

# Opis procesa analize i otklanjanja grešaka programske podrške primenom alata Velgrind

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

Dragutin Ilić, Karadžić Aleksandra, Šuka Aleksandar  
dragutin\_ilic@yahoo.com, karadzic.matf@gmail.com, suka61s@hotmail.com

9. april 2015.

## Sažetak

Programska podrška je sve prisutnija, računari trpe poraste u performansama i mogućnostima. Kako korisnički zahtevi sve više rastu, samim tim raste kompleksnost procesa razvoja softvera. Javlja se potreba za alatima koji služe za detekciju i otklanjanje grešaka, kao i za analiziranje i pravljenje profila korisničkih programa. Cilj ovog rada je da opiše i kroz primere približe jedan takav alat, Velgrind.

## Sadržaj

<b>1</b>	<b>Uvod</b>	<b>2</b>
<b>2</b>	<b>Struktura i upotreba Velgrind alata</b>	<b>2</b>
2.1	Memcheck . . . . .	3
2.2	Cachegrind . . . . .	4
2.3	Helgrind . . . . .	6
2.3.1	Loša upotreba API-ja za programiranje POSIX niti	7
2.3.2	Loš redosled zaključavanja . . . . .	8
2.3.3	Pristup memoriji bez adekvatnog zaključavanja ili sinhronizacije . . . . .	8
2.4	DRD - Data Race Detector . . . . .	8
2.5	Massif . . . . .	10
2.6	Callgrind . . . . .	11
<b>3</b>	<b>Zaključak</b>	<b>12</b>
	<b>Literatura</b>	<b>12</b>

# 1 Uvod

Velgrind je platforma za pravljenje alata za dinamičku binarnu analizu koda. Dinamička analiza obuhvata analizu korisničkog programa u izvršavanju, dok binarna analiza obuhvata analizu na nivou mašinskog koda, snimljenog ili kao objektni kod (nepovezan) ili kao izvršni kod (povezan). Postoje Velgrind alati koji mogu automatski da detektuju probleme sa memorijom, procesima kao i da izvrše optimizaciju samog koda. Velgrind se može koristiti i kao alat za pravljenje novih alata. Velgrind distribucija trenutno broji sledeće alate: detektor memorijskih grešaka, detektor grešaka niti, optimizator skrivene memorije i skokova, generator grafa skrivene memorije i predikcije skoka i optimizator korišćenja dinamičke memorije. Velgrind radi na sledećim arhitekturama: X86/Linux, AMD64/Linux, ARM/Linux, ARM64/Linux, PPC32/Linux, PPC64/Linux, PPC64LE/Linux, S390X/Linux, MIPS32/Linux, MIPS64/Linux, TILEGX/Linux, X86/Solaris, AMD64/Solaris, ARM/Android (2.3.x i novije), ARM64/Android, X86/Android (4.0 i novije), MIPS32/Android, X86/Darwin and AMD64/Darwin (Mac OS X 10.10). U narednim poglavljima biće detaljnije opisana struktura Velgrinda i njegovih alata, kao i primeri prevazilaženja nekih od problema sa kojima se programeri svakodnevno susreću.

## 2 Struktura i upotreba Velgrind alata

Alat Velgrinda se sastoji od alata za dinamičku analizu koda koji se kreira kao dodatak pisan u C programskom jeziku na jezgro Velgrinda (*Jezgro Velgrinda + alat koji se dodaje = Alat Velgrinda*). Jezgro Velgrinda omogućuju izvršavanje klijentskog programa, kao i snimanje izveštaja koji su nastali prilikom analize samog programa. Alati Velgrinda koriste metodu bojenja vrednosti. To znači da oni svaki registar i memorijsku vrednost boje (zamenjuju) sa vrednošću koja govori nešto dodatno o originalnoj vrednosti. Proces rada svakog alata Velgrinda je u osnovi isti. Alat analizira dobijen kod i vrši translaciju - proces koji se sastoji od sledećih koraka:

1. *Disasembliranje (razgradnja)* - prevođenje mašinskog koda u ekvivalentni interni skup instrukcija koje se nazivaju međukod instrukcije.
2. *Optimizacija* predstavlja proces uklanjanja viška instrukcija koje su dodate prilikom razgradnje pojedinačnih instrukcija na jednu ili više međukod instrukcija.
3. *Instrumentacija* predstavlja proces ubacivanja novih instrukcija koje će se koristiti za analizu izvornih instrukcija u dobijen optimizovan skup međukod instrukcija. Treba napomenuti da ubačene instrukcije ne narušavaju konzistentno izvršavanje originalnog koda.
4. *Dodela registara* vrši zamenu virtualnih registara, koji su određeni u svakoj međukod instrukciji, sa jednim od pravih registara koji su za to određeni.
5. *Izvršavanje (emisija) koda* je deo kada se sve međukod instrukcije, originalne i dodate, prevode u mašinske reči ciljne platforme i snimaju u prevedeni osnovni blok.

Translacija se čuva u memoriji i izvršava se po potrebi. Jezgro Velgrinda troši najviše vremena na sam proces pravljenja, pronalaženja i izvršavanja translacije (originalni kod se nikad ne izvršava). Treba napomenuti da sve ove korake osim instrumentacije izvršava jezgro Velgrinda

dok samu instrumentaciju izvršava određeni alat koji smo koristili za analizu izvornog koda. Rezultati instrumentacije svakog alata Velgrinda kao i njihova upotreba će biti opisani u nastavku.

## 2.1 Memcheck

Memorijske greške često se najteže detektuju, a samim tim i najteže otklanjaju. Razlog tome je što se takvi problemi ispoljavaju nedeterministički i nije ih lako reprodukovati. Memcheck je alat koji detektuje memorijske greške korisničkog programa. Kako ne vrši analizu izvornog koda već mašinskog, Memcheck ima mogućnost analize programa pisanom u bilo kom jeziku.

Za programe pisane u jezicima C i C++ detektuje sledeće uobičajne probleme:

1. Pristupanje nedopuštenoj memoriji, na primer prepisivanje blokova na hipu, prepisivanje vrha steka i pristupanje memoriji koja je već oslobođena.
2. Korišćenje nedefinisanih vrednosti, vrednosti koje nisu inicijalizovane ili koje su izvedene od drugih nedefinisanih vrednosti.
3. Neispravno oslobađanje hip memorije, kao što je duplo oslobađanje hip blokova ili neuporenog korišćenja funkcija `malloc/new/new[]` i `free/delete/delete[]`.
4. Preklapanje parametara prosleđenih funkcijama.
5. Čurenje memorije.

U nastavku je kroz primere ilustrovana detekcija navedenih problema sa memorijom.

Na slici 1 je prikazan jednostavan program, koji sadrži tri nedefinisane vrednosti koje uzrokuju grešku. Da bi pustili preveden program kroz Velgrind, potrebno je zadati sledeću komandu u terminalu:

```
valgrind -tool=memcheck ./main
```

Slika 2.1 prikazuje rezultujući izlaz. Prvi deo, odnosno prve tri linije se štampaju prilikom pokretanja bilo kog alata koji je u sklopu Velgrinda, u ovom slučaju Memcheck. Sledeći deo nam pokazuje poruke o greškama koje je Memcheck pronašao u programu. Poslednja linija prikazuje sumu svih grešaka koje je alat pronašao i štampa se po završetku rada.

Na slici 3 je prikazan izlaz iz Velgrinda kada se otkrije korišćenje nedefinisanih vrednosti. U programu nedefinisana promenljiva može više puta da se kopira, Memcheck prati sve to, beleži podatke o tome, ali ne prijavljuje grešku. U slučaju da se nedefinisane vrednosti koriste na način da od te vrednosti zavisi dalji tok programa ili ako je potrebno prikazati vrednosti nedefinisane promenljive, Memcheck prijavljuje grešku. Da bi mogli da vidimo glavni izvor korišćenja nedefinisanih vrednosti u programu, dodaje se opcija `-trace-origins=yes`.

Memcheck beleži podatke o svim dinamičkim blokovima koji su alocirani tokom izvršavanja programa. Kada program prekine sa radom, Memcheck ima podatke o svim neoslobođenim memorijskim blokovima. Ako je opcija `-leak-check` podešena na odgovarajući način, za svaki neoslobođen blok Memcheck određuje da li je moguće pristupiti tom bloku preko pokazivača. Ako je uključena opcija `-leak-check=full`, Memcheck će prikazati detaljan izveštaj o svakom definitivno ili moguće izgubljenom bloku, kao i o tome gde je on alociran.

```

int main()
{
    char *p;

    // Allocation #1 of 19 bytes
    p = (char) malloc(19);

    // Allocation #2 of 12 bytes
    p = (char) malloc(12);

    free(p);

    // Allocation #3 of 16 bytes
    p = (char) malloc(16);

    return 0;
}

```

Slika 1: Primer programa main.c

```

==6942== Memcheck, a memory error detector
==6942== Copyright (C) 2002-2010, and GNU GPL'd, by Julian Seward et al.
==6942== Using Valgrind-3.7.0.SVN and LibVEX; rerun with -h for copyright info
==6942== Command: ./main.out
==6942==
==6942== Invalid free() / delete / delete[] / realloc()
==6942==    at 0x484966C: free (vg_replace_malloc.c:320)
==6942==    by 0x40069B: main (main.c:12)
==6942== Address 0xffffffff90 is not stack'd, malloc'd or (recently) free'd
==6942==
==6942==
==6942== HEAP SUMMARY:
==6942==    in use at exit: 47 bytes in 3 blocks
==6942== total heap usage: 3 allocs, 1 frees, 47 bytes allocated
==6942==
==6942== 12 bytes in 1 blocks are definitely lost in loss record 1 of 3
==6942==    at 0x484AA3D: malloc (vg_replace_malloc.c:212)
==6942==    by 0x4006A3: main (main.c:11)
==6942==
==6942== 16 bytes in 1 blocks are definitely lost in loss record 2 of 3
==6942==    at 0x484AA3D: malloc (vg_replace_malloc.c:212)
==6942==    by 0x4006A7: main (main.c:14)
==6942==
==6942== 19 bytes in 1 blocks are definitely lost in loss record 3 of 3

```

## 2.2 Cachegrind

Cachegrind je alat koji simulira i prati pristup brzoj memoriji mašine na kojoj se program analizira i izvršava. On simulira memoriju mašine, koja ima prvi nivo brze memorije podeljene u dve odvojene nezavisne sekcije, sekcija brze memorije u kojoj se smeštaju instrukcije (I1) i sek-

```

==6942== at 0x484AA90: malloc (vg_replace_malloc.c:212)
==6942== by 0x40066B: main (main.c:8)
==6942==
==6942== LEAK SUMMARY:
==6942== definitely lost: 47 bytes in 3 blocks
==6942== indirectly lost: 0 bytes in 0 blocks
==6942== possibly lost: 0 bytes in 0 blocks
==6942== still reachable: 0 bytes in 0 blocks
==6942== suppressed: 0 bytes in 0 blocks
==6942==
==6942== For counts of detected and suppressed errors, rerun with: -v
==6942== ERROR SUMMARY: 4 errors from 4 contexts (suppressed: 14 from 5)

```

Slika 2: Primer izlaza za program main.c

```

==24228== Conditional jump or move depends on uninitialised value(s)
==24228== at 0x48A4604: vfprintf (in /lib/libc-2.11.1.so)
==24228== at 0x48AEA97: printf (in /lib/libc-2.11.2.so)
==24228== by 0x40067B: main (main1.c:4)

```

Slika 3: Primer ispisa nakon puštanja programa kroz Velgrind

cije u kojoj se smestaju podaci (D1), potpomognute drugim nivoom brze memorije (L2). Ovaj način konfiguracije odgovara mnogim modernim mašinama.

Međutim, postoje mašine koje imaju više od dva nivoa brze memorije (I1, D1 i LL). U ovom slučaju Cachegrind simulira prvi i poslednji nivo brze memorije, s obzirom da poslednji nivo ima najviše uticaja na vreme izvršavanja, jer maskira pristup glavnoj memoriji.

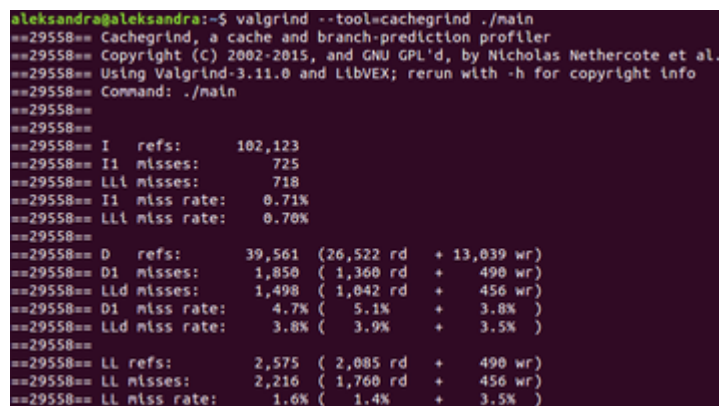
Cachegrind sakuplja sledeće statistike:

- I čitanje brze memorije (Ir, što predstavlja broj izvršenih instrukcija), I1 broj promašenih čitanja (I1mr) i broj promašaja čitanja instrukcija nivoa LL brze memorije (ILmr)
- D čitanje brze memorije (Dr broj čitanja memorije), D1 broj promašaja čitanja (D1mr), broj promašaja čitanja podataka nivoa LL brze memorije (DLmr).
- D pisanje u brzu memoriju (Dw je jednak broju pisanja u memoriju), D1 broj promašaja pisanja u brzu memoriju (D1mw) i broj promašaja pisanja podataka u nivou LL brze memorije.
- Broj uslovno izvršenih grana (Bc) i broj promašaja uslovno izvršenih grana (Bcm).
- Broj indirektno izvršenih grana (Bi) i broj promašaja indirektno izvršenih grana (Bim).

Iz predhodnih statistika možemo da zaključimo da je broj pristupa D1 delu brze memorije jednak zbiru D1mr i D1mw, dok broj pristupa

nivou LL brze memorije je jednak zbiru ILmw, DLMr i DLMw broja pristupa. Ova statistika se može prikupljati na nivou celog programa, ali se može prikupljati i na nivou pojedinačnih funkcija. Može se prikupiti i broj pristupa brzoj memoriji za svaku liniju koda originalnog programa. Kod modernih mašina L1 promašaj košta oko 100 procesorskih ciklusa, LL promašaj košta oko 200 procesorskih ciklusa, a promašaji uslovno i indirektno izvršene grane od 10 do 30 procesorskih ciklusa.

Da bi mogli koristiti ovaj alat neophodno je navesti `-tool = cachegrind` u Velgrindovoj komandnoj liniji.



```

aleksandra@aleksandra:~$ valgrind --tool=cachegrind ./main
==29558== Cachegrind, a cache and branch-prediction profiler
==29558== Copyright (c) 2002-2015, and GNU GPL'd, by Nicholas Nethercote et al.
==29558== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==29558== Command: ./main
==29558==
==29558==
==29558== I   refs:      102,123
==29558== I1 misses:      725
==29558== L1l misses:     718
==29558== I1 miss rate:   0.71%
==29558== L1l miss rate: 0.70%
==29558==
==29558== D   refs:      39,561 (26,522 rd + 13,039 wr)
==29558== D1 misses:     1,850 ( 1,360 rd +   490 wr)
==29558== L1d misses:   1,498 ( 1,042 rd +   456 wr)
==29558== D1 miss rate:  4.7% (  5.1% +  3.8% )
==29558== L1d miss rate: 3.8% (  3.9% +  3.5% )
==29558==
==29558== LL refs:       2,575 ( 2,085 rd +   490 wr)
==29558== LL misses:     2,216 ( 1,760 rd +   456 wr)
==29558== LL miss rate:  1.6% (  1.4% +  3.5% )

```

Slika 4: Cachegrind

## 2.3 Helgrind

Helgrind je alat u sklopu programskog paketa Valgrind koji otkriva greške sinhronizacije prilikom upotrebe modela niti POSIX (Portable Operating System Interface - familija standarda specifikovanih od strane IEEE organizacije za održavanje kompatibilnosti između operativnih sistema). Da bi mogli koristiti ovaj alat neophodno je navesti `-tool = helgrind` u Velgrindovoj komandnoj liniji. Glavne abstrakcije u POSIX nitima su: skup niti koji deli zajednički adresni prostor, kreiranje niti, čekanje da se završi izvršavanje funkcija niti (thread join), izlaz iz funkcije niti, muteks objekti (katanci), uslovne promenljive, čitaj-piši zaključavanje i semafori.

Helgrind može da detektuje naredne tri klase grešaka:

1. Lošu upotrebu API-ja za programiranje POSIX niti.
2. Potencijalne mrtve petlje koje nastaju iz lošeg redosleda zaključavanja i otključavanja promenljivih.
3. Pristup memoriji bez adekvatnog zaključavanja ili sinhronizacije.

Problemi poput ovih često kao rezultat imaju nereproduktivne, vremenski zavisne padove, mrtve petlje i druga loša ponašanja programa koja se mogu teško otkriti drugim sredstvima. Helgrind je svestan svih abstrakcija niti i prati njihove efekte što preciznije može. On najbolje radi ukoliko program koristi isključivo API POSIX niti (možemo naravno

koristiti i druge standarde za niti ali moramo opisati Helgrindu njihovo ponašanje korišćenjem `ANNOTATE_*` makroa definisanih u `helgrind.h` zaglavlju).

### 2.3.1 Loša upotreba API-ja za programiranje POSIX niti

Helgrind presreće pozive mnogih funkcija POSIX biblioteke `pthread.h` i zbog toga je u mogućnosti da izveštava o raznovrsnim problemima. Prisutvo ovakvih grešaka može dovesti do nedefinisanog ponašanja programa i njihovog teškog kasnijeg pronalaženja. Greške koje Helgrind može da detektuje su: otključavanje nevalidnog muteksa, otljučavanje ne zaključanog muteksa, otključavanje muteksa koga drži druga nit, uništavanje nevalidnog ili zaključanog muteksa, rekurzivno zaključavanje nerekurzivnih muteksa, dealociranje memorije koja sadrži zaključani muteks, prosledjivanje muteks argumenata funkcijama koje očekuju čitaj-piši katanace kao argumente i obrnuto, slučajevi kada se funkcije neuspešno izvrše i imaju kod greške koji se mora obraditi, slučajevi kada nit izadje dok još drži zaključan katanac, pozivanje `pthread_cond_wait` sa nezaključanim muteksom, nevalidnim muteksom ili sa muteksom koji je zaključan drugom niti, nekonzistentno vezivanje između uslovnih promenljivih i njihovih odgovarajućih muteksa, nevažeća ili dupla inicijalizacija `pthread_barrier`, uništavanje `pthread_barrier` koji nikad nije inicijalizovan ili koga niti čekaju, čekanje na `pthread_barrier` objekat koji nikad nije inicijalizovan, za sve `pthread` funkcije koje Helgrind presreće, generiše se podatak o grešci ako funkcija vrati kod greške, iako Helgrind nije našao grešku u kodu. Sve provere koje se odnose na mutekse se mogu primeniti i na katanace čitače-pisače.

Na slici 5 prijavljena greška prikazuje i primarno stanje steka koje pokazuje gde je detektovana greška. On može takodje i da sadrži pomoćno stanje steka koje nam pruža neke dodatne informacije. Ukoliko je moguće ispisuje i broj linije u samom kodu gde se greška nalazi. Ukoliko se greška odnosi na muteks, kao što je to ovde slučaj, Helgrind će detektovati i gde je prvi put uočen problematični muteks.

```
Thread #1 unlocked a not-locked lock at 0x7FEFFFA90
  at 0x4C2408D: pthread_mutex_unlock (hg_intercepts.c:492)
  by 0x40073A: nearly_main (tc09_bad_unlock.c:27)
  by 0x40079B: main (tc09_bad_unlock.c:50)
Lock at 0x7FEFFFA90 was first observed
  at 0x4C25D01: pthread_mutex_init (hg_intercepts.c:326)
  by 0x40071F: nearly_main (tc09_bad_unlock.c:23)
  by 0x40079B: main (tc09_bad_unlock.c:50)
```

Slika 5: Primer prikaza greške u programu

### 2.3.2 Loš redosled zaključavanja

Helgrind nadgleda redosled kojim niti zaključavaju promenljive. Time omogućava detekciju potencijalnih mrtvih petlji koje mogu nastati iz redosleda kojim se vrši zaključavanje. Detektovanje takvih nekonzistentnosti je korisno zato što neke mrtve petlje neće biti otkrivene tokom testiranja i mogu kasnije dovesti do padova programa koje se teško otklanjaju.

Recimo prost primer takvog problema je sledeći:

- Zamislimo neki deljeni resurs R, koji je iz bilo kog razloga, je zaključan sa dva katanca, L1 i L2, koja oba moraju držati resurs R kada mu se pristupa.
- Pretpostavimo da nit dobija katanac L1, zatim L2 i nastavlja sa pristupom resursa R. Posledica ovoga je da sve niti u programu moraju dobiti dva katanca i to u redosledu prvo L1, a zatim L2. Ako to nije slučaj rizikujemo nastanak mrtve petlje.
- Mrtva petlja se može desiti ako dve niti, T1 i T2, obe žele da pristupe R. Pretpostavimo da T1 dobija L1, T2 dobija L2 prvo, a zatim žele da dobiju ova druga dva katanca. Problem je što su ti katanca već zauzeti tako da T1 i T2 ulaze u mrtvu petlju.

Helgrind kreira graf koji sadrži sve katanca koje je zaključao u prošlosti. Kada nit zaključa novi katanac, graf se ažurira i zatim proverava da li sad sadrži ciklus. Prisustvo ciklusa upućuje na potencijalnu mrtvu petlju u koju će biti uključeni katanca iz ciklusa. Helgrind će odabrati dva katanca iz ciklusa i prikazće nam kako je njihov redosled zaključavanja doveo do nekonzistentnosti. Helgrinda nam prikazuje tačke u programu u kome je prvi put definisan redosled i tačke u programu koje ga kasnije narušavaju.

### 2.3.3 Pristup memoriji bez adekvatnog zaključavanja ili sinhronizacije

Ovaj problem koji se još naziva i trka za podacima se javlja kada dve niti pristupaju deljenoj memoriji bez korišćenja odgovarajućih katanaca ili drugih vidova sinhronizacije koje nam osiguravaju pristup samo jedne niti. Takva propuštena zaključavanja mogu dovesti do nejasnih, vremenski zavisnih bagova.

Na slici 6 je prikazan pristup promenljivoj bez adekvatne sinhronizacije. Problem je što i nit roditelj i nit dete mogu istovremeno pristupiti deljenoj promenljivoj *var* i promeniti joj vrednost.

Analizom ovog programa Helgrind nama šalje izveštaj kao na slici 7. U izveštaju možemo tačno da vidimo koje niti pristupaju promenljivoj bez sinhronizacije, gde se vrši sam pristup promenljivoj, ime i veličinu promenljive kojoj niti pristupaju.

## 2.4 DRD - Data Race Detector

DRD je Velgrindov alat za detekciju grešaka u C i C++ programima koji koriste više niti. Da bi koristili ovaj alat neophodno je navesti *tool=drd* u Velgrindovoj komandnoj liniji. Alat radi za svaki program koji koristi niti POSIX standarda ili koji koriste koncepte koji su nadogradjeni na ovaj standard. Karakteristike POSIX niti smo objasnili u poglavlju Helgrind. Problemi koji se mogu javiti prilikom korišćenja POSIX niti u programu su:



```

#include <pthread.h>

int var = 0;

void* child_fn ( void* arg ) {
    var++; /* Unprotected relative to parent */ /* this is line 6
*/
    return NULL;
}

int main ( void ) {
    pthread_t child;
    pthread_create(&child, NULL, child_fn, NULL);
    var++; /* Unprotected relative to child */ /* this is line 13
*/
    pthread_join(child, NULL);
    return 0;
}

```

Slika 6: Primer pristupa memoriji bez adekvatne sinhronizacije

```

Thread #1 is the program's root thread

Thread #2 was created
  at 0x511C08E: clone (in /lib64/libc-2.8.so)
  by 0x4E33A4: do_clone (in /lib64/libpthread-2.8.so)
  by 0x4E33A30: pthread_create@GLIBC_2.2.5 (in
/lib64/libpthread-2.8.so)
  by 0x4C299D4: pthread_create@ (hg_intercepts.c:214)
  by 0x400605: main (simple_race.c:12)

Possible data race during read of size 4 at 0x601038 by thread
#1
  at 0x400606: main (simple_race.c:13)
This conflicts with a previous write of size 4 by thread #2
  at 0x4005DC: child_fn (simple_race.c:6)
  by 0x4C29AFF: mythread_wrapper (hg_intercepts.c:194)
  by 0x4E3403F: start_thread (in /lib64/libpthread-2.8.so)
  by 0x511C0CC: clone (in /lib64/libc-2.8.so)
Location 0x601038 is 0 bytes inside global var "var"
declared at simple_race.c:3

```

Slika 7: Izveštaj Helgrinda za pristup promenljivoj bez sinhronizacije

- Trka za podatke (Data races). Jedna ili više niti pristupa istoj me-

Problem koji se detektuje	Helgrind	DRD
Mrtve petlje	DA	NE
Trka za podacima	DA	DA
Nepravilno korišćenje API-ja	DA	DA
Zadržavanje podataka	NE	DA

Tabela 1: Poredjenje Helgrinda i DRD-a na osnovu problema koje uočavaju

moriskoj lokaciji bez odgovarajućeg zaključavanja. Ovi problemi su programerske greške i uzrok su bagova koji se teško pronalaze.

- Zadržavanje katanaca (Lock contention). Jedna nit blokira progres drugih niti zadržavajući katanac predugo.
- Nepravilno korišćenje API-ja POSIX niti. Mnoge implementacije API-ja su optimizovane radi bržeg vremena izvršavanja. Takve implementacije se neće buniti na određene greške (recimo kada muteks otključa neka druga nit a ne ona koja ga je zaključala).
- Mrtve petlje. Javalja se kada dve niti čekaju jedna na drugu neodređeno vreme.
- Lažno deljenje. Ako niti koje se izvršavaju na različitim jezgrima procesora pristupaju različitim promenljivama koje su locirane u istim keš linijama često, za posledicu može imati usporenje tih niti zbog razmenjivanja keš linija.

DRD je u stanju da detektuje prva tri od navedene liste problema (trka za podatke, zadržavanje katanca, nepravilno korišćenje API-ja POSIX niti). Vidimo da ovaj alat izvršava analizu i detektuje slične probleme kao i alat Helgrind. Pitanje je onda zašto u sklopu Velgrinda postoje ova dva alata. Odgovor je da Helgrind proizvodi izlaze koji su lakši za interpretaciju dok DRD ima bolje performanse. Takođe Helgrind može detektovati problem mrtvih petlji, a nije u stanju da detektuje problem zadržavanja katanaca dok je kod DRD-a slučaj obrnut kao što se može videti u tabeli 1.

## 2.5 Massif

Massif je alat za analizu hip memorije korisničkog programa. Obuhvata, kako memoriju kojoj korisnik može da pristupi, tako i memoriju koja se koristi za pomoćne koncepte kao što su book-keeping bajtovi i prostor za poravnanje. Da bi mogli koristiti ovaj alat neophodno je navesti `-tool = massif` u Velgrindovoj komandnoj liniji. Takođe može da se prilagodi da meri veličinu programskog steka.

Analiza programskog hipa, na modernim računarima koji koriste virtualnu memoriju, donosi prednosti u vidu ubrzavanja programa, jer manji programi imaju bolju iskorišćenost keša i izbegavaju straničenje. Kod programa koji zahtevaju veliki količinu memorije, dobra iskoriscenost hipa smanjuje šansu za izglednjivanje *prostora za razmenu - swap space* korisničke mašine.

Takođe postoje propusti u iskorišćenosti memorije koji ne spadaju u klasične probleme curenja memorije, takve propuste ne mogu detektovati alati kao što su Memcheck, kada pokazivač na deo memorije postoji, ali se on ne koristi. U tom slučaju, programi tokom vremena mogu nenužno povećati memoriju koju koriste, Massif može detektovati ovakve situacije. Najvažnije,

Massif ne govori samo o tome koliko korisnički program memorije zauzima, već daje sliku o tome koji delovi programa su zaduženi za alociranje hip memorije.

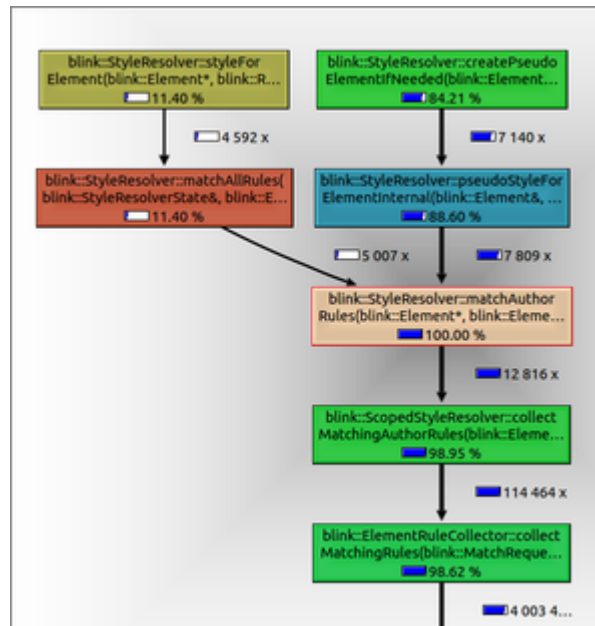
## 2.6 Callgrind

Callgrind je alat koji generiše listu poziva funkcija korisničkog programa, u vidu grafa. U osnovnom podešavanju sakupljeni podaci sastoje se od broja izvršenih instrukcija, njihov odnos sa linijom u izvornom kodu, odnos pozivaoc/pozvan između funkcija, i broj takvih poziva. Dodatna podešavanja omogućavaju analiziranje koda tokom izvršavanja. Da bi mogli koristiti ovaj alat neophodno je navesti `-tool = callgrind` u Velgrindovoj komandnoj liniji.

Podržane komande su:

`callgrind_annotate` - Na osnovu generisanog profila, koji je sačuvan kao fajl nakon zaustavljanja programa, prikazuje listu funkcija. Za grafičku vizuelizaciju preporučuju se dodatni alati (KCAshegrind), koji olakšava navigaciju ukoliko Callgrind napravi veliku količinu podataka.

`callgrind_control` - Ova komanda omogućava interaktivnu kontrolu i nadgledanje programa prilikom izvršavanja. Mogu se dobiti informacije o stanju na steku, može se takođe u svakom trenutku generisati profil.



Slika 8: Callgrind

### 3 Zaključak

U ovom radu opisana je primena alata Valgrind u procesima analize i otklanjanja grešaka koje se javljaju prilikom razvijanja programske podrške na modernim računarima. Opisana je struktura problema koje detektuju alati Memcheck, Cachegrind, Helgrind, DRD, Massif i Callgrind. Navedeni su testovi i njihovi rezultati kojima su prikazani principi rada navedenih alata.

Programi koji se izvršavaju posredstvom Valgrinda bivaju od 20 do 100 puta sporiji, tako da treba ovo uzeti u obzir prilikom analize problema. Korišćenjem Valgrinda se može uštedeti dragoceno vreme, kao i popraviti kvalitet finalnog proizvoda.

### Literatura

- [1] *Valgrind manual*, [valgrind.org/docs/manual/manual.html](http://valgrind.org/docs/manual/manual.html), 2015
- [2] *Прилагођавање алата за динамичку анализу програмског кода Велгринд за архитектуру МИПС*, Дејан Јевтић [http://www.rt-rk.uns.ac.rs/sites/default/files/e57\\_2010-dejan\\_jevtic.pdf](http://www.rt-rk.uns.ac.rs/sites/default/files/e57_2010-dejan_jevtic.pdf), 2011