Verifikacija softvera

— If Broken it is, Fix it You Should —

Milena Vujošević Janičić

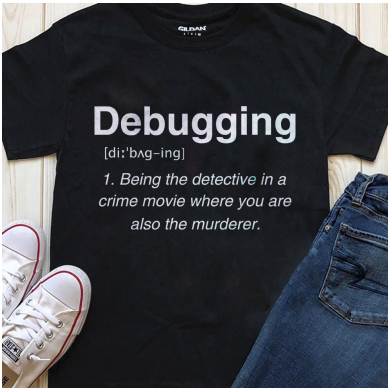
www.matf.bg.ac.rs/~milena

Matematički fakultet, Univerzitet u Beogradu

Pregled

Pregled

Debagovanje...



Debager

Debager (engl. *Debugger*)

Debager je program koji se koristi za praćenje rada drugog programa sa ciljem pronalaženja uzroka greške.

Neophodna podrška

Da bi informacije koje debager daje bile razumljivije, potrebna je podrška kompajlera/linkera.
Da bi debager mogao da radi, potrebna je podrška operativnog sistema i/ili hardvera.

Debageri

Poznati debageri

- GDB - the GNU debugger
- LLDB - LLVM based debugger
- WinDBG - Microsoft
- Microsoft Visual Studio Debugger

Debageri

Front end za debagere

- Debageri se često koriste kroz razvojna okruženja koja im daju grafički korisnički interfejs
- Na primer, GDB se može koristiti kroz QtCreator, Visual Studio Code, Eclipse, NetBeans... (`https://sourceware.org/gdb/wiki/GDBFrontEnds`)
- Udobnost GUIa je važna za lakšu upotrebljivost, ali kao i za sve ostale alate, važi da je to samo korisnički interfejs i ne treba poistovećivati GUI sa samim alatom

Release ili *Debug* mod?

Razlika između *release* i *debug* moda prevođenja

- Prilikom kompilacije projekta postoje razna podešavanja i opcije koje se mogu koristiti a koje služe za preciznije navođenje načina prevođenja projekta
- Jedna od osnovnih opcija je mogućnost prevođenja u *release* ili *debug* modu
- U čemu je razlika?

Prevođenje za korisnika

Release mode

- *Release mode* je prevođenje programa u izvršnu verziju namenjenu krajnjem korisniku
- Ovakvo prevođenje uključuje optimizacije koje omogućavaju efikasno izvršavanje koda
- Nekada je ova efikasnost vidljiva, nekada se ne može jednostavno opaziti
- Optimizacijama se gubi veza sa originalnim kodom, neki delovi koda se usled optimizacija obrišu, neki se pomere, neki se prezapišu...
- Veličina izvršnog fajla je manja nego izvršnog fajla prevedenog u *debug* modu

Prevođenje za programera

Debug mode

- *Debug mode* je prevođenje programa u izvršnu verziju namenjenu programeru
- Ovakvo prevođenje obično isključuje optimizacije sa ciljem lakšeg uparivanja izvornog i izvršnog koda
- Izvršavanje ovakvog programa može da bude manje efikasno
- U izvršnu verziju se umeću podaci koji su potrebni za povezivanje izvornog i izvršnog koda, tj omogućavaju debageru da precizno utvrdi koji deo koda se izvršava u datom trenutku
- Veličina izvršnog fajla je veća nego izvršnog fajla prevedenog u *release* modu, zbog manjka optimizacija i viška dodatnih informacija

Prevođenje za programera

Veza kompajler — debager (šta radi opcija -g?) — DWARF

- Prenos informacija: izvršna verzija programa — debager
- DWARF je format za predstavljanje pomoćnih informacija za debagovanje, koji se koristi od strane programskih prevodioca (kao npr. GCC ili LLVM/Clang) i debagera (kao npr. GNU GDB) da bi se omogućilo debagovanje na nivou izvornog koda. Omogućava podršku za razne programske jezike kao što su C/C++ i Fortran, ali je dizajniran tako da se lako može proširiti na ostale jezike. Arhitekturalno je nezavisan i predstavlja „most" između izvornog koda i izvršne datoteke.
- Format DWARF se primenjuje na programe UNIX-olikih operativnih sistema, kao što su Linux i MacOS.

Prevođenje za programera

DWARF

- Pogledati tekst o formatu DWARF u okviru master rada Đorđa Todorovića
  `http://www.verifikacijasoftvera.matf.bg.ac.rs/vs/predavanja/03_dinamicka_analiza/MasterRadDjordjeTodorovic.pdf`
- Debageri i programski prevodioci u okviru operativnog sistema Windows ne prate standard DWARF prilikom baratanja pomoćnim informacijama za debagovanje, već prate standard Microsoft CodeView.

Različite izvršne verzije mogu imati različita ponašanja

Različita ponašanja *debug* i *release* izvršavanja

- Može se desiti da se neke greške ispoljavaju samo u *release* verziji koda
- To je situacija koja može da nastane iz različitih razloga
  - *Debug* verzija koda, usled dodatnih podataka ili inicijalizacija memorije maskira grešku (dakle greška je ipak u originalnom kodu, ali je ne možemo naći na ovaj način)
  - Greška se pojavljuje kao rezultat greške kompajlera prilikom neke optimizacije (greška je u kompajleru, manja verovatnoća, ali i dalje naš problem jer nam treba izvršna verzija bez greške — treba naći način kako zaobići takvu grešku)

Debager

Debager - osobine

- Debager može da započne proces i da ga prati i debaguje, ili može da se nakači na proces koji se već izvršava.
- Debager omogućava izvršavanje programa instrukciju po instrukciju, omogućava postavljanje prekidnih tačaka (engl. *break point*) i izvršavanje programa do tih tačaka prekida kao i praćenje promenljivih i stanja na steku prilikom izvršavanja.
- Savremeni debageri omogućavaju i izmenu koda koji se izvršava i posmatranje efekta takvih izmena, debagovanje unazad, uslovne prekidne tačke i watchpoints

Debager

Debager - *release* ili *debug* mod

- Može se koristiti za svaku izvršnu verziju programa
- Mogućnosti i informacije koje se dobijaju za debag verziju su povezane sa izvornim kodom i olakšavaju programeru da poveže stanje procesora sa izvornim kodom.
- Za *release* verziju *debag* informacije su često samo uvid u asemblerski kôd, na isti način kao što ih vidi i procesor.
- Međutim, aktivno se radi na poboljšavanju korisničkog iskustva prilikom debagovanja koda sa prisutnim optimizacijama
  (primer rada Vaših kolega, Nikola Prica & Đorđe Todorović):
  `https://www.youtube.com/watch?v=1cWAmLMF1eI` — Debug info in optimized code - how far can we go? Improving LLVM debug info with function entry values

  `https://www.youtube.com/watch?v=ih5v65K10M8` — Improving Debug Information in LLVM to Recover Optimized-out Function Parameters

Ometanje debagovanja

Ometanje debagovanja

- Anti-debagovanje je implementacija jedne ili više tehnika unutar koda koje ometaju pokušaje obrnutog inženjeringa (engl. *reverse engineering*) ili debagovanja ciljanog procesa.
- Aktivno se koristi od strane poznatih izdavača u sferi zaštite od kopiranja, ali se takođe koristi i u malicioznim programima kako bi teže bili detektovani i eliminisani od strane antivirus programa.

Kako rade debageri?

Pojednostavljeno ...

Kada se postavi prekidna tačka u programu sa željom da se na tom mestu zaustavi program, debager umetne na to mesto u softveru instrukciju prekida ili neku nevalidnu instrukciju. Kada se prilikom izvršavanja programa naiđe na ovu instrukciju desi se hardverski izuzetak koji uzrokuje prekid.
Najpre se proveri da li je prekid u listi očekivanih debager prekida (tj da li je u pitanju namerno zaustavljanje ili greška u originalnom kodu). Ukoliko je greška u originalnom kodu, onda se dopusti da se ta greška i izvrši i da program pukne.

Kako rade debageri?

Pojednostavljeno ...

Ukoliko je u pitanju tačka prekida, prekid se prosledi na obradu debageru koji ga onda obradi tako što na tom mestu omogući uvid u sve vrednosti fizičkih registara procesa kao i u stanje memorije. Debager prikazuje pročitane informacije o procesu povezane sa informacijama o izvornom kodu koje su nalaze u programu umetnute od strane kompajlera/linkera prilikom prevođenja programa.
Ukoliko je u pitanju uslovna prekidna tačka, debager proverava uslov i u slučaju da uslov nije ispunjen, preskače se obrada prekida i samo se nastavlja dalje sa izvršavanjem procesa.

Kako rade debageri?

Pojednostavljeno ...

Kada korisnik poželi da nastavi sa izvršavanjem,

1. debager zameni instrukciju prekida sa originalnom instrukcijom,
2. izvrši je,
3. zameni ponovo originalnu instrukciju instrukcijom prekida,
4. prepusti ponovo dalje kontrolu programu.

Potrebna podrška

Operativni sistemi i hardver

- Pošto debageri hvataju sistemske prekide, oni su sistemski zavisni alati i za razumevanje njihovog funkcionisanja potrebno je razumeti procese i sistem prekida na odgovarajućem operativnom sistemu
- Na primer, pod Linux-om za rad debagera od suštinske važnosti je sistemska funkcija **ptrace**
- Za efikasno funkcionisanje, debager može da koristi i direktno neke funkcionalnosti hardvera, ukoliko su dostupne

Potrebna podrška

Operativni sistemi i hardver

- Primer: *watchpoints* — praćenje vrednosti neke promenljive u memoriji, tj da li se neka vrednost u memoriji menja ili se sa nje nešto čita. Ukoliko postoji podrška hardvera (specijalni registri koji pamte i proveravaju odgovarajuće adrese), biće podignut izuzetak koji će debager da obradi. Ukoliko to nije dostupno ili se zahteva praćenje većeg broja vrednosti nego što je to dostupno podrškom hardvera, onda debager mora da izvršava instrukciju po instrukciju i da za svaku proverava šta se dešava na traženim memorijskim lokacijama, što značajno usporava izvršavanje.

Potrebna podrška

Operativni sistemi - *ptrace*

- *ptrace* je sistemski poziv u Unix i njemu sličnim operativnim sistemima. Ime je skraćenica od „proces tragač" (engl. *process trace*). Korišćenjem *ptrace* jedan proces može da kontroliše drugi. Time upravljački proces ima mogućnost da upravlja unutrašnjim stanjem ciljanog procesa. *ptrace* najviše koriste debageri kako bi mogli da zaustavljaju program, da posmatraju memoriju i menjaju je.

```
long ptrace
(enum __ptrace_request request, pid_t pid, void *addr, void *data);
```

- Pogledati primere upotrebe ove funkcije u master radu Đorđa Todorovića ili seminarskom radu Nikole Dimitrijevića (dostupni u okviru strane kursa).

Osnovno debagovanje

Interaktivno debagovanje

- Interaktivno debagovanje obuhvata komunikaciju sa debagerom, tačke prekida, koračanje...
- Sistem za interaktivno debagovanje zahteva mogućnost kontrole toka izvršavanja programa: postavljanjem tačaka prekida (engl. *brakepoints*) izvršavanje programa se pauzira, korišćenjem komandi debagera analizira se progres programa a zatim se ponovo nastavlja sa izvršavanjem programa. Moguće je postaviti i uslovne izraze koji se proveravaju tokom izvršavanja programa i ukoliko se uslovi ispune, izvršavanje programa se zaustavlja i vrše se analize.

Napredne tehnike debagovanja

Udaljeno debagovanje

- Pored interaktivnog debagovanja, postoji i udaljeno debagovanje (engl. *remote debugging*)
- Daljinsko debagovanje je postupak debagovanja programa koji se izvršava na sistemu udaljenom od debagera. Da bi proces započeo, debager mora da se poveže sa udaljenim sistemom preko mreže. Tada debager može da kontroliše izvršavanje programa na udaljenom sistemu i da sakuplja informacije o njegovom stanju. Ovo je moguće ukoliko je udaljeni sistem iste arhitekture kao i arhitektura na kojoj se debager pokreće, ili ukoliko debager poseduje podršku za arhitekturu udaljenog sistema. Daljinsko debagovanje je posebno često za uređaje sa ugrađenim računarom.

Napredne tehnike debagovanja

Post-mortem debagovanje

- Post-mortem debagovanje je postupak debagovanja programa nakon njegovog prekida.
- Za ovu vrstu debagovanja često se koriste datoteke jezgra koje sistem generiše
- Povezane tehnike često uključuju razne tehnike praćenja i/ili analizu snimljenog stanja radne memorije programa u momentu kada je došlo do prekida.
- Tačan momenat prekida procesa može se ustanoviti automatski od strane sistema (npr. kada je proces završen zbog nekog odstupanja), preko instrukcija napisanih od strane programera ili eksplicitno od strane korisnika.

GNU GDB

GNU GDB

- GNU GDB je alat koji omogućava uvid u događanja unutar drugog programa koji se nalazi u fazi izvršavanja, ili u slučaju neregularnog prekida izvršavanja programa uvid u to šta se dešavalo pa je do toga došlo.
- GNU GDB debager takođe omogućava uvid u to šta se dešavalo sa programima i na platformama koje imaju različitu arhitekturu od arhitekture domaćina (engl. *host architecture*). Da bi se to realizovalo koristi se GNU GDB server, što se naziva udaljeno debagovanje, jer se udaljenom uređaju pristupa preko posebnih protokola.

GNU GDB

GNU GDB

- U svrhe debagovanja programa drugih procesorskih arhitektura na ličnim računarima se takođe upotrebljava Multiarch GNU GDB koji koristi biblioteke namenjene drugačijim arhitekturama.
- GDB se razvija u programskom jeziku C
- Programi koji mogu biti analizirani mogu biti napisani u raznim programskim jezicima, kao što su Ada, C, C++, Objective-C, Pascal, Fortran, Go.
- GNU GDB debager se može pokrenuti na najpopularnijim operativnim sistemima UNIX i Microsoft Windows varijanti.

LLDB

LLDB

- Debager koji se aktivno razvija u okviru LLVM projekta
- Podrška interaktivnom debagovanju
- LLDB se razvija u programskom jeziku C++
- Operativni sistemi Unix-oliki (ali i dalje manja podrška nego kod GDBa), od nedavno i za Windows, ali posebno bitan za MacOS
- Programi koji mogu biti analizirani mogu biti napisani u raznim programskim jezicima, kao što su C, C++, Objective-C, Swift
- Moguća integracija u različita razvojna okruženja

WinDbg i Visual Studio Debugger

WinDbg i Visual Studio Debugger

- WinDbg je višenamenski debager za Windows operativne sisteme, manje poznat od Visual Studio Debugger
- Oba omogućavaju interaktivno i post-mortem debagovanje, ali WinDbg ima naprednije mogućnosti
- Podržavaju jezike koji su podržani .Net platformom
- Za Visual Studio Debugger se podrazume integracija u grafičko okruženje Visual Studio

Debagovanje drugih programskih jezika

Ostali debageri

- Debagovanje se obično odnosi na tradicionalne programske jezike
- Na primer, za Javu postoji debager *jdb*

  `https://docs.oracle.com/javase/7/docs/technotes/tools/windows/jdb.html`

  Može se koristiti iz komandne linije na sličan način kao i *gdb*, može se integrisati u neko razvojno okruženje, npr Eclipse
- Postoje moduli koji su podrška debagovanju i u skript jezicima, na primer u Python-u modul pdb `https://docs.python.org/2.0/lib/module-pdb.html`
- Upotreba svih ovih debagera je slična, podrazumevaju se tačke prekida, koračanje i slično

Otvoreni problemi

Debagovanje višenitnih aplikacija

- Pronalaženje grešaka u višenitnim aplikacijama je suštinski teško
- Da bi debager omogućio debagovanje višenitnih aplikacija potrebno je da za to obezbedi dodatnu podršku
- Iako u najvažnijim debagerima postoji podrška za debagovanje višenitnih aplikacija, često je ta podrška loša i nepraktična, a kada postoji komunikacija između niti to često može da zbuni debager
- Pokretanje aplikacije kroz debager može da poremeti redosled izvršavanja u okviru aplikacije i na taj način zamaskira postojeće probleme

Otvoreni problemi

Ograničenost primene

- Debagovanje je primenljivo nad aplikacijama srednje veličina, tj ne skalira uvek
- Za kompleksniji i veći softver proces debagovanja može da bude previše spor

Protivnici debagera

Rob Pike (one of the authors of the Go language)

If you dive into the bug, you tend to fix the local issue in the code, but if you think about the bug first, how the bug came to be, you often find and correct a higher-level problem in the code that will improve the design and prevent further bugs.

Postoji izvesan broj značajnih programera koji ne vole debagere

- Linus Torvalds, the creator of Linux, does not use a debugger.
- Robert C. Martin, one of the inventors of agile programming, thinks that debuggers are a wasteful timesink.

Print umesto debagera

Brian W. Kernighan, Unix for Beginners (1979)

The most effective debugging tool is still careful thought, coupled with judiciously placed print statements.

The author of Python, Guido van Rossum

... uses print statements for 90% of his debugging.

Jedan od osnovnih razloga ...

... je navika uz činjenicu da debageri nisu oduvek bili moćni i uspešni alati kao što su sada.

Print umesto debagera

Prevaziđen stav

- Sa printom je potrebno prevesti ponovo program (build može da traje dosta dugo)
- Umetanje poziva funkcije štampanja remeti memoriju i može da maskira grešku
- Štampanjem se ne može videti vrednost svih interesantnih aspekata programa niti se može zaustaviti program
- Štampanje je statička aktivnost (za bilo kakvu izmenu ili dodatak potrebno je ponovo prevesti i pokrenuti program), a tačke prekida se mogu postavljati i uklanjati u fazi debagovanja

Print umesto debagera

Iako prevaziđen stav, i dalje se koristi...

- Neznanje
- Nekada i nema alternativu: ukoliko ne postoji debager za određenu platformu ili sistem
- Nekada debager ne uspeva da odradi posao jer maskira grešku
- ...

Pregled

Testiranje performansi i optimizacija

Performanse

- Važan deo testiranja performansi obuhvata merenje vremenske i memorijske efikasnosti programa
- Ukoliko program ne zadovoljava postavljene kriterijume, potrebno je naći uzrok i izvršiti optimizaciju

Optimizacija

- Proces optimizacije je neizostavan deo razvoja softvera.
- Da bi se uočili delovi koda koji treba da se poboljšaju i optimizuju, koriste se pomoćni alati, profajleri, koji generišu informacije na osnovu kojih se donose odluke o optimizacijama.

Profajliranje

Profajliranje

Profajliranje predstavlja vid dinamičke analize koda čiji je rezultat skup podataka dobijen izvršavanjem programa sa određenim ulaznim podacima. Profajliranje se zasniva na intrumentalizaciji, tj na ubacivanju dodatnih instrukcija u program kako bi se prikupili podaci o programu za vreme njegovog izvršavanja.

Profil

Profil

- Podaci dobijeni profajliranjem predstavljaju profil programa.
- Željeni podaci o programu obuhvataju, na primer, frekvencije izvršavanja funkcija ili blokova koda, procenat utrošenog vremena u bloku koda, podatke o alokaciji memorije...
- Ovi podaci pomažu u otkrivanju dela koda koji se često izvršava, određivanju pokrivenosti koda datim ulazima, proširivanju skupa testova i u rešavanju još mnogih drugih problema.

Profil

Merenja i rezultati

- Merenja se najčešće odnose na broj izvršavanja određenog dela koda ili vreme provedeno u tom delu koda.
- Važno je napomenuti da tokom programa upravljaju konkretni ulazi i da različiti ulazi daju različite rezultate profajliranja.
- Da bi se donosile odluke o optimizaciji na osnovu profajliranja, važna je pretpostavka da izvršavanje u okviru kojeg se vrši profajliranje reflektuje realnu upotrebu programa, odnosno da su skupljeni podaci na osnovu relevantnih ulaznih podataka ili da su skupljeni podaci na osnovu više različitih skupova ulaznih podataka. U suprotonom, mogu se doneti pogrešne odluke.

Testiranje performansi i optimizacija

Optimizacija

- Optimizaciju na osnovu rezultata profajliranja može vršiti čovek, a može se sprovoditi i automatski.
- Automatska optimizacija može da napravi važna poboljšanja u efikasnosti koda, ali samo čovek može da suštinski izmeni algoritam koji se koristi u implementaciji.
- Obe vrste optimizacija su važne i zapravo imaju različite domete.

Testiranje performansi i optimizacija

Optimizacija

- Automatska optimizacija može da se sprovodi u fazi izvršavanja koda ili u fazi kompilacije
- Optimizacija u fazi kompilacije koristi informacije dobijene profajliranjem da bi se generisala nova efikasnija izvršna verzija.
- Optimizacija u fazi izvršavanja korisiti informacije koje se dobijaju profajliranjem da bi se donele odluke o tome da se neki delovi izvršnog koda optimizuju u fazi izvršavanja

Potrebna podrška

Podrška profajliranju

- Profajliranje može da bude samo softversko, može da ima podršku u okviru hardvera, ili da bude kombinacija hardverskog i softverskog profajliranja
- Podrška hardvera u profajliranju omogućava veću efikasnost profajliranja kao i veći opseg podataka dobijenih na osnovu profajliranja.
- Za neka merenja neophodna je hardverska podrška, na primer, za merenje broja promašaja u keš memoriji ili za merenje utrošenog vremena zbog čekanja protočne obrade (engl. *pipline stall*) neke instrukcije.

Instrumentalizacija

Najvažnije osobine koje dobra instrumentalizacija treba da zadovolji su:

1. Prikuplja samo **potrebne** podatke.
2. Ne utiče na funkcionalnost programa.
3. Ne usporava previše rad programa.

Prvi uslov je važan jer zahtev za previše podataka dodatno usporava program i samu njihovu obradu, dok premalo informacija može biti beznačajno. Drugi uslov ističe da ukoliko dodata instrumentalizacija utiče na funkcionalnost programa onda prikupljeni podaci neće oslikavati pravi način njegovog rada. Poslednji uslov zavisi od tipa aplikacije i njega možemo kontrolisati u zavisnosti od granulacije, to jest od delova programa koji se instrumentalizuju.

Ograničenja profajliranja

Iako postoje mnogi metodi za intrumentalizaciju programa postoje određene situacije gde ih ne možemo lako koristiti.

- Na primer, sistemi u realnom vremenu imaju vrlo stroga vremenska ograničenja koja se profajliranjem mogu poremetiti (prekršiti) i time izazvati štetu
- Problem mogu da budu i memorijska ograničenja uređaja. Dodatni kôd za instrumentalizaciju i njeno rukovanje može uvećati program tako da on ne može da stane na uređaj.

Instrumentalizacija

Instrumentalizacija (engl. *instrumentation*)

- Instrumetalizacija se može podeliti na osnovu toga kako se nove instrukcije ubacuju u program:
  - Može je izvršiti sam programer, manuelnim dodavanjem dodatnih linija na željena mesta u kodu
  - Može se izvršiti automatski u različitim fazama.
- U okviru automatske instrumentalizacije,
  - Istrumentalizaciju može da sprovede kompajler i/ili linker,
  - Istrumentalizacija može da se ubaci u već iskompajlirani program.
  - Istrumentalizacija može da se ubaci za vreme izvršavanja programa.

Osnovne vrste profajliranja

Željeni podaci

Najčešće se prikupljaju informacije o količini izvršavanja određenih delova koda kako bi se prilikom optimizacije obratila pažnja baš na te delove. Ukoliko optimizujemo deo koda koji se retko izvršava, to neće značajno uticati na celokupne performanse. Najbitnije informacije su

1. Sekvence blokova koje se najčešće izvršavaju
2. Instrukcije (blokovi) koje se najčešće izvršavaju

Osnovne vrste profajliranja

Na osnovu željenih podataka, vrste profajliranja se mogu podeliti na

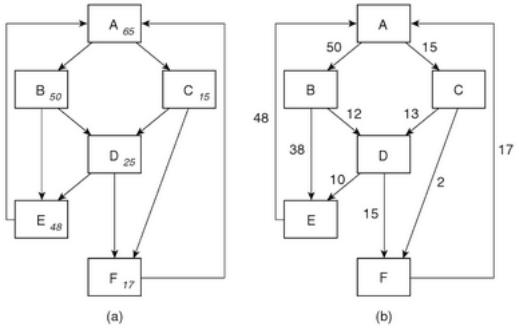
1. Profajliranje putanje (engl. *path profiling*).
2. Profajliranje grana (engl. *edge profiling*).
3. Profajliranje blokova (engl. *basic-block profiling*).

U okviru profajliranja postavljamo brojače kojima utvrđujemo koliko se puta neki događaj desio prilikom izvršavanja programa. Dobijeni profili se mogu koristiti za kompajlerski zasnovane optimizacije i za utvrđivanje pokrivenosti koda testovima.

Profajliranje putanja

Profajliranje putanja

- Profajliranje putanja je složen vid profajliranja kojim se dobijaju informacije o najčešće korišćenim putanjama kroz program.
- Ova vrsta profila u sebi sadrži i informacije o profilima grana i blokova.
- Zahteva kompleksne algoritme i najviše utiče na performanse izvršavanja prilikom profajliranja.



Profil dobijen profajliranjem blokova (a) i profajliranjem grana (b)

Profajliranje blokova

Profajliranje blokova

- Blokovi mogu biti funkcije ili deo koda u kome se ne nalaze instrukcije grananja ili skokova.
- Profajliranjem blokova broji se ukupan broj izvršavanja svakog bloka
- Naivni algoritam: može se ostvariti tako što se u svaki blok umeće brojač čime se dobijaju precizne informacije o broju izvršavanja blokova, ali se i prilično usporava sistem.
- Ovaj način profajliranja ne daje informacije o tome koje su putanje kroz program najčešće kao ni koji su prelazi između blokova česti

Profajliranje grana

Profajliranje grana

- Grana predstavlja prelazak koji se ostvaruje instrukcijom grananja ili skoka kojom se prebacuje tok izvršavanja programa iz jednog bloka u drugi.
- Profajliranjem grana mogu se dobiti i podaci koji se dobijaju profajliranjem blokova.
- Broj izvršavanja svakog bloka može se sračunati pomoću brojača grana tako što sumira sve grane koje ulaze u blok.
- Naivni algoritam: za svaku naredbu skoka umeće se brojač

Profajliranje grana

Efikasnije rešenje

- Profajliranje grana može da se implementira značajno efikasnije, pod pretpostavkom da se radi u fazi kompilacije programa.
- Jedno takvo rešenje prvi je teoretski uveo Donald Knut (engl. *Donald Knuth*) i on je pokazao da je broj umetnutih brojača u njegovom rešenju minimalan.

Knutov algoritam

Knutov algoritam profajliranja grana

- Najpre je neophodno napraviti graf kontrole toka (engl. *control flow graph*), u kojem svaki čvor predstavlja blok instrukcija, a grana naredbu skoka ili grananja.
- Za ovaj graf potrebno je napraviti razapinjuće stablo (engl. *spanning tree*). Broj grana u razapinjućem stablu je $v - 1$, gde je v broj čvorova grafa.
- Granama koje ne pripadaju dobijenom stablu treba dodati brojač.
- Broj izvršavanja grana koje ne sadrže brojač se može izračunati na osnovu sračunatih vrednosti

Podsetnik

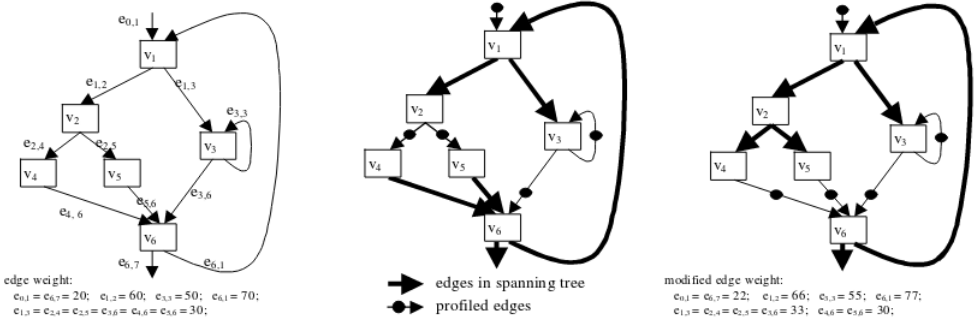
Razapinjuće stablo

Razapinjuće stablo je podskup grafa G koji sadrži sve čvorove pokrivene sa minimalnim mogućim brojem grana. Razapinjuće stablo ne sadrži cikluse i ne može biti nepovezano. Po definiciji, svaki povezan neusmeren graf G ima najmanje jedno razapinjuće stablo.

Profajliranje grana

Efikasnije rešenje

- Knutovo rešenje može istrumentalizovati isti broj grana ali na različite načine, jer standardni algoritam računanja razapinjućeg stabla može vratiti različita stabla u zavisnosti od redosleda obrade grana.
- Optimalno razapinjuće stablo je ono stablo kod kojeg se grane najveći broj puta izvršavaju (ali baš tu informaciju zapravo tražimo!)
- Tomas Bal (engl. *Tomas Ball*) i Džejms Larus (engl. *James R. Larus*) su osmislili način kako da se proceni koji skup grana je optimalan, tj kako da se korišćenjem jednostavne statičke analize instrumentalizuju one grane za koje se predviđa da će se najmanje puta izvršiti.



a. Control flow graph

b. Edge profiling with max-weight spanning tree

c. Edge profiling with modified max-weight spanning tree

Profajliranje grana

Odnos kvalitet — cena

Vreme profajliranja vs efikasnost izvršavanja

- Postizanje pravog poretka između vremena utrošenog prilikom profajliranja i same dobiti na osnovu dobijenih profila je od izuzetne važnosti.
- Profajliranjem se u prvim koracima dobijaju veliki rezultati, a kako se sve više optimizuje program to ga je teže dodatno optimizovati
- Zato se u prvim koracima mogu koristiti i manje precizne tehnike, a kasnije je potrebno koristiti sve preciznije tehnike.

Profajliranje uzimanjem uzoraka

Profajliranje uzimanjem uzoraka (engl. *sample based profiling*)

- Potrebno je smanjiti usporavanje programa instrumentalizacijom.
- Jedan od efikasnih načina koji znatno smanjuje opterećivanje programa je uzimanjem uzorka (engl. *sampling*).
- **Ova tehnika ne određuje način instrumentalizacije, već samo kako smanjiti njene troškove.**
- Uzimanje uzoraka se jako često koristi. Na primer, to je podrazumevan metod profajliranja za *Visual Studio Profiling Tools*

Profajliranje uzimanjem uzoraka

Profajliranje uzimanjem uzoraka

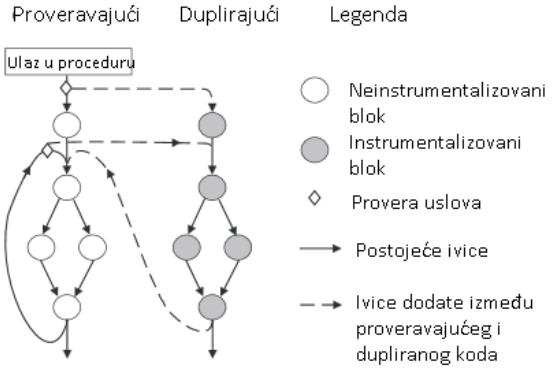
- Uzimaju se „slike" programa u određenim vremenskim intervalima i od njih pravi profil programa.
- Rezultat ovakvog pristupa je znatno manje opterećenje programa, ali kao posledicu ima smanjenu tačnost dobijenih podataka.
- Za dobro izabrane intervale merenja preciznost ove metode može da bude veoma visoka
- Međutim, ukoliko vremenski intervali nisu dobro odabrani može se desiti da se ne zabeleži određeni događaj.
- Najčešće se koristi za pronalaženje najfrekventnijih događaja.
- Primer: uzimanje slika stanja na steku

Algoritam uzorkovanja

Duplirani i proveravajući kôd

- Pored osnovnog koda pravimo duplikat koji će sadržati instrumentalizovani kôd i njega ćemo nazvati duplirani kôd.
- Orginalni kôd zovemo proveravajući zato što se u njemu ispituje uslov koji, ukoliko je ispunjen, treba kontrolisano da pređe u duplirani kôd. Ovim uslovom kontrolišemo koliko vremena će se izvršavati svaki od ova dva koda.
- Kada izvršavanje pređe u duplirani kôd ono tu ostaje ograničeno vreme a zatim se vraća u proveravajući.
- Trenutak prelaska iz proveravajućeg u duplirani kôd se može inicirati hardverski, putem operativnog sistema ili softverski.

Duplirajući kôd



Uslov prelaska

Time based sampling

- Fiksirani vremenski period između dva uzimanja uzoraka: tajmer postavlja bit, uslov proverava li je taj bit postavljen
- Period uzimanja uzoraka je ograničen frekvencijom tajmera što nije praktično za veoma česte događaje
- Kada istekne vreme, sledeći uzorak će biti uzet tek kada se ponovo ispita uslov prelaska

Uslov prelaska

Counter-based sampling

- Čuva se brojač za prelaske koji se dekrementira
- Kada brojač dođe do nule, uzima se uzorak i resetuje se brojač
- Očekuje se da je cena ovoga mala jer se brojač može držati u kešu
- Obezbeđeno je da će se uzorci uzimati proporcionalno broju provera

Counter-based vs Time based sampling — Tačnost

Benchmark	Timed-based(%)	Counter-based(%)
compress	88	98
jess	91	95
db	66	95
javac	59	73
mpegaudio	69	95
mtrt	51	67
jack	45	94
opt-compiler	58	65
pBOB	75	87
Volano	27	71
Avg	63	84

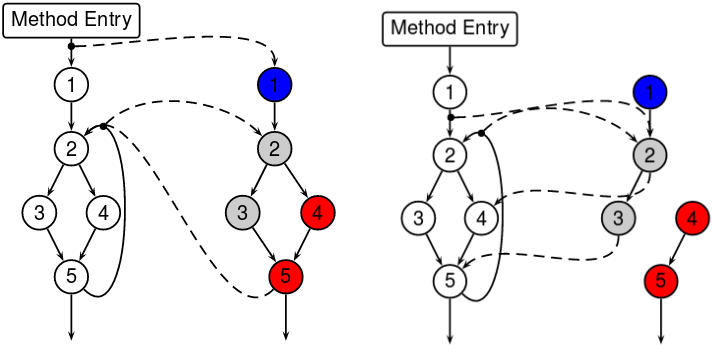
Primer: Timed-based trigger every 10 ms, counter set to approx 30,000

Algoritam uzorkovanja

Duplirani i proveravajući kôd

- Prikazani algoritam povećava potrebnu memoriju kao i vreme kompilacije.
- Mogu se instrumetalizovati samo delovi koda i postoje proširene verzije ovog algoritma koje ne prave celu kopiju koda već samo onih delova koji su vezani za instrumentalizaciju.
- Tj, za rekonstrukciju profila nije potrebno instrumentalizovati svaki blok ili svaku granu
- Delimičnim dupliranjem može se smanjiti upotreba memorije i vreme kompilacije sa potpunim zadržavanjem preciznosti dobijenih informacija

Delimično dupliranje



Bez dupliranja koda

Bez dupliranja koda

- Dodaje se provera ispred svakog instrumentalizovanog čvora
- Narušava se svojstvo proporcionalnosti i dodaje se veće opterećenje
- Rezultati nisu isti kao kod punog ili delimičnog dupliranja
- Profajliranje može da se zaustavi ranije

Dupliranje i bez dupliranja - poređenje

Sample Interval	*Full-Duplication*		*No-Duplication*	
	Overhead	**Accuracy**	**Overhead**	**Accuracy**
1	182.2%	100.0%	269.1%	100.0%
10	29.3%	99.5%	79.5%	99.0%
100	4.2%	98.5%	61.3%	98.0%
1,000	6.3%	95.5%	57.2%	95.5%
10,000	5.1%	88.0%	55.7%	88.5%
100,000	5.0%	77.0%	55.2%	78.5%

Kompajlerski zasnovana dinamička analiza

Ciljevi kompajlerske instrumentalizacije

- Instrumentalizacija može da se vrši i sa drugačijim ciljevima, tj cilj ne mora da bude samo profajliranje i optimizacija koda
- Na primer, česta upotreba instrumentalizacije je da se pronađu greške u radu sa memorijom ili u radu sa nitima
- Primer toga su santajzeri, koji se koriste u fazi razvoja programa kao pomoć u detektovanju grešaka

Kompajlerski zasnovana dinamička analiza

Detektovanje grešaka u radu sa memorijom

- `AddressSanitizer` (ASan) u okviru Clang-a i gcc-a imaju za cilj detektovanje prekoračenja bafera i korišćenje već oslobođene memorije
- Instrumentalizacija je za ovakve upotrebe drugačija i obuhvata i izmene u adresnim prostorima izvršavajućeg programa
- U proseku, instrumetalizacija u ovom slučaju povećava vreme izvršavanja za 73% a upotrebu memorije za 340%

Kompajlerski zasnovana dinamička analiza

Detektovanje grešaka u radu sa memorijom

- (clang) Memorijski sanitajzer (engl. *MemorySanitizer — MSan*) je detektor čitanja neincijalizovane memorije. Neincijalizovane vrednosti javljaju se u trenutku kada se sadržaj memorije koja je dodeljena steku ili hipu pročita pre nego što je u nju upisana ispravna vrednost. Ovaj tip sanitajzera otkriva slučajeve u kojima takve vrednosti utiču na izvršavanje programa.

Kompajlerski zasnovana dinamička analiza

Detektovanje grešaka u radu sa nitima

- `ThreadSanitizer` alat u okviru Clang-a pomaže u detketovanju problema u radu sa nitima.
- Koristi kompajlersku instrumetalizaciju i odgovarajuću run-time biblioteku.
- Tipično uspori izvršavanje od 5 do 15 puta. Memorijsko dodatno opterećenje je od 5 do 10 puta.

Pregled

Analiza izvršnog programa

Analiza izvršnog programa

- Dinamička analiza obuhvata analizu korisničkog programa u izvršavanju.
- Analiza izvršnog programa (binarna analiza, engl. *binary analysis*) obuhvata analizu na nivou mašinskog koda, snimljenog ili kao objektni kôd (nepovezan) ili kao izvršni kôd (povezan).
- Primer alata koji vrši naprednu analizu je Valgrind. Usporenje koje Valgrind nameće izvršavanju programa je od 5 do 100 puta, u zavisnosti od konkretnog alata.
- Ime potiče iz nordijske mitologije, ne od value (vrednost) + grind (mlevenje)
- **Izgovara se Velgrind**

Dostupnost Valgrinda

Valgrind radi na sledećim arhitekturama:

Linux - *x86*, *AMD64*, *ARM*, *ARM64*, *PPC32*, *PPC64*, *S390X*, *MIPS32*, *MIPS64*

Solaris - *x86*, *AMD64*

Android - *ARM*, *ARM64*,*x86* (4.0 i novije), *MIPS32*

Darwin - *x86*, *AMD64*(Mac OS X 10.12)

Valgrind

Valgrind distribucija sadrži naredne alate:

- detektor memorijskih grešaka *Memcheck*,
- praćenje rada dinamičke memorije *Massif*,
- profajler keš memorije *Cachegrind*,
- profajler funkcija *Callgrind*,
- detektor grešaka niti *Helgrind* i *DRD*.

Pokretanje: opcija --tool određuje koji alat će *Valgrind* koristiti.

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

Struktura Valgrinda

Valgrind se može koristiti i kao alat za pravljenje novih alata.

- Svi *Valgrind* alati rade na istoj osnovi, iako informacije koje se emituju variraju.
- Alat za dinamičku analizu koda se kreira kao dodatak, pisan u programskom jeziku *C*, na jezgro *Valgrind*-a.

  Jezgro Valgrind-a + alat koji se dodaje = Alat Valgrind-a
- Jezgro *Valgrind*-a omogućava izvršavanje klijentskog programa, kao i snimanje izveštaja koji su nastali prilikom analize samog programa.
- Informacije koje se emituju mogu se iskoristiti za otklanjanje grešaka, optimizaciju koda ili bilo koju drugu svrhu za koju je alat dizajniran.

Kako radi Valgrind?

Inicijalizacija

- Svaki *Valgrind*-ov alat je statički povezana izvršna datoteka koja sadrži kôd alata i kôd jezgra.
- Izvršna datoteka *valgrind* predstavlja program omotač koji na osnovu --tool opcije bira alat koji treba pokrenuti.
- *Valgrind*-ovo jezgro prvo inicijalizuje podsistem kao što su menadžer adresnog prostora, i njegov unutrašnji alokator memorije i zatim učitava klijentovu izvršnu datoteku.
- Potom se inicijalizuju *Valgrind*-ovi podsistemi kao što su translaciona tabela, aparat za obradu signala, raspoređivač niti i učitavaju se informacije za debagovanje klijenta, ukoliko postoje.

Kako radi Valgrind?

Nijedan deo koda klijenta se ne izvršava u svom izvornom obliku

- Od tog trenutka *Valgrind* ima potpunu kontrolu i počinje sa prevođenjem i izvršavanjem klijentskog programa.
- Može se reći da *Valgrind* vrši JIT (*Just In Time*) prevođenje mašinskog koda programa u mašinski kôd programa dopunjen instrumentalizacijom.
- Alat u originalni kôd umeće operacije u svrhu instrumentalizacije, zatim se takav kôd prevodi. Prevođenje se vrši dinamički.

Kako radi Valgrind?

Translacija

- Proces prevođenja se sastoji iz raščlanjivanja originalnog mašinskog koda u odgovarajuću međureprezentaciju (engl. *intermediate representation*, skraćeno IR) koji se kasnije instrumentalizuje sa alatom i ponovo prevodi u novi mašinski kôd.
- Rezultat ovog procesa se naziva translacija, koja se čuva u memoriji i koja se izvršava po potrebi.
- Jezgro troši najviše vremena na sam proces pravljenja, pronalaženja i izvršavanja translacije.

Kako radi Valgrind?

Osnovni blok

- *Valgrind* deli originalni kôd u sekvence koje se nazivaju osnovni blokovi.
- Osnovni blok je pravolinijska sekvenca mašinskog koda, na čiji se početak skače, a koja se završava sa skokom, pozivom funkcije ili povratkom u funkciju pozivaoca.
- Svaki kôd programa koji se analizira ponovo se prevodi na zahtev, pojedinačno po osnovnim blokovima, neposredno pre izvršavanja samog bloka.
- Veličina osnovnog bloka je ograničena na maksimalno šesdeset mašinskih instrukcija.

Translacija

Koraci translacije

- U nastavku su opisani koraci koje *Valgrind* izvršava prilikom analize programa.
- Postoji osam faza translacije.
- Sve faze obavlja jezgro *Valgrind*-a, osim instrumentalizacije koju obavlja alat *Valgrind*-a.

Translacija

Koraci translacije

Disasembliranje - Proces prevođenja mašinskog koda programa u ekvivalentni asemblerski kôd. *Valgrind* vrši prevođenje mašinskog koda programa u interni skup instrukcija koja se nazivaju međukod (engl. *intermediate representation*, skr. IR). U ovoj fazi međukod je predstavljen stablom. Ova faza je zavisna od arhitekture na kojoj se izvršava.

Optimizacija 1 - Prva faza optimizacije linearizuje *IR* reprezentaciju. Primenjuju se neke standardne optimizacije programskih prevodilaca kao što su uklanjanje redudantnog koda, eliminacija podizraza...

Translacija

Koraci translacije

Instrumentalizacija - Blok koda u *IR* reprezentaciji se prosleđuje alatu, koji može proizvoljno da ga transformiše. Prilikom instrumentalizacije alat u zadati blok dodaje dodatne *IR* operacije, kojima proverava ispravnost rada programa.

Optimizacija 2 - Druga faza optimizacije je jednostavnija od prve, uključuje izračunavanje matematičkih izraza koji se mogu izvršiti pre faze izvršavanja i uklanjanje mrtvog koda.

Izgradnja stabla - Linearizovana *IR* reprezentacija se konvertuje natrag u stablo radi lakšeg izbora instrukcija.

Translacija

Koraci translacije

Odabir instrukcija - Stablo *IR* reprezentacije se konvertuje u listu instrukcija koje koriste virtualne registre. Ova faza se takođe razlikuje u zavisnosti od arhitetkure na kojoj se izvršava.

Alokacija registara - Virtualni registri se zamenjuju stvarnim. Po potrebi se uvode prebacivanja u memoriju. Ne zavisi od platforme, koristi poziv funkcija koje nalaze iz kojih se registara vrši čitanje i u koje se vrši upis.

Asembliranje - Izabrane instrukcije se kodiraju na odgovarajući način i smeštaju u blok memorije. Ova faza se takođe razlikuje u zavisnosti od arhitekture na koji se izršava.

Izazovi uspešnog rada

Sklapanje dva procesa u jedan

- Smeštanje dva programa u jedan proces (klijentski program i program alata) uvodi izazove.
- Mnogi resursi se dele između ova dva programa, kao što su registri ili memorija.
- Sistemski pozivi se izvršavaju bez posredstva *Valgrind*-a, zato što jezgro *Valgrind*-a ne može da prati njihovo izvršavanje u samom jezgru operativnog sistema.

Sistemski pozivi

Sistemski pozivi - podsetnik

- Programi komuniciraju sa operativnim sistemom pomoću sistemskih poziva (engl. *system calls*), tj. preko operacija (funkcija) koje definiše operativni sistem.
- Sistemski pozivi se realizuju pomoću sistema prekida: korisnički program postavlja parametre sistemskog poziva na određene memorijske lokacije ili registre procesora, inicira prekid, operativni sistem preuzima kontrolu, uzima parametre, izvršava tražene radnje, rezultat stavlja na određene memorijske lokacije ili u registre i vraća kontrolu korisničkom programu.

Sistemski pozivi

Sistemski pozivi - podsetnik

Jezgro operativnog sistema deli virtuelnu memoriju na korisničku memoriju i sistemsku memoriju.

- Sistemska memorija je određena za samo jezgro operativnog sistema, njegova proširenja, kao i za upravljačke programe.
- Korisnički prostor je deo memorije gde se nalaze sve korisničke aplikacije prilikom njihovog izvršavanja. Korisničke aplikacije mogu da pristupe ulazno/izlaznim uređajima, virtualnoj memoriji, datotekama i drugim resursima jezgra operatvinog sistema samo koristeći sistemske pozive.

Sistemski pozivi

Sistemski pozivi - podsetnik

- Sistemski pozivi obezbeđuju spregu između programa koji se izvršava i operatvinog sistema. Generalno, realizuju se na asemblerskom jeziku, ali viši prograsmki jezici, poput jezika *C* i *C++*, takođe omogućavaju realizaciju sistemskog poziva.
- Program koji se izvršava može proslediti parametre opravitvnom sistemu na više načina, prosleđivanje parametara u registrima procesora, postavljanjem parametara u memorijskoj tabeli.

Pregled

Memcheck

Memcheck je najpoznatiji alat Valgrinda

- *Memcheck* je alat koji detektuje memorijske greške korisničkog programa. Kako ne vrši analizu izvornog koda već mašinskog, *Memcheck* ima mogućnost analize programa pisanog u bilo kom jeziku.
- Program koji radi pod kontrolom *Memcheck*-a je obično dvadeset do sto puta sporiji nego kada se izvršava samostalno, zbog translacije koda.
- Izlaz iz programa je dopunjen izlazom koji dodaje sam alat *Memcheck*, koji se ispisuje na standardnom izlazu za greške.

Pokretanje Memcheck-a

Poziv sledeće linije u terminalu:

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

Za programe pisane u jezicima C i C++

Detektuje uobičajne probleme u radu sa memorijom, na primer

- upisivanje podataka van opsega hipa i steka
- pristupanje memoriji koja je već oslobođena
- neispravno oslobađanje hip memorije, kao što je duplo oslobađanje hip blokova ili neuparenog korišćenja funkcija `malloc/new/new[]` i `free/delete/delete[]`
- curenje memorije

Pokretanje Memcheck-a

Za programe pisane u jezicima C i C++

Takođe, alat može da detektuje i

- korišćenje vrednosti koje nisu inicijalizovane ili koje su izvedene od drugih neinicijalizovanih vrednosti
- preklapanje parametara prosleđenih funkcijama (npr. preklapanje `src` i `dst` pokazivača kod funkcije `memcpy`).

Primer

Neinicijalizovana memorija

```
int main {
  int x;
  printf("%d\n",x);
  return 0;
}
```

```
==7070== Memcheck, a memory error detector
==7070== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==7070== Using Valgrind-3.14.0.GIT and LibVEX; rerun with -h for copyright info
==7070== Command: ../main
==7070==
==7070== Conditional jump or move depends on uninitialised value(s)
==7070==    at 0x4E814CE: vfprintf (vfprintf.c:1660)
==7070==    by 0x4E8B3D8: printf (printf.c:33)
==7070==    by 0x400548: main (in /export/main)
==7070==
```

...

```
==7070== Conditional jump or move depends on uninitialised value(s)
==7070==    at 0x4E8161C: vfprintf (vfprintf.c:1660)
==7070==    by 0x4E8B3D8: printf (printf.c:33)
==7070==    by 0x400548: main (in /export/main)
==7070==
x = 0
==7070==
==7070== HEAP SUMMARY:
==7070==     in use at exit: 0 bytes in 0 blocks
==7070==   total heap usage: 0 allocs, 0 frees, 0 bytes allocated
==7070==
==7070== All heap blocks were freed -- no leaks are possible
==7070==
==7070== For counts of detected and suppressed errors, rerun with: -v
==7070== Use --track-origins=yes to see where uninitialised values come from
==7070== ERROR SUMMARY: 6 errors from 6 contexts (suppressed: 0 from 0)
```

Memcheck

Prikaz greške

- Prve tri linije se štampaju prilikom pokretanja bilo kog alata koji je u sklopu *Valgrind*-a
- Sledeći deo nam pokazuje poruke o greškama koje je *Memcheck* pronašao u programu.
- Poslednja linija prikazuje rezime svih grešaka koje je alat pronašao i štampa se po završetku rada.

Memcheck

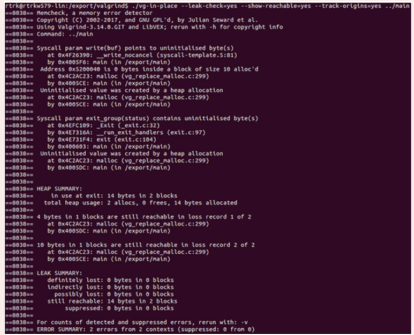
Prikaz greške

- U programu neinicijalizovana promenljiva može više puta da se kopira, *Memcheck* prati sve to, beleži podatke o tome, ali ne prijavljuje grešku.
- U slučaju da se neinicijalizovane vrednosti koriste na način da od te vrednosti zavisi dalji tok programa ili ako je potrebno prikaziti vrednosti neinicijalizovane promenljive, *Memcheck* prijavljuje grešku.

Primer

Sistemski poziv

```
#include <unistd.h>
int main()
{
  char* arr  = malloc(10);
  int*  arr2 = malloc(sizeof(int));
  write( 1 /* stdout */, arr, 10 );
  exit(arr2[0]);
}
```



Memcheck

Prikaz greške

- Prva greška prikazuje da parametar *arr* sistemskog poziva *write()* pokazuje na neinicijalizovanu vrednost.
- Druga greška prikazuje da je podatak koji se prosleđuje sistemskom pozivu *exit()* nedefinisan.
- Takođe, prikazane su i linije u samom programu gde se ove vrednosti koriste.
- U okviru LEEK SUMMARY dato je da postoje „still reachable" bajtovi

Memcheck

Curenje memorije

- *Memcheck* beleži podatke o svim dinamičkim blokovima koji su alocirani tokom izvršavanja programa pozivom funkcija `malloc()`, `new()` i dr.
- Kada program prekine sa radom, *Memcheck* tačno zna koliko memorijskih blokova nije oslobođeno.
- Ako je opcija *--leak-check* uključena, za svaki neoslobođeni blok *Memcheck* određuje da li je moguće pristupiti tom bloku preko pokazivača.

Memcheck

Curenje memorije

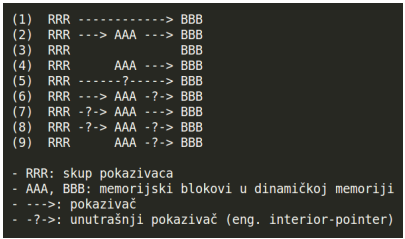
- Postoje dva načina da pristupimo sadržaju nekog memorijskog bloka preko pokazivača.
- Prvi način je preko pokazivača koji pokazuje na početak memorijskog bloka.
- Drugi način je preko pokazivača koji pokazuje na sadržaj unutar memorijskog bloka.
- Drugi način je često problematičan

Memcheck

Unutrašnjost bloka

- Postoji nekoliko načina da nastanu pokazivači koji pokazuje na unutrašnjost nekog memorijskog bloka.
- Jedan način je da je postojao pokazivač koji je pokazivao na početak bloka, ali je namerno (ili nenamerno) pomeren da pokazuje na unutrašnjost bloka.
- Drugi način je kada postoji odgovarajuća vrednost u memoriji, koja je u potpunosti nepovezana i slučajna.

Memcheck



Primer pokazivača na memorijski blok

Na slici je prikazano devet mogućih slučajeva kada pokazivači pokazuju na neke memorijske blokove. *Memcheck* objedinjuje neke od ovih slučajeva, tako da dobijamo naredne četiri kategorije.

Memcheck

Još uvek dostupan (engl. *Still reachable*)

Ovo pokriva primere 1 i 2. Pokazivač koji pokazuje na početak bloka ili više pokazivača koji pokazuju na početak bloka su pronađeni. Zato što postoje pokazivači koji pokazuju na memorijsku lokaciju koja nije oslobođena, programer može da oslobodi memorijsku lokaciju neposredno pre završetka izvršavanja programa.

```
(1)  RRR ------------> BBB
(2)  RRR ---> AAA ---> BBB
(3)  RRR               BBB
(4)  RRR      AAA ---> BBB
(5)  RRR ------?-----> BBB
(6)  RRR ---> AAA -?-> BBB
(7)  RRR -?-> AAA ---> BBB
(8)  RRR -?-> AAA -?-> BBB
(9)  RRR       AAA -?-> BBB

- RRR: skup pokazivaca
- AAA, BBB: memorijski blokovi u dinamičkoj memoriji
- --->: pokazivač
- -?->: unutrašnji pokazivač (eng. interior-pointer)
```

Memcheck

Defininitivno izgubljen (engl. *Definitely lost*)

Ovo se odnosi na slučaj 3. Ovo znači da je nemoguće pronaći pokazivač koji pokazuje na memorijski blok. Blok je proglašen izgubljenim, zauzeta memorija ne može da se oslobodi pre završetka programa, jer ne postoji pokazivač na nju.

```
(1)  RRR ------------> BBB
(2)  RRR ---> AAA ---> BBB
(3)  RRR               BBB
(4)  RRR      AAA ---> BBB
(5)  RRR ------?-----> BBB
(6)  RRR ---> AAA -?-> BBB
(7)  RRR -?-> AAA ---> BBB
(8)  RRR -?-> AAA -?-> BBB
(9)  RRR      AAA -?-> BBB

- RRR: skup pokazivaca
- AAA, BBB: memorijski blokovi u dinamičkoj memoriji
- --->: pokazivač
- -?->: unutrašnji pokazivač (eng. interior-pointer)
```

Memcheck

Indirektno izgubljen (engl. *Indirectly lost*)

Ovo pokriva slučajeve 4 i 9. Memorijski blok je izgubljen, ne zato što ne postoji pokazivač koji pokazuje na njega, nego zato što su svi blokovi koji ukazuju na njega izgubljeni. Na primer, ako imamo binarno stablo i koren je izgubljen, sva deca čvorovi su indirektno izgubljeni. S obzirom na to da će problem nestati ako se popravi pokazivač na definitivno izgubljen blok koji je uzrokovao indirektno gubljenje bloka, *Memcheck* neće prijaviti ovu grešku ukoliko nije uključena opcija *--show-reachable=yes*.

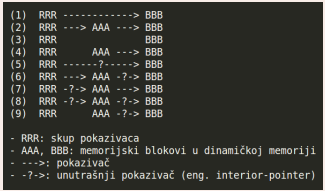
```
(1)  RRR ------------> BBB
(2)  RRR ---> AAA ---> BBB
(3)  RRR               BBB
(4)  RRR       AAA ---> BBB
(5)  RRR ------?-----> BBB
(6)  RRR ---> AAA -?-> BBB
(7)  RRR -?-> AAA ---> BBB
(8)  RRR -?-> AAA -?-> BBB
(9)  RRR       AAA -?-> BBB

- RRR: skup pokazivaca
- AAA, BBB: memorijski blokovi u dinamičkoj memoriji
- --->: pokazivač
- -?->: unutrašnji pokazivač (eng. interior-pointer)
```

Memcheck

Moguće izgubljen (engl. *Possibly lost*)

Ovo su slučajevi od 5 do 8. Pronađen je jedan ili više više pokazivača na memorijski blok, ali najmanje jedan od njih pokazuje na unutrašnjost memorijskog bloka. To može biti samo slučajna vrednost u memoriji koja pokazuje na unutrašnjost bloka, ali ovo ne treba smatrati u redu dok se ne razreši slučaj pokazivača koji pokazuje na unutrašnjost bloka.



Memcheck

Curenje memorije

Ako postoji zabrana prikazivanja greške za određeni memorijski blok, bez obzira kojoj od gore pomenutih kategorija pripada, ona neće biti prikazana.

```
LEAK SUMMARY:
   definitely lost: 47 bytes in 3 blocks
   indirectly lost: 0 bytes in 0 blocks
     possibly lost: 0 bytes in 0 blocks
   still reachable: 0 bytes in 0 blocks
        suppressed: 0 bytes in 0 blocks
```

Rezime curenja memorije

```
16 bytes in 1 blocks are definitely lost in loss record 2 of 3
   at 0x4847838: malloc (vg_replace_malloc.c:299)
   by 0x4007B4: main (in /home/aleksandrak/main)

19 bytes in 1 blocks are definitely lost in loss record 3 of 3
   at 0x4847838: malloc (vg_replace_malloc.c:299)
   by 0x40073C: main (in /home/aleksandrak/main)
```

Izveštaj o curenju memorije

Memcheck

Prikaz greške

- Ako je uključena opcija *--leak-check=yes*, *Memcheck* će prikazati detaljan izveštaj o svakom definitivno ili moguće izgubljenom bloku, kao i o tome gde je on alociran.
- *Memcheck* nam ne može reći kada, kako ili zašto je neki memorijski blok izgubljen.
- Generano, program ne treba da ima nijedan definitvno ili moguće izgubljen blok na izlazu.

Memcheck

Kriterijumi curenja memorije

- Zbog postojanja više tipova curenja memorije postavlja se pitanje koje curenje memorije na izlazu iz programa treba da posmatramo kao „grešku", a koju ne. *Memcheck* koristi sledeći kriterijum:
  - *Memcheck* smatra da je curenje memorije „greška" samo ako je uključena opcija *--leak-check=full*.
  - Definitivno i moguće izgubljeni blokovi se smatraju za pravu „grešku", dok indirektno izgubljeni i još uvek dostupni blokovi se ne smatraju kao greška.

Massif

Massif je alat za analizu hip memorije korisničkog programa.

- Postoje određeni scenariji curenja memorije koji ne spadaju u klasične, i takve propuste ne može detektovati *Memcheck*. Ovo se dešava zato što memorija nije zapravo izgubljena, pokazivač na nju i dalje postoji, ali ona se više ne koristi.
- Programi sa ovakvim tipom curenja memorije dovode do nepotrebnog korišćenja određene količine memorije tokom svog izvršavanja. *Massif* pomaže u otkrivanju baš ovakvih curenja memorije.
- *Massif* ne daje samo informaciju o tome koliko hip memorije se koristi, već i detaljne informacije koje upućuju na to koji deo programa je odgovoran za alociranje te memorije

Upotreba

Upotreba

- Program koji se izvršava pod alatom *Massif* radi veoma sporo. Nakon završetka rada, sve statistike su ispisane u fajl.
- Podrazumevani fajl u koji se piše je *massif.out.<pid>*, gde *<pid>* predstavlja ID procesa. Može se promeniti fajl u kome će se ispisivati komandom *--massif-out-file*.
- Da bi informacije koje je *Massif* sakupio mogli da vidimo u čitljivom formatu, koristimo *ms_print*. Na primer, ako imamo fajl *massif.out.1234*:

```
ms_print massif.out.1234
```

Upotreba

Upotreba

- *ms_print* prozvodi graf koji pokazuje trošenje memorije tokom izvršavanja programa, kao i detaljne informacije o različitim tačkama programa koje su odgovorne o alokaciji memoriji.
- Korišćenje različitih skripti za prezentaciju rezultata je namerno, jer odvaja sakupljanje podataka od prezentacije, kao i da je moguće dodati nov način prikaza podataka u svakom trenutku.

Merenje hipa

Merenje hipa je ograničeno

- Alat *Massif* meri samo hip memoriju, odnosno memoriju koja je alocirana `malloc`, `calloc`, `realloc`, `memalign`, `new`, `new[]` i nekoliko drugih sličnih funkcija.
- To znači da *Massif* ne meri memoriju koja je alocirana sistemskim pozivima kao što su `mmap`, `mremap` i `brk`.

Cachegrind

Cachegrind profajler

- *Cachegrind* je alat koji omogućava softversko profajliranje keš memorije tako što simulira i prati pristup keš memoriji mašine na kojoj se program, koji se analizira, izvršava. Takođe, može se koristiti i za profajliranje izvršavanja grana.
- On simulira memoriju mašine, koja ima prvi nivo keš memorije podeljene u dve odvojene nezavisne sekcije: *I1* - sekcija keš memorije u koju se smeštaju instrukcije i *D1* - sekcija keš memorije u koju se smeštaju podaci.
- Drugi nivo keš memorije koju *Cachegrind* simulira je objedinjen - *L2*. Ovaj način konfiguracije odgovara mnogim modernim mašinama.

Cachegrind

Cachegrind profajler

- Postoje mašine koje imaju i tri ili četiri nivoa keš memorije. U tom slučaju, *Cachegrind* simulira pristup prvom i poslednjem nivou.
- Generalno gledano, *Cachegrind* simulira *I1*, *D1* i *LL* (poslednji nivo keš memorije).
- *Cachegrind* prikuplja naredne statističke podatke o programu koji analizira (skraćenice koje se koriste dalje u tekstu su date u zagradama):

Cachegrind

Podaci o čitanjima instrukcija iz keš memorije uključuju sledeće statistike

Ir - ukupan broj izvršenih instrukcija

I1mr - broj promašaja čitanja instrukcija iz keš memorije nivoa *I1*

ILmr - broj promašaja čitanja instrukcija iz keš memorije nivoa *LL*

Cachegrind

Podaci o čitanjima instrukcija iz keš memorije uključuju sledeće statistike

Dr - ukupan broj čitanja memorije

D1mr - broj promašaja čitanja nivoa keš memorije *D1*

DLmr - broj promašaja čitanja nivoa keš memorije *LL*

Cachegrind

Podaci o čitanjima instrukcija iz keš memorije uključuju sledeće statistike

Dw - ukupan broj pisanja u memoriji

D1mw - broj promašaja pisanja u nivo keš memorije *D1*

DLmw - broj promašaja pisanja u nivo keš memorije *LL*

Cachegrind

Statistike

- Statistika se prikuplja na nivou celog programa, kao i pojedinačno na nivou funkcija.
- Na modernim mašinama *L1* promašaj košta oko 10 procesorskih ciklusa, *LL* promašaj košta oko 200 procesorskih ciklusa.

Cachegrind

Korišćenje *Cachegrind*-a

- Program koji želimo da analiziramo pokrećemo *Cachegrind*-om.
- Na standardni izlaz se ispisuju sumarne informacije, dok se detaljne informacije upisuju u `cachegrind.out.<pid>`, gde *pid* predstavlja jedinsteveni identifikator procesa koji se izvršio.
- Alat grupiše sve troškove po fajlovima i funkcijama kojima ti troškovi pripadaju.
- Globalna statistika se računa prilikom prikaza rezultata. Na ovaj način se štedi vreme. Funkcije koje simuliraju pristup keš memoriji se pozivaju jako često, tako da bi dodavanje još nekoliko instrukcija koje sabiraju, znatno usporilo i ovako sporo izvršavanje alata.

Cachegrind

```
==28165== Cachegrind, a cache and branch-prediction profiler
==28165== Copyright (C) 2002-2015, and GNU GPL'd, by Nicholas Nethercote et al.
==28165== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==28165== Command: trunk/cachegrind/tests/clreq
==28165==
==28165==
==28165== I   refs:      102,190
==28165== I1  misses:        728
==28165== LLi misses:        721
==28165== I1  miss rate:    0.71%
==28165== LLi miss rate:    0.71%
==28165==
==28165== D   refs:       39,582  (26,533 rd   + 13,049 wr)
==28165== D1  misses:      1,853  ( 1,358 rd   +    495 wr)
==28165== LLd misses:      1,498  ( 1,040 rd   +    458 wr)
==28165== D1  miss rate:     4.7% (   5.1%     +    3.8%  )
==28165== LLd miss rate:     3.8% (   3.9%     +    3.5%  )
==28165==
==28165== LL refs:         2,581  ( 2,086 rd   +    495 wr)
==28165== LL misses:       2,219  ( 1,761 rd   +    458 wr)
==28165== LL miss rate:      1.6% (   1.4%     +    3.5%  )
```

Izveštaj alata *Cachegrind*

Cachegrind

Korišćenje *Cachegrind*-a

- Generisani fajl je čitljiv, ali je bolje je koristiti alat *cg_annotate* koji nam lepše prikazuje detaljan izveštaj.
- Alat *cg_merge* sumira u jednu datoteku više izlaza dobijenih višestrukim pokretanjem *Cachegrind*-a nad istim programom. Tu datotetku koristimo kao ulaz u *cg_annotate*.
- Alat *cg_diff* pravi razliku između više izlaza iz alata *Cachegrind*, koje takođe možemo da koristimo kao ulaz u alat *cg_annotate*.

Callgrind

Profajler Callgrind

- *Callgrind* je alat koji generiše listu poziva funkcija korisničkog programa u vidu grafa.
- U osnovnim podešavanjima sakupljeni podaci sastoje se od broja izvršenih instrukcija, njihov odnos sa linijom u izvršnom kodu, odnos pozivaoc/pozvan između funkcija, kao i broj takvih poziva.
- Dodatno može da vrši analizu upotrebe keš memorije i profajliranje grana programa slično kao kod Cachegrinda.

Callgrind

Profajler Callgrind

- Kada se koristi Valgrind, vrši se kompilacija programa za debug mod, kako bi bile prisutne debag informacije koje olakšavaju praćenje stanja programa.
- Ali, za razliku od normalnog korišćenja debagera ili Valgrinda, za pozve Callgrinda (kao i Cachegrinda) kompilacija treba da bude sa optimizacijom, jer nema smisla profajlirati kôd koji je drugačiji od onoga koji će se normalno izvršavati.

Callgrind

Callgrind komande

- Podaci koji se analiziraju se zapisuju u fajl nakon završetka rada programa i alata.
- Podržane komande su:

callgrind_annotate - na osnovu generisanog fajla prikazuje listu funkcija. Za grafičku vizuelizaciju preporučuju se dodatni alati *KCachegrind*, 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.

Callgrind

Ekskluzivni i inkuzivni pristup

- Alat *Cachegrind* sakuplja podatke, odnosno broji događaje koji se dešavaju direktno u jednoj funkciji. Ovaj mehanizam sakupljanja podataka se naziva ekskluzivnim.
- Alat *Callgrind* proširuje ovu funkcionalnost tako što propagira cenu funkcije preko njenih granica. Na primer, ako funkcija `foo` poziva funkciju `bar`, cena funkcije `bar` se dodaje funkciji `foo`. Kada se ovaj mehanizam primeni na celu funkciju, dobija se slika takozvanih inkluzivnih poziva, gde cena svake funkcije uključuje i cene svih funkcija koje ona poziva, direktno ili indirektno.

Callgrind

Callgrind profajliranje

- Zahvaljujući grafu poziva, može da se odredi, počevši od `main` funkcije, koja funkcija ima najveću cenu poziva.
- Pozivaoc/pozvan cena je izuzetno korisna za profilisanje funkcija koje imaju više poziva iz raznih funkcija, i gde imamo priliku optimizacije našeg programa menjajući kôd u funkciji koja je pozivaoc, tačnije redukovanjem broja poziva.

Helgrind i DRD

Helgrind i DRD

- *Helgrind* otkriva greške sinhronizacije prilikom upotrebe modela niti *POSIX*.
- *DRD* je alat za detekciju grešaka u *C* i *C++* programima koji koriste više niti, radi za svaki program koji koristi niti *POSIX* standarda ili koji koriste koncepte koji su nadograđeni na ovaj standard.
- Alati *DRD* i *Helgrind* ne koriste iste algoritme za otkrivanje grešaka, samim tim ne otkrivaju iste tipove greška, iako imaju veliki broj poklapanja.

Helgrind i DRD

Greške koje alati otkrivaju u korišćenju API-ja POSIX niti:

- greške u otključavanju muteksa - kada je muteks nevažeći, nije zaključan ili je zaključan od strane druge niti
- greške u radu sa zaključanim muteksom - uništavanje nevažećeg ili zaključanog muteksa, dealokacija memorije koja sadrži zaključan muteks
- greške prilikom korišćenja funkcije `pthread_cond_wait` - prosleđivanje nezaključanog, nevažećeg ili muteksa koga je zaključala druga nit
- greške sa `pthread barrier` - nevažeća ili dupla inicijalizacija, čekanje na objekat koji nije nikada inicijalizovan...

Helgrind i DRD

Greške koje alati otkrivaju

- Mrtvo blokiranje kao posledica problema u redosledu zaključavanja
- Pristup memoriji bez adekvatnog zaključavanja i sinhornizacije
- DRD — zadržavanje katanca i lažno deljenje

Pregled

Literatura

Literatura

- Virtual Machines: Versatile Platforms for Systems and Processes (The Morgan Kaufmann Series in Computer Architecture and Design) by Jim Smith and Ravi Nair
- Thomas Ball and James R. Larus. 1994. Optimally profiling and tracing programs. ACM Trans. Program. Lang. Syst. 16, 4 (July 1994), 1319-1360. DOI=http://dx.doi.org/10.1145/183432.183527
  https://www.classes.cs.uchicago.edu/archive/2017/fall/32001-1/papers/ball-larus-profiling.pdf
  http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.27.5085&rep=rep1&type=pdf
  ftp://ftp.cs.wisc.edu/wwt/micro96.pdf

Literatura

Literatura

- Thomas Ball, Peter Mataga, and Mooly Sagiv. 1998. Edge profiling versus path profiling: the showdown. In Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '98). ACM, New York, NY, USA, 134-148. DOI=http://dx.doi.org/10.1145/268946.268958
- http://valgrind.org/
- http://valgrind.org/docs/manual/manual.html
- https://clang.llvm.org/docs/AddressSanitizer.html

Literatura

Literatura na srpskom

- Đorđe Todorović, master rad, Podrška za naprednu analizu promenljivih lokalnih za niti pomoću alata GNU GDB `http://www.racunarstvo.matf.bg.ac.rs/MasterRadovi/2018_09_01_DjordjeTodorovic/rad.pdf`
- Nikola Prica, master rad, Podrška za profajliranje softvera za uređaje sa ugrađenim računarom `http://www.racunarstvo.matf.bg.ac.rs/MasterRadovi/2016_06_01_Nikola_Prica/rad.pdf`

Literatura

Literatura na srpskom

- Aleksndra Kardžić, master rad, Alat Valgrind - implementacije konvencije FPXX za arhitekturu MIPS `http://www.racunarstvo.matf.bg.ac.rs/MasterRadovi/2017_03_30_Aleksandra_Karadzic/rad.pdf`
- Seminarski radovi, dostupni na strani kursa