Metode optimizacije u programskom jeziku C

Bašić, Damir

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

Sveučilišni studij

Metode optimizacije u programskom jeziku C

Završni rad

Damir Bašić

Osijek, 2021.

SADRŽAJ

1. UVOD

Razvojem prvih osobnih računala uloga optimizacije programskih kodova bila je ključna zbog velikih memorijskih i procesorskih ograničenja osobnih računala tog doba. Četvrta generacija računala pojavljuje se 70-ih godina 20. stoljeća, od kada je mikroprocesor neizbježna temeljna jedinica računala. Računala tog doba zbog uporabe mikroprocesora postajala su sve brža, jeftinija i manja. Osim toga, četvrtu generaciju računala karakterizira i pojavljivanje objektno orijentiranog programiranja, relacijskih baza podataka te laserskih pisača. Programski jezik C, koji je u više navrata formalno i neformalno standardiziran te pripada trećoj generaciji programskih jezika, jedan je od glavnih predstavnika programskih jezika četvrte generacije računala. Jednostavnost pri čitanju i razumijevanju glavne su karakteristike programskih jezika tog doba. Programski jezik C nalazi svoju nezamjenjivu primjenu u povijesti komercijalne računalne industrije, budući da je prilagođen za sve računalne platforme.

Optimizacija programskog koda jedna je od bitnijih stavki u programskom inženjerstvu. Pod terminom optimizacije programskog koda smatra se poboljšanje performansi kao što su smanjenje vremena izvođenja programa, učinkovitije korištenje resursa itd. Budući da su ugradbeni računalni sustavi specifični po ograničenjima resursa raspoloživih za izvedbu programa na njima, optimizacija takvih programskih kodova od neizmjerne je važnosti. Memorijska, procesorska i prostorna ograničenja na neki način „prisiljavaju“ razvojne programere na svijest o potpunoj optimizaciji. Aplikacije u kojima je potrebno obrađivati slike u stvarnom vremenu također zahtijevaju potpunu optimizaciju. Razvijanje takvih aplikacija danas se često primjećuje u automobilskoj industriji i medicini, a samo jedan od primjera je *O-Arm* kirurški sustav [1] koji omogućava vizualizaciju anatomije pacijenta u realnom vremenu za vrijeme operacije uz visoku kvalitetu slike, veliko polje snimanja u 2D i 3D tehnici te potencijalno manju radijaciju za kirurško osoblje. Na samom početku 1980-ih, tadašnji autori prevoditelja (*engl. compiler*) smatrali su da je optimizacija izuzetno bitna značajka koja bi trebala biti implementirana unutar samih prevoditelja.

Ovaj završni rad fokusira se na detaljnom opisivanju te implementiranju osnovnih metoda optimizacije kako bi se na primjerima pokazalo poboljšanje performansi. U praksi brzina izvođenja programa nikada nije važnija od ispravnog rada programa. Stoga prilikom optimizacije programskog koda treba izrazito puno pažnje pridodati naknadnim testiranjima potpune ispravnosti. Prilikom izrade programskog koda, fokus programera treba biti na pravilnom dizajniranju i ispravnosti samog programskog koda. Naknadno, slijedi provjera performansi programskog koda te eventualna optimizacija ukoliko se procjeni da ima potrebe za istom. Vrlo

važna napomena, vezana uz optimiziranje programskog koda je da jednom optimizirani programski kod u određenim uvjetima okruženja nužno ne znači da je programski kod optimiziran i za apsolutno sva ostala okruženja. Ciljevi optimizacije programskog koda su: uklanjanje redundantnog (ponavljajućeg) dijela koda bez promjene značenja programa, prilagođavanje brzine izvršavanja te prilagođavanje potrošnje memorije.

Slika 1.1. Balans pri optimizaciji programskog koda

Budući da je proces optimizacije programskog koda ponavljajući, točnije nikada se ne može reći da je programski kod 100 % optimiziran, najbolji će se rezultati postizati kada je pronađen balans između vremena izvođenja i potrošnje memorije [2], kao što je prikazano na slici 1.1..

U završnom se radu objašnjava pojam i prednosti prevoditelja s optimizacijskim mogućnostima novije generacije koji su sve više i više moćni u procesu „nevidljive“ optimizacije programskog koda - ukratko je objašnjen princip njihovog rada. Detaljno je obrazložen princip provedbe svih mjerenja i analiza performansi koristeći moderna razvojna okruženja.

2. PREGLED POSTOJEĆIH RJEŠENJA ZA OPTIMIZACIJU PROGRAMSKOG KODA

Optimizacija programskog koda predstavlja odavno poznat pojam, te su samim time metode i strategije optimizacije programskog koda postojane već dugi niz godina. Razvijanje i širenje *Interneta* kao javno dostupne globalne podatkovne mreže uzrokovalo je i nastanak velikog broja radova, članaka i istraživanja vezano uz optimizaciju programskog koda. U ovom poglavlju dan je kratki opis trenutnih postojećih rješenja vezanih za optimizaciju programskog koda, te njihovih prednosti i nedostataka.

U radu [2], dan je uvod u optimizaciju programskog koda te su navedene izuzetno bitne i provjerene činjenice vezane za optimizaciju programskog koda kao što su:

1. proces optimizacije nije isti - u smislu da je proces optimizacije kompleksan te se ništa ne može uzeti „zdravo za gotovo“, znatno ovisi o velikom broju parametara (samom procesoru, prevoditelju, okruženju) - jedino što je sigurno je da optimizaciju treba dokazati mjerenjima,
2. stopostotna (kompletna) optimizacija nije moguća.

Nadalje, u radu je prikazan velik broj metoda optimizacija (60) što kraćih, što dužih, no glavni nedostatak je nedovoljna obrazloženost samih metoda. Poboljšanja performansi nisu dokazana nikakvim mjerenjima, ne postoji niti jedna usporedba već se sve svodi na razinu „činjeničnog znanja“. Osim toga, optimizacije samog prevoditelja nisu spomenute.

U radu [3], objašnjene su temeljne definicije vezane uz poboljšanje performansi i optimizaciju. Rad je fokusiran na opisivanju strategija i metoda optimizacije koda u programskom jeziku C, te na njihovom implementiranju. Postupkom mjerenja dokazano je poboljšanje performansi programskog koda korištenjem opisanih metoda. Glavni nedostatak rada predstavlja činjenica da je prilikom postupka mjerenja uzeto u obzir samo vrijeme trajanja izvođenja programskog koda, dok zauzeće memorije nije niti spomenuto. Nadalje, u radu su izuzetno kratko obrađene optimizacijske mogućnosti samog prevoditelja (samo spomenute), što također predstavlja nedostatak budući da se nalazimo u vremenu u kojem bi trebali koristiti sve raspoložive prednosti novih razvojnih okruženja, proširenja, prevoditelja itd.

U radu [4], najdetaljnije moguće obrazložen je pojam optimizacije te sami proces optimizacije programskog koda. Kao svojevrstan uvod obrazloženi su konkretni pristupi procesu optimizacije programskog koda različitih programera, dajući tako direktne savjete o metodama, „trikovima“ i

sl. Iako su konkretne metode također obrazložene detaljno, glavni nedostatak rada predstavlja izuzetno mali broj obrađenih metoda. Ono što rad čini drukčijim od ostalih je i korištenje *CCCC (C and C++ Code Counter)* koji predstavlja alat za analizu programskog koda u različitim programskim jezicima, generirajući *HTML (HyperText Markup Language)* izvješće o različitim provedenim mjerenjima.

U radu [5], ukratko je objašnjen pojam optimizacije. Glavni dio rada fokusiran je na opisivanju različitih pristupa ručnoj optimizaciji programskog koda. Glavni nedostatak predstavlja način na koji su metode obrađene, točnije sve što je obrađeno je na razini „činjeničnog znanja“. Prednost rada predstavlja činjenica da je obrađeno nekoliko svrsishodnih konkretnih primjera programskih kodova koji se koriste u „svakodnevnom životu“, na njima su primijenjene metode optimizacije programskog koda i uspoređeni su rezultati prije i poslije, te je na taj način jasno pokazano poboljšanje performansi programskog koda.

3. METODE OPTIMIZACIJE

U ovom poglavlju obrađene su metode optimizacije programskog koda te je opisan način provođenja mjerenja. Svaka od metoda optimizacije programskog koda detaljno je obrađena na način da je dan kratki opis, programski kodovi prije i nakon optimizacije koji se nalaze u P.3.1. te usporedba performansi (vrijeme izvođenja i analiza zauzeća memorije programskog koda) prije i nakon optimizacije.

3.1. PROVEDBA MJERENJA I ANALIZA PERFORMANSI

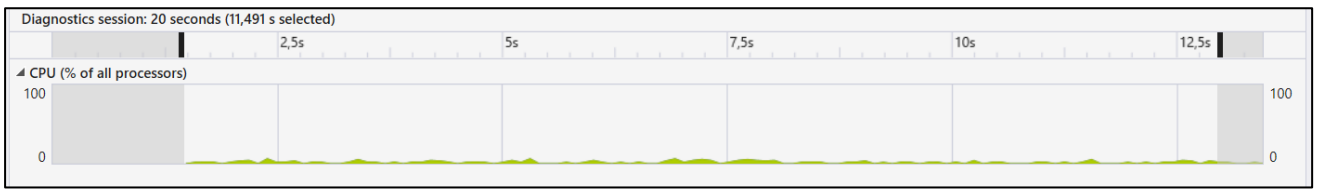
3.1.1. Razvojno okruženje

Integrirano razvojno okruženje koje je korišteno u završnom radu je *Microsoft Visual Studio 2019 Community*. Sva testiranja u završnom radu provedena su na prijenosnom računalu Acer Aspire 5, sa slijedećim karakteristikama: procesor Intel Core i5-8250U 3.4GHz, memorija 8 GB DDR4 te Solid-state 256 GB disk. Za dodatne podatke (analizu performansi) korišten je alat *Performance profiler,* koji je potrebno instalirati naknadno, a predstavlja proširenje osnovnih mogućnosti razvojnog okruženja. U slijedećim potpoglavljima detaljno su opisane mogućnosti *Performance profilera* koje su korištene pri testiranju.

3.1.2. Performance profiler

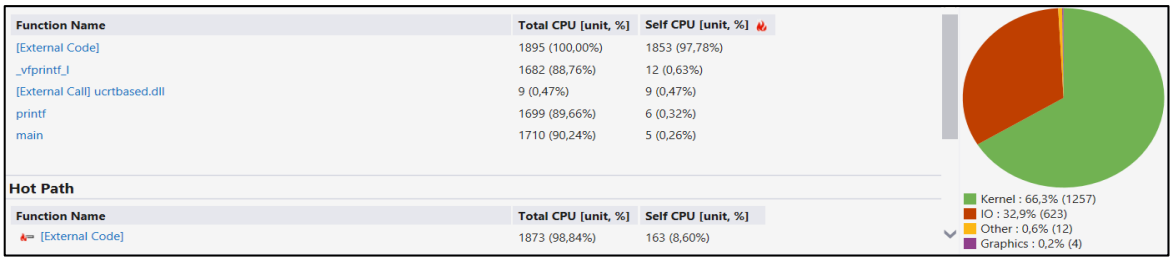
3.1.2.1. Analiza upotrebe središnje jedinice za obradbu (procesora)

Jedan od polaznih koraka pri provjeri performansi programskog koda je analiza upotrebe procesora. Ona nam prikazuje gdje točno procesor troši vrijeme izvršavajući programski kod. Funkcionalnost je to koja je od neizmjerne koristi pri identifikaciji resursa koji ograničavaju brzinu i prohodnost izvođenja (*engl. bottlenecks*). Slika 3.1. prikazuje iskorištenost procesora u postotcima (%) kroz određeno vrijeme izvođenja programskog koda.



Slika 3.1. Iskorištenost procesora kroz određeno vrijeme izvođenja

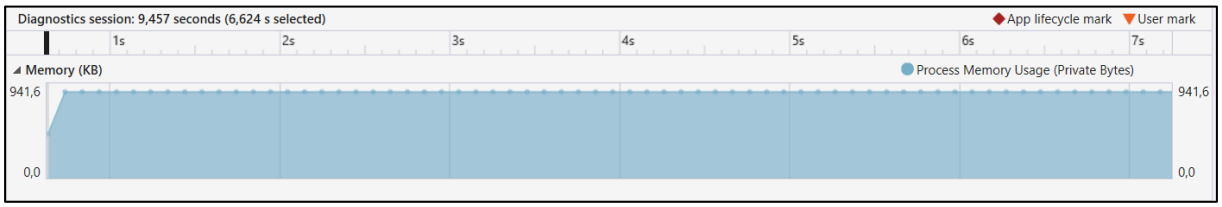
Na dijagramu je moguće odabrati proizvoljan vremenski interval, te na osnovu odabira prikazuju se točni postotci koliko i što (koje funkcije) u tom trenutku opterećuje procesor, kao što je vidljivo na slici 3.2..



Slika 3.2. Podaci o opterećenosti procesora

3.1.2.2. Analiza upotrebe memorije

Alat sa svrhom analize upotrebe memorije pri izvođenju programskog koda omogućava istraživanje iracionalnog korištenja memorije, od manjih „curenja“ (*engl. memory leaks*) pa sve do ekstremnijih - koji će uzrokovati rušenje programa. Problem nastaje u činjenici da svaki puta kada se zatraži više memorije, program uzima dodatni RAM umjesto korištenja memorije koja je programu već stavljena na raspolaganje. Slika 3.3. prikazuje upotrebljivost memorije izražene u KB kroz određeni vremenski period.



Slika 3.3. Podaci o upotrebljenosti memorije kroz određeno vrijeme izvođenja

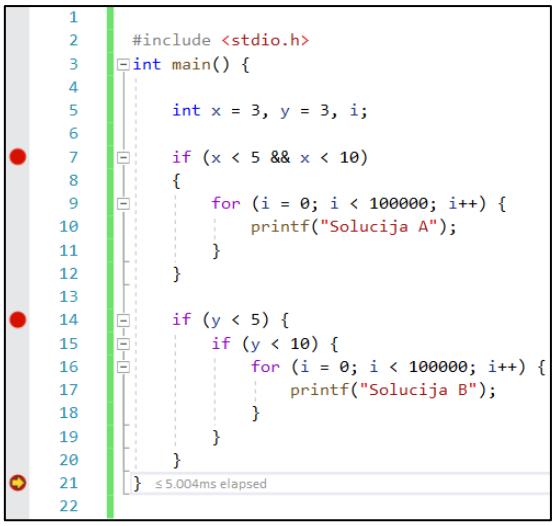
3.1.2.3. Mjerenje vremena izvođenja programa

Podatak o vremenu izvođenja programa jedan je od temeljnih podataka pri mjerenju performansi programskog koda. Postoje razni načini mjerenja vremena izvođenja programskog koda (ili određenog dijela programskog koda) kao što su npr. mjerenje vremena izvođenja programa pomoću `time()` funkcije, `clock()` funkcije, itd. Također, postoji mogućnost ručnog mjerenja vremena izvođenja programskog koda npr. štopericom. No, ne preporučuje se takav pristup budući da su moguće greške mjeritelja, što na kraju može rezultirati eventualnim donošenjem krivih zaključaka.

U završnom su radu korištene točke prekida (*engl. breakpoints*) koji dolaze u osnovnom sklopu dijagnostike programskog koda (*engl. debugger*), te omogućuju:

- jednostavno dodavanje jednim klikom na bilo koji segment/dio programskog koda,
- jednostavnost pri analizi rezultata mjerenja.

Na slici 3.4. prikazan je primjer korištenja točke prekida.



Event	Time	Duration	Thread
Breakpoint: Source.c line 7	0.01s	17ms	[16536]
Breakpoint: Source.c line 14	9.83s	9,816ms	[16536]
Breakpoint: Source.c line 21	14.83s	5,004ms	[16536]

Slika 3.4. Primjer korištenja točki prekida

3.2. Optimizacija petlji

3.2.1. Odmotavanje petlji

Odmotavanje petlji (*engl. loop unrolling*) predstavlja tehniku koja za cilj ima optimizaciju brzine izvođenja programskog koda smanjivanjem ili eliminiranjem instrukcija koje kontroliraju tok petlje.

Primjer:

- Normalna petlja izvršava se 100 000 puta, te pritom poziva određenu funkciju *fja(i)*, kao što je vidljivo na slici 3.5..

Linija ***Kod***

```
1:      int i;
2:      for (i = 0; i < 100000; i++)
3:      {
4:        fja(i);
5:      }
```

Slika 3.5. Kod za odmotavanje petlji prije optimizacije

- Optimizirana petlja izvršava se samo 20 000 puta, kao što je vidljivo na slici 3.6..

Linija ***Kod***

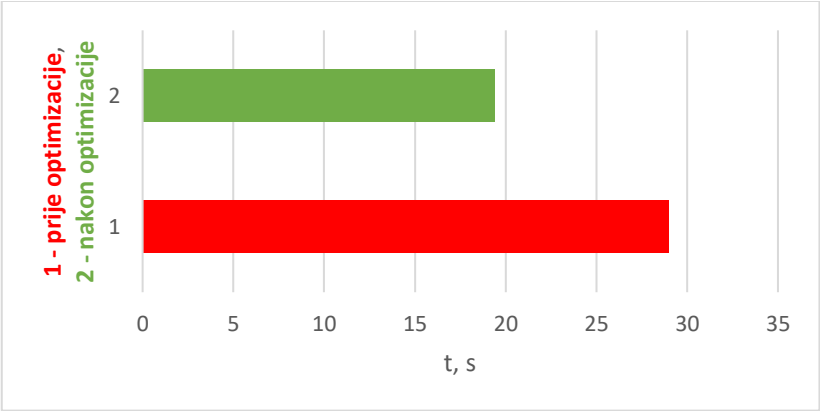
```
1:      int i;
2:      for (i = 0; i < 100000; i+=5)
3:      {
4:        fja(i);
5:        fja(i+1);
```

```
6:          fja(i+2);
7:          fja(i+3);
8:          fja(i+4);
9:      }
```

Slika 3.6. Kod za odmotavanje petlji nakon optimizacije

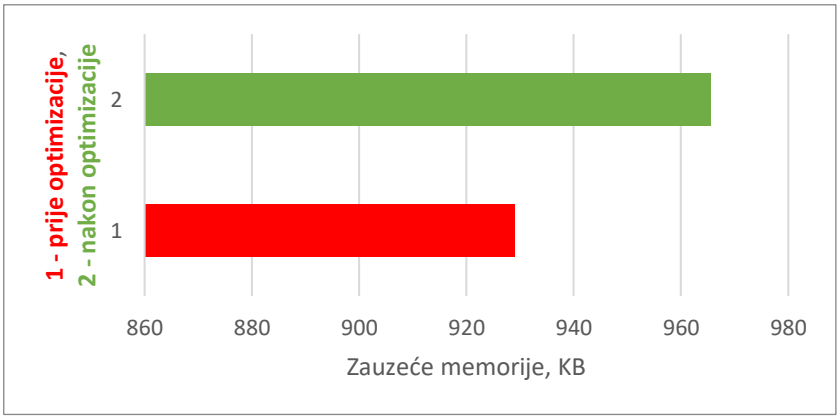
Rezultati:

- Na 100 000 iteracija, uočava se smanjenje vremena izvođenja programa za otprilike 33 %, kao što je vidljivo na slici 3.7..



Slika 3.7. Odnos vremena izvođenja programa prije i nakon odmotavanja petlje

- S druge strane, kod optimiziranog koda uočava se povećanje zauzeća memorije za otprilike 5 %, kao što je vidljivo na slici 3.8..



Slika 3.8. Odnos zauzeća memorije programa prije i nakon odmotavanja petlje

3.2.2. Fuzija petlji

Fuzija petlji (*engl. loop fusion ili loop jamming*) predstavlja tehniku koja za cilj ima zamijeniti više petlji s jednom. Njeno je korištenje izuzetno pogodno u slučajevima kada se manipulira s dva ili više polja iste veličine. Pritom, valja napomenuti da fuzija petlji ne poboljšava performanse uvijek, budući da će u specijalnim slučajevima spajanjem petlji biti narušena lokalizacija pri

pristupu podacima u memoriji (npr. ukoliko su postojane dvije ili više varijable koje su locirane na potpuno različitim mjestima u memoriji).

Primjer:

- Jednostavan primjer u kojem se na određene vrijednosti inicijaliziraju svi elementi tri različita polja iste duljine pomoću 3 `for` petlje, kao što je prikazano na slici 3.9..

Linija ***Kod***

```
1:      int i, array1[100000], array2[100000], array3[100000];
2:      for (i = 0; i < 100000; i++){
3:         array1[i] = -1;
4:       }
5:      for (i = 0; i < 100000; i++){
6:         array2[i] = 0;
7:      }
8:      for (i = 0; i < 100000; i++){
9:         array3[i] = 1;
10:     }
```

Slika 3.9. Kod za fuziju petlji prije optimizacije

- optimizirana petlja predstavlja spoj gore navedene tri petlje, kao što je prikazano na slici 3.10..

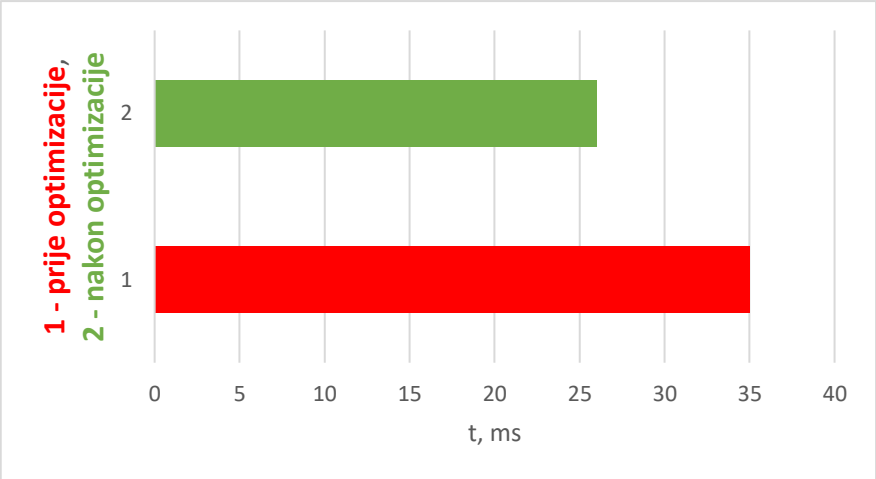
Linija ***Kod***

```
1:      int i, array1[100000], array2[100000], array3[100000];
2:      for (i = 0; i < 100000; i++){
3:      {
4:           array1[i] = 2;
5:           array2[i] = 0;
6:           array3[i] = 1;
7:      }
```

Slika 3.10. Kod za fuziju petlji nakon optimizacije

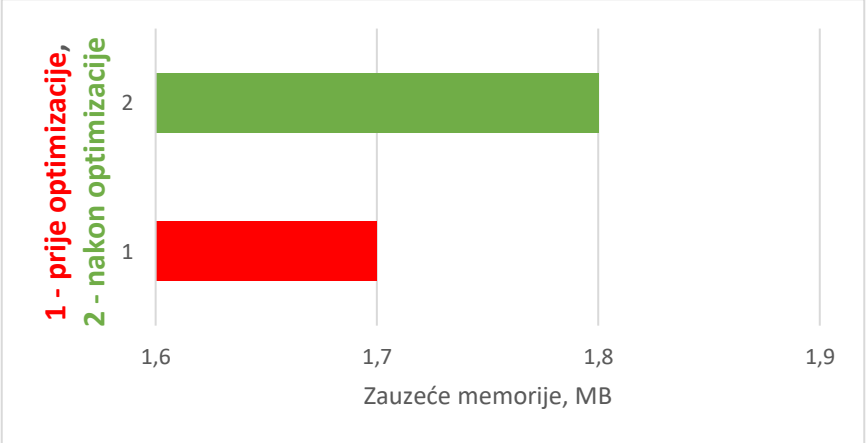
Rezultati:

- Na 100 000 iteracija, uočava se poboljšanje performansi u vidu smanjenja vremena izvođenja programa za 25 %, kao što je vidljivo na slici 3.11..



Slika 3.11. Odnos vremena izvođenja programa prije i nakon fuzije petlji

- S druge strane, kod optimiziranog koda uočava se povećanje zauzeća memorije za otprilike 6 %, kao što je vidljivo na slici 3.12..



Slika 3.12. Odnos zauzeća memorije programa prije i nakon fuzije petlje

3.2.3. Korištenje dekrementirajuće petlje

Budući da je uspoređivanje sa nulom drukčija operacija od uspoređivanja sa nekim drugim brojem (u smislu drukčije interpretacije unutar samih čipova) korištenje dekrementirajuće *for* petlje povećati će brzinu izvođenja programskog koda. Konkretno, na primjeru, broj 100 000 mora biti dohvaćen iz memorije prilikom svake iteracije kako bi se provjerio uvjet. S druge strane, broj 0 predstavlja „specijalnu vrijednost“ budući da većina procesora već ima ugrađene instrukcije za usporedbu brojeva sa 0.

Primjer:

- Ispisivanje *„Testiranje“* 100 000 puta inkrementirajućom *for* petljom, kao što je prikazano na slici 3.13..

Linija ***Kod***

```
1:      for (i = 0; i < 100000; i++)
2:      {
3:         printf(„Testiranje“);
4:      }
```

Slika 3.13. Kod za ispis pomoću inkrementirajuće petlje

- Ispisivanje *„Testiranje“* 100 000 puta dekrementirajućom *for* petljom, kao što je prikazano na slici 3.14..

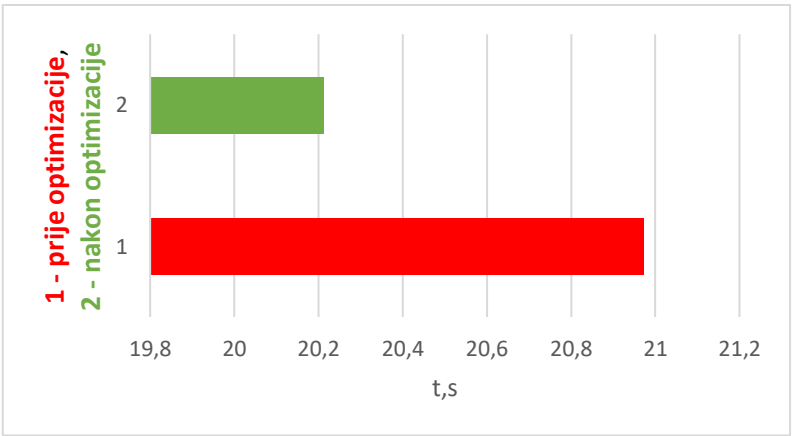
Linija ***Kod***

```
1:      for (i = 100000; i > 0; i--)
2:      {
3:         printf(„Testiranje“);
4:      }
```

Slika 3.14. Kod za ispis pomoću dekrementirajuće petlje

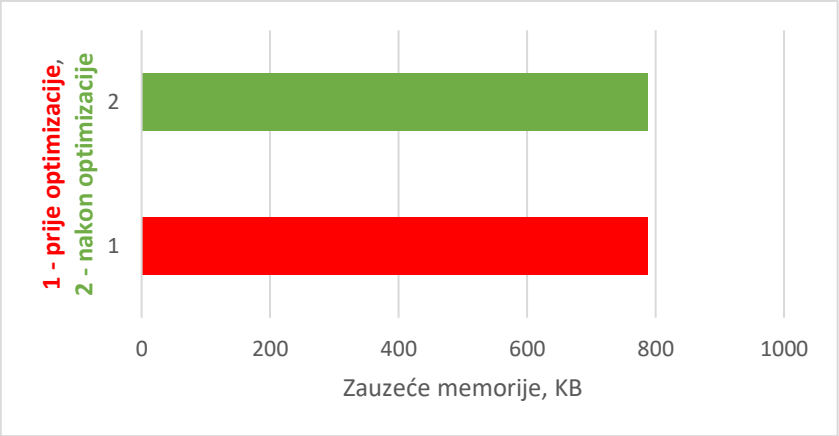
Rezultati:

- Na 100 000 iteracija, uočavaju se manja poboljšavanja performansi u vidu smanjenja vremena izvođenja programskog koda za otprilike 3.6 %, kao što je vidljivo na slici 3.15..



Slika 3.15. Odnos vremena izvođenja programa koristeći inkrementirajuću i dekrementirajuću petlju

- Oba programska koda imaju jednako zauzeće memorije, kao što je vidljivo na slici 3.16.



Slika 3.16. Odnos zauzeća memorije programa koristeći inkrementirajuću i dekrementirajuću petlju

3.2.4. „Razbijanje“ ciklusa unutar petlje

Razbijanje ciklusa (*engl. switching*) petlje odnosi se na donošenje odluka unutar petlji svaki puta kada se petlja izvrši [4]. Ukoliko se odluka ne mijenja tokom izvođenja petlje, potrebno je „razbiti“ petlju donošenjem odluke izvan petlje.

Primjer:

- *IF* naredba nalazi se unutar *for* petlje, kao što je prikazano na slici 3.17..

Linija ***Kod***

```
1:      for (i = 0; i < 100000; i++) {
2:        if(x) {
3:           a[i] = 0; printf(„Element %d polja A postavljen na 0“, i); }
4:        else {
5:           b[i] = 0; printf(„Element %d polja B postavljen na 0“, i); }}
```

Slika 3.17. Kod za testiranje razbijanja ciklusa – if uvjet unutar for petlje

- Nakon transformacije, *IF* naredba nalazi se izvan *for* petlje, kao što je prikazano na slici 3.18..

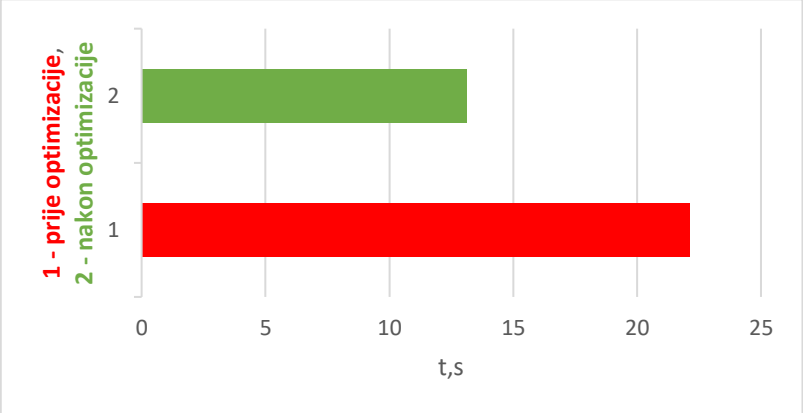
Linija ***Kod***

```
1:      if(x)
2:         for (i = 0; i < 100000; i++) {
3:           a[i] = 0; printf(„Element %d polja A postavljen na 0“, i); }
4:       else
5:         for (i = 0; i < 100000; i++) {
6:           b[i] = 0; printf(„Element %d polja B postavljen na 0“, i); }
```

Slika 3.18. Kod za testiranje razbijanja ciklusa– if uvjet izvan for petlje

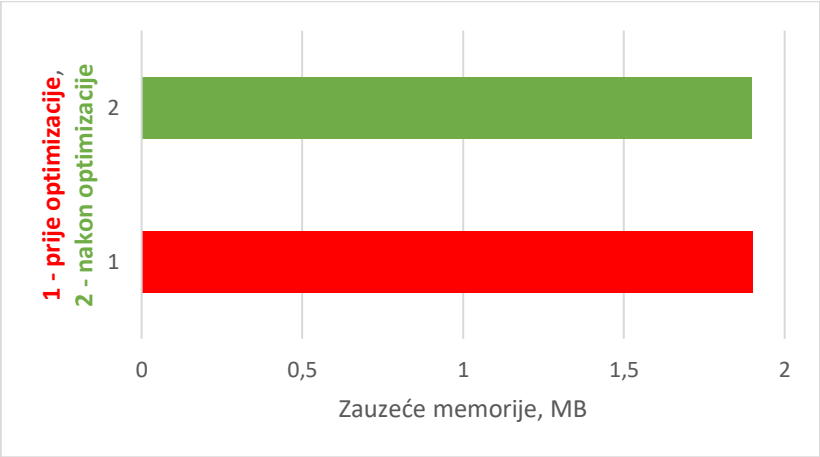
Rezultati:

- kao što je prikazano na slici 3.19., vidljivo je poboljšanje performansi programskog koda u vidu smanjenja vremena izvođenja programskog koda za gotovo 40 %.



Slika 3.19. Odnos vremena izvođenja programa prije i nakon primjene metode razbijanja ciklusa

- Oba programska koda imaju jednako zauzeće memorije, kao što je vidljivo na slici 3.20.



Slika 3.20. Odnos zauzeća memorije programa koristeći metodu razbijanja ciklusa

3.3. Optimizacija polja

Upotreba polja većih dimenzija je „skupa“ [6] te ukoliko postoji mogućnost za npr. strukturiranje podataka u jednodimenzionalno polje umjesto dvodimenzionalno, tada postoji šansa za uštedu određenog vremena.

Primjer:

- Inicijaliziraju se podaci dvodimenzionalnog polja dimenzija `100 x 50` na klasičan način, kao što je prikazano na slici 3.21..

Linija ***Kod***

```
1:      int polje[100][50], i, j;
2:         for (i = 0; i < 100; i++) {
3:             for(j = 0; j < 50; j++) {
4:                  polje[i][j] = 0;
5:              }
6:         }
```

Slika 3.21. Kod za testiranje optimizacije polja – 2D polje

- Dvodimenzionalno polje sada se pretvara u jednodimenzionalno polje sa dimenzijama `100 x 50`, te se podaci inicijaliziraju također na klasičan način, kao što je prikazano na slici 3.22..

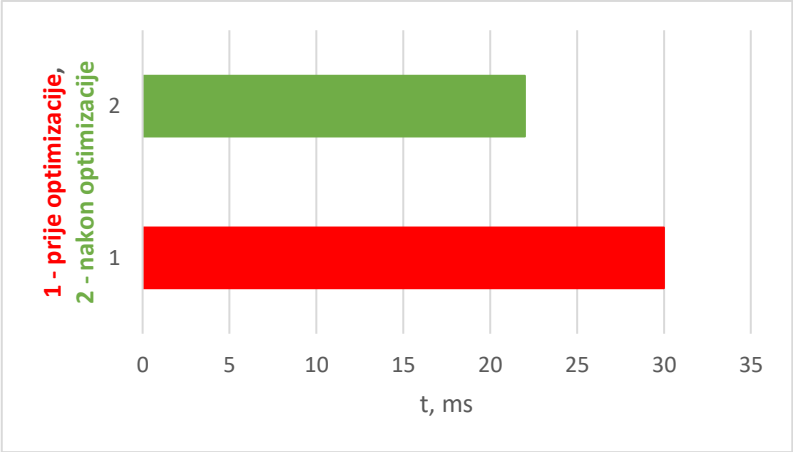
Linija ***Kod***

```
1:      int polje[5000], i;
2:         for (i = 0; i < 5000; i++) {
3:              polje[i] = 0;
4:         }
```

Slika 3.22. Kod za testiranje optimizacije polja – 1D polje

Rezultati:

- Kao što je prikazano na slici 3.23., vidljivo je poboljšanje performansi u vidu kraćeg izvođenja programa za otprilike 27 %.



Slika 3.23. Odnos vremena izvođenja programa koristeći metodu za optimizaciju polja

3.4. Smanjenje snage

Pod smanjenjem snage (*engl. strength reduction*) smatra se korištenje „jeftinijih“ operacija umjesto „skupljih“. Pod terminom „jeftinijih“ smatraju se operacije koje daju isti rezultat, a lakše (brže) se izračunavaju.

Primjer:

- Polje cjelobrojnih brojeva `y[i]` takvo da je varijabla `a = 4` množena sa indukcijskom varijablom `i`, kao što je prikazano na slici 3.24..

Linija ***Kod***

```
1:      int y[100000], i, a = 4;
2:      for (i = 0; i < 100000; i++) {
3:           y[i] = a * i;
4:           printf(„%d „, y[i]); }
```

Slika 3.24. Kod za testiranje smanjenja snage prije optimiziranja

- Isti ishod postiže se sa uzastopnim zbrajanjem, kao što je prikazano na slici 3.25..

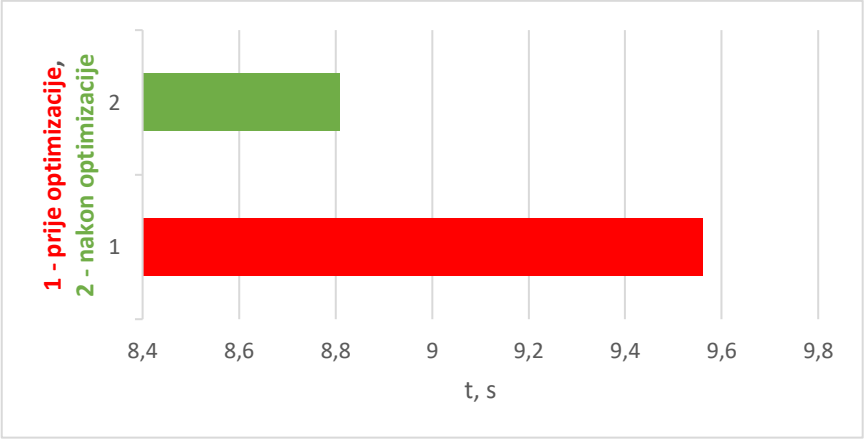
Linija ***Kod***

```
1:      int y[100000], i, a = 4, b = 0;
2:      for (i = 0; i < 100000; i++) {
3:           y[i] = b;
4:           b = b + a;
5:           printf(„%d „, y[i]); }
```

Slika 3.25. Kod za testiranje smanjenja snage nakon optimiziranja

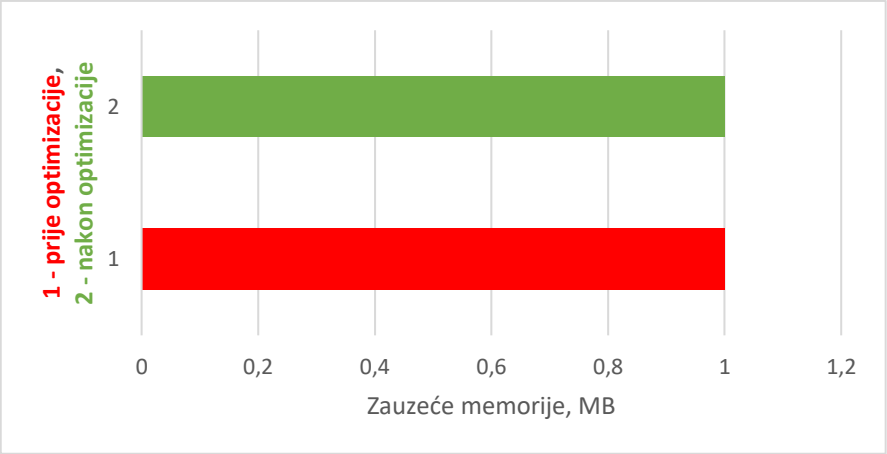
Rezultati:

- Kao što je prikazano na slici 3.26., vidljivo je poboljšanje performansi u vidu smanjenja vremena izvođenja programskog koda za otprilike 8 %.



Slika 3.26. Odnos vremena izvođenja programa koristeći metodu za smanjenje snage

- Oba programska koda imaju jednako zauzeće memorije, kao što je vidljivo na slici 3.27..



Slika 3.27. Odnos zauzeća memorije programa koristeći metodu smanjenja snage

U tablici 3.1. dani su ostali primjeri zamjene koje je potrebno koristiti u programskom kodu.

Izvorno:	Zamjeniti s:
`a = b / 16`	`a = b >> 4`
`a = b * 128`	`a = b << 7`
`a = b * 2`	`a = b << 1`
`a = b * 15`	`a = ( b << 4 ) - b`

Tablica 3.1. Primjeri zamjene kod reduciranja snage

3.5. Logika

Pod terminom logike smatra se pravilnije pisanje logičkih izraza [3]. Npr. izraz:

```
if (a < 100 && a < 150)
```

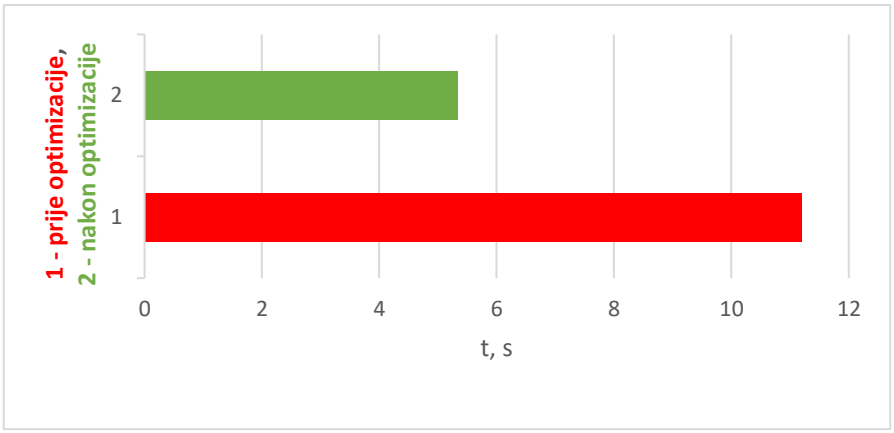
moguće je pravilnije napisati:

```
if (a < 100){
if(a < 150) { }}
```

budući da je broj manji od 100 zasigurno manji od 150. Logika ne predstavlja klasičnu metodu optimizacije programskog koda, no programeri bi svakako morali voditi računa o takvim stvarima, budući da se testiranjima ustanovilo da se vrijeme izvršavanja programskog koda može znatno smanjiti.

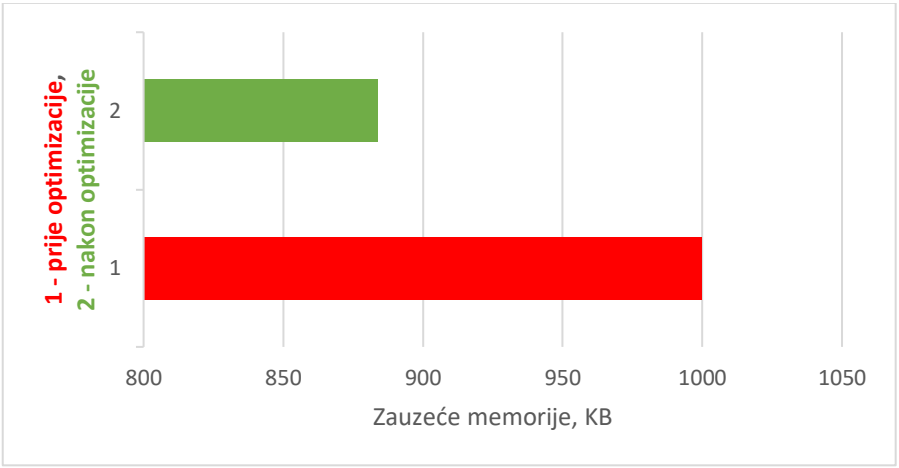
Rezultati:

- Kao što je prikazano na slici 3.28., vidljivo je poboljšanje performansi u vidu smanjenja vremena izvođenja programskog koda za gotovo 53 %.



Slika 3.28. Odnos vremena izvođenja programa koristeći metodu logike

- Osim toga, poboljšanje performansi očituje se i u smanjenom zauzeću memorije za otprilike 11 %, kao što je prikazano na slici 3.29..



Slika 3.29. Odnos zauzeća memorije programa koristeći metodu logike

3.6. Prevoditelji s optimizacijskim mogućnostima

GNU Compiler Collection (*skraćeno: GCC*) [7] predstavlja skup besplatnih prevoditelja s optimizacijskim mogućnostima. Uključivanjem optimizacijskih zastavica prevoditelj pokušava poboljšati performanse programskog koda. *GCC* pruža razine optimizacije (numerirane od 0 do 3), sa različitim učincima:

- O0 – ne vrši se nikakva optimizacija, prevoditelj prevodi izvorni kod na najjednostavniji mogući način,
- O1 – uključuje najčešće optimizacije,
- O2 – uz sve optimizacije prethodne razine, uključuje i dodatne optimizacije, najčešće se koristi budući da pruža maksimalnu optimizaciju bez povećanja veličine izvršne datoteke,

- O3 – uz sve optimizacije prethodnih razina, povećava brzinu izvršavanja programskog koda, ali također povećava i veličinu izvršne datoteke,
- funroll-loops – uključuje odmotavanje petlji, neovisna je od ostalih optimizacijskih razina,
- Os – ova razina ima za cilj smanjenje veličine izvršne datoteke (npr. za sustave sa ograničenim memorijskim resursima).

Prema [8], prikazan je primjer optimizacije pomoću prevoditelja na način da se prevoditelju proslijedi zastavica za optimizaciju `(O1, O2 ili O3)`.

Primjer:

- programski kod koji za svrhu ima prebrojati broj pojavljivanje znaka `'X'` u određenom nizu znakova, kao što je prikazano na slici 3.30..

Linija ***Kod***

```
1:      const size_t n = 100000;
2:      int i, cnt = 0;
3:      char *t = malloc(n);
4:      memset(t, 'X', n - 1);
5:      t[n-1] = '\0';
6:      for (i = 0; i < strlen(t); ++i) {
7:           cnt += (t[i] == 'X');
8:        }
```

Slika 3.30. Kod za prebrojavanje broja pojavljivanja znaka 'x'

Problem predstavlja funkcija `strlen`, budući da se prilikom pojedine iteracije `for` petlje svaki puta izračunava `strlen(t)`, čineći dodatnih 100 000 koraka svaki puta. Spremajući vrijednost `strlen(t)` u određenu varijablu (npr. `y`) , te korigiranjem `for` petlje na način: `for (i = 0; i < y; ++i)` programski kod bi bio optimiziran budući da bi se izračun `strlen(t)` izvršio samo jednom.

Prevoditelj će upravo to i napraviti kada mu se proslijedi zastavica za optimizaciju `(O1, O2 ili O3)`. U konkretnom primjeru, performanse su se poboljšale čak i na najnižoj razini `O1`, pri čemu se vrijeme izvođenja programskog koda skratilo za gotovo 500 puta.

3.7. Korištenje tipova podataka bez predznaka umjesto tipova podataka sa predznakom

Ukoliko je moguće, potrebno je koristiti `unsigned` tipove podataka umjesto `signed` budući da većina procesora brže obrađuje varijable koje su bez predznaka. Koristi se npr. ukoliko se zna da vrijednost varijable nikad neće biti negativna [2].

Primjer:

- Kreirane su 2 funkcije `fja1` i `fja2` koje kao argument primaju cijeli broj `(integer)` – jedna sa predznakom `(signed)`, druga bez predznaka `(unsigned)` te vraćaju broj dijeljen sa 8, kao što je prikazano na slici 3.31..

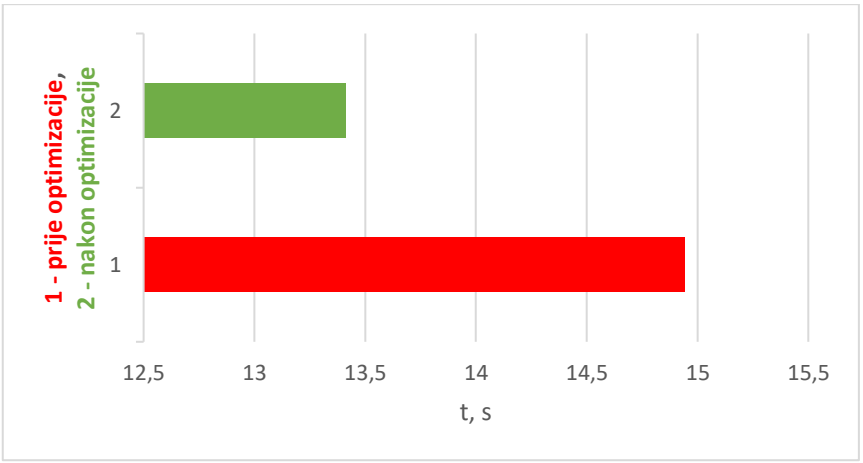
Linija ***Kod***

```
1:      signed int fja1(signed int x) {
2:        return x /= 8;
3:      }
4:      unsigned int fja2(unsigned int x) {
5:        return x /= 8;
6:      }
```

Slika 3.31. Kod za testiranje brzine izvođenja programa sa varijablama sa/bez predznaka

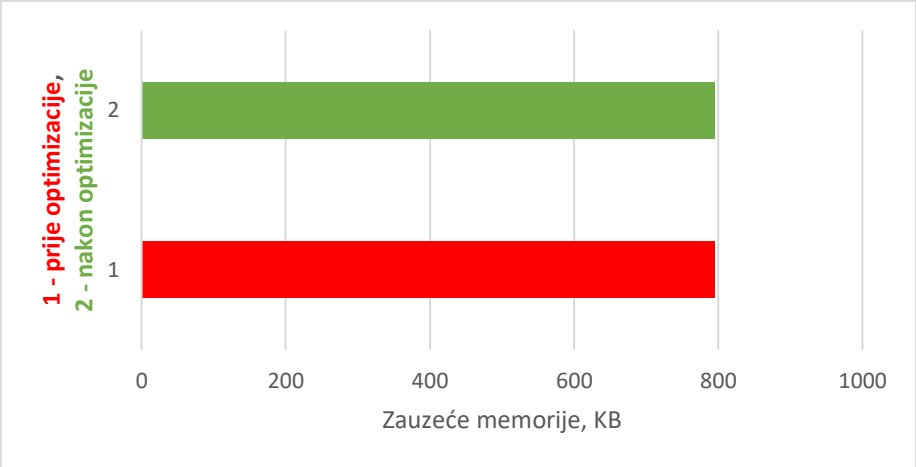
Rezultati:

- Na 100 000 iteracija, vidljivo je poboljšanje performansi u vidu kraćeg vremena izvođenja programa za otprilike 10 %, kao što je vidljivo na slici 3.32..



Slika 3.32. Odnos vremena izvođenja programa koristeći signed i unsigned tipove podataka

- Oba programska koda imaju jednako zauzeće memorije, kao što je vidljivo na slici 3.33..



Slika 3.33. Odnos zauzeća memorije programa koristeći unsigned i signed tipove podataka

3.8. Korištenje cijelih brojeva umjesto decimalnih brojeva sa pomičnim zarezom

Operacije sa cijelim brojevima `(integer)` znatno su brže od operacija sa decimalnim brojevima `(float)`, te ih treba koristiti ukoliko je to u programskom kodu moguće.

Primjer:

- Deklarirano je polje cijelih brojeva od 100 000 elemenata, te su na ekran ispisane vrijednosti pojedinih elemenata polja pomnožene sa brojem 10, kao što je prikazano na slici 3.34..

Linija ***Kod***

```
1:      int polje1[100000], i;
2:      for (i = 0; i < 100000; i++) {
3:          printf(„%d", polje1[i] * 10);
4:      }
```

Slika 3.34. Kod za deklariranje i manipulaciju vrijednosti pojedinih elemenata polja integer-a

- Nadalje, deklarirano je polje jednake veličine, ali tipa `float`, te su na ekran ispisane vrijednosti pojedinih elemenata polja pomnožene sa brojem 10, kao što je prikazano na slici 3.35..

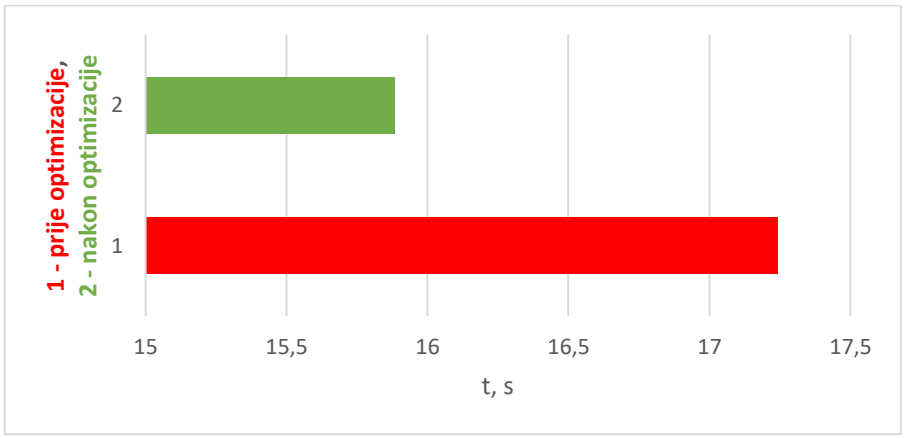
Linija ***Kod***

```
1:      float polje1[100000]; int i;
2:      for (i = 0; i < 100000; i++) {
3:          printf(„%0f", polje1[i] * 10);
4:      }
```

Slika 3.35. Kod za deklariranje i manipulaciju vrijednosti pojedinih elemenata polja float-a

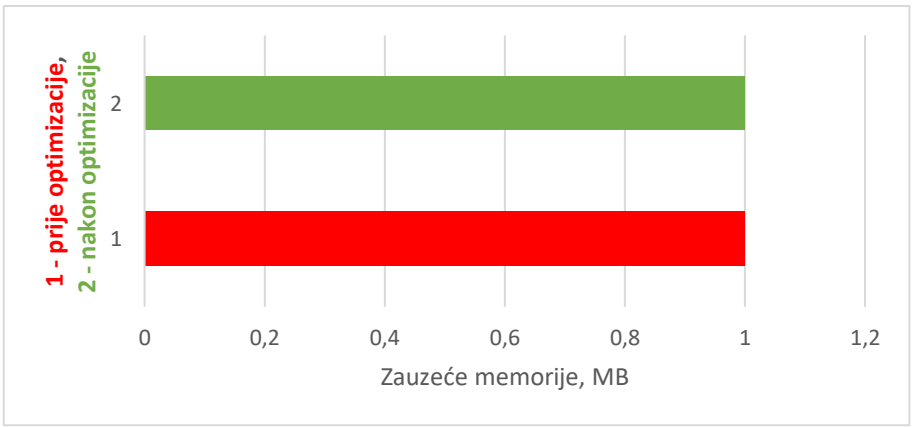
Rezultati:

- Na 100 000 iteracija, vidljivo je poboljšanje performansi u vidu kraćeg vremena izvođenja programa za otprilike 8 %, kao što je vidljivo na slici 3.36..



Slika 3.36. Odnos vremena izvođenja programa koristeći cijele brojeve umjesto decimalnih brojeva sa pomičnom točkom

- Oba programska koda imaju jednako zauzeće memorije, kao što je vidljivo na slici 3.37..



Slika 3.37. Odnos zauzeća memorije programa koristeći cijele brojeve umjesto decimalnih brojeva sa pomičnom točkom

3.9. Aritmetika sa cijelim brojevima

Aritmetika s cijelim brojevima predstavlja metodu koja za cilj ima zamjeniti aritmetiku brojeva s pomičnim zarezom. Razlog postojanja metode leži u činjenici da je aritmetika brojeva s pomičnim zarezom sporija od aritmetike s cijelim brojevima. Drugim riječima, pohranjuju se vrijednosti određene varijable pomnožene sa određenim iznosom (tzv. faktorom skaliranja), te takvu varijablu iskoristimo za obavljanje aritmetičkih operacija. Kada su sve aritmetičke operacije obavljene, pomoću faktora skaliranja retroaktivno dolazimo do rezultata. U programskom jeziku C koriste se operatori pomaka bitova `(<< i >>)`, te se za faktor skaliranja uzimaju se potencije broja 2. Prilikom operacija zbrajanja i oduzimanja, vrijednost koja se dodaje ili oduzima također mora biti pomaknuta (kako bi se „poravnale“ decimalne točke početnog broja i broja kojeg se želi zbrojiti ili oduzeti). S druge strane, prilikom operacija množenja i dijeljenja nije potrebno pomicati bitove.

Primjer:

- Deklarirana je varijabla `x` tipa `float`, te joj je vrijednost inicijalizirana na 101. Na 100 000 iteracija, nad njom se obavljaju tri operacije: zbrajanje, množenje i dijeljenje, kao što je prikazano na slici 3.38..

Linija ***Kod***

```
1:      float x, int i;
2:      for (i = 0; i < 100000; i++) {
3:          x = 101;
4:          x += 5;
5:          x *= 2;
6:          x /= 7.5;
7:          printf („%.3f „, x);
```

Slika 3.38. Kod za testiranje aritmetike sa cijelim brojevima – prije optimizacije

- S druge strane, deklarirana je varijabla `x` tipa `integer`, te joj je vrijednost također inicijalizirana na 101. Na 100 000 iteracija, nad njom se obavljaju tri operacije: zbrajanje, množenje i dijeljenje, kao što je prikazano na slici 3.39..

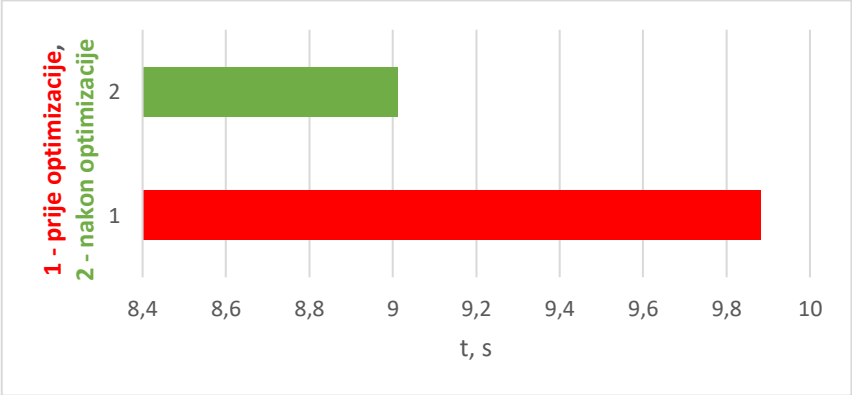
Linija ***Kod***

```
        #define SHIFT_AMOUNT 16
        #define SHIFT_MASK ((1 << SHIFT_AMOUNT) - 1)
1:      int x, i;
2:      for (i = 0; i < 100000; i++) {
3:      x = 101 << SHIFT_AMOUNT;
4:      x += 5 << SHIFT_AMOUNT;
5:      x *= 2;
6:      x /= 7.5;
7:      printf("%d.%lld ", x >> SHIFT_AMOUNT, (long long)
        (x & SHIFT_MASK) * 1000 / (1 << SHIFT_AMOUNT));
```

Slika 3.39. Kod za testiranje aritmetike sa cijelim brojevima – nakon optimizacije

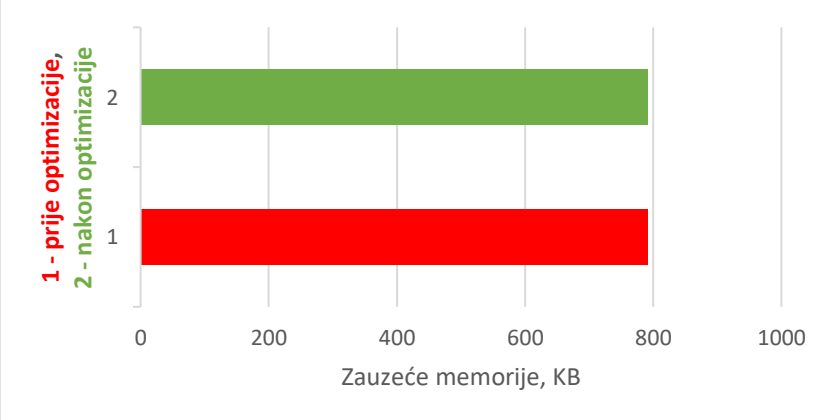
Rezultati:

- Na 100 000 iteracija, vidljivo je poboljšanje performansi u vidu kraćeg vremena izvođenja programa za otprilike 9 %, kao što je vidljivo na slici 3.40..



Slika 3.40. Odnos vremena izvođenja programa koristeći aritmetiku sa cijelim brojevima

- Oba programska koda imaju jednako zauzeće memorije, kao što je vidljivo na slici 3.41..



Slika 3.41. Odnos zauzeća memorije koristeći aritmetiku sa cijelim brojevima

3.10. Pregled rezultata mjerenja

U tablici 3.2. dan je kratki pregled na rezultate mjerenja - prikazana su poboljšanja performansi pojedinih metoda.

Metoda	Performanse
Odmotavanje petlji	Vrijeme izvođenja **smanjeno** za: 33 % Zauzeće memorije **povećano** za: 5 %
Fuzija petlji	Vrijeme izvođenja **smanjeno** za: 25 % Zauzeće memorije **povećano** za: 6 %
Korištenje dekrementirajuće petlje	Vrijeme izvođenja **smanjeno** za: 3.6 % Zauzeće memorije ostalo isto.
Optimizacija polja	Vrijeme izvođenja **smanjeno** za: 27 % Zauzeće memorije ostalo isto.
Smanjenje snage	Vrijeme izvođenja **smanjeno** za: 8 % Zauzeće memorije ostalo isto.
Logika	Vrijeme izvođenja **smanjeno** za: 53 % Zauzeće memorije **smanjeno** za: 11 %
Tipovi podataka bez predznaka umjesto sa predznakom	Vrijeme izvođenja **smanjeno** za: 10 % Zauzeće memorije ostalo isto.
Cijeli brojevi umjesto decimalnih s pomičnom točkom	Vrijeme izvođenja **smanjeno** za: 8 % Zauzeće memorije ostalo isto.
Aritmetika sa cijelim brojevima	Vrijeme izvođenja **smanjeno** za: 9 % Zauzeće memorije ostalo isto.

Tablica 3.2. Tablični pregled svih rezultata mjerenja

4. ZAKLJUČAK

Glavni dio završnog rada bavio se problematikom metoda optimizacije u programskom jeziku C. Optimizacija programskog koda zasigurno je segment na čijem će se napretku i u budućnosti raditi. Fokus je bio na opisivanju osnovnih metoda optimizacije programskog koda, kratkom objašnjenju pojedine metode te usporedbe performansi prije i nakon optimizacije. Na primjerima i kroz različite metode prikazano je poboljšanje performansi u vidu učinkovitijeg korištenja resursa. Predložena osnovna rješenja za optimizaciju programskog koda, kao što je u završnom radu dokazano, mogu pomoći u shvaćanju procesa optimizacije programskog koda. No, baza podataka svih metoda optimizacije programskog koda je izuzetno velika te je potrebno određeno vrijeme prakticiranja kako bi se razvio određeni nivo vještina koji bi eventualno omogućio bržu, lakšu i bolju manipulaciju sa programskim kodom u smislu bržeg pronalaska potencijalnih kritičnih dijelova koda koje je potrebno optimizirati. Uvelike nam u tome pomažu novije mogućnosti razvojnih okruženja. Prevoditelji s optimizacijskim mogućnostima, kao što je navedeno u završnom radu, predstavljaju nemjerljivu pomoć. Izuzetno su moćni, te se na njihovom napretku i dalje radi kako bi se globalno olakšala detaljnija analiza programskog koda. Za svakog tko se ozbiljnije bavi sa programiranjem mora biti jasan način na koji funkcioniraju prevoditelji s optimizacijskim mogućnostima, budući da obavljaju izuzetno puno posla „u sjeni“.

LITERATURA

[1.] Neurokirurgija, Medic, Zagreb, dostupno na: https://www.medic.hr/medicina/neurokirurgija/15 [25.7.2021.]

[2.] V. K. Myalapalli, J. K. Myalapalli and P. R. Savarapu, "High performance C programming," 2015 International Conference on Pervasive Computing (ICPC), 2015, pp. 1-6, doi: 10.1109/PERVASIVE.2015.7087005.

[3.] Primorac, Robert, Strategije i tehnike optimizacije programskog koda, 2011., diplomski rad,, Odjel za elektrotehniku i računarstvo, Dubrovnik, Hrvatska.

[4.] E. A. Efremov, A. A. Chufistov, R. R. Mukaev and V. Skvortsov, "Methods for Optimizing Software Performance," 2021 IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering (ElConRus), 2021, pp. 2072-2075, doi: 10.1109/ElConRus51938.2021.9396146

[5.] Abdulla, M.F., 2012. Manual and fast C code optimization. arXiv preprint arXiv:1203.0681.

[6.] Cvetić, Branko, Tehnike i strategije unapređenja softverskog koda, 2011., Projekt, ITS – Visoka škola strukovnih studija za IT, Beograd, Srbija.

[7.] GNU Compiler Collection (GCC): Optimization levels [online], GNU Project, New York, SAD, dostupno na: https://gcc.gnu.org/onlinedocs/gnat_ugn/Optimization-Levels.html [8.8.2021.]

[8.] Hrvatski Wiki o Programiranju: Optimizacije kompajlera [online], xFER, Zagreb, Hrvatska, dostupno na: https://wiki.xfer.hr/art_strlen/ [8.8.2021.]

[9.] R. Upadhyay, J. Nagarbandhara, S. Bhatt, A practical approach to optimize code implementation, eInfochips

SAŽETAK

U ovom završnom radu obrađena je tematika optimizacije programskog koda. Predstavljene su osnovne metode optimizacije u programskom jeziku C, na način da je dano kratko objašnjenje pojedine metode te su uspoređeni rezultati prije i nakon optimizacije. Naglasak je dan na korištenju novih mogućnosti (proširenja) razvojnih okruženja, koje uvelike pomažu u ostvarivanju ciljeva optimizacije programskog koda. U završnom je radu ukratko opisan način funkcioniranja prevoditelja s mogućnošću optimizacije, čiji se dodatni razvitak očekuje u skorijoj budućnosti.

Ključne riječi: metode optimizacije, optimizacija, optimizacijski prevoditelji, proces optimizacije.

OPTIMIZATION METHODS IN PROGRAMMING LANGUAGE C

ABSTRACT

This final paper deals with the topic of program code optimization. Basic optimization methods in programming language C are presented, in a way that a short explanation of each method is given and the results before and after the optimation are compared. Emphasis is placed on the use of new possibilities (extensions) of development environments, which greatly help in achieving the goals of program code optimization. The final paper briefly describes the way optimization compilers work, whose further development is expected in the near future.

Keywords: optimization methods, optimization, optimizing compilers, optimization process.

ŽIVOTOPIS

Damir Bašić rođen je 17. ožujka 2000. godine u Slavonskom Brodu. Pohađao je osnovnu školu „Blaž Tadijanović“ u Podvinju. Završio je Tehničku školu u Slavonskom Brodu, smjer elektrotehničar. Nakon završetka srednjoškolskog obrazovanja, pristupa ispitima državne mature te upisuje prvi odabir studija na Fakultetu elektrotehnike, računarstva i informacijskih tehnologija u Osijeku, smjer elektrotehnika: komunikacije i informatika.

PRILOZI

P.3.1. – Programski kodovi svih metoda optimizacije nalaze se u digitalnom formatu na CD-u.