УНИВЕРЗИТЕТ У НОВОМ САДУ
ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА

УНИВЕРЗИТЕТ У НОВОМ САДУ
ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА
НОВИ САД
Департман за рачунарство и аутоматику
Одсек за рачунарску технику и рачунарске комуникације

ДИПЛОМСКИ – МАСТЕР РАД

Кандидат: **Дејан Јевтић**
Број индекса: **57/2010**

Тема рада: **Прилагођавање алата за динамичку анализу програмског кода Велгринд за архитектуру МИПС**

Ментор рада: **проф. др Никола Теслић**

Нови Сад, октобар 2011.

УНИВЕРЗИТЕТ У НОВОМ САДУ ● **ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА**
21000 НОВИ САД, Трг Доситеја Обрадовића 6

КЉУЧНА ДОКУМЕНТАЦИЈСКА ИНФОРМАЦИЈА

Редни број, **РБР**:	
Идентификациони број, **ИБР**:	
Тип документације, **ТД**:	Монографска документација
Тип записа, **ТЗ**:	Текстуални штампани материјал
Врста рада, **ВР**:	Дипломски – мастер рад
Аутор, **АУ**:	**Дејан Јевтић**
Ментор, **МН**:	**проф. др Никола Теслић**
Наслов рада, **НР**:	**Прилагођавање алата за динамичку анализу програмског кода Велгринд за архитектуру МИПС**
Језик публикације, **ЈП**:	Српски / латиница
Језик извода, **ЈИ**:	Српски
Земља публиковања, **ЗП**:	Република Србија
Уже географско подручје, **УГП**:	Војводина
Година, **ГО**:	**2011**
Издавач, **ИЗ**:	Ауторски репринт
Место и адреса, **МА**:	Нови Сад; трг Доситеја Обрадовића 6
Физички опис рада, **ФО**: (поглавља/страна/ цитата/табела/слика/графика/прилога)	**10/63/4/31/3/0/0**
Научна област, **НО**:	Електротехника и рачунарство
Научна дисциплина, **НД**:	Рачунарска техника
Предметна одредница/Кључне речи, **ПО**:	**Велгринд, динамичка анализа кода, наменски системи, архитектура МИПС,**
УДК	
Чува се, **ЧУ**:	У библиотеци Факултета техничких наука, Нови Сад
Важна напомена, **ВН**:	
Извод, **ИЗ**:	**Циљ овог рада је да опише процес прилагођавања алата за динамичку анализу програмског кода Велгринд архитектури МИПС, као и да на том примеру покаже оне делове овог алата који су зависни од архитектуре на којој се извршавају. Као референтна платформа одабран је оперативни систем Линукс који је базиран на процесорима МИПС. У раду се налази опис рада Велгринда, као и приказ концепта употребе других алата ослонјених на Велгринд.**
Датум прихватања теме, **ДП**:	
Датум одбране, **ДО**:	
Чланови комисије, **КО**: Председник:	
Члан:	Потпис ментора
Члан, ментор:	

UNIVERSITY OF NOVI SAD ● **FACULTY OF TECHNICAL SCIENCES**
21000 NOVI SAD, Trg Dositeja Obradovića 6

KEY WORDS DOCUMENTATION

Accession number, **ANO**:	
Identification number, **INO**:	
Document type, **DT**:	Monographic publication
Type of record, **TR**:	Textual printed material
Contents code, **CC**:	Master Thesis
Author, **AU**:	**Dejan Jevtić**
Mentor, **MN**:	**Nikola Teslić, Ph. D.**
Title, **TI**:	**Porting Valgrind framework for building dynamic analysis tools to MIPS architecture**
Language of text, **LT**:	Serbian
Language of abstract, **LA**:	Serbian
Country of publication, **CP**:	Republic of Serbia
Locality of publication, **LP**:	Vojvodina
Publication year, **PY**:	**2011.**
Publisher, **PB**:	Author's reprint
Publication place, **PP**:	Novi Sad, Dositeja Obradovica sq. 6
Physical description, **PD**: (chapters/pages/ref./tables/pictures/graphs/appendixes)	**10/63/4/31/3/0/0**
Scientific field, **SF**:	Electrical Engineering
Scientific discipline, **SD**:	Computer Engineering, Engineering of Computer Based Systems
Subject/Key words, **S/KW**:	**Valgrind, dynamic binary analysis, embedded systems, architecture MIPS**
UC	
Holding data, **HD**:	The Library of Faculty of Technical Sciences, Novi Sad, Serbia
Note, **N**:	
Abstract, **AB**:	**The goal of this paper is to describe process of porting tool for dynamic binary analysis to a new architecture. MIPS based Linux OS has been used as a referent platform. The porting effort has indicated major architecture dependencies within the tool. Also, here is presented how Valgrind works and concept of using additional tools based on it.**
Accepted by the Scientific Board on, **ASB**:	
Defended on, **DE**:	
Defended Board, **DB**: President:	
Member:	Menthor's sign
Member, Mentor:	

Zahvalnost

Zahvaljujem se mentoru, Petru Jovanoviću na vođstvu, strpljenju i stručnim savetima prilikom izrade ovog diplomskog-master rada.

Veliko hvala kolegama za nesebičnu pomoć i duge razgovore koji su doveli do razrešenja pojedinih problema, pre svega Aleksandru Simeonov, Vladimiru Mediću, Radovanu Obradoviću i Petru Božinu.

Na kraju bih želeo da se zahvalim roditeljima, bratu i prijateljima na podršci koju su mi pružili.

SADRŽAJ

1. Uvod....1
2. Analiza programa....2
2.1 Statička i dinamička analiza programa....2
2.2 Analiza programskog koda i analiza mašinskog koda....3
3. Velgrind....4
3.1 Bojenje vrednosti....6
3.2 Osnovni blok....6
3.3 Translacija....7
3.4 Model niti POSIX....8
3.5 Sistemski pozivi....9
3.6 Signali....10
3.7 Klijentski zahtevi....10
4. Prilagođavanje Velgrinda arhitekturi MIPS....13
5. Memcheck....18
5.1 Princip rada alata Memcheck....19
5.1.1 Korišćenje neinicijalizovanih vrednosti....21
5.1.2 Korišćenje neinicijalizovane ili neadresirane vrednosti u sistemskom pozivu..22
5.1.3 Nedopušteno oslobađanje memorije....25
5.1.4 Oslobađanje dinamičkog bloka neodgovarajućom funkcijom za oslobađanje memorije 28
5.1.5 Preklapanje izvornog i odredišnog bloka....30
5.1.6 Detekcija curenja memorije.....31
5.2 Izvršni prostori....34

5.3 Struktura alata - Memcheck 35
5.4 Definisanost promenljive 35
5.4.1 V i A biti 36
5.5 Instrumentacija 38
5.6 Karakteristike Memcheck-a 39
6. Cachegrind 43
6.1 Korišćenje Cachegrinda 44
6.2 Cachegrind metapodaci 44
6.3 Instrumentacija 47
6.4 Prikaz statističkih informacija 48
7. Helgrind 49
7.1 Loša upotreba sučelja za programiranje niti POSIX 49
7.2 Potencijalno blokiranje niti 51
7.3 Pristup memoriji bez adekvatnog zaključavanja ili sinhronizacije 52
7.4 Primer pristupa promenljivoj bez adekvatne sinhronizacije 52
7.5 Algoritam detekcije pristupa promenljivoj bez sinhronizacije 54
8. Provera ispravnosti 58
9. Zaključak 61
10. Literatura 63

SPISAK SLIKA

Slika 1: Normalno izvršavanje programa.....5
Slika 2: Izvršavanje programa uz posredstvo alata Velgrind.....5
Slika 3 : Organizacija memorijskih mapa A i V bita 37

SPISAK TABELA

Table 1: Tipovi analize koda....3
Table 2: Translacija koda....8
Table 3 : Primer upotrebe klijentskog zahteva....11
Table 4: Primer upotrebe magične sekvence uz posredstvo Velgrinda....12
Table 5 : Primer upotrebe magične sekvence bez posredstva Velgrinda....12
Table 6: Primer omotača koji se poziva pre sistemskog poziva `sys_time`....15
Table 7: Primer omotača koji se poziva posle izvršenog sistemskog poziva `sys_time`..15
Table 8: Sekvenca instrukcija za prepoznavanje klijentskog zahteva - arhitektura MIPS ..16
Table 9: Primer programa `main.c`....19
Table 10: Primer izlaza za program main.c....21
Table 11: Primer programa main1.c....21
Table 12: Primer izlaza za program main1.c....22
Table 13: Primer programa main1.c....23
Table 14 : Primer izlaza za program `main1.c`....25
Table 15: Primer programa `main.c`....26
Table 16 : Primer izlaza za program `main.c`....28
Table 17: Primer programa `main2.c`....28
Table 18 : Primer izlaza za program `main2.c`....29
Table 19: Primer programa `main3.c`....30
Table 20 : Primer izlaza za program `main3.c`....31
Table 21: Primeri pokazivača na memorijski blok....33
Table 22: Rezime curenja memorije....33

Table 23 : Izveštaj o curenju memorije....34
Table 24: Strukture centralne tabele troškova....46
Table 25: Struktua tabele informacija o instrukcijama....46
Table 26: Primer prikaza greške u programu....51
Table 27: Primer pristupa promenljivoj bez adekvatne sinhronizacije....52
Table 28: Izveštaj Helgrinda za pristup promenljivoj bez sinhronizacije....53
Table 29: „Desilo se pre“ princip....54
Table 30: Primer izlaza iz Helgrinda....56
Table 31: Spisak testova koji padaju....60

SKRAĆENICE

DBI	-*Dynamic binary instrumentation,* dinamička binarna instumentacija
DBA	-*Dynamic binary analysis,* dinamička binarna analiza
ELF	-*Executable and Linkable Format,* izvršni i povezivi format
RICS	-*Reduced instruction set computing,* računar sa smanjenim skupom naredbi
RWX	-*Read, Write, Execute*, prava pristupa: čitaj, piši i izvrši
IR	-*Intermediate Representation,* interna predstava koda
FPU	-*Floating-Point Unit,* jedinica za rad u pokretnom zarezu
SIMD	-*Single Instruction, Multiple Data,* jedna instrukcija više podataka
API	-*Application Programming Interface,* interfejs za programiranje aplikacija

1. Uvod

Pronalaženje grešaka u programima pisanim u bilo kom jeziku predstavlja izazov sa kojim se svakodnevno suočavaju programeri. Korišćenje promenljivih koje nisu definisane je izvor grešaka u programima pisanim u jezicima kao što su C, C++ ili Fortran. Takve greške je veoma lako napraviti, dok je njihovo ponalaženje i otklanjanje veoma teško, i nekada mogu da prođu i godine dok se takve greške ne uoče. Sistemi programske podrške i fizičke arhitekture rastu neverovatnom brzinom, pa je zato i potrebno programerima obezbediti alat koji će im pomoći prilikom analize i verifikacije samog koda.

Alati koji se mogu koristiti za poboljšanje kvaliteta programa, posebno ispravnosti i brzine, su od neprocenjive vrednosti. Mnogi od takvih alata koriste neku vrstu programske analize da otkriju interesantne informacije o samom programu.

Cilj ovog rada je da opiše proces prilagođavanja alata za dinamičku analizu programskog koda Velgrind arhitekturi MIPS, kao i da na tom primeru prikaže one delove ovog alata koji su zavisni od arhitekture na kojoj se izvršavaju. Kao referentna platforma odabran je operativni sistem Linuks koji je baziran na procesorima MIPS. U radu se nalazi opis funkcionisanja Velgrinda, kao i prikaz koncepta upotrebe drugih alata oslonjenih na Velgrind.

2. Analiza programa

2.1 Statička i dinamička analiza programa

Analiza programa se može podeliti u dve grupe, u zavisnosti kada do same analize dolazi.

1. *Statička analiza* obuhvata analizu programskog izvornog koda ili mašinskog koda, bez njegovog pokretanja. Kao jedan od mnogobrojnih alata koji koristi statičku analizu koda jeste i programski prevodilac. On koristi statičku analizu, između ostalog, da bi izvršio proveru ispravnosti tipa podatka. Takođe, alati koji koriste statičku analizu koda mogu da pomognu prilikom identifikacije greške kao i kod vizuelizacije koda. Ovi alati nemaju potrebu da pokreću program da bi izvršili njegovu analizu.
2. *Dinamička analiza* obuhvata analizu korisničkog programa dok se on izvršava. Mnogi alati vrše dinamičku analizu koda, kao što su, na primer, programi za optimizaciju koda ili vizuelni analizatori. Alat za dinamičku analizu koda instrumentalizuje klijentski program sa kodom za analizu. Kod za analizu se ubacuje u originalni kod, a takođe može da sadrži pozive drugih rutina koje se ne nalaze u originalnom kodu. Kod za analizu se pokreće kao deo originalnog programa, tokom njegovog izvršavanja, ne ugrožavajući izvršavanje originalnog koda (osim brzine izvršavanja), ali radeći dodatan posao, kao što je merenje performanse originalnog koda ili otkrivanje grešaka u originalnom kodu. Kod za analizu koda mora da sadrži neku vrstu stanja analize koja se naziva meta-podaci (pojedinačni delovi meta-podataka se nazivaju meta-vrednosti). Meta-podaci su apsolutno krucijalni i predstavljaju samo srce dinamičke analize podataka.

Obe analize su komplementarne. Međutim, dinamička analiza je nešto preciznija jer ona radi sa realnim vrednostima.

2.2 Analiza programskog koda i analiza mašinskog koda

Programska analiza se može podeliti u dve grupe, u zavisnosti od tipa izvornog koda koji se analizira.

1. *Analiza izvornog koda* se odnosi na analizu programa na najvišem nivou, na nivou izvornog koda. Alat koji koristi analizu izvornog koda je programski prevodilac. Ova kategorija obuhvata analizu koja se odnosi na analizu programa koji je direktno izveden iz izvornog koda programa. Analiza izvornog koda se uglavnom vrši u zavisnosti od programskog jezika kojim je pisan izvorni kod i odnosi se uglavnom na: konstrukcije, funkcije, izraze i promenljive u izvornom kodu.
2. *Binarna analiza* obuhvata analizu koda na nivou mašinskog koda, snimljenog ili kao objektni kod (nepovezan) ili kao izvršni kod (povezan). Ova kategorija obuhvata analizu performansi na izvršnom nivou. Binarna analiza se uglavnom vrši nad mašinskim entitetima kao što su: procedure, mašinske instrukcije, registri, i zauzeće memorije.

Kao što je slučaj sa dinamičkom i statičkom analizom i ove dve analize su komplementarne. Analiza izvornog koda je platformski nezavisna, ali zavisna od jezika u kome je pisana; dok je analiza mašinskog koda zavisna od platforme na kojoj se kod izvršava, ali nezavisna od jezika u kome je program pisan. Analiza izvornog koda ima pristup informacijama na visokom nivou, koji je može učiniti efikasnijom; dok analiza mašinskog koda ima pristup nižem nivou informacija (kao što su registri). Definitivna prednost binarne analize je ta što izvorni kod nije potreban, što može biti velika prednost ako program koristi neke biblioteke čiji izvorni kodovi nisu dostupni na određenoj platformi.

	Statička	Dinamička
Izvorni kod	Statička analiza izvornog koda	Dinamička analiza izvornog koda
Mašinski kod	Statička analiza binarnog koda	Dinamička analiza binarnog koda

Table 1: Tipovi analize koda

3. Velgrind

Velgrind je platforma za pravljenje alata za dinamičku binarnu analizu koda. 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, MIPS/Linux, PPC32/Linux, PPC64/Linux, X86/Darwin, AMD64/Darwin (Mac OS X 10.5 and 10.6).

Alat za dinamičku analizu koda se kreira kao dodatak, pisan u C programskom jeziku, na jezgro Velgrinda.

Jezgro Velgrinda + alat koji se dodaje = Alat Velgrinda

Jezgro Velgrinda omogućava izvršavanje klijentskog programa, kao i snimanje izveštaja koji su nastali prilikom analize samog programa.

Alati Velgrinda koriste metodu bojenja vrednosti. Oni zapravo svaki registar i memorijsku vrednost „boje“ (zamenjuju) sa vrednošću koja govori nešto dodatno o originalnoj vrednosti.

Svi Velgrind alati rade na istoj osnovi, iako informacije koje se emituju variraju. Informacije koje se emituju mogu se iskoristiti za otklanjanje grešaka, optimizaciju koda ili bilo koju drugu svrhu za koju je alat dizajniran.

U [1] je pokazan, na primeru Velgrinda, princip rada dinamičke binarne analize kao posebne vrste dinamičke binarne instrumentacije. Alat se umeće u originalan kod na početku, zatim se novi kod prevodi, svaki osnovni blok pojedinačno, koji se kasnije izvršava. Proces prevođenja se sastoji iz raščlanjavanja originalnog mašinskog koda u IR (*intermediate*

representation) koji se kasnije instrumentalizuje sa alatom i ponovo prevodi u novi mašinski kod.

Rezultat svega ovoga naziva se *translacija*, koja se čuva u memoriji i koja se izvršava po potrebi. Jezgro troši najviše vremena na sam proces pravljenja, pronalaženja i izvršavanja *translacije*. Originalni kod se nikada ne izvršava. Jedini problem koji se ovde može dogoditi je ako se vrši translacija koda koji se menja tokom izvršavanja programa.

Slika 1 daje konceptualni pogled na normalno izvršavanje programa, iz ugla klijenta. Klijent može direktno pristupiti korisničkom nivou mašine na kojoj se program izvršava (npr. registirma opšte namene), dok sistemskom nivou može da pristupi samo preko operativnog sistema (OS) koristeći sistemske pozive. Slika 2 pokazuje kakve promene postoje prilikom izvršavanja programa kada se on izvršava posredstvom nekog od alata Velgrinda. Klijentski program i alat su deo istog procesa, samo što alat posreduje u svemu što klijent napravi, dajući potpunu kontrolu klijentu. Jedino sistemski pozivi nisu pod direktnom kontrolom alata, ali i oni su indirektno kontrolisani.

Slika 1: Normalno izvršavanje programa

Slika 2: Izvršavanje programa uz posredstvo alata Velgrind

Postoje mnoge komplikacije koje nastaju prilikom smeštanja dva programa u jedan proces (klijentski program i program alata). Mnogi resursi se dele između ova dva programa, kao što su registri ili memorija. Takođe, alat Velgrinda nikad ne sme da se odrekne totalne kontrole nad izvršavanjem klijentskog programa prilikom izvršavanja sistemskih poziva, signala i niti.

3.1 Bojenje vrednosti

Za proizvoljan graf G, bojenjem (*coloring*) G-a, svakom čvoru grafa se dodeljuje boja na takav način da ne postoje susedni čvorovi u grafu koji imaju istu boju. Bojenje koje koristi k boja naziva se k-coloring, a najmanja vrednost k za dati graf naziva se hromatski-broj (*chromatic number*).

Nakon što je selekcija instrukcija izvršena, u narednom koraku (fazi) virtuelnim registrima dodeljuju se fizički registri. Cilj je da se odredi dodela koja minimizira cenu *spill*-koda.

U toku ove aktivnosti za svaku proceduru se konstruiše *register-interference* graf. Kod *register-interference* grafa čvorovi predstavljaju simboličke registre, a poteg povezuje dva čvora ako je jedan od čvorova u životu, u tački gde je drugi čvor definisan. To znači da poteg između dva čvora ukazuje da su odgovarajuće privremeno promenljive simultano žive pa se zbog, toga ne mogu dodeliti istom registru. Ako ciljni procesor ima k registara, tada graf sa k boja odgovara važećoj dodeli registara. *Spill-code* se generiše kada k-bojenje grafa nije moguće.

Postoje tri karakteristike izvršavanja programa koje su relevantne za metod bojenja vrednosti: (a) stanje programa - S, finalni set lokacija koje sadrže vrednosti (kao što su registri ili korisnički adresni prostor), (b) programske izvršne operacije koje čitaju i upisuju u S, (c) programske izvršne operacije (alokacija i dealokacija) koje čine memorijsku lokaciju aktivnom ili neaktivnom.

Bojenje stanja. Alat koji koristi metod bojenja vrednosti, održava vrednosti iz stanja S', koje sadrži obojene vrednosti za svaku vrednost iz stanja S. U [2] su pokazani svi zahtevi koje Velgrind mora da ispuni da bi obezbedio realizaciju metode bojenja vrednosti.

3.2 Osnovni blok

Velgrind deli originalni kod 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. Svaki kod programa koji se analizira ponovo se prevodi na

zahtev, pojedinačno po osnovnim blokovima, neposredno pre samog izvršavanja osnovnog bloka. Ako uzmemo da su osnovni blokovi klijentskog koda BB1, BB2, ... onda prevedene osnovne blokove obeležavamo sa t(BB1), t(BB2),... Veličina osnovnog bloka je ograničena na maksimalno 60 mašinskih instrukcija. Na procesorima MIPS, instrukcije skoka i grananja imaju takozvano „odloženo izvršavanje“. To znači da se prilikom izvršavanja tih instrukcija izvršava i instrukcija koja se nalazi neposredno iza instrukcije grananja ili skoka. U slučaju da je poslednja (60) instrukcija osnovnog bloka instrukcija grananja, Velgrind učitava i instrukciju koja se nalazi neposredno iza nje (61). Time se omogućava konzistentno izvršavanje programa koji se analizira, kao i u slučaju da se program izvršava bez posredstva Velgrinda. Ukoliko nakon izvršenih 60 instrukcija Velgrind nije naišao na instrukciju grananja, sekvenca instrukcija se deli na dva ili više osnovnih blokova, koji se izvršavaju jedan za drugim.

3.3 Translacija

U nastavku su dati koraci koje Velgrind izvršava prilikom analize programa.

Disasembliranje (razgradnja) - proces prevođenja mašinskog koda u ekvivalentni asemblerski kod. Velgrind vrši prevođenje mašinskog koda u interni skup instrukcija koje se nazivaju međukod instrukcije. Međukod predstavlja redukovan skup instrukcija (RISC).

Optimizacija predstavlja proces koji Velgrind izvršava nakon disasembliranja. Tom prilikom se uklanja višak instrukcija koje su dodate prilikom razgradnje pojedinačnih instrukcija na jednu ili više međukod instrukcija. Praktično, mnoge instrukcije za učitavanje vrednosti iz memorije u registre i obrnuto, koje su dodate prilikom razgradnje, se otklanjaju.

Instrumentacija ubacuje nove međukod instrukcije u kod dobijen optimizacijom međukod instrukcija. Dodate instrukcije se ubacuju da bi se izvršila analiza samog izvornog koda. Dodate instrukcije ne narušavaju konzistentno izvršavanje originalnog koda. Analizu međukod instrukcija vrši alat koji dodaje međukod instrukcije u originalni optimizovani međukod da bi izvršio analizu originalnog programa, u zavisnosti od alata koji je izabran.

Dodela registara vrši zamenu virtuelnih registara, koji su određeni u svakoj međukod instrukciji, sa jednim od pravih registara koji su za to određeni.

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.

Proces disasembliranja izvornog mašinskog koda, instrumentacija, dodela registara i izvršavanje novog koda naziva se translacija.

U [2] je pokazan princip rada Velgrind jezgra koji se sastoji iz: disasembliranja originalnog koda, optimizacije, instrumentacije tako dobijenog alata, kao i same dodele registara procesora i izvršavanje tako dobijenog koda. U sledećoj tabeli dati su koraci prilikom translacije koda kao i koji delovi paketa Velgrind izvršavaju određene korake.

Korak	Izvršava
Deasembliranje	Jezgro Velgrinda
Optimizacija	Jezgro Velgrinda
Instrumentacija	Alat Velgrinda
Alokacija registara	Jezgro Velgrinda
Generisanje koda	Jezgro Velgrinda

Table 2: Translacija koda

Svaka translacija počinje sa umanjenjem vrednosti globalnog brojača. Nakon prevedenih nekoliko hiljada instrukcija ovaj brojač dostiže vrednost nula i Velgrind tada proverava da li su se desili neki neuobičajeni događaji, kao što su signali.

U telu translacije, svaka međukod instrukcija je prevedena nezavisno u mali broj mašinskih instrukcija ciljne platforme (asemblerske kodne rutine). Često je instrukcija koja je generisana iz međukod instrukcije ista onoj u originalnom kodu, ali sa različitim registrima.

3.4 Model niti POSIX

Instrumentalizovani kod može biti pokrenut uz pomoć više odvojenih niti jezgra. Ovo zvuči jednostavno, ali bi u praksi bilo veoma kompleksno i sporo. Unutrašnja struktura podataka jezgra morala bi da poseduje odgovarajuće osobine koje su potrebne da bi niti njima pristupale. Takođe, alat bi morao da zaključa svoje strukture podataka. Za mnoge alate ovo bi značilo da moraju da zaključaju svaki memorijski pristup koji napravi klijentski program, što bi veoma usporilo proces izvršavanja programa. Svaki pristup memoriji u originalnom kodu se translira u pristup originalnoj memoriji, plus pristup obojenoj memoriji. Ne postoje jasne garancije da će, ako više niti originalnog programa pristupa nekoj memorijskoj lokaciji, to isto se desiti i sa obojenom memorijom.

Da bi stavio u stranu ovaj problem, Velgrind podržava samo POSIX model niti. Velgrind obezbeđuje kompatibilne binarne funkcije, koje zamenjuju standardne funkcije iz biblioteke `libpthread`. Ove funkcije, kombinovane sa raspoređivačem u Velgrindu, obezbeđuju paket

niti na korisničkom nivou. Sve aplikacione niti se pokreću na jednoj niti jezgra, i sva komutacija i raspoređivanje niti vrše se pod kontrolom jezgra Velgrinda. Standardne apstrakcije su podržane: semafori, uslovne promenljive itd.

Ovakva šema se pokazuje uspešnom skoro na svim programima koji koriste niti. Ona, takođe, dostavlja alatu informacije o događajima kao što su stvaranje i uništavanje niti, zaključavanje i otključavanje semafora.

3.5 Sistemski pozivi

Aplikacioni 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.

Aplikacija koja želi da koristi neke od resursa, kao što su memorija, procesor ili ulazno/izlazni uređaji, komunicira sa jezgrom operativnog sistema koristeći sistemske pozive. 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 uređaja. Korisnički prostor je deo memorije gde se nalaze sve korisničke aplikacije prilikom njihovog izvršavanja. Korisničke aplikacije mogu da pristupe ulazno/izlaznim uređajima, virtuelnoj memoriji, datotekama i drugim resursima jezgra operativnog sistema koristeći samo sistemske pozive. Sistemski pozivi obezbeđuju spregu između programa koji se izvršava i operativnog sistema. Generalno, realizuju se na asemblerskom jeziku, ali noviji viši programski jezici, poput jezika C i C++, takođe omogućavaju realizaciju sistemskog poziva. Program koji se izvršava može proslediti parametre operativnom sistemu na tri načina:

- prosleđivanjem parametara u registrima procesora;
- postavljanjem parametara u memorijskoj tabeli, pri čemu se adresa tabele prosleđuje u registru procesora;
- postavljanjem parametara na vrh steka (push), koje operativni sistem "skida" (pop).

Sistemski pozivi se izvršavaju bez posredstva Velgrinda, zato što jezgro Velgrinda ne može da prati njihovo izvršavanje u samom jezgru operativnog sistema.

3.6 Signali

Rukovanje signalima po standardu Unix predstavlja specijalan problem za sve alate koji koriste dinamičku binarnu instrumentaciju. Kada aplikacija postavi rukovaoce signalima, predaje jezgru operativnog sistema povratnu adresu, koja se nalazi u aplikacionom prostoru i koja se koristi za dostavljanje signala. Ovo se ne sme dozvoliti, jer se originalni klijentski rukovalac izvršava nepreveden. Ako rukovalac nikada ne skoči na povratnu adresu, nego izvrši `longjmp`, alat će u potpunosti izgubiti kontrolu nad izvršavanjem klijentskog programa. Zato jezgro Velgrinda poseduje mehanizam za presretanje svih `sigaction()` i `sigprocmask()` sistemskih poziva koji se koriste za registraciju rukovaoca signalima. Jezgro Velgrinda čuva adrese određenih rukovalaca signalima i traži od jezgra operativnog sistema da njemu dostavi sve signale.

Jezgro Velgrinda prihvata sve sinhrone signale, kao što je `SIGSEGV`. Kada signal pristigne, jezgro automatski prosledi signal određenoj aplikacionoj niti. Ovo garantuje da će se rukovalac za određeni signal pozvati (ako postoji) pre nego što nit nastavi dalje izvršavanje.

Stvari su još komplikovanije za asinhrone signale zato što ih jezgro Velgrinda blokira. Pomoćna nit jezgra, koja je određena za prijem asinhronih signala, čeka da se signal desi. Kada asinhroni signal stigne, odgovarajuća pomoćna nit (koja je određena od strane jezgra) hvata taj signal i predaje ga raspoređivaču u formi u kojoj on može da ga prepozna. Svakih nekoliko hiljada osnovnih blokova, raspoređivač proverava da li mu je neka pomoćna nit predala informaciju o asinhronom signalu. Ako jeste, on predaje signal odgovarajućoj aplikacionoj niti na isti način kao i sinhroni signal, a pomoćna nit se vraća u stanje čekanja asinhronog signala. Ako je neka pomoćna nit blokirana u sistemskom pozivu, sistemski poziv će biti blokiran na odgovarajući način.

3.7 Klijentski zahtevi

Velgrind poseduje mehanizam koji omogućava klijentskom programu da prosledi poruke ili upite jezgru ili alatu. Ovo je omogućeno ubacivanjem u klijentski kod kratke sekvence instrukcija koje efektivno ne rade ništa.

Klijentski zahtevi mogu biti ubačeni u program pisan u bilo kom jeziku. Klijentski program mora da uključi datoteku zaglavlja i mora biti ponovo preveden sa ubačenim klijentskim zahtevom u sam kod programa, ali ne mora da bude povezan sa dodatnim bibliotekama. Pošto *magična sekvenca* ne vrši nikakvu promenu vrednosti u registrima, novi

program koji se dobije može se izvršiti normalno. Jedina razlika je ta što se novi program može izvršavati sporije zbog dodatih instrukcija.

Na osnovu instrukcije koja se nalazi odmah iza magične sekvence, Velgrind određuje koji klijentski zahtev treba da izvrši.

```
#include <stdio.h>
#include "valgrind.h"
// Originalna funkcija
__attribute__((noinline))
void original ( void )
{
   printf("originalna funkcija\n");
}

void I_WRAP_SONAME_FNNAME_ZU(NONE, original) ( void )
{
   OrigFn fn;
   VALGRIND_GET_ORIG_FN(fn);
   printf("omotac-pre\n");
   CALL_FN_v_v(fn);
   printf("omotac-post\n");
}

int main ( void )
{
   printf("start\n");
   original();
   return 0;
}
```

Table 3 : Primer upotrebe klijentskog zahteva

U tabeli 3 je prikazan program koji poziva funkciju `original()`. Međutim, prilikom izvršavanja programa posredstvom Velgrinda, poziva se funkcija `I_WRAP_SONAME_FNAME_ZU()`, koja poziva funkciju `original()`. Kada Velgrind naiđe na magičnu sekvencu, zna da se radi o klijentskom zahtevu i poziva odgovarajuću funkciju (tabela 4). Prilikom izvršavanja programa bez posredstva Velgrinda (tabela 5), procesor će izvršiti magičnu sekvencu, koja nema efekta na normalno izvršavanje programa, i pozvaće funkciju `original()`.

```
dejan@mipssupport-06:~/valgrind/include$ valgrind --tool=none -q ./test.out
start
omotac-pre
originalna funkcija
omotac-post
```

Table 4: Primer upotrebe magične sekvence uz posredstvo Velgrinda

```
dejan@mipssupport-06:~/valgrind/include$ ./test.out
start
originalna funkcija
```

Table 5 : Primer upotrebe magične sekvence bez posredstva Velgrinda

4. Prilagođavanje Velgrinda arhitekturi MIPS

Prilikom prilagođavanja Velgrinda za arhitekturu MIPS definisani su svi registri koji se nalaze na arhitekturi MIPS (klijentski registri), izabrani su fizički registri koje će Velgrind koristiti prilikom izvršavanja prevedenog koda (v1, a0-a3, t0-t7, s0-s6), kao i registri u kojima će se uvek nalaziti pokazivač na stanje klijentskih registara (s7), pokazivač na originalan kod (v0), pokazivač na prevedeni kod (t9) i pokazivač na sledeći blok koji treba izvršiti (t8).

Napisan je disasembler koji svaku instrukciju pretvara u jednu ili više međukod instrukcija. Međukod instrukcije za svaku originalnu MIPS instrukciju u potpunosti ažuriraju stanje klijentskog programa u memoriji: klijentsko stanje se dobavlja u virtuelne registre, vrše se operacije nad njima, a zatim se snima u klijentsku memoriju.

Predstavljanje koda sa redukovanim setom instrukcija ima dve velike prednosti. Prvo, mnogo je jednostavniji od MIPS instrukcijskog skupa, tako da alat ima mnogo manje slučajeva kada vrši samu instrumentaciju; takođe, instrukcije za upis i čitanje iz memorije su rastavljene na niz prostih instrukcija, tako da alat može veoma lako da detektuje svako pisanje ili čitanje sa određene memorijske lokacije. Drugo, on rastavlja neke složenije MIPS instrukcije (npr. MSUB, MADD, EXT itd.) na više jednostavnih instrukcija, omogućavajući analizu međurezultata. Iako se izvorni kod predstavlja redukovanim skupom instrukcija, redukovani skup instrukcija poseduje i neke instrukcije koje su karakteristične za MIPS arhitekturu. Neke instrukcije, kao što su `RDHWR` (*Read Hardware Register*) nemoguće je predstaviti preko jedne ili više međukod instrukcija. U tom slučaju se definišu blokovi asemblerskih instrukcija koji direktno izvršavaju instrukcije koje je nemoguće predstaviti preko međukod instrukcija. U slučaju da neka od ovih instrukcija snima rezultat u registre ili memoriju procesora MIPS, potrebno je to isto uraditi sa klijentskim registrima ili memorijom procesora MIPS u samom Velgrindu.

Napisan je asembler procesora MIPS koji sve međukod instrukcije, koje su generisane iz originalnog klijentskog koda i međukod instrukcije koje su dodate prilikom instrumentacije samog koda, prevodi u asemblerske instrukcije procesora MIPS.

Sistemski pozivi se izvršavaju bez posredstva Velgrinda, zato što jezgro Velgrinda ne može da prati njihovo izvršavanje u samom jezgru operativnog sistema. Da bi se sistemski pozivi izvršavali korektno na procesorima baziranim na arhitekturi MIPS, prilikom analize koda u Velgrind je potrebno dodati asemblerski blok koji izvršava raspoređivanje parametara sistemskog poziva u registre ili na stek memoriju. Takođe, taj asemblerski blok smešta i broj sistemskog poziva u registar koji je za to predviđen (registar v0), te prilikom uspešnog izvršenja sistemskog poziva, povratne vrednosti snimi u odgovarajuće klijentske registre. Kada se desi sistemski poziv, kontrola se predaje raspoređivaču, koji preduzima sledeće korake:

1. snima alatov pokazivač na stek memoriju;
2. kopira stanje klijentskih registara u fizičke registre procesora, osim programskog brojača;
3. izvršava sistemski poziv;
4. kopira novo stanje klijentskog programa nazad u memoriju, osim programskog brojača;
5. vraća staro stanje alatovog pokazivača na stek memoriju.

Velgrind kopira klijentski pokazivač na stek memoriju, tako da se sam sistemski poziv izvršava na klijentskom pokazivaču steka, kao što bi se izvršavao i bez posredstva Velgrinda.

Kada program pozove sistemski poziv, npr. čitanje sadržaja neke datoteke, kontrola se predaje jezgru Linuksa, koji obrađuje sistemski poziv, i vraća ponovnu kontrolu programu koji je pozvao sistemski poziv. Problem je u tome što jezgro Linuksa prilikom izvršavanja nekih sistemskih poziva menja vrednost pojedinih delova memorije, pa je potrebno o promeni stanja memorije obavestiti i sam alat sa kojim se analizira program. Funkcije koje rukuju obradom sistemskih poziva u Velgrindu nazivaju se omotači. Omotači imaju dve funkcije:

1. Obaveštavaju Velgrind alat šta će se desiti pre izvršavanja sistemskog poziva. Na taj način alat može da izvrši proveru pre samog izvršenja sistemskog poziva, npr. može da proveri da li je memorija u koju će se upisati nova vrednost prilikom izvršavanja sistemskog poziva zauzeta.
2. Obaveštava alat šta se desilo nakon izvršenja sistemskog poziva. Na ovaj način Velgrind može da osveži vrednosti klijentskog programa koga analizira, npr. upis novih vrednosti u memoriju.

U nastavku je dat primer funkcije omotača za sistemski poziv koji se nalazi u mnogim programima na sistemima Unix.

```
PRE(sys_time)
{
   /* time_t time(time_t *t); */
   PRINT("sys_time ( %p )",ARG1);
   PRE_REG_READ1(long, "time", int *, t);
   if (ARG1 != 0) {
      PRE_MEM_WRITE( "time(t)", ARG1, sizeof(vki_time_t) );
   }
}
```

Table 6: Primer omotača koji se poziva pre sistemskog poziva `sys_time`

Prva stvar koja se dogodi pre nego što se sam sistemski poziv izvrši nalazi se u PRE() funkciji. Na početku PRE() funkcije poziva se PRINT() makro. PRINT() makro implementira podršku za *--trace-syscall* opciju komandne linije. Naredni makro govori alatu tip sistemskog poziva, da se sistemskom pozivu prosleđuje jedan parametar, tip parametra koji se prosleđuje i da se vrednost parametra nalazi u registru:

```
PRE_REG_READ1(long, "time", int *, t);
```

U nastavku se alat obaveštava da u slučaju da parametar nije pokazivač sa vrednošću nula, sistemski poziv će upisati novu vrednost u memoriju na koju pokazuje prosleđeni pokazivač:

```
if (ARG1 != 0) {
   PRE_MEM_WRITE( "time", ARG1, sizeof(vki_time_t) );
}
```

```
POST(sys_time)
{
   if (ARG1 != 0) {
      POST_MEM_WRITE( ARG1, sizeof(vki_time_t) );
   }
}
```

Table 7: Primer omotača koji se poziva posle izvršenog sistemskog poziva `sys_time`

Na kraju, veoma važan deo obrade sistemskog poziva je POST() funkcija (tabela 7). Ona obaveštava alat, samo u slučaju da je sistemski poziv uspešno izvršen, da je memorija prepisana.

```
if (ARG1 != 0) {
   POST_MEM_WRITE( ARG1, sizeof(vki_time_t) );
```

```
}
```

Sistemski pozivi koji obuhvataju podelu resursa, kao što je memorija (npr. `mmap()`, `mprotect()`, `brk()`), posebno se proveravaju da se ne bi desio konflikt sa Velgrindom. Velgrind obezbeđuje mehanizam za zaustavljanje sistemskih poziva, kao i za njihovo ponovno pokretanje.

Prilikom analize programa koji kreiraju više od jedne izvršne niti, potrebno je napisati asemblerski blok koji će u potpunosti zameniti sistemski poziv `clone` uz pomoć kojeg se na procesorima MIPS kreira još jedna nit, koja je identična niti roditelj.

Prilikom analize nekih programa koji pozivaju neke od funkcija kao što su `malloc()`, `new()` ili `pthread_create()`, Velgrind pronalazi veliki broj lažnih grešaka. Razlog tome je što su te funkcije veoma optimizovane i specijalno prilagođene da se izvrše najbrže moguće na određenoj platformi. U tom slučaju, Velgrind ove funkcije menja sa svojim funkcijama koje u suštini rade istu stvar, kao i originalne funkcije. Na taj način je omogućeno lakše praćenje i analiza izvršavanja gore pomenutih funkcija.

Velgrind poseduje mehanizam koji omogućava klijentskom programu da prosledi poruke ili upite jezgru ili alatu. Ovo je omogućeno ubacivanjem u klijentski kod kratke sekvence instrukcija koje efektivno ne rade ništa.

```
addiu $t7, $t7, 13
addiu $t7, $t7, 29
addiu $t7, $t7, 3
addiu $t7, $t7, -45
```

Table 8: Sekvenca instrukcija za prepoznavanje klijentskog zahteva - arhitektura MIPS

Kao što se u tabeli 8 može videti, na kraju sekvence instrukcija (tzv. magične sekvence) vrednost u registru t7 je ista kao i vrednost pre sekvence instrukcija. Kada Velgrind otkrije gore prikazanu sekvencu instrukcija tokom disasembliranja programa, on predaje kontrolu preko raspoređivača rukovaocu klijentskim zahtevima. Argumenti mogu biti prosleđeni klijentskom zahtevu, a povratna vrednost može biti preuzeta. Takođe, potrebno je snimiti vrednost globalnog pokazivača (registar $gp), kao i vrednost povratne adrese (registar $ra) na stek memoriju neposredno pre magične sekvence. Neposredno posle magične sekvence potrebno je vratiti stare vrednosti globalnog pokazivača i povratne adrese. Razlog tome je što se ove vrednosti mogu promeniti u Velgrindovim verzijama funkcija sa kojim zamenjujemo originalne. Svaki klijentski zahtev poseduje kod koji odlučuje da li klijentski zahtev treba proslediti jezgru Velgrinda ili alatu. Ovo omogućava da klijentski zahtevi, koji su namenjeni različitim alatima, budu ubačeni u

jedan program. Klijentski zahtevi koji se odnose na alate koji nisu trenutno u upotrebi se ignorišu.

Na osnovu instrukcije koja se nalazi odmah iza magične sekvence, Velgrind određuje koji klijentski zahtev treba da izvrši.

Na ovaj način je omogućeno da se izvrši preusmeravanje poziva nekih funkcija kao što su `malloc()`, `new()` ili `pthread_create()` sa Velgrindovim verzijama istih funkcija, radi lakše analize samog programa.

Velgrind upravlja svojom memorijom izbegavajući upotrebu funkcije `malloc`. Jezgro, takođe, mora da vodi računa da memorija, koja je alocirana za alat, stane u adresni prostor koji je predviđen za alat. Jezgro obezbeđuje alokator koji koristi *`mmap()`* funkciju, da alocira blok memorije.

Jedna od karakteristika Velgrinda jeste da je zavistan od veličine memorijske stranice za ciljnu platformu. Prilikom mapiranja memorije uz pomoć sistemskog poziva `mmap()` ili `mmap2()`, Velgrind kao parametre sistemskog poziva prosleđuje početnu adresu u memoriji koju želi da zauzme i veličinu memorije, koja predstavlja umnožak broja koji predstavlja veličinu stranice. U slučaju da veličina memorije, koju je potrebno mapirati, nije umnožak broja koji predstavlja veličinu stranice, nije moguće izvršiti mapiranje memorije i sistemski poziv će vratiti podatak o nastaloj grešci. Zato je prilikom procesa prevođenja Velgrinda dodata opcija konfigurisanja sa kojom se definiše veličina memorijske stranice za arhitekturu MIPS (veličine koje se mogu koristiti su 4k, 16k ili 64k).

5. Memcheck

Memcheck je dinamički alat za analizu koda koji analizira korišćenje memorije dok se program izvršava. Memcheck proverava četiri vrste memorijskih grešaka.

Prvo, on prati adresiranje svih memorijskih lokacija, ažuriranjem informacija o tome da li je neka memorijska zona alocirana ili oslobođena. Uz pomoć ovih informacija, on može da otkrije sve koji pristupaju neadresiranoj memorijskoj zoni.

Drugo, on prati svu dinamičku memoriju koja je zauzeta sa `malloc()` ili sa `new()`. Uz pomoć ove informacije može da otkrije loše ili višestruko oslobađanje memorije, kao i da otkrije curenje memorije kada se program završi.

Treće, on proverava da li se memorijski blokovi, koji se isporučuju kao parametri funkcije, ne preklapaju.

Četvrto, vrši proveru definisanosti: prati svaki bit podatka koji se nalazi u registru ili u memoriji. Uz pomoć ove informacije može da otkrije greške koje nastaju kao posledica korišćenja nedefinisanih vrednosti i to sa rezolucijom na nivou bita.

Zato što vrši analizu mašinskog koda, a ne izvornog koda, Memcheck ima sledeće osobine:

- Široka primenjenost: radi sa programima pisanim u bilo kom jeziku.
- Totalna pokrivenost: svi delovi klijentskog koda se izvršavaju pod kontrolom Memcheck-a, uključujući i dinamički povezane biblioteke, iako izvorni kod nije dostupan na ciljnoj platformi.
- Jednostavna upotreba: za razliku od drugih alata nije potrebno da se program pre analize priprema (prevodi) na bilo koji način.

5.1 Princip rada alata Memcheck

Memcheck je veoma lak za korišćenje. Kao primer, uzmimo program `main.c` prikazan u tabeli 9. On sadrži tri nedefinisane vrednosti koje uzrokuju grešku.

```
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;
}
```

Table 9: Primer programa `main.c`

Da bi proverio preveden program `main.c` korisnik treba u terminalu da ukuca:

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

`--tool=` opcija određuje koji alat će Velgrind paket koristiti. Program koji radi pod kontrolom Memcheck-a obično je 20-100 puta sporiji nego kad se izvšava samostalno, zbog translacije koda. Izlaz programa je uvećan za izlaz koji dodaje sam Memcheck, koji se ispisuje na standardnom izlazu za greške. Izlaz može biti preusmeren na datoteku ili na utičnicu sa opcijom komandne linije.

Tabela 10 prikazuje rezultirajući izlaz. Prve tri linije se štampaju prilikom pokretanja alata Memcheck. Srednji deo prikazuje dve poruke o greškama koje je Memcheck pronašao. Poslednja linija se štampa prilikom završetka rada alata i predstavlja sumu svih grešaka koje je alat otkrio.

Svaka linija izlaza iz Memcheck-a je prikazana sa prefiksom koji predstavlja ID klijenta, 23216 u ovom slučaju.

Dve poruke o greškama su veoma precizne. Obe govore o tome da je 16 i 19 bajta memorije izgubljeno. U cilju prikazivanja tačnog broja linije, prilikom štampanja stanja steka, klijentski program mora biti preveden sa opcijom za dodavanje informacija za pomoć pri otkrivanju grešaka. Ako program nije preveden sa ovom opcijom, lociranje dela koda gde je nastala greška je manje precizno. U našem slučaju, program je preveden sa opcijom za dodavanje informacija za pomoć pri otkrivanju grešaka, pa možemo tačno videti liniju u programu gde nastaje greška. U našem konkretnom slučaju to su linije 15 i 9, što i jesu linije u `main.c` gde pozivamo funkciju `malloc()`.

```
dejan@mipssupport-06:~$ valgrind --leak-check=yes ./main.out
==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 0xffffff90 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 0x484AA90: malloc (vg_replace_malloc.c:212)
==6942==    by 0x400683: main (main.c:11)
==6942==
==6942== 16 bytes in 1 blocks are definitely lost in loss record 2 of 3
==6942==    at 0x484AA90: 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
```

```
==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)
```

Table 10: Primer izlaza za program main.c

5.1.1 Korišćenje neinicijalizovanih vrednosti

U tabeli 11 je dat primer programa u kome koristimo neinicijalizovanu promenljivu. Greška o korišćenju neinicijalizovane vrednosti se generiše kada program koristi promenljive čije vrednosti nisu inicijalizovane, drugim rečima nedefinisane. Na slici 8 prikazan je izlaz iz Velgrinda u slučaju kada se otkrije korišćenje nedefinisane promenljive.

Kao što se u tabeli 12 vidi, nedefinisana vrednost se koristi u funkciji `main()` (`main1.c` - četvrta linija), prilikom poziva `printf()` funkcije. Tačnije, promenljiva koja se prosleđuje funkciji `printf()` nije definisana.

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

Table 11: Primer programa main1.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)
```

Table 12: Primer izlaza za program main1.c

Važno je da se razume da program može da kopira nedefinisanu vrednost više puta. Memcheck sve ovo prati, beleži podatke o tome, ali ne prijavljuje grešku. Memcheck prijavljuje grešku samo kada program koristi nedefinisane vrednosti na način da od te vrednosti zavisi dalji tok programa ili kada treba korisniku programa prikazati tu vrednost nedefinisane promenljive. U ovom primeru x je nedefinisano. Memcheck prati prosleđivanje promenljive funkciji `printf()` i `vprintf()`, ali ne prijavljuje grešku. Međutim, kada se u funkciji `vprintf()` vrednost nedefinisane promenljive treba prikazati korisniku programa, Memcheck prijavljuje grešku.

Izvori nedefinisanih podataka:

- Lokalne promenljive u procedurama koje nisu definisane, kao u primeru prikazanom gore.
- Čitanje sadržaja dinamičke memorije (koja je alocirana sa `malloc()`, `new()` itd.), pre nego što se u nju upišu neke vrednosti.

Da bi videli glavni izvor korišćenja nedefinisanih vrednosti u programu, dodajemo opciju `--trace-origins=yes` prilikom pokretanja Memcheck-a. Ova opcija čini izvršavanje Memcheck-a sporijim, ali omogućava Memcheck-u da nađe glavni izvor korišćenja nedefinisane vrednosti.

5.1.2 Korišćenje neinicijalizovane ili neadresirane vrednosti u sistemskom pozivu

Memchek prati sve parametre sistemskog poziva:

- Proverava sve parametre pojedinačno, bez obzira da li su inicijalizovani.
- Takođe, proverava da li sistemski poziv treba da čita iz memorije koja je definisana u programu. Memcheck proverava da li je cela memorija adresirana i inicijalizovana.

- Ako sistemski poziv treba da piše u memoriju, Memcheck proverava da li je i ta memorija adresirana.

Posle sistemskog poziva Memcheck precizno ažurira informacije o promeni u memoriji koje su postavljene u sistemskom pozivu.

U sledećoj tabeli je dat primer poziva sistemskog poziva sa neispravnim parametrima.

```
#include <stdlib.h>

#include <unistd.h>

int main( void )

{

    char* arr = malloc(10);

    int* arr2 = malloc(sizeof(int));

    write( 1/*stdout */, arr, 10 );

    exit(arr2[0]);

}
```

Table 13: Primer programa main1.c

Prilikom analize programa prikazanog u tabeli 13 dobijamo izveštaj prikazan u tabeli 14. Kao što možemo videti na tabeli 14, Memcheck je prikazao informaciju o korišćenju neinicijalizovanih vrednosti u sistemskim pozivima. Prva greška prikazuje da parametar `buf` sistemskog poziva `write()` pokazuje na neinicijalizovan podatak. 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.

```
dejan@mipssupport-06:~$ valgrind --leak-check=full --show-
reachable=yes --track-origins=yes ./main1.out
==7069== Memcheck, a memory error detector
==7069== Copyright (C) 2002-2010, and GNU GPL'd, by Julian Seward et
al.
==7069== Using Valgrind-3.7.0.SVN and LibVEX; rerun with -h for
copyright info
==7069== Command: ./test2.out
==7069==
==7069== Syscall param write(buf) points to uninitialised byte(s)
==7069==    at 0x493F7F0: write (in /lib/libc-2.11.2.so)
==7069==    by 0x4006C3: main (test2.c:7)
==7069==  Address 0x49e0030 is 0 bytes inside a block of size 10
alloc'd
==7069==    at 0x484AA90: malloc (vg_replace_malloc.c:212)
==7069==    by 0x40069B: main (test2.c:5)
==7069==  Uninitialised value was created by a heap allocation
==7069==    at 0x484AA90: malloc (vg_replace_malloc.c:212)
==7069==    by 0x40069B: main (test2.c:5)
==7069==
==7069== Syscall param exit_group(status) contains uninitialised
byte(s)
==7069==    at 0x49127B8: _Exit (in /lib/libc-2.11.2.so)
==7069==    by 0x4897DE7: ??? (in /lib/libc-2.11.2.so)
==7069==  Uninitialised value was created by a heap allocation
==7069==    at 0x484AA90: malloc (vg_replace_malloc.c:212)
==7069==    by 0x4006AB: main (test2.c:6)
==7069==
==7069==
==7069== HEAP SUMMARY:
```

```
==7069==     in use at exit: 14 bytes in 2 blocks
==7069==   total heap usage: 2 allocs, 0 frees, 14 bytes allocated
==7069==
==7069== 4 bytes in 1 blocks are still reachable in loss record 1 of 2
==7069==    at 0x484AA90: malloc (vg_replace_malloc.c:212)
==7069==    by 0x4006AB: main (test2.c:6)
==7069==
==7069== 10 bytes in 1 blocks are still reachable in loss record 2 of
2
==7069==    at 0x484AA90: malloc (vg_replace_malloc.c:212)
==7069==    by 0x40069B: main (test2.c:5)
==7069==
==7069== LEAK SUMMARY:
==7069==    definitely lost: 0 bytes in 0 blocks
==7069==    indirectly lost: 0 bytes in 0 blocks
==7069==      possibly lost: 0 bytes in 0 blocks
==7069==    still reachable: 14 bytes in 2 blocks
==7069==         suppressed: 0 bytes in 0 blocks
==7069==
==7069== For counts of detected and suppressed errors, rerun with: -v
==7069== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 14 from
5)
```

Table 14 : Primer izlaza za program `main1.c`

5.1.3 Nedopušteno oslobađanje memorije

U tabeli 15 je dat primer programa u kome nelegalno oslobađamo memoriju. Memcheck prati svako alociranje memorije koje program napravi upotrebom funkcija `malloc/new`, tako da on uvek poseduje informaciju da li su argumenti koji se prosleđuju funkcijama

`free/delete` legitimni ili ne. U našem primeru `main.c`, program oslobađa istu memorijsku zonu dva puta. Izveštaj o nedopuštenom oslobađanju memorije prikazan je u tabeli 16.

Kao što možemo videti u tabeli 16, Memcheck nam je prijavio da je program pokušao dva puta da oslobodi neku memorijsku zonu. Takođe, Memcheck će nam prijaviti i ako program pokuša da oslobodi memorijsku zonu preko pokazivača koji ne pokazuje na početak dinamičke memorije.

```
int main()

{

    char *p;

    // Allocation #1 of 19 bytes

    p = (char) malloc(19);

    // Allocation #2 of 12 bytes

    p = (char) malloc(12);

    free(p);

    free(p);

    // Allocation #3 of 16 bytes

    p = (char) malloc(16);

    return 0;

}
```

Table 15: Primer programa `main.c`

```
dejan@mipssupport-06:~$ valgrind --leak-check=full --show-reachable=yes --track-origins=yes
./main.out

==7103== Memcheck, a memory error detector

==7103== Copyright (C) 2002-2010, and GNU GPL'd, by Julian Seward et al.

==7103== Using Valgrind-3.7.0.SVN and LibVEX; rerun with -h for copyright info

==7103== Command: ./main.out

==7103==

==7103== Invalid free() / delete / delete[] / realloc()

==7103==    at 0x484966C: free (vg_replace_malloc.c:320)

==7103==    by 0x40069B: main (main.c:13)

==7103==  Address 0xffffff90 is not stack'd, malloc'd or (recently) free'd

==7103==

==7103== Invalid free() / delete / delete[] / realloc()

==7103==    at 0x484966C: free (vg_replace_malloc.c:320)

==7103==    by 0x4006A7: main (main.c:14)

==7103==  Address 0xffffff90 is not stack'd, malloc'd or (recently) free'd

==7103==

==7103==

==7103== HEAP SUMMARY:

==7103==     in use at exit: 47 bytes in 3 blocks

==7103==   total heap usage: 3 allocs, 2 frees, 47 bytes allocated

==7103==

==7103== 12 bytes in 1 blocks are definitely lost in loss record 1 of 3

==7103==    at 0x484AA90: malloc (vg_replace_malloc.c:212)

==7103==    by 0x400683: main (main.c:12)

==7103==

==7103== 16 bytes in 1 blocks are definitely lost in loss record 2 of 3

==7103==    at 0x484AA90: malloc (vg_replace_malloc.c:212)

==7103==    by 0x4006B3: main (main.c:16)

==7103==

==7103== 19 bytes in 1 blocks are definitely lost in loss record 3 of 3

==7103==    at 0x484AA90: malloc (vg_replace_malloc.c:212)

==7103==    by 0x40066B: main (main.c:9)

==7103==

==7103== LEAK SUMMARY:

==7103==    definitely lost: 47 bytes in 3 blocks

==7103==    indirectly lost: 0 bytes in 0 blocks
```

```
==7103==      possibly lost: 0 bytes in 0 blocks

==7103==    still reachable: 0 bytes in 0 blocks

==7103==         suppressed: 0 bytes in 0 blocks

==7103==

==7103== For counts of detected and suppressed errors, rerun with: -v

==7103== ERROR SUMMARY: 5 errors from 5 contexts (suppressed: 14 from 5)
```

Table 16 : Primer izlaza za program `main.c`

5.1.4 Oslobađanje dinamičkog bloka neodgovarajućom funkcijom za oslobađanje memorije

U tabeli 17 prikazan je primer programa u kome je memorijski blok alociran sa funkcijom `new[]`, a pogrešno oslobođen sa funkcijom `free()`. Tabela 18. prikazuje izveštaj koji Memcheck pravi prilikom analize programa `main2.c`

```
int main()

{

    char *p = new char[10];

    free(p);

    return 0;

}
```

Table 17: Primer programa `main2.c`

```
dejan@mipssupport-06:~$ valgrind --leak-check=full --show-reachable=yes --track-origins=yes
./main2.out

==7136== Memcheck, a memory error detector

==7136== Copyright (C) 2002-2010, and GNU GPL'd, by Julian Seward et al.

==7136== Using Valgrind-3.7.0.SVN and LibVEX; rerun with -h for copyright info

==7136== Command: ./main2.out

==7136==

==7136== Mismatched free() / delete / delete []

==7136==    at 0x484966C: free (vg_replace_malloc.c:320)

==7136==    by 0x40073B: main (main2.c:10)

==7136==  Address 0x4bdc030 is 0 bytes inside a block of size 10 alloc'd

==7136==    at 0x4849BD8: operator new[](unsigned int) (vg_replace_malloc.c:263)

==7136==    by 0x40072B: main (main2.c:7)

==7136==

==7136==

==7136== HEAP SUMMARY:

==7136==     in use at exit: 0 bytes in 0 blocks

==7136==   total heap usage: 1 allocs, 1 frees, 10 bytes allocated

==7136==

==7136== All heap blocks were freed -- no leaks are possible

==7136==

==7136== For counts of detected and suppressed errors, rerun with: -v

==7136== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 23 from 8)
```

Table 18 : Primer izlaza za program `main2.c`

U programskom jeziku C++ veoma je važno da se memorija oslobodi na način na koji je zauzeta.

- Ako je memorija zauzeta sa `malloc, calloc, realloc, valloc ili memalign`, mora se osloboditi sa `free`.
- Ako je memorija zauzeta sa `new`, mora se osloboditi sa `delete`.
- Ako je memorija zauzeta sa `new[]`, mora se osloboditi sa `delete[]`.

Linuks sistem uopšte se ne buni ako se mešaju ovakve stvari, ali se može desiti da isti program koji se prevede za neku drugu platformu, Solaris na primer, doživi krah.

5.1.5 Preklapanje izvornog i odredišnog bloka

Sledeće C bibliotečke funkcije kopiraju podatke iz jednog memorijskog bloka u drugi (ili nešto slično tome): `memcpy, strcpy, strncpy, strcat, strncat`. Nije dozvoljeno preklapanje izvornih i odredišnih blokova memorije na koje pokazuju pokazivači `src` i `dst`. Po POSIX standardu „Ako se kopiranje vrši između objekata koji se preklapaju, ponašanje je nedefinisano“.

Na primer, za program prikazan u tabeli 19. Memcheck će prijaviti izveštaj prikazan u tabeli 20.

```
int main() {

    char buf[10];

    char *p, *r;

    p = buf + 4;

    r = memcpy(p, buf, 10);

    return 0;

}
```

Table 19: Primer programa `main3.c`

Ne želimo da vršimo kopiranje između dva bloka koja se preklapaju, jer tada vršimo upisivanje pogrešnih vrednosti u blokove.

```
dejan@mipssupport-06:~$ valgrind --leak-check=full --show-reachable=yes --track-origins=yes
./main3.out

==7143== Memcheck, a memory error detector

==7143== Copyright (C) 2002-2010, and GNU GPL'd, by Julian Seward et al.

==7143== Using Valgrind-3.7.0.SVN and LibVEX; rerun with -h for copyright info

==7143== Command: ./main3.out

==7143==

==7143== Source and destination overlap in memcpy(0x7ec186d4, 0x7ec186d0, 10)

==7143==    at 0x484CA10: memcpy (mc_replace_strmem.c:632)

==7143==    by 0x400653: main (main3.c:13)

==7143==

==7143==

==7143== HEAP SUMMARY:

==7143==     in use at exit: 0 bytes in 0 blocks

==7143==   total heap usage: 0 allocs, 0 frees, 0 bytes allocated

==7143==

==7143== All heap blocks were freed -- no leaks are possible

==7143==

==7143== For counts of detected and suppressed errors, rerun with: -v

==7143== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 14 from 5)
```

Table 20 : Primer izlaza za program `main3.c`

5.1.6 Detekcija curenja 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` podešena na odgovarajući način, za svaki neoslobođen blok Memcheck određuje da li je moguće pristupiti tom bloku preko pokazivača.

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.

Postoje tri načina da saznamo da li postoje pokazivači koji pokazuju na unutrašnjost nekog memorijskog bloka:

- Postojao je pokazivač koji je pokazivao na početak memorijskog bloka, ali je namerno (ili nenamerno) pomeren da pokazuje na unutrašnjost bloka.
- Ako postoji neželjena vrednost u memoriji, koja je u potpunosti nepovezana i slučajna.
- Ako postoji pokazivač na niz C++ objekata (koji poseduju destruktore) koji su alocirani sa `new[]`. U ovom slučaju, neki kompajleri čuvaju „magični kolačić“ koji sadrži dužinu niza od početka bloka.

U tabeli 21 prikazano je 9 mogućih slučajeva kada pokazivači pokazuju na neke memorijske blokove. Svaki mogući slučaj kada pokazivač pokazuje na neki memorijski blok može se predstaviti sa jednim od prikazanih 9 slučaja. Memcheck objedinjuje neke od ovih slučajeva, tako da dobijamo rezultirajuće četiri kategorije.

- „Još uvek dostupan“. Ovo pokriva primere 1 i 2 u tabeli 21. Pokazivač koji pokazuje na početak bloka ili više pokazivača koji pokazuju na početak bloka su pronađeni. Zato što postoje pokazivači koji pokazuju na memorijsku zonu koja nije oslobođena, programer može da oslobodi memorijsku zonu neposredno pre završetka izvršavanja programa.
- „Definitivno izgubljena“. Ovo pokriva slučaj 3 prikazan u tabeli 21. To znači da nije moguće naći pokazivač koji pokazuje na memorijski blok. Blok je proglašen izgubljenim, zato što programer ne može da oslobodi zauzetu memoriju pre završetka izvršavanja programa, jer ne postoji pokazivač na zauzetu memoriju.
- „Indirektno izgubljen“. Ovo pokriva slučajeve 4 i 9 u tabeli 21. To znači da je blok izgubljen, ne zato što ne postoji pokazivač koji pokazuje na njega, već zato što su svi blokovi koji ukazuju na njega sami izgubljeni. Na primer, ako imamo binarno stablo i korenski čvor je izgubljen, sva deca čvorovi su indirektno izgubljena. 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 sprecificirano `--show-reachable=yes`.
- „Moguće izgubljen“. Ovo pokriva slučajeve 5 do 8 u tabeli 21. Ovo znači da je pronađen jedan ili 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.

```
(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 pokazivača

  AAA, BBB memorijski blokovi u dinamičkoj memoriji
```

Table 21: Primeri pokazivača na memorijski blok

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

Na sledećoj slici dat je rezime curenja memorije koji ispisuje Memcheck. 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. On nam ne može reći kada, kako ili zašto je neki memorijski blok izgubljen. Generalno, program ne treba da ima nijedan definitivno ili moguće izgubljen blok na izlazu.

```
==7103== LEAK SUMMARY:
==7103==    definitely lost: 47 bytes in 3 blocks
==7103==    indirectly lost: 0 bytes in 0 blocks
==7103==      possibly lost: 0 bytes in 0 blocks
==7103==    still reachable: 0 bytes in 0 blocks
==7103==         suppressed: 0 bytes in 0 blocks
```

Table 22: Rezime curenja memorije

```
==7103== 16 bytes in 1 blocks are definitely lost in loss record 2 of 3

==7103==    at 0x484AA90: malloc (vg_replace_malloc.c:212)

==7103==    by 0x4006B3: main (main.c:16)

==7103==

==7103== 19 bytes in 1 blocks are definitely lost in loss record 3 of 3

==7103==    at 0x484AA90: malloc (vg_replace_malloc.c:212)

==7103==    by 0x40066B: main (main.c:9)
```

Table 23 : Izveštaj o curenju memorije

Kao što vidimo u tabeli 23, Memcheck je odštampao izveštaj o definitivnom gubitku dva bloka veličine 16 i 19 bajta, kao i o liniji u programu gde su oni alocirani.

Zbog postojanja više tipova curenja memorije postavlja se pitanje koje curenje memorije na izlazu iz programa treba da posmatramo kao „grešku“, a koje ne. Memcheck koristi sledeći kriterijum:

- Memcheck smatra da je curenje memorije „greška“ samo ako je uključena opcija `--leak-check=full`. Drugim rečima, ako podaci o curenju memorije nisu prikazani, smatra se da to curenje nije „greška“.
- Definitivno i moguće izgubljeni blokovi se smatraju za pravu „grešku“, dok indirektno izgubljeni i još uvek dostupni blokovi se ne smatraju kao greška.

5.2 Izvršni prostori

Da bi razumeli alat, moramo razumeti tri prostora u kojima se izvršava klijentski program, kao i različite nivoe kontrole koje alat ima nad ovim prostorima.

Korisnički prostor predstavlja prostor u kome se nalazi klijentski program koji se analizira. Alat vidi sav kod koji se nalazi u korisničkom prostoru. Alat može da vrši instrumentalizaciju koda koji se nalazi u korisničkom prostoru. To uključuje sav glavni programski kod, gotovo sve C biblioteke (uključujući i dinamički povezane), kao i ostale biblioteke.

Prostor jezgra obuhvata mali procenat izvršnog programa koji se odvija u potpunosti u okviru jezgra Velgrinda, koje zamenjuje originalan klijentski kod. On obuhvata delove koji rukuju signalima, operacije vezane za raspoređivanje i *pthread* operacije. On ne obuhvata kod koji je namenjen jezgru Velgrinda ili alatu. Alat nikada ne vidi ovaj kod, pa tako i ne može da vrši instrumentaciju ovog koda. Međutim, može da dobije informacije kada se u prostoru jezgra desio događaj kao što je alokacija memorije, signal je dostavljen, semafor niti je zaključan i drugo.

Sistemski prostor pokriva izvršenje u jezgru operativnog sistema. Sistemski poziv ne može direktno biti posmatran od strane alata, ali jezgro poseduje informaciju šta svaki sistemski poziv radi sa argumentima i poseduje informaciju o alokaciji memorije za svaki sistemski poziv.

Velgrind takođe ne može direktno da prati izvršavanje ioctl (eng. input/output control) sistemskih poziva, ali poseduje informaciju šta svaki ioctl sistemski poziv radi sa argumentima i poseduje informaciju o alokaciji memorije za svaki ioctl.

5.3 Struktura alata - Memcheck

Postoje četiri osnovne funkcije koje mora da poseduje svaki alat Velgrinda.

- Inicijalizacione funkcije (`pre_clo_init` i `post_clo_init`) su jednostavne. One parsiraju komandni argument koji se prosleđuje preko komandne linije i vrše odgovarajuću inicijalizaciju.
- Funkcija za instrumentalizaciju izvršava dva prolaza kroz kod koji instrumentalizuje. U prvom prolazu dodaje kod za analizu koda. Većina dodatog koda se odnosi na računanje obojenih V bita koji će biti opisani u nastavku. A i V testni biti su dodati pre `UCode` instrumentalizacije koja ih zahteva; ako ovi testovi padnu, oni pozivaju funkcije koje prikazuju podatke o grešci, a zatim se kontrola izvršavanja vraća i normalno izvršavanje se nastavlja. Veoma je važno da se ove provere izvrše pre nego što se izvrše operacije koje pristupaju neadresiranoj memoriji, koje mogu da dovedu do prekida klijentskog programa. Drugi prolaz vrši optimizaciju koda za analizu koda.
- Memchek-ova završna funkcija štampa neke statističke podatke, kao što su broj bajta koji je alociran i oslobođen, sumu svih grešaka koje su otkrivene i pokreće proveru curenja memorije ako je Memcheck pokrenut sa opcijom `--leak-check=yes`.

Pored ove četiri funkcije, alat mora da poseduje i dodatne funkcije ako želi da koristi određene usluge jezgra, kao što je snimanje grešaka.

5.4 Definisanost promenljive

Osnovna ideja na kojoj se zasniva provera definisanosti neke promenljive je jednostavna.

- Svaki bit podatka *b*, koji se obrađuje u programu, kako u registrima tako i u memoriji, je obojen jednim metapodatkom, koji se naziva *bit definisanosti*.
- Svaka operacija koja stvara nove vrednosti je obojena sa operacijom koja računa V bite na izlazu, na osnovu V bita na ulazu i operacije. Operacije koje se vrše nad obojenim vrednostima moraju da budu brze, praktične i dovoljno precizne da ne izazivaju previše lažnih uzbuna.
- Svaka operacija koja koristi vrednosti na takav način da bi mogla da utiče na vidljivo ponašanje programa se proverava. Ako V biti pokazuju da je bilo koji od ulaza u operacije nedefinisan, poruka o grešci se generiše. V biti se koriste za detekciju nedefinisane vrednosti. Sledeći događaji mogu zavisiti od nedefinisane vrednosti: kontrola protoka transfera, uslovno kopiranje, pristup adresi u memoriji, podaci prosleđeni sistemskom pozivu.

Upotreba V bita udvostručuje količinu memorije koja se koristi. Većina operacija računa nove vrednosti, a time se zahtevaju i obojene operacije koje se sastoje od jedne ili više novih instrukcija.

5.4.1 V i A biti

Konceptualno, svaki bajt ima 8 V bita, koji definišu da li su korespondirajući biti definisani. Takođe, svaki memorijski bajt ima jedan A bit, koji određuje da li klijentski program može da mu bezbedno pristupi (npr. da li je mapiran i da li ima jedno od RWX (*Read, Write, Execute*, čitaj, piši i izvrši) prava pristupa), tako da je svaki *n*-bitni registar obojen sa *n* V bita, a svaki memorijski bajt je obojen sa 8 V i jednim A bitom.

Ako V bit ima vrednost nula, znači da je podatak kome je taj bit dodeljen definisan, a ako V bit ima vrednost jedan, znači da nije. To je nelogično, ali čini neke od operacija nad obojenim vrednostima efikasnijim nego što bi one bile da je obrnuto.

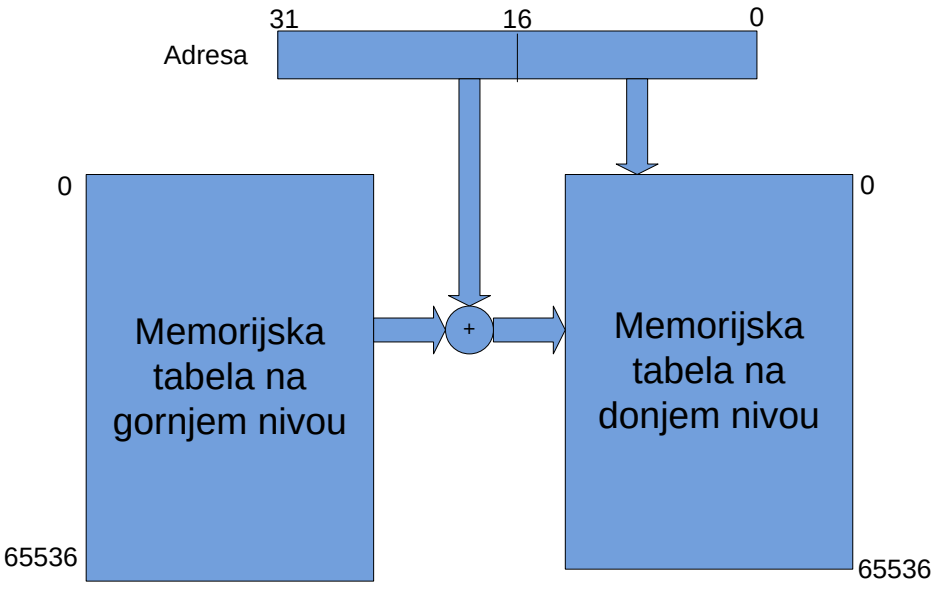
Svaki 32-bitni registar opšte namene je obojen sa 32-bitnim V registrom i svaki bajt memorije je obojen sa V bajtom. Postoje izuzeci za pravilo „jedan V bit za jedan bit podatka“.

- Programski brojač se ne boji. Umesto toga, mi smo definisali da je on uvek definisan i emitujemo grešku samo kada uslovni skok zavisi od nedefinisane vrednosti. Jedan način da otkrijemo takve skokove jeste da pratimo vrednosti koje se dodeljuju programskom brojaču, tj. adrese na koje procesor skače, i u slučaju da te vrednosti (adrese) nisu definisane generišemo podatke o grešci.
- Kontrolni registar za aritmetiku pokretnog zareza (FCSR) takođe se ne boji.

Prilikom implementacije, korišćena su dva načina kompresije da bi izbegli 9 bita memorijskog viška. Memcheck, takođe, snima dodatne informacije o svakom memorijskom bloku u dinamičku memoriju koja je alocirana, u slučaju da zatrebaju prilikom detekcije curenja memorije ili u neke druge svrhe.

Sva čitanja i pisanja se proveravaju u memorijskoj mapi koja sadrži informacije o svoj memoriji u procesima. Na 32-bitnim mašinama memorijska mapa je organizovana na sledeći način. Gornjih 16 bita adrese se koriste za adresiranje memorijske tabele na gornjem nivou, ukupno 65536 stavki. Svaka stavka je pokazivač na tabelu na drugom (donjem) nivou, koja sadrži informacije o dostupnosti i ispravnosti za 65536 bajta koji su indeksirani sa donjih 16 bita adrese (slika 3). Svaki bajt je predstavljen sa dva bita (detalji će biti objašnjeni u nastavku).

Sve stavke u primarnoj mapi (mapa na gornjem nivou) moraju pokazivati na validne mape na sekundarnom (drugom) nivou. Budući da će mnogi delovi memorije od po 64KB imati isti status za svaki bit, npr. nema pravo pristupa (za nekorišćen adresni prostor) ili u potpunosti definisano (za kodni segment), elemente memorijske mape na drugom nivou možemo podeliti u tri osnovne kategorije: nema pravo pristupa, nedefinisan i definisan. Praktično je više od pola adresirane memorije obeleženo sa nedefinisano ili definisano. To nam omogućava da brže postavimo A i V bite za veće memorijske regione u funkciji `set_address_range_perms()`.



Slika 3 : Organizacija memorijskih mapa A i V bita

Na 64-bitnim mašinama je ovakav način podele memorije komplikovan. Ako bi koristili istu analogiju kao na 32-bitnim mašinama, imali bi tabele na četiri nivoa što bi zahtevalo previše memorije. Umesto toga, memorijska mapa na gornjem nivou ima 2^19 elemenata (indeksirani sa bitima 16...34); ovo pokriva donjih 32GB. Svaki pristup gornjem delu memorije se vrši preko

sporije, pomoćne tabele. Menadžer adresnog prostora je napravljen da minimizuje korišćenje gornjih 32GB adresnog prostora.

Svaki bajt memorije je definisan sa 8 V bita i jednim A bitom. U praksi je znatno manji broj bajta delimično definisan u odnosu na broj definisanih bajta, tako da tu činjenicu možemo iskoristiti prilikom kompresije obojene memorije. Svaki bajt predstavljamo sa 2 bita koji prikazuju njegovo stanje:

- 00: nema prava pristupa (neadresiran, ali u potpunosti definisan)
- 01: nedefinisan (adresiran i u potpunosti nedefinisan)
- 10: definisan (adresiran i u potpunosti definisan)
- 11: delimično definisan (adresiran i delimično definisan)

U slučaju kada je bajt delimično definisan, koristimo sekundarnu tabelu u kojoj su smešteni V biti. Svaki element u sekundarnoj V bit tabeli mapira po jedan bajt. Kompresovane A i V bite snimamo u 8-bitne delove (V i A biti za četiri bajta se čuvaju u jednom delu). Ovo nam omogućava da dobavimo V i A bite za 4 bajta odjednom veoma lako (bez ikakvog pomeranja i/ili maskiranja). Kada vršimo operacije nad V bitima, koristimo nekompresovane vrednosti. To znači da se u registrima nalaze stvarne vrednosti V bita, a ne kompresovane. Zato, kada kopiramo vrednosti iz memorije u registre, moramo proširiti V bite. Ovo zahteva posebnu pažnju na par mesta.

5.5 Instrumentacija

U principu je moguće direktno vršiti transformaciju V bita za svaku MIPS instrukciju. U praksi je instrukcijski set za arhitekturu MIPS kompleksan i ne isplati se direktno vršiti transformaciju V bita za svaku instrukciju. UCode (Velgrindova unutrašnja reprezentacija) je RISC i jasno prikazuje sve memorijske i aritmetičke operacije, koje čine Memcheck instrumentaciju mnogo jednostavnijom.

Drugi važan aspekt uvođenja UCode-a je taj što on podržava korišćenje beskonačnog broja virtuelnih registara. Početni prevod se vrši u odnosu na te registre. Memcheck vrši instrumentaciju ubacujući nove UCode instrukcije koje koriste virtuelne registre. Jezgro Velgrinda ima zadatak da izvrši selekciju pravih instrukcija, koje će se izvršavati na ciljnoj platformi u odnosu na UCode instrukcije.

5.6 Karakteristike Memcheck-a

Memcheck analiza, tačnije praćenje V bita, je veoma zahtevno. Prvo, kod za analizu koda se prožima: svaka instrukcija koja zahteva neke vrednosti mora biti instrumentalizovana sa kodom za analizu koda koji obavlja iste operacije samo nad obojenim V registrima, i svako čitanje i pisanje u memoriju mora biti instrumentalizovano sa kodom za analizu koda koji proverava relevantne A bite; količina koda za analizu koja je dodata, je u stvari, veća od količine originalnog koda. Drugo, kod za analizu koda je povezan: kod za analizu koda za svaku instrukciju je dve ili tri instrukcije, koje su povezane međurezultatima. Treće, on prati lokaciju metapodataka: metapodaci (V biti) za svaku vrednost u programu.

Sledeća lista opisuje nekoliko karakteristika koje poseduje platforma Velgrind.

- *Multipleksiranje stanja.* UCode koristi virtuelne registre koji su od ogromne pomoći. Memcheck ne mora da rukuje sa problemima koji nastaju prilikom multipleksiranja dva seta stanja (klijentsko stanje i stanje koda za analizu koda) u jedan set mašinskih registara. Praktično, Memcheck može da manipuliše obojenim vrednostima registara i da računa V bite bez da zna kako originalan kod izgleda. Memcheck ne mora da vodi računa o pronalaženju registara za vrednosti koje su potrebne za unutrašnju analizu. To, takođe, znači da Memcheck ne mora da vodi računa o tome da kod za analizu koda slučajno promeni stanje klijentskog programa.
- *Preplitanje koda.* Kod za analizu koda i klijentski kod su izraženi u UCode-u i jezgro rukuje sa preplitanjem ova dva kodna toka.
- *Fleksibilna instrumentacija.* Memcheck koristi sve tri forme analize koda. Analiza koda je urađena u potpunosti u istom redu, gde god je to moguće; ovo je vitalno za proračun samih V bita. Za složenije operacije, Memcheck poziva različite asemblerske rutine i C funkcije.
- *UCode je RISC* (redukovan set računarskih instrukcija). Memcheck instrumentalizuje znatan broj UCode instrukcija na različit način. Međutim, ovaj broj je daleko manji od broja MIPS instrukcija, što u mnogome olakšava stvari. UCode čitaj/piši instrukcije omogućavaju veoma laku identifikaciju pristupa memorijskim zonama.
- *Podrška bojenju registara.* Velgrind omogućava eksplicitnu podršku za bojenje registara alatu koji to zahteva. Prema obojenim registrima se postupa kao i prema normalnim klijentskim registrima: oni se čuvaju u osnovnom bloku; mogu se

povući u virtuelne registre i nad njima se može izvršiti bilo koja UCode instrumentacija.

- *Podrška za bojenje memorije.* Velgrind poseduje eksplicitnu podršku za bojenje memorije, tako što alocira određeni adresni prostor kada je klijentski kod učitan. Alat određuje koliko mu je memorije potrebno za obojene vrednosti za svaki bajt klijentske memorije. U slučaju alata Memcheck, taj odnos je 9:8 - svaki bajt klijentske memorije je obojen sa jednim A bitom i osam V bita.

  Memcheck instrumentalizuje sve čitaj/piši instrukcije sa odgovarajućim obojenim čitaj/piši instrukcijama, koje su urađene kao pozivi određenih C funkcija. Obojena memorija je podeljena na delove koji se čuvaju u tabeli. Svaki deo sadrži obojene vrednosti za region od 64KB.

Sve ove stavke su zapravo podrška za izračunavanje obojenih vrednosti. Bez ove podrške, alat Memchek bi verovatno mogao da odradi sav neophodan posao, ali bi bio znatno veći i kompleksniji. U nastavku je data lista stavki koje Velgrind obezbeđuje, koje nisu u direktnoj vezi sa instrumentacijom, i koje, takođe, olakšavaju rad alata Memcheck.

- *Povratne vrednosti memorijskih događaja*. Alat Memchek poseduje informacije o svakom memorijskom događaju, posebno o alokaciji i dealokaciji memorije. Ovo pokriva sve segmente memorije: statičku memoriju, dinamičku memoriju (*heap*), stek i ostale mapirane segmente. Događaji uključuju širenje i skupljanje steka, kao i sistemske pozive `mmap()`, `brk()`, `mproteck()`, `mremap()`, `munmap()`, `shmat()`, `shmdt()`. On, takođe, mora da zna o memorijskim pisanjima i čitanjima koji se obavljaju u sistemskom prostoru, tako da može da rukuje sa sistemskim pozivima na odgovarajući način. Na primer, A i V biti se proveravaju pre nego što neki sistemski poziv pročita memoriju, a V biti se ažuriraju posle sistemskih poziva koji upisuju vrednosti u memoriju. Velgrind omogućava alatu da pristupi povratnim vrednostima svake od ovih funkcija, iako alat može sam da odredi povratne vrednosti neke od ovih funkcija (npr. alokacija/dealokacija steka se dešava u korisničkom prostoru i može biti detektovana tokom instrumentacije tako što se identifikuje manipulacija sa stekom u UCode-u). Takođe, Velgrind jezgro mora ove operacije da izvršava veoma efikasno; npr. alokacija/dealokacija steka se dešava veoma često pa mora biti veoma optimizovana.
- *Zamena funkcija*. Velgrind mora da omogući alatu da može da izvrši zamenu bibliotečkih funkcija sa svojom verzijom istih, koje se nalaze u korisničkom prostoru. Zamenjene funkcije su ubačene u deljeni objekat koji je povezan u sekciji klijentske memorije, prepisujući originalnu verziju funkcije. Ubačene funkcije

moraju da zamenjuju akciju zamenjene funkcije, kao i da obavljaju dodatni posao za alat.

Memcheck zamenjuje standardne C i C++ funkcije za manipulaciju sa memorijom (`malloc()` i ostale) sa svojim verzijama istih. Zamenjene funkcije se izvršavaju u korisničkom prostoru, ali koriste klijentski zahtev da predaju kontrolu jezgru, da bi se koristile rutine jezgra koje rukuju sa memorijom. Postoje tri razloga zašto se ove funkcije zamenjuju. Prvo, obezbeđujemo podatke o povratnim vrednostima prilikom rukovanja sa dinamičkom memorijom (tako svi alati, kojima su potrebne ove vrednosti prilikom alokacije dinamičke memorije, moraju da zamene ove funkcije; jezgro omogućava podršku za ovo). Drugo, dinamički blokovi se mogu okružiti sa takozvanom crvenom zonom da bi se poboljšala detekcija njihovog širenja i skupljanja. Treće, verzije funkcija za oslobađanje memorije alata Memcheck odlažu samo oslobađanje dinamičke memorije za određeno vreme, što otvara mogućnost korišćenja tih blokova kasnije kada se izveštaj o nastalim greškama generiše.

Memcheck takođe vrši zamenu i nekih C funkcija koje kao parametre primaju pokazivače na niz znakova: `strlen()`, `strcpy()`, `memcpy()` itd. To je zato što njihova implementacija u MIPS biblioteci `libc` koristi veoma optimizovan asemblerski kod sa kojim Memcheck ne rukuje dobro. Ove funkcije mogu da izazovu generisanje niza grešaka, iako greške stvarno ne postoje. Zato Memchek koristi sopstvene verzije ovih funkcija koje ne dovode do generisanja nepostojećih grešaka. Takođe, verzije funkcija alata Memcheck mogu da provere da li se memorijski blokovi koji se prosleđuju ovim funkcijama preklapaju; ako je greška otkrivena, klijentskim zahtevom se predaje kontrola jezgru, koji generiše podatke o grešci.

- *Snimanje grešaka i suzbijanje njihovog prikazivanja.* Velgrind obezbeđuje ugrađenu podršku za snimanje grešaka koje alat otkrije. Alat može da snimi podatke o grešci tako što će predati sve neophodne informacije o grešci koju je detektovao jezgru. Jezgro izdaje podatke o grešci koja sadrži primljenu informaciju od alata, kao i detalje koji se nalaze na steku. Jezgro čita bilo koju tabelu simbola i informacije za pomoć pri otklanjanju grešaka, koje mogu da omoguće da odmotavanje steka bude informativno. Kada su informacije za pomoć pri otklanjanju grešaka prisutne, tada je moguće pronaći tačnu liniju u kodu na kojoj se generisala greška. Ako alat detektuje grešku koja je već otkrivena (kada su kod greške i lokacija u kodu gde se generiše greška isti kao prethodna greška) tada

jezgro ignoriše novu grešku i ne prikazuje informacije o novoj grešci. Na ovaj način izveštaj o nastalim greškama neće biti pretrpan ponovljenim infomacijama o istoj grešci. Na kraju, jezgro omogućava i zabranu prikazivanja greške (korisnik može da napiše zabranu prikazivanja za neku grešku). Na taj način on obaveštava jezgro da ignoriše informacije o nastaloj grešci koju je detektovao alat. Ovo je veoma važno za greške koje se generišu prilikom izvršavanja funkcije iz neke biblioteke koja nije pod direktnom kontrolom korisnika.

Alat mora da obezbedi jezgru potrebne informacije za poređenje i štampanje izveštaja o nastalim greškama, kao i informacije za čitanje informacija o greškama koje treba da suspenduje. Izveštaj o nastaloj grešci može da se ponavlja, tako da Velgrind mora da poseduje mehanizam kojim će korisniku prikazati samo jedan izveštaj o nastaloj grešci. Ovo je dodatni posao, ali je manji nego da se rukovalac greškama piše od početka. Alat je dobar samo onoliko koliko je podatak o nastaloj grešci dobar. Memcheck zavisi od Velgrindove podrške za rukovanje greškama.

- *Klijentski zahtev.* Alati mogu da definišu sopstvene klijentske zahteve. Ako zahtev nije prepoznat od strane Velgrind jezgra, može biti prosleđen alatu koji ga može interpretirati kako želi. Ovo otvara mogućnost kombinovanja statičke i dinamičke analize. Prevodilac može sistemski da ugradi klijentski zahtev u preveden program, kako bi predao statički izračunate informacije DBA alatu. Kada se program izvršava normalno, klijentski zahtev nema efekta na njegovo izvršavanje, ali ako se program izvršava pod kontrolom DBA alata, alat ima pristup informacijama kojima inače ne bi mogao da pristupi. Ovo omogućava da DBA alat bude efikasniji i tačniji. Međutim, nijedan Velgrindov alat ne koristi ovu mogućnost.

  Memcheck obezbeđuje klijentske zahteve koji omogućavaju programu da proglasi memorijska područja kao: čitljiva, moguće je upisivati u njih ili nisu dostupna. Ovo je povremeno korisno kada program radi nešto neobično što se kosi sa Memchek-ovim pretpostavkama. Na primer, OpenSSL namerno čita nedefinisanu memoriju u nekim okolnostima, kao izvor nasumičnosti. Takođe, Memcheck može da propusti neke greške u programu koji koristi namenski menadžer memorije, a koje bi otkrio da program koristi `malloc()`, `new()` itd. Klijentski zahtevi koji se ugrađuju u namenske memorijske alokatore govore Memcheck-u kada je neki blok alociran ili oslobođen.

6. Cachegrind

Cachegrind je alat koji simulira i prati pristup skrivenoj memoriji mašine na kojoj se program, koji se analizira, izvršava. On simulira skrivenu memoriju mašine, koja ima prvi nivo skrivene memorije podeljen u dve odvojene nezavisne sekcije: I1 - sekcija brze memorije u koju se smeštaju instrukcije i D1 - sekcija brze memorije u koju se smeštaju podaci. Drugi nivo skrivene memorije koju Cachegrind simulira je objedinjen - L2. Ovaj način konfiguracije odgovara mnogim modernim mašinama.

Međutim, postoje mašine koje imaju i treći nivo brze memorije L3. U tom slučaju, Cachegrind simulira pristup trećem nivou. Generalno gledano, Cachegrind simulira L1, D1 i LL (poslednji nivo skrivene memorije).

Cachegrind prikuplja sledeće statističke podatke o programu koga analizira (skraćenice koje se koriste u daljem tekstu su date u zagradama):

- `I` čitanje brze memorije (`Ir`, što predstavlja broj izvršenih instrukcija), `I1` broj promašaja čitanja (`I1mr`) i broj promašaja čitanja instrukcija nivoa LL brze memorije (`ILmr`).
- `D` čitanje brze memorije (`Dr`, što je jednako broju čitanja memorije), `D1` broj promašaja čitanja (`D1mr`), i broj promašaja čitanja podataka nivoa LL brze memorije (`DLmr`).
- `D` pisanje u brzu memoriju (`Dw`, što je jednako broju pisanja u memoriju), `D1` broj promašaja pisanja u brzu memoriju, i broj promašaja pisanja podatka u nivoa LL brze memorije. (`DLmw`).
- Broj uslovno izvršnih grana (`Bc`) i broj promašaja uslovno izvršnih grane (`Bcm`).

- Broj indirektno izvršnih grana (`Bi`) i broj promašaja indirektno izvršnih grana (`Bim`).

Primetimo da je broj pristupa D1 delu brze memorije jednak zbiru `D1mr` i `D1mw` broja pristupa, dok je ukupan broj pristupa nivou LL brze memorije jednak zbiru `ILmw`, `DLmr` i `DLmw` broja pristupa. Ova statistika se prikuplja na nivou celog programa, kao i pojedinačno na nivou funkcije. Takođe, možemo dobaviti i broj pristupa skrivenoj memoriji za svaku liniju koda u originalnom programu. U [4] je pokazano da na modernim mašinama L1 promašaj košta oko 10 procesorskih ciklusa, LL promašaj košta oko 200 procesorskih ciklusa, a promašaji uslovno i indirektno izvršne grane od 10 do 30 procesorskih ciklusa.

6.1 Korišćenje Cachegrinda

Na početku korišćenja alata Cachegrind, program koji želimo da analiziramo pokrećemo sa samim Cachegrind-om. Na taj način prikupljamo informacije koje su nam potrebne za kasnije profilisanje koda. Zatim pokrećemo alat `cg_annotate` u okviru paketa Velgrind koji nam prikazuje detaljan izveštaj o programu koji smo analizirali sa Cachegrid-om. Opciono, možemo da koristimo alat `cg_merge` da sumiramo u jednu datoteku više izlaza koje smo dobili višestrukim pokretanjem Cachegrind-a nad istim programom. Tu datoteku kasnije koristimo kao ulaz u `cg_annotate`. Takođe, možemo da koristimo alat `cg_diff` koji pravi razliku između više izlaza iz alata Cachegrind, koje kasnije koristimo kao ulaz u alat `cg_annotate`.

6.2 Cachegrind metapodaci

Cachegrind metapodaci se čuvaju u strukturama opisanim u nastavku.

Globalno stanje brze memorije. Prva struktura koja se nalazi u sklopu Cachegrind metapodataka je globalno stanje brze memorije. Ona predstavlja stanje tri dela simulirane brze memorije (L1, D1, LL). Njene vrednosti se osvežavaju prilikom izvršene svake instrukcije programa čija se brza memorija simulira, tačnije, prilikom poziva funkcija koje simuliraju pristup brzoj memoriji ciljne platforme. Funkcijama se prosleđuju informacije o pristupu brzoj memoriji, kao što su adrese i veličina memorije kojoj se pristupa.

Simulacija pristupa brzoj memoriji ima sledeće karakteristike.

1. Kada se desi promašaj upisa u brzu memoriju, blok koji je potrebno upisati se smešta u D1 deo brze memorije.

2. Instrukcije koje modifikuju vrednost memorije tretiraju se kao čitanje brze memorije. Naime, instrukcije koje menjaju sadržaj brze memorije najpre čitaju sadržaj brze memorije, modifikuju vrednost i snimaju novu vrednost. Samim tim, upis u brzu memoriju ne može da izazove promašaj, jer je garantovan uspešnim čitanjem. Takođe, cilj Cachegrind-a nije da prikaže koliko puta se pristupa brzoj memoriji, već da prikaže broj promašaja pristupa brzoj memoriji.
3. Linija u brzoj memoriji, kojoj odgovara sadržaj u memoriji sa direktnim pristupom, određuje se kao [M, (M + N - 1)], gde je: veličina linije = 2^M bajta, (veličina brze memorije / veličinom linije) = 2^N bajta.
4. L2 deo brze memorije replicira sve unose u L1 deo brze memorije.
5. Onaj blok u brzoj memoriji koji se najmanje koristi će biti izbačen iz brze memorije ukoliko je potrebno ubaciti novi blok podataka u brzu memoriju.
6. Sa referencama koje pokazuju na dve linije u keš memoriji rukuje se na sledeći način:
    a) ukoliko su pronađena oba bloka u brzoj memoriji, računamo samo jedan pogodak;
    b) ukoliko jedan blok nađemo u brzoj memoriji, a drugi ne, računamo jedan promašaj (i nula pogodaka);
    c) ukoliko oba bloka ne pronađemo u brzoj memoriji, računamo jedan promašaj (ne dva).

Parametri simulirane brze memorije (veličina brze memorije, veličina linije i asocijativnost) određuju se na jedan od dva načina. Prvi način je upotrebom `cpuid` naredbe. Drugi način predstavlja ručno unošenje parametara simulirane brze memorije, prilikom pokretanja samog Cachegrind-a.

Centralna tabela troškova. Druga struktura koja čini metapodatke alata Cachegrind je centralna tabela troškova. Svaka linija u kodu koja se instrumentalizuje, alocira jednu ovakvu tabelu u koju smešta podatke o pristupu brzoj memoriji, broju pogodaka i promašaja pristupa brzoj memorji, koji se dešavaju prilikom izvršavanja same linije koda. U tabeli 24 su prikazane strukture koje predstavljaju centralnu tabelu troškova. `ULong` je 64-bitna celobrojna vrednost. Uzimamo 64-bitnu vrednost, jer broj pristupa može da bude veći nego što može da se predstavi 32-bitnom celobrojnom vrednošću. U CC strukturi m1 i m2 predstavljaju broj promašaja za L1 i LL deo skrivene memorije. Struktura lineCC sadrži tri CC elementa: za čitanje I dela, za čitanje dela D i za pisanje u deo D skrivene memorije. Polje `next` je potrebno jer je centralna tabela troškova predstavljena kao promenljiva „heš“ tabela. Polje `line` predstavlja broj linije u kodu koja odgovara toj tabeli troškova. Sam broj linije nije dovoljan da bi se pronašla linija u kodu

kojoj odgovara centralna tabela troškova (ime fajla je potrebno). U praksi, to znači da centralna tabela troškova ima tri nivoa: troškovi su grupisani po imenu fajla, zatim po imenu funkcije i, na kraju, po broju linije.

```
typedef struct {
   ULong a;            // Broj pristupa
   ULong m1;           // Broj promasaja L1
   ULong m2;           // Broj promasaja L2
} CC;


typedef struct _lineCC lineCC;
struct _lineCC {
   Int line;            // Broj linije u programskom kodu
   CC Ir;               // CC za citanje I dela
   CC Dr;               // CC za citanje D dela
   CC Dw;               // CC za pisanje u D deo
   lineCC* next;        // sledeci cvor lineCC u hesh tabeli
};
```

Table 24: Strukture centralne tabele troškova

```
typedef struct _instr_info instr_info;
struct _instr_info {
   Addr instr_addr;     // adresa instrukcije
   UChar instr_size;    // velicina instrukcije u bajtima
   UChar data_size;// velicina podatka kome instrukcija pristupa
   lineCC* parent;      // lineCC za ovu instrukciju
};
```

Table 25: Struktua tabele informacija o instrukcijama

Tabela informacija o instrukcijama. Treća struktura koja čini metapodatke alata Cachegrind je tabela informacija o instrukcijama. Ona se koristi za čuvanja nepromenljivih

informacija o samim instrukcijama tokom samog procesa instrumentacije. Na ovaj način se smanjuje veličina dodatog koda kojim se analizira kod. Povećava se brzina izvršavanja samog alata, jer se smanjuje broj argumenata koji se prosleđuju funkcijama, koje vrše simulaciju pristupa brzoj memoriji.

Svakoj instrumentalizovanoj instrukciji dodeljuje se po jedno polje u tabeli informacija o instrukcijama, koje sadrži strukturu `instr_info`, prikazanu u tabeli 25. Polje `instr_addr` predstavlja adresu instrukcije, `instr_size` predstavlja veličinu instrukcije izraženu u bajtima, `data_size` čuva veličinu podatka kome instrukcija pristupa (0 ukoliko instrukcija ne pristupa memoriji) i `parent` pokazuje na polje u tabeli troškova za istu liniju koda odakle je instrukcija izvedena.

6.3 Instrumentacija

Prvi korak prilikom instrumentacije koda odnosi se na prolaz kroz sve osnovne blokove pojedinačno radi prebrojavanja instrukcija koje se nalaze u njima. Na osnovu ovog broja se kreira lista `instr_info` elemenata, pri čemu svaki element liste odgovara jednoj instrukciji u osnovnom bloku.

U drugom prolazu, Cachegrind vrši kategorizaciju originalnih instrukcija. Cachegrind deli instrukcije koje analizira u sledećih 5 kategorija:

1. Instrukcije koje ne pristupaju memoriji, npr. `move $t3, $a0`
2. Instrukcije koje čitaju sadržaj memorije, npr. `lw $t3, 4($a0)`
3. Instrukcije koje upisuju sadržaj registra u memoriju, npr. `sw $t3, 4($a0)`
4. Instrukcije koje modifikuju sadržaj memorijske lokacije
5. Instrukcije koje čitaju sadržaj iz jedne memorijske lokacije i taj sadržaj upisuju u drugu memorijsku lokaciju.

Svaka instrukcija sistema baziranog na MIPS procesorima je rastavljena na više UCode instrukcija, tako da Cachegrind određuje kojoj kategoriji originalna instrukcija pripada na osnovu `LOAD` i `STORE` UCode instrukcija. Takođe, Cachegrind čita informacije koje pomažu pri otklanjanju grešaka. Na osnovu ovih informacija on kreira elemente `lineCC` u centralnoj tabeli troškova. Zatim inicijalizuje određene `instr_info` elemente u niz koji je inicijalizovan za svaki osnovni blok pojedinačno (gde n-ti element `instr_info` odgovara n-toj instrukciji u osnovnom bloku). Kada je inicijalizovao sve elemente `lineCC` i `instr_info` alat Cachegrind izvršava proces instrumentalizacije koda koji se sastoji iz poziva odgovarajućih C funkcija, koje simuliraju pristup brzoj memoriji ciljne platforme. Koja C funkcija će biti pozvana zavisi od

kategorije kojoj instrukcija pripada. Postoje samo četiri vrste C funkcija koje simuliraju pristup brzoj memoriji, jer funkcije koje pripadaju drugoj i četvrtoj kategoriji pozivaju istu C funkciju za simuliranje pristupa brzoj memoriji, kao što je i objašnjeno u 6.2 sekciji. Broj parametara koji se prosleđuje C funkcijama se, takođe, određuje na osnovu kategorije kojoj ta funkcija pripada.

6.4 Prikaz statističkih informacija

Prilikom završetka analize programa Cachegrind pohranjuje prikupljenu tabelu troškova u datoteku koji se naziva `cachegrind.out.pid`; pri čemu `pid` predstavlja jedinstveni identifikator procesa koji se izvršio. Alat grupiše sve troškove po fajlovima i funkcijama kojima ti troškovi pripadaju. Globalna statistika (npr. ukupan broj L1 promašaja) se računa naknadno, prilikom prikaza rezultata. Na ovaj način se štedi jako puno vremena prilikom analize koda. Funkcije koje simuliraju pristup brzoj memoriji se pozivaju jako često, tako da bi dodavanje još nekoliko instrukcija koje sabiraju, znatno usporilo i ovako sporo izvršavanje alata.

7. Helgrind

Helgrind je alat u sklopu programskog paketa Velgrind koji otkriva greške sinhronizacije prilikom upotrebe modela niti POSIX.

Glavne apstrakcije modela niti POSIX su: grupa niti koja deli zajednički adresni prostor, formiranje niti, čekanje za završetak izvršenja funkcije niti, izlaz iz funkcije niti, muteks objekti, uslovne promenljive, čitaj-piši zaključavanje i semafori. Helgrind može da otkrije sledeće tri klase grešaka:

1. Loša upotreba sučelja za programiranje niti POSIX.
2. Potencijalno blokiranje niti koje proističe iz lošeg redosleda zaključavanja i otključavanja promenljivih.
3. Pristup memoriji bez adekvatnog zaključavanja ili sinhronizacije.

Problemi kao što su ovi često uzrokuju nereproduktivne, vremenski zavisne padove programa i veoma ih je teško otkriti. Alat Helgrind poseduje mehanizam za veoma precizno praćenje svih apstrakcija koje koriste model niti POSIX. Helgrind daje najbolje rezultate ako program koji se analizira koristi samo sučelje za programiranje niti POSIX.

7.1 Loša upotreba sučelja za programiranje niti POSIX

Helgrind presreće pozive ka funkcijama biblioteke *pthread*, i zbog toga je u mogućnosti da otkrije veliki broj grešaka. Ovakve greške mogu da dovedu do nedefinisanog ponašanja

programa i do pojave grešaka u programima koje je kasnije veoma teško otkriti. Greške koje Helgrind pronalazi su:

- otključavanje nevažećeg muteksa
- otključavanje nezaključanog muteksa
- otključavanje muteksa koga je zaključala druga nit
- uništavanje nevažećeg ili zaključanog muteksa
- rekurzivno zaključavanje nerekurzivnog muteksa
- dealokaciju memorije koja sadrži zaključan muteks
- prosleđivanje muteksa kao argumenta funkcije koja očekuje kao argument reader-writer lock i obrnuto
- kada *pthread* funkcija vrati kod greške koji je potrebno dodatno obraditi
- kada se nit uništi, a da još drži zaključanu neku promenljivu
- prosleđivanje funkciji `pthread_cond_wait` nezaključan muteks, nevažeći muteks, ili muteks koga je zaključala druga nit
- nekonzistentne veze 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 nije nikad 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.

Provere koje se odnose na mutekse se takođe primenjuju i na reader-writer locks. Prijavljena greška prikazuje i primarno stanje steka koje pokazuje gde je detektovana greška. Takođe, ukoliko je moguće ispisuje se i broj linije u samom kodu gde se greška nalazi. Ukoliko se greška odnosi na muteks, Helgrind će prikazati i gde je prvi put detektovao problematični muteks (tabela 26).

```
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)
```

Table 26: Primer prikaza greške u programu

7.2 Potencijalno blokiranje niti

Helgrind prati redosled kojim niti zaključavaju promenljive. Na ovaj način Helgrind detektuje potencijalne delove koda koji mogu dovesti do blokiranja niti. Na ovaj način je moguće detektovati greške koje se nisu javile tokom samog procesa testiranja programa, već se javljaju kasnije tokom korišćenja istog.

Prost primer takvog problema je prikazan u nastavku.

- Pretpostavimo da deljeni objekat R kome da bi pristupili moramo da zaključamo dve promenljive L1 i L2.
- Zamislimo zatim da dve niti T1 i T2 žele da pristupe deljenoj promenljivoj R. Do blokiranja niti dolazi kada nit T1 zaključa L1, a u istom trenutku T2 zaključa L2. Nakon toga nit T1 ostane blokirana jer čeka da se otključa L2, a nit T2 ostane blokirana jer čeka da se otključa T1.

Helgrind kreira graf koji predstavlja sve promenljive koje se mogu zaključati, a koje je otkrio u prošlosti. Kada nit naiđe na novu promenljivu koju zaključava, graf se osveži i proverava se da li graf sadrži krug u kome se nalaze zaključane promenljive. Postojanje kruga u kome se nalaze zaključane promenljive je znak da je moguće da će se niti nekada u toku izvršavanja blokirati. Ako postoje više od dve zaključane promenljive u krugu problem je još ozbiljniji.

7.3 Pristup memoriji bez adekvatnog zaključavanja ili sinhronizacije

Pristup podacima bez adekvatnog zaključavanja ili sinhronizacije se odnosi na problem kada dve ili više niti pristupaju deljenom podatku bez sinhronizacije. Na ovaj način je moguće da dve ili više niti u istom trenutku pristupe deljenom objektu.

7.4 Primer pristupa promenljivoj bez adekvatne sinhronizacije

U nastavku je dat primer pristupa promenljivoj bez adekvatne sinhronizacije.

```
#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;
}
```

Table 27: Primer pristupa promenljivoj bez adekvatne sinhronizacije

Problem je u tome što ništa ne sprečava niti roditelj i dete da u isto vreme pristupe i promene vrednost deljene promenljivoj `var`. Prilikom analize ovakvog programa alatom Helgrind dobijamo sledeći izveštaj (tabela 28).

```
Thread #1 is the program's root thread


Thread #2 was created
   at 0x511C08E: clone (in /lib64/libc-2.8.so)
   by 0x4E333A4: 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
```

Table 28: Izveštaj Helgrinda za pristup promenljivoj bez sinhronizacije

U izveštaju koji je prikazan u tabeli 28 možemo tačno da vidimo koje niti pristupaju promenljivoj bez sinhronizacije, gde se vrši sam pristup promenljivoj, ime i veličinu same promenljive kojoj niti pristupaju radi promene njene vrednosti.

7.5 Algoritam detekcije pristupa promenljivoj bez sinhronizacije

Algoritam za detekciju pristupa promenljivoj bez sinhronizacije odnosi se na „desilo se pre“ princip. U nastavku je dat primer koji objašnjava „desilo se pre“ princip (tabela 29).

Parent thread:	Child thread:
int var;	
// create child thread	
pthread_create(...)	
var = 20;	var = 10;
	exit
// wait for child	
pthread_join(...)	
printf("%d\n", var);	

Table 29: „Desilo se pre“ princip

Nit roditelj kreira nit dete. Zatim obe menjaju vrednost promenljive `var`, a zatim nit roditelj čeka da nit dete izvrši svoju funkciju. Ovaj program nije dobro napisan jer ne možemo sa sigurnošću da znamo koja je vrednost promenljive `var` prilikom štampanja iste. Ako je nit roditelj brža od niti dete, onda će biti štampana vrednost 10, obrnuto će biti štampana vrednost 20. Brzina izvršavanja niti roditelj i dete je nešto na šta programer nema uticaja. Rešenje ovog problema je u zaključavanju promenljive `var`. Na primer, možemo da pošaljemo poruku iz niti roditelj nakon što ona promeni vrednost promenljive `var`, a nit dete neće promeniti vrednost promenljive `var` dok ne dobije poruku. Na ovaj način smo sigurni da će program ispisati vrednost 10. Razmena poruka kreira „desilo se pre“ zavisnost između dve dodele vrednosti: `var = 20;` se događa pre `var = 10;`. Takođe, sada više nemamo pristup promenljivoj bez sinhronizacije. Nije obavezno da šaljemo poruku iz niti roditelj. Možemo poslati poruku iz niti dete nakon što ona izvrši svoju dodelu. Na ovaj način smo sigurni da će se štampati 20.

Alat Helgrind radi na istom ovom principu. On prati svaki pristup memorijskoj lokacij. Ako se lokaciji, u ovom primeru `var`, pristupa iz dve niti, Helgrind proverava da li su ti pristupi povezani sa „desilo se pre“ vezom. Ako nisu, alat prijavljuje grešku o pristupu promenljivoj bez sinhronizacije.

Ako je pristup deljenoj promenljivoj iz dve ili više programske niti povezan sa „desilo se pre“ vezom, znači da postoji sinhronizacioni lanac između programskih niti koji obezbeđuje da se sam pristup odvija po tačno određenom redosledu, bez obzira na stvarne stope napretka pojedinačnih niti.

Standardne primitive niti kreiraju „desilo se pre“ vezu:

- Ako je muteks otključan od strane niti T1, a kasnije ili odmah zaključan od strane niti T2, onda se pristup memoriji u funkciji niti T1 dešava pre nego što nit T2 otključava muteks da bi pristupila memoriji.
- Ista ideja se odnosi i na reader-writer zaključive promenljive.
- Ako je kondiciona promenljiva signalizirana u funkciji niti T1 i ako druga nit T2 čeka na taj signal, da bi nastavila sa radom, onda se memorijski pristup u T1 dešava pre signalizacije, dok nit T2 vrši pristup memoriji nakon što izađe iz stanja čekanja na signal koji šalje nit T1.
- Ako nit T2 nastavlja sa izvršavanjem nakon što nit T1 oslobodi semafor, onda kažemo da postoji „desilo se pre“ relacija između programskih niti T1 i T2.

Helgrind presreće sve gore navedene događaje i kreira usmereni graf koji predstavlja sve „desilo se pre“ relacije u programu. Takođe, on prati sve pristupe memoriji u programu. Ako postoji pristup nekoj memorijskoj lokaciji u programu od strane dve niti i Helgrind ne može da nađe putanju kroz graf od jednog pristupa do drugog, generiše podatak o grešci u programu koga analizira.

Helgrind ne proverava da li postoji pristup nekoj memorijskoj lokaciji bez sinhronizacije ukoliko se svi pristupi toj lokaciji odnose na čitanje sadržaja te lokacije. Dva pristupa su u „desilo se pre“ relaciji, iako postoji proizvoljno dugačak lanac sinhronizacionih događaja između ta dva pristupa. Ako nit T1 pristupa nekoj lokaciji L, zatim signalizira nit T2, koja kasnije signalizira nit T3 koja pristupa lokaciji L, kažemo da su ova dva pristupa između niti T1 i T3 u „desilo se pre“ relaciji, iako između njih ne postoji direktna veza.

Helgrind algoritam za detekciju pristupa memoriji bez sinhronizacije prikupljene informacije prikazuje u formi prikazanoj na sledećem primeru (tabela 30).

```
Thread #2 was created
   at 0x511C08E: clone (in /lib64/libc-2.8.so)
   by 0x4E333A4: 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 0x4008F2: main (tc21_pthonce.c:86)

Thread #3 was created
   at 0x511C08E: clone (in /lib64/libc-2.8.so)
   by 0x4E333A4: 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 0x4008F2: main (tc21_pthonce.c:86)

Possible data race during read of size 4 at 0x601070 by thread #3
   at 0x40087A: child (tc21_pthonce.c:74)
   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)
 This conflicts with a previous write of size 4 by thread #2
   at 0x400883: child (tc21_pthonce.c:74)
   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 0x601070 is 0 bytes inside local var "unprotected2"
 declared at tc21_pthonce.c:51, in frame #0 of thread 3
```

Table 30: Primer izlaza iz Helgrinda

Kao što možemo da vidimo iz tabele 30, Helgrind najpre ispisuje podatke gde su niti koje uzrokuju grešku kreirane. Glavni podatak o grešci počinje sa „`Possible data race during read`“. Zatim se ispisuje adresa gde se nesinhroni pristup memoriji dešava, kao i veličina memorije kojoj se pristupa. U nastavku Helrgind ispisuje gde druga nit pristupa istoj lokaciji. Na kraju, Helgrind pokrenut sa opcijom `-read-var-info=yes` ispisuje i samo ime promenljive kojoj se pristupa, kao i gde je ta promenljiva deklarisana.

8. Provera ispravnosti

Za proveru ispravnosti programskog paketa Velgrind koji je prilagođen arhitekturi MIPS, iskorišćeno je 1500 testova iz programskog skupa testova DejaGnu koji se inače koriste za ispitivanje rada programskih prevodioca. Svi testovi iz tog skupa uspešno se izvšavaju pod Velgrindom.

Velgrind predstavlja programski paket koji može da otkrije širok spektar grešaka. U sklopu Velgrinda se nalaze i preko 400 testova za ispitivanje pojedinačnih alata koji otkrivaju određenu grupu grešaka. Ovim testovima omogućena je lakša realizacija prilagođavanja Velgrinda arhitekturi MIPS. Uz pomoć ovih testova moguće je lokalizovati grešku koja je nastala prilikom analize originalnog koda. Takođe, dodati su testovi koji ispituju translaciju pojedinačnih MIPS instrukcija. U okviru ovih testova izvršena je detaljna provera translacije svih MIPS32 instrukcija za različite vrednosti. Provera ispravnosti je izvršena na sledećim alatima: Nulgrind, Lackey, Memcheck, Cachegrind, Callgrind i Massif. Takođe, dokazana je delimična funkcionalnost eksperimentalnih alata: DHAT, Ptrcheck i BBV. U [4] je prikazan način korišćenja pojedinačnih alata koji su ispitivani, kao i primeri programa sa greškama koje alati za Velgrind otkrivaju.

Sama provera ispravnosti i verifikacija celog paketa Velgrind je izvršena na MIPS 74kc čipu koji radi na taktu od 1GHz. U nastavku je dat spisak svih testova koji se ne izvršavaju uspešno a koji dolaze u okviru samog paketa i služe za verifikaciju Velgrinda.

```
== 440 tests, 53 stderr failures, 2 stdout failures, 0 stderrB
failures, 0 stdoutB failures, 2 post failures ==
memcheck/tests/atomic_incs                (stdout)
memcheck/tests/fprw                       (stderr)
memcheck/tests/origin4-many               (stderr)
```

```
memcheck/tests/origin5-bz2                (stderr)
memcheck/tests/origin6-fp                 (stderr)
memcheck/tests/sigkill                    (stderr)
helgrind/tests/annotate_rwlock            (stderr)
helgrind/tests/free_is_write              (stderr)
helgrind/tests/hg02_deadlock              (stderr)
helgrind/tests/hg03_inherit               (stderr)
helgrind/tests/hg04_race                  (stderr)
helgrind/tests/hg05_race2                 (stderr)
helgrind/tests/locked_vs_unlocked1_fwd    (stderr)
helgrind/tests/locked_vs_unlocked1_rev    (stderr)
helgrind/tests/locked_vs_unlocked2        (stderr)
helgrind/tests/locked_vs_unlocked3        (stderr)
helgrind/tests/pth_barrier1               (stderr)
helgrind/tests/pth_barrier2               (stderr)
helgrind/tests/pth_barrier3               (stderr)
helgrind/tests/rwlock_race                (stderr)
helgrind/tests/tc01_simple_race           (stderr)
helgrind/tests/tc05_simple_race           (stderr)
helgrind/tests/tc06_two_races             (stderr)
helgrind/tests/tc06_two_races_xml         (stderr)
helgrind/tests/tc09_bad_unlock            (stderr)
helgrind/tests/tc14_laog_dinphils         (stderr)
helgrind/tests/tc16_byterace              (stderr)
helgrind/tests/tc18_semabuse              (stderr)
helgrind/tests/tc19_shadowmem             (stderr)
helgrind/tests/tc20_verifywrap            (stderr)
helgrind/tests/tc21_pthonce               (stderr)
helgrind/tests/tc22_exit_w_lock           (stderr)
helgrind/tests/tc23_bogus_condwait        (stderr)
drd/tests/annotate_barrier                (stderr)
drd/tests/annotate_ignore_read            (stderr)
drd/tests/annotate_order_1                (stderr)
drd/tests/annotate_order_2                (stderr)
drd/tests/annotate_order_3                (stderr)
drd/tests/annotate_smart_pointer          (stderr)
drd/tests/annotate_spinlock               (stderr)
drd/tests/annotate_static                 (stderr)
drd/tests/annotate_trace_memory           (stderr)
```

```
drd/tests/fp_race                         (stderr)
drd/tests/pth_cancel_locked               (stderr)
drd/tests/pth_cleanup_handler             (stderr)
drd/tests/pth_once                        (stderr)
drd/tests/sem_as_mutex                    (stderr)
drd/tests/sem_as_mutex3                   (stderr)
drd/tests/sem_open                        (stderr)
drd/tests/sem_open3                       (stderr)
drd/tests/tc04_free_lock                  (stderr)
drd/tests/tc09_bad_unlock                 (stderr)
exp-bbv/tests/mips-linux/ll               (stdout)
exp-bbv/tests/mips-linux/ll               (stderr)
exp-bbv/tests/mips-linux/million          (stderr)


perl tests/vg_regtest memcheck cachegrind callgrind massif lackey none
helgrind drd exp-bbv exp-dhat
```

Table 31: Spisak testova koji padaju

9. Zaključak

U ovom radu opisan je proces prilagođavanja paketa Velgrind novoj arhitekturi (MIPS). Opisani su teorijski principi rada Velgrinda, kao i njihove zavisnosti od platforme na kojoj se izvršavaju. Prikazan je detaljan princip rada alata Memcheck, Cachegrind i Helgrind sa detaljnim opisom transformacija koje oni unose u međukod, kao i šta se tim transformacijama dobija. Prikazana je zavisnost izvršavanja prilagođenog Velgrinda ciljnoj platformi. Prikazani su detaljni elementi međusobne komunikacije između Velgrinda i sistema na kojem se izvršava (sistemski pozivi, zauzimanje memorije, ostalo). Navedeni su svi testovi koje je bilo potrebno napisati da bi se uspešno verifikovao zadatak. U posebnom poglavlju je dat prikaz rezultata testova.

Programi koji se izvršavaju posredstvom Velgrinda se izvršavaju od 20 do 100 puta sporije nego kada se izvršavaju samostalno. Analizom koda posredstvom Velgrinda može se uštedeti dragoceno vreme koje se može izgubiti ako pretražujemo greške u kodu bez korišćenja alata. Takođe, neke greške je izuzetno teško naći, što znatno utiče na kvalitet programa ukoliko se one ne otkriju.

Glavna mana prikazanog rešenja odnosi se na problem analize programa koji implementira atomski čitaj-modifikuj-piši region. Naime, razmak između LL i SC instrukcije, koje na MIPS arhitekturi čine osnovne elemente atomskog čitaj-modifikuj-piši regiona, treba da bude što manji. Između te dve instrukcije ne sme da postoji nijedna druga instrukcija koja čita ili piše sadržaj u memoriju. Prilikom instrumentalizacije ovih instrukcija nekim od alata Velgrinda, pozivaju se funkcije koje vrše sam proces instrumentalizacije. Na taj način se atomski region između te dve instrukcije u nekim slučajevima prekida što dovodi do neuspešnog izvršenja instrukcije SC. Samim tim, program koji se analizira upada u mrtvu petlju usled neuspešno izvršene instrukcije SC.

Dalji tok razvoja podrške paketa Velgrinda arhitekturi MIPS bi se odnosio na dodavanje podrške GDB serveru, rešenje problema atomskog čitaj-modifikuj-piši regiona, kao i na podršku za MIPS64 i MIPS-DSP instrukcijski set.

10. Literatura

[1] Nicholas Nethercote, *Dynamic Binary Analysis and Instrumentation,* PhD Dissertation, University of Cambridge, November 2004.

[2] Julian Seward, Nicholas Nethercote: *A Framework for Heavyweight Dynamic Binary Instrumentation,* Proceedings of ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI 2007), San Diego, California, USA, June 2007.

[3] Julian Seward, Nicholas Nethercote: *Using Valgrind to detect undefined value errors with bit-precision,* Proceedings of the USENIX'05 Annual Technical Conference, Anaheim, California, USA, April 2005.

[4] Valgrind User Manual - http://valgrind.org/docs/manual/manual.html