

УНИВЕРЗИТЕТ У БЕОГРАДУ
МАТЕМАТИЧКИ ФАКУЛТЕТ

Александра Караџић

**АЛАТ *VALGRIND* - ИМПЛЕМЕНТАЦИЈА
КОНВЕНЦИЈЕ *FPXX* ЗА АРХИТЕКТУРУ
*MIPS***

мастер рад

Београд, 2017.

Ментор:

др Милена ВУЈОШЕЦИЋ ЈАНИЧИЋ, доцент
Универзитет у Београду, Математички факултет

Чланови комисије:

др Филип МАРИЋ, ванредни професор
Универзитет у Београду, Математички факултет

др Јелена ГРАОВАЦ, доцент
Универзитет у Београду, Математички факултет

Датум одбране: 15. јануар 2016.

Деди

Наслов мастер рада: Алат *Valgrind* - имплементација конвенције *FPXX* за архитектуру *MIPS*

Резиме:

Кључне речи: анализа, геометрија, алгебра, логика, рачунарство, астрономија

Садржај

1	Увод	1
2	Архитектура <i>MIPS</i>	2
2.1	Регистри у MIPS-у	2
2.2	Floating point регистри у MIPS-у	5
2.3	FPXX конвекција	6
3	Valgrind	7
3.1	О Valgrindu	8
3.2	Memcheck	12
3.3	Cachgrind	21
3.4	Helgrind	27
3.5	Callgrind	34
3.6	Massif	34
3.7	DRD	34
4	FPXX	37
5	Закључак	38
	Библиографија	39

Глава 1

Увод

Глава 2

Архитектура *MIPS*

CISC и *RISC*

Термин архитектура у рачунарству се користи да опише апстрактну машину која се програмира, а не стварну имплементацију те машине. Архитектура процесора у суштини дефинише скуп инструкција и регистара. Архитектура и скуп инструкција се једним именом називају ISA (скр. *Instruction Set Architecture*) [3].

MIPS

MIPS је најелегантнија архитектура међу свим активним RISC архитектура, чак и по мишљењу конкуренције. Елеганција није довољна да би се освојило тржиште, али MIPS микропроцесори су успели бити међу најефикаснији сваком генерацијом остајући међу најједноставнијима. MIPS процесори је један од RISC процесора, рођеног у плодном периоду академских истраживања и развоја. RISC (енг. *Reduced Instruction Set Computing*) је атрактивни акроним, [3]

2.1 Регистри у MIPS-у

Регистри представљају малу, веома брзу меморију, која је део процесора. MIPS процесори могу вршити операције само над садржајима регистара и специјалним константама које су део инструкције.

У MIPS архитектури, постоји 32 регистара опште намене. Само два регистра се понашају другачије од осталих регистара:

\$0 - Увек враћа нулу, без обзира коју му се вредност додели

\$31 - Увек се користи за адресу повратка из функције на коју се скочи инструкцијом *jal*

Сви ови регистри су идентични и могу се користити за било коју инструкцију (може се чак користити и регистар \$0 као дестинација, мада ће резултат да нестане).

Регистри опште намене су описани у наставку:

at - Резервисан за псеудоинструкције које асемблер генерише

v0, v1 - Користи се за враћање резултата при повратку из неке функције. Резултат може бити целобројног типа или број записан у фиксном зарезу.

a0 - a3 - Користи се за прослеђивање прва 4 аргумената функције која се позива

t0 - t9 - по конвенцији која је описана

s0 - s7 - по конвенцији која је описана

k0, k1 - Резервисано за систем прекида, који након коришћења не враћа садржај ових регистара на почетни. Како се прекид не позива из програма који се тренутно извршава, нема примене позивне конвенције. То значи да се садржај регистара које прекинути програм користи може пореметити. Због тога, систем прекида прво сачува садржаје регистара опште намене, који су важни за програм који се у том тренутку извршавао, и чији садржај планира да мења. У те сврхе се користе ови регистри.

gp - Користи се у различите сврхе. У коду који не зависи од позиције (енг. *Position Independent Code* скраћено PIC), свом коду и подацима се приступа преко табеле показивача, познате као GOT (скраћено од енгл. *Global Offset Table*). Регистар *\$gp* показује на ту табелу. PIC је код који се може извршавати на било којој меморијској адреси, без модификација. PIC се најчешће користи за дељење библиотеке, при чему се заједнички код библиотеке може учитати у одговарајуће локације адресних простора различитих програма који је користе.

У регуларном коду који зависи од позиције, регистар **\$gp** се користи као

показивач на средину у статичкој меморији. То значи да се подацима који се налазе 32 KB лево или десно од адресе која се налази у овом регистру може приступити помоћу једне инструкције. Дакле, инструкције *load* и *store* које се користе за читавање, односно складиштење података, се могу извршити у само једној инструкцији, а не у две као што је иначе случај. У пракси се на ове локације смештају глобали подаци који не заузимају много меморије. Оно што је битно је да овај регистар не користи сви системи за компилацију и сва окружења за извршавање.

sp - Показивач на стек. Оно што је битно је да стек расте наниже. Потребне су специјалне инструкције да би се показивач на стек повећао и смањено, тако да *MIPS* ажурира стек само при позиву и повратку из функције, при чему је за то одговорна функција која је позвана. *sp* се при уласку у функцију прилагођава на најнижу тачку на стеку којој ће да приступати у функцији. Тако ј е омогућено да компилатор може да приступи поменљивама на стеку помоћу константног помераја у односу на *\$sp*.

fp - Познат и као *\$s8*, показивач на стек оквир. Користи се од стране функције, за праћење стања на стеку, за случај да из неког разлога компилатор или програмер не могу да израчунају померај у односу на *\$sp*. То се може догодити уколико програм врши проширење стека, при чему се вредност проширења рачуна у току извршавања. Ако се дно стека не може израчунати у току превођења, не може се приступити променљивама помоћу *\$sp*, па се на почетку функције *\$fp* иницијализује на константну позицију која одговара стек оквиру функције. Ово је локално за функцију.

ra - Ово је подразумевани регистар за смештање адресе вратка и то је подржано кроз одговарајуће инструкције скока. Ово се разликује од конвенција које се користе на архитектурама *ц86*, где инструкција позива функције адресз повратка смешта на стек. При уласку у функцију регистар *ra* обично садржи адресу повратка функције, тако да се функције углавном завршавају инструкцијом *jr \$ra*, али у принципу, може се користити и неки други регистар. Због неких оптимизација које врши процесор, препоручује се коришћење регистра *\$ra*. Функције које позивају друге функције морају сачувати садржај регистра *\$ra*.

Постоје два специјална регистра *Hi* и *Lo*, који се користе само при множењу и дељењу. ово нису регистри опште намене, те се не користе при другим инструкцијама. Не може им се приступити директно, већ постоје специјалне инструкции *mfhi* и *mflo* за премештање садржаја ових регистра. Инструкција *mfhi* је облика *mfhi rd*, и она премешта садржај регистар *Hi* у регистар *rd*, док инструкција *mflo* премешта садржај регистра *Lo*.

2.2 Floating point регистри у MIPS-у

MIPS архитектура користи два формата FP (скр. *Floating Point*) препоручена од стране IEEE 754:

- *Једнострука прецизност* (енг. *Single precision*) - Користи се 32 бита за чување у меморији. MIPS компајлери користе једноструку прецизност за променљиве типа *float*
- *Двострука прецизност* (енг. *Double precision*) - Користи се 64 бита за чување у меморији. С компајлери користе двоструку прецизност за *C double* типове података.

Начин да се две речи ширине 32 бита се смештају у меморију као једна реч ширине од 64 бита је начин смештања у меморији (виша половина битова прво, или нижа половина битова прво) и зависи од начина смештања у меморији.

Стандарт IEEE 754 је веома захтеван и поставио је два велика проблема. Први, омогућавање детекције неуобичајних резултата доводи проточну обраду (енг. *pipeline*) тешком. Постоји опција да се имплементира IEEE механизам сигнализације изузетака, али је проблем да се детектују случајеви када хардвер не може да произведе исправан резултат и потребна му је помоћ.

Када се IEEE изузетак деси требало би обавестити и корисника, ово би требало бити синхрно; након заустављања корисник би хтео да види све предходно извршене инструкције и све FP регистре који су у *preinstruction* стању и желе да се увере да ни једна следећа инструкција нема никакав ефекат.

У MIPS архитектури, хардверска заустављања су била овако одрађена. Ово заправо ограничава могућности проточне обраде FP операција, јер се не може извршити следећа инструкција све док хардвер може бити сигуран да операција FP неће произвести заустављање. Зарад избегавања додавања времена за

извршавање, FP операције морају да одлуче да ли ће доћи до заустављања у првој фази.

2.3 FPXX конвекција

Глава 3

Valgrind

Valgrind је платформа за прављење алата за динамичку бинарну анализу кода. Динамичка анализа обухвата анализу корисничког програма у извршавању, док бинаран анализа обухвата анализу на нивоу машинског кода, снимљеног или као објектни код (неповезан) или као извршни код (повезан). Постоје *Valgrind* алати који могу аутоматски да детектују проблеме са меморијом, процесима као и да изврше оптимизацију самог кода. *Valgrind* се може користити и као алат за прављење нових алата. *Valgrind* дистрибуција тренутно броји следеће алате: детектор меморијских грешака, детектор грешака нити, оптимизатор скривене меморије и скокова, генератор графа скривене меморије и предикције скока и оптимизатор коришћења динамичке меморије. *Valgrind* ради на следећим архитектурама:

X86/Linux, AMD64/Linux, ARM/Linux, ARM64/Linux, PPC32/Linux, PPC64/Linux, PPC64LE/Linux, S390X/Linux, MIPS32/Linux, MIPS64/Linux, X86/Solaris, AMD64/Solaris, ARM/Android (2.3.x и новује), ARM64/Android, X86/Android (4.0 и новује), MIPS32/Android, X86/Darwin and AMD64/Darwin (Mac OS X 10.12).

У наредним поглављима биће детаљно описана структура *Valgrind* и његових алата, као и начин употребе са примерима проблема са којима се програмери свакодневно сусрећу.

3.1 О Valgrindu

Алат за динамичку анализу кода се креира као додатак, писан у С програмског језику, на језгро *Valgrind*.

Језгро Valgrinda + алат који се додаје = Алат Valgrinda

Језгро *Valgrind*-а омогућава извршавање клијетског програма, као и снимање извештаја који су настали приликом анализе самог програма.

Алати *Valgrind*-а користе методу бојења вредности. Они заправо сваки регистар и меморијску вредност „боје” (замењују) са вредношћу која говори нешто додатно о оригиналној вредности.

Сви *Valgrind* алати раде на истој основи, иако информације које се емитују варирају. Информације које се емитују могу се искористити за отклањање грешака, оптимизацију кода или било коју другу сврху за коју је алат дизајниран.

Сваки *Valgrind*-ов алат је статички повезана извршна датотека која садржи код алата и код језгра. Извршна датотека *valgrind* представља програм омотач који је на основу `—tool` опције бира алат који треба покренути и покреће га помоћу системског позива *execve*. Извршна датотека алата статички је линкована тако да се учитава почев од неке адресе која је обично доста изнад адресног простора који користе класичан кориснички програм (на *x86/Linux* и *MIPS/Linux* користи се адреса 0x38000000). У ретким случајевима, када та адреса није потреба, *Valgrind* се може прекомпајлирати да користи неку другу адресу. *Valgrind*-ово језгро прво иницијализује под-систем као што су менаџер адресног простора, и његов унутрашњи алокатр меморије и затим учитава клијентову извршну датотеку. Потом се иницијализују *Valgrind*-ови субсистеми као што су транслациона табела, апарат за обраду сигнала, распоређивач нити и учитавају се информације за дебаговање клијента, уколико постоје. Од тог тренутка *Valgrind* има потпуну контролу и почиње са преводињем и извршавањем клијентског програма. Може се рећи да *Valgrind* врши JIT (*Just In Time*) преводиње машинског кода програма у машински код програма допуњен инструментацијом. Ниједан део кода клијента се не извршава у свом изворном облику. Алат се умеће у оригинални код на почетку, затим се нови код преводи, сваки основни блок појединачно, који се касније извршава. Процес преводиња се састоји из рашчлањивања оригиналног машинског кода у IR (скр. *intermediate representation*) који се касније инструментализује са алтом и поново преводи у нови машински код.

Резултат свега овога се назива транслација, која се чува у меморији и која се извршава по потреби. Језгро троши највише времен на сам процес прављења, проналажења и извршавања транслације. Оригинални код се никада се извршава. Једини проблем који се овде може догодити је ако се врши транслација кода који се мења током извршавања програма.

IR има неке *RISC* одлике као што су *load/store*, свака операција ради само једну ствар, кад се линеаризује све операције раде само на привременим промеливама и литералима. Да би се подржале све целобројне, FP и SIMD операције над различитим величинама потребно је више од 200 примитивних аритметичко-логичких инструкција.

Постоје многе компликације које настају приликом смештања два програма у један проц (клијентски програм и програм алата). Многи ресурси се деле између ова два програма, као што су регистри или меморија. Такође, алат *Valgrind*-а не сме да се одрекне тоталне контроле над извршавањем клијетског програма приликом извршавања системских позива, сигнала и нити.

Основни блок

Valgrind дели оригинални код у секвенце које се називају основни блокови. Основни блок је праволинијска секвенца машинског кода, на чији се почетак скаче, а која се завршава са скоком, позивом функције или повратком. Сваки код програма који се анализира поново се преводи на захтев, појединачно по основним блоковима, непосредно пре самог извршавања самог блока. Ако узмемо да су основни блокови клијетског кода $BB1$, $BB2$, ... онда преведене основне блокове обележавамо са $t(BB1)$, $t(BB2)$, ... Величина основног блока је ограничена на максимално 60 машинских инструкција. На процесорима *MIPS*, инструкције скока и гранања имају такозвано „одложено извршавање”. То значи да се приликом извршавања тих инструкција извршава и инструкција која се налази непосредно иза инструкције гранања или скока. У случају да је последња шездесета инструкција основног блока инструкција гранања, *Valgrind* читава и инструкцију која се налази непосредно иза ње, односно шездесет и прва инструкција. Тиме се омогућава конзистентно извршавање програма који се анализира, као и у случају да се програм извршава без посредства *Valgrind*-а. Уколико након извршених 60 инструкција *Valgrind* није наишао на инструкцију гранања, секвенца инструкција се дели на два или више основних блокова, који се извршавају један за другим.

Системски позиви

Апликациони програми комуницирају са оперативним системом помоћу системских позива (енг. **system calls**), тј. преко операција (функција) које дефинише оперативни систем. Системски позиви се реализују помоћу система прекида: кориснички програм поставља параметре системског позива на одређене меморијске локације или регистре процесора, иницира прекид, оперативни систем преузима контролу, узима параметре, извршава тражене радње, резултат ставља на одређене меморијске локације или у регистре и враћа контролу корисничком програму. Апликација која жели да користи неке од ресурса, као што су меморија, процесор или улазно/излазни уређаји, комуницира са језгром оперативног система користећи системске позиве. Језгро оперативног система дели виртуелну меморију на корисничку меморију и системску меморију. Системска меморија је одређена за само језгро оперативног система, његова проширања, као и за управљачке програме. Кориснички простор је део меморије где се налазе све корисничке апликације приликом њиховог извршавања. Корисничке апликације могу да приступе улазно/излазним уређајима, виртуелној меморији, датотекама и другим ресурсима језгра оперативног система користећи само системске позиве. Системски позиви обезбеђују спрегу између програма који се извршава и оперативног система. Генерално, реализују се на асемблерском језику, али новији виши програмски језици, попут језика C и C++, такође омогућавају реализацију системског позива. Програмом који се извршава може проследити параметре оперативном систему на три начина:

- прослеђивање параметара у регистрима процесора;
- постављањем параметара у меморијској табели, при чему се адреса табеле прослеђује у регистру процесора;
- постављањем параметара на врх стека (енг. *push*), које оперативни систем „скида” (енг. *pop*).

Системски позиви се извршавају без посредства *Valgrind*-а, зато што језгро *Valgrind*-а не може да прати њихово извршавање у самом језгру оперативног система.

Транслација

У наставку су описани кораци које *Valgrind* извршава приликом анализе програма. Постоји осам фаза транслације. Све фазе осим инструментације коју обавља алат *Valgrind*-а, обавља језгро *Valgrind*-а.

- **Дисасемблирање** - процес превођења машинског кода у еквивалентни асемблерски код. *Valgrind* врши превођење машинског кода у интерни скуп инструкција која се називају међукод инструкције. Међукод представља редуковани скуп инструкција (скр. енг. *RISC*). Ова фаза је зависна од архитектуре на којој се извршава.
- **Оптимизација 1** - Прва фаза оптимизације линеаризује *IR* репрезентацију. Примењују се неке стандардне оптимизације програмских преводаца као што су уклањање редуваног кода, елиминација подизраза, једноставно одмотавање петљи и сл.
- **Инструментација** - Блок кода у *IR* репрезентацији се прослеђује алату, који може произвољно да га трансформише. Приликом инструментације алат у задати блок додаје додатне *IR* операције, кјима проверава исправност рада програма.
- **Оптимизација 2** - Друга фаза оптимизације је једноставније од прве, укључује множење констати и уклањање мртвог кода.
- **Градња стабла** - Линеаризована *IR* репрезентација се конвертује натраг у стабло ради лакшег избора инструкција.
- **Одабир инструкција** - Стабло *IR* репрезентације се конвертује у листу инструкција које користе виртуалне регистре. Ова фаза се такође разликује у зависности од архитектуре на којој се извршава.
- **Алокација регистара** - Виртуални регистри се замењују стварним. По потреби се уводе пребацивања у меморију. Независна је за платформу, користи позив функција које налазе из који се регистара врши читање и у које се врши упис.
- **Асемблирање** - Изабране инструкције се енкодују на одговарајући начин и смештају у блок меморије. Ова фаза се такође разликује у зависности од архитектуре на којој се извршава. [3]

3.2 Memcheck

Меморијске грешке често се најтеже детектују, а самим тим и најтеже отклањају. Разлог томе је што се такви проблеми испољавају недетерминистички и није их лако репродуковати. *Memcheck* је алат који детектује меморијске грешке корисничког програма. Како не врши анализу изворног кода већ машинског, *Memcheck* има могућност анализе програма писаног у било ком језику.

За програме писане у језицима С и С++ детектује следеће уобичајне проблеме:

- Приступање недопуштеној меморији, на пример преписивање блокова на хипу, преписивање врха стека и приступање меморији која је већ ослобођена.
- Коришћење недефинисаних вредности, вредности које нису иницијализоване или које су изведене од других недефинисаних вредности.
- Неисправно ослобађање хип меморије, као што је дупло ослобађање хип блокова или неупареног коришћења функција *malloc/new/new[]* и *free/delete/delete[]*.
- Преклапање параметара прослеђених функцијама (нпр. преклапање *src* и *dst* показивача код функције *memcpy*).
- Цурење меморије.

Пуштање преведеног програма кроз *Valgrind*, врши се извршавањем следеће линије у терминалу:

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

`--tool` = опција одређује који алат ће *Valgrind* пакет користити. Програм који ради под контролом *Memcheck*-а је обично 20 до 100 пута спорији него када се извршава самостално, због транслације кода. Излаз програм је повећан за излаз који додаје сам алат *Memcheck*, који се исписује на стандардном излазу за грешке.

Коришћење неиницијализованих вредности

На слици 3.1 је дат пример програма у коме користимо неиницијализовану променљиву. Грешка о коришћењу неиницијализоване вредности се генерише

```
#include <stdio.h>

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

Слика 3.1: Пример програма main.c

када програм користи променљиве чије вредности ниси иницијализоване, другим речима недефинисане.

Слика 3.2 приказује излаз *Valgrind*-а који детектује коришћење недефинисаних вредности у програму. Први део, односно пре три линије се штампају приликом покретања било ког алата који је у склопу *Valgrind*-а, у овом случају *Memcheck*. Следећи део нам показује поруке о грешкама које је *Memcheck* пронашао у програму. Последња линија приказује суму свих грешака које је алат пронашао и штампа се по завршетку рада.

На овој слици је приказан излаз из *Valgrind*-а када се открије коришћење недефинисаних вредности. У програму недефинисана променљива може више пута да се копира, *Memcheck* прати све то, бележи податке о томе, али не пријављује грешку. У случају да се недефинисане вредности користе на начин да од те вредности зависи даљи ток програма или ако је потребно приказити вредности недефинисане променљиве, *Memcheck* пријављује грешку. да би могли да видимо главни извор коришћења недефинисаних вредности у програму, додаје се опција *--trace-origins=yes*.

Коришћење неиницијализоване или неадресиране вредности у системском позиву

Memcheck прати све параметре системског позива:

- Проверава све параметре појединачно, без обзира да ли су иницијализовани.
- Проверава да ли системски позив треба да чита из меморије која је дефи-

```
==7070== Memcheck, a memory error detector
==7070== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==7070== Using Valgrind-3.14.0.GIT and LibVEX; rerun with -h for copyright info
==7070== Command: ../main
==7070==
==7070== Conditional jump or move depends on uninitialised value(s)
==7070==   at 0x4E814CE: vfprintf (vfprintf.c:1660)
==7070==   by 0x4E8B3D8: printf (printf.c:33)
==7070==   by 0x400548: main (in /export/main)
==7070==
==7070== Use of uninitialised value of size 8
==7070==   at 0x4E8099B: _itoa_word (_itoa.c:179)
==7070==   by 0x4E84636: vfprintf (vfprintf.c:1660)
==7070==   by 0x4E8B3D8: printf (printf.c:33)
==7070==   by 0x400548: main (in /export/main)
==7070==
==7070== Conditional jump or move depends on uninitialised value(s)
==7070==   at 0x4E809A5: _itoa_word (_itoa.c:179)
==7070==   by 0x4E84636: vfprintf (vfprintf.c:1660)
==7070==   by 0x4E8B3D8: printf (printf.c:33)
==7070==   by 0x400548: main (in /export/main)
==7070==
==7070== Conditional jump or move depends on uninitialised value(s)
==7070==   at 0x4E84682: vfprintf (vfprintf.c:1660)
==7070==   by 0x4E8B3D8: printf (printf.c:33)
==7070==   by 0x400548: main (in /export/main)
==7070==
==7070== Conditional jump or move depends on uninitialised value(s)
==7070==   at 0x4E81599: vfprintf (vfprintf.c:1660)
==7070==   by 0x4E8B3D8: printf (printf.c:33)
==7070==   by 0x400548: main (in /export/main)
==7070==
==7070== Conditional jump or move depends on uninitialised value(s)
==7070==   at 0x4E8161C: vfprintf (vfprintf.c:1660)
==7070==   by 0x4E8B3D8: printf (printf.c:33)
==7070==   by 0x400548: main (in /export/main)
==7070==
x = 0
==7070==
==7070== HEAP SUMMARY:
==7070==   in use at exit: 0 bytes in 0 blocks
==7070==   total heap usage: 0 allocs, 0 frees, 0 bytes allocated
==7070==
==7070== All heap blocks were freed -- no leaks are possible
==7070==
==7070== For counts of detected and suppressed errors, rerun with: -v
==7070== Use --track-origins=yes to see where uninitialised values come from
==7070== ERROR SUMMARY: 6 errors from 6 contexts (suppressed: 0 from 0)
```

Слика 3.2: Детекција неиницијализованих вредности

нисана у програму. *Memcheck* проверава да ли је цела меморија адресирана и иницијализована.

- Ако системски позив треба да пише у меморију, *Memcheck* проверава да ли је та меморија адресирана.

После системског позива *Memcheck* прецизно ажурира информације о промени у меморији које су постављене у системског позиву.

```
#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]);
}
```

Слика 3.3: Пример програма main1.c

На слици 3.3 дат је пример позива системског позива са неисправним параметрима.

На слици 3.4 је извештај који даји добијамо након анализе програма main1.c. Можемо да видимо да је *Memcheck* приказао информације о коришћењу неиницијализованих вредности у системским позивима. Прва грешка приказује да параметар *buf* системског позива *write()* показује на неиницијализовану вредност. Друга грешка приказује да је податак који се прослеђује системског позиву *exit()* недефинисан. Такође, приказане су и линије у самом програму где се ове вредности користе.

Недопуштено ослобађање меморије

На слици 3.5 дат је пример програма у коме се нелегално ослобађа меморија. *Memcheck* прати свако алоцирање меморије које програм направи употребом функција *malloc/new*, тако да он увек поседује информацију да ли су аргументи који се прослеђују функцијама *free/delete* легитимни или не. У нашем примеру, програм ослобађа исту меморијску зону два пута. Извештај о недопуштеном ослобађању меморије приказан је на слици 3.6.

```
rtrk@rtrkw579-lin:/export/valgrind$ ./vg-in-place --leak-check=yes --show-reachable=yes --track-origins=yes ../main
==14143== Memcheck, a memory error detector
==14143== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==14143== Using Valgrind-3.14.0.GIT and LibVEX; rerun with -h for copyright info
==14143== Command: ../main
==14143==
==14143== Syscall param write(buf) points to uninitialised byte(s)
==14143==   at 0x4F26390: __write_nocancel (syscall-template.S:81)
==14143==   by 0x4005F6: main (za_master.c:8)
==14143== Address 0x5200040 is 0 bytes inside a block of size 10 alloc'd
==14143==   at 0x4C2AC23: malloc (vg_replace_malloc.c:299)
==14143==   by 0x4005CE: main (za_master.c:6)
==14143== Uninitialised value was created by a heap allocation
==14143==   at 0x4C2AC23: malloc (vg_replace_malloc.c:299)
==14143==   by 0x4005CE: main (za_master.c:6)
==14143==
==14143== Syscall param exit_group(status) contains uninitialised byte(s)
==14143==   at 0x4EFC109: _Exit (_exit.c:32)
==14143==   by 0x4E7316A: __run_exit_handlers (exit.c:97)
==14143==   by 0x4E731F4: exit (exit.c:104)
==14143==   by 0x400603: main (za_master.c:9)
==14143== Uninitialised value was created by a heap allocation
==14143==   at 0x4C2AC23: malloc (vg_replace_malloc.c:299)
==14143==   by 0x4005DC: main (za_master.c:7)
==14143==
==14143==
==14143== HEAP SUMMARY:
==14143==   in use at exit: 14 bytes in 2 blocks
==14143==   total heap usage: 2 allocs, 0 frees, 14 bytes allocated
==14143==
==14143== 4 bytes in 1 blocks are still reachable in loss record 1 of 2
==14143==   at 0x4C2AC23: malloc (vg_replace_malloc.c:299)
==14143==   by 0x4005DC: main (za_master.c:7)
==14143==
==14143== 10 bytes in 1 blocks are still reachable in loss record 2 of 2
==14143==   at 0x4C2AC23: malloc (vg_replace_malloc.c:299)
==14143==   by 0x4005CE: main (za_master.c:6)
==14143==
==14143== LEAK SUMMARY:
==14143==   definitely lost: 0 bytes in 0 blocks
==14143==   indirectly lost: 0 bytes in 0 blocks
==14143==   possibly lost: 0 bytes in 0 blocks
==14143==   still reachable: 14 bytes in 2 blocks
==14143==   suppressed: 0 bytes in 0 blocks
==14143==
==14143== For counts of detected and suppressed errors, rerun with: -v
==14143== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 0 from 0)
```

Слика 3.4: Пример излаза за програм main1.c

Memcheck је пријавио да је програм покушао два пута да ослободи неку меморијску зону. Такође, *Memcheck* ће нам пријавити и ако програм покуша да ослободи меморијску зону преко показивача који не показује на почетак динамичке меморије.

Детекција цурења меморије

Memcheck бележи податке о свим динамичким блоковима који су алоцирани током извршавања програма позивом функција *malloc()*, *new()* и др. Када

```
#include <stdio.h>

int main( void )
{
    char *p;
    p = (char) malloc(19);
    p = (char) malloc(12);
    free(p);
    free(p);
    p = (char) malloc(16);
    return 0;
}
```

Слика 3.5: Пример програма main2.c

програм прекине са радом, *Memcheck* тачно зна колико меморијских блокова није ослобођено.

Ако је опција *--leak-check* адекватно подешена, за сваки неослобођени блок *Memcheck* одређује да ли је могуће приступити том блоку преко показивача.

Постоје два начина да приступимо садржају неког меморијског блока преко показивача. Први начин је преко показивача који показује на почетак меморијског блока. Други начин је преко показивача који показује на садржај унутар меморијског блока.

Постоји неколико начина да сазнамо да ли постоји показивач који показује на унутрашњост неког меморијског блока:

- Постојао је показивач који је иницијално показивао на почетак блока, али је намерно (или ненамерно) померен да показује на унутрашњост блока.
- Ако постоји нежељена вредност у меморији, која је у потпуности неповезана и случајна.
- Ако постоји показивач на низ C++ објеката (који поседују деструкторе) који су алоцирани са *new*. У овом случају, неки компајлери чувају „магични показивач” који садржи дужину низа од почетка блока.

На слици 3.7 је приказано 9 могућих случајева када показивачи показују на неке меморијске блокове. Сваки могући случај када показивач показује на неки меморијски блок може се представити са једним од приказаних 9 случајева.


```

rtrk@rtrkw579-lin:/export/valgrind$ ./vg-in-place --leak-check=yes --show-reachable=yes --track-origins=yes ../main
==15372== Memcheck, a memory error detector
==15372== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==15372== Using Valgrind-3.14.0.GIT and LibVEX; rerun with -h for copyright info
==15372== Command: ../main
==15372==
==15372== Invalid free() / delete / delete[] / realloc()
==15372==    at 0x4C2BD1D: free (vg_replace_malloc.c:530)
==15372==    by 0x4005B4: main (za_master.c:8)
==15372== Address 0xfffffffffffffa0 is not stack'd, malloc'd or (recently) free'd
==15372==
==15372== Invalid free() / delete / delete[] / realloc()
==15372==    at 0x4C2BD1D: free (vg_replace_malloc.c:530)
==15372==    by 0x4005C0: main (za_master.c:9)
==15372== Address 0xfffffffffffffa0 is not stack'd, malloc'd or (recently) free'd
==15372==
==15372==
==15372== HEAP SUMMARY:
==15372==    in use at exit: 47 bytes in 3 blocks
==15372== total heap usage: 3 allocs, 2 frees, 47 bytes allocated
==15372==
==15372== 12 bytes in 1 blocks are definitely lost in loss record 1 of 3
==15372==    at 0x4C2AC23: malloc (vg_replace_malloc.c:299)
==15372==    by 0x4005A0: main (za_master.c:7)
==15372==
==15372== 16 bytes in 1 blocks are definitely lost in loss record 2 of 3
==15372==    at 0x4C2AC23: malloc (vg_replace_malloc.c:299)
==15372==    by 0x4005CA: main (za_master.c:10)
==15372==
==15372== 19 bytes in 1 blocks are definitely lost in loss record 3 of 3
==15372==    at 0x4C2AC23: malloc (vg_replace_malloc.c:299)
==15372==    by 0x40058E: main (za_master.c:6)
==15372==
==15372== LEAK SUMMARY:
==15372==    definitely lost: 47 bytes in 3 blocks
==15372==    indirectly lost: 0 bytes in 0 blocks
==15372==    possibly lost: 0 bytes in 0 blocks
==15372==    still reachable: 0 bytes in 0 blocks
==15372==    suppressed: 0 bytes in 0 blocks
==15372==
==15372== For counts of detected and suppressed errors, rerun with: -v
==15372== ERROR SUMMARY: 5 errors from 5 contexts (suppressed: 0 from 0)

```

Слика 3.6: Пример излаза за програм main2.c

Memcheck обједињује неке од ових случајева, тако да добијамо наредне четири категорије.

- „Још увек доступан”. Ово покрива примере 1 и 2 на слици 3.7. Показивач који показује на почетак блока или више показивача који показују на почетак блока су пронађени. Зато што постоје показивачи који показују на меморијску локацију која није ослобођена, програмер може да ослободи меморијску локацију непосредно пре завршетка извршавања програма.
- „Дефининитивно изгубљен”. Ово се односи на трећи случај на слици 3.7. Ово значи да је немогуће пронаћи показивач који показује на меморијску блок. Блок је проглашен изгубљеним, заузета меморија не може да се ослободи пре завршетка програма, јер не постоји показивач на њу.

```
(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
```

Слика 3.7: Пример показивача на меморијски блок

- „Индиретно изгубљен”. Ово покрива случајеве 4 и 9 на слици 3.7. Меморијски блок је изгубљен, не зато што не постоји показивач који показује на њега, него зато што су сви блокови који указују на њега изгубљени. На пример, ако имамо бинарно стабло и корен је изгубљен, сва деца чворови су индиректно изгубљени. С обзиром на то да ће проблем нестати ако се порави показивач на дефинитивно изгубљен блок који је узроковао индиректно губљење блока, *Memcheck* неће пријавити ову грешку уколико није укључена опција *--show-reachable=yes*.
- „Могуће изгубљен”. Ово су случајеви од 5 до 8 на слици 3.7. Пронађен је један или више више показивача на меморијски блок, али најмање један од њих показује на унутрашњост меморијског блока. То може бити само случајна вредност у меморији која показује на унутрашњост блока, али ово не треба сматрати у реду док се не разреши случај показивача који показује на унутрашњост блока.

Ако постоји забрана приказивања грешке за одређени меморијски блок, без обзира којој од горе поменутих категорија припада, на неће бити приказана.

На слици 3.8 је дат резиме цурења меморије који исписује *Memcheck*. Ако је укључена опција *--leak-check=yes*, *Memcheck* ће приказати детаљан извештај


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

Слика 3.8: Резиме цурења меморије

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

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

Слика 3.9: Извештај о цурењу меморије

о сваком дефинитивно или могуће изгубљеном блоку, као и о томе где је он алоциран. *Memcheck* нам не може рећи када, како или зашто је неки меморијски блок изгубљен. Генерано, програм не треба да има ниједну дефинитивно или могуће изгубљен блок на излазу.

На слици 3.9 је приказан извештај који нам даје *Memcheck* о дефинитивном губитку два блока величине 16 и 19 бајта, као линију у програму где су они алоцирани.

Због постојања више типова цурења меморије поставља се питање које цурење меморије на излазу из програма треба да посматрамо као „грешку”, а коју не. *Memcheck* користи следећи критеријум:

- *Memcheck* сматра да је цурење меморије „грешка” само ако је укључена опција `--leak-check=full`. Другим речима, ако подаци о цурењу меморије нису приказани, сматра се да то цурење није „грешка”.
- Дефинитивно и могуће изгубљени блокови се сматрају за праву „грешку”, док индиректно изгубљени и још увек доспуни блокови се не сматрају као грешка.

3.3 Cachgrind

Cachgrind је алат који симулира и прати приступ скривеној меморији машине на којој се програм, који се анализира, извршава. Он симулира скривену меморију машине, која има први ниво скривене меморије подељен у две одвојене независне секције: *I1* - секција брзе меморије у кој се смештају инструкције и *D1* - секција брзе меморије у којој се смештају подаци. Други ниво скривене меморије коју *Cachgrind* симулира је обједињен - *L2*. Овај начин конфигурације одговара многим модерним машинама.

Постоје машине које имају и трећи ниво брзе меморије *I3*. У том случају, *Cachgrind* симулира приступ трећем нивоу. Генерално гледано, *Cachgrind* симулира *L1*, *D1* и *LL* (последњи ниво скривене меморије).

Cachgrind прикупља следеће статистичке податке о програму који анализира (скраћенице које се користе даље у тексту су дате у заградама):

- *I* читање брзе меморије (*Ir*, што представља број извршених инструкција), *I1* број промашаја читања (*I1mr*) и број промашаја читања инструкција нивоа *LL* брзе меморије (*ILmr*).
- *D* читање брзе меморије (*Dr*, што је једнако броју читања меморије), *D1* број промашаја читања (*D1mr*, и број промашаја читања података нивоа *LL* брзе меморије (*DLmr*).
- *D* писања у брзу меморију (*Dw*, што је једнако броју писања у меморију), *D1* број промашаја писања у брзу меморију и број промашаја писања података у нивоу *LL* брзе меморије (*DLmw*).
- Број условно извршених грана (*Bc*) и број промашаја условно извршених грана (*Bcm*).
- Број индиректно извршених грана (*Bi*) и број промашаја индиректно извршених грана (*Bim*).

Приметимо да је број приступа *D1* делу брзе меморије једнак збиру *D1mr* и *D1mw*, док је укупан број приступа нивоу *LL* једнак збиру *ILmr*, *DLmr* и *DLmw* број приступа. Ова статистика се прикупља на нивоу целог програма, као и за појединачно на нивоу функција. Може се такође, добити и број приступа скривеној меморији за сваку линију кода у оригиналном програму. На модерним машинама *L1* промашај кошта око 10 процесорских циклуса, *LL* промашај

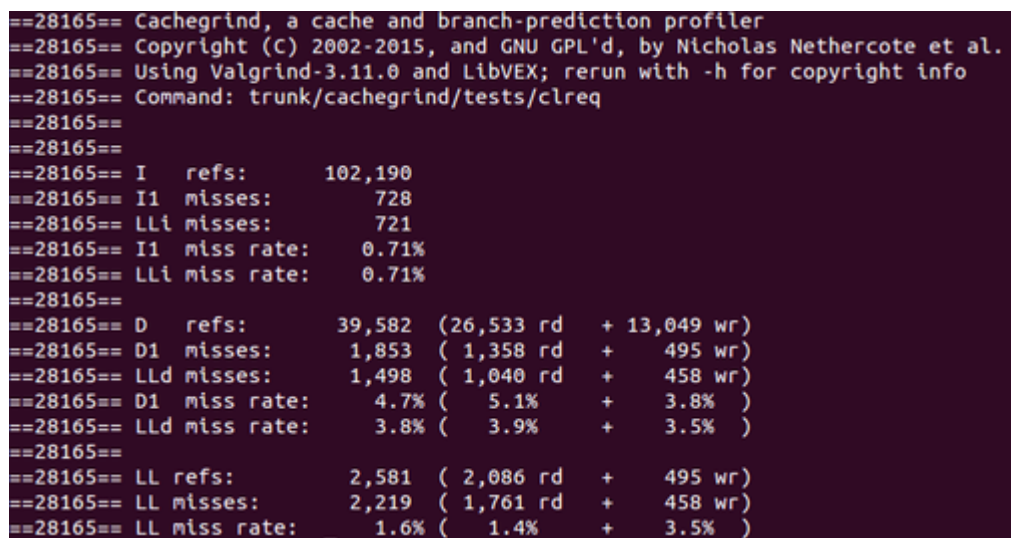
кошта око 200 процесорских циклуса, а промашаји условно и индиректно извршене гране од 10 до 30 процесорских циклуса.

Коришћење Cachgrind-а

На почетку коришћења алата *Cachgrind*, програм који желимо да анализирамо покрећемо самим *Cachgrind*-ом. На тај начин прикупљамо информације које су нам потребне за касније профилисање кода. Затим покрећемо алат *cg_annotate* у оквиру пакета *Valgrind* који нам приказује детаљан извештај о програму који анализирамо са *Cachgrind*-ом. Опционо, можемо да користимо алат *cg_merge* да сумирамо у једну датотеку више излаза које смо добили вишеструким покретањем *Cachgrind*-а над истим програмом. Ту датотетку касније користимо као улаз у *cg_annotate*. Такође, можемо да користимо алат *cg_diff* који прави разлику између више излаза из алата *Cachgrind*, које касније користимо као улаз у алат *cg_annotate*.

Покретање самог алата *Cachgrind* врши се извршавањем следеће линије у терминалу:

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



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

Слика 3.10: Извештај алата *Cachgrind*

Извршава програма кроз *Cachgrind* траје веома споро. Након завршетка рада, добијају се статистике као што је приказано на слици 3.10.

Cachgrind метаподаци

У наставку су описани метаподаци који се чувају у структурама.

Глобално стање брзе меморије. Прва структура која се налази у склопу *Cachgrind* метаподатака је глобано стање брзе меморије. Она представља три дела симулиране брзе меморије (*L1*, *D1*, *LL*). Њене вредности се освежавају приликом извршене сваке инструкције програма чија се брза меморија симулира, тачније, приликом позива функције које симулирају приступ брзој меморији циљне платформе. Функцијама се прослеђују информациј ео приступу брзој меморији, као што су адресе и величина меморије којој се приступа.

Симулација приступа брзој меморији има следеће карактеристике:

- Када се деси помашај уписа у брзу меморију, блок који је потребно уписати се семпта у *D1* део брзе меморије.
- Инструкције које содификују вредност меморије третирају се као читање брзе меморије. Наиме, инструкције које мењају садржај брзе меморије најпре читају садржај брзе меморије, модификују вредност и снимају нову вредност. Самим тим, упис у брзу меморију не може да изсазове промашај, јер је гарантован успешним читањем. Такође, циљ *Cachgrind*-а није да прикаже колико пута се приступа брзој меморији, већ да прикаже број промашаја приступа брзој меморији.
- Линија у брзој меморији, којој одговара садржај у меморији са директним приступом, одређује се као $(M + N - 1)$, где је величина линије = $2\hat{M}$ бајта, (величина брзе меморије / величина линија) = $2\hat{N}$ бајта.
- *L2* део брзе меморије реплицира све уносе у *L1* део брзе меморије.
- Онај блок у брзој меморији који се најмање користи ће бити избачен из брзе меморије уколико је потребно убацити нови блок података у брзу меморију.
- Са референцама које показују на две линије у кеш меморији рукује се на следећи начин:
 - уколико су пронађена оба блока у брзој меморији, рачунамо сам оједан погодат;

- уколико један блок нађемо у брзој меморији, а други не, рачунамо један промашај (и нула погодатака);
- уколико оба блока не пронађемо у брзој меморији, рачунамо један промашај (не два).

Параметри симулиране брзе меморије (величина брзе меморије, величина линије и асоцијативност) одређују се на један од два начина. Први наин је потребом *cruid* наредбе. Други начин представља ручно уношење параметара симулиране брзе меморије, прикиком покретања самог *Cachgrind*.

```
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
};
```

Слика 3.11: Структура централне табеле трошкова

```
typedef struct _instr_info instr_info;

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

Слика 3.12: Структура табеле информација о инструкцијама

Централна табела трошкова. Друга структура која чини метаподатке алата *Cachgrind* је централна табела трошкова. Свака линија у коду која се инструментализује, алоцира једну овакву табелу у коју смешта податке о приступу брзој меморији, броју погодака и промашаја приступа брзој меморији, који се дешавају приликом извршавања саме линије кода. На слици 3.11 приказане су структуре које представљају централну табелу трошкова. *ULong* је 64-битна целобројна вредност. Узимамо 64-битну вредност, јер број приступа може да буде већи него што може да се представи 32-битном целобројном вредношћу. У *CC* структури *m1* и *m2* представљају број промашаја за *L1* и *LL* део скривене меморије. Структура *lineCC* садржи три *CC* елемента: за читање *I* дела, за читање дела *D* и за писање у део *D* скривене меморије. Поље *next* је потребно јер је централна табела трошкова представљена као променљива „*heš*” табела. Поље *line* представља број линије у коду која одговара тој табели трошкова. Сам број линије није довољан да би се пронашла линија у коду којој одговара централна табела трошкова (име фајла је потребно). У пракси, то значи централна табела трошкова има три новца: трошкови су груписани по имену фајла, затим по имену функције и на крају по броју линије.

Табела информација о инструкцијама. Трећа структура која чини метаподатке алата *Cachgrind* је табела информација о инструкцијама. Она се користи за чување непроменљивих информација о самим инструкцијама током самог процеса инструментације. На овај начин се смањује величина додатог кода којим се анализира код. Повећава се брзина извршавања самог алата, јер се смањује број аргумената који се просеђују функцијама, које врше симулацију приступа број меморији.

Свакој инструментализованој инструкцији додељује се по једно поље у табели информација о инструкцијама, које садржи структуру *instr_info*, приказаној на слици 3.12. Поље *instr_addr* представља адресу инструкције *instr_size* представља величину инструкције изражену у бајтовима, *data_size* чува величину података коме инструкција приступа (0 уколико инструкција не приступа меморији) и *parent* показује на поље у табели трошкова за исту линију кода одакле је инструкција изведена.

Инструментација

Први корак приликом инструментације кода односи се на пролаз кроз све основне блокове појединачно ради пребројавања инструкција које се налазе у

њима. На основу овог броја се креира листа *instr _info* елемената, при чему сваки елемент листе одговара једној инструкцији у основном блоку.

У другом пролазу, *Cachgrind* врши категоризацију оригиналних инструкција, *Cachgrind* дели инструкције у следеће категорије:

- Инструкције које не приступају меморији, нпр. *move \$t3, \$a0*
- Инструкције које читају садржај меморије, нпр. *lw \$t3, 4(\$a0)*
- Инструкције које уписују садржај регистара у меморију, нпр. *sw \$t3, 4(\$a0)*
- Инструкције које модификују садржај меморијске локације
- Инструкције које читају садржај из једне меморијске локације и тај садржај уписују у другу меморијску локацију.

Свака инструкција система базираног на *MIPS* процесорима је растављена на више *UCode* инструкција, тако да *Cachgrind* одређује којој категорији оригинална инструкција припада на основу *LOAD* и *STORE UCode* инструкција. *Cachgrind* чита инфромације које помажу при отклањању грешака. На основу ових информација он креира елементе *lineCC* у централној табели трошкова. Затим иницијализује одређене *instr _info* елементе у низ који је иницијализован за сваки основни лбок појединачно (где је *n*-ти елемент *instr _info* одговара *n*-тој инструкцији у основном блоку). Када је иницијализовао све елементе *lineCC* и *instr _info* алат *Cachgrind* извршава процес инструментализације кода који се састоји из позива одговарајућих *C* функција, које симулирају приступ брзој меморији циљне платформе. Која *C* функција ће бити позвана зависи од категорије којој инструкција припада. Постоје само четири врста *C* функција које симулирају приступ брзој меморији, јер функције које припадају другој и четворој категорији позивају исту *C* функцију за симулирање приступа брзој меморији. Број параметара који се прослеђује *C* функција се, одређује на основу категорије којој та функција припада.

Приказ статистичких информација

Приликом завршетка анализе програма *Cachgrind* похрањује прикупљену табелу трошкова у датотеку која се назива *cachgrind.out.pid*; при чему *pid* представља јединствени идентификатор процеса који се извршио. Алат групише

све трошкове по фајловима и функцијама којима ти трошкови припадају. Глобална статистика се рачуна накнадно, приликом приказа резултата. На овај начин се штеди јако пуно времена приликом анализе кода. Функције које симулирају приступ брзој меморији се позивају јако често, тако да би додавање још неколико инструкција које сабирају, знатно успорило и овако споро извршавање алата.

3.4 Helgrind

Helgrind је алат у склопу програмског пакета *Valgrind* који открива грешке синхронизације приликом употребе модела нити *POSIX*.

Главне апстракције модела нити *POSIX* су: група нити која дели заједнички адресни простор, формирање нити, чекање за завршетак извршавања функције нити, излаз из функције нити, мутекс објекти, условне промељиве, читај-пиши закључавање и семафори. *Helgrind* може да открије следеће три класе грешака:

- Лоша употреба интерфејса за програмирање нити *POSIX*.
- Потенцијално блокирање нити које проистиче из лошег редоследа закључавања и откључавања променљивих.
- Приступ меморији без адекватног закључавања или синхронизације.

Проблеми као што су ови често узрокују нерепродуктивне, временски зависне падове програма и веома их је тешко открити. Алат *Helgrind* поседује механизам за веома прецизно праћење свих апстракција које користе модел нити *POSIX*. *Helgrind* даје најбоље резултате ако програм који се анализира користи само интерфејс за програмирање нити *POSIX*.

Лоша употреба интерфејса за програмирање нити *POSIX*

Helgrind пресреће позиве ка функцијама библиотеке *pthread*, и због тога је у могућности да открије велики број грешака. Овакве грешке могу да доведу до недефинисаног понашања програма и до појаве грешака у програмима које је касније веома тешко открити. Грешке које *Helgrind* проналази су: откључавање неважећег мутекса, откључавање мутекса који није закључан, откључавање мутекса кога је закључала друга нит, уништавање неважећег или закључаног мутекса, рекурзивно закључавање нерекурзивног мутекса, деалокација меморије

која садржи закључан мутекс, прослеђивање мутекса као аргумента функције која очекује као аргумент *reader-writer lock* и обрнуто, када *pthread* функција врати код грешке који је потрено додатно обрадити, када се нит уништи, а да још држи закључану промелјиву, прослеђивање функцији *pthread_cond_wait* незакључан мутекс, незважећи мутекс или мутекс кога је закључала друга нит, неконзистентне везе између условних промелјивих и њихових одговарајућих мутекса, неважећа или дупла иницијализација *pthread barrier*, уништавање *pthread barrier* који никада није иницијализован или кога нити чекају, чекање на *pthread barrier* објекта који није никада иницијализован, за све *pthread* функције које *Helgrind* пресреће, генерише се податак о грешци ако функција врати код грешке, иако *Helgrind* није нашао грешке у коду.

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

Thread #1 unlocked a not-locked lock at 0xFFEFFFFCF0
  at 0x4C301D6: mutex_unlock_WRK (hg_intercepts.c:1086)
  by 0x4C33B4C: pthread_mutex_unlock (hg_intercepts.c:1107)
  by 0x400867: nearly_main (tc09_bad_unlock.c:27)
  by 0x4008D3: main (tc09_bad_unlock.c:49)
Lock at 0xFFEFFFFCF0 was first observed
  at 0x4C33A93: pthread_mutex_init (hg_intercepts.c:779)
  by 0x400843: nearly_main (tc09_bad_unlock.c:23)
  by 0x4008D3: main (tc09_bad_unlock.c:49)
Address 0xffefffcf0 is on thread #1's stack
in frame #2, created by nearly_main (tc09_bad_unlock.c:16)
```

Слика 3.13: Пример приказа грешке у програму

Провере које се односе на мутексе се такође примењују и на *reader-writer lock*. Пријављена грешка приказује и примарно стање стека које показје где је детектована грешка. Такође, уколико је могуће исписује се и број линије у самом коду где се грешка налази. Уколико се грешка односи на мутекс, *Helgrind* ће приказати и где је први пут детектовао проблематични мутекс 3.17.

Потенцијално блокирање нити

Helgrind прати редослед којим нити закључава променљиве. На овај начин *Helgrind* детектује потенцијалне делове кода који могу довести до блокорања нити. На овај начин је могуће детектовати грешке које се нису јавиле током самог процеса тестирања програма, већ се јављају касније током коришћења истог.

Илустрација оваквог проблема је дата у наставку.

- Претпоставимо да је дељени објект О коме да би приступили морамо да закључамо две променљиве M1 и M2.
- Замислимо затим да две нити T1 и T2 желе да приступе дељеној променљивој О. До блокорања нити долази када нит T1 закључа M1, а у истом тренутку T2 закључа M2. Након тога нит T1 остане блокирана јер чека да се откључа M2, а нит T2 остане блокирана јер чека да се откључа T2.

Helgrind креира граф који представља све променљиве које се могу закључавати, а које је открио у прошлости. Када нит наиђе на нову променљиву коју закључава, граф се освежи и проверава се да ли граф садржи круг у коме се налазе закључане променљиве. Постојање круга у коме се налазе закључане променљиве је знак да је могуће да ће се нити некада у току извршавања блокирати. Ако постоје више од две закључане променљиве у кругу проблем је још озбиљнији.

Приступ меморији без адекватног закључавања или синхронизације

Приступ подацима без адекватног закључавања или синхронизације се односи на проблем када две или више нити приступају дељеном податку без синхронизације. На овај начин је могуће да две или више нити у истом тренутку приступе дељеном објекту.

Принцип приступа променљивој без адекватне синхронизације

На слици 3.14 приказан је пример програма променљивој без адекватне синхронизације.

```
#include <pthread.h>

int var = 0;

void* child_fn(void* arg) {
    var++;
    return NULL;
}

int main (void) {
    pthread_t child;
    pthread_create(&child, NULL, child_fn, NULL);
    var++;

    pthread_join(child, NULL);
    return 0;
}
```

Слика 3.14: Пример приступа променљивој без адекватне синхронизације

Проблем је у томе што ништа не спречава нити родитељи дете да у исто време приступе и промене вредности дељене променљивој *var*. Приликом анализе оваквог програма алатом *Helgrind* добија се извештај који је приказан на слици 3.15.

У извештају који је приказан на слици 3.15 можемо тачно да видиммо које нити приступају променљивој без синхронизације, где се врши сам приступ променљивој, име и величину саме променљиве којој нити приступају ради промене њене вредности.

Алгоритам детекције приступа променљивој без синхронизације

Алгоритам за детекцију приступа променљивој без синхронизације односи се на „десило се пре” приступ. У наставку је дат пример који објашњава „десило се пре” принцип 3.16.

Нит родитељ креира нит дете. Затим обе мењају вредност променљиве *var*, а затим нит родитеља чека да нит детета изврши своју функцију. Овај програм није добро написан јер не можемо са сигурношћу да знамо која је вредност променљиве *var* приликом штампања исте. Ако је нит родитеља бржа од нити

```
---Thread-Announcement-----
Thread #1 is the program's root thread
---Thread-Announcement-----
Thread #2 was created
  at 0x51620FE: clone (clone.S:74)
  by 0x4E43179: create_thread (createthread.c:102)
  by 0x4E44E20: pthread_create@@GLIBC_2.2.5 (pthread_create.c:677)
  by 0x4C32663: pthread_create_WRK (hg_intercepts.c:427)
  by 0x4C33747: pthread_create@* (hg_intercepts.c:460)
  by 0x400715: main (main.c:12)
-----
Possible data race during read of size 4 at 0x601054 by thread #1
Locks held: none
  at 0x400716: main (main.c:13)

This conflicts with a previous write of size 4 by thread #2
Locks held: none
  at 0x4006D7: child_fn (main.c:6)
  by 0x4C32857: mythread_wrapper (hg_intercepts.c:389)
  by 0x4E446A9: start_thread (pthread_create.c:333)
Address 0x601054 is 0 bytes inside data symbol "var"
-----
Possible data race during write of size 4 at 0x601054 by thread #1
Locks held: none
  at 0x40071F: main (main.c:13)

This conflicts with a previous write of size 4 by thread #2
Locks held: none
  at 0x4006D7: child_fn (main.c:6)
  by 0x4C32857: mythread_wrapper (hg_intercepts.c:389)
  by 0x4E446A9: start_thread (pthread_create.c:333)
Address 0x601054 is 0 bytes inside data symbol "var"
```

Слика 3.15: Извештај *Helgrind*-а за приступ промеливој без синхронизације

дете, онда ће бити штампана вредност 10, у супротном ће бити 20. Брзина извршавања нити родитељ и дете је нешто на шта програмер нема утицаја. Решење овог проблема је у закључавању промеливе *var*. На пример, можемо да

```
Parent thread:                                     Child thread:

int var;

// create child thread
pthread_create(...)
var = 20;

// wait for child
pthread_join(...)
printf("%d\n", var);

var = 10;
exit
```

Слика 3.16: „Десило се пре” принцип

пошаљемо поруку из нити родитељ након што она промени вредност променљиве *var*, а нит дете неће променити вредност променљиве *var* док не добије поруку. На овај начин смо сигурни да ће програм исписати вредност 10. Размена порука креира „десило се пре” зависност између две доделе вредност: *var = 20*; се догађа пре *var = 10*;. Такође, сада више немамо приступ променљивој без синхронизације. Није обавезно да шаљемо поруку из нити родитељ. Можемо послати поруку из нити дете након што она изврши своју доделу. На овај начин смо сигурни да ће се исписати вредност 20.

Алат *Helgrind* ради на истом овом принципу. Он прати сваки приступ меморијској локацији. Ако се локација, у овом примеру *var*, приступа из две нити, *Helgrind* проверава да ли су ти приступи повезани са „десило се пре” везом. Ако нису, алат пријављује грешку о приступу променљивој без синхронизације.

Ако је приступ дељеној променљивој из две или више програмерске нити повезан са „десило се пре” везом, значи да постоји синхронизациони ланац између програмских нити које обезбеђује да се сам приступ одвија по тачно одређеном редоследу, без обзира на стварне стопе напредка појединачних нити.

Стандардне примитиве нити креирају „десило се пре” везу:

- Ако је мутекс откучан од стране нити T1, а касније или одмах закључан

од стране нити T2, онда се приступ меморији у функцији T1 дешава пре него што нит T2 откључа мутекс да би приступила меморији

- Иста идеја се односи и на *reader-writer* закључавање променљивих
- Ако је кондициона променљива сигнализирана у функцији нити T1 и ако друга нит T2 чека на тај сигнал, да би наставила са радом, онда се меморијски приступ у T1 дешава пре сигнализације, док нит T2 врши приступ меморији након што изађе из стања чекања на сигнал који шаље нит T1.
- Ако нит T2 наставља са извршавањем након што нит T1 ослободи семафор, онда кажемо да постоји „десило се пре” релација између програмских нити T1 и T2.

Helgrind пресреће све горе наведене догађаје и креира граф који представља све „десило се пре” релације у програму. Такође, он прати све приступе меморији у програму. Ако постоји приступ некој меморијској локацији у програму од стране две нити и *Helgrind* не може да нађе путању кроз граф од једног приступа до другог, генерише податак о грешци у програму који анализира.

Helgrind не проверава да ли постоји приступ меморијској локацији без синхорнизације уколико се сви приступи тој локацији односе на читање садржаја те локације. Два приступа су у „десило се пре” релацији, иако постији призвољно дугачак ланац синхронизације догађаја између та два приступа. Ако нит T1 приступа локацији M, затим сигнализира нит T2, која касније сигнализира нит T3 која приступа локацији M, кажемо да су ова два приступа између нити T1 и T3 у „десило се пре” релацији. иако између њих не постоји директна веза.

Helgrind алгоритам за детекцију приступа меморији без синхорнизације прикупљене информације приказује у форми приказаној на слици 3.17.

На слици 3.17 можемо да приметимо да *Helgrind* најпре исписује податке где су нити које узрокују грешку направљене. Главни података о грешци почиње са „*Possible data race during read*”. Затим се исписује адреса где се насихорни приступ меморији дешава, као и величина меморије којој се приступа. У наставку *Helgrind* исписује где друга нит приступа истој локацији. На крају, *Helgrind* покренут са опцијом `--read-var-inof=yes` исписује и само име променљиве којој се приступа, као и где у програму је та променљива декларисана.

3.5 Callgrind

Callgrind је алат који генерише листу позива функција користичког програма у виду графа. У основним подашавањима сакупљени подаци састоје се од броја извршених инструкција, њихов однос са линијом у извршном коду, однос позиваоц/позван између функција, као и број таквих позива. Додатна подешавања омогућавају анализирање кода током извршавања.

Подаци који се анализирају се записују у фајл након завршетка рада програма и алата. Подржане команде су:

callgrind_annotate - на основу генерисаног фајла приказује листу функција.

Пример визуелизације листе функција приказан је на слици 3.18. За графичку визуелизацију препоручују се додатни алати (*KCachegrind*), који олакшава навигацију уколико *Callgrind* направи велику количину података.

callgrind_control - ова команда омогућава интерактивну контролу и надгледање програма приликом извршавања. Могу се добити информације о стању на стеку, може се такође у сваком тренутку генерисати профил.

Алат *Cachgrind* сакупља податке, односно броји догађаје који се дешавају директно у једној функцији. Овај механизам сакупљања података се назива ексклузивним.

Алат *Callgrind* проширује ову функционалност тако што пропагира цену функције до њених граница. На пример, ако функција *foo* позива функцију *bar*, цена функције *bar* се додаје функцији *foo*. Када се овај механизам примени на целу функцију, добија се слика такозваних инклузивних позива, где цена сваке функције укључује и цене свих функција које она позива, директно или индиректно.

Захваљујући графу позива, може да се одреди, почевши од *main* функције, која функција има највећу цену позива. Позиваоц/позван цена је изузетно корисна за профилисање функција која имају више позива из разних функција, и где имамо прилику за оптимизацију нашег програма мењајући код у функцији која је позиваоц, тачније редуковањем броја позива.

Могућност детектовања свих позива функција, као и завистно инструкција алата *Callgrind* зависи од платформе на којој се извршава. Овај алат најбоље ради на *x86* и *amd64*, али нажалост не даје најтачније резултате на следећим

платформама *PowerPc*, *ARM* и *MIPS*. Разлог томе је што код наведених платформи не постоји експлицитан позив или инструкција у скупу инструкција, па *Callgrind* мора да се ослања на хеуристике да би детектовао позиве или инструкције.

3.6 Massif

3.7 DRD


```

---Thread-Announcement-----

Thread #3 was created
  at 0x5157FBE: clone (clone.S:74)
  by 0x4E43199: do_clone.constprop.3 (createthread.c:75)
  by 0x4E448BA: create_thread (createthread.c:245)
  by 0x4E448BA: pthread_create@@GLIBC_2.2.5 (pthread_create.c:611)
  by 0x4C3167A: pthread_create_WRK (hg_intercepts.c:427)
  by 0x4C32758: pthread_create@* (hg_intercepts.c:460)
  by 0x400960: main (tc21_pthonce.c:87)

---Thread-Announcement-----

Thread #2 was created
  at 0x5157FBE: clone (clone.S:74)
  by 0x4E43199: do_clone.constprop.3 (createthread.c:75)
  by 0x4E448BA: create_thread (createthread.c:245)
  by 0x4E448BA: pthread_create@@GLIBC_2.2.5 (pthread_create.c:611)
  by 0x4C3167A: pthread_create_WRK (hg_intercepts.c:427)
  by 0x4C32758: pthread_create@* (hg_intercepts.c:460)
  by 0x400960: main (tc21_pthonce.c:87)

-----

Possible data race during read of size 4 at 0x601084 by thread #3
Locks held: none
  at 0x4008CF: child (tc21_pthonce.c:74)
  by 0x4C3186E: mythread_wrapper (hg_intercepts.c:389)
  by 0x4E44183: start_thread (pthread_create.c:312)
  by 0x5157FFC: clone (clone.S:111)

This conflicts with a previous write of size 4 by thread #2
Locks held: none
  at 0x4008D8: child (tc21_pthonce.c:74)
  by 0x4C3186E: mythread_wrapper (hg_intercepts.c:389)
  by 0x4E44183: start_thread (pthread_create.c:312)
  by 0x5157FFC: clone (clone.S:111)
Location 0x601084 is 0 bytes inside global var "unprotected2"
declared at tc21_pthonce.c:51

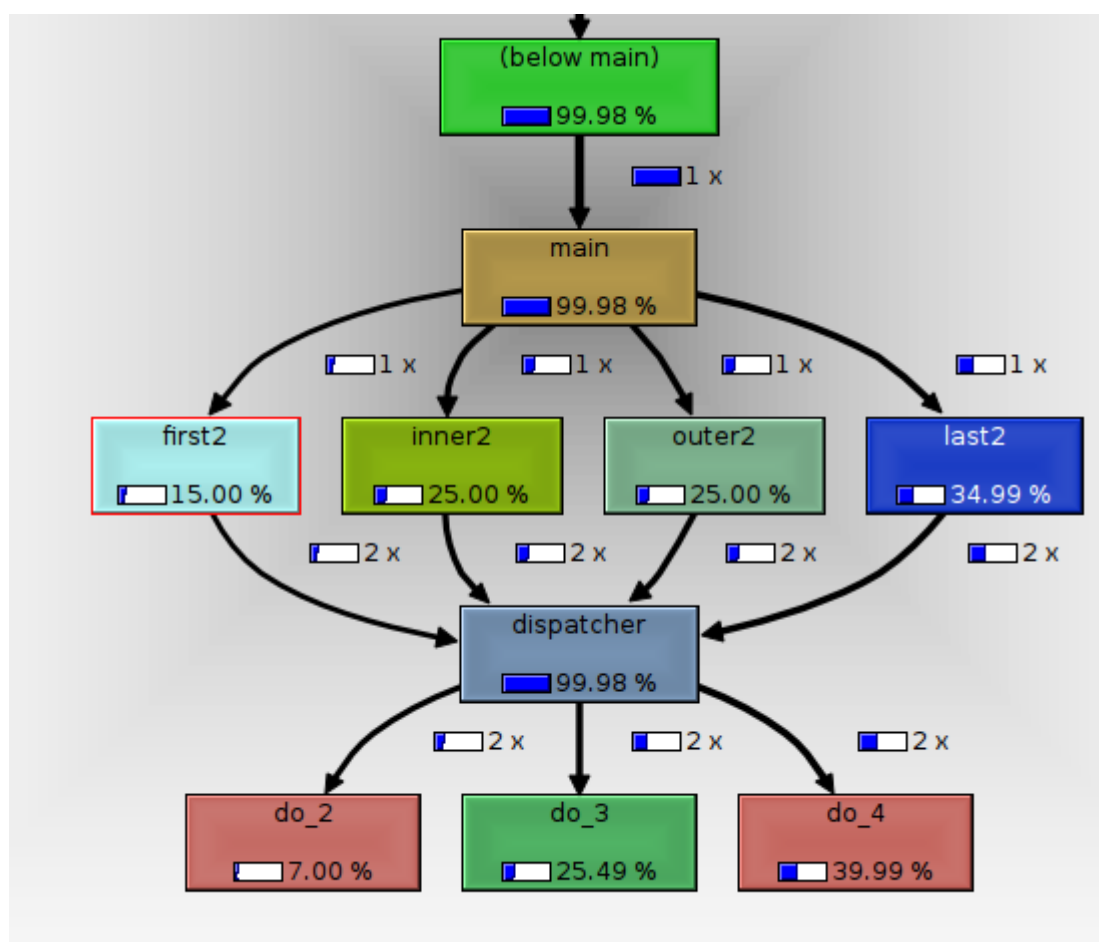
-----

Possible data race during write of size 4 at 0x601084 by thread #3
Locks held: none
  at 0x4008D8: child (tc21_pthonce.c:74)
  by 0x4C3186E: mythread_wrapper (hg_intercepts.c:389)
  by 0x4E44183: start_thread (pthread_create.c:312)
  by 0x5157FFC: clone (clone.S:111)

This conflicts with a previous write of size 4 by thread #2
Locks held: none
  at 0x4008D8: child (tc21_pthonce.c:74)
  by 0x4C3186E: mythread_wrapper (hg_intercepts.c:389)
  by 0x4E44183: start_thread (pthread_create.c:312)
  by 0x5157FFC: clone (clone.S:111)
Location 0x601084 is 0 bytes inside global var "unprotected2"

```

Слика 3.17: Пример излаза из *Helgrind*-а



Слика 3.18: Пример визуелизације функција

Глава 4

FPXX

Глава 5

Закључак

Библиографија

- [1] Yuri Gurevich and Saharon Shelah. Expected computation time for Hamiltonian path problem. *SIAM Journal on Computing*, 16:486–502, 1987.
- [2] Petar Petrović and Mika Mikić. Naučni rad. In Miloje Milojević, editor, *Konferencija iz matematike i računarstva*, 2015.
- [3] Dominic Sweetman. *See MIPS Run*. Wiley, 2007.

Биографија аутора

Вук Стефановић Караџић (*Трипић, 26. октобар/6. новембар 1787. — Беч, 7. фебруар 1864.*) био је српски филолог, реформатор српског језика, сакупљач народних умотворина и писац првог речника српског језика. Вук је најзначајнија личност српске књижевности прве половине XIX века. Стекао је и неколико почасних доктората. Учествовао је у Првом српском устанку као писар и чиновник у Неготинској крајини, а након слома устанка преселио се у Беч, 1813. године. Ту је упознао Јернеја Копитара, цензора словенских књига, на чији је подстицај кренуо у прикупљање српских народних песама, реформу ћирилице и борбу за увођење народног језика у српску књижевност. Вуковим реформама у српски језик је уведен фонетски правопис, а српски језик је потиснуо славеносрпски језик који је у то време био језик образованих људи. Тако се као најважније године Вукове реформе истичу 1818., 1836., 1839., 1847. и 1852.