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

Мотивација

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 одговарајући излаз предвиђен 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 велики, често и 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.

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 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.

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 реалног извршавања.

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 пратећи дату путању.

2.3 Алати за верификацију

2.4 Систем ЛАВ

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 [4].

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 инструкција које се извршавају након његовог покретања. Секвенцијалне програме одликује серијско извршавање инструкција. 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 [7].

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]:

$$S = T_s/T_p$$

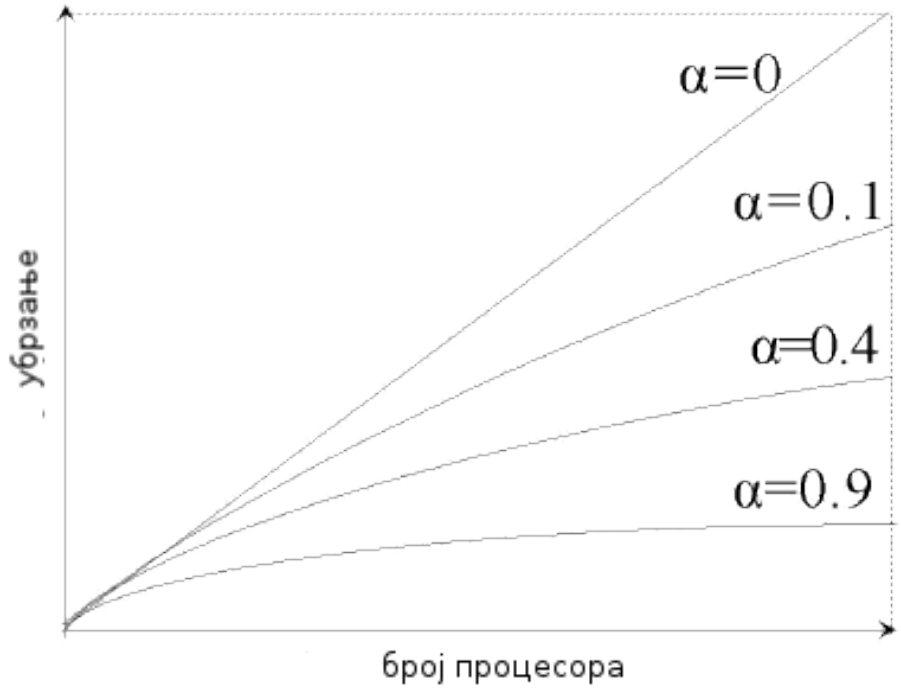
где T_s представља време извршавања секвенцијаног а T_p време извршавања паралелног програма за исти проблем.

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

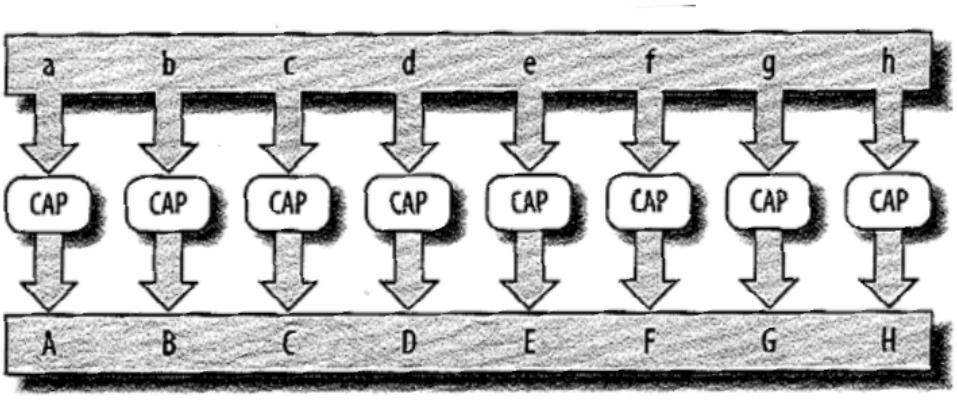
У пракси, време извршавања програма на паралелним системима је углавном веће од теоријски израчунате вредности јер зависи и од других параметара попут комуникације и синхронизације. Амдалов модел не узима у обзир ова времена и разматра само случајеве у којима је димензија проблема фиксирана. Поред Амдаловог модела постоје и други модели као што су Густафсонов, Гинтеров, модел Сун Ни-ја који превазилазе нека ограничења Амдаловог модела [5].

3.2 Врсте паралелизације

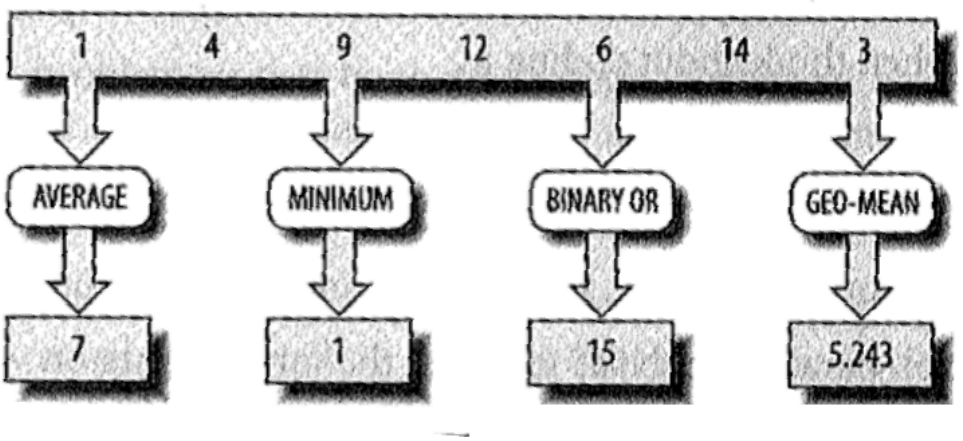
Програми се могу паралелизовати на различите начине. Паралилзацију може обављати програмер експлицитно или коришћењем неких алата. Последњих година развијени су многи алати који омогућују аутоматску паралелизацију. Коришћењем ових алата програмеру је олакшан процес паралелизације уз ограничену контролу. 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 инструкција 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 *сарадња* 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. [6]

Синхронизација отвара (нека друга реч ?) 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 завршетка рада у неком тренутку, односно да константно прави прогрес током свог извршавања. Уколико ово својство није задовољено може да се деси да програм никада не заврши свој рад. 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. deadlock) 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. [3]

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 такође 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 Ц, Ц++ 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 Ц++ 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 Ц++ језику. Као такав користи многе екстерне (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 Ц++. У наставку текста ће бити описана архитекура и имплементација модула за паралелизацију као и начини његовог коришћења у оквиру система ЛАВ.

4.1 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. Композицијом ових делова имплементиран је модул који паралелизује неке делове алгоритма анализе програмског кода у овиру система ЛАВ.

4.2 Имплементација модула

За имплементацију модула коришћен је језик Ц++ због компатибилности са системом ЛАВ. Приликом имплементације појединачних нити коришћен је модел нити из библиотеке `<thread>` језика Ц++. Ова библиотека је јавно доступна и нуди интерфејс (енг. application programming interface, API) за конструкцију и управљање нитима.

Класа `ThreadPool` представља контролни део модула, тзв. базен нити. Она садржи контролну нит чија је улога контрола радних нити које се такође налазе у оквиру ове класе (`ThreadPool`). Посао радних нити је да извршавају задатке који су прослеђени базену нити. Због тога се унутар базена нити чува и ред задатака (класа `FixedQueue`) које је потребно извршити. Класа `FixedQueue` је шаблонска класа и може садржати ред објеката било ког типа. Задаци који се смештају у ред су објекати Ц++ 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.

Задатке извршавају радне нити, објекти класе `std::thread`, које су обмотане класом `SignalingThread`. Класа `std::thread` (која постоји у оквиру Ц++ библиотеке `<thread>`) нуди многобројне корисне функције за рад са нитима и као што су нпр. `join, detach, joinanble,...` Базен нити садржи објекат реда `FixedQueue` као и низ радних нити. Могуће је задати број нити али уколико другачије није наглашено конструисаће се онолико радних нити колико оперативни систем дозвољава.

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 са реда један по један задатак и извршавају га. Како све нити приступају истом објекту реда, потребно је синхронизовати процес скидања задатака. Ц++ језик, у оквиру библиотеке `<atomic>` нуди разне типове над којим су подржане атомичне операције. Све операције над овим типовима су безбедне у контексту вишенитног окружења. Класа `FixedQueue` садржи јединствен показивач на објекат атомичног типа (`std::unique_ptr<std::atomic_uint>`). Овај објекат садржи цео број и чува информацију о томе колико је задатака скинуто са реда (индекс следећег задатка који треба скинути). У тренутку када радна нит затражи задатак из реда позива се функција `fetch_add` над

овим објектом. Функција `fetch_add` враћа вредност објекта (цео број) и након тога увећава његову вредност, обезбеђујући атомичност ових операције. 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).

Након што изврши задатак, нит обавештава базен нити емитовањем догађаја, објекат класе `Event`. Свака нит садржи посебан објекат догађаја тако да се базен нити претплаћује на ослшкивање догађаја сваке нити посебно. Резултат извршавања задатака нити обрађују и шаљу сигнал базену нити емитовањем одговарајућег догађаја. Контролна нит која се налази у базену нити тумачи сигнал и предузима потребне акције.

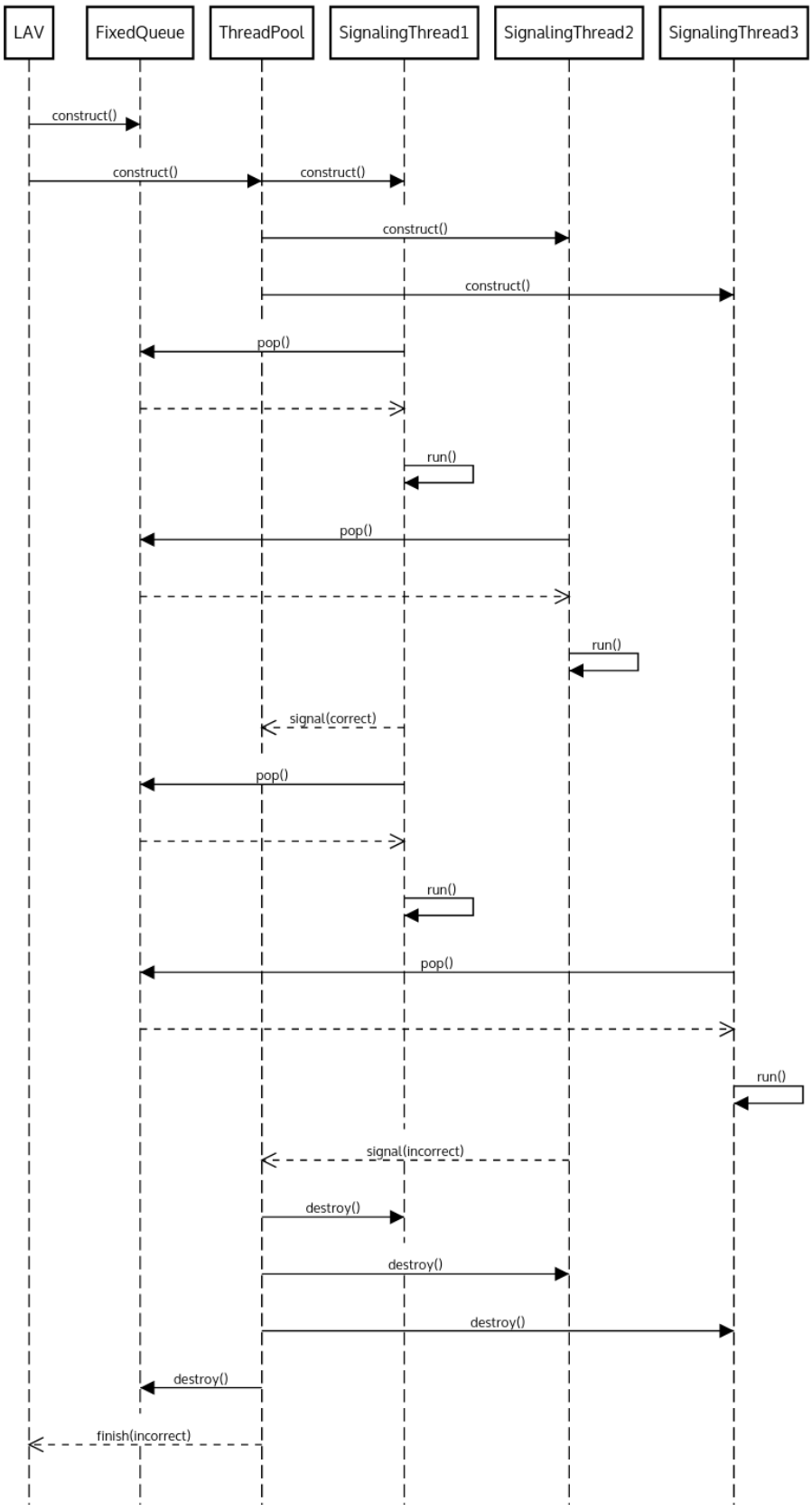
4.3 Интеграција модула са системом ЛАВ

Паралелизација је имплементирана у контексту анализе блока програмског кода. 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 анонимна функција која позива SMT решавач. Анонимна 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 анонимну 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 анонимна 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` базен нити у таквој ситуацији зауставља све нити. У супротном нити настављају са радом. Уколико су све функције из реда извр-

шене и сви услови су били задовољени, блок се сматра исправним и тако бива означен.

На слици 4.1 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.



Слика 4.1: Дијаграм тока извршавања

Глава 5

Експериментални резултати

5.1 Архитектура рачунара

5.2 Опис корпуса

5.3 Начини покретања

5.4 Измерена времена

5.5 Објашњавање експерименталних резултата

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

[4] 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.

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

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

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