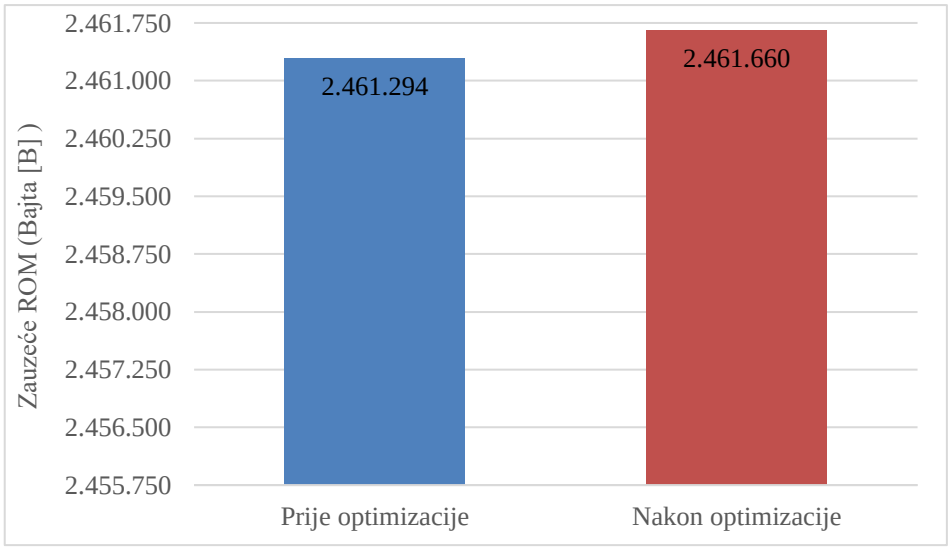
```
...
{
    structS.variableA = someOtherVariable;
    strictS.variableB = structS.variableA + variableVar1;
    variableVar2 = functionA(structS.variableB);
    printf("%d\n", variableVar2);
}

{
    structS2.variableA = someOtherVariable;
    strictS2.variableB = structS2.variableA + variableVar1;
    variableVar2 = functionA(structS2.variableB);
    printf("%d\n", variableVar2);
}

{
    structS3.variableA = someOtherVariable;
    strictS3.variableB = structS3.variableA + variableVar1;
    variableVar2 = functionA(structS3.variableB);
    printf("%d\n", variableVar2);
}
...
```

Slika 3.1. Primjer inicijalnog koda

```
...
int genericFunction(int *x, int *y){
    *x = someOtherVariable;
    *y = *x + variableVar1;
    variableVar2 = functionA(*y);
    printf("%d\n", variableVar2);
}
...
{
    genericFunction(&structS.variableA, &structS.variableB);
}
{
    genericFunction(&structS2.variableA, &structS2.variableB);
}
{
    genericFunction(&structS3.variableA, &structS3.variableB);
}
...
```

Slika 3.2. Primjer koda nakon primjene metode uvođenja generičke funkcije

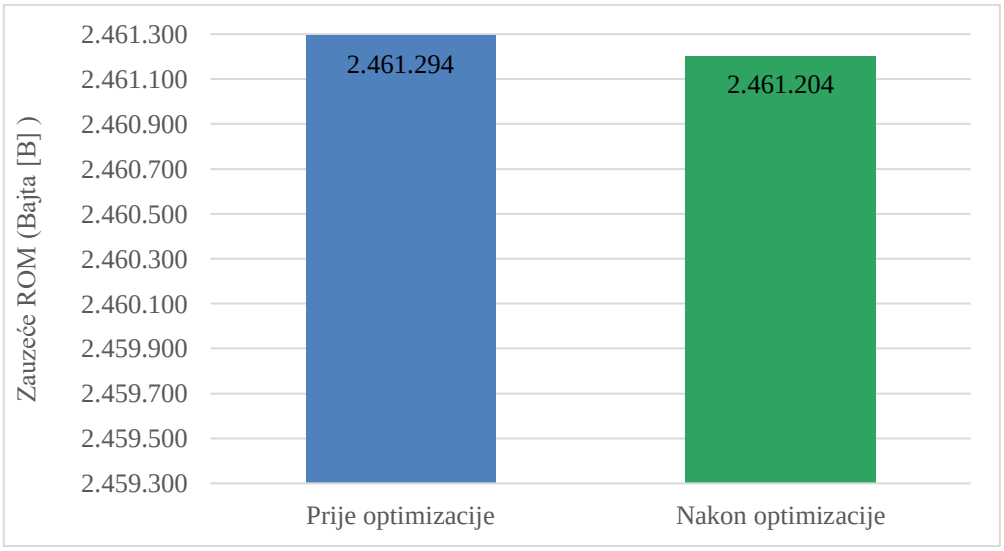
Ovakvim postupkom prilikom prevođenja koda povećalo se zauzeće programske memorije, ali tim postupkom se utvrdilo da uvođenje funkcija ne utječe na izvođenje sustava za testiranje. Na slici 3.3. se može vidjeti koliko se povećalo zauzeće memorije nakon što su dva bloka koda zamijenjena pozivom funkcije, a funkcija je napisana unutar same programske komponente. Iz ovakvog postupka gotovo da bi se moglo zaključiti da nema smisla uvoditi ovakve promjene u svrhu optimizacije programskog koda zbog rezultata koji prikazuju da dolazi do povećanja zauzeća programske memorije umjesto do njegovog smanjenja.



Slika 3.3. Zauzeće programske memorije prije i poslije uvođenja funkcije unutar jedne programske komponente, zamjena dva bloka koda

Nastavkom dodavanja poziva funkcija umjesto starih blokova koda, uz odgovarajuće predane parametre dolazi do promjene u potrošnji programske memorije. Već nakon zamjene pet blokova programskog koda, dobiva se pozitivan rezultat kao što se vidi na slici 3.4. Prve četiri zamjene blokova programskog koda nisu davale pozitivan rezultat zbog toga što su provedene izmjene u kodu kompenzirale novu nadodanu funkciju.

Nakon dobivenog pozitivnog rezultata uštede programske memorije, ali i samog izvođenja programa, slijedi integracija rješenja u TEG kako bi se jednostavno primijenila metoda optimizacije programske memorije na sve programske komponente.



Slika 3.4. Zauzeće programske memorije ITF testnog programa prije i poslije uvođenja funkcije unutar jedne programske komponente, zamjena pet blokova koda

3.2.2. Izdvajanje ponavljajućih generičkih funkcija u datoteku zaglavlja

Nakon provedene prethodne metode optimizacije na sve programske komponente ITF-a (poglavlje 3.3.1.), potrebno je ukloniti redundantne funkcije unutar svake programske komponente, odnosno koristi se metoda optimizacije navedena u poglavlju 3.1. pod b). Kako je prvonapisana funkcija za primjenu prve metode optimizacije napisana u 206 linija koda i to se ponavlja u 25 programskih komponenti, postoji veliki broj redundantnog koda koji je moguće ukloniti izdvajanjem te funkcije u datoteku zaglavlja gdje se nalaze ostale funkcije koje se koriste u više programskih komponenti. Provedbom ove metode optimizacije iz samih SWC-a se uklanja tih 206 linija koda, ali je potrebno modificirati poziv funkcije jer je zbog izdvajanja funkcije u zaglavlje potrebno predati neke nove argumente zbog njihove vidljivosti u programskom kodu. Osim toga, zbog smanjenja broja argumenata, ali ujedno i zbog dodatne uštede programske memorije, četiri strukture koje se nalaze unutar svakog SWC-a su premještene u datoteku zaglavlja gdje se nalaze deklaracije funkcija. Ovim postupkom se iz svake programske komponente uklanja 50 linija programskog koda. Nadalje prošireni poziv funkcije u programsku komponentu ubacuje dodatnih 30 linija koda, te je konačno poziv funkcije napisan u 61 liniji programskog koda. Također potrebna je i modifikacija same funkcije prebačene u datoteku zaglavlja, pa se tako prijašnjih 206 linija koda proširuje na ne tako veći broj, odnosno 209 linija koda. Modifikacije unutar funkcije su potrebne zbog uvođenja dodatnih argumenata, te se određene prijašnje varijable zamjenjuju varijablama iz popisa argumenata. Zbog prijenosa struktura u datoteku zaglavlja, nije moguće izgraditi programski kod za samo jednu programsku komponentu, već je potrebno koristiti TEG i primijeniti promjene na sve SWC-e. Tako nije moguće prikazati utjecaj ove metode unutar jedne programske komponente na zauzeće programske memorije, ali se mogu vidjeti rezultati kada se provede metoda na svim programskim komponentama u poglavlju 3.3.2.

3.2.3. Uklanjanje mrtvog koda

Nedostatak TEG-a je što određene dijelove koda zadano generira u svim SWC-ima, jer tijekom tog koraka generiranja nije poznato je li taj dio potreban ili ne. Za primjenu metode uklanjanja mrtvog koda glavnu ulogu ima TEG Python alat. Kako se TEG-om uklone sve funkcije koje se smatraju mrtvim kodom, ručno pisanim C jezikom se osigurava postojanje funkcije za slučajeve kada se ipak koriste neke od uklonjenih funkcija. U generiranom kodu postoji velik broj dupliciranih funkcija i one se smatraju mrtvim programskim kodom zbog toga što se u velikom broju slučajeva ne koriste. Tako je unutar prijašnjih funkcija napisanih u datoteci zaglavlja dodano novih šest linija programskog koda koje zamjenjuju funkcije mrtvog koda. Pomoću uvjetnog grananja i postavljene zastavice određeno je koja će se linija koda izvesti umjesto poziva

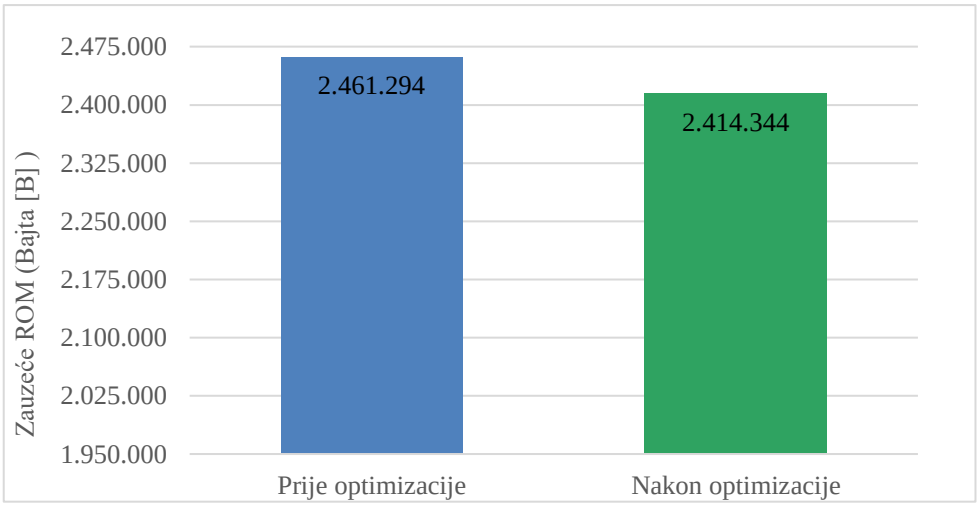
spomenute funkcije. Iako je proširen programski kod unutar funkcija iz datoteke zaglavlja, ovim postupkom nije narušena potrošnja programske memorije jer se tako omogućuje uklanjanje velikog broja definiranih funkcija i sučelja koji se smatraju mrtvim kodom. Tako je C jezikom riješen problem kada se pojedini dijelovi koda ipak koriste, a uklanjanje mrtvog koda provedeno je TEG-om i opisano u poglavlju 3.3.3.

3.3. Integracija rješenja u TEG Python alat

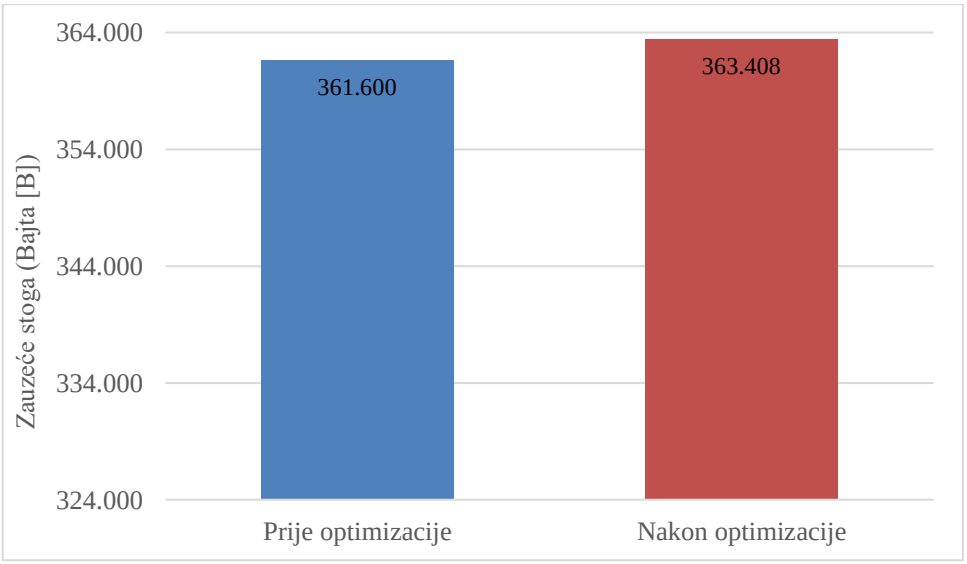
S obzirom na to da se testno okruženje sastoji od više programskih komponenata koje se generiraju pomoću TEG Python alata, zbog lakšeg uvođenja generičkih funkcija u sve programske komponente, potrebno je modificirati fragment u C jeziku iz kojeg TEG generira programske komponente.

3.3.1. Primjena generičke funkcije na sve SWC-e

U spomenutom fragmentu, na određenom mjestu u kodu piše se funkcija koja je prvobitno „ručno“ dodana u jednu programsku komponentu. Prilikom pokretanja TEG alata, napisana se funkcija generira u svim programskim komponentama, te ju je moguće koristiti pozivom i predajom potrebnih argumenata. Kako se dodana funkcija ponavlja u svakoj programskoj komponenti, dolazi do problema jer se radi o jednoj te istoj funkciji što stvara redundantni kod, a to nije optimalno za programsku memoriju. Drugi problem je što prevoditelj ne dozvoljava višestruko definiranje identičnih funkcija, stoga se u svakoj programskoj komponenti nalazi naziv funkcije koji sadrži ime SWC-a, kako prevoditelj ne bi javio grešku. Ovim postupkom dolazi do znatne uštede programske memorije iako je funkcija definirana unutar svake programske komponente, to jest, iako postoji redundantnost koda. Odnos programske memorije inicijalnog koda i koda nakon provedbe ove metode optimizacije vidi se na grafu sa slike Slika **3.5.** Može se vidjeti da se na ITF testnom programu zamjenom samo te jedne funkcije, uz ostavljeni redundantni kod ostvaruje ušteda od 1,91% što je dobar rezultat s obzirom na to da postoji sveukupno sedam tipova ovakvih blokova koda koji se mogu zamijeniti generičkim funkcijama. Ovom metodom optimizacije u programski kod se uvode nove funkcije koje se puno puta pozivaju što ima negativan utjecaj na memoriju stoga (engl. *stack*), čije se zauzeće zbog mnogobrojnih poziva funkcija povećava. Odnos memorije stoga prije i nakon primijenjene metode optimizacije vidljiv je na slici 3.6.

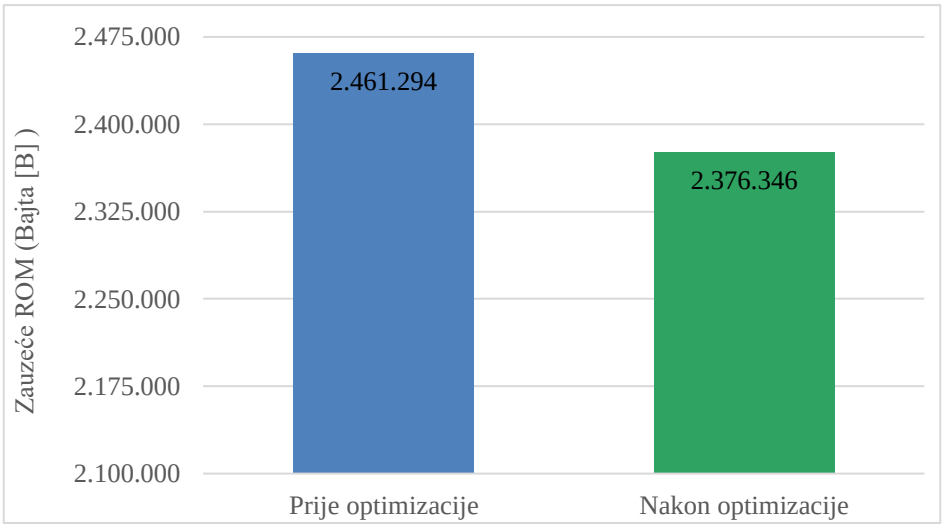


Slika 3.5. Stanje programske memorije na ITF testnom programu nakon generirane nove funkcije pomoću TEG-a

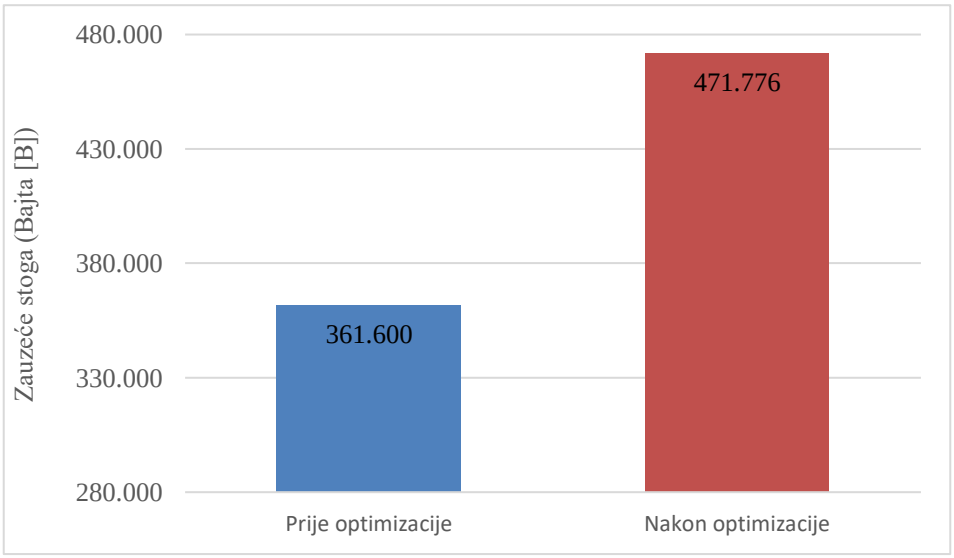


Slika 3.6. Zauzeće memorije stoga na programu sučelja (ITF)

Kako je ovakav postupak rezultirao uštedom prilikom optimizacije programske memorije, a uz to je ostala i nepromijenjena funkcionalnost koda, uvedene su generičke funkcije za preostalih šest vrsta ponavljajućih blokova koda. Novih šest generičkih funkcija pisano je odmah u C datoteci koju koristi TEG za generiranje testova kako bi se ubrzao postupak provođenja ove metode optimizacije na sve programske komponente. Generiranim testovima, nakon primjene metode uvođenja generičkih funkcija za svih sedam vrsta ponavljajućih blokova koda, smanjena je upotreba programske memorije za 3,45%, odnosno za gotovo 85 kB što se može vidjeti na slici 3.7. Kako je primjenom ove metode uvedeno šest novih funkcija unutar svake programske komponente, povećalo se i zauzeće memorije stoga. Na slici 3.8. vidi se povećanje zauzeća memorije stoga u odnosu na referentni kod za 110 kB, odnosno čak 30,47%.



Slika 3.7. Zauzeće programske memorije na ITF testnom programu nakon uvođenja svih 7 generičkih funkcija unutar SWC-a

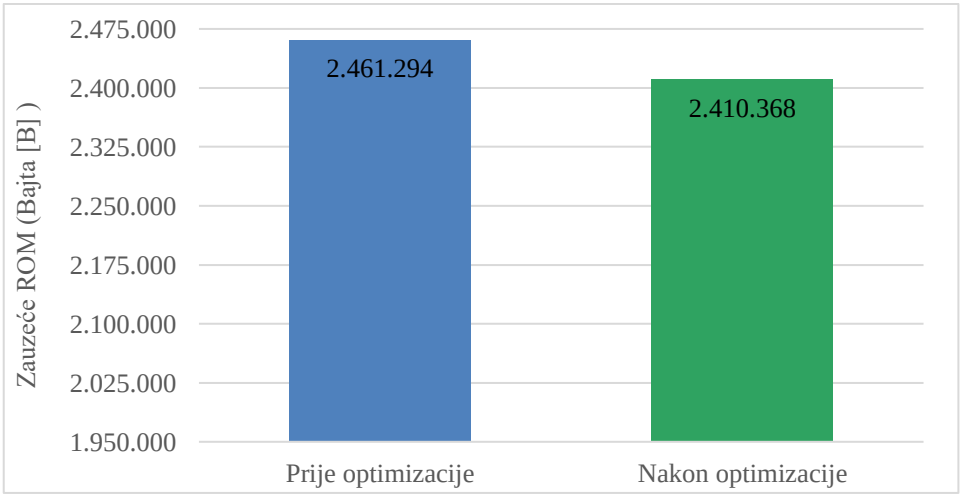


Slika 3.8. Zauzeće memorije stoga na ITF testnom programu nakon uvođenja svih 7 generičkih funkcija unutar SWC-a

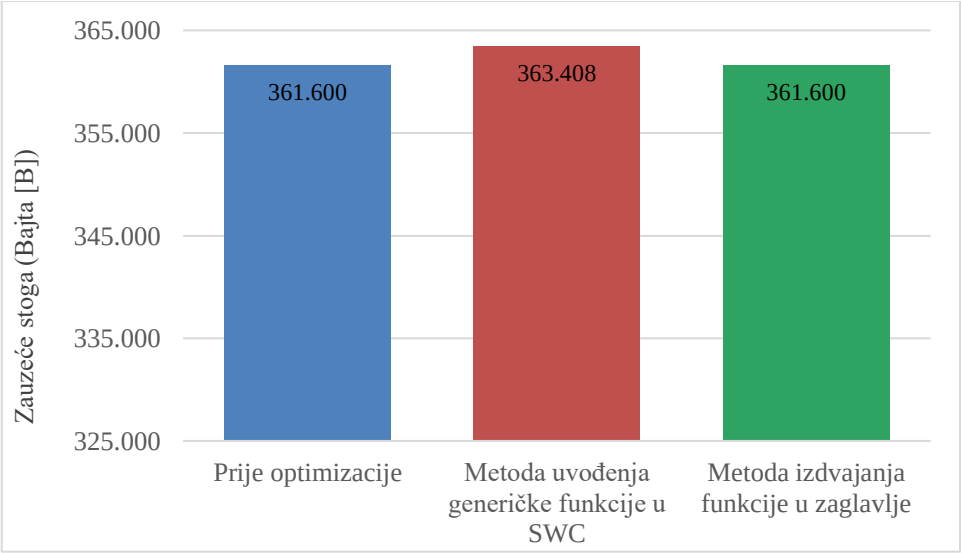
3.3.2. Generiranje SWC-a s izdvojenim strukturama i funkcijama u datoteku zaglavlja

Za provedbu druge metode optimizacije navedene u poglavlju 3.1. pod b) potrebno je ponovno preurediti ranije spomenuti fragment. Potrebno je ukloniti strukture koje su prebačene u datoteku zaglavlja što je spomenuto u poglavlju 3.2.2., te je potrebno ukloniti deklaracije i definicije funkcija napisanih prilikom provedbe prethodne metode optimizacije. Osim navedenog potrebno je izmijeniti pozive funkcija zbog promjene imena funkcije i dodatnih argumenata koje funkcija prima. Nakon provedbe navedenoga, na slici 3.9. se može vidjeti veća ušteda programske memorije u odnosu na prethodnu metodu (Slika 3.5.) za 0,16%, tj. gotovo 4 kB više. Ukupno je

izdvajanjem jedne generičke funkcije u datoteku zaglavlja uštađeno 50,926 kB programske memorije što iznosi 2,07%. Osim dodatne uštede programske memorije, primjenom metode izdvajanja funkcija u datoteku zaglavlja smanjeno je zauzeće stoga u odnosu na metodu uvođenja generičkih funkcija u svakoj programskoj komponenti. Usporedba zauzeća stoga se može vidjeti na grafu sa slike Slika **3.10.**, gdje je vidljivo da je zauzeće memorije stoga jednako kao u slučaju prije optimizacije.



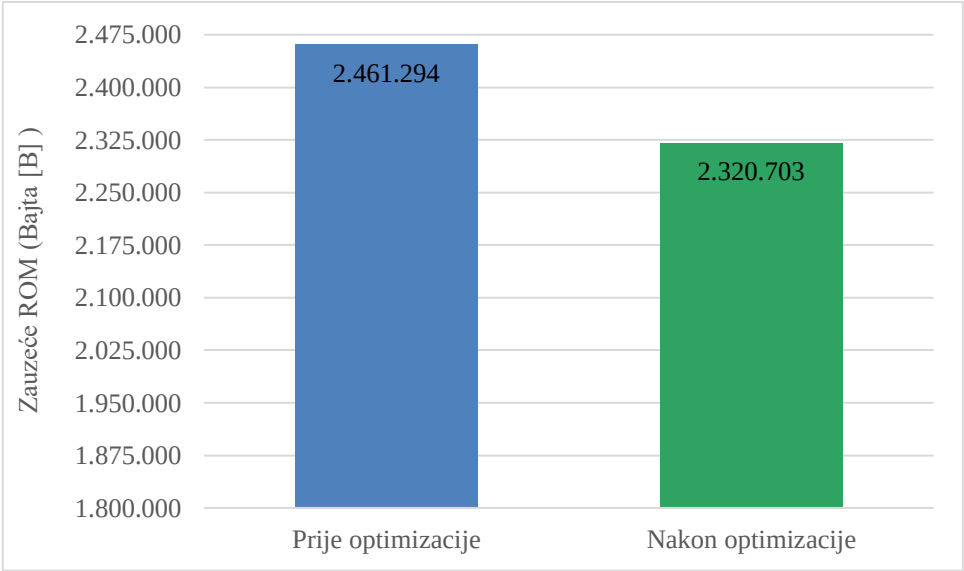
Slika 3.9. Stanje programske memorije nakon generirane nove funkcije u zaglavlju pomoću TEG-a



Slika 3.10. Zauzeće memorije stoga na programu sučelja (ITF)

Nakon što se utvrdilo da primjena metode izdvajanja funkcija u datoteku zaglavlja daje pozitivan rezultat kod uštede programske memorije, bez narušavanja funkcionalnosti programskog koda, ova metoda primijenjena je i na ostale generičke funkcije. U C datoteci iz koje se generiraju programske komponente, prepravljeni su pozivi za svaku funkciju, jer izdvajanjem funkcija u datoteku zaglavlja, funkciji je potrebno predati dodatne podatke kao argumente. Također je svaka

napisana generička funkcija izbrisana iz te C datoteke kako se ne bi više generirala u programskim komponentama. Funkcije su izdvojene u datoteku zaglavlja, ali u odnosu na prvobitno napisane, ispravljen je popis argumenata zbog varijabli koje je potrebno dodatno predati funkciji. Nakon što su generirane programske komponente i nakon što je izgrađen programski kod C prevoditeljem, dobiveni su novi pozitivni rezultati. Kao što se vidi na slici 3.11., u odnosu na referentni programski kod uštađeno je 140,591 kB programske memorije, što je ušteda od 5,71%.

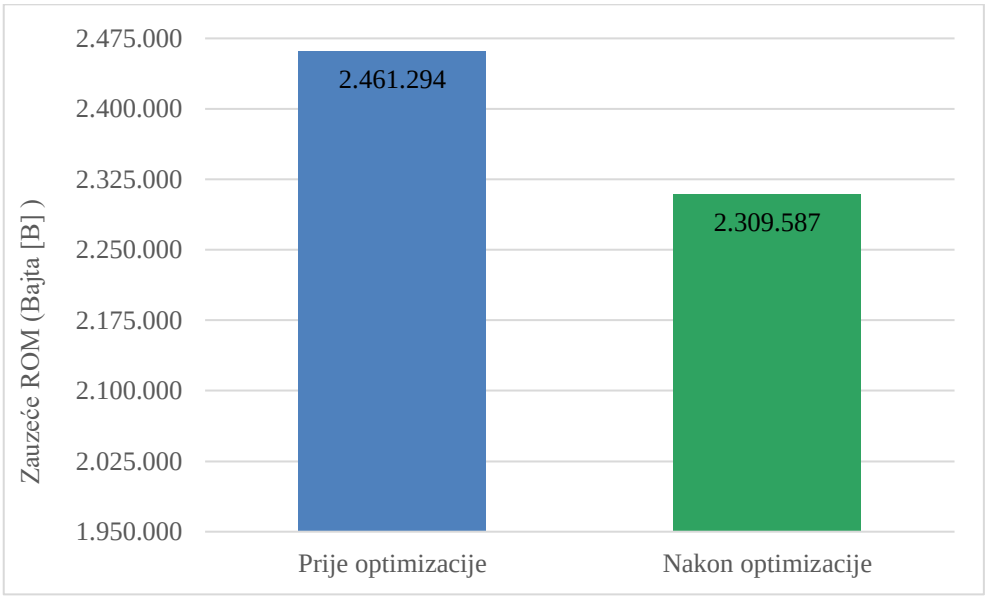


Slika 3.11. Stanje programske memorije nakon što su generirane sve nove funkcije u datoteci zaglavlja pomoću TEG-a

3.3.3. Uklanjanja mrtvog koda TEG Python alatom

Za uklanjanje mrtvog programskog koda iz programskih komponenti potrebno je izmijeniti svojstvene Python datoteke koje određuju što će se sve generirati pokretanjem TEG alata. Tako je unutar određene svojstvene datoteke pronađen naziv sučelja i funkcija koje se smatraju mrtvim programskim kodom, te su uklonjeni ti nazivi iz određenog polja koje TEG-u određuju što je potrebno generirati. Nakon pokretanja TEG Python alata generirane su programske komponente u kojima više nema sučelja i funkcija koje se ne koriste u velikom broju slučajeva. Tako je programskim komponentama broj linija koda smanjen za 400 do 4000 linija programskog koda, ovisno o kojoj se programskoj komponenti radi. Samom usporedbom generiranih programskih komponenti po broju linija programskog koda vidi se da je primjena ove metode donijela pozitivan rezultat kod uštede programske memorije, te da izmjena funkcija iz datoteka zaglavlja opisana u poglavlju 3.2.3. nema negativan utjecaj na uštedu. Primjena ove metode na ITF testni program povećala je uštedu programske memorije na 6,16%, odnosno kao što se vidi na slici 3.12., zauzeće programske memorije smanjeno je za 151 kB u odnosu na referentni programski kod. Kako je

metoda uklanjanja mrtvog programskog koda primijenjena nakon što su funkcije izdvojene u datoteku zaglavlja, nema promjene u zauzeću memorije stoga, već je ona jednaka kao i kod referentnog početnog koda.



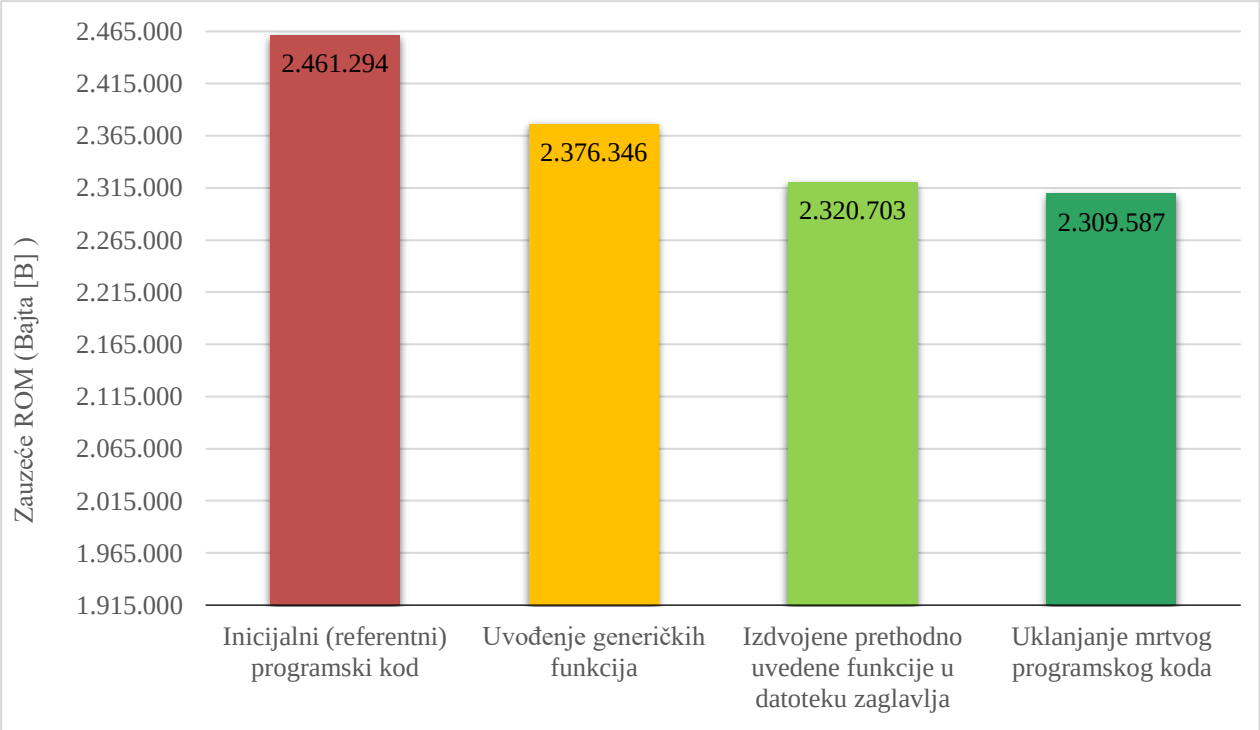
Slika 3.12. Usporedba zauzeća programske memorije početnog koda i nakon primjene metode uklanjanja mrtvog programskog koda, na prethodno provedene metode optimizacije, na ITF testnom programu

4. EKSPERIMENTALNA ANALIZA

U ovom poglavlju opisani su konačni rezultati uštede programske memorije za ITF testni program. Kako bi rezultati testova bili zadovoljavajući, potrebno je izgrađeni programski kod učitati na razvojnu ploču te proći sve potrebne testove. U drugom djelu poglavlja prikazani su rezultati uštede programske memorije na ostalim testnim programima. Ostali testni programi nisu testirani na ploči, stoga se ne može sa sigurnošću zaključiti narušavaju li primijenjene metode optimizacije zahtijevane funkcionalnosti programskog koda.

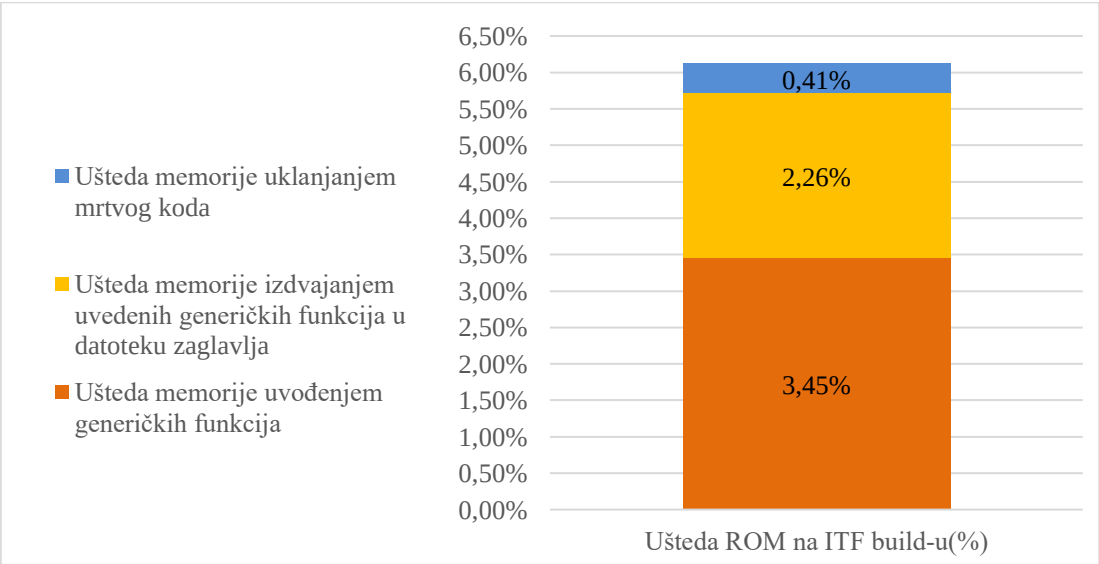
4.1. Rezultati uštede programske memorije na ITF testnom programu

Primjene metode optimizacije programske memorije uvođenjem generičkih funkcija i izdvajanjem funkcija u datoteke zaglavlja usko su povezane, te zajedno daju zadovoljavajući rezultat. Metoda izdvajanja funkcija u datoteke zaglavlja primijenjena je na prvu metodu uvođenja generičkih funkcija, te su nove funkcije tom metodom izdvojene u datoteku zaglavlja. Druga metoda optimizacije vratila je zauzeće memorije stoga na inicijalno stanje jer su izbačene mnoge funkcije iz samih programskih komponenti. Samim time ubrzano je izvođenje programskog koda i smanjeno je dodatno opterećenje stoga koje se uvelo primjenom prve metode optimizacije. Metoda uklanjanja mrtvog koda potpuno je neovisna o prve dvije metode, ali je značajno optimizirala memoriju programskog koda. Na slici 4.1. prikazano je početno zauzeće programske memorije, te konačni rezultati uštede programske memorije svake od primijenjenih metoda, ali svaka sljedeća metoda optimizacije nadovezuje se na prethodno primijenjenu.



Slika 4.1. Prikaz zauzeća programske memorije nakon primjene metoda optimizacije nadovezanih jedne na drugu

Sa slike 4.1. se može uočiti da je u konačnici na ITF testnom programu uštađeno sveukupno 151,707 kB programske memorije. Ušteda programske memorije u postotcima prikazana je na slici 4.2. gdje je vidljivo da je ukupno zauzeto 6,16% manje programske memorije u odnosu na referentni programski kod. Na slici 4.2. također je u postotcima prikazana ušteda programske memorije kada se svaka od korištenih metoda optimizacije samostalno primjeni na referentni programski kod. Sa slike se može zaključiti da je metoda uvođenja generičkih funkcija najviše pridonijela prilikom uštede programske memorije.



Slika 4.2. Prikaz uštede programske memorije pojedine metode u postotcima, te ukupna ušteda

Na slici 4.3. prikazana je statistika prolaznosti testova na testnoj ploči inicijalnog, odnosno referentnog programskog koda, dok je na slici 4.4. prikazana statistika prolaznosti optimiziranog programskog koda. Prvo polje na slikama prikazuje broj postojećih testova, drugo polje prikazuje koliko je testova provedeno na testnoj ploči, dok treće polje prikazuje broj testova koji se nisu izveli na ploči. Na slikama je zelenom bojom označen broj testnih slučajeva koji su uspješno i bez pogreške prošli na testnoj ploči, dok je crvenom bojom označen broj testnih slučajeva koji su iz nekog razloga računalu koje komunicira s testnom pločom prijavili neku vrstu pogreške. Iz prikazanih podataka može se vidjeti da nakon primjene metoda optimizacije uspješno prolazi 90% izvedenih testnih slučajeva. Na temelju toga može se zaključiti da je primjenom navedenih metoda optimizacije u velikoj mjeri očuvana funkcionalnost programskog koda.

Statistics

Overall number of test cases	185	
Executed test cases	185	100% of all test cases
Not executed test cases	0	0% of all test cases
Test cases passed	177	96% of executed test cases
Test cases failed	8	4% of executed test cases

Slika 4.3. Statistika prolaznosti testova na testnoj ploči inicijalnog (referentnog) programskog koda

Statistics

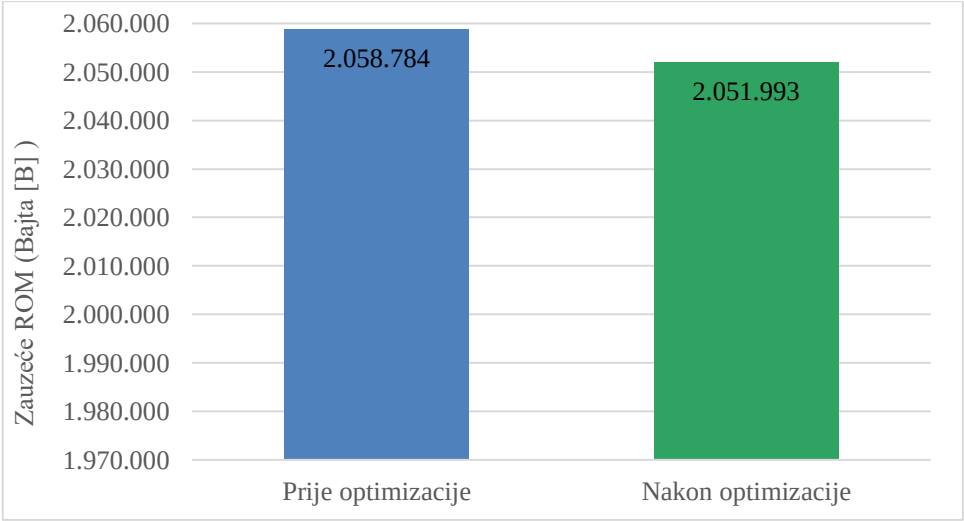
Overall number of test cases	185	
Executed test cases	185	100% of all test cases
Not executed test cases	0	0% of all test cases
Test cases passed	166	90% of executed test cases
Test cases failed	19	10% of executed test cases

Slika 4.4. Statistika prolaznosti testova na testnoj ploči optimiziranog programskog koda

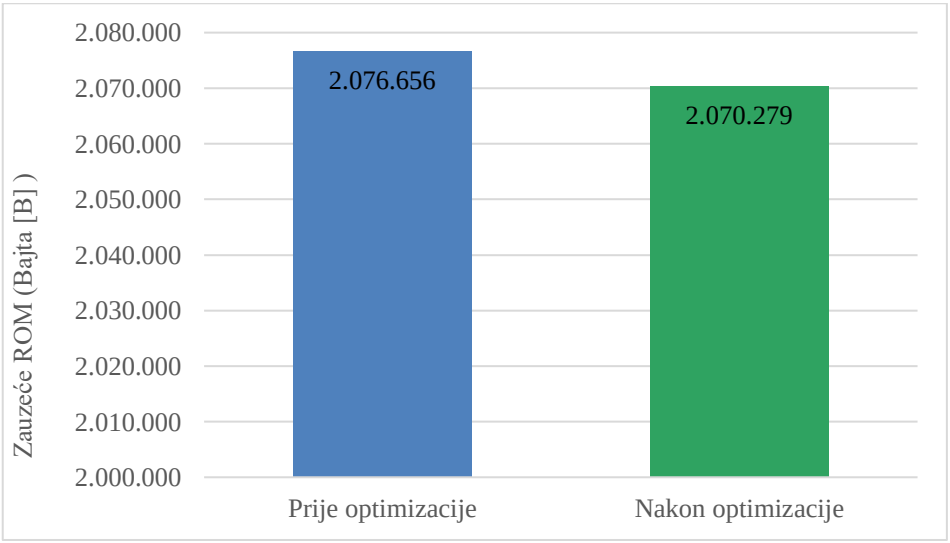
4.2. Ušteda programske memorije ostalih testnih programa

Sve provedene metode na ITF testnom programu primijenjene su također na BUS i PER testnim programima uz odgovarajuće izmjene jer svaki testni program ima svoje određene posebnosti. Primjene spomenutih metoda optimizacije programske memorije dale su zadovoljavajuće rezultate uštede potrebne programske memorije. Kako dobiveni referentni programski kod nije bio u distribucijskoj (engl. *release*) verziji, prilikom testiranja programskih kodova, referentnih i optimiziranih, nisu postojali potrebni podaci na sabirnicama, te nije bilo moguće provesti testiranje programskih kodova na testnoj ploči. Iako nije provedeno testiranje kodova na testnoj ploči, postoje rezultati primijenjenih metoda uštede programske memorije koji bi se mogli u novom radu primijeniti na distribucijskoj verziji koda. Tada bi se moglo provjeriti jesu li primijenjene metode

uz zadovoljavajući rezultat uštede programske memorije ostvarile i zadovoljavajući rezultat funkcionalnosti optimiziranih programskih kodova. Dobiveni konačni rezultat uštede potrebne programske memorije za BUS testni program prikazan je na slici 4.5., gdje se vidi da je ušteđeno 7 kB potrebne programske memorije što u postotcima iznosi 0,33%. Za PER testni program zauzeće programske memorije prije i nakon optimizacije prikazano je na slici 4.6., te se može vidjeti da je primjenom istih metoda ušteđeno 6 kB programske memorije što u postotcima iznosi 0,31%.



Slika 4.5. Prikaz zauzeća programske memorije za BUS testni program referentnog i optimiziranog koda



Slika 4.6. Prikaz zauzeća programske memorije za PER testni program referentnog i optimiziranog koda

Kod BUS i PER testnog programa značajno je manja ušteda potrebne programske memorije u odnosu na ITF testni program zbog toga što se na ITF testnom programu koristi puno više već spomenutih ponavljajućih blokova koda kojih nema toliko puno u programskom kodu na preostala dva testna programa.

5. ZAKLJUČAK

U okviru diplomskog rada dobiven je programski kod koji je napisan na način da ima što bolje performanse izvođenja programa što ostavlja prostora za optimizaciju programske memorije. Zbog toga, a i zbog činjenice da je programski kod namijenjen za ugradbene sustave, na njemu je provedena optimizacija programske memorije.

Optimizacija programske memorije kod automatski generiranih testova za AUTOSAR RTE provedena je u dva koraka i pritom su se koristile tri metode optimizacije. U prvom koraku provedena je optimizacija programske memorije izmjenom izabrane programske komponente gdje je ručno prepravljen kod napisan u C programskom jeziku. Ovaj korak proveden je za svaku od primijenjenih metoda u svrhu testiranja utjecaja primijenjene metode optimizacije na performanse i ispravnost izvođenja programskog koda. U drugom koraku korišten je TEG Python alat pomoću kojeg su automatski generirane sve programske komponente za svaki testni program. Za TEG je potrebno izmijeniti fragment koda napisan u C programskom jeziku pomoću kojeg se generiraju sve programske komponente te se tako olakšava primjena metoda optimizacije na njih.

U dobivenom referentnom programskom kodu prvo su pronađeni slični ponavljajući blokovi programskog koda koji su napisani na takav način zbog postizanja boljih performansi izvođenja programa. Na temelju istražene literature na ove blokove programskog koda primijenjena je metoda uvođenja generičkih funkcija koja sve te ponavljajuće blokove koda zamjenjuje samo pozivima generičkih funkcija. Primjenom ove metode na sve programske komponente TEG-om postignuta je znatna ušteda potrebne programske memorije, ali kako je tim postupkom svaka generička funkcija ponovljena u svakoj programskoj komponenti, znatno se povećalo zauzeće memorije stoga zbog velikog broja poziva „različitih“ funkcija tijekom izvođenja programa. Te su funkcije identične, ali kako su definirane u svakoj programskoj komponenti, samo su različitog naziva i time se smatraju različitim funkcijama. Vezano na prvu metodu optimizacije veličine programskog koda, primijenjena je druga metoda optimizacije, a to je izdvajanje funkcija u datoteku zaglavlja. Provedbom druge metode, sve identične funkcije iz različitih programskih komponenti spojene su u jednu funkciju u datoteci zaglavlja, te se time dodatno smanjilo zauzeće programske memorije, a osim toga i zauzeće memorije stoga. U konačnici, neovisno o prve dvije metode optimizacije, primijenjena je posljednja metoda optimizacije veličine programskog koda, uklanjanje mrtvog programskog koda. Dodatnom analizom koda, kao i kod svakog većeg projekta, pronađeni su dijelovi koda (funkcije i varijable) koji se nikada ne koriste prilikom izvođenja programa. Primjenom ove metode ti su dijelovi programskog koda uklonjeni i time je dodatno

umanjeno zauzeće programske memorije. Primjena svake od metoda optimizacije programske memorije provedena je za tri od četiri testna programa uz male izmjene zbog posebnosti svakog testnog programa. U konačnici na ITF testnom programu zauzeće potrebne programske memorije smanjeno je za 6,16%, na BUS testnom programu za 0,33% i na PER testnom programu za 0,31%.

Optimizacija programske memorije zaključena je provedbom testova na testnoj ploči. Testiran je optimizirani ITF testni program, te su rezultati testiranja zadovoljavajući, odnosno bez nedozvoljenog odstupanja od rezultata testiranja referentnog programskog koda. Time se može zaključiti kako je primjena navedenih metoda optimizacije uspješno smanjila zauzeće potrebne programske memorije, točnije za 6,16%, uz zadovoljavajuće očuvanje funkcionalnosti programa. Kako dobiveni programski kod nije bio u distribucijskoj verziji, ostala dva testna programa nisu instalirana na testnu ploču, te nisu provedeni testovi. Kako bi se potvrdila ušteda programske memorije na njima, potrebno je primijeniti spomenute metode optimizacije na distribucijskoj verziji programa i obaviti testiranje na testnoj ploči. Provođenjem testova potvrdilo bi se je li primjenom provedenih metoda optimizacije narušena funkcionalnost programskog koda kod BUS i PER testnih programa i mogu li se koristiti u produkciji nakon primjene metoda optimizacije veličine programskog koda.

LITERATURA

[1] S. Ducasse, M. Rieger, S. Demeyer, A language independent approach for detecting duplicated code, Proceedings IEEE International Conference on Software Maintenance - 1999 (ICSM'99). 'Software Maintenance for Business Change' (Cat. No.99CB36360), str. 109-118, Oxford, Engleska, kol. 1999.

[2] M. Rieger, S. Ducasse, Visual Detection of Duplicated Code, Lecture notes in computer science, European COnference On Programming languages (ECOOP) Workshops, sv. 1543, Bruxelles, Belgija, srp. 1998.

[3] T. J. K. Edler von Koch, B. Franke, P. Bhandarkar, A. Dasgupta, Exploiting Function Similarity for Code Size Reduction, Proceedings of the 2014 SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems, br. 5, sv. 49, str. 85-94, Edinburgh, Velika Britanija, lip. 2014.

[4] K. Hotta, Y. Sasaki, Y. Sano, Y. Higo, S. Kusumoto, An Empirical Study on the Impact of Duplicate Code, Advances in Software Engineering, br. 5, sv. 2012, Osaka, Japan, sij. 2012.

[5] M. Olsson, C++20 Quick Syntax Reference: A Pocket Guide to the Language, APIs, and Library, Apress, 2020.

[6] A. V. Aho, R. Sethi, J. D. Ullman, M. S. Lam, Compilers: Principles, Techniques, and Tool(2 ed.)., Pearson, Boston, Massachusetts, SAD, 2006.

[7] N. E. Johnson, Code size optimization for embedded processors, Cambridge, Velika Britanija, 2004.

[8] H. H. Karer, P. B. Soni, Dead Code Elimination Technique in Eclipse Compiler for Java, 2015 International Conference on Control, Instrumentation, Communication and Computational Technologies (ICCICCT), str. 275-278. Kumaracoil, Indija, pro. 2015.

[9] N. AlAbwaini, A. Alda`aje, T. Jaber, M. Abdallah, A. Tamimi, Using Program Slicing to Detect the Dead Code, 2018 8th International Conference on Computer Science and Information Technology (CSIT), str. 230-233, Amman, Jordan, 2018.

[10] AUTOSAR, [online], AUTOSAR, dostupno na: https://www.autosar.org/fileadmin/ABOUT/AUTOSAR_EXP_Introduction.pdf [7.7.2020.]

[11] Renesas, [online], Renesas Electronics Corporation, dostupno na: https://www.renesas.com/sg/en/solutions/automotive/technology/autosar/autosar-layered-architecture.html [8.7.2020.]

SAŽETAK

Tehnološkim napretkom u automobilskoj industriji javlja se potreba za sve većim brojem elektroničkih komponenti i odgovarajućih upravljačkih programa, odnosno programske podrške. Izgradnja programske podrške temelji se na AUTOSAR standardu, dok se za potrebnu elektroniku koriste ugradbeni računalni sustavi. S obzirom na to da ugradbeni računalni sustavi imaju ograničene resurse, potrebno je optimizirati programsku podršku. U sklopu diplomskog rada optimizirana je potrebna programska memorija dobivene programske podrške. Prilikom optimizacije programske memorije korištene su sljedeće metode optimizacije: uvođenje generičkih funkcija, izdvajanje ponavljajućih funkcija u datoteku zaglavlja i uklanjanje mrtvog koda. Metode optimizacije primijenjene su na način da je prvo izmijenjen C programski kod i tako se ručno primijenila metoda na odabranoj programskoj komponenti. Zatim je testirano očuvanje funkcionalnosti tako optimiziranog programskog koda. Kada je potvrđeno očuvanje funkcionalnosti programa, izmijenjene su potrebne datoteke kako bi se TEG-om metoda optimizacije programske memorije primijenila na sve programske komponente (SWC-e). Zatim je odrađeno testiranje programskog koda na testnoj ploči te je dobivena statistika prolaznosti testova koji potvrđuju funkcionalnost programskog koda nakon optimizacije. Primijenjene metode su optimizirale zauzeće programske memorije (smanjenje od 6.16%) i u odgovarajućoj mjeri su očuvale funkcionalnost programskog koda.

Ključne riječi: AUTOSAR, ugradbeni računalni sustavi, optimizacija, programska memorija, TEG, programske komponente (SWC), programski jezik C

ABSTRACT

Program memory optimization for automatically generated tests for AUTOSAR RTE

With the technological advancements in the automotive industry, the need for more electronic components has increased as well as for the appropriate software, that is, appropriate drivers. The development of such software is based on the AUTOSAR standard, while the required electronics use embedded computer systems. Considering that the embedded computer systems have limited resources, the software needs to be optimized. In this paper, the program memory of the obtained software was optimized. The optimization was done using the following methods: introduction of generic functions, extraction of repetitive functions in the header files and dead code elimination. The optimization methods were applied by changing a program code written in C programming language. In that way, the method was applied manually on a selected software component (SWC). Then, the preservation of functionality of such optimized program code was tested. When the preservation of functionality was confirmed, necessary files were altered so that the method of optimization the program memory could be applied to all software components (SWCs) using TEG. Afterward, the program code testing was done on a testing board. The results of the testing provided the test passing statistics which prove the functionality of the optimized program code. The applied methods optimized the memory usage (6.16% reduction) and preserved the functionality of the program code to an appropriate extent.

Key words: AUTOSAR, embedded systems, optimization, program memory, TEG, software components (SWCs), C programming language

ŽIVOTOPIS

Kristijan Vlašiček rođen je 29. ožujka 1997. godine u Virovitici, Republika Hrvatska. Živi u obiteljskoj kući u Sedlarici. Od 2003. godine pohađao je Osnovnu školu Petra Preradovića u Pitomači. Nakon završene osnovne škole, 2011. godine upisuje Strukovnu školu Đurđevac u Đurđevcu, smjer tehničar za računalstvo. U drugom razredu srednje škole sudjeluje na državnom natjecanju iz elektrotehnike i mjerenja u elektrotehnici. Srednju školu završava 2015. godine te iste godine upisuje Preddiplomski sveučilišni studij elektrotehnike na Fakultetu elektrotehnike, računarstva i informacijskih tehnologija, pri Sveučilištu J. J. Strossmayera u Osijeku. Preddiplomski sveučilišni studij završava 2018. godine i stječe zvanje sveučilišni prvostupnik (baccalaureus) inženjer elektrotehnike i informacijske tehnologije, zatim iste godine upisuje Diplomski sveučilišni studij Mrežne tehnologije na istom fakultetu.