Optimizacija potrošnje RAM memorije kod automatski generiranih testova za Autosar RTE

Bartulović, Ante

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

Sveučilišni studij

OPTIMIZACIJA POTROŠNJE RAM MEMORIJE KOD AUTOMATSKI GENERIRANIH TESTOVA ZA AUTOSAR RTE

Diplomski rad

Ante Bartulović

Osijek, 2020.

FERIT
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA OSIJEK

Obrazac D1: Obrazac za imenovanje Povjerenstva za diplomski ispit

Osijek, 22.09.2020.

Odboru za završne i diplomske ispite

Imenovanje Povjerenstva za diplomski ispit

Ime i prezime studenta:	Ante Bartulović
Studij, smjer:	Diplomski sveučilišni studij Računarstvo
Mat. br. studenta, godina upisa:	D-969R, 23.09.2019.
OIB studenta:	05730958398
Mentor:	Doc.dr.sc. Zdravko Krpić
Sumentor:	Dr.sc. Ivan Vidović
Sumentor iz tvrtke:	Zoran Bartulović
Predsjednik Povjerenstva:	Izv. prof. dr. sc. Marijan Herceg
Član Povjerenstva 1:	Doc.dr.sc. Zdravko Krpić
Član Povjerenstva 2:	Izv. prof. dr. sc. Mario Vranješ
Naslov diplomskog rada:	Optimizacija potrošnje RAM memorije kod automatski generiranih testova za Autosar RTE
Znanstvena grana rada:	**Programsko inženjerstvo (zn. polje računarstvo)**
Zadatak diplomskog rada:	U automobilskoj industriji svi ugradbeni računalni sustavi zasnovani na Autosar standard su unaprijed definirani s obzirom na broj sučelja između komponenti. Broj sučelja može biti i preko 1000, a kako je potrebno testirati svako od njih pojedinačno, ovo je moguće izvršiti jedino automatskim testovima. Na raspolaganju je razvijeno referentno testno okruženje u programskom jeziku Python koje generira testni C kod za svako sučelje pojedinačno. U referentnom se okruženju C kod za svaki test ponavlja. Budući da su ugradbeni računalni sustavi ograničeni količinom dostupne memorije, za veliki broj sučelja generirani C kod može biti veći od raspoloživih resursa. U okviru ovog diplomskog rada potrebno je izvršiti analizu načina rada pojedinačnog
Prijedlog ocjene pismenog dijela ispita (diplomskog rada):	Izvrstan (5)
Kratko obrazloženje ocjene prema Kriterijima za ocjenjivanje završnih i diplomskih radova:	Primjena znanja stečenih na fakultetu: 3 bod/boda Postignuti rezultati u odnosu na složenost zadatka: 3 bod/boda Jasnoća pismenog izražavanja: 2 bod/boda Razina samostalnosti: 3 razina
Datum prijedloga ocjene mentora:	22.09.2020.

Potpis mentora za predaju konačne verzije rada u Studentsku službu pri završetku studija:	Potpis:
	Datum:

FERIT
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA
I INFORMACIJSKIH TEHNOLOGIJA **OSIJEK**

IZJAVA O ORIGINALNOSTI RADA

Osijek, 05.10.2020.

Ime i prezime studenta:	Ante Bartulović
Studij:	Diplomski sveučilišni studij Računarstvo
Mat. br. studenta, godina upisa:	D-969R, 23.09.2019.
Turnitin podudaranje [%]:	2

Ovom izjavom izjavljujem da je rad pod nazivom**: Optimizacija potrošnje RAM memorije kod automatski generiranih testova za Autosar RTE**

izrađen pod vodstvom mentora Doc.dr.sc. Zdravko Krpić

i sumentora Dr.sc. Ivan Vidović

moj vlastiti rad i prema mom najboljem znanju ne sadrži prethodno objavljene ili neobjavljene pisane materijale drugih osoba, osim onih koji su izričito priznati navođenjem literature i drugih izvora informacija.
Izjavljujem da je intelektualni sadržaj navedenog rada proizvod mog vlastitog rada, osim u onom dijelu za koji mi je bila potrebna pomoć mentora, sumentora i drugih osoba, a što je izričito navedeno u radu.

Potpis studenta:

SADRŽAJ

1. UVOD

Tehnološki je napredak automobilske industrije i informacijske tehnologije sa sobom donio novu razinu simbioze računarstva i strojarstva. U današnje vrijeme automobili nude različite mogućnosti uređenja interijera, eksterijera i dodatne opreme. Navedene mogućnosti prati sve složenije sklopovlje i programska podrška. Osim napretka u dizajnu i programskoj podršci vozila, napredak je nastupio i u segmentu pisanja i izvršavanja testova potrebnih za validaciju i verifikaciju, kako zasebnih komponenti, tako i cjelokupnog sustava. Verifikacija sustava predstavlja proces provjere programske podrške na zahtjeve zadane dokumentacijom sustava. Proces validacije potvrđuje potrebu određene funkcionalnosti programske podrške za rad sustava. Sam proces testiranja, koji spada u proces verifikacije, ne garantira rad sustava bez pogrešaka, već služi kao potvrda o tome ispunjava li programska podrška zahtjeve zadane od razvojnog tima ili korisnika.

U automobilskoj industriji svi ugradbeni računalni sustavi zasnovani na AUTOSAR standardu su unaprijed definirani s obzirom na broj sučelja između komponenti. Broj sučelja može biti i preko 1000, a kako je potrebno testirati svako od njih pojedinačno, ovo je moguće izvršiti jedino automatskim testovima. U ovom diplomskom radu optimizirat će se radna memorija za sustav upravljan Infineon Tricore AURIX mikrokontrolerom koji je namijenjen automobilskoj industriji i u potpunosti je kompatibilan s AUTOSAR standardom. Za svako sučelje koje koristi Infineon Tricore AURIX mikrokontroler testno okruženje u programskom jeziku Python pojedinačno generira testni C programskim kod. U testnom se okruženju C programski kod za svaki test ponavlja. Budući da su ugradbeni računalni sustavi ograničeni količinom dostupne memorije, za veliki broj sučelja generirani C programski kod može zahtijevati više resursa nego što je raspoloživo na sustavu. U okviru ovog diplomskog rada potrebno je izvršiti analizu rada pojedinačnog testa u C programskom jeziku za AUTOSAR RTE sučelje te na odgovarajući način modificirati postojeće testove tako da potrošnja radne memorije bude minimalna. Nakon optimizacije dobivenog programskog koda potrebno je modificirati postojeći Python alat kako bi uz pomoć TEG-a generirao tako optimizirani C programski kod koji ispunjava zahtjeve programske podrške prvobitnog programskog koda.

U drugom poglavlju navedeni su i opisani znanstveni radovi koji pokrivaju metode optimizacije korištene i u ovom diplomskom radu. U trećem poglavlju navedene su i ukratko opisane tehnologije koje su korištene tijekom izrade diplomskog rada kako bi se čitatelju dočarali trenutni standardi u razvoju programske podrške u automobilskoj industriji. Nakon toga, u četvrtom poglavlju opisane su metode korištene za optimizaciju potrošnje radne memorije te su prikazani primjeri na generičkom programskom kodu. U petom poglavlju prikazan je pristup optimizaciji potrošnje radne memorije automatski generiranih testova AUTOSAR RTE te su prezentirani podaci o uštedi radne memorije za svaku pojedinu metodu i program.

2. PREGLED LITERATURE I POSTOJEĆA RJEŠENJA

Prilikom optimizacije radne memorije postoje dvije vrste pristupa: optimizacija programske podrške sustava i optimizacija sklopovskog dijela sustava. Prilikom optimizacije programske podrške sustava koriste se različite metode optimizacije petlji, struktura i dodjeljivanja memorije, dok se kod sklopovske optimizacije koristi veća količina memorije i brža memorija. U većini povezane literature pozornost se pridaje optimizaciji programske podrške, što je slučaj i kod optimizacije radne memorije automatski generiranih testova za AUTOSAR RTE. Osim optimizacije radne memorije, u povezanim radovima proučava se i optimizacija programske memorije, povećanje performansi procesora i smanjenje potrošnje električne energije ugradbenog računalnog sustava. Autori u povezanoj literaturi tipologiziraju više vrsta optimizacije. Spominje se optimizacija podataka, optimizacija petlji, optimizacija niza i spremnika, optimizacija nanizanih podataka, predmemorijska optimizacija, optimizacija glavne memorije, optimizacija programa i optimizacija procesa.

Tijekom dizajniranja ugradbenog računalnog sustava, potrebno je posvetiti pažnju svojstvima poput performansi procesora, disipaciji topline i ukupnom trošku implementacije. Kako bi se navedena svojstva ostvarila, nužno je pravilno korištenje memorije sustava, kako radne, tako i programske. Prema [1] za optimizaciju memorije i performansi sustava predloženo je optimiziranje rada programskih petlji. Metode koje spominju autori su otpetljavanje (engl. *loop unrolling*), inverzija (engl. *loop inversion*) i invarijant petlje (engl. *loop invariants*). U [2] i [3] se navodi mogućnost smanjenja potrošnje energije korištenjem metode otpetljavanja petlje, dok u [4] autor uvođenjem otpetljavanja petlje uspjeva smanjiti potrošnju radne memorije za 56%. Uz metode optimizacije petlji, autori u [1] predlažu proučavanje rada korištenog prevoditelja (engl. *compiler*) i njegovih razina optimizacije. Prema [5] metodu otpetljavanja petlje uvodi sam prevoditelj neovisno o programskom kodu tijekom prevođenja. S obzirom na navedene metode optimizacije petlji prema [1] odlučeno je primijeniti navedene metode pri optimizaciji radne memorije automatski generiranih testova za AUTOSAR RTE sustav. Metode otpetljavanja i inverzije petlje su implementirane u rad, no nisu donijele uštedu radne memorije, dok za metodu invarijante petlje nije postojala mogućnost implementacije zbog nepostojanja petlje sa računskom operacijom koja koristi varijablu nevezanu uz samu petlju. Predložena metoda promjene razine

optimizacije prevoditelja prema [1] i [5], donijela je najveće uštede radne memorije, iako se koristila na već optimiziranom programskom kodu.

Efektivno korištenje radne memorije omogućava procesoru brži pristup podacima te samim tim i veće performanse sustava. Kako bi se radna memorija efektivnije koristila, potrebno je pravilno rukovanje sa memorijskim adresama i podacima. U svrhu poboljšanja korištenja radne memorije, autori u [6] predlažu poravnavanje podataka unutar memorije ovisno o širini memorije procesora. Autori u [6] također prikazuju pogled na memoriju s programerske i procesorske strane. Iz programerske perspektive memorija se promatra kao beskonačni slijed bajtova, dok se iz procesorske perspektive memorija promatra kao niz blokova veličine 2, 4, 8, 16 ili čak 32 bajta. Osim povećavanja performansi rada procesora, koristi se poravnavanje podataka unutar struktura kako bi se izbjeglo nepotrebno trošenje memorije, što je posebno korisno pri pisanju programskog koda za ugradbene računalne sustave, što je slučaj i u ovom radu. Prema [7], poravnavanje memorije nije potrebno za sve računalne sustave, već za one koji imaju ograničenu radnu memoriju i performanse. Nadalje, prema [7], pri mjerenju brzine izvođenja na stolnom računalnu, program se izveo u gotovo identičnom vremenu neovisno o poravnavanju, dok su rezultati na računalu Raspberry Pi prikazali potrebu poravnanja podataka za sustave ograničene resursima. Kako bi poravnavanje podataka u memoriji bilo moguće, potrebno je znati širinu memorijske adrese procesora. Poravnavanje podataka potrebno je koristiti u svim načinima deklaracije podataka. Zanemarivanjem poravnavanja podataka velika količina radne memorije ostaje neiskorištena te usporava pristup procesora podacima. Iste metode za poravnavanje podataka kao i u [6], [7] i [8] primijenjene su tijekom optimizacije radne memorije AUTOSAR RTE automatski generiranih testova u ovom radu. Poravnavanje je korišteno unutar funkcija i struktura te je donijelo uštedu radne memorije.

Prema [8] ugradbeni računalni sustavi u pravilu koriste tri glavna resursa za rad, to su glavna procesorska jedinica, radna memorija potrebna za izvođenje programa i programska memorija potrebna za skladištenje programa. Parametar kojim se vrednuje rad glavne procesorske jedinice je vrijeme potrebno za izvođenje programa ili njegove funkcionalnosti, za radnu memoriju je parametar količina korištene memorije sustava potrebne za izvođenje programa, dok je parametar za programsku memoriju veličina programa pohranjenog u memoriji. Autori u [8] predlažu korištenje mehanizma otpuštanja memorije za alociranu memoriju koja nije potrebna tijekom

cijelog izvođenja rada programa. Uz to, umjesto korištenja malloc() funkcije za alokaciju memorije, predlažu korištenje statičkih polja. Unaprijeđenom procjenom potrošnje memorije preporučenom u [8] i [9], omogućeno je preddodjeljivanje memorije sustava za sve njegove module, što smanjuje fragmentaciju hrpe zbog rjeđe alokacije tijekom rada programa. Prema [9] i [10], fragmentacija memorije se izbjegava korištenjem brzopristupne memorije. Dodjeljivane spremnika brzopristupnoj memoriji dolazi do velikih ušteda radne memorije te se smanjuje broj malloc() i free() poziva prema [8] i [6]. Tijekom rada sa varijablama potrebno je paziti na tip korištenih podataka, odnosno odraditi testiranje veličina varijabli, jer se ispravnim odabirom smanjuje potrošnja radne memorije. Prema [8], ovom se metodom potrošnja radne memorije može smanjiti do 75%. Metoda smanjenja veličina varijabli navedena u [8] korištena je tijekom optimizacije radne memorije ovog rada. Sama metoda donijela je pozitivne rezultate pri uštedi radne memorije programske podrške. Metode dodjeljivanja podataka brzopristupnoj memoriji i njeno korištenje nije bilo moguće implementirati zbog same arhitekture sustava korištenog u ovom radu.

Prema [11] i [12] memorija sustava se navodi kao glavni prinosnik performansama i potrošnji električne energije kod ugradbenog računarstva. Autori u navedenim radovima se baziraju na cjelokupnoj optimizaciji sustava, a ne samo na optimizaciji radne memorije. Osim potrošnje električne energije sustava i njegovih performansi, potrebno je posvetiti pažnju potrošnji memorije radi smanjenja troška proizvodnje računalnog sklopovlja. Korištenje memorije ugradbenog sustava ima najveću ulogu u potrošnji električne energije te je bitno pristupiti njenoj optimizaciji, posebno kod sustava sa vanjskim napajanjem. Zadovoljavanje tri glavne komponente ugradbenog računalnog sustava: visoke performanse, velika računalna snaga i niski troškovi u većini situacija nije moguće. Različita sklopovlja zahtijevaju različite pristupe optimizaciji, tako, primjerice, sklopovlje namijenjeno obradi video signala koristi procesor i memoriju na drugačiji način od mrežnog procesora. Kako bi se smanjila potrošnja radne memorije sustava, potrebno je posvetiti pažnju optimalnom korištenju predmemorije sustava (engl. *cache memory*). Kao jedan od najvećih faktora optimizacije radne memorije, autori u [11] navode veličinu korištenih spremnika. Osim rada s memorijom, veličina spremnika utječe i na brzinu izvođenja programa i njegovu potrošnju električne energije. Optimalna veličina spremnika ovisi o veličini bloka datotečnog sustava, veličine predmemorije procesora i kolebanju kašnjenja (engl. *latency*) predmemorije. Kao i kod [8], u [11] se predlaže metoda smanjivanja veličine varijabli koja je donijela pozitivne rezultate

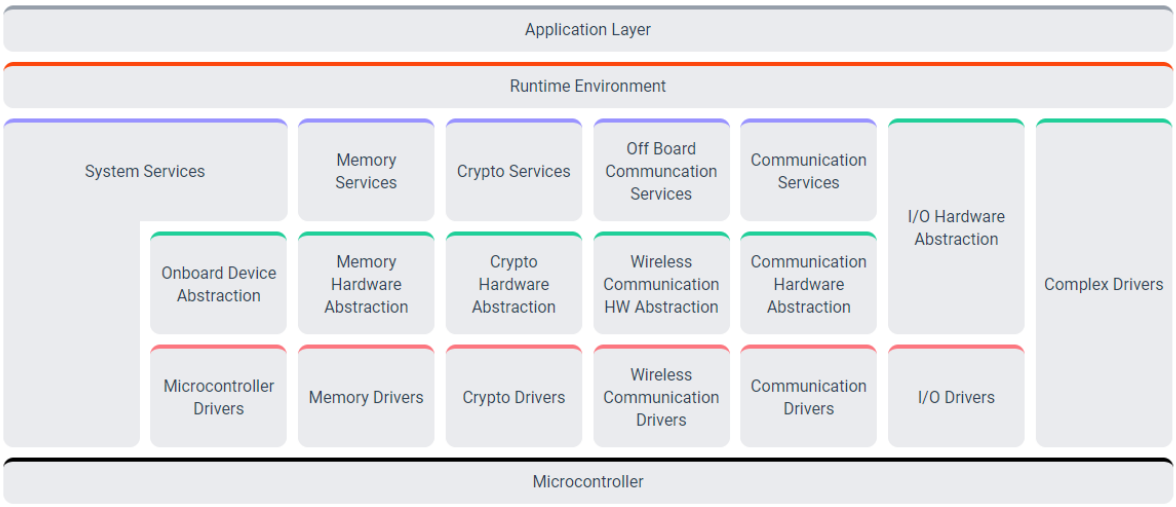
potrošnje radne memorije. Predložena metoda optimizacije niza podataka iz [11] donijela je najveće uštede na potrošnju radne memorije izuzevši metodu mijenjanja razina optimizacije prevoditelja.

3. KORIŠTENE TEHNOLOGIJE

U nastavku ovog poglavlja navedeni su i opisani standardi, tehnike i tehnologije korištene tijekom optimizacije radne memorije u ovom radu.

3.1 AUTOSAR

Standard Automotive Open System Architecture, AUTOSAR, predstavlja skup normi i pravila pisanja i korištenja programskog koda unutar automobilske industrije. Omogućava modularnost proizvodnje programske podrške između globalnih proizvođača. Arhitektura AUTOSAR standarda prikazana je na slici 3.1., a sastoji se od programskog sloja (engl. *Application Layer*), sloja izvršnog okruženja (engl. AUTOSAR *Runtime Environment*), servisnog sloja (engl. *Services Layer*), sloja apstrakcije elektroničke kontrolne jedinice (engl. *ECU Abstraction Layer*), sloja apstrakcije mikroupravljača (engl. *Microcontroller Abstraction Layer*) i sloja mikroupravljača. AUTOSAR standard omogućava kontinuirani i sigurni napredak automobilske industrije te ostvarivanje autonomne vožnje.



Slika 3.1. Arhitektura AUTOSAR RTE standarda, slika preuzeta sa [1313]

3.2 AUTOSAR Run-Time Environment (RTE)

Sloj izvršnog okruženja (engl. *Run-Time Environment* - RTE), tj. međuprogramski (engl. *middleware*) sloj, je sastavni dio središnje upravljačke jedinice (engl. *electronic control unit* –

ECU) i predstavlja sponu programske podrške sa zasebnim aplikacijama te im omogućava komunikaciju, razmjenu informacija i izvršavanje programskih funkcija. Prema AUTOSAR standardu, RTE ima mogućnost distribucije i integracije programske podrške neovisno o sklopovlju koje se koristi. RTE-u pristupa veći broj programskih komponenti (engl. *Software Component* - SWC), operacijski sustav, servisi, ulazno izlazne jedinice, upravljački program kompleksnih uređaja (engl. *Complex Device Driver* - CDD) i komunikacijski stog (engl. *stack*) koji se sastoji od programskih modula za CAN (engl. *Controller area network*), LIN (engl. *Local Interconnect Network*), FlexRay i Ethernet sabirnice.

3.3 MotionWise

MotionWise predstavlja modularnu platformu, razvijenu od stane TTTech grupacije, čija je namjena sigurno upravljanje radom elektroničkih komponenti autonomnih vozila. Kako bi sigurno upravljanje bilo omogućeno, MotionWise koristi centralni kontroler za asistenciju vozača (engl. *central driver assistance controll*er – zFAS), NVIDIA grafički procesor (engl. *graphics processing unit* – GPU) i ARM centralni procesor (engl. *central processing unit* – CPU) na NVIDIA Tegra K1 sustav na čipu (engl. *system on a chip* – SoC). Zadatak zFAS kontrolera je prikupljanje informacija sa različitih senzora automobila te na osnovu prikupljenih informacija odrediti sljedeći korak u određenoj radnji automobila. MotionWise je sigurna i skalabilna platforma za komunikaciju sustava autonomne vožnje namijenjena vozilima druge do pete razine autonomnosti te u potpunosti podržava AUTOSAR standard. Programski kod je optimiziran u ovom radu je testni programski kod MotionWise platforme.

3.4 Test Environment Generator (TEG)

Generator testnog okruženja (engl. *Test Environment Generator* – TEG), je skup alata potrebnih za generiranje testnog programskog koda i njegovog okruženja. TEG, razvijen od strane TTTech grupacije, služi za generiranje testnih komponenti, testnih stres slučajeva i testnih slučajeva. TEG generira četiri testna programa, to su ITF (*Inteface*), BUS (*Canbus*), PER (*Persistency*) i PFF (*Platform Functions*). Generirani testni slučajevi koriste se za potvrđivanje ispravnosti rada komunikacije između fizičkih i programskih komponenti automobila. Nakon generiranje navedenih testova, oni se izgrađuju (engl. *build*) i izvršavaju. Nakon izvršavanja

testova, generira se izvještaj o uspješnosti samog testa. Izvještaj o uspješnosti testa prikazuje sve izvedene testove, vrijeme izvođenja testova i status testa. Koristeći TEG alat na MotionWise platformi, generira se testni programski kod.

3.5 Radna memorija (RAM)

Tijekom rada programske podrške, za pisanje i čitanje se koristi radna memorija (engl. *Random-access memory* - RAM). Dvije glavne vrste radne memorije su statična radna memorija (engl. *Static random access memory* - SRAM) i dinamična radna memorija (engl. *Dynamic random access memory*, DRAM). SRAM se koristi kao brzopristupna predmemorija procesora te je brža od DRAM-a, dok se DRAM koristi kao glavna radna memorija sustava i ima veći kapacitet od SRAM-a. Za potrebe korištena radne memorije u svakom trenutku mora biti osiguran izvor napajanja sklopovlja, jer je radna memorija sama po sebi nestalna (engl. *volatile*), za razliku od programske memorije (engl. *Read Only Memory* - ROM) sustava za koju nije potrebno napajanje. U slučaju gubitka napajanja, informacije pohranjene unutar radne memorije fizički nestaju. Na radnu memoriju se zapisuje program u izvođenju i podaci tijekom izvođenja. Podaci unutar radne memorije zapisuju se u memorijske registre, a veličina samog registra ovisi o širini memorije (engl. *memory width*), što je detaljnije opisano u četvrtom poglavlju.

4. OPTIMIZACIJA POTROŠNJE RADNE MEMORIJE

Radna memorija sustava koristi se tijekom izvođenja programa, a služi za čitanje i pisanje informacija koje su potrebne u trenutnom radu sustava. Potrošnja radne memorije ovisi o količini korištenih informacija i funkcionalnosti programskog koda. U slučaju MotionWise platforme, potrošnja radne memorije može se gledati zasebno za svaki izgrađeni programski objekt ili za sveukupni izgrađeni program.

U nastavku poglavlja navedene su i opisane metode korištene pri optimizaciji radne memorije izgrađenog programskog koda. Korištene metode su podijeljene u tri cjeline. Prvo će biti prikazane metode optimizacije petlji, nakon toga metode optimizacije struktura i varijabli te na kraju ostale metode koje podrazumijevaju heurističke metode optimizacije funkcionalnosti sustava i razine optimizacije prevoditelja. Osim korištenja navedenih vrsta optimizacije, moguće je smanjiti potrošnju radne memorije koristeći određene zastavice (engl. *flags*) prevoditelja.

4.1 Metode optimizacije petlje

Metode optimizacije petlji su metode koje omogućuju smanjenje memorije uvođenjem instrukcijskog paralelizma (*engl*. instruction-level parallelism). Instrukcijski paralelizam omogućava procesoru istovremeno izvršavanje više instrukcija. Metode korištene pri optimizaciji programskih petlji su otpetljavanje i inverzija petlji koje su prikazane i objašnjenje u nastavku poglavlja.

4.1.1 Otpetljavanje petlje

Otpetljavanje petlje predstavlja metodu optimizacije petlji koje imaju mali broj iteracija. Ova metoda se koristi kako bi se izbjeglo nepotrebno ponavljanje inkrementiranja i provjeravanja vrijednosti podataka [14]. Nepovoljno je koristiti otpetljavanje za petlje koje imaju veliki broj iteracija ili u slučaju da broj iteracija određene radnje nije poznat unaprijed.

Na slici 4.1. prikazana je neoptimizirana verzija petlje. Generička petlja pri svakoj od deset iteraciji ispisuje proizvoljan tekst. Ovakav način zahtjeva provjeravanje vrijednosti varijable *i* i njeno inkrementiranje pri svakoj iteraciji.

```
1.  int main()
2.  {
3.  int i=0;
4.      for (i=0; i<10; i++)
5.      {
6.          printf("Optimizacija\n");
7.      }
8.      return 0;
9.  }
```

Slika 4.1. Prikaz programskog koda prije otpetljavanju petlje

Nakon što je uvedeno otpetljavanje petlje, broj ponovljenog inkrementiranja i provjeravanja vrijednosti varijable *i* se smanjuje, u ovome primjeru za pet puta. *For* petlja, kao što je prikazano na slici 4.2., izvršava se dva puta te pri svakom izvršenju ispisuje proizvoljan tekst pet puta. Neki programski prevoditelji automatski odrađuju metodu otpetljavanja pri prevođenju programskog koda i/ili uz korištenje posebnih postavki (zastavica).

```
1.  int main()
2.  {
3.  int i=0;
4.      for (i=0; i<2; i++)
5.      {
6.           printf("Optimizacija\n");
7.           printf("Optimizacija\n");
8.           printf("Optimizacija\n");
9.           printf("Optimizacija\n");
10.          printf("Optimizacija\n");
11.     }
12.     return 0;
13. }
```

Slika 4.2. Prikaz programskog koda nakon otpetljavanja petlje

4.1.2 Inverzija petlje

Inverzija petlje predstavlja način optimizacije pri kojem se *while* petlja zamjenjuje uvjetnim grananjem te nakon toga *do-while* petljom [15]. Navedene zamjena se radi kako bi se izbjeglo zastajanje cjevovoda (engl. *pipeline stall*) procesora zbog obavljanja dodatnih skokova unutar *while* petlje. Kod rada s *while* petljom, prvo se provjerava zadani uvjet, ako uvjet nije ispunjen, odrađuje se jedna iteracija petlje te se izvođenje programa vraća na početak provjere vrijednosti, kao što je prikazano na slici 4.3.

```
1.  int main()
2.  {
3.  int i = 0;
4.      while(i<5){
5.          printf("Optimizacija\n");
6.          i++;
7.      }
8.      return 0;
9.  }
```

Slika 4.3. Prikaz programskog koda prije inverzije petlje

Nakon preuređenja programskog koda inverzijom petlje, prikazano na slici 4.4., prvo se provjerava uvjet, a ako uvjet nije ispunjen odrađuje se još jedna iteracija, nakon toga se opet provjerava uvjet te ako je ispunjen izvršava se još jedna iteracija petlje.

```
1.  int main()
2.  {
3.      int i = 5;
4.      if(i<5){
5.      do{
6.          printf("Optimizacija\n");
7.          i++;
8.      }
9.      while(i<5);
10.     }
11.     return 0;
12. }
```

Slika 4.4. Prikaz programskog koda nakon inverzije petlje

4.2 Metode optimizacije varijabli i struktura

Metode optimizacije varijabli i struktura omogućavaju smanjenje potrošnje radne memorije pri skladištenju vrijednosti varijabli. Pri optimizacije varijabli i struktura korištene su metode: promjena veličine varijabli, promjena zapisa struktura unutar memorije, zamjena struktura i unija, promjena veličine polja te zamjena globalnih i lokalnih varijabli koje su prikazane i objašnjenje u nastavku poglavlja.

4.2.1 Promjena veličine varijabli

Kod deklariranja varijabli bitno je poznavati raspone vrijednosti varijabli koje će se skladištiti u radnu memoriji sustava. U slučaju da raspon vrijednosti varijabli nije poznat, potrebno

je testirati različite tipove podataka kako bi se pronašao optimalan tip podatka. Svaki tip podatka ima zadanu veličinu u memorijskom zapisu neovisno o skladištenoj vrijednosti. Korišteni tipovi podataka unutar neoptimiziranog programskog koda skladište vrijednosti u rasponu od 1 do 32 bita. U slučaju smanjenja veličine varijable sa 32 na 16 bit-a, rezultira se uštedom od 16 bita, odnosno 2 bajta po varijabli.

```
1.  uint32 var1;  //varijabla ima vrijednosti u rasponu 450-25000
2.  uint16 var2;  //varijabla ima vrijednosti u rasponu 1000-80000
3.  uint8  var3;  //varijabla ima vrijednosti u rasponu 0-255
```

Slika 4.5. Deklaracija neoptimiziranih varijabli

U navedenom primjeru sa slike 4.5., može se vidjeti da su poznati rasponi vrijednosti varijabli. Prva deklarirana varijabla ima veličinu 32 bit-a, a njena maksimalna vrijednost računa se putem

$$max = 2^n - 1\,, \tag{4-1}$$

gdje n predstavlja njenu veličinu. Za primjer varijable tipa *uint32* maksimalna vrijednost koju može poprimiti prema (4-1) je $2^{32} - 1 = 4.294.967.295$. Ukoliko je korištena vrijednost puno manja od maksimalne vrijednosti *uint32*, a spada u raspon vrijednosti koje može poprimiti podatak tipa *uint16* (od 0 do 65.535), memorijski je učinkovitije koristiti *uint16* tip podatka. Na taj se način može uštedjeti 2 bajta po varijabli.

4.2.2 Promjena zapisa struktura unutar memorije

Prilikom rada sa strukturama, bitno je paziti na mjesto u programskom kodu gdje će se inicijalizirati varijable kako bi se učinkovito iskoristila radna memorija sustava. Razvojna ploča na kojoj je rađena optimizacija posjeduje 32-bitne registre, odnosno širinu memorije, u koje se podaci spremaju kao zapisi od četiri bajta. Unutar jedne memorijske adrese moguće je skladištenje četiri osam bitne varijable ili dvije 16 bitne varijable, jedne 16 bitne i dvije osam bitne varijable ili jedne 32 bitne varijable. Na slici 4.6. prikazan je neučinkovit način deklariranja varijabli unutar strukture.

```
1.  struct Primjer{
2.      uint32 var1;
3.      uint16 var2;
4.      uint32 var3;
5.      uint8  var4;
6.      uint32 var5;
7.      uint8  var6;
8.      uint8  var7;
9.      uint8  var8;
10.     uint16 var9;
11.     uint8  var10;
12.     uint8  var11;
13.     uint16 var12;
14. };
```

Slika 4.6. Deklaracija neoptimizirane strukture

Kako bi se optimizirala potrošnja radne memorije, potrebno je rasporediti varijable unutar strukture po njihovoj veličini, odnosno popuniti zapise veličine četiri bajta na način opisan u gornjem dijelu ovog poglavlja [14]. Za navedeni primjer prije optimizacije potrošeno je osam memorijskih zapisa, odnosno 32 bajta, što je prikazano na slici 4.7., dok je nakon optimizacije, prikazano na slici 4.9., potrošeno šest memorijskih adresa, što predstavlja uštedu od osam bajta. Programski kod nakon promjene zapisa strukture u memoriji prikazan je na slici 4.8.

<table>
<tr><th>Adresa</th><th colspan="3">Memorija</th></tr>
<tr><td>0</td><td colspan="3">uint32</td></tr>
<tr><td>1</td><td colspan="3">uint16</td></tr>
<tr><td>2</td><td colspan="3">uint32</td></tr>
<tr><td>3</td><td colspan="3">uint8</td></tr>
<tr><td>4</td><td colspan="3">uint32</td></tr>
<tr><td>5</td><td>uint8</td><td>uint8</td><td>uint8</td></tr>
<tr><td>6</td><td>uint16</td><td>uint8</td><td>uint8</td></tr>
<tr><td>7</td><td colspan="3">uint16</td></tr>
</table>

Slika 4.7. Zauzeće registra neoptimizirane strukture

```
1.  struct Primjer{
2.      uint32 var1;
3.      uint32 var2;
4.      uint32 var3;
5.      uint16 var4;
6.      uint16 var5;
7.      uint8  var6;
8.      uint8  var7;
9.      uint8  var8;
10.     uint8  var9;
11.     uint16 var10;
12.     uint8  var11;
13.     uint8  var12;
14. };
```

Slika 4.8. Deklaracija optimizirane strukture

Adresa	Memorija			
0	uint32			
1	uint32			
2	uint32			
3	uint16		uint16	
4	uint8	uint8	uint8	uint8
5	uint16		uint8	uint8

Slika 4.9. Zauzeće registra optimizirane strukture

4.2.3 Promjena veličine polja

Prilikom deklariranja polja, važno je unaprijed paziti na njihovu veličinu. Pri korištenju iste strukture za više različitih funkcionalnosti, pažnju treba posvetiti njenoj modularnosti. Osim modularnosti, moguće je i definiranje različitih veličina polja ako za to ima potrebe. Na slici Slika 4.10. prikazan je standardni način deklariranje strukture s poljima.

```
1.  uint8 velicinaPolja = 8;
2.  struct Promjenjiva_polja{
3.      uint8 polje1[velicinaPolja];
4.  }
```

Slika 4.10. Deklariranje polja

Za strukturu prikazanu na slici 4.10. , može se vidjeti da je veličina polja „polje1“ uvijek ista, tj. poprima vrijednost varijable „velicinaPolja“ koja iznosi osam. U proizvoljnim situacijama, jedna struktura može se koristiti za različite potrebe, gdje veličina polja ne mora uvijek biti ista. Na slici Slika 4.11. prikazan je primjer optimizacije veličine polja za različite programe. Varijable unutar deklariranog polja imaju tip *uint8*, što predstavlja veličinu od jednog bajta. Prije izmjene, za četiri polja, zauzeće memorije iznosi 32 bajta. Nakon izmjene, zauzeće memorije iznosi 20 bajta.

```
#if defined (Build_1)
#define velicinaPolja        6
#define prazneInformacije   { 0, 0, 0, 0, 0, 0 }
#elif defined (Build_2)
#define velicinaPolja        2
#define prazneInformacije   { 0, 0 }
#elif defined (Build_3)
#define velicinaPolja        4
#define prazneInformacije   { 0, 0, 0, 0 }
#elif defined (Build_4)
#define velicinaPolja        8
#define prazneInformacije   { 0, 0, 0, 0, 0, 0, 0, 0 }
#endif
```

Slika 4.11. Definiranje promjenjive veličine polja

4.2.4 Zamjena struktura unijama

Skup više međusobno povezanih varijabli, moguće je definirati pomoću strukture ili unije. Struktura omogućava korištenje više varijabli istovremeno te skladištenje njihovih vrijednosti, ali svaka varijabla zauzima maksimalnu vrijednost svog tipa podatka, a struktura zbroj njihovih maksimalnih vrijednosti. Zamjenom strukture unijom gubi se mogućnost pristupa više varijabli u isto vrijeme, ali je potrošnja radne memorije sustava u tom slučaju jednaka veličini najvećeg tipa podatka unutar unije.

Na slici 4.12. prikazana je deklaracija strukture s pet različitih varijabli. Na deklariranoj strukturi moguć je rad sa više varijabli istovremeno zbog spremanja vrijednosti na različitim memorijskim adresama. Suprotno od strukture, unija, prikazana na slici 4.13., ne omogućuje rad s više različitih varijabli u isto vrijeme. Razlog tome je spremanje vrijednosti varijabli na istu memorijsku adresu radi memorijske učinkovitosti. U prikazanom slučaju, deklarirana struktura zauzima 13 bajta radne memorije, dok unija zauzima maksimalnu vrijednost najvećeg tipa podatka, odnosno 4 bajta radne memorije, što predstavlja uštedu od 9 bajta.

```
struct Primjer{
    uint32 var1;
    uint16 var2;
    uint8 var3;
    float var4;
    char var5[2];
}
```

Slika 4.12. Deklaracija strukture

```
union Primjer{
    uint32 var1;
    uint16 var2;
    uint8 var3;
    float var4;
    char var5[2];
}
```

Slika 4.13. Deklaracija unije

4.2.5 Zamjena lokalnih varijabli s globalnim varijablama

Kako bi se izbjegla konstantna deklaracija istih, često korištenih varijabli, umjesto lokalne deklaracije varijabli, varijable su deklarirane jednom, ali globalno za sve funkcije sustava. Globalne varijable se zapisuju na fiksnu memorijsku lokaciju koju određuje prevoditelj. .Na početku pokretanja programskog koda njihova vrijednost se zapisuje unutar memorije, dok se kod lokalnih varijabli vrijednost zapisuje na stog u početku izvršavanja funkcije te uništava nakon izvršavanja iste. Osim razlike u memorijskom zapisu, globalne varijable zadržavaju svoju vrijednost nakon izvršavanja funkcije. Zadržavanje vrijednosti globalnih varijabli može stvoriti dodatne problem ako se vrijednost varijable ne postavi na traženu vrijednost prije rada s njom. Tražena vrijednost globalne varijable može se postaviti na početku ili kraju izvršavanja funkcije, petlje ili uvjetnog grananja.

4.3 Ostale metode optimizacije radne memorije

Ostale metode koje se koriste za optimizaciju radne memorije, a ne odnose se na optimizaciju rada programskih petlji ili varijabla i struktura. Od takvih metoda mogu se izdvojiti heurističke metode optimizacije funkcionalnosti sustava i korištenje predefiniranih postavki prevoditelja. Navedene su metode opisane u nastavku poglavlja.

4.3.1 Heurističke metode optimizacije funkcionalnosti sustava

Osim navedenih metoda optimizacije radne memorije na kojima se rade istraživanja kako bi pomogle u industriji, heurističkim se metodama pristupilo optimizaciji funkcionalnosti sustava. Funkcionalnosti sustava su pisane u više iteracija te testirane kako bi se došlo do verzije sa najmanjom potrošnjom radne memorije. Za potrebe optimizacije funkcionalnosti sustava svi navedeni primjeri optimizacije se testiraju na raznim funkcionalnostima, gdje je to moguće.

4.3.2 Postavke prevoditelja

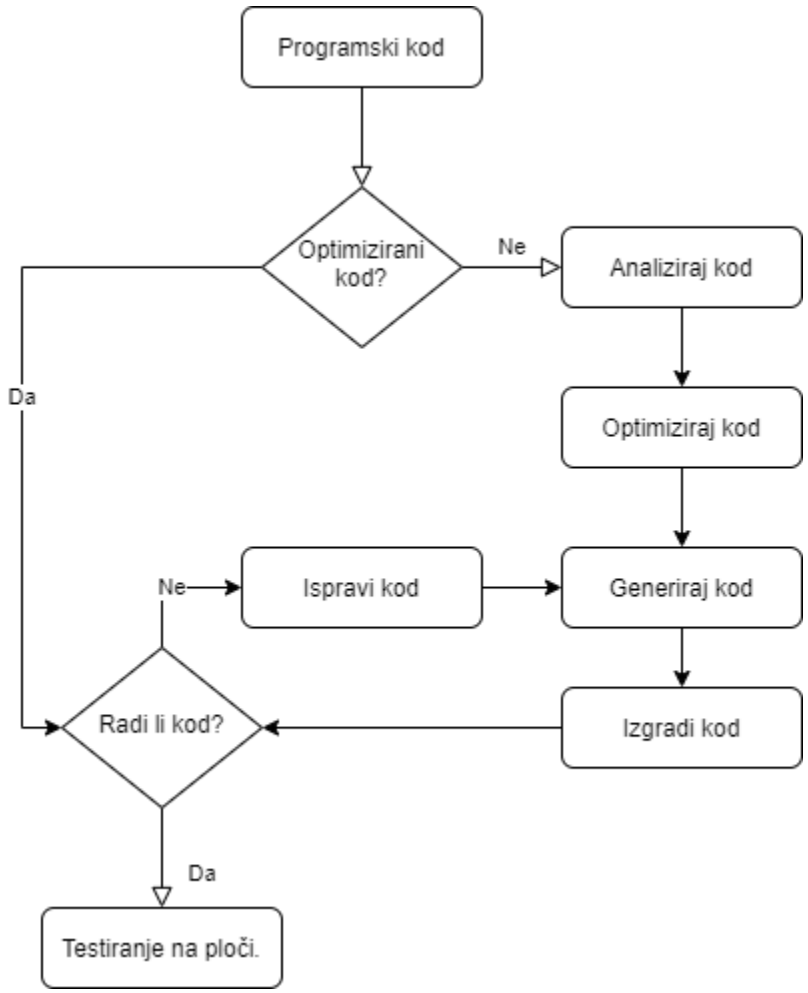
Prevoditelj programskog koda predstavlja sponu između viših programskih jezika, u ovom slučaju C-a, i nižih programskih jezika, u ovom slučaju asemblera. Prije pokretanja prevoditelja, potrebno je odabrati razinu korištene optimizacije prevoditelja i zastavice koje omogućuju prevoditelju kompromis (engl. *tradeoff*) između brzine rada procesora, potrošnje radne i programske memorije. Za prevođenje korišten je Tasking Tricore 4.2 prevoditelj. Navedeni prevoditelj nudi četiri razine optimizacije programskog koda. Pri optimizaciji programskog koda korištena je razina O0 koja ne optimizira veličinu programskog koda ili brzinu izvođenja. Razina O0 zadana je zahtjevima na programsko rješenje. Osim razine O0 postoje još razine O1, O2 i O3 koje su namijenjene smanjenju potrošnje memorije i bržem izvođenju programskog koda. O1 razina donosi najmanje pogodnosti, dok O3 razina donosi najveće pogodnosti prilikom optimizacije. Optimizacija programske podrške pomoću promjena razina prevoditelja odradit će se zasebno od ostalih optimizacija te neće biti testirana na razvojnoj ploči.

5. EKSPERIMENTALNA ANALIZA

Programska podrška koja će biti optimizirana u ovom radu napisana je u C programskom jeziku i namijenjena je testiranju interne komunikacije MotionWise sustava. Koristeći Python alate nad C programskim kodom generira se programski kod za provođenje testova. Navedeni testovi zapisani su u programskom jeziku C. Nakon što se testni programski kod generira, potrebno ga je izgraditi. Izgrađeni programski kod se sastoji od četiri različita programa, to su ITF, BUS, PER i PFF. Svaki izgrađeni program odrađuje zasebne testove funkcionalnosti komunikacije između SWC-ova. Nad izgrađenim programskim kodom izvršena je optimizacija radne memorije koristeći metode navedene u četvrtom poglavlju. Kako bi ušteda radne memorije bila uspješna, svaki izgrađeni program mora proći testiranje rada na razvojnoj ploči pogonjenoj Infineon Tricore AURIX mikrokontrolerom i generirati izvješće o uspješnosti izgrađenog programskog koda. Rezultati optimizacije biti će opisani u nastavku poglavlja te u konačnici podijeljeni po izgrađenom programskom kodu.

5.1 Pristup optimizaciji radne memorije

Na slici 5.1. prikazan je dijagram toka optimizacije radne memorije. Prije optimiziranja radne memorije potrebno je analizirati programski kod kako bi se istražila mogućnost optimizacije. Ručnom analizom programskog koda otkrivene su mogućnosti primjene metoda optimizacije nad varijablama, strukturama i petljama programskog koda te se metode opisane u četvrtom poglavlju primjenjuju na programski kod. Nakon svake optimizacije programski kod je generiran i izgrađen kako bi se testiralo postoji li smanjenje potrošnje radne memorije. Ako nakon izgrađivanja programskog koda nema pogrešaka ili prelijevanja podataka (engl. *overflow*), izgrađeni program se može testirati na razvojnoj ploči. U nastavku poglavlja prikazat će se način korištenja metoda optimizacije radne memorije koje su opisane u poglavlju 4.



Slika 5.1. Dijagram toka optimizacije radne memorije

5.2 Postupak optimizacije radne memorije

Testni programski kod podijeljen je u više dijelova, ovisno o funkcionalnosti. Podijeljeni programski kodovi nazivaju se fragmenti. Svaki fragment zadužen je za određenu funkcionalnost poput čitanja informacija na ulazu u sustav. Ručnom analizom programskog koda uočeno je korištenje velikog broja struktura, varijabli i petlji nad kojima će se primijeniti metode optimizacije navedene u četvrtom poglavlju. Na početku optimizacije radne memorije programskog koda varijable sustava koje se koriste više od jednom, a deklarirane su lokalno, prebačene su u globalnu deklaraciju. Metoda prebacivanja lokalnih varijabli u globalne nije donijela uštedu radne memorije na niti jednom izgrađenom programu. Nakon toga izdvojene su varijable sustava veće od jednog bajta, odnosno veće od *uint8* tipa podatka te je nad njima vršeno prepolovljavanje veličine tipa

podatka, sve dok je postojala mogućnost smanjivanja, odnosno dok veličina varijable nije postigla minimalnu vrijednost bez prelijevanja podataka. Metoda promjene veličina varijabli donijela je smanjenje potrošnje radne memorije. Identična metoda primijenjena je i za varijable unutar strukture. Nakon promjene veličina varijabli unutar strukture, bilo je potrebno poravnati varijable unutar strukture u memoriji te popuniti sva četiri bajta po memorijskoj adresi. Zadnja promjena u radu sa strukturama bila je promjena strukture kao složenog tipa podatka u uniju, što bi teoretski donijelo značajne uštede u potrošnji radne memorije. Metoda promjene strukture u uniju sama po sebi donosi velike uštede, ali u korištenom programskom kodu korištenje unije onemogućava izvršavanje funkcionalnosti sustava. Promjena veličine varijabli unutar struktura i njihovo poravnavanje donijeli su smanjenje potrošnje radne memorije. Transformacija strukture u uniju donosi smanjenje potrošnje radne memorije, ali je nepoželjna kod ovakvog sustava jer je onemogućen istovremeni pristup prema više varijabli unije. Metoda promjene veličine varijabli i njihovo poravnavanje najveće su uštede donijeli na ITF programu, a najmanje uštede na PFF programu. Uštede na ITF programu iznose 3.288 bajta ili 1,04% ukupne memorije, dok su uštede na PFF programu iznosile 248 bajta ili 0,084%. Nakon što su varijable u programskom kodu optimizirane i poravnate, potrebno je optimizirati veličinu korištenih polja. Polja korištena u programskom kodu su statički deklarirana te za svaki program imaju istu veličinu. Prilikom rada sa programskim kodom, uočeno je da svaki program koristi različitu veličinu polja, stoga je veličinu polja potrebno prilagoditi korištenom programu. Promjenom veličine korištenih polja ovisno o programu, uspješno je smanjena potrošnja radne memorije**.** Na ITF programu je promjena veličine korištenih polja donijela najveće uštede te najmanje uštede na PFF programu, kao i kod optimizacije veličine varijabli i njihova poravnanja. Optimizacija veličine korištenih polja na ITF programu donijela je uštedu od 9.432 bajta ili 2,97%, što čini ukupnu uštedu od 12.720 bajta ili 4,01%, dok ista metoda nije donijela uštede na PPF programu. Nakon optimizacije veličine varijabli i struktura, optimiziran je rad programskih petlji. Za optimizaciju programskih petlji korištene su dvije metode, otpetljavanje petlje i inverzija petlje. Za *for* petlje se koristila metoda otpetljavanja, dok je za *while* petlje korištena metoda inverzije petlje. Niti jedna metoda optimizacije petlji nije donijela smanjenje potrošnje radne memorije jer prevoditelj tijekom prevođenja sam implementira metode optimizacije petlji.

Nakon odrađivanja navedenih metoda, programski kod je testiran na ploči za slučaj ITF programa. Nakon testiranja, generirano je izvješće o prolaznosti testova prikazano na slikama 5.2. i 5.3.

Prolaznost testova prije optimizacije prikazana je na slici 5.2., dok je na slici 5.3. prikazana prolaznost testova nakon optimizacije radne memorije. Izvješće se sastoji od pet redova, a svaki red ima po tri stupca. Prvi red izvješća prikazuje ukupni broj testnih slučajeva. Drugi red prikazuje ukupan broj izvedenih testnih slučajeva, dok treći red prikazuje ukupan broj neizvedenih testnih slučajeva. U četvrtom redu prikazan je ukupan broj testova s pozitivnim ishodom te postotak testova s pozitivnim ishodom u odnosu na ukupan broj izvršenih testova. Posljednji red prikazuje broj testnih slučajeva s negativnim ishodom te postotak testova s negativnim ishodom u odnosu na ukupan broj izvršenih testova. Program je uspješno prošao testiranje s 98% uspješnosti testova, kao i prije optimizacije radne memorije. Nad programskim kodom prije optimizacije odrađeno je 834 testa, od kojih je 820 uspješno prošlo, a 14 ih je prošlo neuspješno. Na optimiziranom programskom kodu odrađeno je 874 testa, od čega ih je 857 bilo uspješno, dok ih je 17 bilo neuspješno.

Statistics

Overall number of test cases	834	
Executed test cases	834	100% of all test cases
Not executed test cases	0	0% of all test cases
Test cases passed	820	98% of executed test cases
Test cases failed	14	2% of executed test cases

Slika 5.2. Rezultati testiranja neoptimiziranog ITF programskog koda

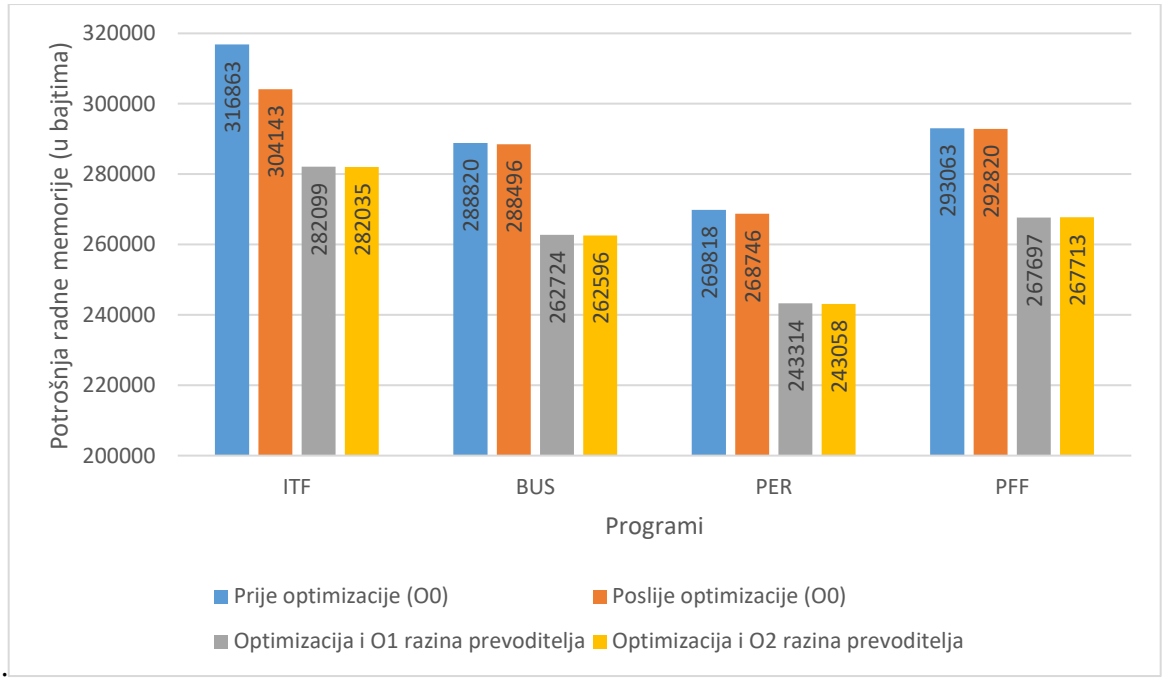
Statistics

Overall number of test cases	874	
Executed test cases	874	100% of all test cases
Not executed test cases	0	0% of all test cases
Test cases passed	857	98% of executed test cases
Test cases failed	17	2% of executed test cases

Slika 5.3. Rezultati testiranja optimiziranog ITF programskog koda

Osim navedenih metoda testirana je potrošnja radne memorije koristeći različite razine optimizacije programskog prevoditelja, zastavicama O1, O2, O3 na već optimiziranom programskom kodu. Na O1 razini dolazi do velikih ušteda radne memorije sustava, gdje je najveća

uštede na ITF programu i iznosi dodatnih 22.108 bajta ili 7,25%, odnosno sveukupno uštedu od 34.764 bajta ili 10,97%. Na O2 razini su manji rezultati uštede radne memorije u odnosu na O1 razinu. Najveća ušteda ostvarena uz O2 razinu je na PER programu i iznosi 256 bajta ili 0,1% smanjenja potrošnje korištene radne memorije u odnosu na O1 razinu. Na PFF programu potrošnja radne memorije je povećana koristeći O2 razinu u odnosu na O1 razinu u vrijednosti od 16 bajta. Rezultati testiranja na O3 razini nisu dostupni jer prevoditelj nije uspijevao izgraditi program. Rezultati optimizacije prevoditelja za sve izgrađene programe prikazani su na slici 5.4.

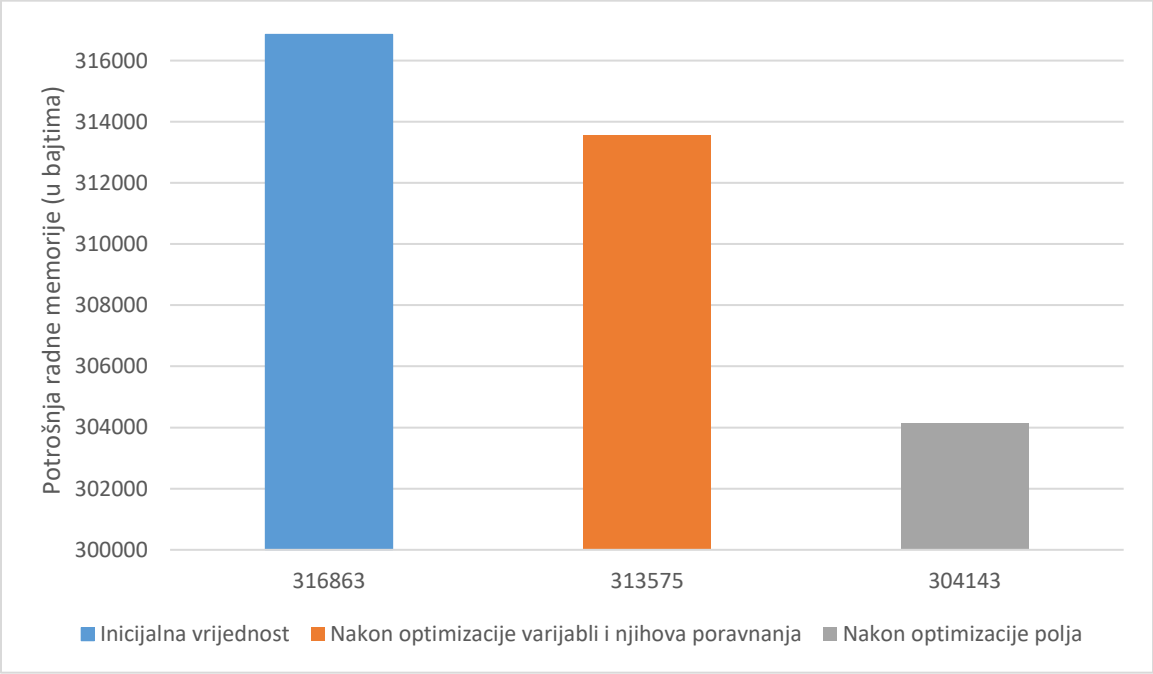


Slika 5.4. Rezultati optimizacije prevoditelja za ITF program

5.3 Potrošnja radne memorije nakon optimizacije podijeljena po programima

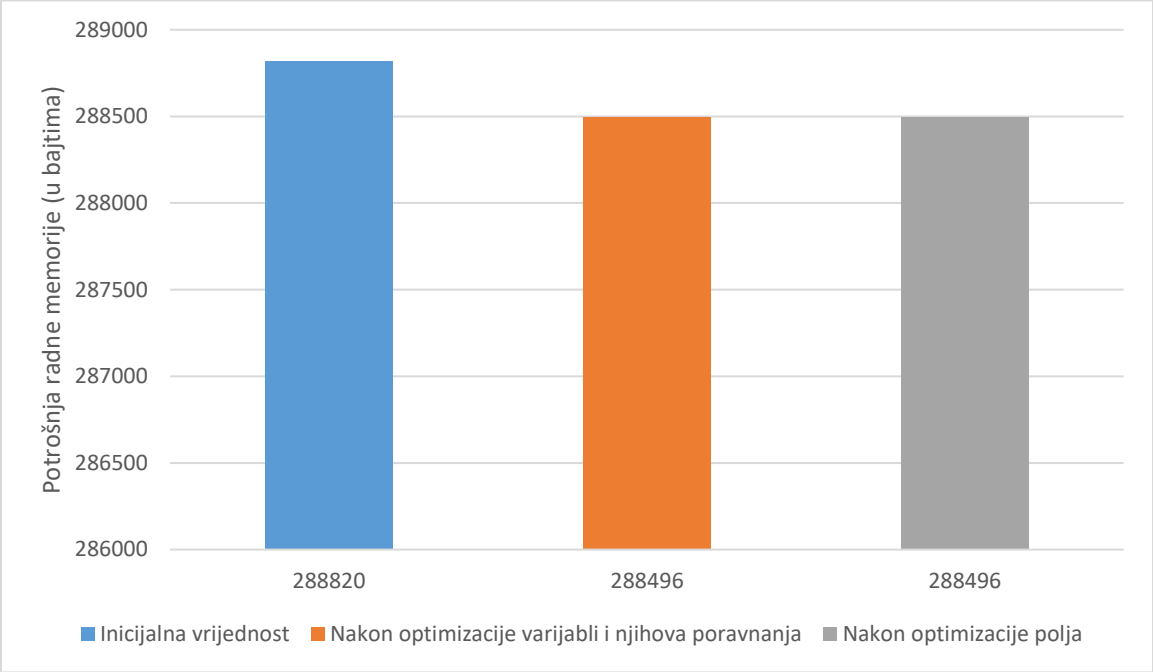
U nastavku rada prikazani su konačni rezultati optimizacije radne memorije sustav za sva četiri programa. Rezultati su prikazani u obliku grafa i tekstualne interpretacije istog za metode smanjenja veličine varijabli, njihova poravnanja u memoriji i smanjenje veličine korištenih polja, jer su te metode donijele smanjenje potrošnje radne memorije.

Potrošnja radne memorije prvog programa prije optimizacije iznosila je 316.863 bajta**.** Nakon smanjenja veličine varijabli i poravnavanje istih u memorijskim adresama, potrošnja radne memorije je spuštena na 313.575 bajta. Smanjenje veličine polja postiglo je najveće uštede radne memorije u vrijednosti od 9.432 bajta. Ukupna potrošnja radne memorije nakon optimizacije iznosi 304.143 bajta, što predstavlja uštedu od 12.720 bajta ili 4,01%. Rezultati optimizacije prikazani su na slici 5.5.



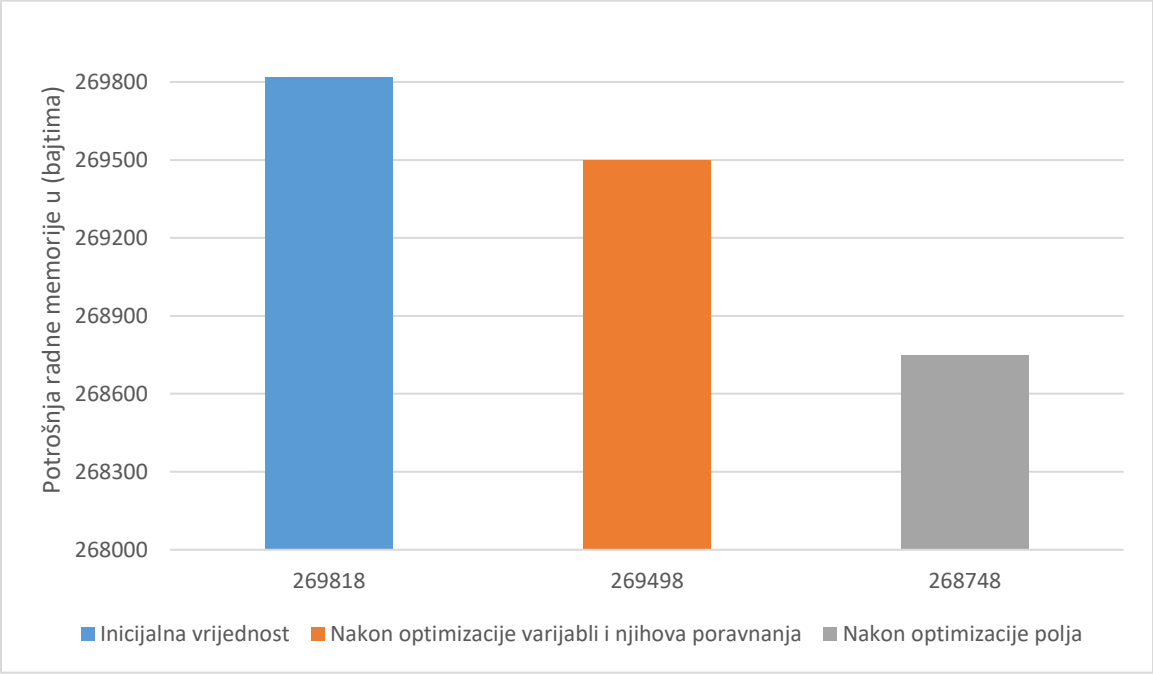
Slika 5.5. Potrošnja radne memorije ITF programa

Kod drugog programa prvobitna potrošnja radne memorije iznosila je 288.820 bajta**.** Optimizacija varijabli i njihovo poravnavanje u memoriji smanjilo je potrošnju radne memorije na 288.496 bajta ili 0,11%**.** Smanjenje veličine korištenih polja nije donijelo uštedu radne memorije drugog programa. Rezultati optimizacije prikazani su na slici 5.6.



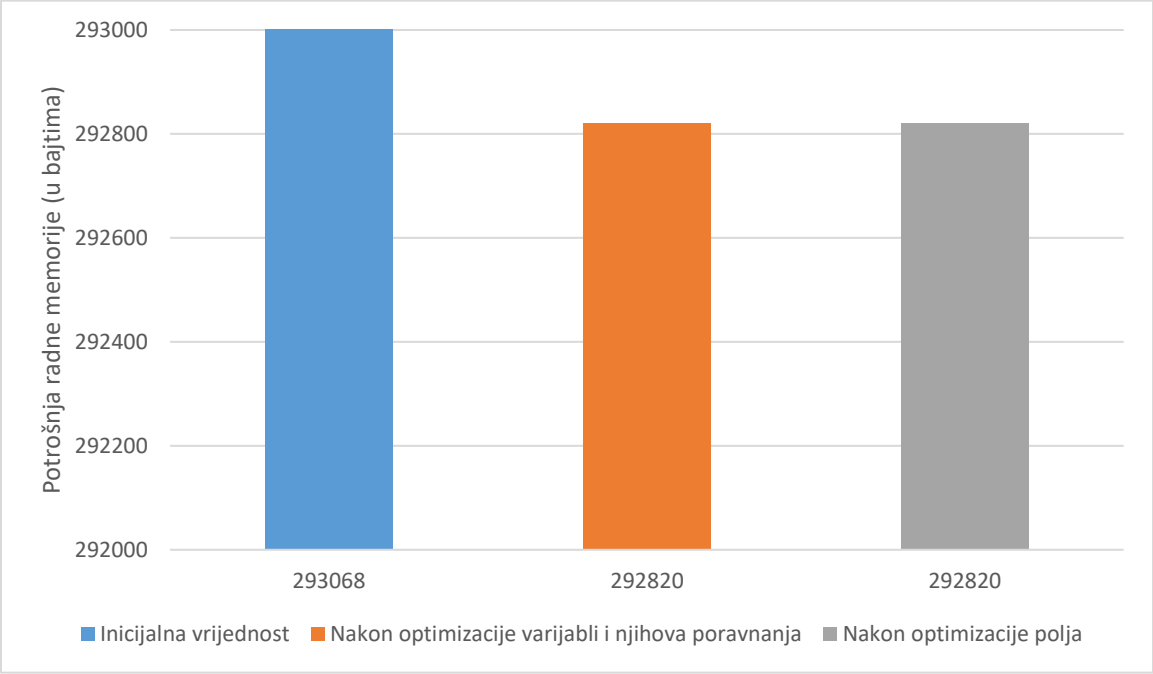
Slika 5.6. Potrošnja radne memorije BUS programa

Kod trećeg programa potrošnja radne memorije prije optimizacije iznosila je 269.818 bajta**.** Smanjenje veličine korištenih varijabli te njihovim poravnanjem u memoriji uštađeno je 320 bajta ili 0,12%**.** Dodatna radna memorija uštedila se promjenom veličine polja. Ušteda smanjenjem veličine polja iznosi dodatnih 750 bajta, što čini konačnu uštedu od 1.070 bajta ili 0,39% Rezultati optimizacije prikazani su na slici 5.7.



Slika 5.7. Potrošnja radne memorije PER programa

Količina potrebne radne memorije četvrtog programa prije optimizacije iznosila je 293.068 bajta**.** Nakon optimizacije veličine varijabli i njihova poravnavanja, potrošnja radne memorije iznosi 292.820 bajta**.** Smanjenjem veličine korištenih polje, potrošnja radne memorije je ostala ista. Na četvrtom programu je sveukupno uštađeno 248 bajta ili 0,084% Rezultati optimizacije četvrtog programa prikazani su na slici 5.8.



Slika 5.8. Potrošnja radne memorije PFF programa

6. ZAKLJUČAK

Razvoj automobilske industrije sa sobom je donio tehnološku revoluciju u obliku autonomne vožnje. Kako bi autonomna vožnja bila moguća, potrebno je osigurati velike količine ograničenih računalnih resursa za analizu i obradu okoline vozila. Jedan od tih ograničenih resursa je radna memorija. U radu su opisane tehnologije i metode korištene pri optimizaciji potrošnje radne memorije automatski generiranih testova za AUTOSAR RTE. Automatski generirani testovi pokreću se na ugradbenom računalnom sustavu pogonjenom Infineon Tricore AURIX mikroupravljačem.

Korištene metode pri optimizaciji radne memorije automatski generiranih testova za AUTOSAR RTE podijeljene su u tri cjeline. U prvu cjelinu spadaju optimizacije petlji, u drugu optimizacije veličine varijabli i struktura te na kraju ostale optimizacije poput korištenja raznih optimizacija prevoditelja. Optimizacija radne memorije započela je optimizacijom varijabli, tj. prebacivanjem lokalnih varijabli u globalne. Nakon toga su testirane veličine varijabli te mogućnost smanjivanja veličine istih. Nakon optimiziranja varijabli, uslijedilo je optimiziranje struktura. Strukture su poravnate u memorijskim adresama te je testirano prebacivanje struktura u unije. Posljednja metoda pri radu sa varijablama i strukturama bila je prilagođavanje veličine polja potrebama programa. Nakon optimizacije varijabli i struktura, uslijedilo je optimiziranje rada programskih petlji. Prvo je primijenjena metoda otpetljavanja petlje za *for* petlje te nakon toga metoda inverzije petlje za *while* petlje. Optimizacija radne memorije završena je testiranjem izgrađenih programa na razvojnoj ploči. Osim navedenih metoda, programi su izgrađeni pomoću različitih razina optimizacije prevoditelja, no taj programski kod nije testiran na ploči.

Smanjenje potrošnje radne memorije donijele su metode promjene veličine varijabli i njihova poravnavanja u memorijskim adresama i metoda promjene veličine polja. Na ITF programu, ušteda pri promjeni veličine varijabli i njihova poravnanja u memoriji iznosi 3.288 bajta ili 1,04% ukupno zauzete radne memorije, dok metoda promjene veličine polja donosi uštedu od 9.432 bajta ili 2,97%. Navedene metode zajedno donose uštedu od 12.720 bajta ili 4,01% što čini značajnu uštedu za ugradbene računalne sustave. Nakon izgradnje, ušteda radne memorije potvrđena je testiranjem na razvojnom sustavu. Promjene razina optimizacije prevoditelja nisu testirane na ploči, no donose velike uštede potrošnje radne memorije. Na ITF programu radna memorija je

smanjena za 22.108 bajta ili 7,25% ukupno zauzete radne memorije za O1 razinu optimizacije. O2 razina optimizacije donosi uštedu od 34.764 bajta ili 10,97% na ITF programu.

Osim odrađenih metoda optimizacije radne memorije, moguće je dodatno optimizirati radnu memoriju programskim i sklopovskim rješenjima. Sa programske strane moguća je dodatna analiza programske podrške te programiranje na nivou asemblera. Sklopovska strana omogućava korištenje veće količine radne memorije i bržu memoriju, što se programskim optimizacijama pokušava izbjeći.

LITERATURA

[1] C. Jubb, „Loop Optimizations in Modern C Compilers“, Columbia University , New York, str. 15., 2014.

[2] M. Booshehri, A. Malekpour, P. Luksch, „An Improving Method for Loop Unrolling“, u International Journal of Computer Science and Information Security (IJCSIS), sv. 11, izd. 5, str. 73-76, svi. 2013.

[3] H. Yang, G. R. Gao, A. Marquez, G. Cai, Z. Hu, „Power and Energy Impact by Loop Transformations“, str. 8., 2000.

[4] D. Shires, "Effects of Loop Unrolling and Loop Fusion on Register Pressure and Code Performance", Army Research Laboratory, 1997.

[5] R. A. Ashraf, R. Gioiosa, G. Kestor, R. F. DeMara, „Exploring the Effect of Compiler Optimizations on the Reliability of HPC Applications“, u 2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), str. 1274–1283, svi. 2017

[6] N. R. Patel, „Data Structure Alignment“, IJETT, sv. 45, izd. 8, str. 388–390, ožu. 2017

[7] S. H. Haase, Effiziente Programmierung in C, [Na internetu]. Dostupno na: https://hps.vi4io.org/_media/teaching/wintersemester_2013_2014/epc-14-haase-svenhendrik-alignmentinc-paper.pdf [Pristup ostvaren: 22. srpnja 2020.]

[8] A. Mistry, R. Kher, „Embedded Software Optimization for Computation - Intensive Applications“, Journal of Electrical and Electronic Engineering. Special Issue: Soft Computing Methods for Electrical and Electronics Engineering Applications. sv. 8, izd. 2, str. 42-46, 2020

[9] P. R. Panda i ostali, „Data and memory optimization techniques for embedded systems“, ACM Trans. Des. Autom. Electron. Syst., sv. 6, izd. 2, str. 149–206, tra. 2001.

[10] O. Zendra, „Memory and compiler optimizations for low-power and –energy“, 1st ECOOP Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems, str. 8., srp. 2006

[11] W. Wolf, M. Kandemir, „Memory system optimization of embedded software“, Proc. IEEE, sv. 91, izd. 1, str. 165–182, sij. 2003.

[12] L. Oliveira, J. C. B. Mattos, L. Brisolara, „Survey of Memory Optimization Techniques for Embedded Systems“, u 2013 III Brazilian Symposium on Computing Systems Engineering, str. 65–70, stu. 2013.

[13] AUTOSAR Classic platform, [Na internetu], AUTOSAR GbR, dostupno na https://www.autosar.org/standards/classic-platform/ [Pristup ostvaren: 22. srpnja 2020.]

[14] A. Aho, M. Lam, R. Sethi, J. D. Ullman, Compilers: Principles, Techniques, and Tools (2nd Edition), Addison Wesley, Boston, Massachusetts, SAD ,2006.

[15] M. E. Lee, Optimization of Computer Programs in C, [Na internetu], Ontek Corporation, dostupno na: http://leto.net/docs/C-optimization.php, [Pristup ostvaren: 18.09.2020.]

SAŽETAK

Napretkom automobilske industrije dolazi do povećane potrebe za razvojem i održavanjem programske podrške ugradbenih sustava u automobilu. Ugradbeni računalni sustavi su ograničeni resursima potrebnima za rad, pa tako i radnom memorijom. U ovom radu bilo je potrebno smanjiti potrošnju radne memorije automatski generiranih testova ugradbenog računalnog sustava zasnovanog na Infineon Tricore AURIX mikrokontroleru. Testni programski kod u C programskom jeziku generira se koristeći Python alate nad C programskim kodom. U svrhu optimizacije radne memorije primijenjene su metode inverzije petlje, otpetljavanje petlje, smanjenje veličine varijabli, zamjena globalnih i lokalnih varijabli, poravnavanje memorije, zamjena strukture s unijom i heurističke metode. Osim navedenih metoda, testirane su i razine optimizacije prevoditelja, no one nisu testirane na razvojnom sustavu. Optimizirani programski kod je uspješno prošao testiranja te je korištenjem navedenih metoda uštađeno do 4,01% radne memorije.

Ključne riječi: AUTOSAR, radna memorija, tehnike optimizacije, ugradbeni računalni sustav, TEG

ABSTRACT

RAM usage optimization for automatically generated tests for AUTOSAR RTE

With the advancement of the automotive industry, a need to develop and maintain software for in-car embedded systems has risen. Embedded systems are limited with the resources needed to operate, including random access memory. In this paper, it was necessary to reduce the consumption of random access memory of automatically generated tests for embedded computer systems based on Infineon Tricore AURIX microcontroller. The test program code in the programming language C was generated using the Python tools on the C programming code. The techniques used for reducing the consumption of RAM are loop inversion, loop unrolling, reducing the sizes of the variables, switching global and local variables, memory alignment, replacing structures with unions and heuristic methods. Also, compiler optimization options were used but not tested on the system. The optimized program code successfully apssed all the tests and a total consumption of RAM was reduced by up to 4.01%.

Key words: AUTOSAR, random access memory, optimization techniques, embedded system, TEG

ŽIVOTOPIS

Ante Bartulović rođen je 29. prosinca 1994. godine u Zagrebu. OŠ Ivan Mažuranić završava u Vinkovcima u kojima živi sve do upisa na fakultet. U Vukovaru 2009. godine upisuje Matematičku Gimnaziju, maturira 2013. godine te iste te godine upisuje Elektrotehnički fakultet u Osijeku, današnji Fakultet elektrotehnike, računarstva i informacijskih tehnologija, smjer Računarstvo. Preddiplomski studij Računarstva završava 2018. godine te iste te godine upisuje diplomski studij Računalno inženjerstvo.