УНИВЕРЗИТЕТ У БЕОГРАДУ
МАТЕМАТИЧКИ ФАКУЛТЕТ

Бранислава Б. Живковић

ПАРАЛЕЛИЗАЦИЈА СТАТИЧКЕ ВЕРИФИКАЦИЈЕ СОФТВЕРА

мастер рад

Београд, 2016.

Ментор:

др Милена Вујошевић Јаничић, доцент
Универзитет у Београду, Математички факултет

Чланови комисије:

др Саша Малков, ванредни професор
Универзитет у Београду, Математички факултет

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

Датум одбране: 2016.

Брату, мами и тати

Наслов мастер рада: Паралелизација статичке верификације софтвера

Резиме:

Кључне речи: паралелизација, верификација, рачунарство

Садржај

Глава 1

Увод

Развој и примена нових технологија почиње све више да утиче на друштво. Рачунари су присутни свуда око нас и пружају нам многе погодности и олакшице како у необавезним тако и у пословним активностима. Институције попут банака, здравства, просвете, и других теже да аутоматизују своје процесе коришћењем разних софтверских алата.

Грешке у софтверу могу имати катастрофалне последице, на пример отказивање медицинског уређаја у току операције или отказивање аутоматског пилота у току лета. Како бисмо предухитрили овакве ситуације, потребно је детаљно и прецизно утврдити исправност развијеног софтвера.

Испитивање исправности софтвера може бити динамичко и статичко. Статичка верификација софтвера представља најпоузданији начин испитивања исправности програма, уз битно теоријско ограничење, Халтинг проблем [16]. Наиме, није могуће направити алгоритам који потпуно прецизно испитује исправност произвољног програма у коначном времену користећи коначне ресурсе.

Мотивација

Аутоматизовано статичко утврђивање исправности софтвера захтева анализу програмског кода без његовог извшавања, коришћењем техника статичке анализе. Ове технике могу бити веома сложене и захтевне у пољу времена и ресурса, што додатно ограничава сам процес.

Алати за аутоматску статичку верификацију софтвера због свих поменутих ограничења могу бити веома комплексни, због чега је погодно покретати

их на рачунарима са јаким хардверским перформансама. Временом су процесори са више језгара почели да доминирају тржиштем и данас готово сваки нов рачунар садржи вишејезгарни процесор. Такође, рачунари са више процесора постају све доступнији што нам даје могућност да пишемо софтвер који имплементира конкурентно извршавање и користи сав потенцијал вишепроцесорских и вишејезгарних система.

У оквиру овог рада надограђен је систем за аутоматску верификацију софтвера ЛАВ [7]. Процес испитивања исправности програма је паралелизован на два нивоа и приказани су експериментални резултати поређења секвенцијалног и паралелног извршавања система ЛАВ као и експериментално поређење са алатом CBMC [4]. Резултати показују да паралелизација може значајно да убрза, а у неким ситуацијама заправо и омогући ефикасан процес утврђивања исправности програма.

Глава 2

Верификација

Верификација софтвера представља дисциплину рачунарства која се бави провером и доказивањем исправности програма. Програм је исправан уколико задовољава задату спецификацију, односно уколико за сваки улаз даје одговарајуће понашање предвиђено спецификацијом. Постоје два основна приступа верификацији *динамичка* и *статичка* верификација.

2.1 Динамичка верификација

Динамичка верификација програма се врши током његовог извршавања. Грешке у програму се покушавају пронаћи исцрпним тестирањем што је и циљ динамичке верификације. Битно је нагласити да тестирањем није могуће доказати исправност програма већ је могуће пронаћи грешке и на тај начин оповргнути претпоставку о исправности програма.

Да би се програм тестирао потребно је пронаћи одговарајући скуп улазних података помоћу којих се врши тестирање. С обзиром на то да је простор могућих улаза углавном превелики, није могуће тестирати програм за све могуће улазе. Због тога треба издвојити одговарајући подскуп улазних података који што боље описује спецификацију програма и покрива што већи број случајева. Избор улазних података се углавном врши коришћењем програмског кода и спецификације.

Постоје две основне методе тестирања, метод *црне* и *беле* кутије. Методом црне кутије генерисање тестова се врши на основу спецификације програма не узимајући у обзир детаље имплементације. Методом беле кутије тестови се генеришу на основу кода и структуре програма. Такође, постоји и метод сиве

кутије који представља мешавину ова два приступа. У зависности од тога шта је потребно тестирати бира/користи се одговарајући метод [8].

2.2 Статичка верификација

Статичка верификација програма представља испитивање исправности програма анализом програмског кода, без његовог извршавања. Анализа програмског кода се врши углавном над изворним или објектним кодом. Особине програма и услови исправности се описују одговарајућим формулама изабране математичке теорије. Изграђене формуле се даље анализирају коришћењем стриктних математичких метода. Неодлучивост халтинг проблема нам говори да није могуће испитати да ли је нека наредба програма достижна, па тиме ни да ли је програм потпуно исправан. Због тога се особине програма апроксимирају и описују одлучивим математичким теоријама. Аутоматизоване технике статичке анализе су *проверавање модела*, *апстрактна интерпретација* и *симболичко извршавање*.

Проверавање модела је техника верификације којом се испитује да ли модел система/програма задовољава одговарајућу спецификацију. Модел програма се описује коначним аутоматом који се састоји од стања и прелаза између стања, а спецификација се описује логичком формулом. Испитивање исправности програма се врши исцрпним и систематским обиласком стања аутомата како би се доказали услови задати спецификацијом. Уколико доказивање није могуће, генерише се одговарајући контрапример [5].

Апстрактна интерпретација представља метод верификације код кога се семантика програма апроксимира математичким моделом. Понашање програма се описује одговарајућим апстрактним доменом и релацијама над њиме. Анализом апстрактног домена могуће је добити информације о резултатима рада програма без његовог извршавања [5].

Симболичко извршавање је метод верификације који анализира понашање програма на основу симболичких вредности променљивих. Путање програма се описују симболичким изразима а испитивање исправности програма се врши анализом конструисаних израза. Резултати анализе неке путање програма важе за све могуће улазне вредности променњивих које прате дату путању [9].

2.3 Алати за верификацију

У наставку ће бити описани неки верификациони алати који користе поменуте технике приликом испитивања исправности програма.

LLBMC (енг. Low-Level Bounded Model Checker) је алат који проверава исправност C и C++ програма. Користи технике за проверавање ограничених модела над LLVM међукодом. Проналази грешке дељења нулом, прекорачење бафера, неисправног и двоструког ослобађања меморије, прекорачења у аритметичким изразима и проверава кориснички задате услове. За проверавање услова исправности користи SMT решаваче STP i Boolector [12].

PAGAI је верификациони алат отвореног кода који ради над LLVM инфраструктуром. Имплементира технике апстрактне интерпретације и генеришући аутоматски инваријанте петљи које касније уписује у анализирани код. Такође може да проверава кориснички задате услове кроз макрое претпоставки (енг. assertion macro) [6].

KLEE је алат који врши симболичко извршавање и генерисање тест примера над програмима који су писани у језику C. Настао је на Универзитету Илиноис и јавно је доступан. KLEE анализира LLVM међукод користећи SMT решавач STP приликом испитивања услова исправности [3].

CBMC је алат за верификацију C и C++ програма првенствено намењен за верификацију уграђених система (енг. embedded systems). CBMC врши симболичко израчунавање тако што изворни код програма претаче у SMT формулу и испитује њену задовољивост [4].

2.4 ЛАВ

ЛАВ је алат за статичку верификацију софтвера [7]. Отвореног је кода и доступан је на адреси http://argo.matf.bg.ac.rs/?content=lav.

ЛАВ имплементира статичку анализу, генерисање и испитивање услова исправности императивних програма комбинујући методе описане у поглављу 2.2. Користи LLVM инфраструктуру ради трансформације програма у форму која је погодна за анализу и првенствено је намењен за анализу програма написаних у

програмском језику C. Међутим, универзалност LLVM платформе омогућава и анализу других процедуралних језика који се могу компајлирати у LLVM код, као што су језици C++, Ada и Fortran. ЛАВ моделује понашање програма и генерише услове исправности трансформишући их у формуле одговарајуће теорије логике првог реда. Теорије које су подржане су аритметика бит-вектора, линеарна аритметика, теорија неинтерпретираних функција (или акерменизација) и теорија низова. Симболичким извршавањем ЛАВ генерише формуле изабране теорије логике првог реда које описују понашање сваког блока LLVM међукода као и њихове релације.

ЛАВ генерише формуле исправности неке наредбе и испитује њихову задовољивост користећи SMT решавач. Комбинацијом ових формула граде се формуле које описују понашање програма.

Формуле се конструишу по потреби у различитим контекстима. Контекст дефинише информације из околине наредбе које ће бити узете у разматрање приликом расуђивања о наредби. Контексти који постоје су празан контекст, контекст блока, контекст функције и контекст у коме је функција позвана. Разликују се по ширини од најужег до најширег. Приликом испитивања исправности наредбе почиње се са празним контекстом. Након генерисања формуле у оквиру једног контекста, формула се шаље SMT решавачу на проверу. У зависности од резултата испитују се и шири контексти.

На основу резултата решавача ЛАВ генерише извештај о безбедности наредби програма [17].

Глава 3

Паралелизација

Термини паралелизам и конкурентност у рачунарству су углавном испреплетани и погрешно схваћени. Често се грешком поистовећују и сматрају синонимима иако то нису. Због тога је битно да их правилно дефинишемо и разликујемо [13].

Дефиниција 1. *Конкурентност је својство програма које се односи на то да два или више задатака могу бити истовремено у току. (исправити превод)*

Дефиниција 2. *Паралелизам је својство програма да извршава два или више задатака истовремено. (исправити превод)*

Битно је напоменути да постоји разлика у томе да ли се два задатка истовремено извршавају или су истовремено у току. Наиме, паралелизам изискује/захтева конкурентност, док обрнуто не важи. Може се рећи да је конкурентност начин структуирања програма а паралелизам начин извршавања програма. Конкурентни програми се могу извршавати паралелно али не морају. Паралелизам захтева архитектуру која има више процесорских јединица, док се конкурентност може остварити и на једном процесору.

Паралелно програмирање је област рачунарства која се бави архитектуром система и софтверским проблемима програма са паралелним извршавањем. Програм се може дефинисати као низ наредби које се извршавају након његовог покретања. Секвенцијалне програме одликује серијско извршавање наредби. Паралелизам је карактеристика програма која се односи на независност његових израчунавања. Независна израчунавања се могу истовремено односно паралелно извршавати на више процесорских јединица.

3.1 Мотивација

Интересовање за паралелизацију се јавља касних 1950-их са зачетком теоријских основа док се први технички напредак осећа почетком 1960-их и и даље се развија/расте. Први суперрачунари су се појавили 60-их година и имали су више процесора који су могли паралелно да раде са дељеном меморијом. Даљим развојем 80-их година се појављују кластери, системи који се састоје од великог броја рачунара тзв. чворова међусобно повезаних преко мреже. 90-их година са експанзијом интернета се појављује рачунарство у облаку, док данас већина кућних рачунара садржи процесоре са више језгара.

Може се рећи да перформансе рачунара експоненцијално расту од 1945 године за фактор 10 сваких 5 година. Први рачунари су израчунавали десетине операција са бројевима у покретном зарезу у секунди, паралелни рачунари 1990-их достижу број од пар десетина милијарди операција у секунди. Рачунарске/софтверске архитектуре су морале да испрате овакав нагли раст што се постиже преласком са секвенцијалног на паралелно програмирање [18].

Перформансе софтверских решења зависе од времена извршавања основних операција, попут операција са бројевима у покретном зарезу, као и од броја оваквих операција које се могу извршавати паралелно. С обзиром на то да ово време зависи од брзине откуцаја часовника процесора која полако тежи ка теоријском максимуму (брзина светлости) не можемо се ослонити на то да ће бржи процесори подићи перформансе нумеричких израчунавања. Главна мотивација и циљ паралелног програмирања је подизање перформанси рачунарских система односно убрзање програма. Потреба за паралелним приступом расте и због тога што је секвенцијалним приступом решавање многих комплексних проблема временски захтевно.

Убрзање паралелизацијом је мера која показује колико пута паралелни програми брже решавају исте проблеме него секвенцијални програми. Формула убрзања је следећа [1]:

$$S = T_s/T_p$$

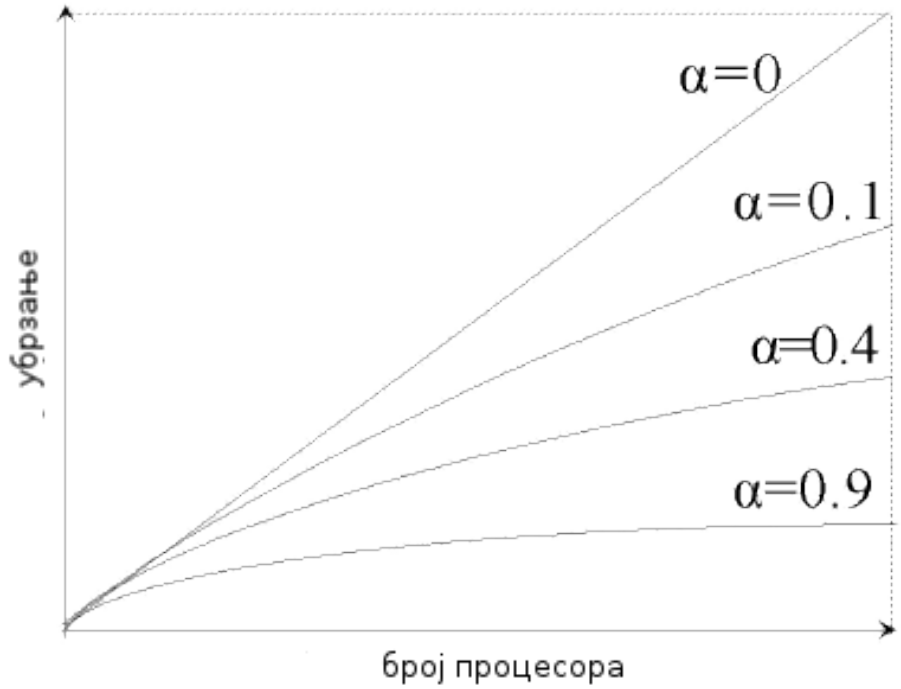
где T_s представља време извршавања секвенцијаног а T_p време извршавања паралелног програма за исти проблем.

По Амдаловом закону, извршавање паралелног програма на паралелном рачунару углавном обухвата и део операција које се не могу извршавати паралелно. Означимо са α део програма који се мора извршавати секвенцијално на

једном процесору, а остатак $(1-\alpha)$ се може извршити паралелно. Ако је N број процесорских јединица, формула убрзања је:

$$S = 1/(\alpha + (1-\alpha)/N)$$

Ова формула нам показује да убрзање никада не може прећи $1/\alpha$, тј. број процесорских јединица не утиче на део програма који се мора извршавати секвенцијално. На слици 3.1 је приказана зависност убрзања од броја процесора и дела посла који се мора обавити секвенцијално.



Слика 3.1: Зависност убрзања од броја процесора за неке вредности α

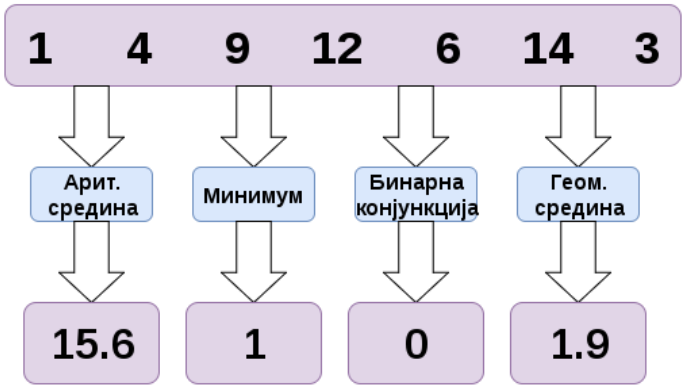
У пракси, време извршавања програма на паралелним системима је углавном веће од теоријски израчунате вредности јер зависи и од других параметара попут комуникације и синхронизације. Амдалов модел не узима у обзир ова времена и разматра само случајеве у којима је димензија проблема фиксирана. Поред Амдаловог модела постоје и други модели као што су Густафсонов, Гинтеров, модел Сун Ни-ја који превазилазе нека ограничења Амдаловог модела [14].

3.2 Врсте паралелизације

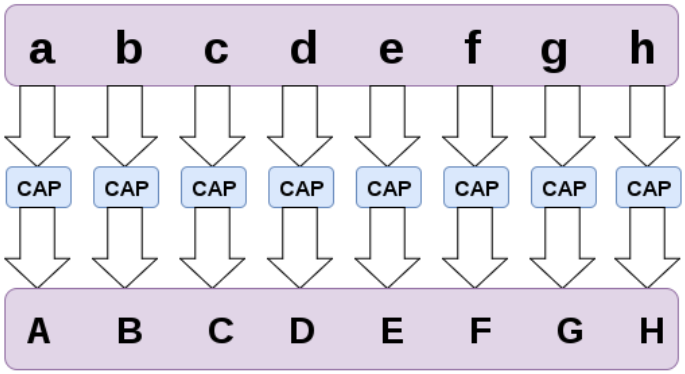
Програми се могу паралелизовати на различите начине. Паралилзацију може обављати програмер експлицитно или коришћењем неких алата. Последњих година развијени су многи алати који омогућују аутоматску паралелизацију. Коришћењем ових алата програмеру је олакшан процес паралелизације уз ограничену контролу. Овакав начин паралелизације је погодан за велике и комплексне системе код којих би ручна паралелизација била спора и компликована.

Са друге стране, ручна паралелизација програма захтева добро обучене програмере и углавном је сложенија али пружа програмерима потпуну контролу над самим процесом паралелизације. Овакав приступ је погоднији за паралелизацију специфичних проблема.

Битно је нагласити да није могуће паралелизовати све делове сваког алгоритма. Посао програмера је да пронађе и одлучи који делови алгоритма се могу паралелизовати и на који начин. Два најчешћа приступа дизајнирању паралелних алгоритама су *паралелизација задатака* и *паралелизација података*. Паралелизација задатака представља раслојавање алгоритма на независне задатке који се могу извршавати било којим редоследом над истим скупом података. Паралелизација података представља раслојавање података тако да се један задатак може независно извршавати над дисјунктним деловима података било којим редоследом [2]. На слици 3.2 је приказан пример паралелизације задатака а на слици 3.3 пример паралелизације података



Слика 3.2: Пример паралелизације података: примена функције capslock над сваким словом појединачно



Слика 3.3: Пример паралелизације задатака: примена различитих функција над свим подацима

3.3 Проблеми

Често није могуће раслојити алгоритам на потпуно независне задатке који се могу паралелно извршавати већ је присутан одређен ниво зависности између њих. На пример, уколико задаци могу приступити истој променљивој у програму и променити њену вредност, потенцијално више задатака може истовремено покушати да је измени. У таквим ситуацијама задаци се надмећу за приступ дељеним подацима. Овакви и овоме слични проблеми индукују постојање *критичне секције*. Критична секција представља низ наредби који мора задовољавати следећи услов: уколико је један задатак ушао у критичну секцију и почео да је извршава, ниједан други задатак је не сме извршавати истовремено. У складу са тиме постоје бројна решења и механизми који омогућавају *комуникацију* и *синхронизацију* између задатака.

Комуникација представља било који вид размене информација између задатака. Може се остварити преко дељене меморије или слањем односно примањем порука. Комуникација преко порука се одвија тако што један задатак експлицитно шаље податке другом задатку који их прихвата и обрађује. Неки од познатих механизама за овакав вид комуникације су сигнали, цеви, сокети и канали. Постојање дељене меморије има своје добре и лоше стране. Понекад је потребно старати се о редоследу читања односно писања дељене меморије као и о евентуалним утркивањима и сукобима. Због тога комуникација преко дељене меморије може захтевати одређен ниво синхронизације. Кроз механизме синхронизације програмер може контролисати редослед извршавања задатака и приступ дељеној меморији. Две основне врсте овакве синхронизације су *са-*

радња и *такмичење*. Синхронизација сарадње између два задатка је потребна уколико један задатак зависи од резултата рада другог. Синхронизација такмичења је неоходна у случајевима када два задатка истовремено захтевају исти ресурс. Механизми који се користе за имплементацију синхронизације су мутекси, катанци, семафори, монитори и други [15].

Синхронизација отвара многе проблеме, нпр. могућност *узајамног блокирања*, *живог блокирања* и *изгладњивања* између задатака. Ови проблеми се односе на концепт *напредовања* (енг. liveness) програма. Концепт напредовања програма представља својство програма да у току свог извршавања напредује доводећи до предвиђеног догађаја у неком тренутку, односно да константно прави прогрес током свог извршавања. Уколико ово својство није задовољено може да се деси да програм не може да настави са својим извршавањем. Узајамно блокирање (енг. deadlock) се дешава у ситуацији када два задатка чекају један на другог како би наставили са радом и на тај начин губе напредак. Живо блокирање (енг. livelock) представља ситуацију када сви задаци раде али нема напретка. Изгладњивање се односи на могућност да један задатак спречава извршавање другог. Поменути проблеми се могу спречити одговарајућим алгоритмима [11].

3.4 Алати за паралелизацију

У овом поглављу ће бити описане неке библиотеке које се користе за паралелизацију програмског кода. Акценат ће бити на библиотекама за језик Ц++.

Алате можемо поделити у две категорије имплицитне и експлицитне. Имплицитни алати олакшавају програмеру имплементацију паралелних алгоритама јер се старају о прављењу, управљању и синхронизацији нити. Експлицитни алати пружају већу флексибилности и контролу захтевајући од програмера да управља свим аспектима вишенитности.

PThreads (енг. POSIX Threading interface) је интерфејс за паралелно програмирање на нивоу оперативног система и доступан је у оквиру већине UNIX-оликих оперативних система. Имплементиран је у оквиру заглавља `pthread.h` језика Ц који садржи скуп константи, типова и функција за паралелизацију. Програмеру је омогућено прављење нити и управљање њиховим извршавањем. Комуникација се обавља преко дељене меморије коју та-

кође програмер контролише. Дељена меморија се имплементира коришћењем глобалних променљивих које су видљиве свим нитима. Садржај заглавља `pthread.h` уз одговарајућу документацију се може наћи на адреси http://man7.org/linux/man-pages/man7/pthreads.7.html.

OpenMP (енг. Open Specification for Multi-Processing) је интерфејс за програмирање који омогућава паралелно програмирање у језицима C++, C++ и Фортран . Заснива се на моделу паралелизације коришћењем дељене меморије. Садржи скуп компајлерских директива, рутина и глобалних променљивих које служе за обележавање делова програмског кода. Програм се дели на регионе који се извршавају серијски и регионе који се извршавају паралелно. Региони се означавају директивама које управљају процесом додељивања задатака нитима, комуникацијом и синхронизацијом. Променљиве могу бити дељене, односно видљиве свим нитима, и приватне, односно видљиве у оквиру нити унутар које су декларисане. Детаљна документација се може наћи на адреси http://www.openmp.org.

TBB (енг. Thread Building Blocks) је C++ библиотека за паралелно програмирање на вишепроцесорским системима развијена од стране *Intel*-а. Библиотека се састоји од бројних шаблона који имплементирају паралелне алгоритме, контејнере, примитиве за синхронизацију и управљач задацима. Програмери дефинишу задатке који ће се извршавати паралелно након чега се управљач задацима стара о току извршавања и техничким детаљима. Више о овој библиотеци се може наћи на адреси http://www.threadingbuildingblocks.org.

Глава 4

Имплементација

Модул који се бави паралелизацијом је развијен у складу са специфичним захтевима система ЛАВ. Систем ЛАВ је сложен верификациони алат писан у C++ језику. Као такав користи многе спољне библиотеке, алате, као и СМТ решаваче. Архитектура самог система је модуларна, функционалне целине су издвојене у посебне модуле и по потреби увезиване и коришћене. Модул за паралелизацију представља посебну издвојену целину тако да се може универзално користити у различитим деловима система. Имплементиран је коришћењем и комбинацијом различитих библиотека језика C++. У наставку текста ће бити описана архитекура и имплементација модула за паралелизацију као и начини његовог коришћења у оквиру система ЛАВ. Изворни код је јавно доступан [1].

4.1 Опис проблема

Процес верификације у систему ЛАВ је имплементиран секвенцијално. На улазу се задаје модул (скуп датотека) који је потребно анализирати. ЛАВ разлаже модул на његове функције и почев од `main` функције (име почетне функције може да се зада) прави граф позива функција на основу кога се функције верификују. Даље, функције се разлажу на блокове, а блокови на појединачне наредбе. Симболичким извршавањем конструише се SMT формула која описује услов исправности неке потенцијално небезбедне наредбе. SMT решавачу се на проверу шаље негација ове формуле и уколико је она незадовољива, онда

[1] Може се пронаћи на адреси https://github.com/milenavj/LAV (директоријуми `include/lav/Threads` и `lib/Threads`) а интеграција са ЛАВ-ом се налази унутар класа `LBlock`, `LModule` и `LState`.

то значи да је употреба одговарајуће наредбе исправна. Уколико се за потенцијално небезбедне наредбе једног блока покаже да су исправне, онда то значи да је анализирани блок исправан. Уколико су сви блокови унутар једне функције исправни, онда је и сама функција исправна. Слично важи и за модул.

Као што знамо, програми могу бити комплексни, са великим бројем функција (блокова и наредби), због чега и овај процес може бити дуг и исцрпан. Додатно, функције често позивају једна другу и међусобно су зависне. Због тога, неретко се догађа да функција А позива функцију Б, функција Б позива функцију Ц, док је функција Д неисправна али њена верификација се може обавити тек након што се заврши верификација функција А, Б и Ц. Ова и сличне ситуације су биле мотивација за паралелизацију процеса верификације. Паралелизација је имплементирана на два нивоа. Први ниво је паралелно верификовање различитих наредби у оквиру једног блока, а други ниво је паралелно верификовање самих функција.

Систем ЛАВ имплементира алгоритам који услове исправности програма описује формулама изабране логике првог реда и испитује њихову задовољивост. Испитивање задовољивости је временски веома захтевно и може да варира у зависности од формуле. ЛАВ редом шаље формуле SMT решавачу након чега чека његов одговор. Имајући у виду разноврсност формула могуће је да испитивање првих неколико формула траје кратко, затим да испитивање наредне формуле траје доста дуже, након чега следе формуле чије испитивање поново траје кратко. Тешка формула у оваквој ситуацији блокира цео систем све док се не разреши. Ово може бити значајно успорење система, посебно уколико нека од формула која следи након тешке формуле показује да је наредба, чију исправност испитујемо, неисправна. Како би се избегле овакве ситуације идеја је да се паралелизује овај процес коршћењем могућности вишепроцесорског хардвера. Циљ овог рада је имплементација паралелизације процеса испитивања задовољивости формула које се генеришу у оквиру контекста блока једне наредбе.

4.2 Опис архитектуре

Архитектура модула за паралелизацију је осмишљена тако да испуњава захтеве система ЛАВ. Модул за паралелизацију има три основна дела: контролни део, радне нити и комуникациони део. Улога контролног дела је да управља

радним нитима, ослушкује и прихвата сигнале тј. догађаје које емитују радне нити и обрађује њихове резултате. Радне нити извршавају задатке и обавештавају контролни део о резултатима. Део који се бави комуникацијом омогућава комуникацију између контролног дела и радних нити. Резултати извршавања радних нити се могу сачувати унутар посебно структуре података. Композицијом ових делова имплементиран је модул који паралелизује неке делове алгоритма анализе програмског кода у овиру система ЛАВ.

4.3 Имплементација модула

Модул је имплементиран у језику C++ због компатибилности са системом ЛАВ уз употребу неких системских позива Linux оперативног система. Нека коришћена заглавља C++ језика су: [10]

`<thread>` - Садржи класу `thread` која служи за конструкцију и управљање нитима. Конструктор ове класе може прихватити функцију коју ће нит извршавати. Овако конструисана инстанца има свој јединствени идентификатор и налази се у стању извршавања (*joinable*). Позивом функције `join()` над објектом нити можемо сачекати да нит заврши са извршавањем односно позивом функције `detach()` можемо откачити нит и дозволити да настави своје извршавање независно од главне нити. Још неке од корисних функција су: `hardware_concurrency()` - враћа број нити које су подржане на систему [2], `native_handle()` - враћа објекат руковалаца нити дефинисан имплементацијом и друге.

`<atomic>` - Садржи компоненте које обезбеђују атомичне операције, операције које су безбедне приликом рада у конкурентном окружењу. Класа `atomic` представља атомичан податак са јасно дефинисаним понашањем у ситуацијама када две или више нити покушају да раде са њиме. Објекат ове класе може садржати податак било ког типа. Неке од корисних функција су: `load()` - враћа вредност податка, `store(drugi_podatak)` - замењује тренутни податак другим податком, `fetch_add(drugi_podatak)` - додаје вредност другог податка на тренутни податак и друге.

`<future>` - Олакшава асинхроно извршавање задатака као и приступ и управљање вредностима које су направљене асинхроно. Посебни снабдевачи (објекти класа `package_task`, `promise` или позиви функције `async`) могу извршавати задатке асинхроно и ажурирати вредности које се налазе у дељеном стању. Остали објекти могу читати вредности из дељеног стања или чекати на његову иницијализацију помоћу објекта типа `future`.

`<functional>` - Пружа подршку за рад са објектима који су посебно дизајнирани тако да се могу користити као функције. Такви објекти се називају функционални објекти. Инстанце класе `std::function` представљају један пример таквих објеката. Функција `bind(fun_objekat, arg1, arg2,`

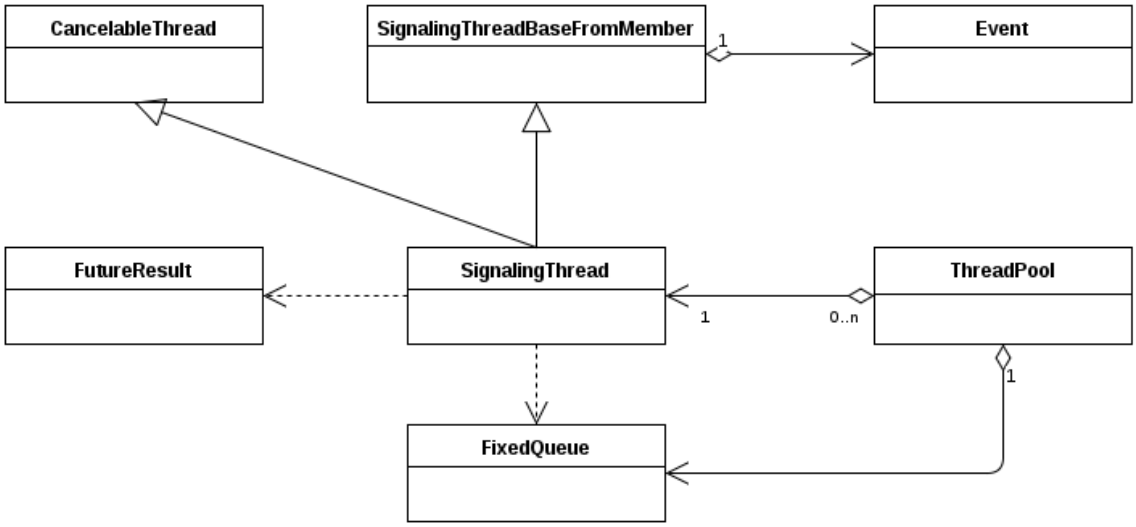
[2] Овај број не мора нужно бити број физичких језгара на систему.

`...)` везује један или више аргумента функционалног објекта и враћа објекат истог типа, другим речима врши парцијалну апликацију функције на прослеђене аргументе.

Приликом имплементације појединачних нити коришћен је модел нити из библиотеке `<thread>`. Класа `ThreadPool` представља контролни део модула, тзв. базен нити. Ова класа садржи контролну нит, вектор радних нити и ред задатака које радне нити треба да изврше. Могуће је задати број нити али уколико другачије није наглашено конструисаће се онолико радних нити колико је потребно (односно минимум броја потребних нити и броја нити које оперативни систем дозвољава). Радне нити су објекти класе `SignalingThread` која енкапсулира објекат типа `std::thread` и омогућава отказивање нити (што се у позадини врши системским позивом оперативног система). Свакој радној нити се приликом иницијализације прослеђује дељени показивач (`std::shared_ptr`) на ред задатака, тако да све нити имају приступ истом објекту реда. Нити скидају један по један задатак са реда задатака и извршавају га. Ред задатака који се налази у базену нити је објекат класа `FixedQueue`). Класа `FixedQueue` је шаблонска класа и може садржати ред објеката било ког типа. Задаци који се смештају у ред су објекати C++ апстракције анонимних (ламбда) функција, и због специфичности проблема имају следећи потпис: `int f()`. Наравно, потпис ових функција се може уопштити, али за потребе овог рада то није било неопходно. Како све нити приступају истом објекту реда, потребно је синхронизовати процес скидања задатака. Класа `FixedQueue` садржи јединствен показивач на објекат атомичног типа (`std::unique_ptr<std::atomic_uint>`). Овај објекат чува информацију о томе колико је задатака скинуто са реда (индекс следећег задатка који треба скинути) и омогућава атомичне операције додавања и читања вредности коју чува. Радне нити могу затражити задатак из реда, и уколико две или више нити у исто време покушају скинути задатак са реда, свака ће добити различит задатак, тако да се приликом испоруке задатака гарантује да ће све нити добити различите задатке. На овај начин је избегнуто утркивање нити као и синхронизација коришћењем традиционалних метода (мутекси, закључавање, и др). Радне нити садрже објекат класе `Event` који служи за емитовање догађаја. `Event` садржи посебну вредност (*event file descriptor*) чијом променом се сигнализира догађај. Овај механизам је имплементиран помоћу системских позива оперативног система због оптимизације. Контролна нит се претплаћује на ослушкивање догађаја сваке нити посебно.

Након што изврше задатак, радне нити емитују догађај који контролна нит дохвата и обрађује. Уколико је нека нит емитовала одређен догађај, контролна нит отказује све радне нити и завршава са радом. Због специфичних потреба приликом паралелне анализе функција, направљена је и класа `FutureResult` која чува статус верификације једне функције. Она садржи дељену вредност коју асинхроно иницијализује објекат типа `std::promise`. Дељеној вредности се приступа помоћу објекта типа `std::future`. Приступање дељеној вредности постаје могуће тек након њене иницијалитације

У наставку ће бити описане неке битније функције појединих класа, док се дијаграм класа налази на слици 4.1. Све класе модула имплементирају померачку семантику (енг. move semantic) како би се онемогућило копирање објеката што је јако важно приликом имплементације конкурентне логике.



Слика 4.1: Дијаграм класа

Класа `Event`

- `void Signal(Sigval value = 1)` - емитује сигнал који је прослеђен као вредност `value`
- `Sigval Value(bool block = false)` - враћа вредност сигнала који је емитован
- `static std::vector<std::size_t> WaitForEvents(std::vector<Event::Pointer>& events, const std::chrono::milliseconds & waitMs = {}, bool block = true)` - омогућава ослушкивање више догађаја истовремено

Класа `SignalingThread`

- `const Event::Pointer& ShareEvent() const` - враћа дељени показивач на објекат догађаја

Класа `FixedQueue`

- `inline std::size_t Size() const` - враћа број задатака у реду
- `inline bool Empty() const` - испитује да ли је ред задатака празан
- `T* Pop()` - скида задатак са реда и враћа његов показивач

Класа `FutureResult`

- `void setResult(int val)` - иницијализује дељену вредност
- `std::shared_future<int>& getSharedFuture()` - враћа референцу на објекат помоћу кога се може прочитати дељена вредност

Класа `ThreadPool`

- `void Init(std::vector<std::function<int()» &&tasks, uint64_t num_threads = std::thread::hardware_concurrency() -1)` - поставља број нити, број задатака и конструише ред задатака
- `void StartWorkerThreads()` - покреће радне нити
- `void StartControlThread()` - покреће контролну нит
- `void Work()` - покреће базен нити

4.4 Интеграција модула са системом ЛАВ

Паралелизација је имплементирана у контексту анализе наредби и функција модула који се верификује.

Паралелна анализа наредби

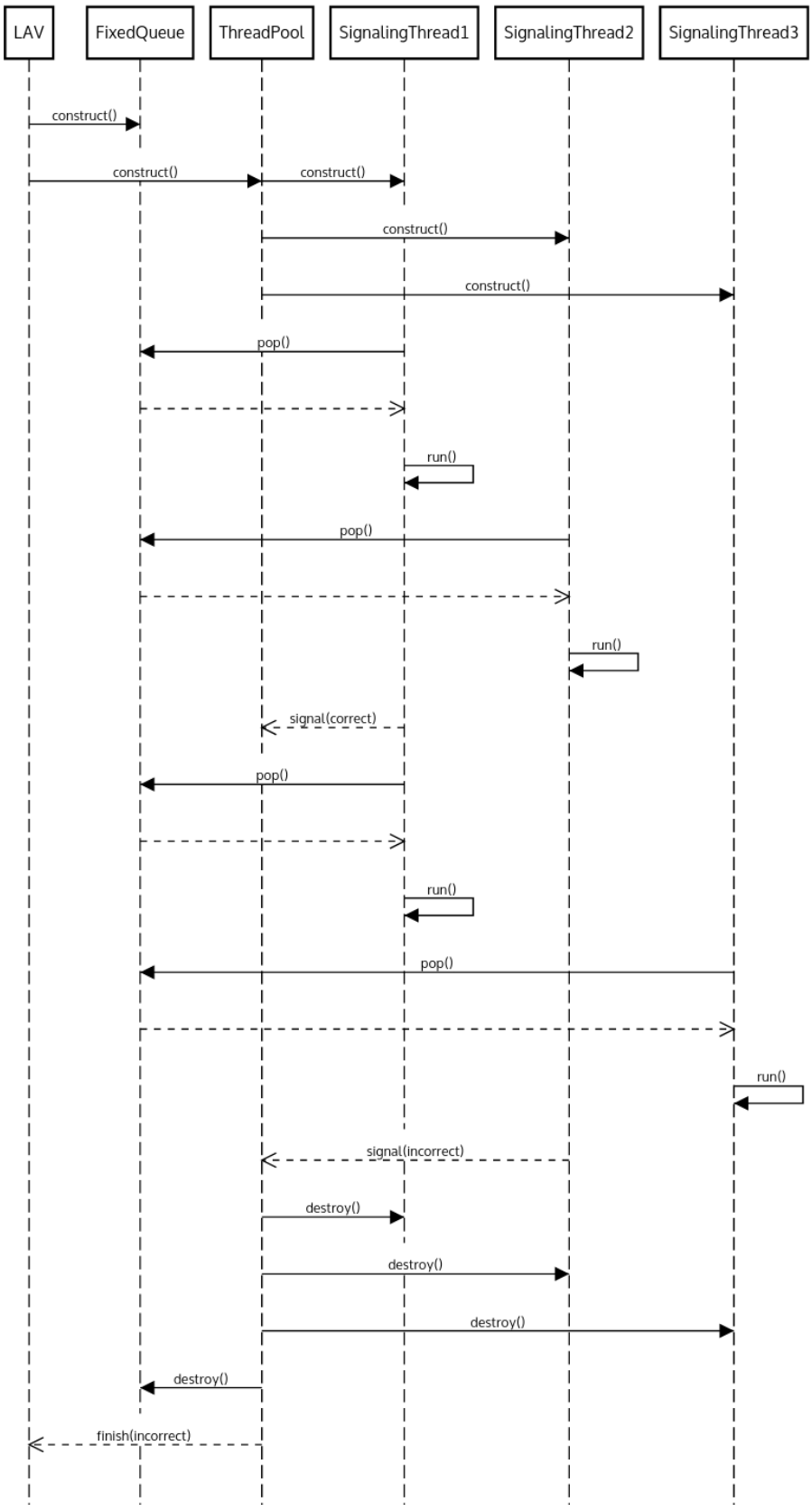
Класа `LBlock` система ЛАВ служи за рад са блоковима кода. Њена функција `CalculateConditions` конструише формуле које представљају услове исправности блока и позивају SMT решавач за сваку формулу. Модул за паралелизацију омогућава да се ови позиви решавача извршавају паралелно.

За сваку формулу, услов исправности, унутар функције `CalculateConditions` конструише се функција која позива SMT решавач. Функција као резултат враћа индикатор да ли је услов исправности испуњен или не. Направљене функције се смештају у ред `FixedQueue` и прослеђују инстанци класе `ThreadPool` (базен нити). Базен нити прави радне нити и покреће их. Свака нит извршава једну по једну функцију, скидајући их са реда и обавештава базен нити о резултату извршавања. Уколико се наиђе на услов исправности који није задовољен, нема потребе испитивати остале услове јер се тада блок сматра неисправним. У контексту имплементације то значи да уколико нека функција врати индикатор да услов исправности није испуњен, нити могу престати са радом јер се задат блок означава као неисправан. Ако је приликом покретања система ЛАВ (била) задата опција `-find-first-flawed` користи се понашање које је описано - прекидање у случају наиласка на неисправну наредбу. Ако та опција није присутна, онда се редом све испитује. Када све функције из реда заврше тако да су сви услови су били задовољени, блок се сматра исправним и тако бива означен.

На слици 4.2 је приказан један могући сценарио. На почетку се врши конструкција и иницијализација свих потребних објеката. Базен нити конструише три радне нити које узимају задатке са реда. Нити `SignalingThread1` прва узима задатак са реда, а након ње и `SignalingThread2` и обе почињу да их извршавају. Нит `SignalingThread1` прва завршава успешно, пре него што је нит `SignalingThread3` узела задатак са реда. Након тога обе нити, `SignalingThread1` и `SignalingThread3` покушају узети следећи задатак. Имајући у виду то да један ред задатака деле све нити, овај процес узимања задатака ће се извршити секвенцијално (користећи погодне функције из библиотеке `<atomic>`) тако да нит `SignalingThread1` прва добија задатак са реда. Како нит `SignalingThread1` наилази на услов исправности који није испуњен, шаље сигнал базену нити након чега остале нити бивају заустављене и систем ЛАВ бива обавештен о неисправном резултату. Можемо приметити да је редослед акција прављења нити, узимање задатака са реда и брзина извршавања задатака у овом примеру конкретизован. Наравно, у општем случају тај редослед је произвољан и зависи од много фактора као што су специфичности оперативног система, сложеност задатака, број задатака у реду, и слично.

Паралелна анализа функција

Анализа функција је имплементирана користећи исте механизме уз мале додатке због различитости проблема. Класа `LModule` је одговорна за анализу функција једног модула. Функције могу зависити једне од других због чега је потребно чувати информације о резултату верификације сваке функције појединачно. Верификација модула почиње позивом метода `Run` класе `LModule` у оквиру кога се конструишу нити које анализирају функције тог модула. Класи `LModule` је додат низ објеката `FutureResult` за сваку функцију тог модула. Класа `FutureResult` представља структуру података унутар које се чува податак о завршетку верификације неке функције. Ова класа садржи објекат класе `std::promise` који чува вредност (информацију о завршетку верификације) и објекат `std::future` помоћу кога нити приступају тој вредности. Свака нит има дељени показивач на низ објеката `FutureResult` тако да у сваком тренутку може погледати резултат верификације неке друге функције. Уколико се верификација тражене функције није завршила, тренутна нит ће сачекати на њен резултат позивом функције `wait` над објектом `std::future` за ту функцију. Када се верификација тражене функције заврши, резултат ће се уписати у њен објекат `std::promise` и функција `wait` ће завршити са чекањем. Наиласком на неисправну функцију, верификација се завршава, програм се означава неисправним, а остале нити се отказују.



Слика 4.2: Дијаграм тока извршавања

Глава 5

Експериментални резултати

У наставку текста ће бити приказани резултати поређења модификованог система ЛАВ са алатом CBMC.

5.1 Архитектура рачунара

С обзиром на то да се рад заснива на паралелној импмементацији, експерименти су покретани на кластер рачунару са четрдесет процесорских језгара и Ubuntu 16.04 оперативним системом. Максимално време извршавања програма је постављено на пет минута и мерено је системским програмом `time`.

5.2 Опис корпуса

Експерименти су покретани на два корпуса, први демонстрира побољшање услед паралелне верификације блокова, а други демонстрира побољшање услед паралелне верификације функција програма.

Први корпус

Овај корпуса има за циљ да провери како се понаша паралелизације наредби услед великог броја сложених наредби у једном блоку. Корпус садржи двадесет програма писаних у програмском језику C. Сви програми у корпусу су конструисани на основу наведеног примера повећавајући број наредби. Програми су преведени у 32b и 64b формат.

Код 5.1: Пример програма

```
int m(int a, int b, int c, int d) {

        // --- I skup ---
        // naredbe koje simuliraju slozena izracunavanja
        a = (b<<3)*((c>>2)/3);
        b = (a<<3)*((c>>2)/3);
        c = (b<<3)*((a>>2)/3);

        // --- II skup ---
        // naredbe koje mogu dovesti do deljenja nulom
        a = b/c + b/a + c/(++b);
        b = a/d;

        return b;
}
```

Програми су генерисани копирањем наредби из првог скупа одређен број пута. Понављање ових наредби симулира комплексна израчунавања која блок потенцијално може да садржи. За наредбе другог скупа је потребно утврдити да ли могу да доведу до дељења нулом. На пример, провера да ли су `a`, `b++` и `c` различите од нуле укључује комплексне формуле настале симболичким извршавањем наредби првог скупа, док је провера да ли је `d` различито од нуле једноставна.

Други корпус

Наредни корпус се састоји од програма који садрже одређен број функција како би се демонстрирала верификација мало сложенијих програма. Корпус је подељен у две категорије од којих свака има две верзије и садржи 2000 програма писаних у програмском језику C.

Прва категорија садржи програме са различитим бројем функција (од једне до педесет функција). Свака функција се састоји од одређеног број наредби које симулирају комплексна извршавања. Повећањем броја наредби отежава се про-

вера коректности функција као и самог програма. Испитивање коректности потенцијално проблематичне наредбе дељења у првој верзији ове категорије не зависи од резултата претходних израчунавања, док у другој верзији зависи од резултата претходних израчунавања. Ово својство програме прве верзије чини лакшим за паралелну верификацију од програма друге верзије. Главна `main` функција позива све постојеће функције, због чега је за испитивање исправности програма потребно испитати исправност свих функција. Примери програма прве и друге верзије ове категорије се могу видети на сликама 5.2 и 5.3.

Друга категорија се састоји од програма са различитим бројем функција које садрже позиве других функција. Повећањем дубине позива функција значајно се компликује испитивање коректности програма. Свака функција, као и у првој категорији, садржи наредбе које симулирају комплексна израчунавања и потенцијално проблематичне наредбе дељења. Такође, прва и друга верзије ове категорије се разликују по сложености проблематичних наредби. Примери програма прве и друге верзије ове категорије се могу видети на сликама 5.4 и 5.5.

Код 5.2: Пример програма прве категорије (прва верзија)

```
int f1(int a, int b, int c) {

 // naredbe koje simuliraju slozena izracunavanja
 a = (b<<1)*((a>>1)/3);
 b = (a<<1)*((b>>1)/3);

 // potencijalno problematicna naredba deljenja
 if(c!=0)
   return a/c;

 return a+b;
}

int main() {
 int a, b, c, r = 0;
```

```
 scanf("%d%d%d", &a, &b, &c);

 r += f1(a,b,c);

 return r;
}
```

Код 5.3: Пример програма прве категорије (друга верзија)

```
int f1(int a, int b) {

 // naredbe koje simuliraju slozena izracunavanja
 a = (b<<1)*((a>>1)/3);
 b = (a<<1)*((b>>1)/3);

 // potencijalno problematicna naredba deljenja
 if(b!=0)
   return a/b;

 return a+b;
}

int main() {
 int a, b, r = 0;

 scanf("%d%d", &a, &b);

 r += f1(a,b);

 return r;
}
```

Код 5.4: Пример програма друге категорије (прва верзија)

```
int _f1(int a, int b, int c) {
```

```
 // naredbe koje simuliraju slozena izracunavanja
 a = (b<<1)*((a>>1)/3);
 b = (a<<1)*((b>>1)/3);

 // potencijalno problematicna naredba deljenja
 if(c!=0)
   return a/c;

 return a+b+c;
}

int f1(int a, int b, int c) {

 // naredbe koje simuliraju slozena izracunavanja
 a = (b<<1)*((a>>1)/3);
 b = (a<<1)*((b>>1)/3);

 // potencijalno problematicna naredba deljenja
 // i poziv druge funkcije
 if(c!=0)
   return a/c + _f1(a,b,c);

 return a+b+c;
}

int main() {

 int a, b, c, r = 0;

 scanf("%d%d%d", &a, &b, &c);

 r += f1(a,b,c);

 return r;
```

```
}
```

Код 5.5: Пример програма друге категорије (друга верзија)

```
#include <stdio.h>

int _f1(int a, int b, int c) {

 // naredbe koje simuliraju slozena izracunavanja
 a = (b<<1)*((a>>1)/3);
 b = (a<<1)*((b>>1)/3);

 // potencijalno problematicna naredba deljenja
 if(b!=0)
   return a/b;

 return a+b+c;
}

int f1(int a, int b, int c) {

 // naredbe koje simuliraju slozena izracunavanja
 a = (b<<1)*((a>>1)/3);
 b = (a<<1)*((b>>1)/3);



 // potencijalno problematicna naredba deljenja
 // i poziv druge funkcije
 if(b!=0)
   return a/b + _f1(a,b,c);

 return a+b+c;
}

int main() {
 int a, b, c, r = 0;
```

```
 scanf("%d%d%d", &a, &b, &c);

 r += f1(a,b,c);

 return r;
}
```

Начини покретања

ЛАВ је покретан са опцијом заустављања приликом наиласка на прву невалидну наредбу, решавач који је коришћен је Z3. Број нити које користи ЛАВ није експлицитно задат[1]. CBMC је покретан са параметром који испитује проблем дељења нулом.

Пример покретања ЛАВ-а:

```
./LAV 25_lines.o -solver=Z3-BV-ARR-EUF -find-first-flawed -enable-parallel
-starting-function=m
```

Пример покретања CBMC-а:

```
./cbmc -div-by-zero-check -32 -function m 25_lines.c
```

5.3 Анализа резултата

Први корпус

Добијени резултати приликом верификације програма из првог корпуса се налазе у табели 5.2, измерена времена су приказана у секундама.

Резултати показују да је време потребно алату CBMC за верификацију програма значајно веће од времена које је потребно алату ЛАВ уз коришћење имплементиране паралелизације на нивоу блока. Разлог овоме је то што алат

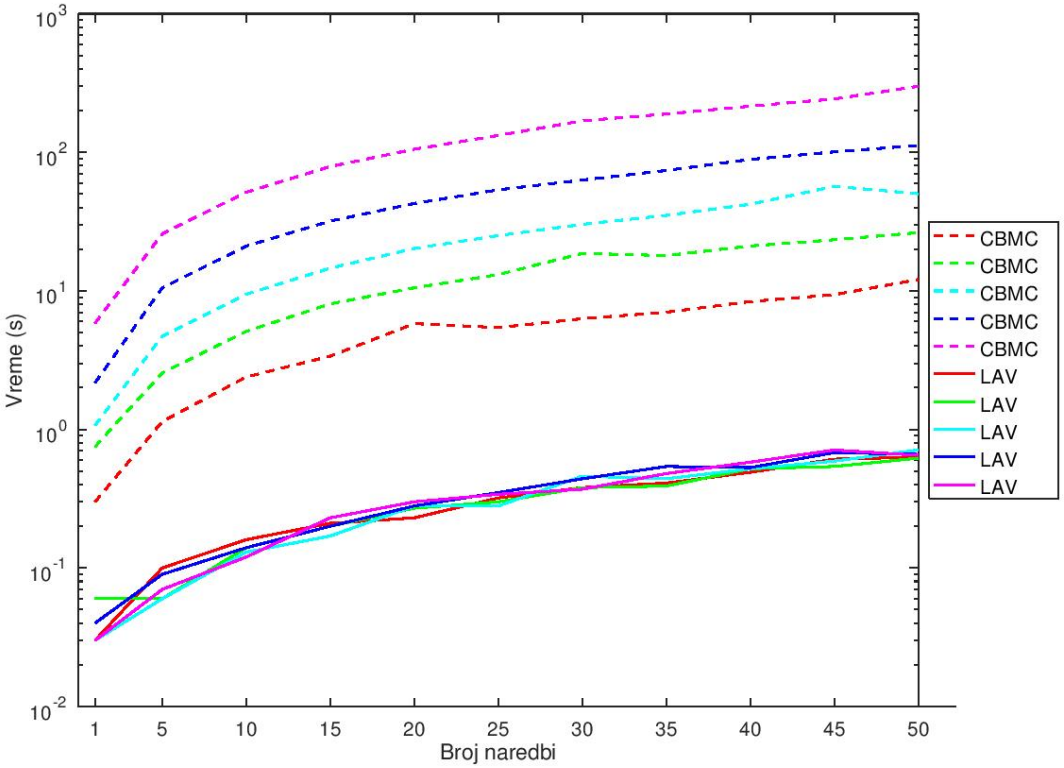
[1] То је број потребних нити односно број који је добијен као препорука оперативног система (и не мора представљати број физичких језгара на рачунару) уколико је потребно више нити него што систем дозвољава.

број наредби	32b		64b	
	LAV	CBMC	LAV	CBMC
12	0.08	0.72	0.07	0.82
13	0.23	0.75	0.25	0.72
14	0.38	0.93	0.08	0.93
15	0.08	1.21	0.08	1.10
16	0.27	1.42	0.09	1.46
17	0.08	2.16	0.10	2.15
18	0.10	3.08	0.23	3.05
19	0.26	4.12	0.09	4.15
20	0.11	7.80	0.21	7.93
21	0.11	11.73	0.23	12.09
22	0.22	16.50	0.23	17.28
23	0.09	33.91	0.10	34.78
24	0.11	52.50	0.10	53.46
25	0.12	75.32	0.09	74.72
26	0.11	157.01	0.10	154.87
27	0.13	246.89	0.12	253.92
28	0.12	↗	0.12	↗
29	0.12	↗	0.13	↗
30	0.14	↗	0.13	↗
60	0.18	↗	0.20	↗

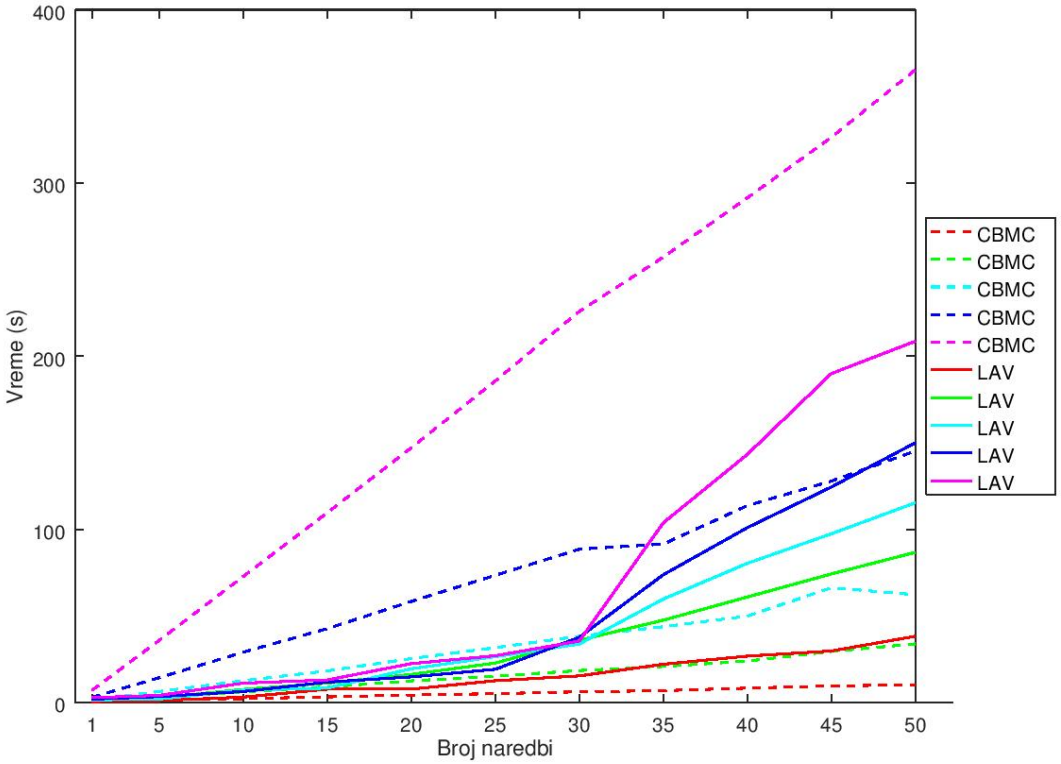
Табела 5.2: Експериментални резултати алата ЛАВ и CBMC на првом корпусу

CBMC користи стандардне секвенцијалне технике испитивања исправности програма. Са повећањем броја наредби, повећавају се СМТ формуле које описују програм и чију задовољивост је потребно испитати, због чега се укупно време потребно за верификацију драстично увећава. Приметимо да је време потребно а верификацију програма који садржи 26 наредби користећи алат CBMC више него двоструко веће од времена које је потребно за верификацију програма од 25 наредби. Док је разлика тих времена користећи алат LAV мала. Овакви резултати показују да паралелним испитивањем исправности наредби програма можемо драстично смањити укупно време које је потребно за верификацију програма и на тај начин унапредити постојеће алате.

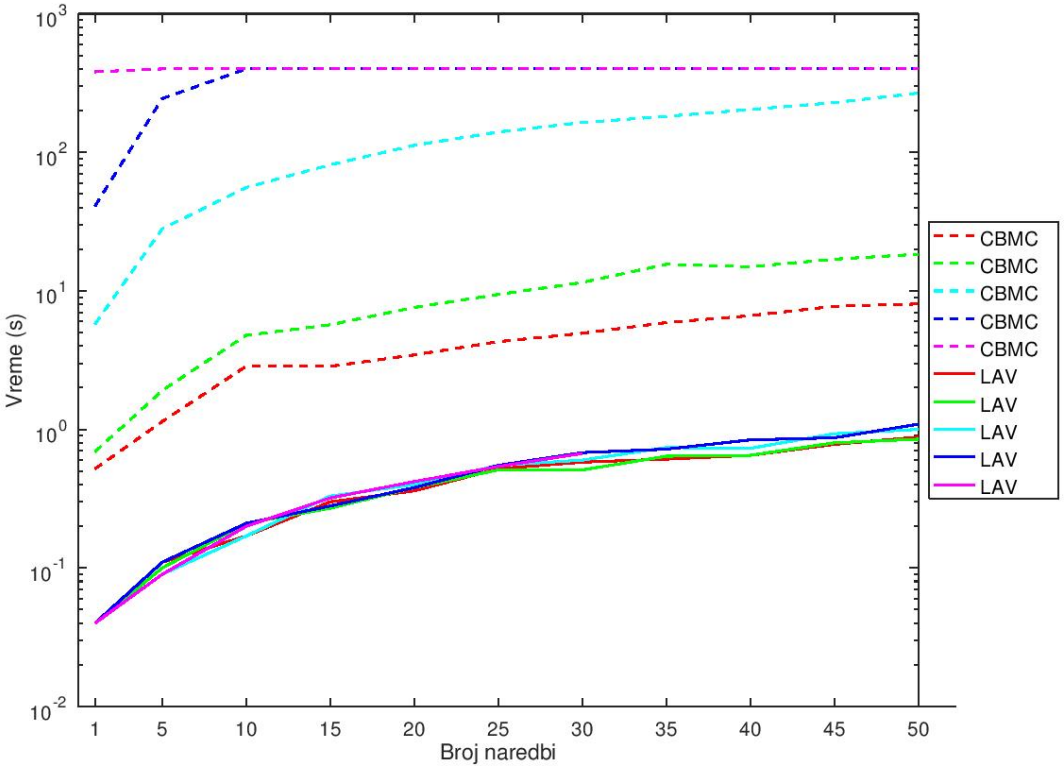
Резултати верификације програма из другог корпуса се налазе на графицима 5.1, 5.2, 5.3 и 5.4.



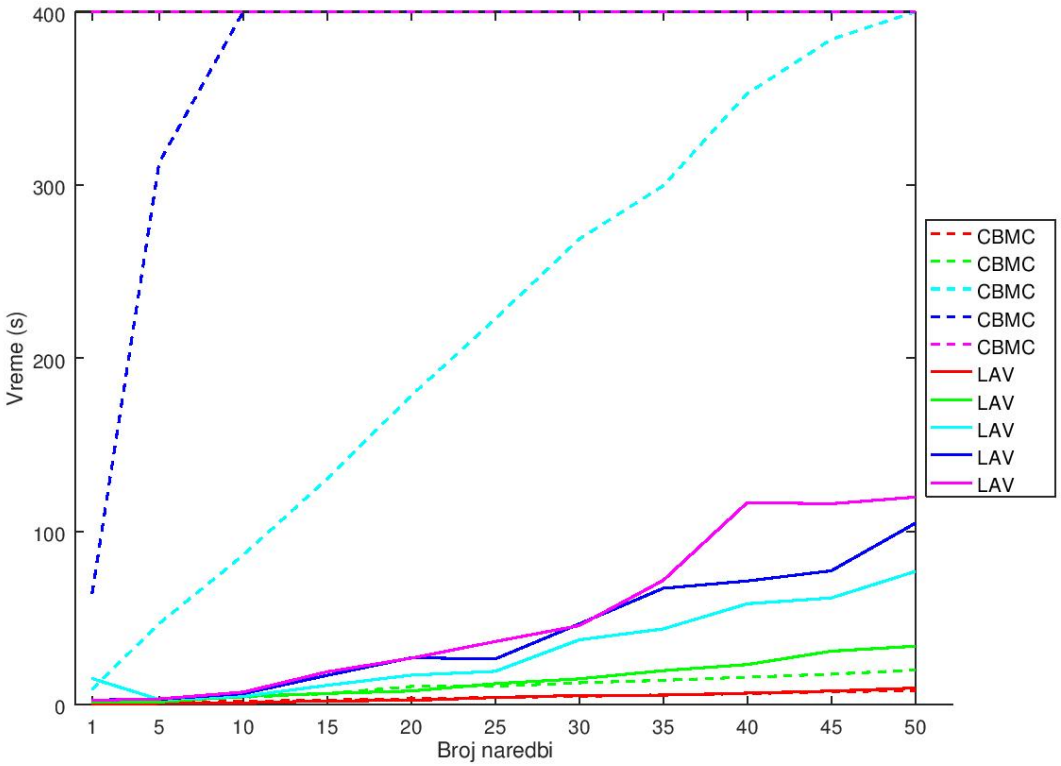
Слика 5.1: Први ниво



Слика 5.2: Први ниво T



Слика 5.3: Други ниво



Слика 5.4: Други ниво T

Глава 6

Закључак

Библиографија

[1] Ankita Bhalla. Various ways of parallelization of sequential programs. *International Journal of Engineering Research and Technology*, 3, 1 2014.

[2] Clay Breshears. *The Art of Concurrency: A Thread Monkey's Guide to Writing Parallel Applications*. O'Reilly Media, Inc., 2009.

[3] Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*. USENIX Association.

[4] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In Kurt Jensen and Andreas Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 2004.

[5] Vijay D'Silva, Daniel Kroening, and Georg Weissenbacher. A survey of automated techniques for formal software verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 2008.

[6] Julien Henry, David Monniaux, and Matthieu Moy. Pagai: A path sensitive static analyser. *Electron. Notes Theor. Comput. Sci.*, 289, 2012.

[7] Milena Vujošević Janičić. *Automatsko generisanje i proveravanje uslova ispravnosti programa*. PhD thesis, Matematički fakultet, Univerzitet u Beogradu, 2013.

[8] Cem Kaner, Jack L. Falk, and Hung Quoc Nguyen. *Testing Computer Software, Second Edition*. John Wiley & Sons, Inc., New York, NY, USA, 2nd edition, 1999.

[9] James C. King. Symbolic execution and program testing. *Commun. ACM*, 1976.

[10] C++ language reference. http://en.cppreference.com/w/.

[11] Miroslav Marić. *Operativni sistemi*. Univerzitet u Beogradu - Matematički fakultet, 2015.

[12] Florian Merz, Stephan Falke, and Carsten Sinz. Llbmc: Bounded model checking of c and c++ programs using a compiler ir. In *Proceedings of the 4th International Conference on Verified Software: Theories, Tools, Experiments*. Springer-Verlag.

[13] Cristóbal A. Navarro, Nancy Hitschfeld-Kahler, and Luis Mateu. A survey on parallel computing and its applications in data-parallel problems using gpu architectures. *Communications in Computational Physics*, 15:285–329, 2 2014.

[14] Sartaj Sahni and Venkat Thanvantri. Parallel computing: Performance metrics and models. 1995.

[15] Michael L. Scott. *Programming Language Pragmatics, Third Edition*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 3rd edition, 2009.

[16] Alan M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 1936.

[17] Kuncak Viktor Vujosevic Janicic Milena M. Development and evaluation of lav: An smt-based error finding platform system description. *VERIFIED SOFTWARE: THEORIES, TOOLS, EXPERIMENTS*, 2012.

[18] Gregory V. Wilson. The history of the development of parallel computing (https://webdocs.cs.ualberta.ca/ paullu/c681/parallel.timeline.html).