

Analiza projekta koriscenjem alata za verifikaciju softvera

Seminarski rad u okviru kursa
Verifikacija softvera
Matematički fakultet sara.kapetinic.sk@gmail.com

Sara Kapetinić

Avgust 2023.

Sažetak

Verifikacija softvera obuhvata proces evaluacije sistema ili komponente kako bi se utvrdilo da li proizvodi trenutne faze razvoja zadovoljavaju uslove postavljene na pocetku date faze. Ova analiza softvera obuhvata skup metoda, alata i procesa za utvrđivanje ispravnosti softvera. Tehnike verifikacije se dele na dinamicke i staticke. Dinamicke obuhvataju ispitivanje koda tokom njegovog izrsavanja, dok su staticke osvrnute na izvorni kod.

Sadržaj

1	Uvod	2
2	Valgrind	2
2.1	Callgrind	2
2.1.1	KCacheGrind	3
2.2	Memcheck	5
2.3	Cachegrind	6
2.4	Massif	8
3	Coverage tools	8
3.1	Koriscenje GCov alata	9
4	Clandg i CppCheck	9
5	Zakljucak	12

1 Uvod

Projekat *Tudu* predstavlja jednostavni projekat u svrhu organizacije vremena. *Tudu* je nedeljni drag & drop planer, sa integrisanom listom obaveza (todo listom). Uz pomoć ove aplikacije možete organizovati svoje planove na pregledan i sistematizovan način. U listu obaveza se dodaju sve obaveze koje planirate da obavite u narednom periodu i zatim im se dodeljuje prioritet, tako da imate sistematizovan spisak obaveza koji omogućava da nikad ne zaboravite na bitan sastanak, porodični ručak ili neku drugu obavezu. Iz liste obaveze možete premestiti u nedeljni kalendar i tako na pregledan način u svakom trenutku videti šta vas očekuje ove, ili bilo koje naredne nedelje.

2 Valgrind

Valgrind je GPL sistem za debugovanje i profajliranje Linux programa. Sa Valgrindovim paetom alata mozete detektovati mnoge probleme kod upravljanja memorije ili rada sa nitima. Takodje mozete izvrstiti detaljno profilisanje da biste ubrzali svoje programe. Prednosti Valgrinda:

- Valgrind je besplatan.
- Mozete se koristiti na razlicitim Linux platformama kao sto su: x86/Linux, AMD64/Linux PPC32/Linux.
- Radi na svim glavnim Linux distribucijama.
- Moze pomoci u ubrzanju programa.
- Znacajno skracuje vreme debugovanja.
- Lak je zakorisćenje. Nije potrebno rekompajliranje programa vec samo pokretanje sa prefiksom valgrind.

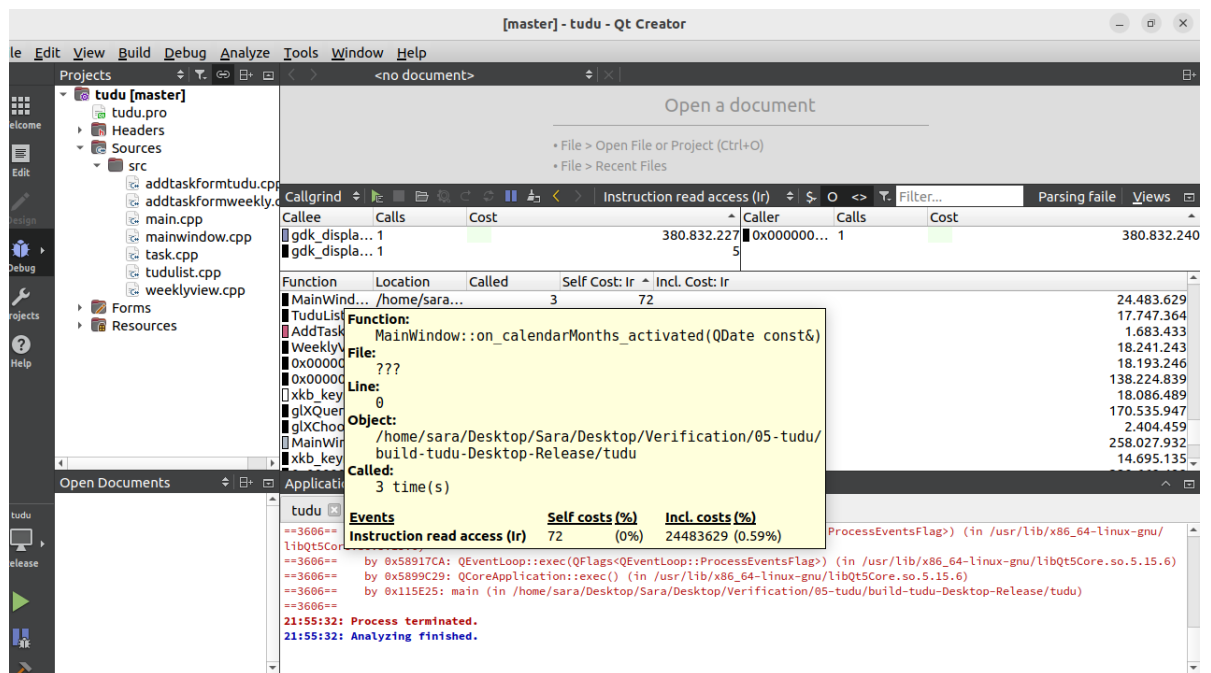
Neki od Valgrindovih alata koji su korisceni na *Tudu* projektu su Callgrind, Memcheck i Massif. U narednom delu reci cemo nesto vise o njima i deonstrirati njihovu upotrebu.

2.1 Callgrind

Callgrind je alat zaz profilisanje koji belezi istoriju poziva medju funkcijama u programu koji se izvrsava kao graf poziva. Prikupljeni podaci se sastoje od: broja izvršenih instrukcija, njihovog odnosa sa izvornim linijama koda, odnosa pozivalac/pozvani medju funkcijama i broja takvih poziva.

Posto je *Tudu* projekat radjen u Qt-u koji ima dosta ugradjenih Valgrind alata, simuliracemo prvo koriscenje Callgrind-a kroz Qt.

Postrebno je u Qt Creator-u uci u Debug mode i namestiti Callgrind. Zatim cemo pokrenuti projekat sa Callgrindom i testirati sto vise funkcionalnosti. Kao sto mozemo videti na slici 1 dobicemo izlistane sve funkcije kroz koje smo prosli i koje su se pozivale tokom naseg testiranje programa. Ukoliko stanemo na neku funkciju, mozemo detaljnije pogledati informacije o njoj.



Slika 1: Callgrind Qt Creator

Drugi nacin pokretanja sa Callgrindom je iz terminala. Na ovaj nacin mozemo generisati callgrind izlazni fajl. Ovaj fajl nije preterano citljiv, ali se koristi kao ulaz za neke druge alate.

Komanda pomocu koje dobijamo callgrind izlazni fajl:

```
$ valgrind --tool=callgrind --callgrind-out-file=callgrind.out ./tudu
```

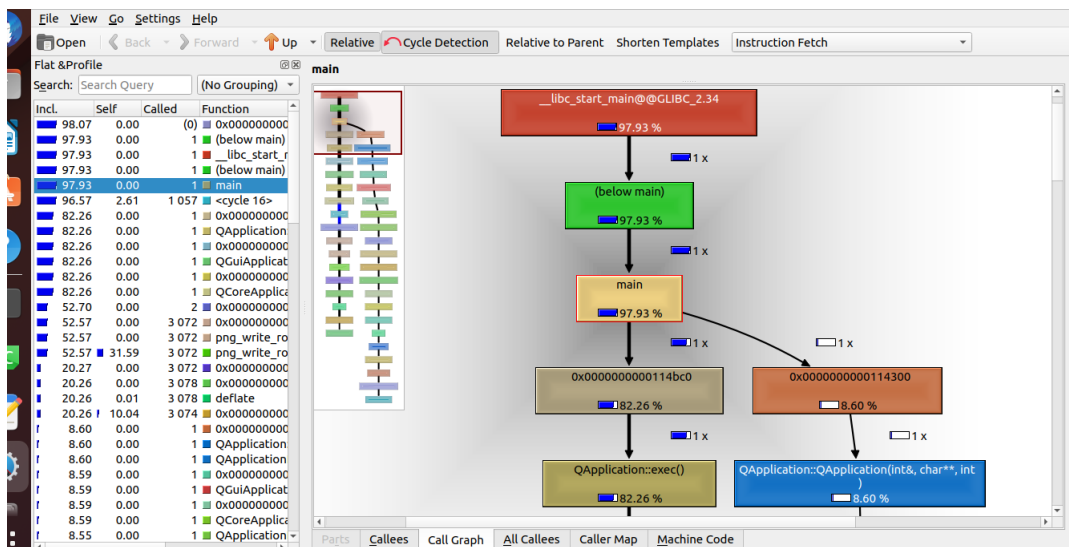
Pomocu ovog alata mozemo videti i broj pozivanja sistemskih funkcija, cime dobijamo sliku i o njihovom koriscenju. Ovaj alat dozvoljava pametnu manipulaciju kodom jer nam daje informacije o broj poziva metoda. Prema tome cesto pozivane metode treba da budu sto efikasnije. Izvršavanje programa zajedno sa callgrind-om je znatno sporije nego bez njega.

2.1.1 KCacheGrind

Kcachegrind je alat za vizuelizaciju podataka profila, koja se koristi za odredjivanje delova koji oduzimaju najvise vremena pri izvršavanju programa.

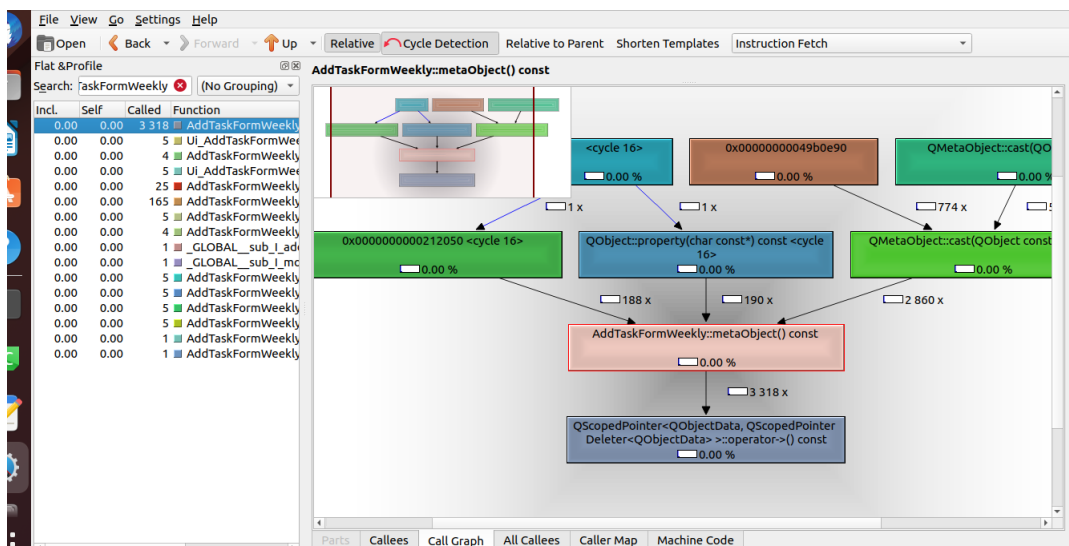
Komanda za pokretanje kcachegrind alata, kao ulaz se prosledjuje izlazni fajl dobijen koriscenjem callgrind-a:

```
$ kcachegrind callgrind.out
```



Slika 2: KCachegrind

Na slici 2 mozemo videti graf poziva funkcija na nasem *Tudu* projektu. Procenti na grafu poziva oznacavaju procenat od ukupnog utroska resursa(u vecini slucajeva procesorsko vreme), dok brojevi na strelicama se odnose na broj puta koliko metoda pozivalac poziva drugu metodu.



Slika 3: KCachegrind Add new weekly task

Na slici 3 je prikazan graf poziva pri dodavanju neke nedeljne obaveze u tudu listu. Mozemo uociti da razlicite funkcije dosta puta pozivaju klasu *AddTaskFromWeekly*.

Sa leve strane u KCachegrind-u nam se nalazi stack sa svim pozivima funkcija kroz koji se mozemo kretati. Ovaj alat se koristi i u kombinaciji

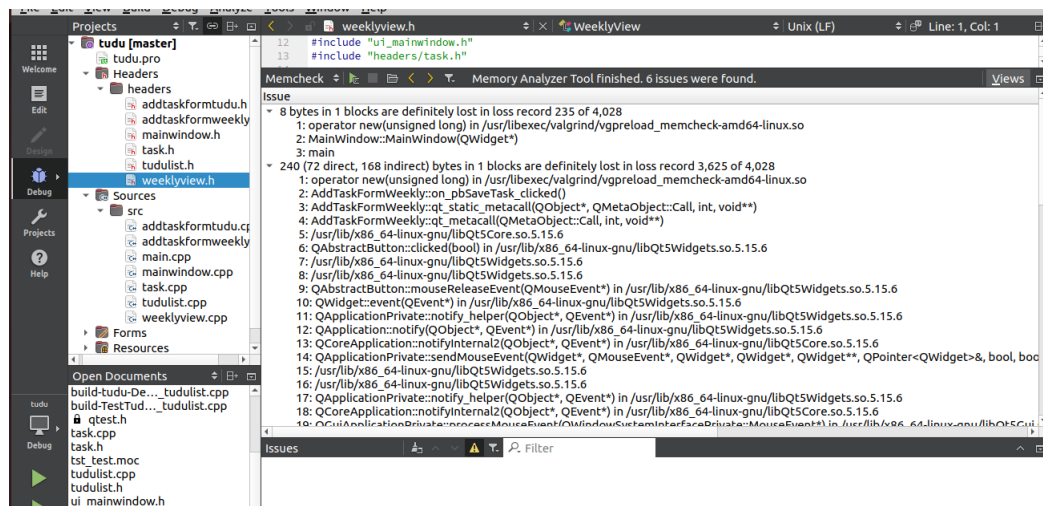
sa nekim drugim valgrind alatima sto cemo videti u nastavku.

2.2 Memcheck

Meecheck je Valgrind-ov alat koji služi za detektovanje gresaka vezanih za memoriju. Koristi se za programe pisane u C i C++ jezicima. Neka od svojstava koje Memcheck može detektovati:

- Prisupanje nedozovljenoj memoriji
- Koriscenje nedefinisanih vrednosti
- Nekorektno oslobadjanje hip memorije
- Curenje memorije

Kao i Callgrind, Memcheck nije tezak za koriscenje. U okviru Qt Creator-a se nalazi podrška za Memcheck. Na slici 4 mozemo videti koriscenje ovog alata kroz Qt na projektu Tudu. Pri prvom pokretanju memcheck-a iz Qt Creatora nisu se pojavile nikakve greske sa memorijom. S obzirom da je ovo mali projekat, to bi moglo biti ocekivano, medjutim prisledecem koriscenju memchecka dobili smo drugacije rezultate. Dosta problema sa memorijom se pojavilo sto mozemo videti na slici 4.



Slika 4: Memcheck in Qt

Memcheck se može koristiti i iz terminala, pokretanjem komande:

```
$ valgrind --leak-check=full --leak-resolution=high --show-leak-kinds=all  
--xtree-leak=yes ./tudu
```

Flag-ovi koji su korišćeni u prethodnoj komandi označavaju:

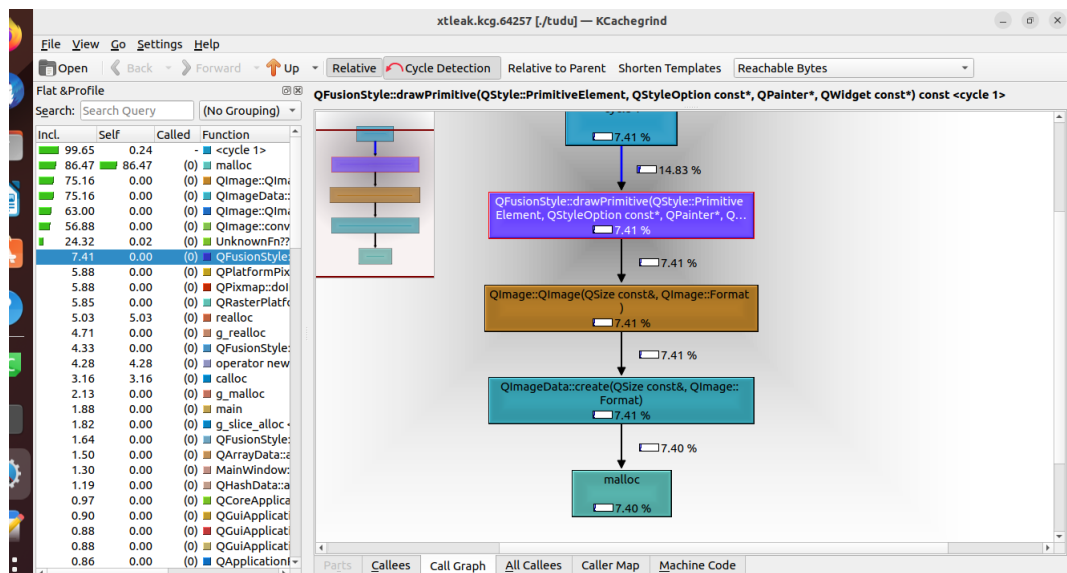
- `--leak-check=full` svako curenje memorije će biti zabeleženo i bice uracunato u greske
- `--leak-resolution` svi unosi moraju da se poklapaju
- `--show-leak-kinds=all` i `--xtree-leak=yes` za dobijanje izlaznog fajla u callgrind formatu

```

==87690== HEAP SUMMARY:
==87690==    in use at exit: 37,529,148 bytes in 58,657 blocks
==87690== total heap usage: 701,675 allocs, 643,018 frees, 175,025,006 bytes allocated
==87690==
==87690== xtree leak report: /home/sara/Desktop/Sara/Desktop/Verification/05-tudu/build-tudu-Desktop-Debug/xtleak.kcg.87690
==87690== LEAK SUMMARY:
==87690==    definitely lost: 920 bytes in 11 blocks
==87690==    indirectly lost: 3,348 bytes in 41 blocks
==87690==    possibly lost: 86,040 bytes in 741 blocks
==87690==    still reachable: 37,363,176 bytes in 57,171 blocks
==87690==                of which reachable via heuristic:
==87690==                  newarray      : 24 bytes in 1 blocks
==87690==                  multipleinheritance: 10,472 bytes in 13 blocks
==87690==    suppressed: 0 bytes in 0 blocks
==87690==
==87690== For lists of detected and suppressed errors, rerun with: -s
==87690== ERROR SUMMARY: 641 errors from 641 contexts (suppressed: 0 from 0)

```

Slika 5: Memcheck output file



Slika 6: Memcheck in KCachegrind

Pokretanjem ove komande dobili smo izlazni fajl koji sadrži izveštaj o memoriji, kao i drugi izvršni fajl koji se može otvoriti pomoću kcache-grind-a.

Na slici 5 nalazi se izveštaj o korišćenju hip memorije, dok na slici 6 možemo videti izgenerisan callgrind fajl u kcache-grind-u. Prikazana je Call Graph opcija gde se može videti procentualno alokiranje memorije određenih funkcija.

2.3 Cachegrind

Cachegrind je alat Valgrind-a koji omogućava softversko profajliranje kes memorije tako što simulira i prati pristup kes memoriji masine na kojoj se program, koji se analizira, izvršava. Takođe, može se koristiti i za profajliranje izvršavanja grana. Cachegrind simulira memoriju masine,

koja ima prvi nivo kes L1 memorije podeljene u dve odvojene nezavisne sekcije: I1 - sekcija kes memorije u koju se smestaju instrukcije D1 - sekcija kes memorije u koju se smestaju podaci. Drugi nivo kes memorije koju Cachegrind simulira je objedinjen - LL. Ovaj nacin konfiguracije odgovara mnogim modernim masinama. Vazno je da u okviru makefile-a ne posledimo kompajleru g++ -O0 jer zelimo da testiramo program u njegovom normalnom izvršavanju.

Komanda za pokretanje ovog alata:

```
$ valgrind --tool=cachegrind --cachegrind-out-file=cachegrind.out
./tudu
```

Ovom komandom smo generisali detaljniji izlazni fajl koji mozemo pomocu cg_annotate ispisati. Na slici 7 mozemo videti osobine kesa koje se nalaze na pocetku generisanog fajla. Prvi broj predstavlja velicinu kesa, drugi broj predstavlja velicinu linije dok treci predstavlja asocijativnost kesa. Prilikom pozivanja alata Cachegrind moguće je da promenimo velicinu LL kesa. Na slici 8 nalazi se stanje kesa tokom izvršavanja Tudu projekta.

```
-----
I1 cache:      32768 B, 64 B, 8-way associative
D1 cache:      32768 B, 64 B, 8-way associative
LL cache:      3145728 B, 64 B, 12-way associative
Command:       ./tudu
Data file:     cachegrind.out
Events recorded: Ir I1mr I1Lmr Dr D1mr D1Lmr Dw D1mw D1Lmw
Events shown:   Ir I1mr I1Lmr Dr D1mr D1Lmr Dw D1mw D1Lmw
Event sort order: Ir I1mr I1Lmr Dr D1mr D1Lmr Dw D1mw D1Lmw
Thresholds:     0.1 100 100 100 100 100 100 100 100
Include dirs:
User annotated:
Auto-annotation: on
-----
```

Slika 7: Cachegrind information about cache

```
== by 0x103FE: Math (Math.cpp:10)
== I refs:      1,239,641,233
== I1 misses:   10,697,404
== LLi misses:  297,466
== I1 miss rate: 0.86%
== LLi miss rate: 0.02%
==
== D refs:      449,700,528 (304,107,527 rd + 145,593,001 wr)
== D1 misses:   14,308,745 ( 8,368,898 rd + 5,939,847 wr)
== Lld misses:   2,627,958 ( 582,806 rd + 2,045,152 wr)
== D1 miss rate: 3.2% ( 2.8% + 4.1% )
== Lld miss rate: 0.6% ( 0.2% + 1.4% )
==
== LL refs:      25,006,149 ( 19,066,302 rd + 5,939,847 wr)
== LL misses:    2,925,424 ( 880,272 rd + 2,045,152 wr)
== LL miss rate: 0.2% ( 0.1% + 1.4% )
a-Inspiron-15-3567: ~/Desktop/Sara/Desktop/Verification/05-tudu/build-tudu-Desktop-Debu
```

Slika 8: Cachegrind

2.4 Massif

Massif je hip profajler. Meri koliko hip memorije program koristi. Ovo podraumeva i koriscenu memoriju i onu ekstra alociranu u svrhe poravnjanja. Takodje moze da meri velicinu steka programa. Profiliranje hipa moze da pomogne u redukciji memorije koju program koristi. Na modernim masinama sa virtuelnom memorijom, ovo moze doprineti:

- Vecoj brzini izvršavanja programa.
- Ako program koristi dosta memorije, smanjuje se sansa da se iscrpi sva memorija masine.

Massif se koristi lako kroz terminal, pokretanjem komande:

```
$valgrind --tool=massif ./tudu
```

Ovako dobijamo fajl massif.out <PID>, koji se moze citati pomocu ms_print komande. U ovom fajlu se nalaze detaljne informacije o koriscenju hip-a, kao sto je prikazano na slici 9. U sklopu massif fajla se nalazi i graf koji predstavlja memorijsku potrošnju. Tabela koja je prikazana na slici sastoji se od broja snepshota, vremena koje je potroseno, totalne memorijske potrošnje, broja koriscenih i dodatnih hip bajtova kao i velicinu steka.

Number of snapshots: 54
Detailed snapshots: [9, 10, 11, 13, 14, 22, 33, 34, 35, 37, 38, 44, 46, 49 (peak)]

n	time(i)	total(B)	useful-heap(B)	extra-heap(B)	stacks(B)
0	0	0	0	0	0
1	19,410,074	521,056	469,508	51,548	0
2	50,837,170	523,592	471,972	51,620	0
3	88,216,689	995,192	907,763	87,429	0
4	118,043,336	906,264	813,396	92,868	0
5	141,713,518	950,640	856,890	93,750	0
6	159,638,560	988,432	894,418	94,014	0
7	188,005,229	995,480	901,427	94,053	0
8	211,893,989	1,138,816	1,001,488	137,328	0
9	238,467,124	1,807,560	1,588,714	218,846	0

87.89% (1,588,714B) (heap allocation functions) malloc/new/new[], --alloc-fns, etc.

Slika 9: Massif

3 Coverage tools

GCov je alat koji se koristi zajedno sa GCC-om da bi se dobila pokrivenost koda u programu. Ovaj alat pomaze u pisanju efikasnije i brzeg koda, kao i u pronalazenju nedostiznih delova programa. Moze da se koristi kao alat za profilisanje, cime dobijamo ideju gde ce optimizacije koda biti ucinkovitije. Osnovne informacije koje dobijamo koriscenjem GCov alata:

- Koliko cesto se svaka linija koda izvršava
- Koje linije se zapravo izvršavaju
- Koiko racunarskog vremena koristi koji deo koda

3.1 Koriscenje GCov alata

Alat GCov primenicemo na Tudu projektu. Koristicemo alat iz terminala pomocu naredne komande

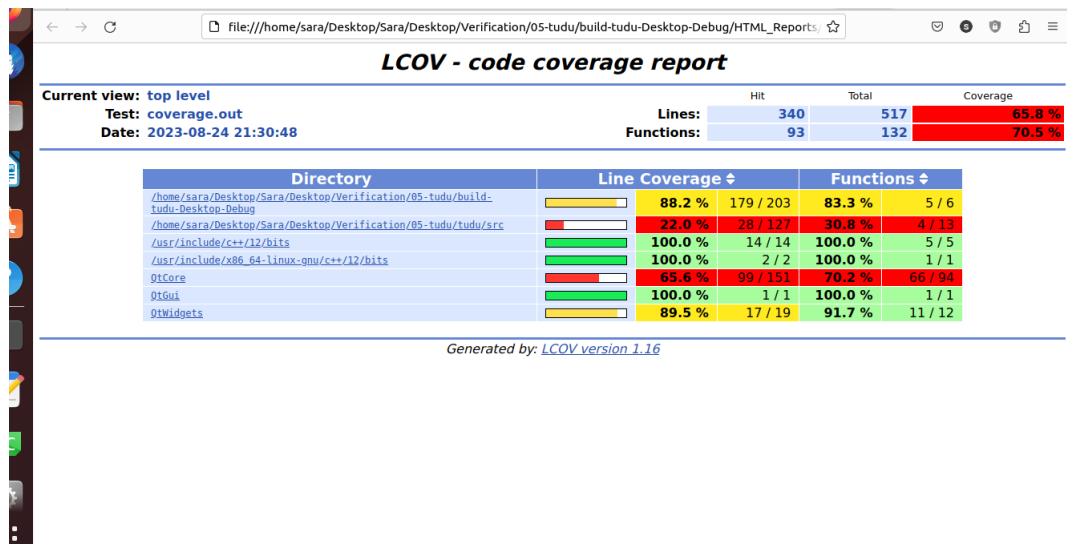
```
$lcov --rc lcov-branch-coverage=1 -c -d . -o coverage.out
```

Generisali smo .gcda fajlove kao i coverage.out fajl. Da bi imali lepši vizuelni prikaz podataka koristicemo genhtml komandu, kojom cemo generisati html stranicu sa svim podacima koje smo dobili GCov-om.

Komanda za generisanje html stranice na osnovu .out fajla:

```
$ genhtml --rc lcov-branch-coverage=1 -o HTML_Reports coverage.out
```

Svi potrebni podaci smesteni su u direktorijum HTML_Reports, odakle mozemo otvoriti html stranicu i analizirati dobijene informacije.

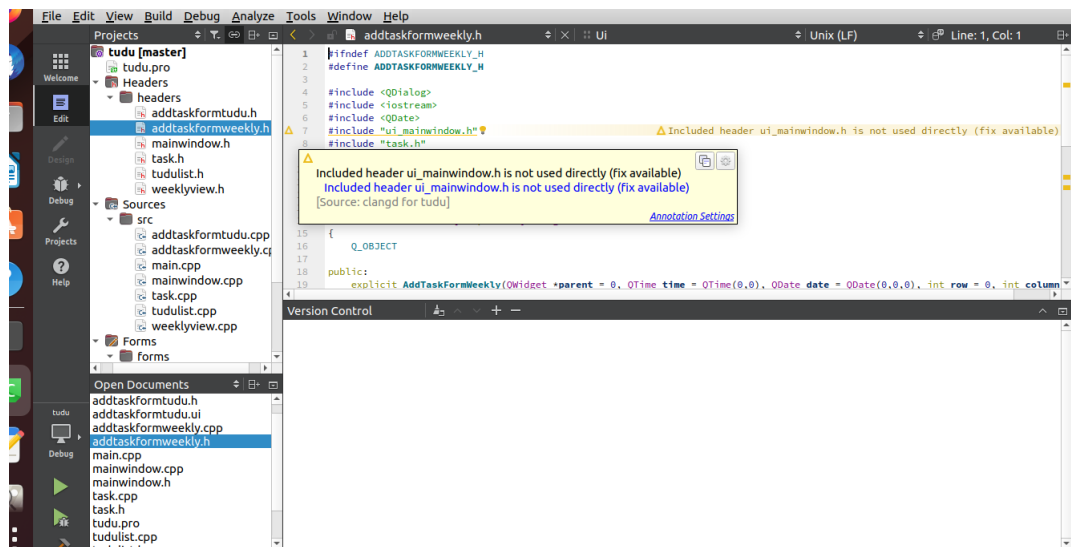


Slika 10: GCov alat primenjen na Tudu projektu

Na slici 10 mozemo videti neke dodatne informacije. Posto je Tudu jednostavan projekat, kroz prolazjenje nekih osnovnih funkcija, uspele smo da predjemo dve trecine linija koda i funkcija. U tabeli nam je oznaceno u kojim delovima se nalaze funkcije koje nisu pokrivene, ili linije koda koje nisu dostignute.

4 Clandg i CppCheck

Clandg je jezicki server koji postoji ugradjen u mnogim razvojnim okruzenjima. Zasnovan je na Clang C++ kompilatoru i deo je LLVM projekta. Koristi se za proveru ispravnosti koda: ispituje kompletnost, nalazi kompilacione greske.. Clandg mozemo koristiti kroz Qt uz pomoc sledecih koraka: Tools -> Options -> C++ -> Clandg. Koriscenjem clangd-a na projektu Tudu, dobili upozorenje prikazano na slici 11. Linije koja predstavlja upozorenje je zuta, sa zutim trouglom. Kada stanemo na trougao mozemo videti detaljniji opis problema, kao i to da nam je clangd prijavio upozorenje.

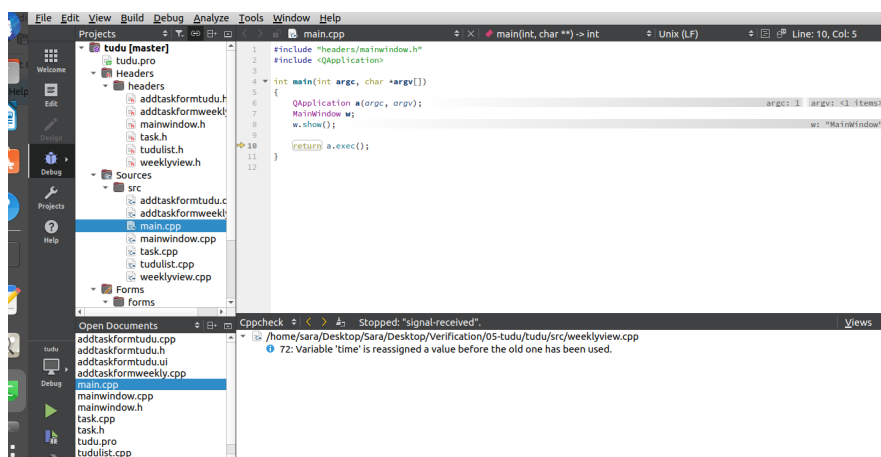


Slika 11: Clangd primenjen na Tudu projektu

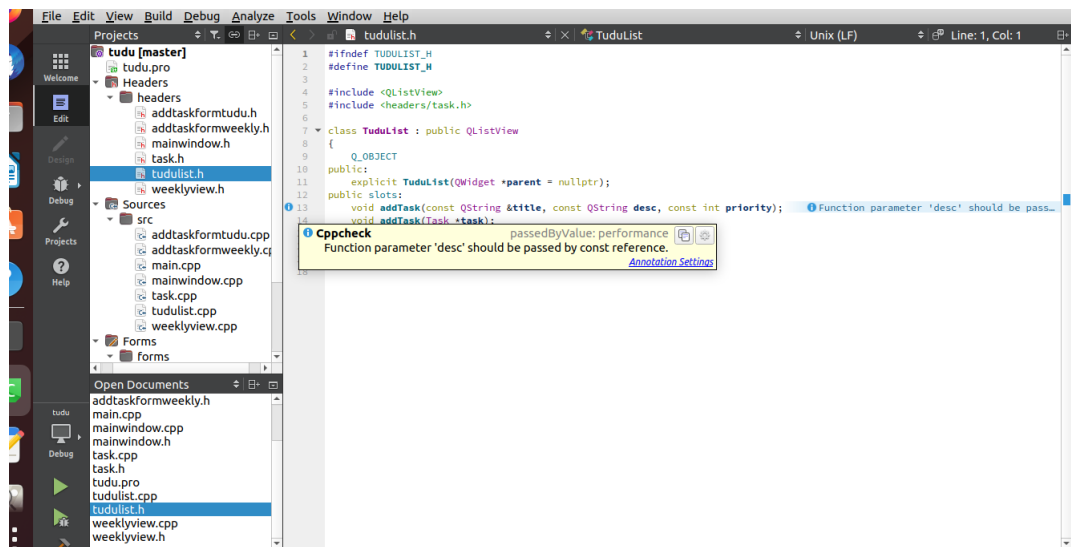
CppCheck se koristi za statičku analizu C/C++ programa. Daje nam jedinstvenu analizu koda za detektovanje bagova i fokusira se na pronalaženje nedefinisanog ponasanja i opasnih struktura. Primeri nedefinisanog ponasanja:

- Deljenje nulom.
- Neinicijalizovane varijable.
- Nevalidne konverzije.
- Nekorisceni pokazivaci.

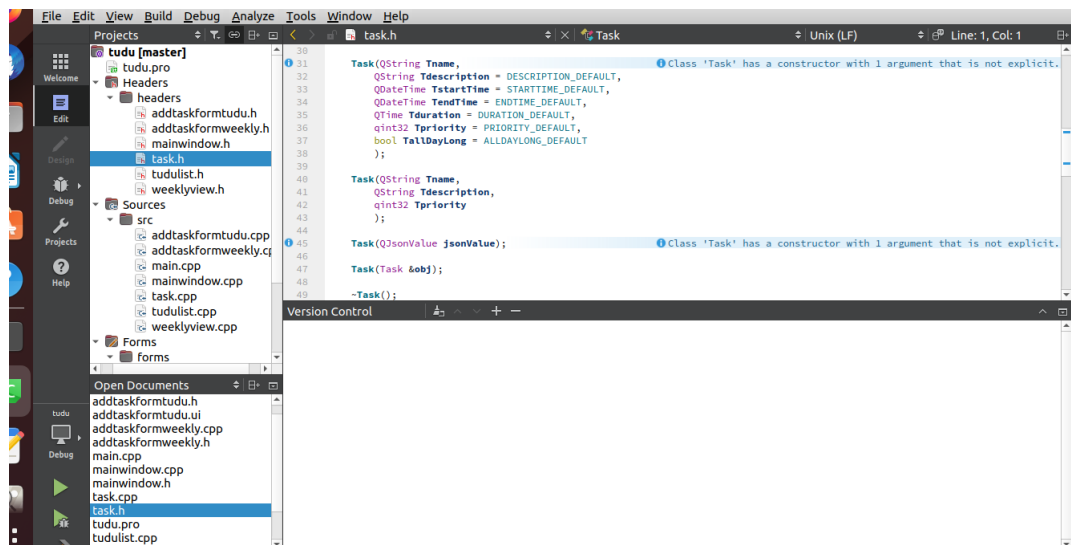
Podrška za korišćenje CppCheck alata postoji u okiru Qt-a, a sad ćemo kroz Tudu projekat prikazati neke od navedenih gresaka koje je CppCheck nasao.



Slika 12: CppCheck i korišćenje promenljivih



Slika 13: CppCheck i const parametri



Slika 14: CppCheck i konstruktori

Na prethodnim slikama mogli smo videti neke od predloga izmena koje nam je CppCheck predlozio. Deo koda na kome je predstavljen predlog za promenljivu *time* se nalazi ispod. Promenljiva je inicijalizovana, ali joj se vrednost odmah menja, i cppcheck predlaze da inicijalizacija nije neophodna.

```
QString time;
for (int i=0; i<HOURS_IN_DAY; i++) {
    for (int j=0; j<MINUTES_IN_HOUR; j+=MINUTE_INCREMENTS) {
```

```

        time = "";
//      time.sprintf("%02d:%02d", i, j);
        time = QString("%1:%2")
            .arg(i, 2, 10, QLatin1Char('0'))
            .arg(j, 2, 10, QLatin1Char('0'));
        m_verticalHeaders << time;
    }
}

```

5 Zaključak

U slučaju razvijanja velikih i kompleksnih programa prethodni alati nisu samo preporučljivi, već u većini slučajeva mogu biti i neophodni. Svaki alat ima svoje prednosti i mane, i većina programera nije navikla da ih svakodnevno koristi, ali bi to trebalo promeniti. Kao što smo mogli da primetimo, čak i na malom projektu kao što je Tudu, uočili smo neke greske i propuste. Koliko bi ih tek bilo da je projekat dva, tri ili deset puta veći? Analiziranje svih tipova memorije, procesorskog vremena, efikasnosti programa, pokrivenost koda pomazu u izgradnji softvera koji je od veće pouzdanosti i boljih performansi. Nekad je potrebno razvijati softver u ograničenim uslovima, gde su resursi mali. Tada bi posebno bilo potrebo obratiti pažnju na neke od prethodnih alata.