Proširenje alata za profajliranje mogućnošću prikupljanja i prikaza podataka o pokrivenosti koda tokom izvršavanja

Marina Nikolić, Mladen Nikolić i Darko Ristivojević

***Apstrakt*—U ovom radu su opisana proširenja postojećih GCC alata za profajliranje. Postojeće rešenje daje mogućnost dobijanja podataka o poktivenosti koda tek na kraju izvršavanja. Prikazano je softversko rešenje problema dobijanja ovih podataka tokom izvršavanja koda u specifičnim slučajevima kada se izvršavanje koda ne sme prekinuti. Rad pokriva i oblik prezentovanja podataka o pokrivenosti koda korisniku na način koji bi trebao da olakša njihovu analizu u delu koji se odnosi na novi grafički alat za prezentovanje prikupljenih podataka.**

***Ključne reči* — profajliranje; pokrivenost koda; GCC; GCOV;**

I. UVOD

U procesu razvoja softvera, važni segmenti su testiranje performansi i optimizacija koda. Olakšavanje ovih segmenata razvoja omogućavaju podaci o pokrivenosti koda, koje pružaju alati za profajliranje. Jedan od ovih alata se nalazi u okviru samog GCC-a pod nazivom GCOV. On pruža veoma korisne informacije o izvršavanju programa, poput broja izvršenih/neizvršenih linija/funkcija blokova i slično. Korisnik ove informacije može dobiti samo na kraju izvršavanja. Međutim, u nekim specifičnim uslovima, kada se procesi ne smeju prekidati radi profajliranja, od velike je važnosti prikupiti ovakve podatke tokom rada programa. U ovom radu će biti prikazan jedan način prilagođavanja postojećeg GCC-ovog alata ovim potrebama, proširenjem njegovih mogućnosti, kao i jedan način prezentovanja ovakvih podataka grafičkim alatom za generisanje i prikaz GCOV statistike.

II. PROFAJLIRANJE I POKRIVENOST KODA

Razvoj softvera, kao proizvoda, je mnogo više od pisanja koda. Planiranje, usklađivanje sa zahtevima klijenata, testiranje, optimizacija, održavanje su samo neki od procesa čiji značaj i kompleksnost uviđa jedino sam razvojni tim. Zbog toga je posvećena velika pažnja proučavanju novih načina za olakšavanje ovih segmenata razvoja.

Analiza programa je jedan od automatizovanih načina za pružanje pomoći programerima pri testiranju korektnosti i performansi, kao i pri optimizaciji. Ona može biti statička i dinamička.

Za statičku analizu nije neophodno izvršiti program. Vrši se obično nad izvornim kodom i pre svega se orijentiše na samu strukturu. To su razne softverske metrike, koje pružaju podatke o broju linija, količini komentara, broju praznih linija, broju operatora, operanada, paketa, klasa, metoda i slično. Ona je brža i nepristrasnija, ali sa druge strane nepreciznija i manje informativna.

Dinamicka analiza [1] koda je prikupljanje podataka o programu tokom njegovog izvršavanja. Pored strukture koda, ona uzima u obzir i ulazne podatke i ponašanje programa u realnim situacijama, pa je dosta efikasnija u pronalaženju problema. Vid dinamičke analize na koji će se ovaj rad skoncentrisati je profajliranje.

Profajliranje [2] ima za cilj prikupljanje podataka o memorijskim zahtevima programa, složenosti, iskorišćenosti pojedinih delova koda i slično. Profajliranje se vrši instrumentalizacijom programskog koda alatom koji se naziva profajler. Profajleri se mogu koristiti u razne svrhe: testiranje programa na različitim arhitekturama, određivanje kritičnih delova programa, procena kvaliteta algoritama, detektovanje petlji sa velikim brojem iteracija, nepotrebnih grananja i slično. Tehnike i podaci koje profajleri pružaju mogu se razlikovati, ali svima im je krajnji cilj isti: olakšavanje testiranja i optimizacije.

Instrumentalizacija [2] koda je sposobnost merenja izvesnih karakteristika programa, čije vrednosti ukazuju na performanse programa ili potencijalne greške u njemu. Jedna od tih karakteristika je i pokrivenost koda.

Pokrivenost koda [4], [5], [6], [7], [8] je „stepen izvršenosti koda“. Predstavlja odnos izvršenih i neizvršenih linija, blokova, grana ili funkcija izražen u procentima. Mala pokrivenost je dobar pokazatelj da u kodu postoje greške u logici ili suvišni delovi. Neki uzroci male pokrivenosti:

1. „ostaci” ranijih verzija (funkcije, klase ili neki drugi segmenti koda koji se više ne koriste, a i dalje postoji njihova definicija u izvornom kodu)
2. iscrpljivanje svih mogućnosti (preveliki broj specijalnih slučajeva dovodi do mnogo grananja i segmenata koji se u praksi vrlo retko ili nikada ne izvrše)
3. „trka” za resurse (neki procesi ne uspevaju da dobiju processor ili neki drugi resurs neophodan za svoje izvršavanje)
4. greške u logici (loše definisani uslovi u granama ili petljama, pozivi pogrešnih funkcija i slično)

Marina Nikolić – Univerzitet u Beogradu, Matematički fakultet, Studentski trg 16, Beograd, Srbija, Institut RT-RK za sisteme bazirane na računarima, Novi Sad, Srbija (e-mail: marina.nikolic@rt-rk.com)

Mladen Nikolić – Institut RT-RK za sisteme bazirane na računarima, Novi Sad, Srbija (e-mail: mladen.r..nikolic@rt-rk.com)

5. „preopterećenost” pojedinih niti (pucanje nekih niti koje obavljaju najzahtevnije poslove može dovesti do polovično izvršenog programa)

III. Postojeća rešenja

U okviru GNU-ovog GCC-a se nalazi jedan od alata za testiranje pokrivenosti koda pod nazivom GCOV [9], [10]. Na osnovu izvornog koda i podataka prikupljenih tokom izvršavanja programa, on generiše izveštaj sa ekstenzijom .gcov, u kome za svaku liniju postoji informacija o tome da li je izvršena i koliko puta. Takođe izveštaj sadrži i informacije o broju pokretanja programa, kao i ukupnu statistiku u vidu procenta izvršenih linija naspram ukupnog broja.

Korišćenjem posebnih opcija, GCOV izveštaj se može unaprediti i statističkim podacima o blokovima, granama, funkcijama i slično. Testiranje pokrivnosti koda GCOV alatom se sprovodi u nekoliko koraka:

1. Prevođenje izvornog koda sa posebnim opcijama (flegovima): -fprofile-arcs -ftest-coverage, koje služe za generisanje dodatnih informacija potrebnih GCOV-u
2. Izvršavanje programa proizvoljni broj puta
3. Pozivanje alata ključnom rečju GCOV, nakon čega sledi putanja do izvršnog fajla. U ovom koraku se mogu navesti i gore pomenute posebne opcije za unapređivanje izveštaja.

Pored izveštaja, GCOV generiše i dva tipa pomoćnih fajlova:

1. GCOV NOTES FILE (.gcno), koji se generiše pri prevođenju i sadrži informacije o strukturi izvornog koda
2. GCOV DATA FILE (.gcda), koji se generiše (ili samo modifikuje ako već postoji) nakon svakog izvršavanja i sadrži konkretne vrednosti brojača linija, funkcija i slično

Podaci o programu se tokom izvršavanja čuvaju u posebnoj, globalno definisanoj strukturi, tipa gcov_info. Ovi podaci predstavljaju jezgro informacija koje GCOV pruža, ali GCOV ih ne može direktno koristiti iz tri razloga:

1. Nalaze se u oviru memoriskog segmenta analiziranog programa i nisu dostupni alatima koji ih mogu koristiti
2. nisu akumulativni (odnose se samo na poslednje izvršavanje)
3. nisu u odgovarajućem formatu (GCOV ih ne ume pročitati)

Ove probleme rešava dinamička biblioteka: libgcov, koja od podataka iz structure tipa gcov_info generiše gcda fajlove, koji zadovoljavaju gore pomenuto.

Kako se gcda fajlovi generišu tek na kraju programa, ovi podaci nisu dostupni GCOV alatu, pa ni samom korisniku, tokom izvršavanja. Ovo može predstavljati veliki nedostatak u nekim specifičnim slučajevima, poput programa kod kojih:

1. prekid nije dozvoljen
2. vreme rada je izuzetno dugo
3. ovi podaci se koriste samo tokom rada

Konkretni primeri bi bili sistemi za rad u realnom vremenu, serveri i slično. Na taj nedostatak će se koncentrisati dalje istraživanje u okviru ovog rada.

IV. Razmatrana rešenja

Problem prikupljanja podataka tokom izvršavanja se suštinski može rešiti na dva načina:

1. prilagođavanjem samog alata
2. prilagođavanjem biblioteke

Prilagođavanje samog alata bi označavalo promenu formata ulaznih podataka za generisanje izveštaja. Izvršni program bi u deljenoj memoriji definisao strukturu gcov_info, kojoj bi se pristupalo iz GCOV alata. Pored toga što bi implementiranje ove ideje bilo znatno složenije, zbog velikih promena koje bi se morale sprovesti, kako u samom alatu, tako i u prevodiocu, i same performanse bi značajno opale. Pristup deljenoj memoriji bi postao kritična sekcija oba programa, pa bi bilo neophodno uspostaviti neki vid zaštite konzistentnosti podataka. Postavljanje semafora bi rešilo problem konkurentnog pristupa brojačima korisničkog programa i GCOV-a, ali bi dosta usporilo izvršavanje oba procesa.

Analiziranjem implementacije GCOV alata za profajliranje, može se zaključiti da je kontrola nad dobijanjem podataka o pokrivenosti koda zapravo prepuštena biblioteci libgcov. Modifikacijom trenutka kreiranja gcda fajlova, može se mnogo jednostavnije i efikasnije postići cilj pravovremenog dobijanja podataka. Zbog toga je proširenje GCOV alata, kojim se ovaj rad bavi, sprovedeno upravo na drugi, gore pomenuti, način.

Uzrok nemogućnosti dobijanja podataka o pokrivenosti koda tokom izvršavanja programa je, na prvi pogled, prekasno generisanje gcda fajlova. Međutim, to je u stvari samo posledica takvog pristupa biblioteke libgcov, u kome ona preuzima potpunu kontrolu nad podacima od interesa. Funkcije biblioteke su definisane tako da se odvijaju bez interakcije sa korisnikom. Potrebno je, dakle, učiniti vidljivom interfejsnu funkciju za generisanje gcda fajlova, odnosno dati korisniku kontrolu nad njenim pozivanjem. Sva ostala unapređenja proističu iz prilagođavanja novonastaloj situaciji.

V. Implementacija

Implementacija podrazumeva izgradnju nove dinamičke biblioteke pod nazivom: libcoverage. Ona je zasnovana na već postojećoj biblioteci libgcov i koristi se umesto nje kada su podaci potrebni u toku izvršavanja.

Proces inicijalizacije globalno definisane strukture tipa gcov_info je isti u oba slučaja. Funkcija gcov_exit, čija je uloga ekstrakcija podataka iz ove strukture i generisanje gcda fajlova na osnovu njih, je uz određene izmene evoluirala u funkciju drew_coverage, koja je dostupna korisničkom programu kao interfejs biblioteke.

U okviru same funkcije gcov_exit izvršeno je nekoliko manjih korekcija:

1. pristup brojačima strukture tipa gcov_info je baziran na nizovima, umesto na računanju pomeraja u memoriji
2. ispis u gcda fajl se vrši jednim sistemskim pozivom za veći broj bajtova, umesto pojedinačnog poziva za svaki bajt
3. nešto drugačiji pristup računanju statističkih podataka

Biblioteka libcoverage drugačije upravlja i verzijama alata GCOV. U okviru same biblioteke se vrši prilagođavanje

verziji alata, pa je biblioteka jedinstvena za sve verzije GCC-a nastale nakon promene strukture gcov_info (4.7 i sve kasnije verzije). Smanjena je i zavisnost od drugih zaglavlja standradne biblioteke programskog jezika C. Smanjena je zavisnost koda od drugih delova GCOV biblioteke i optimizovana je uklanjanjem struktura koje biblioteka ne koristi. Spajanje podataka tekućeg izvršavanja i onih sačuvanih u gcda fajlu je dosta pojednostavljeno i u potpunosti prepušteno jednoj funkciji: __gcov_merge_add__.

Prevođenjem programa sa posebnim flegovima za istrumentalizaciju, omogućava se praćenje makropodataka, koji opisuju samo izvršavanje programa. Ti podaci se nalaze u listi posebnih struktura, koja se inicijalizuje i popunjava u toku izvršavanja programa. Za svaki objektni fajl postoji jedna struktura u listi. Ova struktura se naziva gcov_info i tokom izvršavanja programa se nalazi u deljenoj memoriji. Formatirana je na sledeći način:

```
struct gcov_info{
  gcov_unsigned_t version;
  struct gcov_info *next
  gcov_unsigned_t stamp;
  const char *filename;
  gcov_merge_fn merge[GCOV_COUNTERS];
  unsigned n_functions;
  const struct gcov_fn_info **functions;
};
```

U svakoj gcov_fn_info strukturi se nalaze podaci o datoj funkciji: njeno ime, identifikator, kao i vrednosti brojača koji čuvaju informaciju o tome koliko je puta funkcija izvršena.

Implementacija funkcije drew_coverage:

```
for (ptr = gcov_info_list; ptr; ptr = ptr->next)
  values[i]:=ptr->functions[i]->values;
end
open(gcda_file)
check_and_write(magic_number,version,timestamp)
for (f = 0; f < values.length; f++)
  check_and_write(function_tag,values[f])
end
check_and_write(program_summary)
write_buffer_to_file()
```

Zadatak funkcije drew_coverage je da iterira kroz listu gcov_info struktura i prikupi potrebne podatke. Nakon prikupljanja, ovakve podatke treba obraditi (izračunati sumarne podatke po određenom kriterijumu: po funkciji, po modulu, ukupno, ...) i generisati gcda fajlove. Vrednosti brojača se upisuju u fajl sa ekstenzijom gcda po određenom formatu (razlikovanje se vrši na osnovu posebnih oznaka definisanih u alatu gcov). Na osnovu ovog fajla i gcno fajla dobijenog prevođenjem, koristeći i informacije iz izvornog koda, GCOV alat generiše izveštaj. U momentu poziva funkcije drew_coverage, uzimaju se podaci koji su do tada upisani u gcov_info strukturu.

VI. PRIMER RADA BIBLIOTEKE

Neka je dat fajl test.c sa sledecim sadržajem:

```
#include <stdio.h>
#include "coverage.h"

int write_dot(void){
  printf(".\n");
}
int main(){
  int i = 1;
  for (i = 1; i <=10 ; i++){
     write_dot();
  }
  if(i == 10)
     printf("If branch\n");
  else
     printf("Else branch\n");
  drew_coverage();
  printf("After drew_coverge\n");
  return 0;
}
```

Prevođenje se vrši komandom:

```
gcc -fprofile-arcs -ftest-coverage test.c
-lcoverage -o test
```

Rezultat:

1. Neprazan binarni fajl: test.gcno
2. Izvršni fajl: test

Pokretanje se vrši komandom:

```
./test
```

Izvršni fajl se može pokrenuti proizvoljan broj puta (npr. 2).

Rezultat:

1. Neprazan binarni fajl: test.gcda
2. Ispis u terminalu:

```
.
.
.
.
.
.
.
.
.
.
Else branch
After drew_coverge
.
.
.
.
.
.
.
.
.
.
Else branch
After drew_coverge
```

Poziv alata:

```
gcov test
```

Rezultat:

1. Izveštaj: test.c.gcov
2. Ispis u terminalu:

```
File 'test.c'
Lines executed:76.92% of 13
Creating 'test.c.gcov'
```

Izgled izveštaja:

```
    -:    0:Source:1.c
    -:    0:Graph:1.gcno
    -:    0:Data:1.gcda
    -:    0:Runs:3
    -:    0:Programs:1
    -:    1:#include <stdio.h>
    -:    2:#include "coverage.h"
    -:    3:
   20:    4:int write_dot(void){
   20:    5:    printf(".\n");
   20:    6:}
    2:    7:int main(){
    2:    8:    int i = 1;
   22:    9:    for (i = 1; i <=10 ; i++){
   20:   10:        write_dot();
    -:   11:    }
    2:   12:    if(i == 10)
#####:   13:        printf("If branch\n");
    -:   14:    else
    2:   15:        printf("Else branch\n");
    2:   16:    drew_coverage();
#####:   17:    printf("After drew_coverge\n");
#####:   18:    return 0;
    -:   19:}
```

Zapažanje:

1. linije 17 i 18 su označene kao neizvršene, jer je poziv funkcije drew_coverage pre njih
2. brojači ostalih linija se poklapaju sa očekivanim vrednostima izvršavanja tih linija

VII. PERFORMANSE

A. *Memorija i bezbednost*

Alatom za memorijsko profajliranje Valgrind, utvrđeno je da biblioteka libcoverage ne uzrokuje probleme poput curenja memorije, pisanja van alociranog prostora, niti je karakteriše bilo kakvo drugo nepravilno upravljanje memorijom.

Oslobađanje celokupne dinamički alocirane memorije postignuto je komandama:

1. free(gcov_var.buffer);
2. fclose(gcov_var.file);

Valgrind je potvrdio porukom:

```
==1878== Memcheck, a memory error detector
==1878== Copyright (C) 2002-2015, and GNU GPL'd, by
Julian Seward et al.
==1878== Using Valgrind-3.11.0 and LibVEX; rerun
with -h for copyright info
==1878== Command: ./test
==1878==
.
.
.
.
.
.
.
.
.
.
Else branch
Code after drew_coverge function
==1878==
==1878== HEAP SUMMARY:
==1878==     in use at exit: 8,544 bytes in 2 blocks
==1878==   total heap usage: 2 allocs, 0 frees,
8,544 bytes allocated
==1878==
==1878== LEAK SUMMARY:
==1878==    definitely lost: 0 bytes in 0 blocks
==1878==    indirectly lost: 0 bytes in 0 blocks
==1878==      possibly lost: 0 bytes in 0 blocks
==1878==    still reachable: 8,544 bytes in 2 blocks
==1878==         suppressed: 0 bytes in 0 blocks
==1878== Rerun with --leak-check=full to see details
of leaked memory
==1878==
==1878== For counts of detected and suppressed
errors, rerun with: -v
==1878== ERROR SUMMARY: 0 errors from 0 contexts
(suppressed: 0 from 0)
```

B. *Složenost*

Složenost u najgorem slučaju ostaje ista, jer izmene u kodu u odnosu na staru biblioteku ne menjaju samu složenost, nego je kod reorganizovan.

Korišćenje stare biblioteke je potpuno ekvivalentno korišćenju nove, gde bi se funkcija drew_coverage zvala na samom kraju programa, korišćenjem atexit funkcije. Zbog toga, složenost je u najgorem slučaju (atexit) ista. Međutim, u velikim sistemima, sa velikom količinom koda, zvanje funkcije pre kraja izvršavanja, može dovesti do poboljšanja složenosti rada same biblioteke, usled obrade manjeg skupa podataka.

C. *Upotreba*

Upotreba je relativno jednostavna. Prilikom prevođenja od objektnog do izvršnog fajla, treba linkovati dinamičku biblioteku libcoverage. U okviru korisničkog programa, neophodno je još uključiti zaglavlje coverage.h i pozvati funkciju drew_coverage u željenom delu programa.

VIII. UNAPREĐENJE ILI NOVI ALAT

U situacijama kada se korisniku poverava upravljanje određenim procesima, uvek se postavlja pitanje o ispravnosti te odluke. U ovom slučaju, korisnici su obično i sami programeri, te su dovoljno stručni da im se poveri poziv jedne

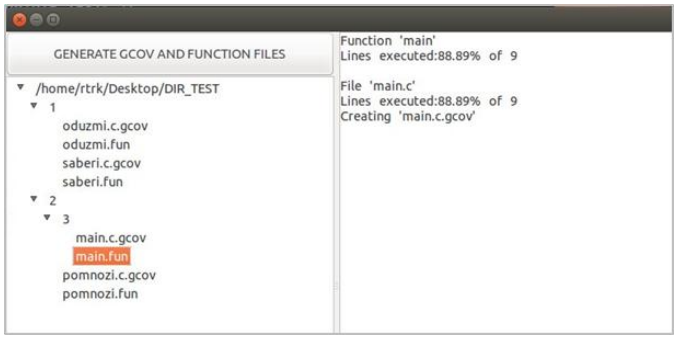
funkcije. Takođe, pozivanjem funkcije drew_coverage kao argumenta funkcije atexit iz standardne biblioteke stdlib, postiže se isti efekat kao i pri korišćenju biblioteke libgcov.

IX. GRAFIČKI ALAT ZA PREZENTOVANJE PODATAKA O POKRIVENOSTI KODA

U ovom delu rada biće predstavljen novi grafički alat za prezentovanje podataka o pokrivenosti koda, prikupljenih GCC-ovim alatom za profajliranje GCOV, pod nazivom code_coverage_viewer. Alat je razvijen u programskom jeziku Python, korišćenjem biblioteke za grafičku podršku wxPython. Osnovna uloga ovog alata je generisanje tekstualnih izveštaja o pokrivenosti koda i njihov prikaz korisniku na način koji bi trebao da olakša dalji rad sa njima.

A. *Opis interfejsa*

Interfejs se sastoji iz dva panela i linije menija. Pored standardnih opcija za upravljanje veličinom prozora i izlaz iz programa, linija menija sadrži i opciju za odabir radnog direktorijuma u okviru menija File, pod nazivom: „Choose workspace“. Levi panel sadrži jedno dugme za generisanje fajlova: „Generate gcov and fun files” i drvoliku strukturu koja prikazuje lokaciju generisanih fajlova u radnom direktorijumu. U desnom panelu, nalazi se polje u kome se prikazuje sadržaj odabranog fajla.



Sl. 2. Interfejs grafičkog alata

Generišu se dva tipa fajlova:

1. Standardni gcov fajlovi (sa ekstenzijom .gcov), koji sadrže statistiku po linijama
2. Fun fajlovi (sa ekstenzijom .fun), koji sadrže statistiku po funkcijama

Generisanje fajlova se vrši iz aplikacije, pozivom alata GCOV, kome se prosleđuje putanja do radnog direktorijuma sa izvornim kodom koji se analizira. Prvi tip se generiše standardnim pozivom alata bez opcija, dok se drugi tip dobija preusmeravanjem izlaza poziva GCOV-a sa opcijom –f.

```
        -:    0:Source:main.c
        -:    0:Graph:main.gcno
        -:    0:Data:main.gcda
        -:    0:Runs:1
        -:    0:Programs:1
        -:    1:#include <stdio.h>
        -:    2:#include "coverage.h"
        -:    3:
        -:    4:extern int saberi(int a, int b);
        -:    5:extern int oduzmi(int a, int b);
        -:    6:extern int pomnozi(int a, int b);
        -:    7:
        1:    8:int main(){
        -:    9:
        1:   10:  int a = 2;
        1:   11:  int b = 3;
        1:   12:  int c = saberi(a,b);
        1:   13:  int d = oduzmi(a,b);
        1:   14:  int e = pomnozi(a,b);
        -:   15:
        -:   16:
        1:   17:  printf("Zbir je %d, razlika je %d, proizvod je %d\n", c, d, e);
        1:   18:  drew_coverage();
        -:   19:
    #####:   20:  return 0;
        -:   21:
        -:   22:}
```

Sl. 3. Prikaz izveštaja u grafiičkom alatu

```
Function 'main'
Lines executed:88.89% of 9

File 'main.c'
Lines executed:88.89% of 9
Creating 'main.c.gcov'
```

S2. 4. Prikaz funkcijskog izveštaja u grafiičkom alatu

B. *Dalja unapređenja*

Ovim alatom pokrivene su samo dve mogućnosti GCOV profajlera: statistika po linijama i statistika po funkcijama. Dalje unapređivanje alata bi moglo teći u smeru generisanja više vrsta izveštaja. To se može postići dodatnim pozivanjima alata GCOV sa n ekim drugim opcijama, za druge vrste statistika. Razvoj novih funkcionalnosti je moguć, i zavisi samo od potreba budućih razvijaoca.

X. ZAKLJUČAK

U ovom radu je opisano jedno softversko rešenje problema prikupljanja i prezentovanja podataka o pokrivenosti koda u toku izvršavanja programa.

Za tu svrhu je izgrađena nova dinamička biblioteka libcoverage, po modelu postojeće libgcov, koja pruža GCC-ovom alatu GCOV ovu funkcionalnost. Poziv funkcije za prosleđivanje podataka GCOV alatu u vidu fajlova formatiranih na način na koji ih on može pročitati i pretočiti u izveštaj (gcda fajlovi), je prepušten razvijaocu softvera. Ovim je omogućeno da se na nivou samog koda koji se profajlira, donese odluka o trenutku u kome su podaci o pokrivenosti koda neophodni.

Takođe, izgrađen je i grafički korisnički interfejs za struktuirani prikaz ovih podataka. On omogućava generisanje i prezentovanje dva tipa izveštaja: linijski i funkcijski orijentisani izveštaji. Prikaz se vrši u vidu drveta koje odražava strukturu koda po direktorijumima, a

implementirana je i mogucnost prikaza sadržaja željenog izveštaja.

Korišćenjem ovakvog profajliranja, procesi testiranja performansi i optimizacije su omogućeni, ili barem znatno olakšani, za programe čije se izvršavanje ne sme prekidati, poput sistema za rad u realnom vremenu ili servera.

U vreme pisanja ovog rada, GCC nije imao mogućnost profajliranja u toku izvršavanja programa. Stoga je ovaj istraživački rad bio orijentisan ka implementiranju ove funkcionalnosti u GCC.

Novom bibliotekom zadržana je i stara funkcionalnost: profajliranje na kraju izvršavanja programa (pozivom funkcije drew_coverage kao argumenta atexit funkcije), ali i omogućena nova: profajliranje u bilo kom trenutku (statički ili interaktivno). Ovime je postignuto unapređenje stare biblioteke.

U okviru LLVM-a postoji biblioteka koja omogućava profajliranje i u toku izvršavanja programa. Poređenje i unapređivanje performansi po ugledu na nju je tema narednih istraživanja.

Dalji tok razvoja bi takođe mogao biti orijentisan ka pružanju ove funkcionalnosti drugim alatima za profajliranje, drugih prevodilaca, kao i razvoj sličnih grafičkih alata, koji bi drugačije pristupili izlaganju prikupljenih podataka.

ZAHVALNICA

Ovaj rad je delimično finansiran od strane Ministarstva za prosvetu, nauku i tehnološki razvoj Republike Srbije, na projektu broj: III_044009_2.

LITERATURA

[1] V. Gupta, *Measurement of Dynamic Metrics Using Dynamic Analysis of Programs*, APPLIED COMPUTING CONFERENCE (ACC '08), Istanbul, Turkey, May 27-30, 2008.

[2] A. Homescu, "Profile-guided automated software diversity", Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO). IEEE Computer Society, 2013, https://www.ics.uci.edu/~ahomescu/multicompiler_cgo13.pdf

[3] Source code instrumentation overview, https://www.ibm.com/support/knowledgecenter/SSSHUF_8.0.0/com.ibm.rational.testrt.doc/topics/cinstruovw.html

[4] L. Brader, H. Hilliker, A. Wills, "Unit Testing: Testing the Inside" in *Testing for Continuous Delivery with Visual Studio 2012*, ch. *2*, sec. *6*, Microsoft Developer Guidance, 2013.

[5] L. William, B. Smith, S. Heckman, "Test Coverage with EclEmma", Technical Report Raleigh, 2008.

[6] A. Glower, "In pursuit of code quality: Don't be fooled by the coverage report", IBM Developer Works blog post, 2006.

[7] B. Marick, "How to misuse code coverage", Proceedings of the 16th Interational Conference on Testing Computer Software,1999, http://www.exampler.com/testing-com/writings/coverage.pdf

[8] A. Piziali, "Code coverage," in *Functional verification coverage measurement and analysis*, ch. *5*, Springer Science & Business Media, 2007.

[9] Gcov,*https://gcc.gnu.org/onlinedocs/gcc/Gcov.html*

[10] Code Coverage using Gcov, *https://web.archive.org/web/20140409083331/http://xview.net/papers/gcov/code_coverage_gcov.pdf*

ABSTRACT

This paper describes the extension of the existing GCC profiling tools. The existing solution gives the possibility of obtaining data on code coverage only at the end of execution. Here is presented a software solution to the problem of obtaining these data during the execution of code in specific cases where the execution of code must not be terminated. The paper also covers forms of data presentation of the code coverage in a way that should facilitate their analysis in the part that relates to the new graphical tool for presenting collected data.

Expansion of profiling tools with possibility of collecting and displaying data on code coverage during execution

Marina Nikolić, Mladen Nikolić and Darko Ristivojević