

УНИВЕРЗИТЕТ У БЕОГРАДУ
МАТЕМАТИЧКИ ФАКУЛТЕТ

Александра Караџић

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

мастер рад

Београд, 2017.

Ментор:

др Милена ВУЈОШЕВИЋ 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: Програмска подршка је све присутнија, напредком технологије долази до пораста у перформансама и могућностима рачунара. Како кориснички захтеви све више расту, самим тим расте комплексност процеса развоја софтвера, јавља се потреба за алатима који служе за детекцију и отклањање грешака, као и за анализу и прављење профила корисничких програма. Један такав алат јесте *Valgrind*.

MIPS архитектура је архитектура са процесором смањеног скупа наредбни и једна од заступљенијих архитектура на тржишту система са уграђеним рачунаром. *FPXX* конвенција је специфична за *MIPS* и као таква овде је детаљно обрађена. Упознаћемо се са свим специфичностима ове конвенције. Циљ овог рада је упознавање са алатом *Valgrind* и свим могућностима које нам он нуди, као и упознавање са *FPXX* конвенцијом и приказ њене имплементације у сам алат.

Кључне речи: *Valgrind*, *MIPS*, *FP* регистри, *FPXX* конвенција

Садржај

1	Увод	1
2	Архитектура <i>MIPS</i>	3
2.1	<i>CISC</i> и <i>RISC</i>	3
2.2	<i>MIPS</i>	4
2.3	Регистри у <i>MIPS</i> -у	6
2.4	Регистри за рад са бројевима у покретном зарезу у <i>MIPS</i> -у	8
2.5	<i>FPXX</i> конвенција	9
3	Алат <i>Valgrind</i>	13
3.1	О <i>Valgrind</i> -у	14
3.2	<i>Memcheck</i>	17
3.3	<i>Cachgrind</i>	26
3.4	<i>Helgrind</i> и <i>DRD</i>	32
3.5	<i>Callgrind</i>	41
3.6	<i>Massif</i>	42
4	Имплементација <i>FPXX</i> конвенције	46
4.1	Превођење алата <i>Valgrind</i> са опцијом <i>-mfpxx</i>	47
4.2	Детектовање режима у којем ради алат <i>Valgrind</i>	48
4.3	Одређивање режима у којем програм почиње са радом	50
4.4	Пресретање системског позива <i>prctl()</i>	50
4.5	Тестирање	52
5	Закључак	55
	Библиографија	57

Глава 1

Увод

Тражење разлога неправилног рада система може трајати поприлично дуго, поготово ако се систем састоји из десетина хиљада линија кода и десетине па и стотине операција алоцирања и деалоцирања меморије. Под овим, подразумевамо грешке које компајлер не пријављује, попут цурења меморије или коришћење неиницијализованих вредности. За неке, још сложеније, вишеничне, системе, ове грешке се могу јавити у виду неочекиваног приступа дељеним подацима, односно утркивању за приступ истим. Један од алата који помажу у откривању оваквих грешака јесте *Valgrind* [7]. *Valgrind* представља веома користан алат, погодан за анализу свих нивоа меморије. Пуштање програма кроз *Valgrind* јесте значајно спорије, од двадесет до сто пута спорије, али са лакоћом може открити неправилности у раду са меморијом.

Са развојем и променама саме архитектуре *MIPS*, мора да се и мења део алата *Valgrind* специфичан за ову архитектуру. Једна од иновација је и *FPXX* конвенција [8]. Да би корисници овог алата на *MIPS* архитектурама могли исправно да га користе, морају да настану измене у самом алату. Циљ овог рада је омогућавање корисницима алата *Valgrind* да врше анализу програма или система који су преведени у складу са *FPXX* конвенцијом, као и превођење самог овог алата у складу са овом конвенцијом.

MIPS архитектура је веома распрострањена у системима са угређеним рачунаром [13]. У глави 2 ће бити описана архитектура *MIPS*. Биће описани регистри који се користе у овој архитектури и регистри за рад са бројевима у покретном зарезу. На крају ове главе биће описана *FPXX* конвенција. У овом делу ће бити представљене све карактеристике и захтеви који су морали бити испуњени при имплементацији у алату *Valgrind*.

У глави 3 ће бити описан сам алат *Valgrind*, начин рада и архитектуре за које је подржан. Биће детаљно описан начин рада свих алата уз примере грешака које се најчешће јављају приликом програмирања, као и начин њиховог детектовања.

У глави 4 биће ближе представљена архитектура алата *Valgrind*. Биће описана и сама имплементација *FPXX* конвенције. Биће описани кораци по којима се одвијао процес имплементације и биће представљене измене које су примењене.

У закључку ће бити дат мали осврт на целокупан рад. У овој глави ће бити поменути даљи планови рада на овом пројекту.

Глава 2

Архитектура *MIPS*

У овој глави описана је *MIPS* архитектура процесора. У поглављу 2.1 описане су архитектуре процесора *CISC* (скраћено од енгл. *Complex Instruction Set Computing*) и *RISC* (скраћено од енгл. *Reduced Instruction Set Computing*), док је у поглављу 2.2 описана архитектура *MIPS*. У поглављу 2.3 су описани регистри архитектуре *MIPS*, а у поглављу 2.4 су детаљно обрађени регистри за рад са бројевима у покретном зарезу. На крају, у поглављу 2.5 је описана конвенција *FPXX*, чија ће имплементација у алату *Valgrind* бити описана у глави 4.

2.1 *CISC* и *RISC*

Дизајнери хардвера раде на проналаску нових технологија и алата који би им олакшали посао имплементације архитектуре која може да испуни сва њихова очекивања. Архитектура хардвера може да буде имплементирана тако да буде или хардверски специфична или специфична за софтвер. Ако посматрамо хардвер процесора, постоје два концепта за имплементацију хардвера процесора, једна је *RISC* а друга је *CISC* [10].

CISC архитектуру процесора карактерише богат скуп инструкција. Овај приступ имплементације хардвера покушава да смањи број инструкција по програму, жртвовањем броја циклуса по инструкцији. Рачунари засновани на *CISC* архитектури су дизајнирани да смањују трошкове меморије. Великим програмима је потребно много меморије, чиме се повећава трошак меморије и велика меморија постаје скупља. Да би се решио овај проблем, број инструкција по програму може бити смањен уградњом више операција у једну инструкцију,



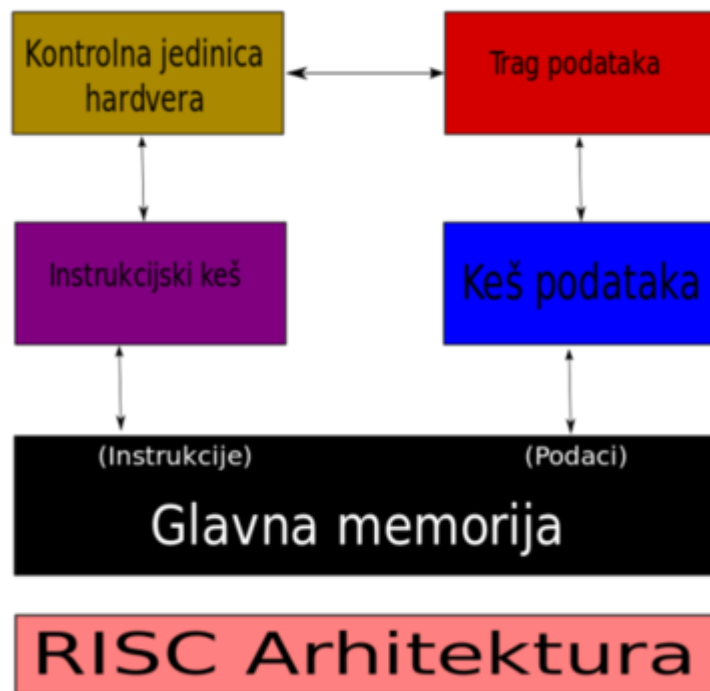
Слика 2.1: *CISC* архитектура

правећи тако комплексније инструкције [10].

RISC архитектура користи високо оптимизован скуп инструкција. Код *RISC* архитектуре мотив је обрнут у односу на *CISC* архитектуру, смањује се број циклуса по инструкцији по цени броја инструкција по програму. Проточна обрада (енг. *pipeline*) је једна од јединствених одлика архитектуре *RISC*, која је постигнута преклапањем извршавања неколико инструкција. Због проточне обраде *RISC* архитектура има велику предност у перформансама у односу на *CISC* архитектуре [10].

2.2 *MIPS*

MIPS је *RISC* архитектура процесора, рођена у плодном периоду академских истраживања и развоја, осамдесетих година прошлог века. *MIPS* пројекат је један од пионирских пројеката на Станфорд у, на ком је радио Џон Хенеси са својим студентима.



Слика 2.2: *RISC* архитектура

Релативна једноставност је била комерцијална нужност за *MIPS* процесоре, која се 1985. године развила из академског пројекта за израду и пласира на тржиште чипова. Као резултат, ова архитектура је имала (можда још увек има) највећи ранг произвођача у индустрији, прозводећи од *ASIC* језгара (*MIPS Technologies*) до веома јефтиних процесора (*IDT*, *AMD/Alchemy*), укључујући и само 64-битне процесоре (*PMC-Sierra*, *Toshiba*, *Broadcom*).

Процесори засновани на *MIPS* скупу инструкција се често користе код наменских уређаја и ручних рачунара (енг. *handheld PC*). Мобилни уређаји, сет-топ боксови, паметни телевизори, за које се често користе *MIPS* процесори, повлаче са собом велики број апликација са интензивним израчунавањем као што су: процесирање слике и видеа, интеракција између човека и компјутера, анализа података...

2.3 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 која му се вредност додели

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

Сви ови регистри се могу 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 - Користе се за враћање резултата при повратку из неке функције. Резултат може бити целобројног типа или број записан у фиксном зарезу.

a0 - a3 - Користе се за прослеђивање прва четири аргумената функције која се позива.

t0 - t9 - Користе се као привремени регистри.

s0 - s7 - Садржај ових регистара мора остати непромењен након извршавања сваке функције, што се постиже привременим чувањем ових регистара на стеку уколико се њихова вредност мења у току функције и враћањем након завршетка функције. Ово су регистри које чува позвана функција (енг. *callee saved registers*).

k0, k1 - Резервисано за систем прекида, који након коришћења не враћа садржај ових регистара на почетни. Систем прекида прво сачува садржај 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*), регистар *\$gp* показује на табелу показивача, познате као *GOT* (скраћено од енг. *Global Offset Table*), преко које приступа деловима кода и подацима. *PIC* је код који се може извршавати на било којој меморијској адреси, без модификација. *PIC* се најчешће користи за дељене 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 глобални подаци који не заузимају много меморије. Оно што је битно је да овај регистар не користе 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 се приступити у функцији. Тако је омогућено да компајлер може да приступи поменљивама на стеку помоћу константног помераја у односу на *\$sp*.

fp - Познат и као *\$sp*, показивач на стек оквир. Користи се од стране функције, за праћење стања на стеку, за случај да из неког разлога компајлер 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 повратка и то је подржано кроз одговарајуће инструкције скока. Ово се разликује од конве-

ције која се користи на архитектури x86, где инструкција позива функције адресу 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*.

2.4 Регистри за рад са бројевима у покретном зарезу 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. C компајлери користе двоструку прецизност за C *double* типове података.

Начин на који се 64-битна реч смешта у меморију, односно две речи од којих се он састоји, зависи од начина на који процесор смешта податке у меморију (нижи бит на нижој адреси или виши бит на нижој адреси).

Стандард IEEE 754 је веома захтеван и поставио је два велика проблема. Први проблем омогућавање детекције неуобичајних резултата, доводи проточну обраду тешком. Постоји опција да се имплементира IEEE механизам сигнализирања 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 синхронно; након заустављања стање *FP* регистра је исто као и пре почетка извршавања инструкције која је довела до изузетка.

У MIPS архитектури, хардверска заустављања су имплементирана на начин који је описан. Ово заправо ограничава могућности проточне обраде *FP* операција, јер се не може извршити следећа инструкције све док хардвер није

сигуран да следећа *FP* операција неће произвести заустављање. Зарад избегавања додавања времена за извршавање, *FP* операције морају да одлуче да ли ће доћи до заустављања у првој фази.

MIPS процесори имају 32 *FP*, који су обично обележени $\$f0$ - $\$f31$. Са изузетком неких јако старих процесора као што је *MIPS I*, сваки 64-битни регистар може да садржи вредност двоструке прецизности.

Први *MIPS* процесор је имао 16 регистара. Заправо, постајало је 32 32-битних регистара, али од сваког пара парно/непарних регистара направљена је јединица за математичке операције (укључујући и математичке операције за двоструку прецизност). Непарни регистри се користе за операције учитавања, чувања и премештања између *FP* регистара и регистара за целобројне вредности.

MIPS I је избачен из употребе, али касније верзије процесора имају такозвани „бит компатибилности” у регистру *SR(FR)*. Уколико се у овај регистар постави вредност нула добијају се операције за процесор *MIPS I*. Још увек је у употреби велики број софтвера који раде на овај начин. Да би програм могао да користи *FP* регистре мора да постоји подршка компајлера, као и да цео систем на ком желимо да покренемо програм буде конзистентности са коришћењем *FP* регистара.

MIPS FP регистри се такође користе за чување и манипулацију означених целобројних вредности (32 и 64 битних). Тачније, када у програму постоји конверзија из целобројне вредности у бројеве са покретним зарезом, све те операције конверзије користе само *FP* регистре - целобројна вредност у *FP* регистру је конвертована у вредност у покретном зарезу у *FP* регистру [13].

2.5 *FPXX* конвенција

MIPS O32 ABI је 32-битни скуп инструкција и правила за *MIPS* архитектуру процесора. *FPXX* конвенција је додатак *MIPS O32 ABI* скупу правила и дефинише услове које треба да задовољи машински програм како би се коректно извршавао независно од режима у коме ради јединица за операције са бројевима у покретном зарезу. Програмски кôд који поштује ову конвенцију практично користи подскуп инструкција које су заједничке за оба режима и као такав није оптималан, али је погодан за комбиновање са кодом преведеним за било који режим рада у покретном зарезу. Идеја је да дељене библиотеке и

кориснички програми који треба да буду портабилни, буду преведени у складу са *FPXX* конвенцијом.

MIPS ABI је мењан током времена како се мењала архитектура. Промене које су настале у архитектури захтевале су да се преиспита стање *O32 ABI* и процени да ли постоји могућност да се направи *ABI* који би био компатибилан са тадашњим и свим будућим унапређивањима архитектуре. Три главна разлога за проширивање *O32 ABI*-ја су била увођење *MSA ASE* (*MSA* је проширење *MIPS* архитектуре модулom *SIMD*, нове инструкције омогућавају ефикасно паралелно извршавање векторских операција [12]), жеља да се искористи *FR=1* режим *FPU*-а и *MIPS32r6* архитектура која подржава само *FR=1* режим [8].

FR=0 режим (FP32) описује *FPU* са 32 32-битне регистре. Ови регистри су нумерисани од \$f0 до \$f31. Парови парних и непарних регистара се користе за добијање 64-битних контејнера и нумерисани су \$f0 до \$f30. Операције двоструке прецизности не смеју да користе непарне регистре.

FR=1 режим (FP64) описује *FPU* који има 32 64-битна регистра. Ови регистри су нумерисани од \$f0 до \$f31 и сваки од њих се може користити и за вредности једноструке и двоструке прецизности. Када се користе за операције са једноструком прецизношћу виших 32-бита постаје недефинисано.

FRE=1 режим је уведен са процесором *MIPS32r5*. Овај режим се користи у конјункцији са *FR=1* режимом, *FPU* има 32 64-битна регистра али је понашање одређених инструкција као у *FR=0* режиму. Операције са 64-битним или ширим форматима се извршавају на исти начин као да се извршавају у *FR=1* режиму, али 32-битни формати имају понашање као у *FR=0* режиму. Посебна карактеристика *FRE* режима се врти око руковања регистара са прецизношћу од 32-бита и понашање посебно са непарним регистрима. Да би се *FP32* режим извршавао коректно када се користе непарни регистри мора да се ажурира виши 32-битни део парног 64-битног регистра, такође ажурирање парних 64-битних регистара има за последицу ажурирање непарних 32-битних регистара. *FRE* режим ово достиже тако што преусмерава читање и писање са непарних регистара на горњих 32-бита парних регистара.

Табела 2.1: Опције за превођење програма

Опције	Значење
-mfr32	Генерише се кôд који претпоставља да ће радити само на FP=0 FPU процесору
-mfrxx	Генерише се кôд који може да се улинкује са кôд који је преведен са опцијом -mfr32 или -mfr64 и може да се покрене на FP=0 или FP=1 FPU процесору
-mfr64	Генерише се кôд који претпоставља да ће радити само на FP=1 FPU процесору

Прве верзије *MIPS*-а су подржавале само 32-битни копроцесор, док код осталих верзија ко-процесор може бити и 32-битни и 64-битни. Приликом писања програма програмер, односно компајлери који се користе морају да буду свесни у ком режиму ће се извршавати програм и у складу са тим да бирају инструкције које ће се користити. Додате су нове опције приликом превођења програма за одабир једног од три режима, приказаних у табели 2.1.

Паралелно са увођењем *FPXX* конвенције уведена је и *.MIPS.abiflags* секција објектног фајла. Ова секција садржи структуру података која представља суштинске информације које омогућавају одређивање, између осталог и, режима у ком програм ради. У табели 2.2 у колони Опције су представљене опције са којима се преводи програм, док у колони *FP_ABI* су вредности које се том приликом уписују у одговарајуће поље *.MIPS.abiflags* секције објектног фајла. Приликом одлучивања у ком режиму ће радити програм, језгро чита вредност *FP_ABI* из самог програма као и из интерпретера уколико се ради о динамички преведеном програму. На основу те две вредности се у језгру одлучује у ком режиму ће програм радити. На пример, ако је програм преведен са опцијама -mabi=32 -mfr32 (*FP_ABI*=1), а интерпретер са опцијама -mabi=32 -mfrxx (*FP_ABI* = 5) процес ће започети рад у режиму *FR* = 0. Ако је програм преведен са опцијама -mabi=32 -mfrxx -modd-spreg (*FP_ABI* = 6), а интерпретер са опцијама -mabi=32 -mfrxx (*FP_ABI* = 5) процес ће започети рад у режиму *FR* = 1.

Инструкција *mtc1* копира вредност из *GP* регистра у *FP* регистар, тачније нижих 32-бита из *GP* регистра преписује у нижих 32-бита *FP* регистра. Инструкција *mfc1* копира вредност из *FP* у *GP* регистар [11]. Инструкције *mtc1* и *mfc1* се не смеју користити за приступање вишег дела регистра двоструке прецизности. Уколико је потребно ове инструкције се могу користити за приступ

Табела 2.2: Опције за превођење програма и вредности променљиве *FP_ABI*

Опције	<i>FP_ABI</i>
-mabi=32 -mfp32	1
-mabi=64 -mfp64	1
-msingle-float	2
-msoft-float	3
-mabi=32 -mfpxx	5
-mabi=32 -mfp64 -modd-spreg	6
-mabi=32 -mfp64 -mno-odd-spreg	7

нижем делу регистра двоструке прецизности. Сваки пренос 64-битних података између *GP* и *FP* регистара двоструке прецизности мора се радити кроз меморију. Архитектуре које подржавају инструкције *MTHC1/MFHC1* могу оптималније приступити вишем делу регистра двоструке прецизности [8]. Инструкција *MTHC1* копира садржај из *GP* регистра у горњих 32-бита *FP* регистра, док инструкција *MFHC1* копира горњих 32-бита из *FP* регистра у *GP* регистар [11].

Системски позив *prctl()* се позива са првим аргументом који говори шта треба да се ради, док остали зависе умногоме од првог аргумента. Од верзије језгра 4.1 системском позиву *prctl()* додате су две нове опције којима може да се одредити и променити тренутни режим *FPU* регистара. Опцијом *PR_GET_FP_MODE prctl()* системски позив нам враћа режим, док са опцијом *PR_SET_FP_MODE prctl()* можемо да мењамо тренутни режим рада. *Prctl()* системски позив може да контролише тренутни регистарски режим - режим се може погледати и нови режим може бити постављен. *Prctl()* системски позив може да промени режим свих нити које су у том тренутку активне [9].

Глава 3

Алат *Valgrind*

Valgrind је платформа за прављење алата за динамичку бинарну анализу кода. Динамичка анализа обухвата анализу корисничког програма у извршавању, док бинарна 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 (*Memcheck*), детектор грешака нити (*Helgrind*), оптимизатор брзе меморије и скокова (*Cachgrind*), генератор графа скривене меморије и предикције скока (*Callgrind*) и оптимизатор коришћења динамичке меморије (*Massif*). *Valgrind* ради на следећим архитектурама:

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

Solaris - *x86*, *AMD64*

Android - *ARM*, *ARM64*, *x86* (4.0 и новије), *MIPS32*

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

У наредним поглављима биће детаљно описана структура *Valgrind*-а и његових алата, као и начин употребе са примерима проблема са којима се програмери свакодневно сусрећу. У поглављу 3.2 биће описан алат *Memcheck*, у поглављу 3.3 ће бити више речи о алату *Cachgrind*, у поглављу 3.4 су описани алати *Helgrind* и *DRD*, у поглављу 3.5 описан је алат *Callgrind*, у поглављу 3.6 биће речи о алату *Massif*.

3.1 О Valgrind-u

Алат за динамичку анализу кода се креира као додатак, писан у *C* програмском језику, на језгро *Valgrind*-а.

Језгро Valgrind-a + алат који се додаје = Алат Valgrind-a

Језгро *Valgrind*-а омогућава извршавање клијентског 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 на основу `--tool` опције бира алат који треба покренути. 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). *Valgrind*-ово језгро прво иницијализује подсистем као што су менаџер адресног простора, и његов унутрашњи алокатор меморије и затим учитава клијентову извршну датотеку. Потом се иницијализују *Valgrind*-ови подсистеми као што су трансляциона 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 превођењем и извршавањем клијентског програма. Може се рећи да *Valgrind* врши JIT (*Just In Time*) превођење машинског кода програма у машински код програма допуњен инструментацијом. Ниједан део кода клијента се не извршава у свом изворном облику. Алат умеће код у оригинални код на почетку, затим се нови код преводи, сваки основни блок појединачно, који се касније извршава. Процес превођења се састоји из рашчлањивања оригиналног машинског кода у одговарајућу међурепрезентацију (енгл. *intermediate representation*, скраћено IR) који се касније инструментализује са алатом и поново преводи у нови машински код.

Резултат свега овога се назива трансляција, која се чува у меморији и која се извршава по потреби. Језгро троши највише времена на сам процес прављења,

проналажења и извршавања транслације. Оригинални код се никада не извршава. Једини проблем који се овде може догодити је ако се врши транслација кода који се мења током извршавања програма.

Постоје многе компликације које настају приликом смештања два програма у један процес (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 клијентског 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 у функцију позиваоца. 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 клијентског кода $BB1$, $BB2$, ... онда преведене основне блокове обележавамо са $t(BB1)$, $t(BB2)$, ... Величина основног блока је ограничена на максимално шездесет 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* учитава и инструкцију која се налази непосредно иза ње, односно шездесет и прву инструкцију. Тиме се омогућава конзистентно извршавање програма који се анализира, као и у случају да се програм извршава без посредства *Valgrind*-а. Уколико након извршених шездесет 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

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, виртуалној 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. Програм који се извршава може проследити параметре оперативном систему на више начина, прослеђивање параметара у регистрима процесора, постављањем параметара у меморијској табели. Адреса табеле се 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 редувантног 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, којима проверава исправност рада програма.

Оптимизација 2 - Друга фаза оптимизације је једноставнија 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. Не зависи од платформе, 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 [13].

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.

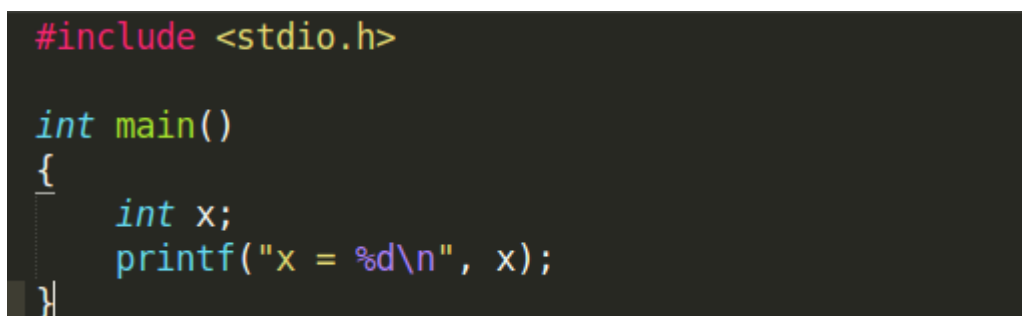
За програме писане у језицима C и C++ детектује уобичајне проблеме као што је приступање недопуштеној меморији, на пример преписивање блокова на хипу, преписивање врха стека и приступање меморији која је већ ослобођена. Алат детектује и коришћење неиницијализованих вредности, вредности које нису иницијализоване или које су изведене од других неиницијализованих вредности. Неисправно ослобађање хип меморије, као што је дупло ослобађање хип блокова или неупареног коришћења функција `malloc/new/new[]` и `free/delete/delete[]` је још један од проблема који алат *Memcheck* детектује. Као и преклапање параметара прослеђених функцијама (нпр. преклапање *src* и *dst* показивача кода функције *memcpy*) и 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 двадесет до сто пута спорији него када се извршава самостално, због транслације кода. Излазни програм је повећан за излаз који додаје сам алат *Memcheck*, који се исписује на стандардном излазу за грешке [6].

Коришћење неиницијализованих вредности



```
#include <stdio.h>

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

Слика 3.1: Пример програма који користи неиницијализовану променљиву

На слици 3.1 је дат пример програма у коме користимо неиницијализовану променљиву. Грешка у коришћењу неиницијализоване вредности се генерише када програм користи променљиве чије вредности нису иницијализоване.

```
==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

Слика 3.2 приказује излаз *Valgrind*-а који детектује коришћење неиницијализованих вредности у програму. Први део, односно прве три линије се штампају приликом покретања било ког алата који је у склопу *Valgrind*-а, у овом случају

алата *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 неиницијализованих вредности. У програму неиницијализована променљива може више пута да се копира, *Memcheck* прати све то, бележи податке о томе, али не пријављује грешку. У случају да се неиницијализоване вредности користе на начин да од те вредности зависи даљи ток програма или ако је потребно приказити вредности неиницијализоване променљиве, *Memcheck* пријављује грешку. Да би могли да видимо главни извор коришћења неиницијализованих вредности у програму, додаје се опција `--trace-origins=yes` [6].

Коришћење неиницијализоване или неадресиране вредности у системском позиву

Memcheck прати све параметре системског позива. Проверава сваки параметар појединачно, без обзира да ли је иницијализован или не. Проверава да ли системски позив треба да чита из меморије која је дефинисана 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 системском 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 позива системског позива са неисправним параметрима

```
rtk@rtk579-lin:/export/valgrind$ ./vg-in-place --leak-check=yes --show-reachable=yes --track-origins=yes ../main
==8038== Memcheck, a memory error detector
==8038== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==8038== Using Valgrind-3.14.0.GIT and LibVEX; rerun with -h for copyright info
==8038== Command: ../main
==8038==
==8038== Syscall param write(buf) points to uninitialised byte(s)
==8038==   at 0x4F26390: __write_nocancel (syscall-template.S:81)
==8038==   by 0x4005F6: main (in /export/main)
==8038==   Address 0x5200040 is 0 bytes inside a block of size 10 alloc'd
==8038==   at 0x4C2AC23: malloc (vg_replace_malloc.c:299)
==8038==   by 0x4005CE: main (in /export/main)
==8038==   Uninitialised value was created by a heap allocation
==8038==   at 0x4C2AC23: malloc (vg_replace_malloc.c:299)
==8038==   by 0x4005CE: main (in /export/main)
==8038==
==8038== Syscall param exit_group(status) contains uninitialised byte(s)
==8038==   at 0x4EFC109: _Exit (_exit.c:32)
==8038==   by 0x4E7316A: __run_exit_handlers (exit.c:97)
==8038==   by 0x4E731F4: exit (exit.c:104)
==8038==   by 0x400603: main (in /export/main)
==8038==   Uninitialised value was created by a heap allocation
==8038==   at 0x4C2AC23: malloc (vg_replace_malloc.c:299)
==8038==   by 0x4005DC: main (in /export/main)
==8038==
==8038== HEAP SUMMARY:
==8038==   in use at exit: 14 bytes in 2 blocks
==8038==   total heap usage: 2 allocs, 0 frees, 14 bytes allocated
==8038==
==8038== 4 bytes in 1 blocks are still reachable in loss record 1 of 2
==8038==   at 0x4C2AC23: malloc (vg_replace_malloc.c:299)
==8038==   by 0x4005DC: main (in /export/main)
==8038==
==8038== 10 bytes in 1 blocks are still reachable in loss record 2 of 2
==8038==   at 0x4C2AC23: malloc (vg_replace_malloc.c:299)
==8038==   by 0x4005CE: main (in /export/main)
==8038==
==8038== LEAK SUMMARY:
==8038==   definitely lost: 0 bytes in 0 blocks
==8038==   indirectly lost: 0 bytes in 0 blocks
==8038==   possibly lost: 0 bytes in 0 blocks
==8038==   still reachable: 14 bytes in 2 blocks
==8038==   suppressed: 0 bytes in 0 blocks
==8038==
==8038== For counts of detected and suppressed errors, rerun with: -v
==8038== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 0 from 0)
```

Слика 3.4: Пример излаза за програм који у себи садржи позив системског позива са неисправним параметрима

На слици 3.3 дат је пример позива системског позива са неисправним параметрима. На слици 3.4 је извештај који добијамо након анализе програма са слике 3.3. Можемо да видимо да је *Memcheck* приказао информације о коришћењу неиницијализованих вредности у системским позивима. Прва грешка приказује да параметар системског позива *write()* показује на неиницијализовану вредност. Друга грешка приказује да је податак који се прослеђује системском позиву *exit()* недефинисан. Такође, приказане су и линије у самом програму где се ове вредности користе [6].

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

```
#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 нелаганог ослобађања меморије

```
rtk@rtk579-lin:/export/valgrind$ ./vg-in-place --leak-check=yes --show-reachable=yes --track-origins=yes ../main
==8524== Memcheck, a memory error detector
==8524== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==8524== Using Valgrind-3.14.0.GIT and LibVEX; rerun with -h for copyright info
==8524== Command: ../main
==8524==
==8524== Invalid free() / delete / delete[] / realloc()
==8524==    at 0x4C2BD1D: free (vg_replace_malloc.c:530)
==8524==    by 0x4005AB: main (in /export/main)
==8524== Address 0xfffffffffffffa0 is not stack'd, malloc'd or (recently) free'd
==8524==
==8524== Invalid free() / delete / delete[] / realloc()
==8524==    at 0x4C2BD1D: free (vg_replace_malloc.c:530)
==8524==    by 0x4005B8: main (in /export/main)
==8524== Address 0xfffffffffffffa0 is not stack'd, malloc'd or (recently) free'd
==8524==
==8524== HEAP SUMMARY:
==8524==    in use at exit: 47 bytes in 3 blocks
==8524==    total heap usage: 3 allocs, 2 frees, 47 bytes allocated
==8524==
==8524== 12 bytes in 1 blocks are definitely lost in loss record 1 of 3
==8524==    at 0x4C2AC23: malloc (vg_replace_malloc.c:299)
==8524==    by 0x40059B: main (in /export/main)
==8524==
==8524== 16 bytes in 1 blocks are definitely lost in loss record 2 of 3
==8524==    at 0x4C2AC23: malloc (vg_replace_malloc.c:299)
==8524==    by 0x4005C2: main (in /export/main)
==8524==
==8524== 19 bytes in 1 blocks are definitely lost in loss record 3 of 3
==8524==    at 0x4C2AC23: malloc (vg_replace_malloc.c:299)
==8524==    by 0x40058E: main (in /export/main)
==8524==
==8524== LEAK SUMMARY:
==8524==    definitely lost: 47 bytes in 3 blocks
==8524==    indirectly lost: 0 bytes in 0 blocks
==8524==    possibly lost: 0 bytes in 0 blocks
==8524==    still reachable: 0 bytes in 0 blocks
==8524==    suppressed: 0 bytes in 0 blocks
==8524==
==8524== For counts of detected and suppressed errors, rerun with: -v
==8524== ERROR SUMMARY: 5 errors from 5 contexts (suppressed: 0 from 0)
```

Слика 3.6: Пример излаза за програм у ком се нелагално ослобађа меморија

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.

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 [6].

Детекција цурења меморије

Memcheck бележи податке о свим динамичким блоковима који су алоцирани током извршавања програма позивом функција *malloc()*, *new()* и др. Када 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. Први начин је да је постојао показивач који је показивао на почетак блока, али је намерно (или ненамерно) померен да показује на унутрашњост блока. Други начин, ако постоји нежељена вредност у меморији, која је у потпуности неповезана и случајна. И трећи начин, ако постоји показивач на низ C++ објеката (који поседују деструкторе) који су алоцирани са *new[]*. У трећем случају, неки компајлери чувају „магични показивач” који садржи дужину низа од почетка блока.

На слици 3.7 је приказано девет могућих случајева када показивачи показују на неке меморијске блокове. *Memcheck* обједињује неке од ових случајева, тако да добијемо наредне четири категорије.

Још увек доступан (енг. *Still reachable*) - 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.

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

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

Слика 3.7: Пример показивача на меморијски блок

Дефинитивно изгубљен (енг. *Definitely lost*) - Ово се односи на случај 3 на слици 3.7. Ово значи да је немогуће пронаћи показивач који показује на меморијски блок. Блок је проглашен изгубљеним, заузета меморија не може да се ослободи пре завршетка програма, јер не постоји показивач на њу.

Индиректно изгубљен (енг. *Indirectly lost*) - 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 поправи 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 (енг. *Possibly lost*) - 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, она неће бити приказана.

```
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

На слици 3.8 је дат резиме цурења меморије који исписује *Memcheck*. Ако је укључена опција `--leak-check=yes`, *Memcheck* ће приказати детаљан извештај 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 и 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 [6].

3.3 Cachgrind

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

Постоје машине које имају и три или четири нивоа брзе меморије. У том случају, *Cachgrind* симулира приступ првом и последњем нивоу. Генерално гледано, *Cachgrind* симулира *I1*, *D1* и *LL* (последњи ниво брзе меморије).

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

- Подаци о читањима инструкција из брзе меморије укључују следеће статистике

Ir - укупан број извршених инструкција

I1mr - број промашаја читања инструкција из брзе меморије нивоа *I1*

ILmr - број промашаја читања инструкција из брзе меморије нивоа *LL*

- Подаци о читањима брзе меморије укључују следеће статистике

Dr - укупан број читања меморије

D1mr - број промашаја читања нивао брзе меморије *D1*

DLmr - број промашаја читања нивао брзе меморије *LL*

- Подаци о писањима у брзу меморију укључују следеће статистике

Dw - укупан број писања у меморији

D1mw - број промашаја писања у ниво брзе меморије *D1*

DLmw - број промашаја писања у ниво брзе меморије *LL*

- 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* броју приступа. Ова статистика се прикупља на нивоу целог програма, као и 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 [1].

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 више-струким 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== Cachegrind, a cache and branch-prediction profiler
==28165== Copyright (C) 2002-2015, and GNU GPL'd, by Nicholas Nethercote et al.
==28165== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==28165== Command: trunk/cachegrind/tests/clreq
==28165==
==28165== I  refs:      102,190
==28165== I1 misses:     728
==28165== L1i misses:    721
==28165== I1 miss rate:  0.71%
==28165== L1i 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== L1d misses:   1,498 ( 1,040 rd +   458 wr)
==28165== D1 miss rate:  4.7% (  5.1% +  3.8% )
==28165== L1d 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*

Извршавање програма кроз *Cachgrind* траје веома споро. Након завршетка рада, добијају се статистике као што је приказано на слици 3.10 [1].

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 глобално 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 информације о приступу брзој меморији, као што су адресе и величина меморије којој се приступа.

Симулација приступа брзој меморији има следеће карактеристике када се деси промашај уписа у брзу меморију, блок који је потребно уписати се смешта у *D1* део брзе меморије. Инструкције које модификују вредности 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 изазове 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 једнако 2 на M бајта, (величина брзе меморије / величина линије) једнако је 2 на 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 само један погодак, уколико један блок нађемо у брзој меморији, а други не, рачунамо један промашај (и нула погодатака) и 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 начин је употребом *cpuid* наредбе. Други начин представља ручно уношење параметара симулиране брзе меморије, приликом 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

Централна табела трошкова. Друга структура која чини метаподатке алата *Cachgrind* је централна табела трошкова. Свака линија у коду која се

```
typedef struct _instr_info instr_info;

struct _instr_info {
    Addr instr_addr;    // адреса инструкције
    UChar instr_size;   // величина инструкције у байтовима
    UChar data_size;    // величина податка коме инструкција припада
    lineCC* parent;     // lineCC за ову инструкцију
}
```

Слика 3.12: Структура табеле информација о инструкцијама

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 брзе меморије. Структура *lineCC* садржи три *CC* елемента: за читање *I* дела, за читање дела *D* и за писање у део *D* брзе 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 нова: 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 самњује број аргумената који се просеђују функцијама, које врше симулацију приступа брзој 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 [1].

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 информације 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 блок 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 врсте *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 функцији програмског језика *C* се, одређује на основу категорије којој та функција припада [1].

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 [1].

3.4 Helgrind и DRD

Helgrind је алат у склопу програмског пакета *Valgrind* који открива грешке синхронизације приликом употребе модела нити *POSIX*, док је *DRD* алат за детекцију грешака у *C* и *C++* програмима који користе више нити. *DRD* ради за сваки програм који користи нити *POSIX* стандарда или који користе концепте који су надограђени на овај стандард.

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.

Лоша употреба интерфејса за програмирање нити *POSIX*

Многе имплементације интерфејса су оптимизоване ради бржег времена извршавања. Такве имплементације се неће бунити на одређене грешке (ако мутекс откључа нека друга нит, а не она која га је закључала).

Грешке које *Helgrind* и *DRD* проналазе су:

- Грешке у откључавању мутекса - када је мутекс неважећи, није закључан или је закључан од стране друге нити.
- Грешке у раду са закључаним мутексом - 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.
- Грешке са *pthread barrier* - неважећа или дупла иницијализација, уништавање *pthread barrier* који никада није иницијализован или кога нити чекају или чекање на објекат који није никада иницијализован.
- Грешке приликом коришћења функције *pthread_cond_wait* - прослеђивање незакључаног, неважећег или мутекса кога је закључала друга нит.
- *Pthread* функција врати код грешке који је потрено додатно обрадити, када се нит уништи, а да још држи закључану промелјиву.
- Неконзистентне везе између условних промелјивих и њихових одговарајућих мутекса.

Овакве грешке могу да доведу до недефинисаног понашања програма и до појаве грешака у програмима које је касније веома тешко открити. *Helgrind* пресреће позиве ка функцијама библиотеке *pthread*, и због тога је у могућности да открије велики број грешака. За све *pthread* функције које *Helgrind* пресреће, генерише се податак о грешци ако функција врати код грешке, иако *Helgrind* није нашао грешке у коду.

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 3.13 [4].

Потенцијално блокирање нити

Helgrind прати редослед којим нити закључава промелјиве. На овај начин *Helgrind* детектује потенцијалне делове кода који могу довести до блоковања

```
---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

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 М1 и М2.
- Замислимо затим да две нити Т1 и Т2 желе да приступе дељеној променљивој О. До блокарања нити долази када нит Т1 закључа М1, а у истом тренутку Т2 закључа М2. Након тога нит Т1 остане блокирана јер чека да се откључа М2, а нит Т2 остане блокирана јер чека да се откључа Т2.

Helgrind креира граф који представља све променљиве које се могу закључати, 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 [4].

Алат *DRD* не открива овај тип грешака.

Трка за подацима

DRD

```

rtrk@rtrkw579-lin:/export/valgrind$ ./vg-in-place --tool=drd --read-var-info=yes drd/tests/rwlock_race
==17000== drd, a thread error detector
==17000== Copyright (C) 2006-2017, and GNU GPL'd, by Bart Van Assche.
==17000== Using Valgrind-3.14.0.GIT and LibVEX; rerun with -h for copyright info
==17000== Command: drd/tests/rwlock_race
==17000==
==17000== Thread 3:
==17000== Conflicting load by thread 3 at 0x006010d8 size 4
==17000==   at 0x400900: thread_func (rwlock_race.c:29)
==17000==   by 0x4C31074: vgDrd_thread_wrapper (drd_pthread_intercepts.c:444)
==17000==   by 0x4E51183: start_thread (pthread_create.c:312)
==17000==   by 0x5164FFC: clone (clone.S:111)
==17000== Location 0x6010d8 is 0 bytes inside global var "s_racy"
==17000== declared at rwlock_race.c:18
==17000== Other segment start (thread 2)
==17000==   at 0x4C3C1B7: pthread_rwlock_rdlock_intercept (drd_pthread_intercepts.c:1581)
==17000==   by 0x4C3C1B7: pthread_rwlock_rdlock (drd_pthread_intercepts.c:1592)
==17000==   by 0x40090C: thread_func (rwlock_race.c:28)
==17000==   by 0x4C31074: vgDrd_thread_wrapper (drd_pthread_intercepts.c:444)
==17000==   by 0x4E51183: start_thread (pthread_create.c:312)
==17000==   by 0x5164FFC: clone (clone.S:111)
==17000== Other segment end (thread 2)
==17000==   at 0x4C3D837: pthread_rwlock_unlock_intercept (drd_pthread_intercepts.c:1766)
==17000==   by 0x4C3D837: pthread_rwlock_unlock (drd_pthread_intercepts.c:1780)
==17000==   by 0x400925: thread_func (rwlock_race.c:30)
==17000==   by 0x4C31074: vgDrd_thread_wrapper (drd_pthread_intercepts.c:444)
==17000==   by 0x4E51183: start_thread (pthread_create.c:312)
==17000==   by 0x5164FFC: clone (clone.S:111)
==17000==
==17000== Conflicting store by thread 3 at 0x006010d8 size 4
==17000==   at 0x400916: thread_func (rwlock_race.c:29)
==17000==   by 0x4C31074: vgDrd_thread_wrapper (drd_pthread_intercepts.c:444)
==17000==   by 0x4E51183: start_thread (pthread_create.c:312)
==17000==   by 0x5164FFC: clone (clone.S:111)
==17000== Location 0x6010d8 is 0 bytes inside global var "s_racy"
==17000== declared at rwlock_race.c:18
==17000== Other segment start (thread 2)
==17000==   at 0x4C3C1B7: pthread_rwlock_rdlock_intercept (drd_pthread_intercepts.c:1581)
==17000==   by 0x4C3C1B7: pthread_rwlock_rdlock (drd_pthread_intercepts.c:1592)
==17000==   by 0x40090C: thread_func (rwlock_race.c:28)
==17000==   by 0x4C31074: vgDrd_thread_wrapper (drd_pthread_intercepts.c:444)
==17000==   by 0x4E51183: start_thread (pthread_create.c:312)
==17000==   by 0x5164FFC: clone (clone.S:111)
==17000== Other segment end (thread 2)
==17000==   at 0x4C3D837: pthread_rwlock_unlock_intercept (drd_pthread_intercepts.c:1766)
==17000==   by 0x4C3D837: pthread_rwlock_unlock (drd_pthread_intercepts.c:1780)
==17000==   by 0x400925: thread_func (rwlock_race.c:30)
==17000==   by 0x4C31074: vgDrd_thread_wrapper (drd_pthread_intercepts.c:444)
==17000==   by 0x4E51183: start_thread (pthread_create.c:312)
==17000==   by 0x5164FFC: clone (clone.S:111)
==17000==
Result: 2
==17000==
==17000== For counts of detected and suppressed errors, rerun with: -v
==17000== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 30 from 30)

```

Слика 3.14: Пример детекције трке за подацима

DRD исписује поруку сваки пут када открије да је дошло до трке за подацима у програму. Треба имати у виду пар следећих ствари приликом тумачења исписа који нам алат *DRD* даје. Прво, *DRD* додељује свакој нити јединствени број *ID*. Бројеви који се додељују нитима крећу од један и никада се не користи исти број за више нити. Друго, термин сегмент се односи на секвенцу

узастопних операција чувања, читања и синхронизације које се извршавају у једној нити. Сегмент увек почиње и завршава се операцијом синхорнизације. Анализа трке за подацима се извршава између два сегмента уместо између појединачних операција читања и чувања података, искључиво због учинка. На крају, увек постоје два приступа меморији приликом трке за подацима. *DRD* штампа извештај о сваком приступу меморији које је довело до трке за подацима.

На слици 3.14 је дат испис алата *DRD* када пронађе да је дошло до трке за подацима у програму [3].

Helgrind

Трка за подацима може да се јави услед недостатка 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 [4].

Принцип приступа променљивој без адекватне синхорнизације

На слици 3.15 приказан је пример програма у ком се користи променљива 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 3.15: Пример приступа променљивој без адекватне синхронизације

У овом примеру проблем настаје јер ништа не спречава нит родитељ и нит дете да у исто време приступе и промене вредност дељене променљиве *var*. Приликом анализе оваквог програма алатом *Helgrind* добија се извештај који је приказан на слици 3.16.

У извештају који је приказан на слици 3.16 можемо тачно да видимо које нити приступају променљивој без синхронизације, где се врши сам приступ променљивој, име и величину саме променљиве којој нити приступају ради промене њене вредности [4].

Алгоритам детекције приступа променљивој без синхронизације

Алгоритам за детекцију приступа променљивој без синхронизације односи се на „десило се пре” приступ (енгл. „*happens-before*” relation). У наставку је дат пример који објашњава „десило се пре” принцип 3.17.

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 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 променљиве *var*. На пример, можемо да 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 пошаљемо 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

```
---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 3.16: Извештај *Helgrind*-а за приступ промеливој без синхронизације



```
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 3.17: „Десило се пре” принцип

програмских нити које обезбеђује да се сам приступ одвија по тачно одређеном редоследу, без обзира на стварне стопе напретка појединачних нити.

Стандардне примитиве нити креирају „десило се пре” везу:

- Ако је мутекс откључан 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 промељива 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 „десио се пре” релацији, иако постији призвољно дугачак ланац синхорнизације 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 „десио 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 3.16.

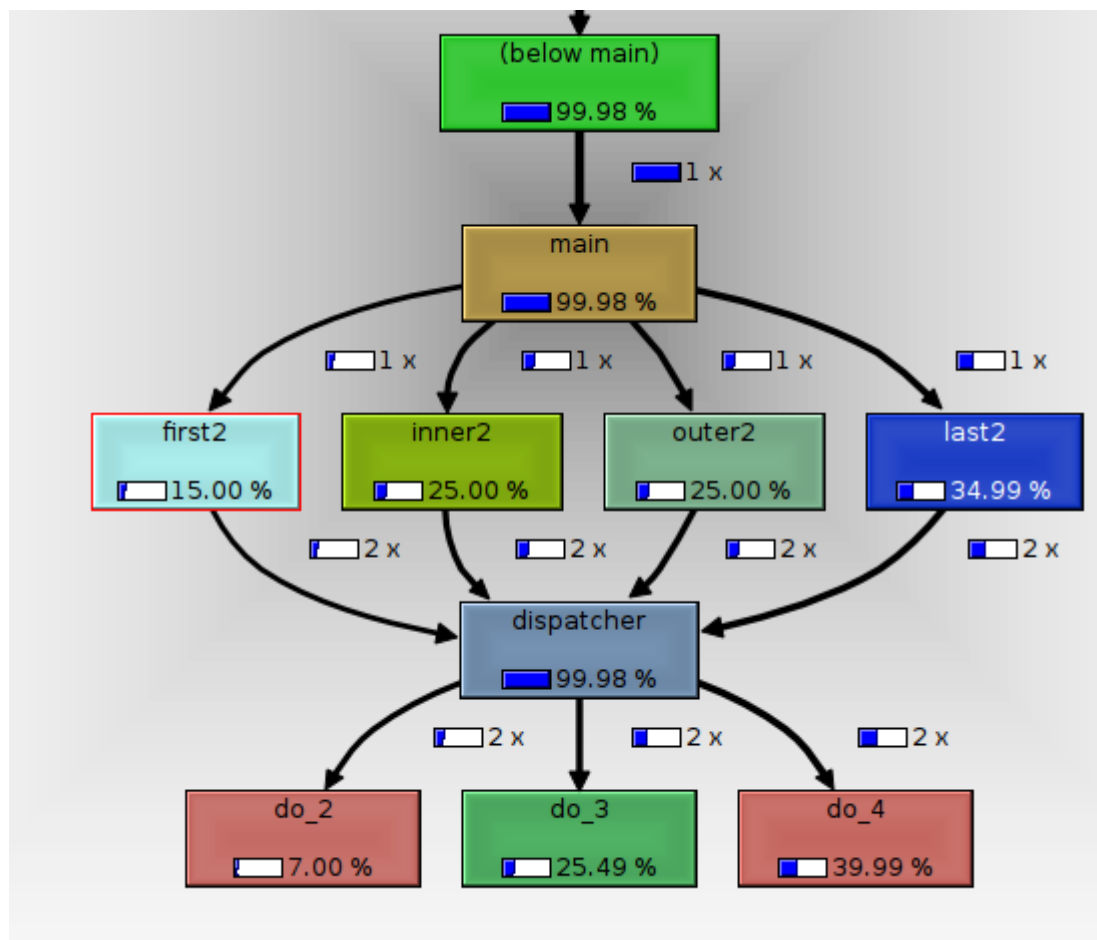
На слици 3.16 можемо да приметимо да *Helgrind* најпре исписује податке где су нити које узрокују грешку направљене. Главни података о грешци почиње са „*Possible data race during read*”. Затим се исписује адреса где се несинхрони приступ меморији дешава, као и величина меморије којој се приступа. У наставку *Helgrind* исписује где друга нит приступа истој локацији. На крају, *Helgrind* покренут са опцијом `--read-var-info=yes` исписује и само име променљиве којој се приступа, као и где у програму је та променљива декларисана [4].

Задржавање катанаца

Приликом рада нити може доћи до појаве задржавања катанца, при којој једна нит не може да ради због блокирања других нити. Дешава се да нит мора да чека да мутекс или синхорнизациони *reader-write* објекат буду откључани од стране друге нити. Оваква појава је непожељна у вишенитним системима, алат *DRD* открива овај тип проблема.

Задржавање катанаца ствара кашњења, која би требало да буду што је могуће краћа. Опције `--exclusive-threshold=<n>` и `--shared-threshold=<n>` омогућавају да *DRD* открије претерано задржавање катанца, тако што ће пријавити свако задржавање катанца које је дужи од задатог прага [3].

Алат *Helgrind* не открива овакав тип грешака.



Слика 3.18: Пример визуелизације функција

3.5 Callgrind

Callgrind је алат који генерише листу позива функција корисничког програма у виду графа. У основним подешавањима 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 изршавања.

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 (*KCAshegrind*), 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 [2].

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

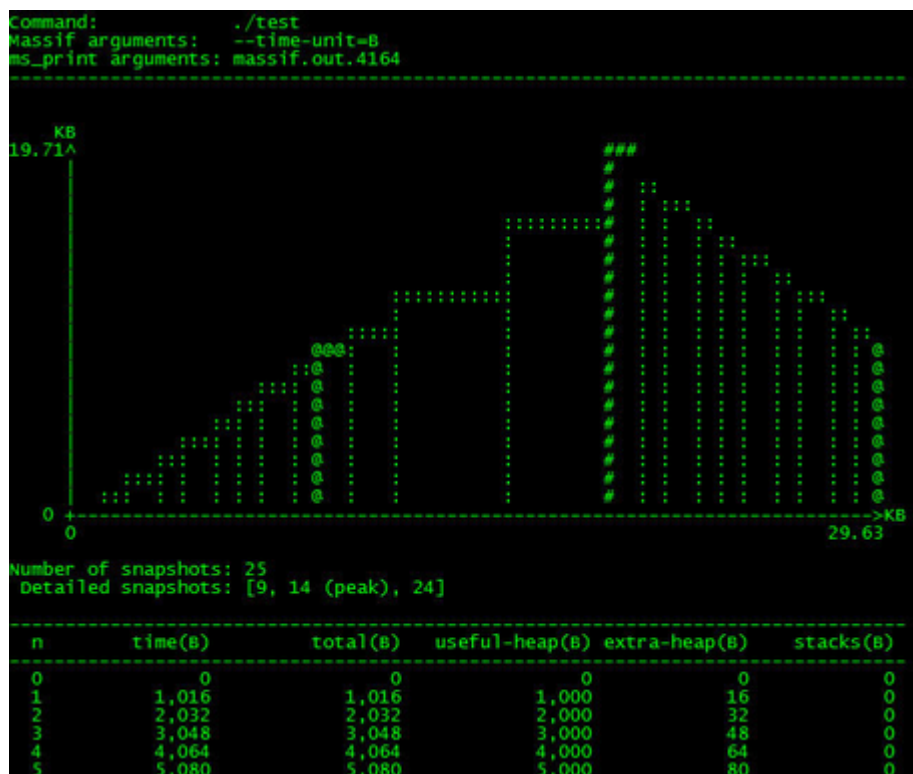
Алат *Callgrind* проширује ову функционалност тако што пропагира цену функције до њених граница. На пример, ако функција *foo* позива функцију *bar*, цена функције *bar* се додаје функцији *foo*. Када се овај механизам примени на целу функцију, добија се слика такозваних инклузивних позива, где цена 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 које 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 зависност 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 [2].

3.6 Massif

Massif је алат за анализу хип меморије корисничког програма. Обухвата, како меморију којој корисник може да приступи, тако и меморију која се користи за помоћне концепте као што су *book-keeping* бајтова и простор за поравнање. Може да израчуна величину стек меморије програма, али ова опција није подразумевана, већ мора експлицитно да се наведе.

Слика 3.19: Приказ оптерећења хипа коришћењем *Massif* алата

Анализа програмског хипа, на модерним рачунарима који користе виртуалну меморију, доноси предност у виду убрзавања програма, јер мањи програми имају бољу искоришћеност кеша и избегавају страничење. Код програма који захтевају велику количину меморије, добра искоришћеност хипа смањује шансу за изгладљивањем простора за размену (енг. *swap space*) корисничке машине.

Постоје одређени сценарији цурења меморије који не спадају у класичне, и такве пропусте не могу детектовати алати као што је *Memcheck*. Ово се дешава зато што меморија није заправо изгубљена, показивач на њу и даље постоји, али она се више не користи. Програми са оваквим типом цурења меморије доводе до непотребног коришћења одређене количине меморије током свог извршавања. *Massif* помаже у откривању баш оваквих цурења меморије.

Massif не даје само информацију о томе колико хип меморије се користи, већ и детаљне информације које упућују на то који део програма је одговоран за алоцирање те меморије [5].

Коришћење *Massif*-а

Програм који се извршава под алатом *Massif* ради веома споро. Након завршетка рада, све статистике су исписане у фајл. Подразумевани фајл у који се пише је *massif.out.<pid>*, где *<pid>* представља ID процеса. Може се променити фајл у коме ће се исписивати командом *--massif-out-file*.

Да би информације које је *Massif* сакупио могли да видимо у читљивом формату, користимо *ms_print*. Ако имамо фајл *massif.out.1234*:

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

ms_print прозводи граф који показује на трошење меморије током извршавања програма, као и детаљне информације о различитим тачкама програма које су одговорне о алокацији меморији. Коришћење различитих скрипти за презентацију резултата је намерно, јер одваја сакупљање података од презентације, као и да је могуће додати нов начин приказа података у сваком тренутку.

На слици 3.19 приказан је пример излаза из алата *Massif*. Величина графа може бити промењена коришћењем *ms_print* опција *--x* и *--y*. Свака вертикала представља пресек стања искоришћености меморије у одређеном тренутку времена. Текст на дну слике 3.19 показује да смо имали 25 пресека. *Massif* почиње тако што одради пресек за сваку алокацију и деалокацију хипа, али ако се програм извршава дуж *Massif* све ређе врши пресеке. У случају сложених програма, који се извршавају дуж *Massif* не чува почетне пресеке када достигне максималну вредност пресека. Подразумевана количина пресека коју алат *Massif* чува је 100, ово се може променити коришћењем опције *--max-snapshots*. Ово значи да је одговарајући број пресека стања сачуван у сваком тренутку рада програма.

Одређени пресеци су детаљније обрађени. Обрађени пресеци су представљени у графу знаком '@'. На дну слике 3.19 је наглашено да постоје три стања која су детаљније обрађена (стање 9, 14 и 24). Подразумевано је да се свако десето стање обрађује детаљније, али и ово се може променити опцијом *--detailed-freq*.

Детаљније обрађени пресеци могу бити представљени и знаком '#', али у том случају значи да је искоришћеност меморије у том тренутку била највећа. Овакав пресек мора да се појави барем једном у графу. На слици 3.19 се види да је такав пресек, пресек број 14.

Утврђивање који од пресека је пресек са најискоришћенијом меморијом није увек тачан. Разлог томе је да *Massif* узима у обзир само пресеке код којих се десила деалокација, овим се избегава обрађивање многих непотребних пресека, али значи да ако програм никада не деалоцира меморију, неће бити обрађених пресека са најискоришћенијом меморијом. Још један разлог јесте да обрађивање пресека са најискоришћенијом меморијом одузима доста времена. Због тога *Massif* чува само она стања чија величина износи 1% од величине пресека где је искоришћеност меморије била највећа [5].

Мерење целокупне меморије

Треба нагласити да алат *Massif* мери само хип меморију, односно меморију која је алоцирана *malloc*, *calloc*, *realloc*, *memalign*, *new*, *new[]* и пар других сличних функција. Ово значи да *Massif* не мери меморију која је алоцирана системским позивима као што су *mmap*, *mremap* и *brk*. Такође, у програму могу да постоје системски позиви за алоцирање меморије, *Massif* неће узети у обзир ту меморију током анализе програма.

Уколико нам је од великог значаја да се узме у обзир сва алоцирана меморија у нашем програму, потребно је укључити опцију `--pages-as-heap=yes`. Укључивањем ове опције, *Massif* неће профајлирати хип меморију, већ странице у меморији [5].

Глава 4

Имплементација *FPXX* конвенције

Сама структура *Valgrind* језгра се може поделити на два дела *VEX* и *coregrind*. У *VEX* делу су имплементирани инструкције за сваку од архитектура коју *Valgrind* подржава, односно овај део је одговоран за дисасемблирање и асемблирање. У *VEX* делу најбитнија функција за дисасемблирање је *disInstr_MIPS*, ова функција је различита за сваку архитектуру и њено име је у складу са сваком архитектуром. У овој функцији се врши диасемблирање свих инструкција у *IR*. Приликом асемблирања избор инструкција врши се рекурзивним обиласком стабла инструкција у дубину, где су за обилазак зависно од типа вредности израза коришћене следеће функције: *iselWordExpr_R* користи се за целобројне вредности, *iselInt64Expr* за 64-битне на 32-битном *MIPS*-у, *iselDblExpr* за 64-битне вредности са покретним зарезом на 32-битним системима и *iselFltExpr* за вредности са покретним зарезом. Обилажење се почиње функцијом *iselSB_MIPS* којој се прослеђује сам исказ. Све ове функције смештене су у фајлу *VEX/priv/host_mips_isel.c*. У фајлу *VEX/priv/guest_mips_helpers.c* садржане су помоћне функције које се умећу у *Valgrind*-ов код ради иницијализације и израчунавања флегова који постављају инструкције које раде са бројевима у покретном зарезу. У фајловима *coregrind/m_syswrap/syswrap-mips32-linux.c* и *coregrind/m_syswrap/syswrap-mips64-linux.c* је одређено пресретање системских позива за 32-битну и 64-битну архитектуру *MIPS*.

У *coregrind* је смештен управљачки део *Valgrind*-а, имплементација системских позива и сигнала. Најбитнији део овде јесте фајл *coregrind/m_scheduler/scheduler.c*. У овом фајлу се организује извршавање програма који се пушта кроз *Valgrind* и покреће *Valgrind*-ове системе за управљање системским позивима и сигнаlima. Пре покретања програма кроз функције у фајлу *coregrind/*

m_scheduler/scheduler.c потребно је утврдити могућности система, што се ради у фајлу *m_machine.c*, као и постављање система за управљање меморијом програма који се извршава. У фајлу *coregrind/m_ume/elf.c* се налазе функције које читају и обрађују заглавље учитаног програма.

У оквиру рада су решавана два проблема. Први проблем је био да се омогући превођење самог алата *Valgrind* са опцијом *-mfpxx*, док је други проблем био да се омогући да се програми преведени са опцијом *-mfpxx* коректно извршавају кроз *Valgrind*.

Да би могле да се виде измене које су прихваћене потребно је имати цео кôд алата *Valgrind*. Преузимање кода врши се покретањем следеће команде у терминалу:

```
git clone git://sourceware.org/git/valgrind.git
```

Након тога треба се лоцирати у директоријуму алата. У наставку ће бити дате команде које је потребно извршити из терминала и директоријума у коме се налази код алата *Valgrind*.

4.1 Превођење алата *Valgrind* са опцијом *-mfpxx*

Да би омогућили превођење алата *Valgrind* са опцијом *-mfpxx*, било је потребно прилагодити асемблерске делове *Valgrind*-а тако да буду у складу са *FPXX* конвенцијом. На слици 4.1 је приказана промена асемблерског дела алата *Valgrind*, односно уклањање из коришћења регистара са непарним индексом. Инструкција *mtc1* замењена је инструкцом *ldc1* пошто она ради са 64-битним вредностима, односно смешта 64-битну вредност у *FP* регистар. Ова измена је имплементира у *VEX/priv/guest_mips_helpers.c*. Као што се види на слици промењена су два макроа која се позивају из функције *mips_dirtyhelper_calculate_FCSR_fp32*. Ова измена се може видети покретањем следеће команде у терминалу

```
git show 2746f7e70b2737b0744592409564463a9203c5f0
```

Имплементацијом ове измене, више није познато у ком режиму ће се *Valgrind* извршавати, што повлачи додатне измене кода алата *Valgrind*, које су описане у наставку.

```

#define ASM_VOLATILE_UNARY32_DOUBLE(inst)
__asm__ volatile("cfc1 $t0, $31" "\n\t"
-             "ctc1 %3, $31" "\n\t"
-             "mtc1 %1, $f20" "\n\t"
-             "mtc1 %2, $f21" "\n\t"
+             "ctc1 %2, $31" "\n\t"
+             "ldc1 $f20, 0(%1)" "\n\t"
             #inst" $f20, $f20" "\n\t"
             "cfc1 %0, $31" "\n\t"
             "ctc1 $t0, $31" "\n\t"
             : "=r" (ret)
-             : "r" (loFsVal), "r" (hiFsVal), "r" (fcsr)
+             : "r" (&fsVal), "r" (fcsr)
             : "t0", "$f20", "$f21"
);

#define ASM_VOLATILE_BINARY32_DOUBLE(inst)
__asm__ volatile("cfc1 $t0, $31" "\n\t"
-             "ctc1 %5, $31" "\n\t"
-             "mtc1 %1, $f20" "\n\t"
-             "mtc1 %2, $f21" "\n\t"
-             "mtc1 %3, $f22" "\n\t"
-             "mtc1 %4, $f23" "\n\t"
+             "ctc1 %3, $31" "\n\t"
+             "ldc1 $f20, 0(%1)" "\n\t"
+             "ldc1 $f22, 0(%2)" "\n\t"
             #inst" $f20, $f20, $f22" "\n\t"
             "cfc1 %0, $31" "\n\t"
             "ctc1 $t0, $31" "\n\t"
             : "=r" (ret)
-             : "r" (loFsVal), "r" (hiFsVal), "r" (loFtVal),
-             : "r" (hiFtVal), "r" (fcsr)
+             : "r" (&fsVal), "r" (&ftVal), "r" (fcsr)
             : "t0", "$f20", "$f21", "$f22", "$f23"
);

```

Слика 4.1: Прилагођавање асемблерских делова алата *Valgrind*

4.2 Детектовање режима у којем ради алат *Valgrind*

Након омогућавања превођења алата *Valgrind* са опцијом *-mfpxx*, следећи задатак је био детектовање режима у ком се *Valgrind* извршава. Детекција самог режима имплементирана је у *coregrind* делу, тачније у фајлу *coregrind/m_machine.c* у функцији *machine_get_hwcaps*. У овој функцији се одређују карактеристике система, као што је начин уписа у меморију. Овде је одређена и детекција *FP* режима у ком алат *Valgrind* треба да ради. Претходна имплементација, која је и уклоњена, није узимала у обзир могућност више *FP* режима. На слици 4.2

је приказана имплементација саме детекције.

```
-      /* Check if CPU has FPU and 32 dbl. prec. FP registers */
-      int FIR = 0;
-      __asm__ __volatile__(
-          "cfc1 %0, $0" "\n\t"
-          : "=r" (FIR)
-      );
-      if (FIR & (1 << FP64)) {
-          vai.hwcaps |= VEX_MIPS_CPU_32FPR;
-      }
-
-      VG_(convert_sigaction_fromK_to_toK>(&saved_sigill_act, &tmp_sigill_act);
-      VG_(sigaction)(VKI_SIGILL, &tmp_sigill_act, NULL);
-      VG_(sigprocmask)(VKI_SIG_SETMASK, &saved_set, NULL);
-
-#      if defined(VGP_mips32_linux)
+      Int fpmode = VG_(prctl)(VKI_PR_GET_FP_MODE);
-#      else
+      Int fpmode = -1;
-#      endif
+
+      if (fpmode < 0) {
+          /* prctl(PR_GET_FP_MODE) is not supported by Kernel,
+           * we are using alternative way to determine FP mode */
+          double result = 0;
+          /* Bit representation of (double)1 is 0x3FF0000000000000. */
+          __asm__ volatile(
+              ".set push\n\t"
+              ".set noreorder\n\t"
+              ".set oddspreg\n\t"
+              "lui $t0, 0x3FF0\n\t"
+              "ldc1 $f0, %0\n\t"
+              "mtc1 $t0, $f1\n\t"
+              "sdc1 $f0, %0\n\t"
+              ".set pop\n\t"
+              : "+m"(result)
+              : "t0", "$f0", "$f1", "memory");
+
+          fpmode = !(result == 1);
+      }
+
+      if (fpmode == 1)
+          vai.hwcaps |= VEX_MIPS_HOST_FR;
+
+      VG_(debugLog)(1, "machine", "hwcaps = 0x%x\n", vai.hwcaps);
+      VG_(machine_get_cache_info>(&vai);
```

Слика 4.2: Детектовање режима

Овде се прво врши провера да ли језгро подржава *PR_GET_FP_MODE* и *PR_SET_FP_MODE* опције . Уколико их не подржава извршава се асем-

блерски код. Инструкција *sdc1* се извршава другачије у зависности од режима и помоћу ње ми добијамо информацију у ком режиму треба да ради сам алат *Valgrind*.

Ова измена се може видети покретањем следеће команде у терминалу

```
git show 030cea68c804abc61facd95e894a1c8b2418904f
```

4.3 Одређивање режима у којем програм почиње са радом

Valgrind као виртуална машина ради у једном *FPU* режиму, ако је преведен са опцијом *-mfpvx* онда ће радити у режиму које језгро одреди. С друге, стране он емулира јединицу за операције са покретним зарезом, с тога може да емулира другачији режим рада. Због тога је потребно имплементирати алгоритам за избор *FPU* режима као што је то урађено у језгру. У фајлу *coregrind/m_ume/elf.c* имплементиране су следеће функције *arch_elf_pt_proc()* и *arch_check_elf()*. За имплементацију ових функција као референца се може користити функције из језгра, тачније функције у фајлу *linux/arch/mips/kernel/elf.c*. Функција *arch_elf_pt_proc()* проверава заглавље учитаног програма да провери исправност и/или подобност у односу на систем на ком се извршава. Функција *arch_check_elf()* одређује у ком режиму ће радити програм који се пушта кроз *Valgrind*. На слици 4.3 дат је део функције *arch_check_elf()* у ком се одређује режим рада.

Ова измена се може видети покретањем следеће команде у терминалу

```
git show 030cea68c804abc61facd95e894a1c8b2418904f
```

4.4 Пресретање системског позива *prctl()*

У одређеним ситуацијама *Valgrind* мора да пресретне системске позиве и сам их обрађује. Потреба за таквим случајем јавља се и приликом имплементације овог решења. Пресретање системског позива *prctl()* у *Valgrind*-у је било неопходно да би подршка за режим *FPXX* била потпуна. Пресретање системског позива *prctl()* је одрађена у фајловима *coregrind/m_syswrap/syswrap-mips32-linux.c* и *coregrind/m_syswrap/syswrap-mips64-linux.c*. Овај системски позив је обавијен макроом *PRE*, што значи да алат *Valgrind* неће допустити да

```

/* Determine the desired FPU mode

Decision making:

- We want FR_FRE if FRE=1 and both FR=1 and FR=0 are false. This
  means that we have a combination of program and interpreter
  that inherently require the hybrid FP mode.
- If FR1 and FRDEFAULT is true, that means we hit the any-abi or
  fpxx case. This is because, in any-ABI (or no-ABI) we have no FPU
  instructions so we don't care about the mode. We will simply use
  the one preferred by the hardware. In fpxx case, that ABI can
  handle both FR=1 and FR=0, so, again, we simply choose the one
  preferred by the hardware. Next, if we only use single-precision
  FPU instructions, and the default ABI FPU mode is not good
  (ie single + any ABI combination), we set again the FPU mode to the
  one is preferred by the hardware. Next, if we know that the code
  will only use single-precision instructions, shown by single being
  true but frdefault being false, then we again set the FPU mode to
  the one that is preferred by the hardware.
- We want FP_FR1 if that's the only matching mode and the default one
  is not good.
- Return with ELIBADD if we can't find a matching FPU mode. */
if (prog_req.fre && !prog_req.frdefault && !prog_req.fr1)
    state->overall_fp_mode = VKI_FP_FRE;
else if ((prog_req.fr1 && prog_req.frdefault) ||
         (prog_req.single && !prog_req.frdefault))
    state->overall_fp_mode = VEX_MIPS_HOST_FP_MODE(vai.hwcaps) ?
                           VKI_FP_FR1 : VKI_FP_FR0;
else if (prog_req.fr1)
    state->overall_fp_mode = VKI_FP_FR1;
else if (!prog_req.fre && !prog_req.frdefault &&
         !prog_req.fr1 && !prog_req.single && !prog_req.soft)
    return VKI_ELIBBAD;

/* TODO: Currently, Valgrind doesn't support FRE and doesn't support FR1
  emulation on FR0 system, so in those cases we are forced to
  reject the ELF. */
if ((state->overall_fp_mode == VKI_FP_FRE) ||
    ((state->overall_fp_mode == VKI_FP_FR1) &&
     !VEX_MIPS_HOST_FP_MODE(vai.hwcaps)))
    return VKI_ELIBBAD;

return 0;
}

```

Слика 4.3: Одређивање режима у којем програм почиње са радом

prctl() оде до самог језгра, већ ће он сам одрадити све што је потребно. Имплементација пресретања системског позива *prctl()* дата је на слици 4.4. На слици се види да се одрађена имплементација флегова *PR_GET_FP_MODE* и *PR_SET_FP_MODE*. Сви флегови који су дефинисани у самом коду језгра, дефинисани су и у алату *Valgrind*, једина разлика је у префиксу *VKI*.

Приликом имплементације флега *PR_SET_FP_MODE*, врши се пролазак кроз све активне нити и мења се режим у ком оне раде. Функција *SET_STATUS_Success(0)* спречава системски позив да оде до самог језгра и изврши се, већ се овде зауставља.


```

PRE(sys_prctl)
{
    switch (ARG1) {
        case VKI_PR_SET_FP_MODE:
        {
            VexArchInfo vai;
            VG_(machine_get_VexArchInfo)(NULL, &vai);
            /* Reject unsupported modes */
            if ((ARG2 & ~VKI_PR_FP_MODE_FR) ||
                ((ARG2 & VKI_PR_FP_MODE_FR) &&
                 !VEX_MIPS_HOST_FP_MODE(vai.hwcaps))) {
                SET_STATUS_Failure(VKI_EOPNOTSUPP);
            } else {
                if (!(VG_(threads)[tid].arch.vex.guest_CP0_status &
                     MIPS_CP0_STATUS_FR) != !(ARG2 & VKI_PR_FP_MODE_FR)) {
                    ThreadId t;
                    for (t = 1; t < VG_N_THREADS; t++) {
                        if (VG_(threads)[t].status != VgTs_Empty) {
                            if (ARG2 & VKI_PR_FP_MODE_FR) {
                                VG_(threads)[t].arch.vex.guest_CP0_status |=
                                    MIPS_CP0_STATUS_FR;
                            } else {
                                VG_(threads)[t].arch.vex.guest_CP0_status &=
                                    ~MIPS_CP0_STATUS_FR;
                            }
                        }
                    }
                    /* Discard all translations */
                    VG_(discard_translations)(0, 0xfffffffful, "prctl(PR_SET_FP_MODE)");
                }
                SET_STATUS_Success(0);
            }
        }
        break;
    }
    case VKI_PR_GET_FP_MODE:
    {
        if (VG_(threads)[tid].arch.vex.guest_CP0_status & MIPS_CP0_STATUS_FR)
            SET_STATUS_Success(VKI_PR_FP_MODE_FR);
        else
            SET_STATUS_Success(0);
        break;
    }
    default:
        WRAPPER_PRE_NAME(linux, sys_prctl)(tid, layout, arrghs, status, flags);
        break;
    }
}

```

Слика 4.4: Пресретање системског позива *prctl()*

Ова измена се може видети покретањем следеће команде у терминалу

```
git show 030cea68c804abc61facd95e894a1c8b2418904f
```

4.5 Тестирање

У почетним фазама развоја тестирано је само правилно препознавање *FP* режима у којем ради програм. Ово је рађено помоћу једноставног програма који

је превођен на различите начине. Касније, како је развој привођен крају, тестирање је вршено помоћу теста који током свог извршавања мења режим рада. Овај тест је сада један од тестова којим се свакодневно проверава исправност рада алата *Valgrind*. Путања до теста, уколико се налазимо у самом директоријуму *Valgrind* алата, је `none/tests/mips32/change_fp_mode.c`. На слици 4.5 је приказан део теста где се детектује и мења режим рада. Као што може да се види, на два начина се детектује режим, један је помоћу асемблерског кода, док је други помоћу системског позива *prctl()* и флага *PR_GET_FP_MODE*. Врши се провара променљивих у којима су сачуване вредности режима, да ли имају исту вредност. Након тога се мења режим рада помоћу системског позива *prctl()* и флага *PR_SET_FP_MODE* и опет се врши детекција режима. Током извршавања тестова алат *Valgrind* користи скрипту која врши проверу да ли је излаз који даје тест исти као очекивани, односно као излаз програма када ради без *Valgrind* алата. Сваки тест има фајл у којем се налази излаз без посредства *Valgrind* алата. Ако се ова два излаза поклапају, значи да је тест прошао и да алат *Valgrind* исправно ради.

```

static int get_fp_mode(void) {
    unsigned long long result = 0;
    __asm__ volatile(
        ".set push\n\t"
        ".set noreorder\n\t"
        ".set oddspreg\n\t"
        "lui $t0, 0x3FF0\n\t"
        "ldcl $f0, %0\n\t"
        "mtcl $t0, $f1\n\t"
        "sdcl $f0, %0\n\t"
        ".set pop\n\t"
        : "+m"(result)
        : "t0", "$f0", "$f1", "memory");

    return (result != 0x3FF000000000000ull);
}

static void test(int* fr_prctl, int* fr_detected) {
    *fr_prctl = prctl(PR_GET_FP_MODE);
    *fr_detected = get_fp_mode();

    if (*fr_prctl < 0) {
        fatal_error("prctl(PR_GET_FP_MODE) fails.");
    }
    printf("fr_prctl: %d, fr_detected: %d\n", *fr_prctl, *fr_detected);

    if (*fr_prctl != *fr_detected) {
        fatal_error("fr_prctl != fr_detected");
    }
}

int main() {
    int fr_prctl, fr_detected;

    test(&fr_prctl, &fr_detected);

    /* FP64 */
    if (fr_prctl == 1) {
        /* Change mode to FP32 */
        if (prctl(PR_SET_FP_MODE, 0) != 0) {
            fatal_error("prctl(PR_SET_FP_MODE, 0) fails.");
        }
        test(&fr_prctl, &fr_detected);
        /* Change back FP mode */
        if (prctl(PR_SET_FP_MODE, 1) != 0) {
            fatal_error("prctl(PR_SET_FP_MODE, 1) fails.");
        }
        test(&fr_prctl, &fr_detected);
    }
    return 0;
}

```

Слика 4.5: Тест *change_fp_mode.c*

Глава 5

Закључак

Откривање грешака попут цурења меморије и утркивања приликом приступа податку од стране више нити у великим системи може бити веома напорно, дуготрајно па и немогуће у неким случајевима. Анализом кода посредством *Valgrind* алата може се уштедети драгоцено време. Помоћу *Valgrind* алата могу се открити грешке које не утичу на исправност рада програма, али знатно утичу на перформансе програма, уколико се не открију. Грешке које могу да се пронађу су многобројне. Алат *Memcheck* открива меморијске грешке које су често и најтеже за детектовање. Неки од проблема које може да открије јесу коришћење неицијализованих вредности или неиправно ослобађање хип меморије. Алат *Cachgrind* симулира и прати приступ брзој меморији машине на којој се програм извршава. Алати *Helgrind* и *DRD* су одлични за детектовање и уклањање грешака које се јављају приликом рада са нитима. Алат *Massif* анализира хип меморију корисничког програма и алат *Callgrind* врши анализу графа позива функција.

Циљ овог рада је био упознавање са алатом *Valgrind* и додавање подршке за *FPXX* конвенцију. Како је ова конвенција специфична за *MIPS* архитектуру, на почетку рада дата је кратка историја и осврт на специфичности ове архитектуре. Описани су регистри који су битни у *MIPS* архитектури, а мало већи акценат је стављен на регистре за рад са бројевима у покрентом зарезу. *MIPS* архитектура користи два формата ових регистара, регистре са једноструком и двоструком прецизношћу.

У овом раду је описана *FPXX* конвенција као и њена имплементација у алату *Valgrind*. Једна од првих карактеристика је била забрана коришћења регистара са непарним индексом, па је то био један од првих задатака током

имплементације. Обрађен је и системски позив *prctl()*, коме је посвећена пажња приликом анализе, па и имплементације. Након завршетка имплементације, алат *Valgrind* има много већи спектар употребе и анализе програма.

Као могући правац даљег рада, планира се додавање имплементације *FRE* режима. Наиме, овај режим је саставни део *FPXX* конвенције при чему алат *Valgrind* тренутно нема подршку за овај режим. Да би се додао овај режим, потребно је додавање подршке за *MIPS R6* скуп инструкција.

Библиографија

- [1] Cachegrind: a cache and branch-prediction profiler. <http://valgrind.org/docs/manual/cg-manual.html>, 2000-2017.
- [2] Callgrind: a call-graph generating cache and branch prediction profiler. <http://valgrind.org/docs/manual/cl-manual.html>, 2000-2017.
- [3] DRD: a thread error detector. <http://valgrind.org/docs/manual/drd-manual.html>, 2000-2017.
- [4] Helgrind: a thread error detector. <http://valgrind.org/docs/manual/hg-manual.html>, 2000-2017.
- [5] Massif: a heap profiler. <http://valgrind.org/docs/manual/ms-manual.html>, 2000-2017.
- [6] Memcheck: a memory error detector. <http://valgrind.org/docs/manual/mc-manual.html>, 2000-2017.
- [7] Valgrind. <http://valgrind.org/>, 2000-2017.
- [8] MIPS O32 ABI - FR0 and FR1 Interlinking. https://dmz-portal.mips.com/wiki/MIPS_O32_ABI_-_FR0_and_FR1_Interlinking, 2015.
- [9] Linux Programmer's Manual. <http://man7.org/linux/man-pages/man2/prctl.2.html/>, 2017.
- [10] What is RISC and CISC Architecture with Advantages and Disadvantages. <http://www.edgefxkits.com/blog/what-is-risc-and-cisc-architecture/>, 2017.
- [11] MIPS by Imagination. *MIPS Architecture For Programmers Volume II-A: The MIPS32 Instruction Set*. MIPS Technologies, Inc, 2013.

- [12] MIPS by Imagination. *MIPS SIMD Architecture*. MIPS Technologies, Inc, 2014.
- [13] Dominic Sweetman. *See MIPS Run*. Morgan Kaufman Publishers, 2007.