# Profajliranje Haskell programa

Seminarski rad u okviru kursa
Metodologija stručnog i naučnog rada
Matematički fakultet

Jovana Bošković
jboskovic97@gmail.com

Ana Jakovljević
ana.jakovljev98@gmail.com

Nikola Perić
nikola.peric303@gmail.com

Mateja Trtica
matejarkvc@gmail.com

21. april 2020.

**Sažetak**

Tema ovog rada je profajliranje Haskell programa. Profajliranje je deo svakog razvoja softvera i kao takvo je bitno za razumevanje. Ono što ga čini dodatno potrebnim kod programiranja u funkcionalnim jezicima je visok nivo apstrakcije zbog koga osobine programa ne mogu lako da se naslute. U okviru rada osvrćemo se na probleme profajliranja Haskell programa imajući u vidu osobine samog jezka. Predstavljene su i osnove profajliranja programa korišćenjem alata koje nudi GHC.

**Ključne reči:** profajliranje, haskell, centar troškova, ghc, zastavice

## Sadržaj

**1 Uvod** 2

**2 Profajliranje** 2
2.1 Metode profajliranja 3

**3 Specifičnosti programskog jezika Haskell** 4
3.1 Ponovno korišćenje funkcija 4
3.2 Lenjo izračunavanje 4
3.3 Transformacije i optimizacije 5

**4 GHC profajliranje** 5
4.1 Centar troškova 5
4.1.1 Unošenje centara troškova 5
4.1.2 Pravila za dodelu troškova 6
4.2 Prikupljanje informacija o izvršavanju programa 6
4.3 Vremensko i alokacijsko profajliranje 7
4.4 Prostorno profajliranje 9

**5 Zaključak** 11

**Literatura** 11

**A Dodatak** 12

# 1 Uvod

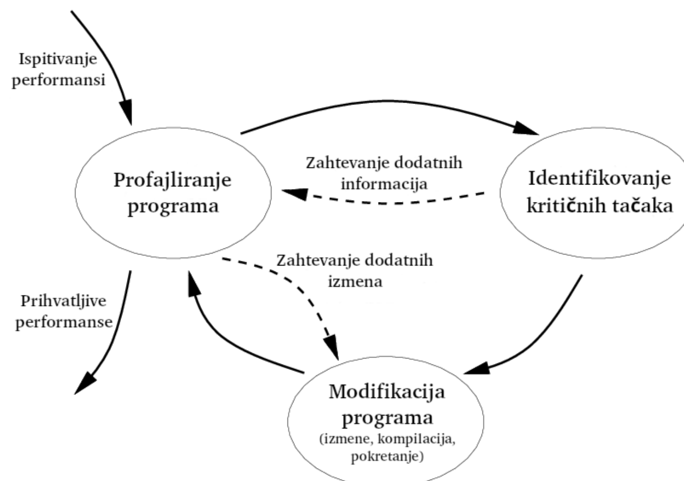
Za aplikaciju je važno da se u pogledu vremena i prostora izvršava na najbolji mogući način. Performanse programa su nešto o čemu programer treba da vodi računa. Međutim, ako bi program bio pisan na najbolji mogući način, proces razvoja softvera bi bio težak i spor, a takvi programi bi bili nerazumljivi. Glavnu ulogu u poboljšanju performansi programa igra profajliranje, na osnovu čijih rezultata rada programer poboljšava bitne delove koda. To olakšava proces pisanja i održavanja koda, pa je i produktivnost veća. Delove koda koje ne optimizuje programer optimizovaće kompajler ukoliko mu je to naznačeno. Pri dizajniranju kompajlera profajliranje ima bitnu ulogu pri proveri optimizacionih tehnika [6, 8].

U nastavku se nalaze opšte informacije o profajlerima (2). Zatim su navedeni specifičnost i problemi profajliranja programa pisanih na Haskell-u (3). Na kraju su kroz korišćenje GHC kompajlera predstavljeni način na koji se ti problemi rešavaju, postupak za generisanje profila i izgledi profila koji se dobijaju (4).

# 2 Profajliranje

Profajliranje je metoda pomoću koje se programer upoznaje sa dinamikom ponašanja programa. Procesom profajliranja programer dobija prikupljene podatke o izvršavanju programa koje dalje koristi u različite svrhe. Kao metoda dinamičke analize, profajliranje se vrši tokom izvršavanja programa. Kako bi analiza prikupljenih podataka tokom izvršavanja bila relevantna potrebno je da ciljni program bude izvršen dovoljan broj puta. Ovakva analiza podataka se koristi prilikom optimizacije programa u cilju povećanja brzine izvršavanja, smanjenja alokacije memorije i bolje iskorišćenosti drugih sistemskih resursa [6].

Alat koji vrši proces profajliranja naziva se profajler. Glavna funkcija profajlera je da programeru omogući identifikaciju kritičnih tačaka izvršavanja i opažanje uticaja određene izmene. Na slici 1 je prikazan tok analize programa korišćenjem profajlera [8].




Slika 1: Tok profajliranja

**Ko i zašto koristi profajliranje?** Podatke dobijene profajliranjem mogu koristiti projektanti hardvera kako bi proverili kako se programi izvršavaju na različitim arhitekturama. Podaci se mogu koristiti za proveru optimizacionih tehnika pri dizajniranju kompajlera. Programeri pomoću analize proveravaju programe tj. utvrđuju da li se ponašaju u skladu sa očekivanjima i koji deo koda zahteva optimizaciju da bi se postigle zadovoljavajuće performanse [6].

## 2.1 Metode profajliranja

Postoji više metoda profajliranja koje generišu različite profile. Tehnike koje se koriste i profili koji se dobijaju spadaju u nekoliko opštih kategorija [8].

**Profajliranje vremena**

1. *Uzimanje uzoraka* - Uzorkovanje se vrši prekidanjem izvršavanja programa u određenim vremenskim intervalima pri čemu se beleži koji deo programa se izvršava. Ako izvršavanje traje dovoljno da postoji dovoljan broj uzoraka onda su prikupljeni podaci od značaja. Rezultati poziva se međusobno razlikuju i nisu precizni, ali su realistični. Prednosti su malo zauzimanje memorije i što se vreme izvršavanja glavne aplikacije ne menja značajno.
2. *Brojač frekvencija* - Ubacuju se brojači u svaki bazni blok programa kako bi se odredilo koliko je puta svaki izvršen. Prednost je proveravanje svih funkcija, dok je broj poziva funkcija definisan. Ističe unutrašnje petlje, otkriva neizvršene delove koda i dinamiku ponašanja korišćenih algoritama. Mane su usporavanje izvršavanja glavne aplikacije i velika potrošnja memorije.
3. *Prolazno vreme procedure* - Ubacuju se izrazi koji čitaju sistemski sat na ulazu i izlazu programske jedinice. Dobija se informacija o vremenu provedenom u svakoj proceduri. Nedostatak je cena pristupa sistemskom satu koja može biti skupa, a tačnost profila zavisi od tačnosti sistemskog sata.

**Profajliranje prostora** Vreme izvršavanja nije jedini problem. Iako korišćenje velike količine memorije može da utiče na vreme izvršavanja, potrebno je znati izvor i uzrok memorijskog troška.
Sistem upravljanja memorijom može biti eksplicitan ili automatski. U zavisnosti od toga se koriste drugačiji profili.

1. *Profili alokacije* - Pruža osnovni uvid u korišćenje dinamičke memorije. Dobijene informacije o alokaciji su povezane sa izvornom lokacijom odgovornom za njih. Može otkriti kritične tačke, ali može napraviti dosta propusta. Naizgled nešto ne zauzima značajan deo prostora i neće biti istaknuto u profilu alokacije, ali je dugotrajno zastupljeno i na taj način crpi veliku količinu memorije.
2. *Profili curenja* - Služe za identifikaciju curenja memorije. Identifikuju se hip objekti koji nikad nisu dealocirani i izveštavaju o sekvenci poziva koja je odgovorna za alokaciju takvih objekata. Problemi povezani sa curenjem memorije u sistemima sa eksplicitnim upravljanjem memorijom su rezultirali nizom implementacija koje sadrže sakupljače otpadaka kako bi se uklonila potreba za eksplicitnim upravljanjem.

3. *Hip profili* - Profil koji koristi u sistemima sa implicitnim upravljanjem memorijom. Pruža informacije o živim podacima koji zauzimaju hip i pripisuje ih izvornim mestima odgovornim za njihovu dodelu. Pošto se sadržaj hipa vremenom menja, profil mora opisivati ponašanje hip objekata tokom vremena. Zato se vrši izveštavanje predstavom određenog broja „snimaka“ objekata koji zauzimaju hip tokom izvođenja.

# 3 Specifičnosti programskog jezika Haskell

Haskell je čist funkcionalni programski jezik visokog nivoa. Osnovni koncepti poput lenjog izračunavanja, polimorfizma i funkcija višeg reda su njegova prednost. Oni omogućavaju da programi koji se pišu budu koncizni i lako razumljivi. Sa druge strane, ovi koncepti proces dobijanja informacija o izvršavanju programa čine težim.

Apstraktnost struktura podataka koju pruža Haskell prikriva složenost implementacije na najnižem nivou, a transformacije kroz koje kod prolazi do izvršne verzije su značajne. Apstraktnost omogućava programeru da se koncentriše na suštinu problema bez potrebe da razmišlja o implementaciji na niskom nivou, ali onemogućava da na osnovu koda proceni performanse napisanog programa.

Informacije koje profajler pruža moraju da odgovaraju realnom izvršavanju. U skladu sa osobinama Haskell-a, to podrazumeva da *poredak izvršavanja* ostane isti, bez narušavanja *lenje semantike* forsiranjem delova programa da se izvrše. Dalje, program koji se profajlira treba da bude *optimizovan* od strane kompajlera, jer je to upravo onaj program koji se pokreće i koristi. Dodatna *cena profajliranja* bi trebalo da bude zanemarljiva. Narušavajući bilo koje od ovih pravila, dobijene informacije postaju pogrešne i ne mogu biti od koristi [4, 8].

## 3.1 Ponovno korišćenje funkcija

Pre svega, potrebno je bliže objasniti dva koncepta: funkcije višeg reda i polimorfizam. Funkcija višeg reda je funkcija koja za argument ili povratnu vrednost ima funkciju. Polimorfizam podrazumeva da tip funkcije sadrži promenljive tipova koje se instanciraju konkretnim tipovima u zavisnosti od upotrebe funkcije [2].

Upravo na ovim konceptima se zasniva ponovno korišćenje funkcije koje predstavlja postojanje opšte funkcije višeg reda koja može biti specijalizovana nekom drugom funkcijom. Ova velika ponovna upotreba malog broja funkcija čini pronalazak izvora troška težim. Podaci o potrošnji vremena i prostora na neku funkciju višeg reda moraju biti povezani sa mestom poziva, jer se primena takve funkcije može razlikovati i univerzalna informacija o svim pozivima ne bi bila od značaja [8].

## 3.2 Lenjo izračunavanje

Izraz se izračunava onda kada postane potreban, kada je njegov rezultat zahtevan od nekog drugog izraza. Neki delovi koda ne moraju nikada biti izvršeni i nema potrebe voditi brigu o takvim potencijalno neefikasnim delovima koda.

Program se ne izvršava u navedenom redosledu jer se javlja isprepletanost izvršavanja. Zbog toga se ispoljava problem povezivanja dinamički prikupljenih informacija sa određenim mestima u izvornom kodu.

Potrebno je razgraničiti čemu se pridružuje trošak izračunavanja: mestu na kome se definiše izraz ili mestu koje zahteva izračunavanje izraza. Pridruživanje troška mora biti razrešeno i u situaciji kada više izraza zatraži rezultat istog izraza, pri čemu se rezultat izračunava samo prvi put, a svaki sledeći zahtev nailazi na već izračunat rezultat [8].

## 3.3 Transformacije i optimizacije

Razvoj funkcionalnih programa uključuje transformacije i optimizacije koje nastaju kao posledica visokog nivoa apstrakcije.

U procesu kompajliranja Haskell kod se prevodi na međujezik Core koji predstavlja poslednju funkcionalnu verziju programa pre prevođenja na imperativni kod niskog nivoa (više o Core na [7]). Većina optimizacionih transformacija se vrši upravo nad Core reprezentacijom.

Kod koji se dobije se značajno razlikuje od izvornog: uvode se skrivene funkcije prevođenjem *sintaksičkih oblika* koji prikrivaju složenost (eng. syntactic sugars), uvode se pomoćne funkcije pri transformaciji izraza i kombinuju se rezultati svih transformacija i optimizacija. Bitno je da tokom procesa kroz koje kod prolazi do izvršne verzije troškovi ostanu povezani sa odgovarajućim centrima [4, 8].

# 4 GHC profajliranje

GHC (*The Glasgow Haskell Compiler*) je kompajler i interaktivno okruženje za programski jezik Haskell. GHC sadrži sistem za vremensko i memorijsko profajliranje. Proces profajliranja pomoću GHC se sastoji iz tri koraka [10]:

1. Rekompajlirati program uz opciju za profajliranje
2. Pokrenuti program da bi se generisao profil. Postoje različiti profili koji mogu biti generisani biranjem opcija. Neki profili zahtevaju dalje procesuiranje korišćenjem dodatnih alata
3. Ispitati generisan profil. Koristeći dobijene informacije razmotriti proces optimizacije i po potrebi ponoviti proces

## 4.1 Centar troškova

Sistem profajliranja primenjen u GHC rešava problem povezivanja. To je ostvareno korišćenjem *centara troškova* (eng. cost centers). Centar troškova je struktura deklarisana označenim delom koda čije su informacije o izvršavanju od interesa [8]. Trošak predstavlja vremensku ili memorijsku zahtevnost izraza.

Princip funkcionisanja: vrši se povezivanje izraza od interesa iz izvornog koda sa centrima troškova. Tokom izvršavanja se identifikuje centar troškova povezan sa izrazom koji se trenutno izračunava i podaci o izvršavanju se skupljaju na identifikovani centar. GHC pamti stek centara troškova za svaki izraz u toku izvršavanja i generiše *stablo poziva* (eng. call-tree) sa pripisanim troškovima [10].

### 4.1.1 Unošenje centara troškova

Kada se prvi put vrši profajliranje programa koristi se opcija za automatsko generisanje oznaka da bi se dobila prva informacija o radu programa. Na osnovu nje je moguće odrediti kritične tačke izvršavanja na

koje je potrebno fokusirati se u daljem procesu. Tada se prelazi na ručno dodavanje centara troškova pomoću kojih se preciziraju mesta o kojima je potrebno znati više [3].

Dodavanjem opcije *-fprof-auto* kompajleru se sugeriše da automatski ubaci oznake za centre troškova.

Ručno označavanje centra troškova se vrši pomoću oznake SCC (eng. set cost center) što znači "postaviti centar troška". Sintaksa je sledeća:

```
{-# SCC "name" #-} expression
```

Ovakva naredba vraća vrednost navedenog izraza, ali ima bočni efekat koji omogućava profajliranje. Opseg važenja oznake može biti kontrolisan zagradama. Moguće je da više izraza ima navedenu istu oznaku i tada se troškovi svih izraza prikupljaju na isti centar troška.

Pridruživanje centra troškova funkcijama vrši se prosleđivanjem imena funkcije definisane u istom modulu, pri čemu ime centra troška ostaje ime navedene funkcije:

```
{-# SCC func #-}
```

Ukoliko je potreban drugi naziv, različit od naziva funkcije, koristi se izraz:

```
{-# SCC func "name" #-}
```

### 4.1.2 Pravila za dodelu troškova

Da bi prikupljeni podaci bili korisni, mora biti definisana semantika troškova tj. razgraničeno koje cene se povezuju na koji centar troškova. Troškovi izračunavanja koje zahtevaju sve instance nekog izraza se povezuju sa centrom koji određuje taj izraz. Cene svih izračunavanja iniciranih izrazom su nasleđene i povezane sa zatvarajućim centrom troškova. Ovo omogućava sumiranje troškova ugnježdenih funkcija i sagledavanje odlika logičkih delova programa.

Tokom izvršavanja i profajliranja programa GHC održava stek sa centrima troškova. Kada program naiđe na izračunavanje izraza pod SCC oznakom, navedeni centar se postavlja na trenutni stek.

Kreiranjem *lenjog izraza* (eng. thunk), trenutni stek centara troškova se uskladišti u taj izraz, a vraća se kada se izraz izračuna. Na ovaj način stek centara troškova je nezavisan od redosleda izračunavanja. Pozivom funkcije, GHC uzima stek uskladišten u pozivajućoj funkciji i koristi ga kao trenutni stek.

Lenji izrazi najvišeg nivoa koji se kreiraju kada se program kompajlira nazivaju se CAF ("Constant Applicative Form"). GHC svakom CAF-u u modulu dodeljuje stek koji se sastoji od jedinstvenog centra troškova M.CAF, gde M predstavlja naziv modula. Moguće je dati svakom CAF-u različit stek, koristeći opciju *-fprof-cafs*. Ovo je posebno od koristi kada kompajliramo opcijom *-ffull-laziness*, jer će konstante u telima funkcija postati CAF-ovi. Najverovatnije će biti potrebno konsultovati se sa jezgrom da bi se utvrdilo čemu odgovaraju ti CAF-ovi [8, 10].

## 4.2 Prikupljanje informacija o izvršavanju programa

Program se priprema za profajliranje navođenjem zastavice *-prof* pri prevođenju, čime je omogućeno osnovno vremensko i memorijsko profajliranje. Dodatne zastavice koje se mogu proslediti prevodiocu nalaze se u tabeli 1.

Tabela 1: Zastavice koje se mogu proslediti prilikom prevođenja programa

| Zastavica | Opis |
|---|---|
| -prof | svi moduli moraju biti kompajlirani i povezani ovom opcijom kako bi svaka dodata oznaka bila obrađena |
| -fprof-auto | sva vezivanja koja nisu označena sa *inline*, izvezena ili ne, najvišeg nivoa ili ugnježdena biće automatski označena za profajliranje (funkcijama označenim sa *inline* SCC oznake se pridružuju ručno) |
| -fprof-auto-top | GHC će automatski dodati SCC oznaku svim vezivanjima najvišeg nivoa koja nisu označena sa *inline* |
| -fprof-auto-exported | GHC će automatski dodati SCC oznaku svim izvezenim funkcijama koje nisu označene *inline* |
| -fprof-auto-calls | dodaje automatsku oznaku svim mestima poziva (ovo je korisno kada se profajliranje koristi za generisanje izveštaja o aktivnim stek okvirima) |
| -fprof-cafs | svi CAF-ovi dobijaju svoj centar troška (troškovi svih CAF-ova u modulu se obično povezuju na jedan veliki CAF centar troškova) |

Da bi se dobili podaci o izvršavanju koriste se specijalne zastavice *+RTS* i *-RTS*, kojima se razdvajaju argumenti rezervisani za sistem izvršavanja od argumenata poziva programa. Pomoću ovih zastavica mogu se zatražiti podaci o aktivnosti sakupljača otpadaka, o iskorišćenosti memorije, o promeni količine dostupne memorije na hipu ili steku i može se kontrolisati broj niti operativnog sistema. Aplikacija nema uvid u dodatne zastavice jer se one momentalno obrade od strane Haskell sistema za izvršavanje [3].

Jednostavan način za ispis informacija o pokrenutom programu je zadavanje zastavice *-sstderr* (nakon zastavice *+RTS*) kojom se ispisuju osnovne informacije o programu poput zauzeća memorije i aktivnost sakupljača otpadaka na standardni izlaz za greške.

## 4.3 Vremensko i alokacijsko profajliranje

Prilikom klasičnog vremenskog i alokacijskog profajliranja koristi se zastavica *-prof*. Vrednosti bez argumenata je potrebno kompajlirati jedanput, a rezultat deliti za kasnije potrebe. Takve vrednosti zapravo nisu deo *pozivajućeg grafa* (eng. call graph) programa jer se ne izračunavaju pri svakom pozivu, ali potrebno je znati koliko je koštao taj jedan poziv izračunavanja. Da bi se dobili tačni brojevi tih vrednosti, poznatih kao CAF-ovi, koristi se zastavica *-caf-all* [3].

Dodavanje centara troškova može da promeni način izvršavanja programa jer nameće dodatne troškove [3]. Da se ne bi desilo prekoračenje stek memorije, moguće je pri pokretanju programa proslediti zastavicu za proširivanje te memorije. Zastavica se navodi uz količinu megabajta kojom se stek proširuje (-K100M proširuje memoriju steka za 100MB).

Dobijanje profila vremena i alokacije zahteva da se prilikom pozivanja

kompajliranog programa proslede neke od RTS opcija. Osnovna od njih je *-p* koja daje izveštaj o profilu vremena zapisan u fajlu *imePrograma.prof* (slika 2). Naziv fajla može biti izmenjem opcijom *-po novoIme*.

```
$ ghc -prof -fprof-auto -rtsopts Main.hs
$ ./Main +RTS -p
```

```
    Tue Mar 31 23:52 2020 Time and Allocation Profiling Report  (Final)

        Main +RTS -p -RTS

    total time  =        0.06 secs   (56 ticks @ 1000 us, 1 processor)
    total alloc =  56,947,248 bytes  (excludes profiling overheads)

COST CENTRE       MODULE    SRC                     %time %alloc

process.word_occs Main      Main.hs:(30,9)-(31,46)   67.9   33.4
process.ws        Main      Main.hs:(25,9)-(27,20)   26.8   57.5
process.ws.\      Main      Main.hs:26:27-60          3.6    0.0
main              Main      Main.hs:(10,1)-(19,27)    1.8    7.8


                                                                                     individual      inherited
COST CENTRE               MODULE                  SRC                     no.     entries  %time %alloc   %time %alloc

MAIN                      MAIN                    <built-in>              119          0    0.0    0.0   100.0  100.0
 CAF                      Main                    <entire-module>         237          0    0.0    0.0     0.0    0.0
  main                    Main                    Main.hs:(10,1)-(19,27)  238          1    0.0    0.0     0.0    0.0
 CAF                      GHC.Conc.Signal         <entire-module>         228          0    0.0    0.0     0.0    0.0
 CAF                      GHC.IO.Encoding         <entire-module>         217          0    0.0    0.0     0.0    0.0
 CAF                      GHC.IO.Encoding.Iconv   <entire-module>         215          0    0.0    0.0     0.0    0.0
 CAF                      GHC.IO.Exception        <entire-module>         209          0    0.0    0.0     0.0    0.0
 CAF                      GHC.IO.Handle.FD        <entire-module>         207          0    0.0    0.1     0.0    0.1
 CAF                      GHC.IO.Handle.Internals <entire-module>         206          0    0.0    0.0     0.0    0.0
 CAF                      System.Posix.Internals  <entire-module>         179          0    0.0    0.0     0.0    0.0
 CAF                      GHC.IO.FD               <entire-module>         146          0    0.0    0.0     0.0    0.0
 CAF                      GHC.IO.Handle.Text      <entire-module>         145          0    0.0    0.0     0.0    0.0
 main                     Main                    Main.hs:(10,1)-(19,27)  239          0    1.8    7.8   100.0   99.9
  process                 Main                    Main.hs:(23,1)-(37,42)  240          1    0.0    0.9    98.2   92.1
   process.sorted_by_occs Main                    Main.hs:34:9-63         241          1    0.0    0.3     0.0    0.3
   process.word_occs      Main                    Main.hs:(30,9)-(31,46)  242          1   67.9   33.4    67.9   33.4
    process.word_occs.\   Main                    Main.hs:30:32-54        245        301    0.0    0.0     0.0    0.0
   process.ws             Main                    Main.hs:(25,9)-(27,20)  243          1   26.8   57.5    30.4   57.5
    process.ws.\          Main                    Main.hs:26:27-60        244     103834    3.6    0.0     3.6    0.0
```

Slika 2: Sadržaj fajla Main.prof

Prvi deo fajla predstavlja informacije o nazivu programa i korišćenim zastavicama. Izveštaj sadrži ukupno vreme i ukupnu alociranu memoriju tokom izvršavanja programa (ukupna alocirana memorija nije isto što i količina žive memorije potrebne programu u svakom trenutku izvršavanja). U datom primeru ukupno vreme tokom izvršavanja programa iznosi 0.06 sekundi i program se izvršava na jednom procesoru, a alocirana memorija iznosi 56,947,248 bajtova, ne uključujući neophodne troškove profajliranja.

Drugi deo sadrži podelu po centrima troškova, sortiranu opadajuće po cenama. Za svaki centar troškova dati su podaci kom modulu pripada, u kom fajlu se nalazi i procenat iskorišćenog vremena i alocirane memorije koji koristi. U navedenom primeru centar process.word_occs koji se nalazi u modulu Main, u fajlu Main.hs (kod se može videti u dodatku A), troši 67.9% vremena i koristi trećinu alocirane memorije.

Treći deo sadrži podelu profila prema skupu centara troškova. Ovo je grubo stablo poziva programa. Kolona *entries* predstavlja koliko je puta konkretan čvor u stablu posećen. Kolona *individual* pruža informaciju o količini vremenskih i memorijskih resursa koje je centar potrošio nezavisno od ostalih, a kolona *inherited* o količini koje je potrošio zajedno sa naslednicima (izraženo u procentima). U primeru se može videti da najskuplji poziv, process.word_occs, dolazi iz funckije process koja dolazi iz main. Njegov broj poseta u stablu je 1, vreme i alocirana memorija u procentima ista kao u drugom delu, kao i ukupan procenat i svih potpoziva.

Moguće je korišćenje *-P* opcije kojom se dobijaju sledeći dodatni podaci: *ticks* – sirov broj vremenskih otkucaja koji su pripisani ovom centru troškova i *bytes* – broj bajtova alociranih na hipu u ovom centru troškova. Dodatne RTS opcije za vremensko i alokacijsko profajliranje se mogu naći u tabeli 2.

Tabela 2: Zastavice koje se mogu proslediti prilikom pokretanja programa između +RTS i -RTS zastavica

| Zastavica | Opis |
|---|---|
| -p | generiše profil vremena zapisan u *imePrograma.prof* fajlu (naziv može biti promenjen opcijom *-po novoIme*) |
| -P | generiše detaljniji profil vremena i alociranih podataka |
| -pa | generiše najopštiji izveštaj koji sadrži informacije o vremenu i alokaciji dopunjene svim centrima troškova |
| -pj | generiše izveštaj u JSON formatu ispisanom u fajlu *imePrograma.prof* |
| -V | postavlja interval koji RTS sat otkucava i ujedno postavlja interval uzorkovanja za profil vremena i alokacije |
| -hT | deli grafikon po vrsti hip zatvorenja |
| -hc | deli grafikon prema steku centara troškova |
| -hm | deli živi hip po modulu koji sadrži kod koji proizvodi trošak |
| -hd (desc) | restrikcija profila na zatvorenja sa specijalnim opisnim stringovima |
| -hy (type) | restrikcija profila na zatvorenja sa specijalnim tipovima |
| -hb (bio) | restrikcija profila na zatvorenja sa specijalnom biografijom, gde ⟨bio⟩ može biti jedna od lag, drag, void, ili use oznaka |
| -l | emituje uzorke profila na GHC event log (ovaj format je izražajniji od .hp formata i može biti povezan sa drugim događajima nad izvršnim programom) |

Što se tiče rekurzivnih funkcija i uzajamne rekurzije u grupama funkcija, GHC čuva informacije o tome koje grupe funkcija zovu jedna drugu rekurzivno, ali te informacije nisu prikazane u osnovnom profilu vremena i alokacije. Umesto toga, *graf poziva* (eng. call-graph) se ispravlja u drvo po sledećem pravilu: poziv funkcije koja se pojavljuje u nekom drugom delu na trenutnom steku neće postaviti jos jedan unos na stek. Umesto toga, cena za ovaj poziv se izračunava u pozivaocu [10].

## 4.4 Prostorno profajliranje

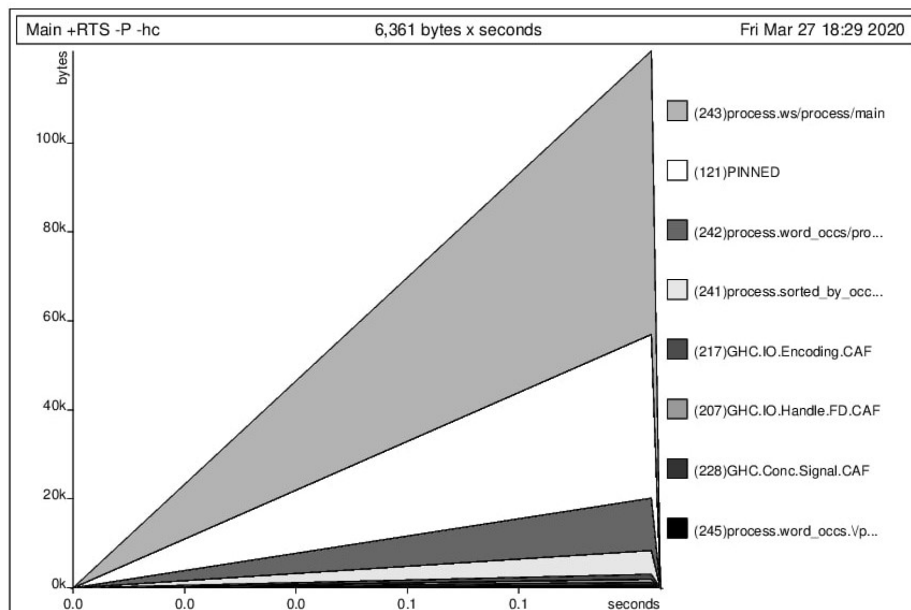
U programu se može javiti zadržavanje memorije koja se ne upotrebljava, označeno kao curenje memorije. Ovakva pojava za posledicu ima intenzivnu aktivnost sakupljača otpadaka, čiji zadatak je da takvu me-

moriju detektuje i oslobodi. Upotreba sakupljača otpadaka za posledicu ima trošenje dodatnih resursa tokom izvršavanja programa. GHC pruža mogućnost dobijanja informacija o zauzeću memorije na hipu tokom izvršavanja programa u vidu grafikona [3, 10].
Da bi se generisao hip profil programa potrebno je [10]:

1. Kompajlirati program za profajliranje pomoću odgovarajućih opcija obrađenih ranije
2. Pokrenuti ga sa nekom od opcija za profajliranje hipa navedenih u tabeli 2, čime se dobija fajl *prog.hp*
3. Pokrenuti alat hp2ps nad *prog.hp* za Postscript ispis fajla *prog.ps* [1]
4. Prikazati hip profil pomoću Postscript pregledača kao što je Ghostview [9]

Program se prevodi na isti način kao i za vremensko, korišćenjem zastavica *-prof-auto-all -caf-all*. Nakon toga, izvršna datoteka se pokreće uz zastavicu *-hc*, čime se dobija nova datoteka sa ekstenzijom .hp koja sadrži neobrađene informacije o zauzeću memorije tokom izvršavanja programa. Unutar datoteke se nalaze podaci o iskorišćenosti memorije u centrima troškova uzorkovanim u određenim trenucima tokom izvršavanja programa. Češće uzorkovanje može se postići zastavicom *-N* (N je broj sekundi između dva uzorkovanja). Da bi se dobio prikaz u vidu grafikona, ovu datoteku treba proslediti kao ulaz alatu hp2ps, a izlaz alata otvoriti uz pomoć Postscript pregledača. Grafikon dobijen na ovaj način za program Main.hs može se videti na slici 3.

```
$ ghc -prof -fprof-auto -caf-all Main.hs
$ ./Main +RTS -p -hc
$ hp2ps -b Main.hp
```




Slika 3: Zauzeće hipa tokom izvršavanja programa

# 5 Zaključak

Ovim radom je obuhvaćena osnova i motiv upotrebe profajlera. Uveden je pojam centra troškova, koji predstavlja osnovu za profajliranje. Iznet je specifičan slučaj profajliranja pomoću GHC alata dat kroz primer, kao i dodatne zastavice koje mogu biti korišćene. Naprednija primena profajliranja programa sa paralelnim i konkurentnim izvršavanjem može se naći na [5]. Dobijene informacije treba dalje primenjivati. Odluke koje treba doneti i akcije koje treba preduzeti nisu obrađene u ovom radu. U cilju dajle optimizacije programa treba obratiti pažnju na nekoliko dodatnih stvari poput korišćenja striktnih tipova i repne rekurzije. Mogu se primeniti i naprednije tehnike optimizacije kao što je stapanje. Više o tome na [3].

# Literatura


[1] CORPORATE Adobe Systems Inc. *PostScript Language Reference (3rd Ed.)*. Addison-Wesley Longman Publishing Co., Inc., USA, 1999.

[2] Richard Bird and Philip Wadler. *An Introduction to Functional Programming*. Prentice Hall International (UK) Ltd., GBR, 1988.

[3] John Goerzen Bryan O'Sullivan, Don Stewart. *Real World Haskell*. O'Reilly Media, 1005 Gravenstein Highway North, Sebastopol, CA 95472, 2008.

[4] K. Hammond, D.N. Turner, and P.M. Sansom. *Functional Programming, Glasgow 1994: Proceedings of the 1994 Glasgow Workshop on Functional Programming, Ayr, Scotland, 12–14 September 1994*. Workshops in Computing. Springer London, 2013.

[5] David J. King, Jon G. Hall, and Philip W. Trinder. A strategic profiler for glasgow parallel haskell. pages 88–102, 1998.

[6] Dušan Okanović. *Model adaptivnog sistema za praćenje i predikciju rada distribuiranih aplikacija*. PhD thesis, Univerzitet u Novom Sadu, Fakultet tehničkih nauka u Novom Sadu, September 2012.

[7] Simon Peyton Jones and Andre Santos. A transformation-based optimiser for haskell. *Science of Computer Programming*, 32(1):3–47, October 1997.

[8] Patrick M. Sansom. *Execution Profiling for Non-strict Functional Languages*. PhD thesis, University of Glasgow, April.

[9] Artifex Software. Ghostview, 2019. on-line at: https://gsview.com/.

[10] The Glasgow Haskell Compiler Team. Glasgow Haskell Compiler User's Guide, 2015. on-line at: https://downloads.haskell.org/ghc/latest/docs/html/users_guide/.

# A Dodatak

```haskell
import qualified Data.List as List
import qualified Data.Char as Char
import System.IO

main :: IO ()
main = do
    text <- readFile "tekst.txt"
    putStrLn $ process text

process ::  String -> String
process text =
    let ws = words $ map Char.toLower $
             map (\ c -> if Char.isLetter c then c else ' ') $
             text
        word_occs = map (\g -> (List.length g, g !! 0)) $
             List.group $ List.sort ws
        sorted_by_occs = List.sortBy (flip compare) $ word_occs
    in unlines $ map show $ sorted_by_occs
```

Main.hs