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

2. 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 neophodnijim kod programiranja u funkcionalnima jezima je visok nivo apstrakcije zbog koga osobine programa ne možemo lako da naslutimo. U okviru rada osvrćemo se na opšte načine 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

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 pisao programe na najbolji mogući način, proces razvoja softvera bi bio veoma 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, a produktivnost se uvećava i do nekoliko puta. Delovi koda koje ne optimizuje programer, optimizovaće optimizator kompajlera ukoliko mu je to naznačeno. U procesu razvoja optimizatora profajliranje takođe igra bitnu ulogu.

U nastavku se nalaze opšte informacije o profajlerima (2). Zatim su navedeni specifičnost i problemi profajliranja na lenjim funkcionalnim jezicima visokog nivoa kakav je Haskell (3). I na kraju način na koji se u GHC kompajleru ti problemi rešavaju (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. Profajliranje kao metoda dinamičke analize se vrši tokom izvršavanja programa, a kako bi analiza prikupljenih podataka tokom izvršavanja bila relevantna potrebno je da ciljni program bude izvršen dovoljan broj puta. Analiza podataka na ovaj način se koristi prilikom optimizacije programa u cilju povećanja brzine izvršavanja, smanjenja alokacije memorije i bolje iskorišćenosti drugih sistemskih resursa.

Alat koji vrši proces profajliranja naziva se profajler.
Glavna funkcija profajlera je da programeru omogući identifikaciju kritičnih tačaka izvršavanja i opažanje uticaja određene izmene. Jednom kada programer utvrdi koje su kritične tačke izvršvanja, potrebna je mogućnost utvrđivanja uzroka.

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 kako bi utvrdili da li se ponašaju u skladu sa očekivanim performansama i koji deo koda zahteva optimizaciju da bi se postigle zadovojavajuće performanse [4].

2.1 Metode profajliranja

Postoji više metoda profajliranja koje nam daju različite profile. Informacije koje ti profili pružaju se međusobno razlikuju i imaju različitu primenu [6].

Profajliranje vremena

1. *Uzimanje uzoraka* - Uzorkovanje se vrši prekidanjem izvršavanja programa u periodičnim 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. Računa se vreme između dva uzorka ili koliko je vremena potrebno da se izvede jedan. 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 i otkriva neočekivani kod i dinamiku ponašanja algoritama koji se koriste. Mane su usporavanje izvršavanja glavne aplikacije, 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 potreba za velikom količinom memorije može da utiče i na vreme izvršavanja, potrebno je znati izvor i uzrok takvog memorijskog troška.
Sistem upravljanja memorijom može biti eksplicitan ili automatski. Na osnovu toga može nam biti drugačiji profil izvršavanja.

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 taćke ali može napaviti dosta propusta. Naizgled nešto zauzima beznačajno prostora ali je dugotrajno zastupljeno i tu nastaje problem.
2. *Profili curenja* - Specijalni profili za identifikaciju curenja memorije. Identifikuju hip objekte koji nikad nisu dealocirani i izveštavaju o sekvenci poziva koja je odgovorna za alokaciju takvih objekata. Problemi povezani sa curenjem memorije u eksplicitnim sistemima upravljanja prostorom su rezultirali nizom implementacija koje sadrže šeme sakupljanja otpada kako bi se uklonila potreba za eksplicitnim upravljanjem.
3. *Hip profili* - Metoda koja se koristi u sistemima sa implicitnim upravljanjem skladištenjem. Pruža informacije o živim podacima koji zauzimaju hip i pripisuje ih izvornom mestu odgovornom za njegovu 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 Profajliranje u Haskellu

Haskell je programski jezik visokog nivoa. Osnovni koncepti poput lenjog izračunavanja, polimorfizma i funkcija višeg reda su njegova glavna 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.

Da bi bilo moguće iz posmatranja koda donositi zaključke, potrebno je poznavanje reprezentacije na niskom nivou. Haskell koristi funkcionalni jezik "Core" (više o jeziku Core ovde [5]) kao međukod koji predstavlja poslednju funkcionalnu verziju programa pre prevođenja na imperativni kod niskog nivoa.

Informacije koje profajler pruža moraju da odgovaraju realnom izvršavanju. *Poredak izvšravanja* mora ostati isti i ne sme se narušiti *lenja semantika* forsiranjem delova programa da se izvrše. Ovaj uslov onemogućava merenje vremena potrebnog da se izvrši deo koda između dve tačke. Dalje, program koji profajliramo treba da bude *optimizovan* jer je to upravo onaj program koji se pokreće i koristi, a dodatna *cena profajliranja* bi trebala da bude zanemarljiva. Narušavajući bilo koje od ovih pravila dobijene informacije postaju pogrešne i ne mogu biti od koristi [2, 6].

3.1 Ponovno korišćenje funkcija

Polimorfizam i funkcije višeg reda omogućavaju ponovno korišćenje funkcija. Ogleda se u mogućnosti postojanja 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 funkcije višeg reda moraju biti povezane sa mestom poziva jer se njihova primena može razlikovati, a univerzalna informacija o svim pozivima ne bi bila od značaja.

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 da voditi brigu o takvim potencijalno neefikasnim delovima koda.

Program se ne izvršava u očiglednom 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 pridružujemo trošak izračunavanja: mestu na kome se definiše izraz ili mestu koje zahteva izračunavanje izraza. Pridruživanje troška mora se razrešeti i u situaciji kada nekoliko izraza zatraži rezultat pri čemu se taj izraz izračunava samo prvi put, a svaki sledeći zahtev nailazi na izračunat izraz.

3.3 Transformacije i optimizacije

Implementacija funckionalnih programa uključuje transformacije i optimizacije koje nastaju kao posledica visokog nivoa apstrakcije. Kod koji se dobije se značajno razlikuje od izvornog: uvode se skrivene funkcije prevođenjem sintaksičkih oblika koji prikrivaju složenost, uvode se pomoćne funkcije kada se izrazi transformišu i kombinuju se rezultati svih transformacija i optimizacija.

4 GHC profajliranje

GHC u sebi sadrži sistem za vremensko i memorijsko profajliranje. Proces profajliranja se sastoji iz tri koraka:

1. Rekompajlirati program za profajliranje dodavanjem opcije *–prof*.
2. Nakon kompajliranja programa za profajliranje, potrebno je pokrenuti ga da bi se generisao profil. Postoji dosta različitih profila koji mogu biti generisani biranjem različitih opcija [7]. Neki zahtevaju dalje procesuiranje korišćenjem dodatnih alata nakon pokretanja programa.
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 *centra troškova* (eng. cost centers). Centar troška je oznaka na koju povezujemo cenu izvršavanja i predstavlja lokaciju u programu za koju želimo da sakupljamo podatke [1]. Trošak predstavlja vremensku ili memorijsku zahtevnost izraza.

Princip funkcionisanja: Vrši se povezivanje izraza od interesa iz originalnog izvora sa centrima troškova. Bitno je očuvanje povezanosti tokom transformacije i optimizacije kompajlera. Tokom izvršavanja, potrebno je identifikovati mesta troška povezanog sa izrazom koji se trenutno izračunava. Podaci o izvršavanju se skupljaju na identifikovani centar troška. GHC pamti stek centara troškova za svaki izraz u toku izvršavanja i od toga generiše *stablo poziva* (eng. call-tree) sa pripisanim troškovima [7].

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 cena troškova pomoću kojih se preciziraju mesta o kojima je potrebno znati više [1].

Dodavanjem opcije *-fprof-auto* kompajleru sugerišemo da automatski ubaci oznake za centre troškova.
Sintaksa ručnog označavanja centra troškova za izraz je sledeća:

```
{-# SCC "id" #-} <expr>
```

SCC = Set Cost Center. Ovakva naredba vraća vrednost izraza expr, 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 računaju na isti centar troška.

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

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

Ukoliko je potrebna promena imena centra troška, koristi se izraz:

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

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.

Svaki centar troškova je povezan troškovima izračunavanja koje zahtevaju sve instance izraza određeni određuje taj centar troškova. Cene svih izračunavanja iniciranih izrazom su nasleđene i povezane sa zatvarajućim centrom troškova. Ovo omogućava sumiranje troškova ugnježdenih funkcija i tako omogućava 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 cena 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 bismo utvrdili čemu odgovaraju ti CAF-ovi [7, 6].

4.2 Prikupljanje informacija o izvršavanju programa

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 otpada, o iskorišćenosti memorije, o promeni količine dostupne memorije na hipu ili steku, ili se može kontrolisati broj niti operativnog sistema. Aplikacija nema uvid u dodatne zastavice jer se one momentalno obrade od strane Haskell sistema za izvršavanje [1].

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

Da bi se program pripremio za profajliranje potrebno je prevesti ga uz zastavicu *-prof*, čime je omogućeno osnovno vremensko i memorijsko profajliranje. Dodatne zastavice koje se mogu prosledeiti prevodiocu se mogu videti u tabeli 1.

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 dobile tačni brojevi tih vrednosti, poznatih kao CAFs, koristi se zastavica *-caf-all* [1].

Dodavanje centara troškova može da promeni način izvršavanja programa jer nameće dodatne troškove [1]. Da se ne bi desilo prekoračenje stek memorije, moguće je pri pokretanju programa proslediti zastavicu

Zastavica	Opis
-prof	- svi moduli moraju biti kompajlirani i povezani opcijom –prof - svaka SCC oznaka ručno dodata biće obrađena
-fprof-auto	- sve veze koje nisu obeležene inline-om dobiće automatske SCC oznake - funkcijama obeleženim inline-om se mora ručno dodati centar troškova
-fprof-auto-top	- sva top-level povezivanja koja nisu obeležena inline-om dobiće automatske SCC oznake - funkcijama obeleženim inline-om se mora ručno dodati centar troškova
-fprof-auto-exported	- sve izvezene funckije koje nisu obeležene inline-om dobiće automatske SCC oznake - funkcijama obeleženim inline-om se mora ručno dodati centar troškova
-fprof-auto-calls	- za sva mesta poziva dodaje automatkse SCC oznake - više detalja pogledati zastavicu –xc RTS
-fprof-cafs	- svi CAF-ovi dobijaju svoj troškovni centar.

Tabela 1: Zastavice koje se mogu proslediti prilikom prevođenja programa

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 ime_programa.prof (Slika 2). Naziv fajla može biti izmenjem opcijom *-po <novo_ime>*

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

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 0.09 secs 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 65.2% vremena i koristi trećinu alocirane memorije.

Treći deo sadrži podelu profila prema skupu centara troškova. Ovo je grubo stablo poziva programa. Kolona *entery* predstavlja koliko je puta konkretni čvor u stablu unesen. Kolone *individual* i *inherited* pružaju informacije o količini vremenskih i memorijskih resursa koje je centar potrošio nezvisno od ostalih i količini koje je potrošio računajući naslednike (izraženo u procentima). U primeru možemo jasno videti da najskuplji

```
    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 1: Sadržaj fajla Main.prof

poziv, process.word_occs, dolazi iz funckije process koja dolazi iz main. Njegov broj unosa u stablo je 1, vreme i alocirana memorija u procentima ista kao u drugom delu, kao i ukupan procenat i svih potpoziva.

Moguće je korišćenje *-P RTS* 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.

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 [7].

4.4 Prostorno profajliranje

U programu se može javiti nepotrebno zadržavanje memorije označeno kao curenje memorije. Deo memorije je zauzet ali se ne upotrebljava. Ovakva pojava za posledicu ima intenzivnu aktivnost sakupljača otpada, čiji zadatak je da takvu memoriju detektuje i oslobodi. Upotreba sakupljača otpada dalje 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 [1, 7].

Da bismo generisali hip profil programa potrebno je:

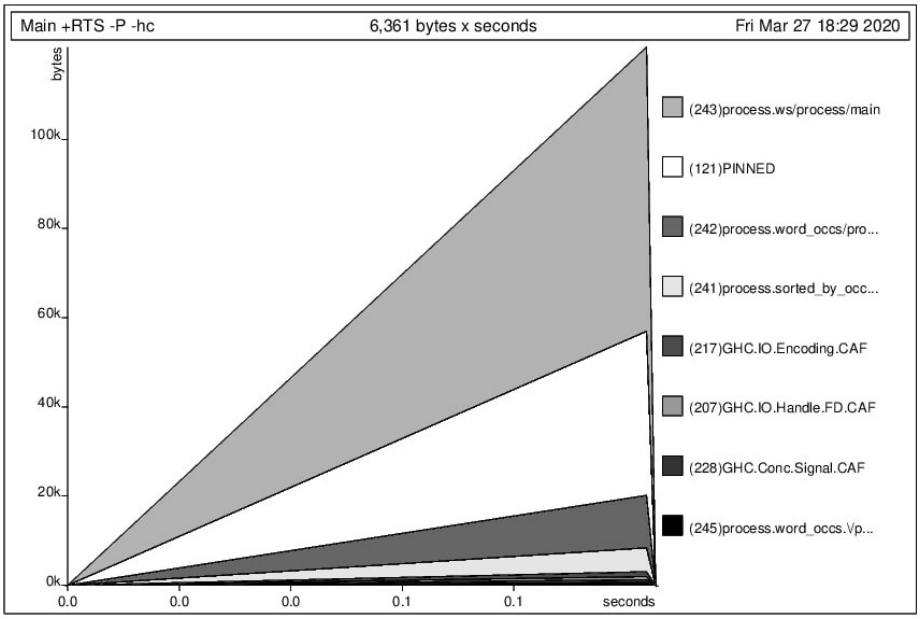
1. Kompajlirati program za profajliranje pomoću odgovarajućih opcija obrađenih ranije
2. Pokrenuti ga sa nekom od hip profajler opcija navedenih u tabeli 2 čime se dobija fajl prog.hp
3. Pokrenuti hp2ps nad prog.hp za Postscript ispis fajla prog.ps
4. Prikazati hip profil pomoću Postscript viewer-a kao što je Ghostview

Program se prevodi na isti način kao i za vremensko, koristeći zastavice *-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 koji generiše grafikon.

Zastavica	Opis
-p	- daje izveštaj o profilu vremena zapisan u fajlu <stem>.prof, stem podrazumevano predstavlja ime programa, ali može biti izmenjeno opcijom –po <stem>zastavice
-P	- daje opštiji izveštaj koji sadrži tačno vreme i alocirane podatke
-pa	- daje najopširniji izveštaj koji sadži sve centre troškova u dodatku sa vremenskim i alociranim podacima.
-pj	- daje izveštaj vreme/alokacija u JSON formatu ispisanom u fajlu <program>.prof.
-V (sec)	- postavlja interval koji RTS sat otkucava, koji ujedno predstavlja uzorački interval vremenskog i alokacijskog profila - podrazumevana vrednost je 0.02 secs
-hT	- deli grafikon hipa prema vrsti pregrada
-hc -h	- deli grafikon prema steku centara troškova koje podaci proizvode
-hm	- deli trenutno stanje hipa prema modulu koji sadrzi kod koji proizvodi podatke
-hd	- deli grafikon prema opisu pregrada.
-hy	- deli grafikon prema skupu pratilaca
-hb	- deli grafikon prema biografiji
-l	- šalje profil uzorke GHC event logu

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

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



Slika 2: Zauzeće hipa tokom izvršavanja programa

5 Zaključak

Ovim radom je obuhvaćena osnova i motiv upotrebe profajlera. Izloženi su koncepti profajliranja funkcionalnog jezika kao i specifičan slučaj profajliranja pomoću ghc alata. Naprednija primena profajliranja programa sa paralelnim i konkurentnim izvršavanjem može se naći na [3]. 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 [1].

Literatura

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

[2] 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.

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

[4] 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.

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

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

[7] 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

```
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