УНИВЕРЗИТЕТ У НОВОМ САДУ

ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА

Владимир Маринковић

Прилог аутоматској паралелизацији секвенцијалног машинског кода

– ДОКТОРСКА ДИСЕРТАЦИЈА –

Ментор:

проф. др Мирослав Поповић

Нови Сад, 2018

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

КЉУЧНА ДОКУМЕНТАЦИЈСКА ИНФОРМАЦИЈА

Редни број, **РБР**:		
Идентификациони број, **ИБР**:		
Тип документације, **ТД**:		Монографска документација
Тип записа, **ТЗ**:		Текстуални штампани материјал
Врста рада, **ВР**:		Докторски рад
Аутор, **АУ**:		Владимир Маринковић
Ментор, **МН**:		Проф. др Мирослав Поповић
Наслов рада, **НР**:		Прилог аутоматској паралелизацији секвенцијалног машинског кода
Језик публикације, **ЈП**:		Српски
Језик извода, **ЈИ**:		Српски
Земља публиковања, **ЗП**:		Република Србија
Уже географско подручје, **УГП**:		Војводина
Година, **ГО**:		2018.
Издавач, **ИЗ**:		Ауторски репринт
Место и адреса, **МА**:		Нови Сад; Трг Доситеја Обрадовића 6
Физички опис рада, **ФО**: (поглавља/страна/ цитата/табела/слика/графика/прилога)		6 поглавља / 71 страна / 43 цитата / 3 табеле / 20 слика
Научна област, **НО**:		Електротехничко и рачунарско инжењерство
Научна дисциплина, **НД**:		Рачунарска техника и рачунарске комуникације
Предметна одредница/Кључне речи, **ПО**:		Паралелне архитектуре, паралелно програмирање, вишејезгарна обрада, асемблер, распоређивање процеса, наменски системи, програмски алати
УДК		
Чува се, **ЧУ**:		У библиотеци Факултета техничких наука, Нови Сад
Важна напомена, **ВН**:		
Извод, **ИЗ**:		Докторска теза анализира подршку за вишејезгарне и многојезгарне системе у циљу повећања искоришћења њихове снаге. Предмет истраживања је проналажење решења које би без уплитања програмера (аутоматски) паралелизовало постојеће секвенцијалне програме на бинарном нивоу који се извршавају на једном језгру (или процесору). Резултат истраживања је израда решења и алата за паралелизацију секвенцијалног машинког кода, који самостално стварају програме који се извршавају паралелно на више језгара вишејезгарног процесора, и тиме постижу уравнотежено оптерећење процесора. Основни циљ је добијање убрзања извршења програмског кода на вишејезгарном процесору ради омогућавања рада у реланом времену за задата ограничења. Добијено решење би се могло искористити и за смањење потрошње смањивањем радног такта процесора уз задржавање полазног времена извршења програма.
Датум прихватања теме, **ДП**:		22.02.2018.
Датум одбране, **ДО**:		
Чланови комисије, **КО**:	Председник:	Др Никола Теслић, редовни професор
	Члан:	Др Илија Башичевић, ванредни професор
	Члан:	Др Миодраг Ђукић, доцент
	Члан:	Др Мило Томашевић, редовни професор
	Члан, ментор:	Др Мирослав Поповић, редовни професор

Потпис ментора

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

KEY WORDS DOCUMENTATION

Accession number, **ANO**:	
Identification number, **INO**:	
Document type, **DT**:	Monographic publication
Type of record, **TR**:	Textual printed material
Contents code, **CC**:	PhD Thesis
Author, **AU**:	Vladimir Marinkovic
Mentor, **MN**:	Prof. Miroslav Popovic, PhD
Title, **TI**:	An approach to automatic parallelization of sequential machine code
Language of text, **LT**:	Serbian
Language of abstract, **LA**:	Serbian
Country of publication, **CP**:	Republic of Serbia
Locality of publication, **LP**:	Vojvodina
Publication year, **PY**:	2018.
Publisher, **PB**:	Author's reprint
Publication place, **PP**:	Novi Sad, Dositeja Obradovica sq. 6
Physical description, **PD**: (chapters/pages/ref./tables/pictures/graphs/appendixes)	6 chapters / 71 pages / 43 references / 3 tables / 20 pictures
Scientific field, **SF**:	Electrical and Computer Engineering
Scientific discipline, **SD**:	Computer engineering and communications
Subject/Key words, **S/KW**:	Parallel architectures, parallel programming, multicore processing, assembly, processor scheduling, embedded system, software tools
UC	
Holding data, **HD**:	The Library of Faculty of Technical Sciences, Novi Sad, Serbia
Note, **N**:	
Abstract, **AB**:	PhD thesis analyzes a support for multicore and manycore systems in terms of better processing power utilization. Purpose of this study is finding a solution for automatic parallelization of existing sequential code which executes on single core (or processor), at the binary level. The research intents to develop a solution and tools for parallelization of the sequential machine code, which can create a program running simultaneously on all the cores of the multi-core processor, and for achieving optimal load-balancing. The primary goal is obtaining execution speedup of the program running on the multicore processor, for meeting real-time processing constraints. Given solution could be also used for energy saving, by lowering system clock and keeping program execution runtime.
Accepted by the Scientific Board on, **ASB**:	22.02.2018.
Defended on, **DE**:	

Defended Board, **DB**:	President:	prof. Nikola Teslić, PhD	
	Member:	associated prof. Ilija Bašičević, PhD	
	Member:	assistant prof. Miodrag Djukic, PhD	Mentor's sign
	Member:	prof. Milo Tomasevic, PhD	
	Member, Mentor:	prof. Miroslav Popovic, PhD	

Желим да се захвалим свом ментору, проф. др Мирославу Поповићу на искреној помоћи, подршци и свакодневном подстреку у току истраживања, писања научних радова и, коначно, у реализацији ове дисертације као круне заједничког научно-истраживачког рада.

Велику захвалност дугујем и колегама доц. др Миодрагу Ђукићу, доц. др Ивану Каштелану и проф. др Небојши Пјевалици од којих сам много ствари научио, као и колегама Бранislаву Кордићу и Дејану Бокану, са којима сам нове ствари заједнички учио и на корисним сугестијама и саветима који су допринели мом истраживачком раду на докторским студијама.

Највећу захвалност дугујем својој породици која ме је бодрила и омогућила да истрајем у току докторских студија.

САЖЕТАК

Појава вишејезгарних и новијих многојезгарних система је понудила могућност за повећање брзине извршавања програма, односно смањење потрошње енергије уз задржавање постојећег времена извршења. Стога, настали системи чине паралелно извршење програма есенцијалним. Паралелизација програма се може урадити ручно, аутоматски дириговано од стране програмера или потпуно аутоматски.

Главна тема ове докторске дисертације је аутоматска паралелизација кода и конкретно, развој алата за паралелизацију на бинарном нивоу, као и валидација овог решења. Развијен је нов алгоритам за паралелизацију асемблерског кода на нивоу инструкција који, након SSA анализе, користи имена регистара за откривање независних блокова кода и распоређује независне блокове по језгрима коришћењем METIS алгоритама, како би постигао равномерно оптерећење.

Конзистентност са секвенцијалним програмом је верификована и валидација је урађена мерењем времена извршења програма на циљној архитектури. Развијен алат за паралелизацију је омогућио велико убрзање (које је узето као основна мера перформансе у процесу валидације) и оптимално оптерећење по језгрима за вишејезгарне RISC процесоре са 2 до 16 језгара (нпр. MIPS, MicroBlaze, итд.). Конкретно, за 16-језгарни процесор, просечно убрзање за извршене тестове износи 7.92x, док у неким случајевима достиже и 14x.

Прилог аутоматској паралелизацији из ове докторске дисертације може бити користан истраживачима и инжењерима из области паралелизације, као основа за даље оптимизације, као задњи део преводиоца или као алат за паралелизацију кода за наменски систем попут циљног система коришћеног у истраживању.

ABSTRACT

The appearance of multicore and novel manycore systems has provided opportunities to increase the speed of program execution or to save energy keeping the old program execution time. Hence, emerging multicores and even manycores make a parallel program execution essential. Parallelization could be done manually, automatically programmer directed, and fully automatically.

As the main topic of this PhD thesis, automatic code parallelization is considered, specifically developing a parallelization tool at the binary level as well as the validation of this approach. The novel instruction-level parallelization algorithm for assembly code which uses the register names after SSA to find independent blocks of code and then to schedule independent blocks using METIS to achieve good load balance is developed.

The sequential consistency is verified and the validation is done by measuring the program execution time on the target architecture. The developed parallelization tool have enabled great speedup (which is taken as the performance measure in the validation process) and optimal load balance on processor cores for multicore RISC processors with 2 to 16 cores (e.g. MIPS, MicroBlaze, etc.). In particular, for 16 cores, the average speedup is 7.92x, while in some cases it reaches 14x.

An approach to automatic parallelization provided by this PhD thesis is useful to researchers and developers in the area of parallelization as the basis for further optimizations, as the back-end of some compiler, or as the code parallelization tool for an embedded system like the targeted one.

САДРЖАЈ

Списак слика

Списак табела

Скраћенице

CCC	Cirrus C преводилац (Cirrus C Compiler)
CFG	Граф тока управљања (Control Flow Graph)
DCT	Дискретна косинусна трансформација (Discrete Cosine Transform)
DDG	Граф зависности података (Data Dependecy Graph)
DSP	Дигитална обрада сигнала (Digital Signal Processing)
FFT	Брза Фуријеова трансформација (Fast Fourier Transform)
FPGA	Програмабилна секвенцијална мрежа (Field-Programmable Gate Array)
ILP	Паралелизам на нивоу инструкција (Instruction-Level Parallelism)
IR	Међукод или међурепрезентација (Intermediate Representation)
LMB	Локална меморијска магистрала (Local Memory Bus)
LUT	Таблица истинистости унутар FPGA (Look Up table)
RISC	Рачунар са смањеним скупом инструкција (Reduced Instruction Set Computer)
SMP	Физичка архитектура са заједничком меморијом (Symmetric Multi Processor)
SSA	Јединствена додела вредности променљивој (Static Single Assignment)
TBB	Intel Threading Building Blocks
TLP	Паралелизам на нивоу нити (Thread-Level Parallelism)

ПОГЛАВЉЕ 1.

Увод

У данашње време сви сегменти рачунарског тржишта, од моћних сервера до малих потрошачких уређаја, укључују вишепроцесорске системе. Појава ових система је донела могућност за повећање брзине извршења програма, односно уштеду енергије уз задржавање полазног времена извршења. Стога, новонастали вишејезгарни и многојезгарни системи чине паралелно извршење програма круцијалним. С друге стране, највећи део наменских програма је и даље написан и пише се секвенцијално, па подршка за поменуте системе, уз овакву праксу, захтева паралелизацију секвенцијалних програма. Паралелизација секвенцијалних програма се може урадити: (1) ручно - пребацујући се на модел нити, (2) аутоматски дириговано од стране програмера, и (3) потпуно аутоматски што је још увек отворени проблем.

Ручно натерати програм да се извршава на више језгара истовремено је компликован, подложан грешкама, напоран, временски захтеван и итеративан процес. Подразумева се пребацивање на модел нити (нпр. POSIX Threads), што захтева много измена у изворном коду, посебно у случају да секвенцијалан програм већ постоји. Ово решење је прикладније у случају да се паралелни програм пише од самог почетка. И тада програмер мора да размишља паралелно и пише програм на тај начин, што доноси много мању продуктивност по линији написаног кода у поређењу са серијским програмирањем [1]. Са друге стране, ова метода може дати најбоље резултате по питању добијеног убрзања.

Аутоматска метода, али диригована од стране програмера најчешће имплицира унос само малих измена у код (нпр. додавање директива или заставица преводиоца, коришћење постојећих шаблона, итд). На жалост, ово често није једностан задатак. Ипак, ово је једноставнија метода за паралелизацију постојећег кода у односу на ручну методу, али најчешће доноси лошије резултате по питању убрзања. Најистакнутији приступи у овој групи су: Intel Threading Building Blocks (TBB) [2] [3] [4], Intel Cilk Plus [2] [5] [6], Open Multi-Processing (OpenMP) [7], Open Computing Language (OpenCL) [8], итд. Сви ови приступи доносе задовољавајуће убрзање у одређеним случајевима. Неке од ових библиотека анализира [9]. Аутори су показали да TBB библиотека може на режију да утроши и до 47% од укупног времена извршења на процесорском систему са 32 језгра, док OpenMP даје и лошије резултате, услед закључавања брава. Једноставна паралелизација је могућа коришћењем постојећих шаблона, али само у специфичним апликацијама. У осталим, општим случајевима, паралелизација је далеко компликованија. Програмер мора да препозна место погодно за паралелизацију и да је примени нетривијалним изменама које уноси у код.

У овом раду ће бити описана аутоматска паралелизација кода и, конкретније, развој алата за паралелизацију секвенцијалног кода на бинарном (машинском) нивоу који производи паралелизован програм. Уједно, рад ће обухватати и валидацију решења. Поред аутоматске паралелизације, есенцијална могућност оваквог алата се огледа у уравнотежењу оптерећења свих језгара доступних на циљном процесору.

Још један проблем у паралелизацији је валидација исте. Разматрају се времена на симулационом моделу платформе и на самој физичкој архитектури. Сличан приступ са симулационим моделом физичке вишепроцесорске архитектуре се користи у [10]. Убрзање, дефинисано као однос времена извршења секвенцијалног и паралелизованог програма, користи се као мера перформансе у процесу валидације. Такође дефинисан је модел перформансе који се користи за процену времена извршења програма, а биће верификован поређењем процењених времена са временима измереним на стварној платформи. Оптимално оптерећење свих језгара омогућава значајно убрзање за

опште апликације, и најважније, без измена изворног кода секвенцијалног програма.

Основни циљ је добијање убрзања извршења ради омогућавања рада у реланом времену за задата ограничења. Добијено решење би се могло искористити и за смањење потрошње смањивањем радног такта процесора уз задржавање полазне брзине што би било омогућено бољим искоришћењем процесорске снаге, али детаљније разматрање те идеје излази ван оквира овог истраживања.

Хипотеза овог рада је да је могуће пронаћи решење односно направити алат који би без уплитања програмера (аутоматски) паралелизовао постојеће секвенцијалне апликације које се извршавају на једном језгру (и једном процесору) тако да се паралелно извршавају на више језгара вишејезгарног процесора, смањујући време извршења бољим искоришћењем, тј. уравнотеженим оптерећењем процесора без измена његових перформанси.

У почетним плановима сматрало се да ће идеја имати вишеструку примену, уколико би истраживање постигло задовољавајуће резултате по питању убрзања и скалабилности кода који се добија паралелизацијом секвенцијалног кода. Приступ аутоматској паралелизацији који ће бити описан је користан другим истраживачима и инжењерима из области паралелизације као основа за даље оптимизације, као задњи део неког преводиоца или као комплетна метода (или алат) за паралелизацију кода у наменским системима као што је циљни систем у овом раду.

Област примене разматраног паралелизатора је спектар вишејезгарних наменсих палтформи примарно намењених за апликације са тврдим временским ограничењима (енг. hard real-time). За одабир конкретне циљне архитектуре се морају укључити још нека разматрања. Према [11], за разлику од секвенцијалног програмирања, паралелно програмирање никада није успешно коришћено за општенаменске програмске моделе. Главни разлог је кохереницја скривене (енг. cache) меморије. Присуство скривене или виртуелне меморије детерминистичке системе чини недетерминистичким и значајно умањује време извршења у најгорем случају, и последично, није прихватљиво за системе са тврдим временским ограничењима. Даље, аутори рада [11] тврде да пралелно

програмирање мора да користи моделе и архитектуре специфичне за конкретну примену, са локалним паралелним меморијама и без скривених меморија, и то не само за системе у реалном времену већ и за било коју конкретну примену. Водећи се тим ставом, паралелизатор има за циљ примену на симетричним вишејезгарним наменским платформама без скривене меморије.

Детаљи су изложени у наредних пет поглавља. У Поглављу 2. је дат преглед постојећих модела паралелизације секвенцијалног кода, предности и мане појединих модела, са посебним освртом на модел аутоматске паралелизације. Такође, за сваки од модела дат је актуелни преглед истраживања и постојећих решења. Описана је област примене, у смислу домена апликација које се паралелизују и могућих циљних архитектура. На крају су наведени и конкретне циљне архитектуре, али и алати за које се описани паралелизатор жели направити као независни додатак. Поглавље 3. даје прецизан опис циљних физичких архитектура за које је паралелизатор направљен, односно које су коришћене као наменске платформе у току истраживања, у циљу валидације и верификације решења. У Поглављу 4. је дат опис паралелизатора машинског секвенцијалног кода, као проблема који се решава, те је ово поглавље од суштинске важности. Детаљно је приказана архитектура паралелизатора и сваки њен појединачан блок. Поглавље 5 нуди експерименталану валидацију у симулатору на једном примеру примене, а затим и верификацију на стварној циљној платформи, уз постигнуте резултате, при чему је као тест вектор коришћено неколико апликација типичних за наменске системе. У Поглављу 6. су дати закључци, детаљнији преглед области примене решења имајући у виду добијене резултате, као и могући правци будућег истраживања.

ПОГЛАВЉЕ 2.

СТАЊЕ У ОБЛАСТИ

Потпуно аутоматска метода паралелизације је и даље отворен проблем. Идеја је да се омогући потпуно аутоматска паралелизација секвенцијалног кода, без потребе за било каквим променама изворног кода. Добијање мало мањег убрзања у односу на остале методе је нешто што ова метода може имати за последицу, али сигурно прихватљиву када се узме у обзир корист од аутоматизације процеса.

Колико је аутору познато, постоје само неколико метода које се баве аутоматском паралелизацијом на бинарном нивоу. Штавише, не постоји алат за аутоматску паралелизацију ирегуларних програма на бинарном нивоу. Већина постојећих алата покрива статичку паралелизацију независних петљи или оних са *петљом-ношеним зависностима* или динамичку паралелизацију петљи и рекурзивних структура. У [12] је представљена метода аутоматске паралелизације унутар бинарног преписивача, иза које је идеја најсличнија идеји из овог рада. За разлику од ове методе, аутор поменутог рада се бавио само паралелизацијом петљи те и претпоставља да ће већина програма за које је метода намењена у основи имати петљу, док је циљ овог рада паралелизација свих програма на нивоу инструкција. У том смислу се идеја која ће овде бити презентована може сматрати генерализацијом поменутог рада.

У овој докторској дисертацији је описано решење аутоматске паралелизације на бинарном (машинском) нивоу.

Према [12], паралелизација на бинарном нивоу има неколико предности у односу на паралелизацију на међукоду (унутар преводиоца), поред већ поменутих: (1) може се користи за све програмске језике и преводиоце; (2) није потребна замена софтверских алата за превођење; (3) велика портабилност с обзиром да се алат направљен за једну архитектуру може користити уз било који преводилац намењен истој архитектури; (4) није потребан изворни код да би се бинарни код паралелизовао; (5) може се користити на асемблерском коду директно; и (6) осим програмера, могу је корисити и крајни корисници.

У наставку је дат преглед различитих решења, предлога решења и метода, подељен у четири групе, према повезаности са конкретном темом која је овде разматрана, односно која представља део методе која се предлаже: (1) Методе статичке аутоматске паралелизације, (2) Методе динамичке аутоматске паралелизације, (3) Методе паралелизације које су намењене прототиповима процесора који су направљени у FPGA и (4) Оптимизације засноване на партиционисању графова. Додатно, преглед стања се прожима кроз цео рад, на примереним местима, у виду аргументовања појединих тврдњи.

2.1 Методе статичке аутоматске паралелизације

Значајан напредак у аутоматској паралелизацији је направљен у [13], где су аутори предложили 2 алгоритма као главна дела паралелизатора секвенцијалног C кода који је намењен модерном DSP процесору, Cirrus Logic Coyote 32. Користе предњи део преводиоца Cirrus C Compiler (CCC) како би полазни код свели на међукод, који представља улаз у блок за паралелизацију. Блок за паралелизацију прави N партиција (кодова) још увек представљених у форми међукода, а оне се потом прослеђују задњем делу преводиоца CCC који потом прави N извршних датотека. Аутори су блок за паралелизацију направили помоћу: (1) блока за одређивање животног века променљивих, (2) блока за SSA (енг. Static Single Assignment) анализу и (3) алгоритма за паралелизацију. Анализа животног века променљивих и SSA анализа уводе нове променљиве и тиме умањују зависности података. Понуђена су два алгоритма за

паралелизацију и оба, на основу Бернштајнових критеријума, издвајају независне целине у коду и мапирају их на језгра процесора. Оба алгоритма подразумевају и прорачун цене инструкције (нпр. регистарским инструкцијама је потребно мање инструкцијских циклуса него инструкцијама које приступају меморији). Тај приступ је показао добре резултате и убрзање од чак 7.96x за 8-језгарни процесор у примерима са малом зависношћу података и то су показали на симулационом моделу. Ипак, све претпоставке које су у раду изнете остале су недоказане на циљној платформи. Штавише, аутори су направили неколико чврстих ограничења попут непостојања скокова, петљи, анализе алијаса и других ограничења која решење чине практично непримењивим за већину C програма. Стога, решење није довољно заокружено, иако показује добар потенцијал и самим тим представља добру основу за целокупно решење аутоматског паралелизатора који постиже висока убрзања.

Асемблерски код се већ показао као погодан за паралелизацију [14] уз добијено скоро линеарно убрзање за неке тест векторе. Аутори су, користећи gcc преводилац, добијали асемблерски код од полазног C кода, трансформисали га у SSA форму, те на основу партиционисања графа зависности података (енг. Data Dependency Graph - DDG), METIS алгоритмом, коначно створили партиције за које су пре самог генерисања паралелног кода радили доделу регистара. Постигли су скоро линеарно убрзање за одређење примере али уз значајна ограничења. Паралелизатор ради само са кодом који не садржи гранања или петље, а значајно ограничење представља и колизија код приступа меморији јер адресе меморијских локација нису узимане у разматрање.

Неколико покушаја по питању аутоматске паралелизације и оптимизације преводилаца побољшањем анализе података са нагласком на оптимизацију петљи су направљени у [15] и [16]. Аутори су покушали да пронађу нове технике анализе зависности података узимајући у обзир нелинеарне изразе. На тај начин су успели да обраде и веома сложене случајеве и повећају паралелизам програма. Ограничења зависности у изворном коду са нелинеарним и симболичким изразима, комплексним границама бројача у петљама, вишедимензионалним низовима код којих се иста променљива користи за индексирање по свакој димензији и са условним исказима, показана су низом техника са

полиномијалним временом извршења, а помажу преводиоцу у даљим трансформацијама. Показали су да је алат за анализу зависности тачан, ефикасан и делотворнији у паралелизацији програма од других тестова зависности. Последично, резултати паралелизације су побољшани и дају већа убрзања на неким тестовима. У поређењу са тим радом, решење које се предлаже у овој дисертацији подржава било који тип петљи, паралелизацијом тела петље (тзв. DOACROSS паралелизам), што је идеално за случај петљи са *петљом-ношеним зависностима* (енг. Loop-carried dependencies), док су слична анализа независних петљи и паралелизација дељењем итерационог простора (тзв. DOALL паралелизам) планиране као следећа унапређења предложеног решења.

Аутори [17] су понудили DSWP решење које користи финији паралелизам проточне структуре, којим успешно решава проблем нити које се дуготрајно и конкурентно извршавају. Њихова потпуно аутоматска имплементација преводиоца показује значајна побољшања, али су она потврђена само на симулационом моделу двојезгарног процесора. DSWP решење користи паралелизацију на нивоу нити (енг. Thread-Level Parallelism – TLP) и конкретно паралелизацију тела петљи (DOACROSS паралелизам). За разлику од тога, у решењу које описује ова дисертација користи се паралелизам на нивоу инструкција (енг. Instruction-Level Parallelism - ILP), што укључује и поменути DOACROSS паралелизам петљи. Штавише, решење које се овде предлаже је скалабилно и верификовано на стварној циљној архитектури.

Аутори [18] су описали и дали оцену за решење под називом HELIX, технику за потпуно аутоматску паралелизацију петљи која узастопне итерације петље додељује одвојеним нитима. Интересантно, аутори ублажавају зависности међу итерацијама код петљи са *петљом-ношеним зависностима* оптимизацијом кода, тако што уводе помоћне нити које унапред уносе сигнале за синхронизацију. HELIX постиже значајна убрзања коришћењем паралелизације на нивоу нити и паралелизацију петљи дељењем итерационог простора за петље већ оптимизоване на нивоу изворног кода, уз мању неефикасност насталу услед додатног сигнализирања и уведене комуникације између нити. Решење које се предлаже у овој докторској дисертацији користи паралелизам на нивоу

инструкција на машинском коду и DOACROSS паралелизам петљи без додатне комуникације између нити.

Аутори [19] су направили спој NLVI-тест идеја и анализе ланаца рекурентности. Кроз предности које донесе обе методе, појачали су тестове зависности података и одређивања нивоа паралелизма како би могли да анализирају комплексне нелинеарне изразе и петље. Резултати показују да нове технике откривају већи број паралелних петљи односно већи ниво паралелизма.

У раду [20] направљен је значајан напредак ка аутоматској паралелизацији секвенцијалног C кода. Аутори овог рада су описали алат под називом Cetus који представља инфраструктуру преводиоца, који за циљ има C програме, а паралелизацију изводи трансформацијама над самим изворним кодом. За разлику од њиховог решења, ово решење за циљ има паралелизацију асемблерског кода.

Аутори [21] су понудили алат који аутоматски додаје OpenMP директиве како би олакшао паралелну обраду на ширем спектру вишепроцесорских физичких архитектура са заједничком меморијом (енг. Symmetric Multi Processor - SMP). Поменути алат статички дели секвенцијални C програм на крупније задатке, анализира зависности међу задацима и формира OpenMP паралелни код. Аутори користе паралелизацију на нивоу нити (TLP) уз статичке анализе, трансформације над изворним кодом и формирају нови код како би побољшали перформансе преко граница паралелизације самих петљи, што су и верификовали на x86 машини са два тестна програма. Слично, у овом решењу је циљна архитектура SMP типа, уз методу статичке аутоматске паралелизације, али на машинском нивоу, коришћењем паралелизације веће резолуције на нивоу инструкција (ILP).

2.2 Методе динамичке аутоматске паралелизације

У [22] је предложена метода за динамичко аутоматско побољшање перформанси оптимизованих секвенцијалних машинских кодова. У суштини, направљен је софтверски слој за поновно превођење бинарних датотека, које производи паралелизован и/или векторизован код, како предвиђањем контроле, тако и паралелизацијом петљи и рекурзивних рутина. Њихова метода се делом преклапа са методом предложеном у овом раду јер као и ова ради са бинарним

датотекама, али се и значајно разликује јер се фокусира на паралелизацију петљи и рекурзивних рутина, док је у овом раду фокус на паралелизацији високе резолуције општих иргуларних програма, на нивоу инструкција.

Аутори рада [23] дали су опис потпуно аутоматизованог преводиоца под називом POSH, који динамички, у време покретања открива паралелизам на нивоу нити (енг. Thread-Level Speculation - TLS), изграђеног као додатак на постојећи преводилац gcc. POSH дели код на задатке који обједињавају подрутине и петље издвојене једноставним профилисањем кода. За разлику од решења које се предлаже у овој дисертацији, њихов динамички алат је проверен само на симулационом моделу, конкретно на моделу TLS вишепроцесорске физичке архитектуре са 4 суперскаларна језгра.

У радовима [24] и [25] описан је први аутоматски спекулативан систем DOALL паралелизације за кластере, који се састоји од преводиоца за паралелизацију у време превођења и спекулативне паралелизације у току извршења. Смањује трошкове комуникације и провере спекулативног извршења. Постиже убрзање и до 43.8x на кластеру са 120 језгара. Са друге стране, у решењу приложеном у овој диертацији, циљају се вишејезгарни процесори и статичка паралелизација машинског кода на нивоу инструкција (ILP).

Аутори [26] су приказали технике за аутоматско проналажење паралелизма унутар скрипти, применљиве на различите динамичке скрипт језике. Комбинујући скрипту са одговарајућим интерпретером, кроз технике специјализације програма, технике које аутори предлажу примењују различите облике паралелизације у оквиру скрипте и тиме добијају комбиновани програм који се накнадно паралелизује унапређеним аутоматским спекулативним методама. Тестирањем са два различита скрипт интерпретера отвореног кода (Lua и Perl), сваки са по 6 улазних линеарних алгебарских скрипти, техника постиже убрзање од 5.10x на 24-језгарној платформи. Насупрот овој техници за динамичке скрипт језике, решење предложено у овој дисертацији намењено је паралелизацији машинског кода.

2.3 Методе паралелизације које су намењене прототиповима процесора који су направљени у FPGA

У [27] дат је допринос оптимизацији секвенцијалних програма у току извршења (енг. runtime) кроз предлог под називом MP-Tomasulo. MP-Tomasulo је модел извршења секвенцијалних програма који, на нивоу задатака, може да мења редослед извршења. Аутори предлажу схему распређивања рутина ван распореда (енг. out-of-order) за било коју физичку архитектуру, а потврдили су је на прототипу MPSoC вишепроцесорске рачунске платформе направљене у FPGA. MP-Tomasulo има способност аутоматског елиминисања неких зависности података како би омогућио извршење ван распореда, на нивоу задатака. Постигнуто је теоријско убрзање са врхом од преко 95% за различите облике зависности података између задатака. За разлику од решења које аутори предлажу, решење описано у овој дисертацији предлаже паралелизацију секвенцијалних програма статичким распоређивањем које се може извршити приликом превођења (енг. compile-time) односно у току покретања (енг. just-in-time), а слично као и MP-Tomasulo, верификовано је на моделу вишејезгарног процесора направљеном у FPGA.

Аутори [28] су направили јединствену скалабилну структуру копроцесора са линеарним низом процесних елемената. Структура копроцесора је направљена и мапирана на FPGA физичкој архитектури. Коришћена је као окружење за извршавање партиција алгоритама за декомпозицију великих матрица проточном структуром веће резолуције. Подела на партиције је изведена статички и, имајући у виду да је заснована на подели итерационог простора петљи у алгоритмимама и конкретно спољне петље у двонивовској структури петљи, ограничена је на четири различита алгоритма за декомпозицију матрица. Сви покривени алгоритми имају исту двонивовску структуру петљи и исте зависности података. Слично, у решењу описаном у овој докторској дисертацији, користи се модел вишејезгарног процесора направљен у FPGA, али паралелизатор може да дели било који проблем на партиције намењене различитим процесорским језгрима, а не само алгоритме за декомпозицију матрица.

У [29] је приказана паралелна архитектура са идејом мањег утрошка расположивих ресурса у поређењу са другим архитектурама, што су аутори и показали на примеру брзе Фуријеове трансформације (енг. Fast Fourier Transform - FFT). Са друге стране, решење које је овде описано се фокусира на једноставност развоја C кода уз аутоматску паралелизацију за вишејезгарне процесоре направљене у FPGA, занемарујући могућност већег утрошка расположивих ресурса.

2.4 Оптимизације засноване на партиционисању графова

Идеја поделе графа на партиције у циљу паралелизације су већ описане и проверене у неким радовима. Аутори [30] и [31] користе сличне методе за поделу великих модела података на партиције које се даље обрађују независно и (потенцијално) паралелно. Модел су представили као DDG. DDG се у првој фази, коришћењем METIS алгоритма [32], подели на партиције. Подела на партиције је накнадно оптимизована кроз неколико стратегија.

Аутори [33] су направили флексибилно окружење за превођење, под називом Flextream. Ово окружење динамички прилагођава покренуту апликацију променљивим карактеристикама циљне физичке архитектуре. Користе статичко превођење, као што је случај и у решењу из ове дисертације, али уз технике динамичког прилагођавања у каснијим фазама, конкретно током извршавања. Користе StreamIt преводилац као полазну тачку, да би касније дељење извесних графова на партиције радили помоћу METIS алгоритма и одређених хеуристика. Flextream се фокусира на област рачунарства везану за репродуковање садржаја по преузимању (енг. streaming computing), односно бави се распоређивањем различитих независних фаза репродукције преузетог садржаја. Насупрот томе, решење описано у овој дисертацији има за циљ распоређивање машинског кода на инструкцијском нивоу за опште апликације, дакле ради поделу веће резолуције. Такође, аутори су верификовали своје решење на хетерогеном вишејезгарном систему, кокретно на IBM Cell процесору, док решење описано у овој дисертацији има за циљ хомогени SMP систем.

ПОГЛАВЉЕ 3.

ПРИМЕРИ ЦИЉНИХ ФИЗИЧКИХ АРХИТЕКТУРА

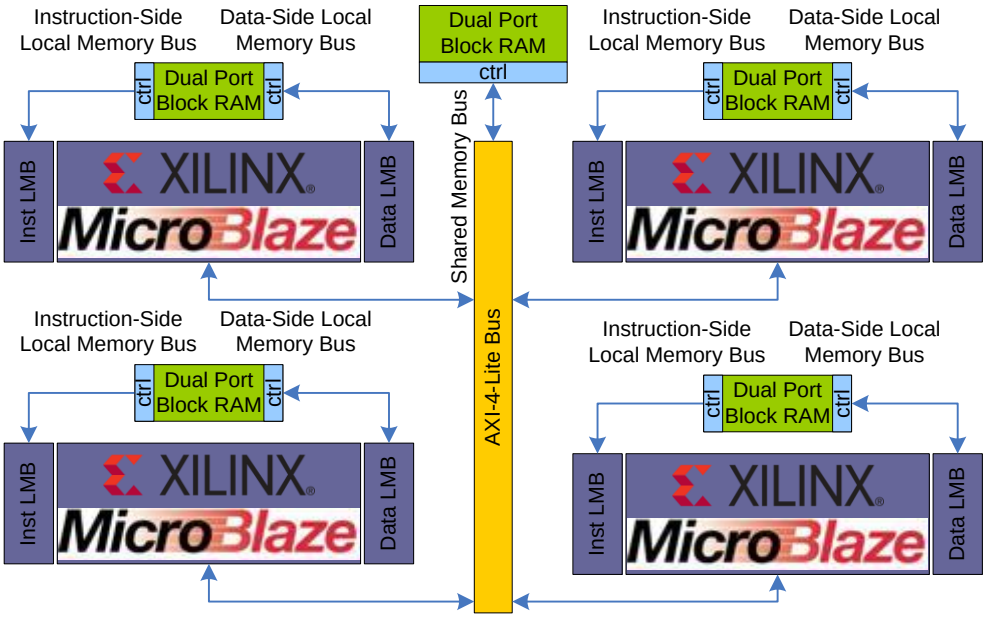
Спектар подржаних архитектура има одређена ограничења односно претпоставке уведене на почетку. Стога и платформа која ће бити описана мора да задовољи одређене критеријуме. Због једноставног механизма синхронизације података помоћу load-store комбинације инструкција, сва језгра морају бити повезана на заједничку дељену меморију. Такође, из већ појашњених и аргументованих разлога, архитектура не садржи скривену меморију.

Вишејезгарна MIPS32 платформа, као симетрични вишепроцесорски систем са дељеном меморијом (енг. Symmetric Multi Processor - SMP) задовољава све наведене услове. Регистарске инструкције које користе 32 општенаменска регистра имају тачно дефинисана и детерминистична времена извршења. Још једна предност је већ постојећи преводилац језика C (gcc) којим се може добити асемблерски код. Најпре је за валидацију коришћен софтверски симулациони модел управо те платформе, уз симулатор компаније Open Virtual Platforms™ под називом OVPsim™. Различит број MIPS32 језгара уз једну дељену меморију су коришћени у процесу валидације. Прецизније, коришћена је OR1K SMP платформа. За време симулације процесор је извршавао паралелизован асемблерски код у очекиваном броју корака (инструкција).

Валидација овог приступа на физичкој архитектури је додатни изазов. У те сврхе је израђен модел процесора који се може имплементирати на FPGA и

представља циљну архитектуру која је такође имплементирана у оквиру истраживања. Тренутно веома популаран софтверски процесор имплементиран у FPGA долази из компаније Xilinx под називом MicroBlaze™. То је 32-битна RISC Харвард архитектура и може се конфигурисати тако да не користи скривену меморију. Даље, могуће је инстанцирати више од једног процесора (језгра), па је направљен модел са 1, 2 и 4 језгра, како би се показала скалабилност на примерима. Сва језгра су повезана на заједничку дељену меморију, као што је урађено у [35], где аутори добијено решење користе за валидацију њиховог приступа распоређивању нити. У случају прикључивања више језгара на исту дељену меморију, арбитер унутар меморијског контролера опслужује захтеве који долазе са више језгара. Како је у раду већ наглашено, сваки приступ меморији се сматра приступом истој меморијској локацији, па је могућност истовременог приступа меморији искључена, а самим тим и могућност недетерминистичког редоследа извршења инструкија која би променила семантику паралелног програма.

Модел циљне архитектуре развијен је и синтетисан за уређај XC6SLX150 из фамилије Spartan 6 која долази из компаније Xilinx. Радни такт сваког језгра износи 108 MHz, а утрошак ресурса на наведеном уређају износи 9%, конкретно 1% доступних регистара и 5% LUT табела.



Слика 1 Четворојезгарна Xilinx MicroBlaze циљна архитектура

У приказаном моделу (Слика 1), осим на дељену меморију, свако језгро је повезано и на локалну инструкцијску меморију односно меморију података путем LMB магистрале. Сегмент података неке преведене апликације, мапира се на дељену меморију тако да језгра могу међусобно да размењују податке. Сви остали сегменти (нпр. програмски сегмент, магацинска меморија, динамичка меморија, итд.) су мапирани на локалну меморију, тако да свако језгро може независно да извршава код који је њему намењен.

Предност овог модела процесора на FPGA је софтверска симулација, уз могућност покретања на физичкој архитектури. На тај начин, приступ се може проверити и на инструкцијском нивоу.

У плану је и израда архитектуре са већим бројем језгара, али таква архитектура не може да се имплементира коришћењем једне дељене меморије којој могу да приступају сва језгра, већ се мора направити више нивоа меморије, а самим тим и сложенији систем синхронизације података. Ефикасна комуникација између процесорских језгара имплементираних на FPGA би била прилично захтевна. Један пример такве комуникационе мреже, ефикасне по питању утрошка ресурса, приложен је у [10].

ПОГЛАВЉЕ 4.

ПАРАЛЕЛИЗАТОР МАШИНСКОГ СЕКВЕНЦИЈАЛНОГ КОДА

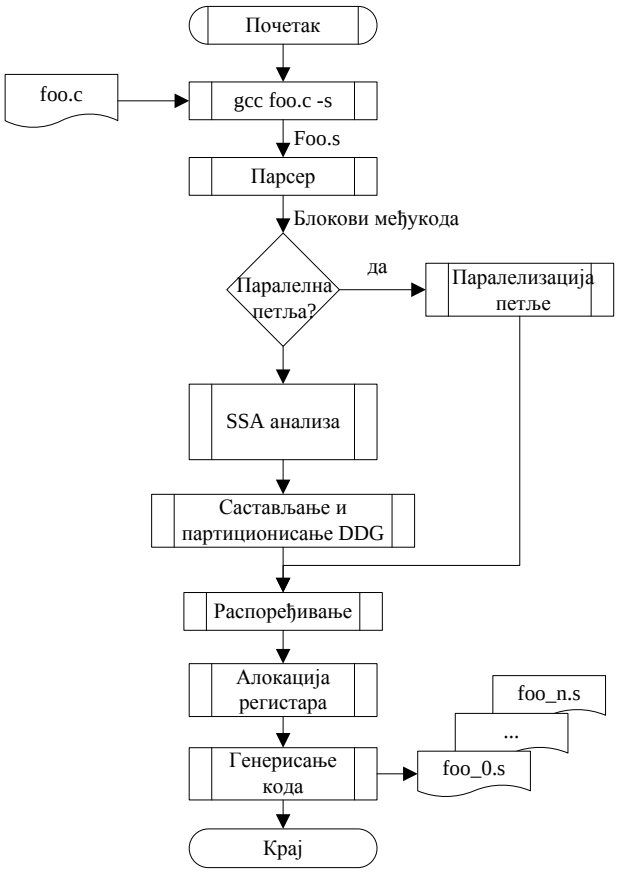
На основу изнетих чињеница у овом истраживању је разматрана аутоматска паралелизација на бинарном (машинском) нивоу. Машински код у бинарном облику је узет као улаз у паралелизатор јер садржи све детаље о циљној архитектури. Иако је маневарски простор за машински независне и макро-оптимизације изгубљен доласком на бинарни ниво где се губе информације са виших нивоа, све информације неопходне за паралелизацију на нивоу инструкција су добијене. С друге стране, машински независне оптимизације су већ размотрене од стране модерних преводилаца (нпр. gcc), те би стога паралелизатор који се предлаже могао успешно бити коришћен као посебна етапа у задњем делу истог преводиоца. У овој методи се показује да је паралелизација могућа без симболичких информација или информација о индексима низова. Добро уравнотежење оптерећења постиже се новим алгоритмом за паралелизацију на нивоу инструкција, развијеном за асемблерски код. Алгоритам користи имена регистара након познате SSA анализе у циљу откривања независних целина (блокова) кода који се могу партиционисати и распоредити на језгра процесора најпре коришћењем METIS алгоритма, уз неколико унапређења уведих у етапама које следе након тога. Идеја партиционисања графа у циљу паралелизације је већ описана и испробана у

неким радовима. Слична метода је коришћена у [30] и [31] за партиционисање великоих модела података за каснију дистрибуирану обраду. Аутори су модел представили као DDG. DDG се партиционише коришћењем METIS [32] алгоритама у почетку, уз каснију фину дораду помоћу неколико стратегија.

4.1 Архитектура паралелизатора

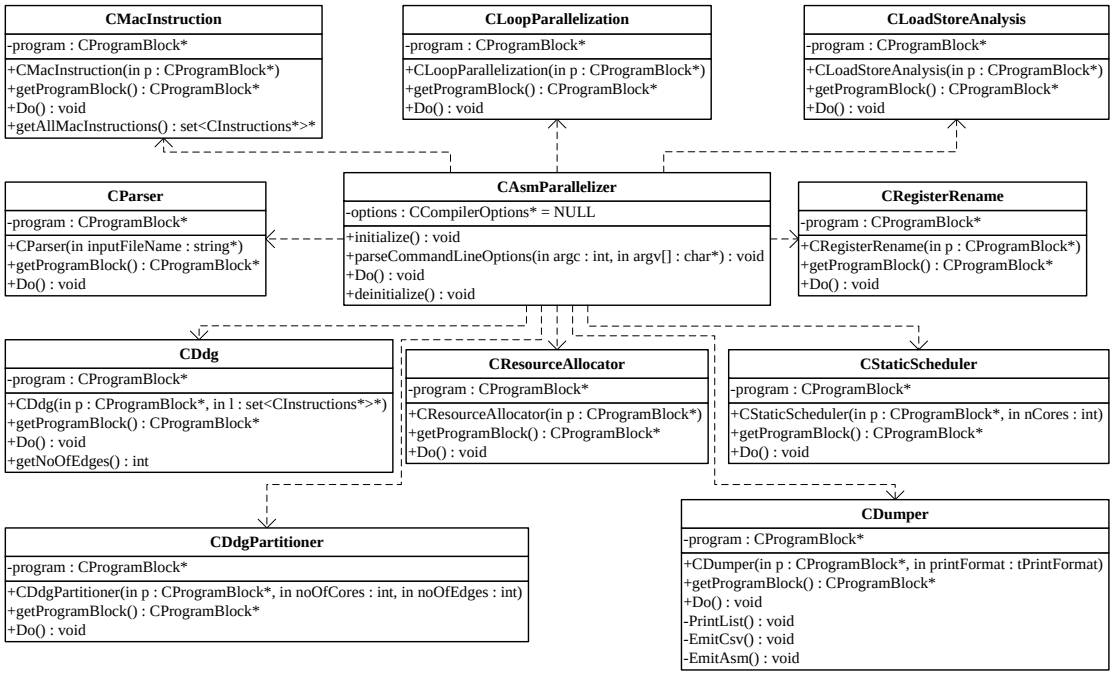
Како се асемблерски код показао као згодно место за паралелизацију [14], одабран је као улаз и у овом случају. Асемблерски код, осим већ написаног у том облику, може се добити превођењем изворног кода (нпр. написаног програмским језиком C) или чак дисасемблирањем извршног бинарног кода. Идеја је да се улазна асемблерска датотека парсира, затим направе одређене трансформације на њој, затим паралелизује чиме би се добио асемблерски код за свако језгро процесора и да се на крају сачува код за свако језгро у одговарајућем формату.

Да би се опис тока методе и детаљи важнијих блокова који следе учинили разумљивијим, у наставку је дат поједностављен опис репрезентације кода која ће се користити за време паралелизације. Назива се међурепрезентација или међукод (енг. Intermediate Representation - IR). То је структура података у облику стабла, а састоји се од следећих елемената: (1) програм – коренски елемент који представља цео асемблерски програм и садржи једну или више функција, (2) функција – одговара једној асемблерској функцији и садржи један или више основних блокова (енг. basic block), (3) основни блок – део кода који представља непрекидив низ инструкција из којег се не може искочити, нити се у њега може ускочити, може почети лабелом и/или завршити се инструкцијом скока, а садржи једну или више инструкција од којих ни једна није скок, (4) инструкција – престаљва асемблерску инструкцију која садржи операнде и (5) операнди – најмањи елементи, тј. лишће и могу бити дељени између више инструкција.



Слика 2 Ток паралелизације асемблерског кода

Ток паралелизације асемблерског кода приказује Слика 2. Програм написан програмским језиком C (датотека "foo.c") се прослеђује преводиоцу gcc уз параметар "-s", који преводиоцу означава да након процедуре превођења треба да направи асемблерску датотеку уместо извршне машинске датотеке. Асемблерска датотека "foo.s" се потом парсира и преводи у IR. Преостали блокови образују сваки основни блок понаособ. У случају да је основни блок препознат као паралелна петља (тј. петља са независним итерацијама) партиционише се унутар блока за паралелизацију петљи, док се у осталим случајевима основни блок сматра општим и пролази кроз трансформацију у SSA форму и граф зависности података (DDG) који се коначно партиционише унутар одговарајућег блока. Партиционисан основни блок се потом паралелизује распоређивањем инструкција на различита језгра уз уважавање информација добијених у току фазе партиционисања као почетне тачке, али уз додавање синхронизационих инструкција по потреби. Како постоји могућност да су нове промењљиве уведене у току различитих фаза, линеарна додела регистара се



Слика 3 Дијаграм класа паралелизатора асемблерског кода

примењује на сваки блок, за свако процесорско језгро. Коначно, генератор кода од паралелизованог IR прави одвојене асемблерске датотеке за свако језгро.

Десет главних делова се могу издвојити из овог тока: (1) парсер асемблерског кода, (2) паралелизатор петљи, (3) анализа животног века променљивих и преименовање регистара, (4) генератор графа зависности података (DDG), (5) анализатор инструкција за приступ меморији, (6) анализатор макроинструкција, (7) METIS-базирани блок за партиционисање, (8) статички распоређивач инструкција, (9) додела регистара и (10) генератор кода. Ови делови, односно класе и њихове релације су илустровани дијаграмом класа (Слика 3), а детаљи о овим деловима су презентовани у наставку овог поглавља.

4.1.1 Парсер асемблерског кода

Овај блок реализован у класи *CParser* је синтаксни анализатор асемблерског језика који преводи текстуалне датотеке са асемблерским кодом у програмске објекте, односно претходно описану структуру података названу IR.

Анализатор чита линије кода једну по једну и сваку раздваја на токене, тако што их дели на тип инструкције и операнде помоћу једноставног аутомата са коначним бројем стања.

Нпр. линија

```
ADD $t1,$t2,$t3
```

се подели на 4 токена од којих се направи објекат инструкција. Токен *ADD* би био препознат као тип инструкције, док су *$t1* одредишни (дефинисани) операнд, а *$t2* и *$t3* изворишни (коришћени) операнди.

Парсирајући сваку линију независно, овај блок попуњава IR структуру, тј. раздваја на програм, функције и основне блокове. Раздвајање блокова (већих од једне инструкције) се ради на основу пропознавања асемблерских директива и познатих конструкција. Сходно томе, свака инструкција је окарактерисана типом инструкције и помоћу две листе, *usesList* и *defsList*, које представљају листу променљивих које дата инструкција користи и листу променљивих које инструкција користи, тим редом.

4.1.2 Паралелизатор петљи

Сваки основни блок који се препозна као пребројива петља са независним итерацијама може да се паралелизује поделом итерационог простора на *N* мањих простора, где је *N* број процесорских језгара. Метода за препознавање паралелних петљи није део класе *CLoopParallelization*, већ се користи претпоставка да информација о овим петљама већ постоји, у виду назнаке везане за основни блок који представља такву петљу. Другим речима, сматра се су те петље већ пропознате, односно да су означени почетак и крај сваке петље, пре саме паралелизације. Ознака за почетак петље такође садржи и иницијалну вредност бројача *i* и број итерација *n* (нпр. *#begin_parallel_loop i=0, n=4*). Крај петље је означен са *#end_parallel_loop*.

Када паралелизатор наиђе на назнаку *#begin_parallel_loop*, код који се налази између те назнаке и назнаке *#end_parallel_loop*, сматра се паралелном петљом, односно *parallelLoop* основним блоком. *ParallelLoop* основни блок се увек прослеђује блоку за паралелизацију петљи, који такав основни блок дели на *N* основних блокова. Сваки новонастали блок представља исту петљу, али над другачијим итерационим простором. Итерациони простор блока *x* почиње од итерације i'_x и има *n'* итерација, што се рачуна као:

$i'_x = i + x * n'$, где је

$x = 0, 1, \ldots, N$

$n' = n/N$.

Очито инструкције из *parallelLoop* основног блока које се користе за иницијализацију бројача петље и проверу услова за прекид петље морају да се посматрају другачије од осталих инструкција. Оне се не паралелизују, већ се понављају у сваком блоку, за свако језгро, али уз измене у складу са новом почетном и граничном вредношћу бројача. За пример секвенцијалне петље који приказује Слика 4, резултујуће блокове добијене након паралелизације показује Слика 5. Линија означена са *#loop body* представља тело пралелизоване петље и копира се у основне блокове свих језгара.

```
Loop:
li $2, 0   # loop_init
li $3, 8   # loop_limit
#loop body
addi $2, $2, 1
bne $2, $3, Loop
```

Слика 4 Пример секвенцијалне петље

```
Loop:
li $2, 0   # loop_init
li $3, 3   # loop_limit
#loop body
addi $2, $2, 1
bne $2, $3, Loop
```

а) језгро 0

```
Loop:
li $2, 4   # loop_init
li $3, 8   # loop_limit
#loop body
addi $2, $2, 1
bne $2, $3, Loop
```

б) језгро 1

Слика 5 Пример петље након паралелизације

4.1.3 Преименовање регистара

Зависности података у већини случајева представљају уско грло за паралелизацију и друге оптимизационе технике. Да би се смањиле зависности података међу инструкцијама, може се избећи редефиниција исте промењиве, тј. вишеструки упис у исту промењиву. Тиме се добија више промењивих, али са краћим животним веком, а самим тим се и смањује број промењивих које су у сметњи, те се редукују и зависности података. SSA форма обезбеђује управо то и, узимајући у обзир да решење које се предлаже ради у домену једног основног блока, може се користити поједностављен облик SSA. Применом на један основни блок, SSA форма се за потребе овог решења може интерпретирати као: свака промењива се дефинише (уписује) само једном и након тога се може семо користити (читати) [36]. Ово се постиже раздвајањем сваке промењиве на више инстанци. Нова инстанца исте промењиве се прави као потпуно нова промењива са именом добијеним додавањем суфикса (јединственог идентификатора) на корен имена (име полазне промењиве). Нова промењива се прави на месту сваке редефиниције полазне промењиве. На тај начин се добијају веома једноставни ланци коришћења и дефиниције, са само једним елементом. Табела 1 приказује разлику између примера изворног кода и одговарајућег кода у SSA форми.

На нивоу асемблера, за циљне архитектуре (нпр. MIPS, Microblaze, итд.), полазни IR се може трансформисати у SSA форму методом преименовања регистара, јер су регистри главна веза и, сходно томе, главна зависност између инструкција. Регистри који су од виталног значаја за коректно извршење асемблерског програма (претходно обојене промењиве) се морају изузети из

Табела 1 Ток SSA анализе и доделе регистара

Изворни код	SSA форма	Живе промењиве	Регистри
A=5	A.1=5	A.1=res1	R1
B=7	B.1=7	A.1=res1, B.1=res2	R1, R2
A=A*B	A.2=A.1*B.1	A.1=res1, B.1=res2, A.2=res3	R1, R2, R3
A=A+1	A.3=A.2+1	A.2=res3, A.3=res1	R1, R3

процеса трансформације кода у SSA форму. Преименовање ових регистара, тј. променљивих, те додела нових променљивих различитим регистрима могло би да доведе до неконзистетног извршења програма. Скуп оваквих регистара је различит за сваку архитектуру. За MIPS и Microblaze асемблер, скуп оваквих регистара се састоји од: регистра *zero* (*$0*) са констратном вредношћу 0, привременог асемблерског регистра (*$at*) који користи асемблер ради превођења псеудо-инструкција у машинске инструкције, регистара за повратне вредности функција, те срачунавање израза (*$v0* и *$v1*), регистара за аргументе функција (*$a0*, *$a1*, *$a2* и *$a3*), регистара резервисаних за језгро оперативног система (*$k0* и *$k1*), регистра за глобални показивач (*$gp*) и регистра повратне адресе (*$ra*) [37].

Полазни IR се трансформише у SSA форму за сваки основни блок понаособ, у једном пролазу, увећањем бројача инстанци сваке променљиве (касније јединственог идентификатора који се користи у формирању имена нове променљиве) на месту сваке редефиниције. Истовремено, новонастала променљива се додаје у IR, а све је реализовано у класи *CRegisterRename*.

4.1.4 Генератор DDG

Главни проблем у паралелизацији је одабир репрезентације секвенцијалног кода који би био погодан за партиционисање и распоређивање на процесорска језгра. Након издвајања партиција, број и цена зависности између партиција би требало да буду минимални.

Апстракцијом програма у виду графа, добро су покривене структуре зависности које постоје у програму. DDG изведен преко свих инструкција је већ коришћена структура у [14]. У овом случају су чворови графа инструкције, док везе представљају зависности две инструкције преко података. Да би се одредиле зависности међу инструкцијама, разматрају се променљиве из сваке инструкције. Две инструкције се сматрају зависним било да користе или да дефинишу исту променљиву.

DDG се унутар класе *CDdg* генерише као листа свих инструкција, где свака инструкција поседује листу инструкција од којих зависи. Поред цене инструкције (тежина чворова графа) која је већ дефинисана унутар блока *Парсер асемблерског кода*, за партиционисање је неопходна и цена зависности међу

њима (тежина везе у графу). У време прављења графа, тежина сваке везе се одређује као производ броја променљивих које уносе зависност две инструкције и цене инструкција потребних за синхронизацију једне променљиве. Тежине чворова користи блок за партиционисање да би одредио оптерећење сваке партиције и учионио оптерећења уравнотеженим, док се тежина везе користи за одређивање исплативости приликом сечења неке везе ради поделе партиције, те прављења нових партиција које би се извршавале истовремено. У овом случају, сечење веза би значило потребу за синхронизационим инструкцијама, тј. додавање инструкција за пренос података са једног језгра на друго.

4.1.5 Анализатор инструкција за приступ меморији

Према Бернштајновим условима [38], два програмска сегмента P_i и P_j су независна и могу да се извршавају истовремено акко је задовољено следеће:

$$I_j \cap O_i = \varnothing,$$
$$I_i \cap O_j = \varnothing \text{ и}$$
$$O_i \cap O_j = \varnothing,$$

где је I_i скуп улазних променљивих и O_i скуп излазних променљивих сегмента P_i, а слично важи и за P_j. Примењено на ниво инструкција, две инструкције су зависне акко приступају истој променљивој, а да бар једна инструкција приступа тој променљивој ради писања. У асемблеру се ово може односити и на регистре и на променљиве, односно на све типове меморијских ресурса. Проблем код регистара је неутрализован претходно описаном техником преименовања ресурса, док меморија остаје проблем. Стога се мора избећи истовремени приступ једној променљивој, где је циљ бар једног приступа упис у меморијску локацију.

За први корак је примењенно најједноставније, конзервативно решење овог проблема, а то је да се цела меморија сматра једном локацијом, те да инструкције за упис у меморију никада не распоређују тако да буду паралелне са другом инструкцијом за приступ меморији, већ обавезно смакнуте бар за 1 такт. Међутим, то је и најмање ефикасно решење јер значајно смањује паралелизам.

Анализом адреса локација којима приступају инструкције за приступ меморији, односно инструкције за читање и упис (енг. load/store instructions), неки случајеви се могу препознати и неке (али не све) локације којима се приступа овим инструкцијама се могу сматрати различитим, као што тврде и аутори [39]. Стога, ове инструкције за читање и упис се могу сматрати као независне и могу бити распоређене тако да се извршавају истовремено. У овом алату искоришћена је једноставна, али довољно ефикасна метода за анализу меморије, омогућена чињеницом да се ради са асемблерским кодом. Наиме, асемблерски код често адресира меморијске локације са истом основном (базном) адресом, али различитим одстојањем. То је веома типично за инструкције за читање и упис, попут:

```
lw $t0, 4($t1)
```

где је *$t0* одредишни регистар, *$t1* је регистар који садржи базну адресу, а *4* је одстојање од адресе у регистру *$t1*, које чини адресу меморијске локације потпуно дефинисаном. У овом случају, реч се са локације на адреси *($t1+4)* чита у регистар *$t0* и то јесте пример индиректног регистарског адресирања. У овој анализи, откривају се две инструкције за читање или упис које адресирају различите локације где је базна адреса у регистру има исту вредност, док се одстојање разликује. Поређење вредности базне адресе у регистру није потребно јер, према описаној модификованој SSA форми IR, регистри са истим именом увек имају исту вредност. Ово чини анализу у класи *CLoadStoreAnalysis* веома једноставном, али ипак и веома корисном, јер се показало да открива многе независне инструкције у смислу приступа сигурно различитим меморијским локацијама, нпр. у случају када се приступа елементима неког низа.

Аутори [34] су направили приступ одређивању зависности података над асемблерским кодом. Дефинисан је напредан алгоритам за пропагацију симболичких вредности, којим се добијају зависности података између меморијских операција и то не само на основу адресе, већ и на основу вредности. Користи се за оптимизацију асемблерског кода (нпр. за повећање паралелизма на нивоу инструкција) правећи много тачније анализе зависности података у већини

случајева. Решење описано овде се такође бави паралелизмом на нивоу инструцкија, али са фокусом на зависности регистара, користећи једноставну методу за анализу зависности. Решење које су понудили аутори се свакако може користити као додатна оптимизациона техника у решењу описаном у овој докторској дисертацији.

4.1.6 Анализатор макроинструкција

Под макроинструкцијама се сматра низ од две или више инструкција које чине логичку целину и имају имплицитну зависност преко специфичног регистра које користе (нпр. акумулатора, привременог асемблерског регистра *$*at и слично). Сврха увођења блока за анализу макроинструкција јесте спречавање случаја да блок за партиционисање DDG, инструкције које припадају једној макроинструцкији, мапира на различите партиције, а самим тим и, потенцијално, на различита језгра. Овакво партиционисање би довело до скупе режије због синхронизације два језгра, али и потенцијално до проблема преплитања са другом макроинструкцијом, односно неком инструкцијом унутар ње која имплицитно користи исти специфични регистар. Ово би даље могло да доведе до недетерминизма, односно неконзистетног извршења програма, те до лоших резултата неке обраде. Другим речима, овај блок помаже *METIS-базирани блок за партиционисање графа* да све инструкције које чине једну макроинструкцију задржи заједно и распореди на исто језгро.

Као што је већ поменуто, проблем са овим инструкцијама је имплицитна употреба регистара попут *$acc* или *$at* који су чак и преходно обојени (не може се радити преименовање регистара). Без анализе макроинструкција, *METIS-базирани блок за партиционисање графа* не може да препозна да постоји зависност података између ових инструкција.

Анализа макроинструкција се ради у једном пролазу и имплементирана је у класи *CMacInstruction* и заправо је слична анализи животног века променљиве примењеној на специфичан регистар попут *$acc*. Познато је да инструкције попут *mult*, *div*, *mthi* и *mtlo* имплицитно дефинишу и регистар *$acc*, односно резултат операције смештају у наменски акумулатор. То се сматра почетком макроинструкције. Макроинструкција завршава се последњом инструкцијом која користи овај регистар *$acc* (последња употреба пре наредне дефиниције истог

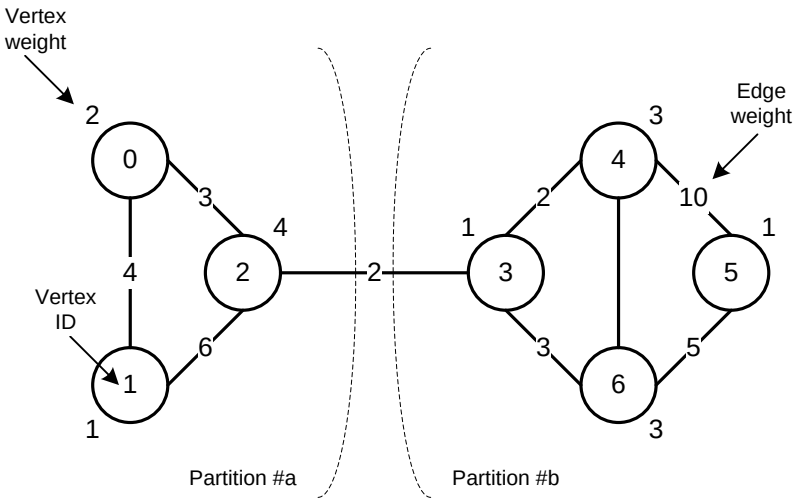
регистра неком од инструкција *mult, div, mthi* и *mtlo*). Све инструкције унутар макроинструкције се потом умотавају у једну инструкцију која је названа *macroinstruction*. Њене *use* и *def* листе садрже све променљиве (њихову унију) из *use* и *def* листа инструкција које обухвата. Слично, цена (тежина чвора) једне овакве макроинструкције се рачуна као збир цена свих инструкција које обухвата.

Након анализе, свака макроинструкција се додаје у један скуп инструкција и тако представља концепт сличан табели макродефиницаја који је типичан за асемблерске језике. Касније се, након партиционисања кода где се низ ових инструкција сматрао једном скупом и недељивом инструкцијом, попут замене макроинструкције одговарајућом макродефиницијом у типичном асемблерском језику, макроинструкција (омотач око више инструкција) мења њеном дефиницијом која се може пронаћи у табели.

4.1.7 METIS-базирани блок за партиционисање графа

У раду се предлаже метода паралелизације која користи METIS Multilevel k-way [40] алгоритам за партиционисање графа, где је граф сачињен од асемблерског кода уз уважавање неколико типова зависности, на основу доступних ресурса на циљној архитектури који представљају уска грла у овом процесу. Може се сматрати да су све зависности заправо зависности података, те се генерише DDG на раније описан начин. Информације које се добијају партиционисањем DDG се користе касније као полазна тачка правог партиционисања, тј. паралелизације са статичким распоређивањем инструкција. Другим речима, партиционисање се касније унапређује, а за почетак се као подразумевано одредиште (партиција) неке инструкције сматра она која је добијена као резултат рада овог блока.

На бази DDG, METIS алгоритам се користи за поделу инструкција (чворова графа) на различита језгра процесора (партиције). Алгоритам дели граф на партиције, задржавајући уравнотеженост оптерећења у свим партицијама, на основу претходно дефинисаних цена инструкција (тежине темена). Алгоритам настоји да минимизује број веза које се секу, односно укупну цену свих пресечених веза, на основу цене сваке везе дефинисане у време генерисања DDG. Пример партиционисања DDG приказује Слика 6. У датом примеру, овај



Слика 6 Партиционисање графа зависности података

блок пресеца везу која има тежину 2. То је најбоље решење за дати пример, јер ствара једну партицију са 3 чвора (инструкције) укупне тежине 7 и другу са 4 чвора укупне тежине 8. Остала могућа решења дају лошије уравнотежено оптерећење и/или већу цену пресечених веза што би довело до скупље режије. Веза која је пресечена има цену 2, што значи да се ради синхронизације података морају додати синхронизационе инструкције чија је укупна цена 2, што би у најгорем случају могло довести до продужења основног блока за 2.

Да би се и у паралелној верзији програма одржала семантика секвенцијалног програма, неке инструкције, попут раније откривених макроинструкција описаних у претходној секцији, изузимају се из процеса мапирања на различита језгра. Примера ради, инструкције које припадају једној макроинструкцији имају зависност преко заједничког ресурса, тј. специфичног регистра попут акумулатора *$acc*. Распоређивање ових инструкција на различита језгра би довело до пресецања прикривених веза и на крају до неконзистетног извршавања програма, те стога морају да се држе заједно, на истом језгру.

4.1.8 Статички распоређивач инструцкија

Користећи се информацијама добијених партиционисањем, статички распоређивач поставља инструкције у одговарајуће партиције, како би попунио различита језгра процесора. Такође, овај блок може да попут компактора мења редослед инструкција како би направио боље уравнотежено оптерећење међу језгрима. Коначно, овај блок додаје пролог и епилог у виду низа инструкција на

сваки основни блок, како би обезбедио редукцију података, слично као што то ради TBB [3]. Тиме је обезбеђен *синхронизам извршења* (енг. execution synchronism) и *синхронизам података* (енг. data synchronism) кроз сва процесорска језгра.

Синхронизам извршења подразумева да сва језгра започну извршавање наредног основног блока у истом тренутку, што гарантује конзистентност података и котроле. Овај принцип је сличан *операцији баријери* (енг. Barrier operation) коју користи OpenMP [7], OpenCL [8], Python, и др. За разлику од сличне методе коришћене у поменутим решењима, *синхронизам извршења* који је овде описан представља генерализовану *операцију баријере* имајући у виду да језгра која достижу исту баријеру не морају нужно да извршавају исти код.

Синхронизам података подразумева да сва процесорска језгра започињу извршење наредног основног блока са истим скупом неопходних података. Ова техника је слична *редукцији података* коју користи TBB [2] [3] [4] јер подразумева међујезгарну комуникацију за дистрибуцију неопходних података у минималном броју преноса.

Попут METIS-базираног блока за партиционисање графа, и овај блок мора на посебан начин да третира неке инструкције, тј. да спречи промену редоследа или преплитање неких инструкција и то не само оних где је та потреба очита, односно где је веза две инструкције због зависности података експлицитна. Примера ради, инструкције које чине једну *макроинструкцију*, не смеју бити преплетене са инструкцијама које чине другу *макроинструкцију* односно инструкцијама које имплицитно или експлицитно користе привремени акумулатор *$acc*.

Још једна важна улога овог блока је синхронизација података између језгара. Синхронизација је неопходна када инструкција на једном језгру (*core#a*) користи неку променљиву коју дефинише инструкција на другом језгру (*core#b*), а у питању су регистарске променљиве (меморија је заједничка па не представља проблем). Синхронизација је омогућена додавањем инструкције за чување променљиве на језгру *core#b* на којем се променљива дефинише, и додавањем инструкције за учитавање из меморије на другом језгру *core#a* пре коришћења, али након записа у језгру *core#b*.

```
Input: Basic block B, number of cores N.
Output: B partitioned to coreBlocks.
for i = 1 to N do
   CB[i] ← {};
end
IL ← B.instructionList();
while IL ≠ Ø do
   W ← doUrgencyRule(IL);
   while W ≠ Ø do
      foreach instruction I ∈ W do
         doSyncRule(I);
      end
      doFillRule(W);
      T ← W.top();
      CB[T.core] ← CB[T.core] ∪ T;
      W ← W \ T;
   end
end
maxLen ← max{Cx | Cx = |CB[x]|, 1 ≤ x ≤ N};
for i = 1 to N do
   len ← |CB[i]|;
   for j = 1 to maxLen - len do
      CB[j] ← CB[j] ∪ NOP;
   end
   CB[i] ← CB[i] ∪ B.epilogue();
end
return CB;
```

Слика 7 Псеудокод статичког распоређивача инструкција

Алгоритам распоређивања је имплементиран на следећи начин. У првом кораку се направи *N* основних блокова (*CB*), где је *N* број језгара. Сваки *CB* ће бити попуњен инструкцијама намењеним одговарајућем језгру. Алгоритам затим пролази кроз све инструкције у листи *IL* полазног основног блока (*B*), и за сваку инструкцију, коришћењем рутине *doUrgencyRule*, издваја у листу *W* све инструкције које су спремне за распоређивање јер су им све променљиве дефинисане. Даље, *W* листа се обрађује док не постане празна. Обрађује се у петљи, најпре рутином *doSyncRule*, која по потреби додаје синхронизационе инструкције, а позива се за сваку појединачну инструкцију *I* из листе *W*. Након петље, *doFillRule* рутина се позива за целу листу *W*, где се као у компактору додате *NOP* инструкције покушавају заменити корисним инструкцијама ради оптимизације. Прва инструкција *I* из *W* се потом коначно пребацује у блок *CB*.

Након што је листа *IL* обрађена, свим блоковима *CB* се, додавањем *NOP* инструкција, изједначе дужине, у смислу изједначавања укупне цене свих инструкција које садрже. У последњем кораку се инструкције из листе *epilogue* полазног блока *B* додају сваком блоку *CB*. Ово је најчешће инструкија гранања која чини крај основног блока или низ инструкција која омогућава *синхронизам извршења* и *синхронизам података*, а на овај начин је омогућено да се сви блокови заврше на исти начин и даље наставе заједно од познатог тренутка са конзистентним скупом података. Псеудокод алгоритма приказује Слика 7.

Рутина *doUrgencyRule(IL)* обрађује прослеђену листу инструкција *IL*. Она проверава да ли постоје инструкције које су спремне за распоређивање. Уколико постоје *спремне инструкције*, она преузима све ове инструкције из листе *IL* и пребацује их у листу спремних инструкција *RIL*. Листа *RIL* је такође и повратна вредност ове рутине. *Спремна инструкција* је она којој су сви подаци које користи већ дефинисани, односно инструкције које дефинишу те податке су већ распоређене на дато језгро или су њени подаци улазни у дати блок, па су већ унапред спремни у регистрима што је обезбеђено већ на крају претходног основног блока, раније описаном операцијом *редукције података*. Уколико размотримо пример који приказује Слика 8, улазни регистри се могу одредити помоћу претходно генерисаног графа тока управљања (енг. Control Flow Graph – CFG) и то су регистри *$4* и *$5*. Стога, само инструкције које користе ове регистре

```
LW $3, 12($4)
LW $2, 12($5)
LW $8, 0($4)
ADDU $7, $3, $8
ADDI $11, $7, 2
SUBU $10, $8, $2
```

Слика 8 Пример кода за одређивање инструкција спремних за распоређивање

(или евентуално не користе регистре, али таквих у овом примеру нема), могу бити пребачене из листе *IL* у листу *RIL*. Ово су прве 3 инструкције (*LW*). Инструкције *ADDU*, *ADDI*, и *SUBU* остају у листи *IL* јер регистри које оне користе још нису дефинисани (*$2*, *$3*, *$7*, и *$8*). У наредном позиву рутине *doUrgencyRule*, листа *RIL* је поново празна и, у односу на претходни позив, регистри *$2*, *$3*, и *$8* су дефинисани, тако да се инструкције *ADDU* и *SUBU* постале спремне. Сходно томе, инструкције *ADDU* и *SUBU* се могу пребацити из листе *IL* у листу *RIL*. Слично се поступак понавља, сваким позивом рутине *doUrgencyRule*, све док листа *IL* не постане празна, што би се у датом примеру десило већ након следећег позива.

Рутина *doSyncRule(I)* узима прослеђену инструкцију *I* која ће бити распоређена на *језгро А* и проверава да ли она користи неки регистар који јесте већ дефинисан, али на неком другом језгру, нпр. *језгру Б*. Уколико то јесте случај, ова рутина додаје неопходне синхронизационе инструкције, тј. меморијске инструкције писања и читања (енг. load/store) на *језгро А* и *језгро Б*. Инструкција (*SW*) за писање вредности из спорног регистра на меморијску локацију *X* се додаје на *језгро Б*, после дефиниције тог регистра за који се ради синхронизација, док се инструкција (*LW*) за читање вредности из меморијске локације *X* у спорни регистар додаје додаје на *језгро А*, обавезно у времену након инструкције *SW*, а пре инструкције која користи регистар за који се ради синхронизација. Две синхронизационе инструкције не смеју да се извршавају истовремено, већ смакнуто, у наведеном редоследу, а то се по потреби решава уметањем *NOP* инструкција на *језгро А*. Пример асемблерског кода након примене описаног поступка синхронизације показује Слика 9. У датом примеру, инструкција *ADDI* са *језгра 0* користи регистар *$3* који се на *језгру 1* дефинише

```
NOP                 LI $3, 4
NOP                 SW $3, 34($0)
LW $3, 34($0)
ADDI $4, $3, 1
```

а) језгро 0 б) језгро 1

Слика 9 Исечак асемблерског кода након додавања синхронизационих инструкција

инструкцијом *LI*, па је синхронизациона инструкција *SW* додата на *језгро 1* након инструкције *LI*, док је синхронизациона инструкција *LW* додата на *језгро 0* у времену после *SW*, а пре него што је распоређена инструкција *ADDI*. Уједно, *NOP* инструкције су додате на *језгро 0* како би се сачекало на дефиницију регистра *$3* и на упис вредности из тог регистра на локацију *34($0)* са које ће иста вредност бити прочитана у регистар *$3*, али на *језгру 0* и тиме завршена синхронизација овог податка.

Рутина *doFillRule(W)* мења *NOP* инструкције из листе *W* неким корисним инструкцијама, уколико је то могуће. Листа *W* је листа спремних инструкција, а спремне инструкције одабране у истом позиву *doUrgencyRule(IL)* су међусобно независне (тако су биране), тако да им се редослед слободно може мењати. Уклањање непотребне *NOP* инструкције се ради само заменом редоследа инструкција у листи *W*. На овај начин, број инструкција у листи *W* се може смањити, што се може приметити на примеру листе *W* за *језгро* 0 и *језгро 1* које приказује Слика 10. У примеру, *NOP* инструкција се може уклонити из *Језгра 1*. *NOP* инструкција се најпре замени корисном инструкцијом *ADDI $9,$7,2*

```
LW $2,12($5)        LW $7,4($4)
LW $10,0($5)        NOP
ADDU $11,$2,$10     NOP
SW $11,34($0)       NOP
                    LW $11,34($0)
                    ADDI $8,$11,1
                    ADDI $9,$7,2
```

а) језгро 0 б) језгро 1

Слика 10 Исечак асемблерског кода пре замене *NOP* инструкција

```
LW $2,12($5)        LW $7,4($4)
LW $10,0($5)        ADDI $9,$7,2
ADDU $11,$2,$10     NOP
SW $11,34($0)       NOP
                    LW $11,34($0)
                    ADDI $8,$11,1
```

а) језгро 0 б) језгро 1

Слика 11 Исечак асемблерског кода након замене *NOP* инструкција *doFillRule* рутином

која је на месту прве *NOP* инструкције свакако већ спремна, односно сви регистри које она користи су већ дефинисани у датом тренутку. Изглед листа *W* за *језгро* 0 и *језгро 1* након ове оптимизације приказује Слика 11.

4.1.9 Додела регистара

Под доделом ресурса се подразумева додела физичких ресурса доступних на циљној архитектури, конкретно процесорских регитара и меморије, подацима представљених у виду IR. У овом случају се разматра само додела регистара јер су неке промењиве повезане са регистрима у секвенцијалном коду потенцијално преименоване, тј. потенцијално су уведене нове промењиве у току трансформације у SSA форму.

Насупрот многим преводиоцима који користе алгоритам бојења графа за доделу регистара, овај приступ користи линеарну доделу регистара. Линеарна додела регистара је до 70 пута бржа од конвенционалне доделе регистара [41]. Са друге стране, ефикасност овог типа доделе није много мања него у случају конвенционалног алгоритма [42]. Како би регистри могли бити додељени након направљених трансформација, неопходно је поновити анализу животног века промењивих. Та анализа се ради у једном проласку кроз све инструкције и реализована је у класи *CResourceAllocator*. Сваки основни блок представља део секвенцијалног кода, па је позиција промењиве у листи инструкција (редни број инструкције која користи промењиву) једина потребна информација, поред цене сваке инструкције.

Табела 1 показује пример трансформације кода у SSA форму, анализу животног века промењљивих и доделу регистара за дати пример. У примеру ресурс *res1* живи у све 4 инструкције. Регистар *R1* се додељује том ресурсу. Ресурс *res2* живи на инструкцијама 2 и 3 и додељен му је регистар *R2*, док је ресурсу *res3* који живи на инструкцијама 3 и 4 додељен регистар *R3*.

Сви регистри опште намене учествују у додели регистара и налазе се у скупу слободних регистара, док се остали регистри (тзв. регистри посебне намене) не додељују у току доделе регистара, нити су били део преименовања у току преласка на SSA форму. Регистри који се не смеју мењати су: регистар са константном вредношћу 0 (*$0*), привремени регистар намењен асемблеру (*$at*), регистри за повратне вредности функција (*$v0* и *$v1*), регистри за пренос параметара функцији (*$a0*, *$a1*, *$a2* и *$a3*), регистри резервисани за језгро система (*$k0* и *$k1*), глобални показивач (*$gp*), регистар повратне адресе (*$ra*), показивач на врх магацинске меморије (*$sp*) и показивач на врх оквира потпрограма (*$fp*) [37]. Промењљиве којима су додељени ови регистри (претходно обојене промењљиве) представљају ограничење у додели и другим анализама, као што је и овде случај. Операција доделе регистра некој промењљивој која почиње да живи (на месту њене дефиниције) одговара узимању елемента из скупа слободних регистара. Када се заврши живот промењљиве (последњом употребом исте) регистар који јој је био додељен се враћа у скуп слободних регистара. Јединственост регистра је гарантована коришћењем скупа регистара.

4.1.10 Генератор кода

У класи *CDumper*, реализован је једноставан генератор асемблерских кодова добијених партиционисањем полазног кода. За свако процесорско језгро, прави се одвојена датотека, која садржи одговарајући програмски блок. Нови програмски сегмент се добија спајањем свих основних блокова намењеним одговарајућем језгру, у редоследу који дефинише граф тока управљања, док се сегмент података проширује новим меморијским локацијама, односно новим литералима. Зависно од вредности аргумената прослеђених приликом позива, могуће је, поред датотеке са програмским кодом, направити и прегледну датотеку у табеларној (CSV) форми, ради јаснијег приказа додатних

информација, коришћених у анализи резултата паралелизације, односно резултујућег убрзања. Генератор кода такође рачуна и додатне информације, укључујући убрзање остварено за осваки основни блок понаособ, за функције, али и за цео програм.

Након што генератор кода направи програмски код, од одговарајућег програмског кода се прави извршна датотека за свако процесорско језгро помоћу преводиоца gcc и одговарајућих алата (*as* и *ld*).

ПОГЛАВЉЕ 5.

ЕКСПЕРИМЕНТАЛНИ РЕЗУЛТАТИ И ДИСКУСИЈА

За време експерименталне евалуације, различити тест вектори су потребни да би се верификовао овај приступ, мерило постигнуто убрзање и показало како се овај приступ понаша у општем случају. Наиме, 4 различита тест вектора су коришћена: DCT без петљи, DCT са петљама, Страсеново множење матрица и проблем *N*-краљица.

5.1 Коришћени тестови

DCT је пример који се односи на дискретну косинусну трансформацију (енг. Discrete Cosine Transformation) и за први тест је имплементиран без коришћења петљи, док је за други тест имплементиран са петљама, односно скоковима. Очекивано је да скокови доведу до стварања више мањих основних блокова (енг. basic block) и самим тим смање могуће убрзање јер је мања величина независних целина и могућност појаве независних целина уопште.

Страсенов алгоритам, назван по Волкеру Страсену је алгоритам множења матрица, бржи од класичног множења матрица и користан за велике матрице јер уноси мање зависности. Стога је очекивано да даје и добре резултате по питању убрзања након паралелизације.

Проблем *N*-краљица решава проблем постављања N краљица на таблу величине NxN. Конкретно, коришћена је имплементација 8-краљица без рекурзије, као компликован алгоритам са великом зависношћу података.

5.2 Поступак, резултати и дискусија

Валидација је урађена експерименталном евалуацијом заснованом на процени времена извршења програма и симулацијом на симулационом моделу платформе са 1, 2, 4, 8 и 16 језгара. Мера перформансе коришћена у процесу валидације је убрзање, које је дефинисано на следећи начин: нека је T_s време извршења секвенцијалног кода и T_p време извршења паралелног кода, тада је убрзање S:

$$S = T_s / T_p$$

Убрзање за сваки тест вектор и за различит број језгара приказује Табела 2. Израчунато просечно убрзање из последње колоне илуструје просечно убрзање одабраних тест вектора. На самом крају, процењене вредности, али и сама метода, верификовани су на описаној циљној архитектури са 1, 2 и 4 језгра.

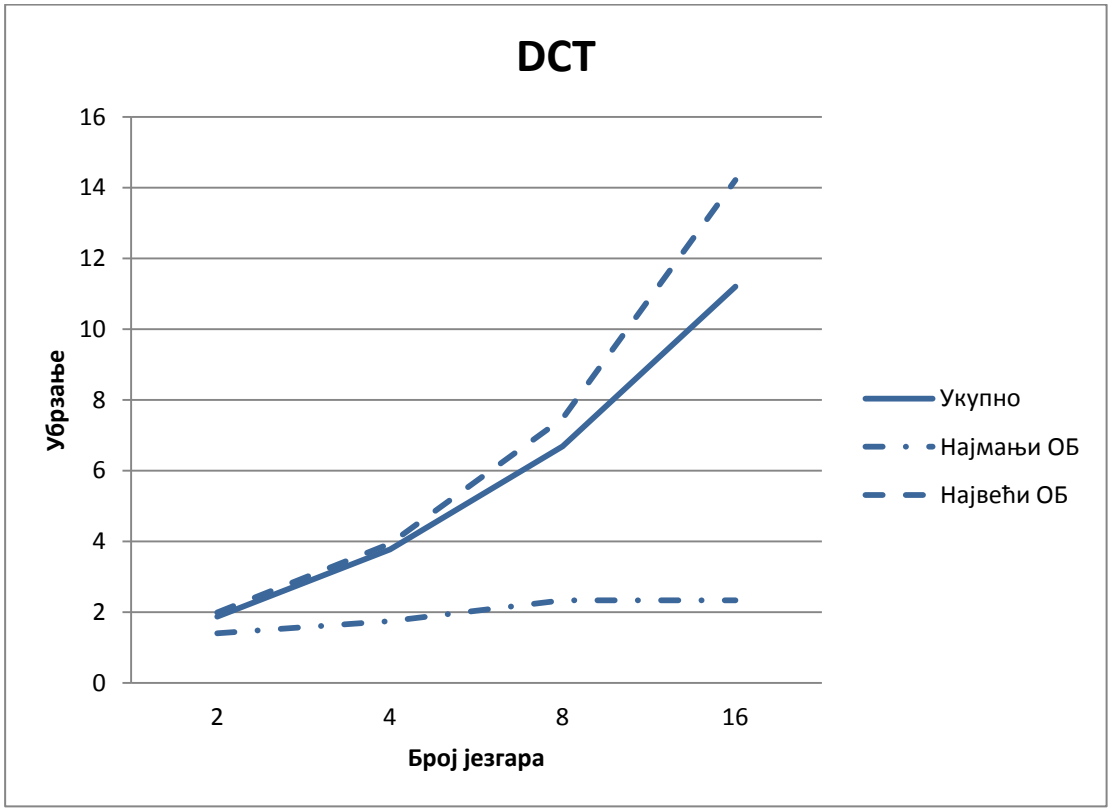
Убрзања која приказује Табела 2 представљају убрзања целог програма, за сваки пример. Њихове криве су такође приказане у сликама које следе. Да би се нагласила зависност убрзања од величине основног блока, криве убрзања су, осим за цео програм, приказане и за најмањи, односно највећи основни блок у програму. Ово је важно јер је укупно убрзање често деградирано већим бројем

Табела 2 Убрзања добијена паралелизацијом програма

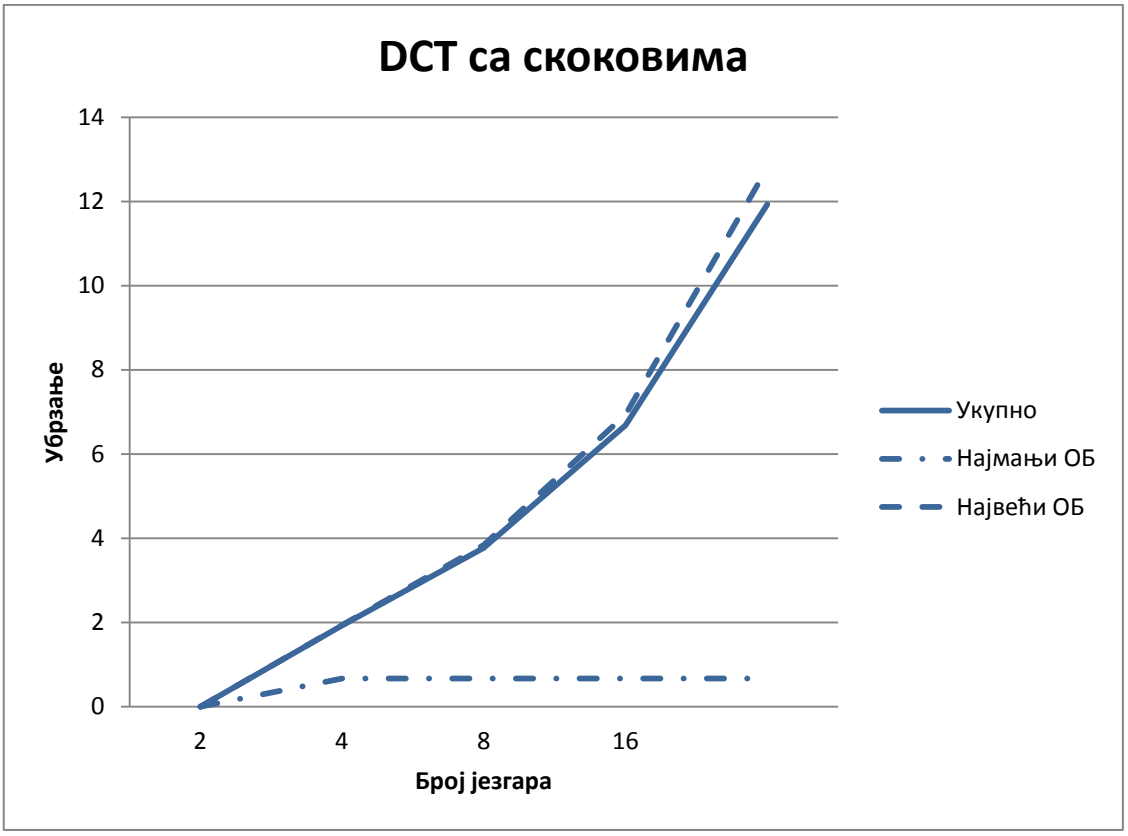
Број језгара	**Убрзање**				
	DCT	**DCT са скоковима**	**Страсенов алгоритам**	**8-краљица**	**Просечно**
2	1.87	1.93	1.85	1.36	1.75
4	3.77	3.77	3.25	1.60	3.10
8	6.68	6.68	5.13	1.70	5.05
16	11.21	11.93	6.83	1.70	7.92

малих основних блокова, који најчешће и нису део алгоритма већ само иницијализационе и деиницијализационе рутине. Такође, на датим примерима је показано да када се велики основни блокови понављају више пута (у петљи), што јесте најчешћи случај у наменским системима, утицај малих блокова постаје занемарљив и укупно убрзање се приближава убрзању највећег основног блока, односно оних блокова који се најчешће понављају. Наиме, у наменским системима често за случај имамо неке припремне рутине (иницијализацију) која се деси само једном, на почетку, и уноси почетно кашњење, које се касније не акумулира јер не утиче на главну петљу, која представља основну рутину која се обавља са периодичним понављањем и која треба, у случају система са временским ограничењима, да се завршава у задатим роковима.

Идеја која је описана даје убрзање 6.68x за 8 језгара, односно 11.21x за 16 језгара на примеру DCT. Међутим, убрзање највећег основног блока у истом примеру износи 7.47x за 8 језгара, што приказује Слика 12. Ово је за нијансу лошији резултат од оног постигнутог паралелизатором из [13], који са друге стране није погодан за паралелизацију кодова који садрже скокове, већ само за програме који имају један основни блок.



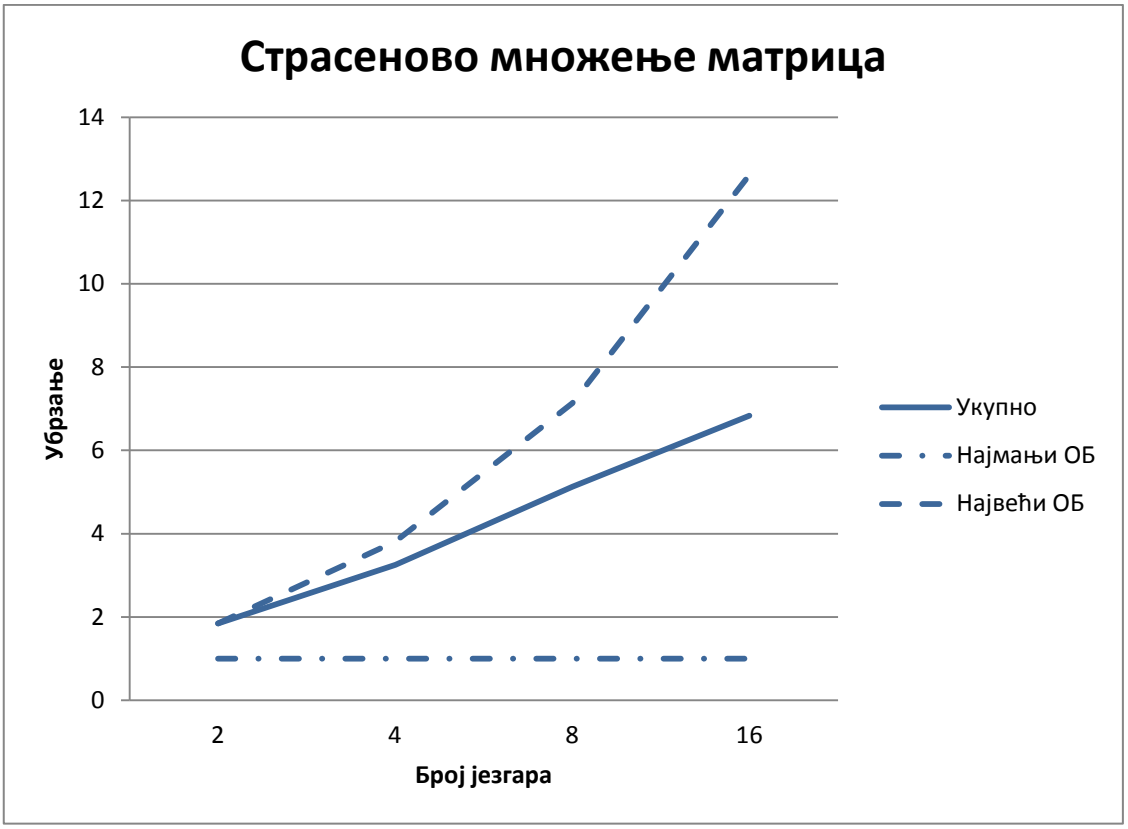
Слика 12 Анализа убрзања за DCT пример



Слика 13 Анализа убрзања за DCT пример са скоковима

За DCT пример са скоковима, упркос укљученим петљама и скоковима, основни блокови који се добију оптимизационим техникама током превођења су довољно велики. Стога, пример садржи велике основне блокове са малим зависностима, што даје скоро линеарно убрзање након паралелизације, што приказује и Слика 13. Постигнуто убрзање је такође за нијансу мање него за сличан пример у [13], али тај приступ и приступ из [14] не подржавају скокове, а самим тим ни овај пример у целости.

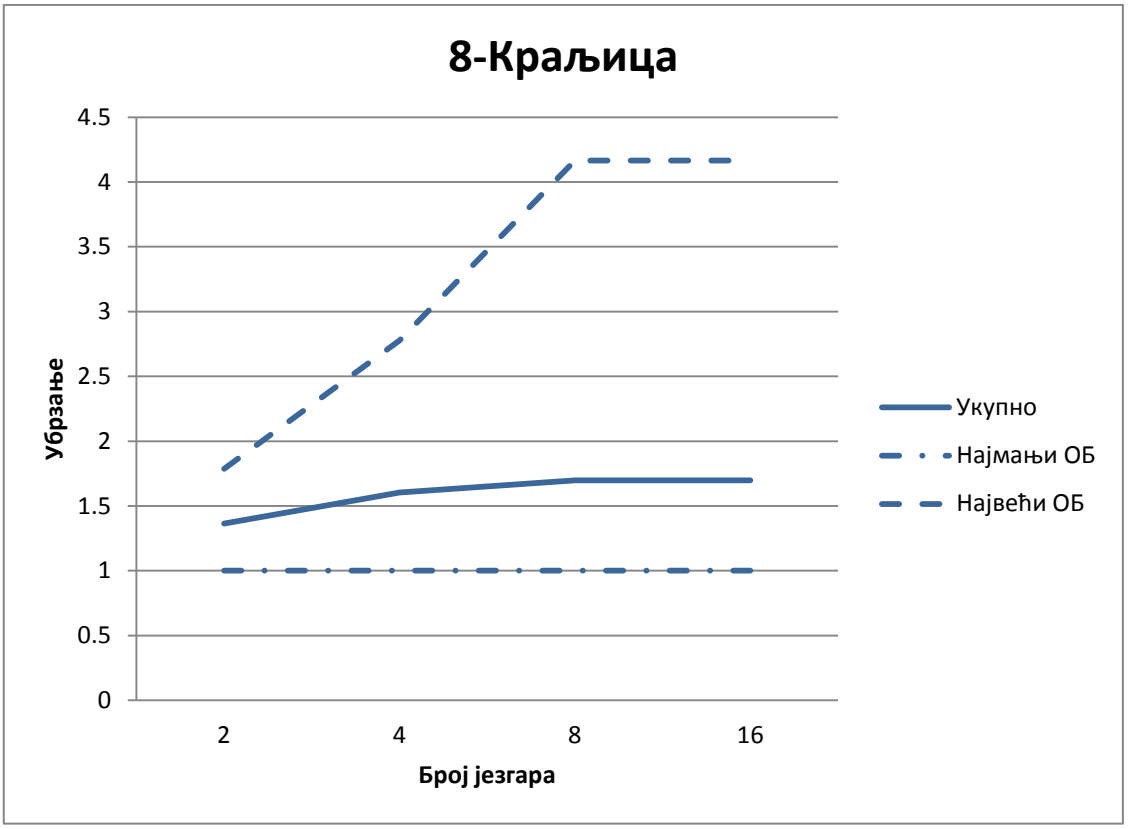
Убрзања постигнута за пример Страсеновог множења матрица на најмањем основном блоку, највећем основном блоку и укупно на целом програму показује Слика 14. Убрзање за највећи основни блок је скоро линеарно (7.13x за 8 језгара и 12.61x за 16 језгара) и то је последица мале зависности података у алгоритму множења матрица. Ипак, већи број малих основних блокова са веома малим убрзањем (нпр. најмањи основни блок има убрзање 1x) има значајан утицај на укупно убрзање које је сходно томе редуковано, што се такође види на датом графику (крива укупног убрзања има значајно мањи раст). Ипак, укупно убрзање



Слика 14 Анализа убрзања за пример Страсеновог множења матрица

убрзање за 8 језгара износи 5.13x, што је боље од убрзања постигнутог у [13] за исти пример.

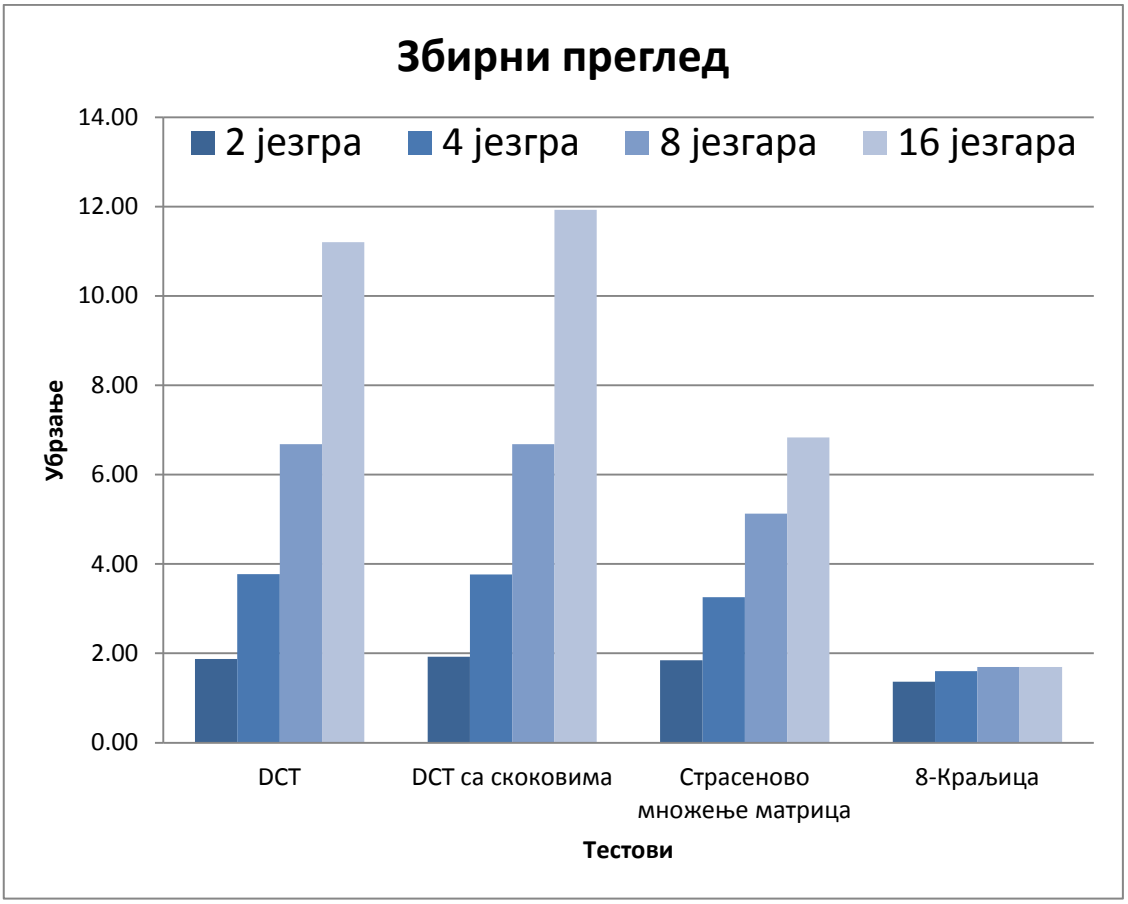
У примеру 8-краљица, најмањи основни блок садржи само једну инструкцију, па ни убрзање 1x за исти није изненађујуће. Међутим, интересантан је пораст убрзања који се добија повећањем броја језгара, када је у питању највећи основни блок, Слика 15. Повећање је приметно до 8 језгара, где износи приближно 4.16x. На том месту крива достиже максимум и улази у засићење за више од 8 језгара. То је последица постојања 8 блокова са малом зависношћу података који могу да се издвоје из основног блока предложеним алгоритмом, а што је условљено величином табле (N=8). Иако је алату понуђено више од 8 језгара (нпр. 16 језгара), направиће само 8 партиција. Прављење више од 8 партиција би деградирало убрзање јер би цена режије постала већа од добитка због пресецања скупих веза, те додавања великог броја синхронизационих инструкција. Укупно убрзање је приближно 1.7x за 8 језгара и то је последица већег броја малих основних блокова који директно деградирају праралелизам.



Слика 15 Анализа убрзања за пример 8-краљица

Упоредни приказ убрзања *S* за све тестове и различит број језгара даје Слика 16. Приметно је да се *S* повећава са повећањем броја језгара (за сваки тест вектор) до засићења и са смањивањем броја инструкција скока (нпр. 8-краљица са више скокова од DCT примера такође показује и мање убрзање).

Претходно је појашњено да је убрзање добијено над целим програмом често деградирано малим основним блоковима који најчешће нису део самог алгоритма. Са друге стране, код наменских система би се ситуација заправо могла изменити јер се код њих врло често алгоритам понавља у петљи, најчешће бесконачној, а основни део алгоритма у свим наведеним примерима јесте практично највећи основни блок за који је показано да предложено решење, скалабилно, даје добро убрзање након паралелизације. Потврда горенаведене тврдње да укупно убрзање код наменских система највише зависи од убрзања добијеним над основним блоковима који се најчешће понављају у програму, изнета је у наставку. Показан је утицај броја итерација петље на једном од обрађених примера. За пример је узето Страсеново множење матрица, али је уз већ описан основни пример, анализиран и сличан пример у којем се множење



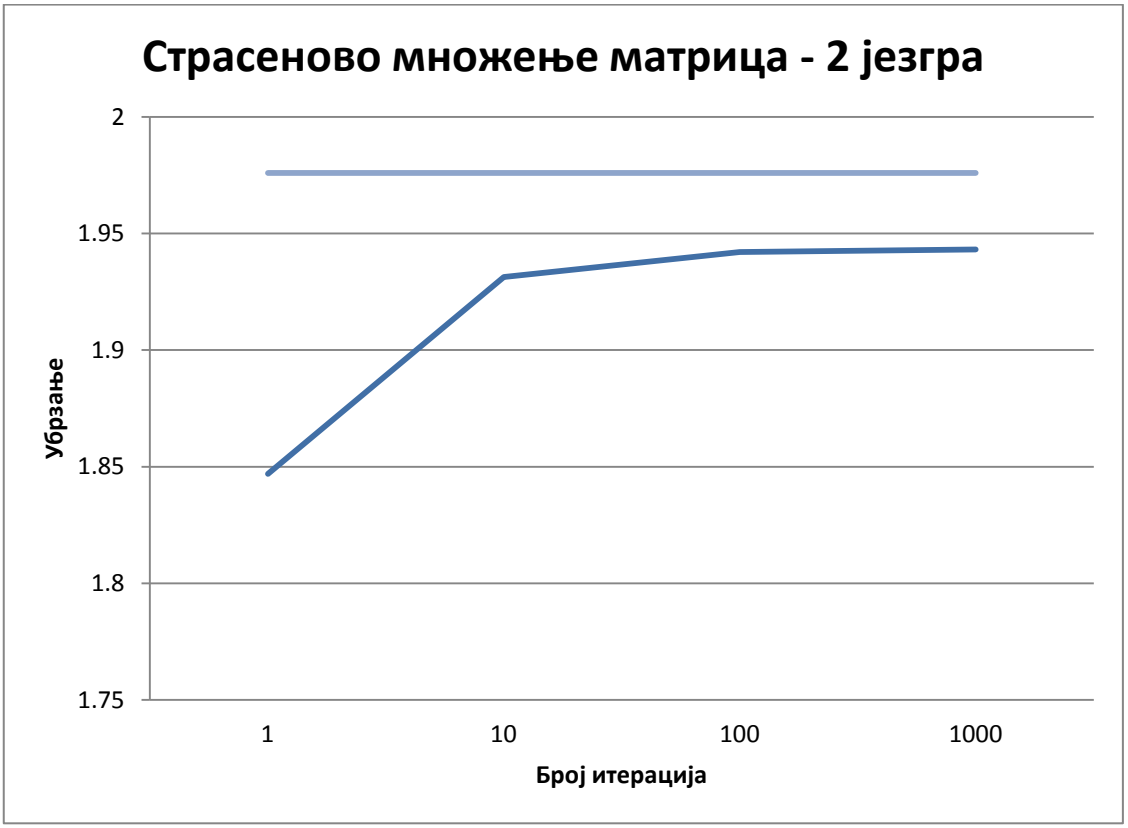
Слика 16 Упоредни приказ убрзања *S* за све тест векторе

матрица спроводи по 10, 100 и 1000 пута. Тест је спроведен на моделима процесора 1, 2, 4 и 8 и 16 језгара, а добијена убрзања приказује Табела 3. Показује се да се повећањем броја итерација главне петље, укупно убрзање приближава убрзању које се за дати број језгара добија за највећи основни блок, а које се за пример Страсеновог множења матрица може видети на графичком приказу, Слика 14.

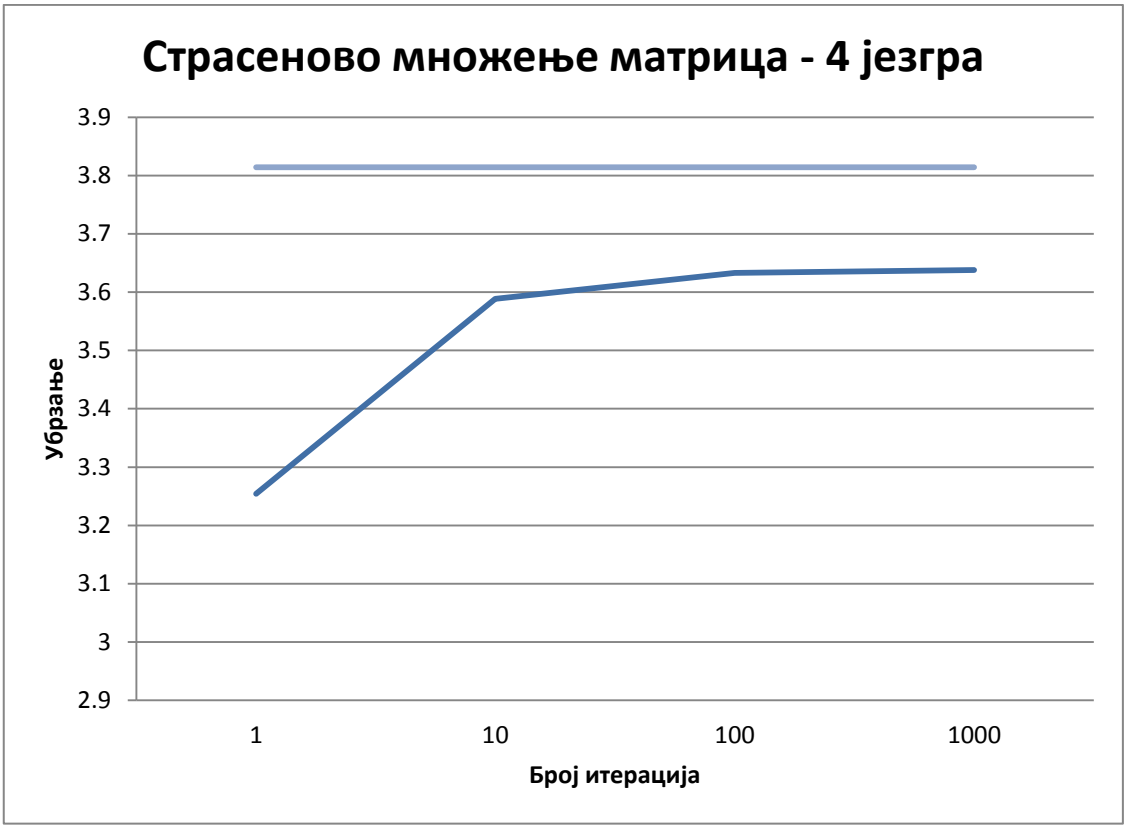
Табела 3 Резултат паралелизације Страсеновог алгоритма множења матрица са петљом

Број језгара	**Убрзање**			
	1 итерација	**10 итерација**	**100 итерација**	**1000 итерација**
2	1.85	1.93	1.94	1.94
4	3.25	3.59	3.63	3.64
8	5.13	6.16	6.32	7.33
16	6.83	9.53	10.00	10.05

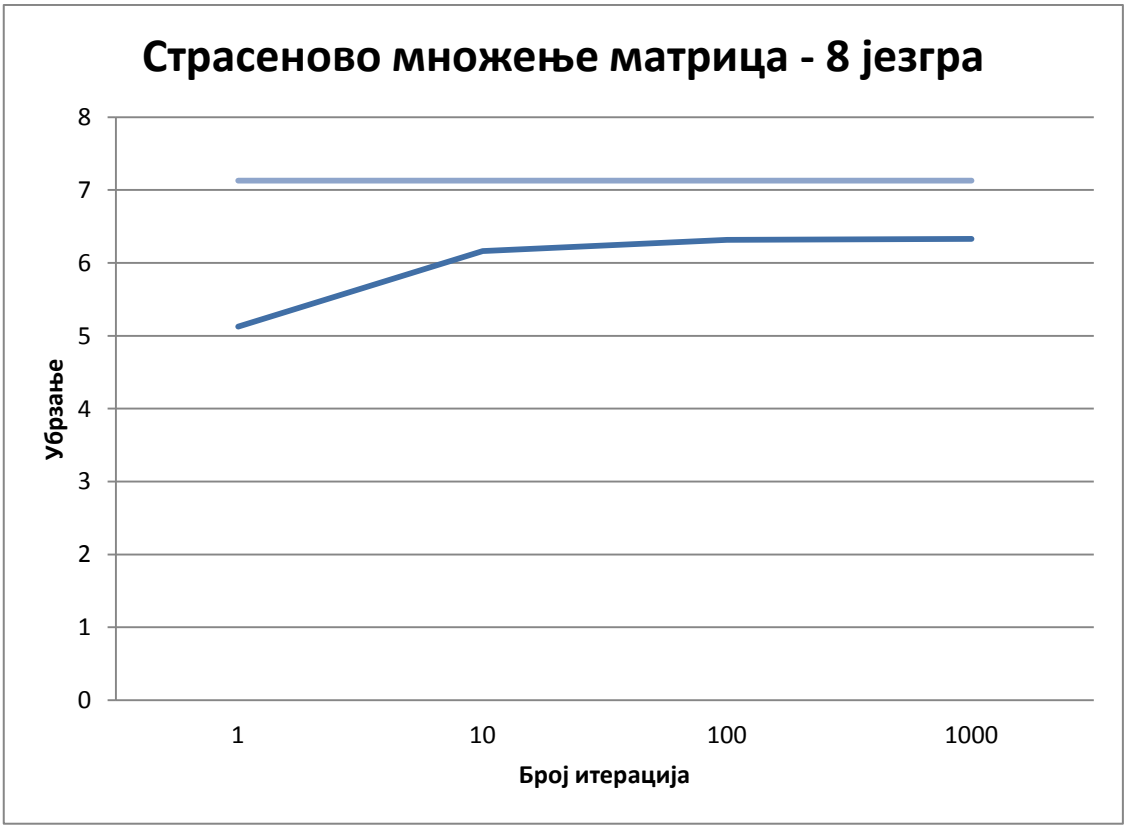
Појединачне графичке приказе са трендом убрзања услед промене броја итерација петље, те приближавање горњој граници убрзања добијеног за највећи основни блок за процесоре са 2, 4, 8 и 16 језгара приказују Слика 17, Слика 18, Слика 19 и Слика 20, тим редом. На графичким приказима се може приметити нагли раст убрзања услед повећања броја итерација и да већ за 100 итерација улази у засићење, уз мало повећање за 1000 итерација. Такође приметно је да се не достиже граница убрзања добијеног за највећи основни блок, већ мало раније улази у засићење, а разлог томе је што се у оваквој петљи која понавља неки алгоритам, најчешће осим основних блокова са алгоритмом, понавља и један или више мањих основних блокова који представљају петљу, позив функције алгоритма и сл. Ти мањи основни блокови, који се најчешће не могу ни паралелизовати већ се практично у изворном (секвенцијалном) облику извршавају на свим језгрима, имају приметан утицај на укупно убрзање, а утицај је све већи како се повећава број језгара, па је тако највећи за процесор са 16 језгара где они почињу да доминирају. Ипак, показано је да би у сценарију уобичајеном за наменске системе, убрзање било још боље од оног добијеног у првим тестовима.



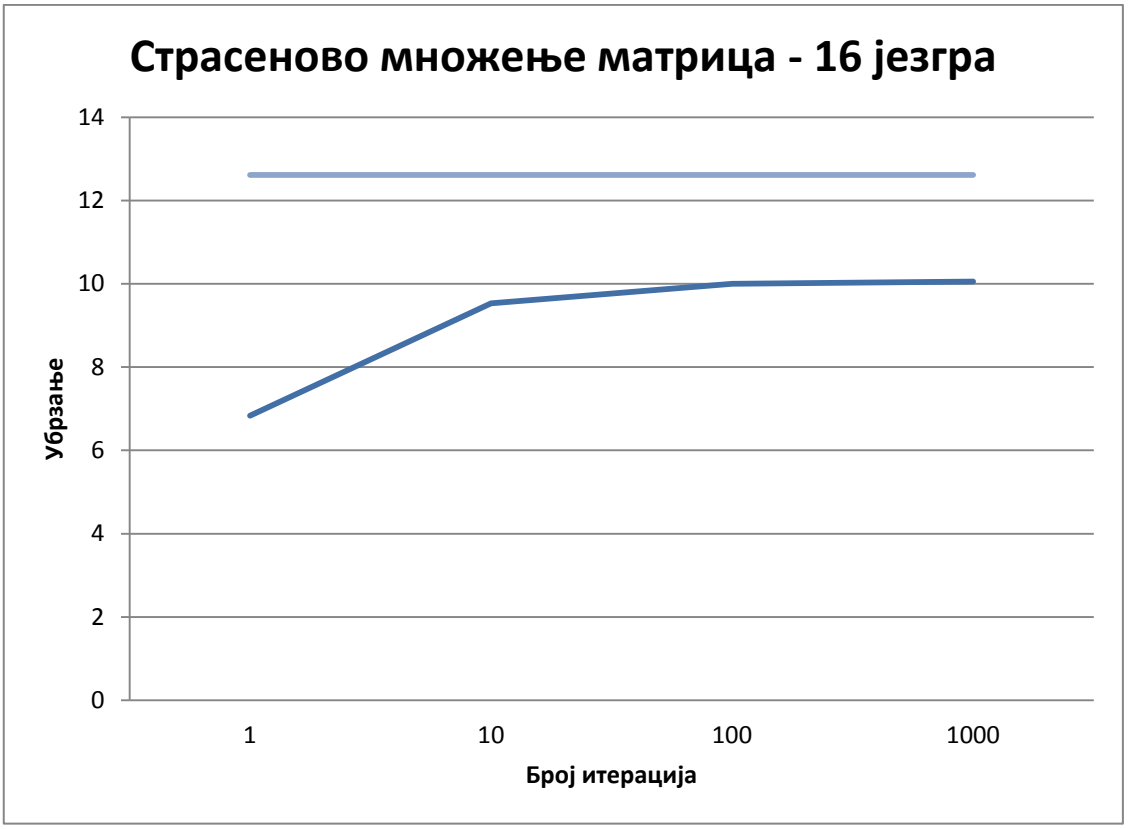
Слика 17 Страсенов алгоритам на 2 језгра – зависност убрзања од броја понављања



Слика 18 Страсенов алгоритам на 4 језгра – зависност убрзања од броја понављања



Слика 19 Страсенов алгоритам на 8 језгара – зависност убрзања од броја понављања



Слика 20 Страсенов алгоритам на 16 језгара – зависност убрзања од броја понављања

ПОГЛАВЉЕ 6.

ЗАКЉУЧАК И БУДУЋИ РАД

Ово истраживање требало је да да одговор на основну претпоставку да је могуће пронаћи решење односно алат за паралелизацију постојећег секвенцијалног машинког кода, који без уплитања програмера (аутоматски) ствара програме који се извршавају паралелно на више језгара вишејезгарног процесора, смањујући време извршења уравнотеженим оптерећењем процесора. Судећи по претходно описаном решењу, његовим детаљима и добијеним резултатима могу се извести основни закључци и потврдити да истраживање даје потврдан одговор на дато питање.

Показано је да је паралелизација асемблерског кода актуална тема [14]. Конкретно, [14] препознаје методу понуђену у [30] и [31] као погодну за примену на паралелизацију асемблерског кода. Природно, додатно истраживање је било неопходно, али се на већ доказану праксу може ослонити.

У овој докторској дисертацији приказан је приступ аутоматској паралелизацији асемблерског кода унапређеном верзијом алгоритма за партиционисање графа заснованој на METIS алгоритму. Ограничен је на статичко распоређивање у време превођења (енг. compile-time scheduling) или у време покретања (енг. just-in-time scheduling) извршне датотеке на свако језгро. Штавише, омогућава оптимално, односно уравнотежено оптерећење по језгрима, које даје значајно убрзање што је показано на одабраним тест векторима.

Просечно убрзање од 7.92x је постигнуто за 16-језгарни процесор. У неким случајевима, достигнуто је и убрзање од 14x за исти процесор. Решење је скалабилно, што је показано на моделима процесора са 2, 4, 8 и 16 језгара и показано је да најбоље резултате даје за веће основне блокове унутар тест вектора. Осим тога, приступ је верификован на описаној циљној архитектури са 2 и са 4 језгра, што је потврдило тачност резултата добијених проценом на моделу перформансе.

Прилог аутоматској паралелизацији из ове докторске дисертације може бити користан истраживачима и инжењерима из области паралелизације, као основа за даље оптимизације, као задњи део преводиоца или као алат за паралелизацију кода за наменски систем попут циљног система коришћеног у истраживању.

Алат се тренутно развија у две путање: (1) побољшање постојећег оптимизујућег C преводиоца за DSP платформе [43] и (2) развој засебног алата за вишејезгарне наменске RISC архитектуре (нпр. MIPS, MicroBlaze, итд.), који би могао бити коришћен као алат након превођења C кода (нпр. помоћу gcc-а). Обе путање разматрају примену описаног приступа у облику потпуног и независног алата за паралелизацију наменских система са тврдим временским ограничењима.

У претходним поглављима су на погодним местима поменути неки могући и планирани правци даљег истраживања и усавршавања предложеног и описаног решења. Даље истраживање на ову тему је планирано у два правца. Први правац је валидација решења на вишејезгарним системима, сличним систему имплементираном унутар FPGA, са 16 или више језгара. Други правац ће бити унапређење препознавања паралелних петљи. Тренутно је подржана паралелизација тела петље (тзв. DOACROSS parallelism) који јесте идеалан за петље које немају независне итерације. Анализа петљи са независним итерацијама слична оној коју описују аутори [15] и [16], укључујући већ имплементирану паралелизацију петљи засновану на подели итерационог простора (тзв. DOALL parallelism), планирана је као први следећи корак у циљу унапређења описаног алата.

Литература

[1] L. Hochstein, J. Carver, F. Shull, S. Asgari, V. Basili, "Parallel programmer productivity: A case study of novice parallel programmers," in *in SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, Washington, DC, USA, 2005.

[2] M. Popovic, M. Djukic, V. Marinkovic, N. Vranic, "On task tree executor architectures based on Intel parallel building blocks," *Computer Science and Information Systems*, vol. 10, no. 1, pp. 369-392, 2013. doi: 10.2298/CSIS120519008P.

[3] J. Reinders, Intel Thread Building Blocks, 1005 Gravenstein Highway North, Sebastopol, CA, USA: O'Relly Media Inc., 2007.

[4] A. Kukanov, M. Voss, "The foundations for scalable multi-core software in Intel Threading Building Blocks," *Intel Technology Journal*, vol. 11, no. 4, pp. 309-322, 2007.

[5] K. Randall, *Cilk: Efficient multithreaded computing*, Cambridge, MA, USA: Ph.D. dissertation, Massachusetts Institute of Technology, 1998.

[6] R. Blumofe, C. Leiserson, "Scheduling multithreaded computations by work stealing," *Journal of the ACM*, vol. 46, no. 5, pp. 720-748, 1999.

[7] R. Chandra, R. Menon, L. Dagum, D. Kohr, D. Maydan, J. McDonald, Parallel programming in OpenMP, 525 B Street, San Francisco: Academic press, 2001.

[8] D. Kirk, W. Hwu, Programming massively parallel processors, 30 Corporate Drive, Suite 400, Burlington: Mogran Kaufmann Publishers, 2010.

[9] A. Bhattacharjee, G. Contreras, M. Martonosi, "Parallelization Libraries: Characterizing and Reducing Overheads," *ACM Trans. Archit. Code Optim*, vol. 8, no. 1, pp. 5:1-5:29, 2011.

[10] X. Wang, S. Thota, "A resource-efficient communication architecture for chip multiprocessors on FPGAs," *Journal of Computer Science and Technology*, vol. 26, no. 3, pp. 434-447, 2011.

[11] U. Vishkin, "Is multicore hardware for general-purpose parallel processing

Broken?," *Communications of the ACM*, vol. 57, no. 4, pp. 35-39, 2014.

[12] A. Kotha, K. Anand, M. Smithson, G. Yellareddy, R. Barua, "Automatic Parallelization in a Binary Rewriter," in *In Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '43)*, Washington, DC, USA, 2010.

[13] N. Vranic, V. Marinkovic, M. Djukic, M. Popovic, "An approach to parallelization of sequential C code," in *Engineering of Computer Based Systems (ECBS-EERC), 2011 2nd Eastern European Regional Conference on the*, Bratislava, 2011.

[14] D. Kovacevic, M. Stanojevic, V. Marinkovic, P. M., "A solution for automatic parallelization of sequential assembly code," *Serbian Journal of Electrical Engineering*, vol. 10, no. 1, pp. 91-101, 2013.

[15] K. Kyriakopoulos, K. Psarris, "Data dependence analysis techniques for increased accuracy and extracted parallelism," *International Journal of Parallel Programming (IJPP)*, vol. 32, no. 4, pp. 317-359, 2004.

[16] K. Kyriakopoulos, K. Psarris, "Non-linear symbolic analysis for advanced program parallelization," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 20, no. 5, pp. 623-640, 2009.

[17] G. Ottoni, R. Rangan, A. Stoler, D. I. August, "Automatic thread extraction with decoupled software pipelining," in *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 38)*, Washington, 2005. doi: 10.1109/MICRO.2005.13.

[18] S. Campanoni, T. Jones, G. Holloway, V. J. Reddi, G. Y. Wei, D. M. Brooks, "HELIX: Automatic Parallelization of Irregular Programs for Chip Multiprocessing," in *Proceedings of the Tenth International Symposium on Code Generation and Optimization (CGO '12)*, San Jose, 2012. doi: 10.1145/2259016.2259028.

[19] J. Birch, K. Psarris, "Discovering maximum parallelization using advanced data dependence analysis," in *Proceedings of the 10th IEEE International Conference on High Performance Computing and Communications (HPCC 2008)*, DaLian, China, 2008.

[20] C. Dave, H. . Bae, S. Min, S. Lee, R. Eligenmann, S. Midkiff, "Cetus: A source-to-source compiler infrastructure for multicores," *Computer*, vol. 42, no. 12, pp. 36-42, 2009.

[21] M. Mathews, J. P. Abraham, "Automatic Code Parallelization with OpenMP task constructs," in *Proceedings of the 2016 International Conference on Information Science (ICIS '16)*, Kochi, 2016. doi: 10.1109/INFOSCI.2016.7845333.

[22] E. Yardimci, M. Franz, "Dynamic parallelization and mapping of binary executables on hierarchical platforms," in *In Proceedings of the 3rd conference on Computing frontiers (CF '06)*, New York, NY, USA, 2006.

[23] W. Liu, J. Tuck, L. Ceze, W. Ahn, K. Strauss, J. Renau, J. Torrellas, "POSH: a TLS compiler that exploits program structure," in *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '06)*, New York, 2006. doi: 10.1145/1122971.1122997.

[24] H. Kim, N. P. Johnson, J. W. Lee, S. A. Mahlke, D. I. August, "Automatic speculative DOALL for clusters," in *Proceedings of the Tenth International Symposium on Code Generation and Optimization (CGO '12)*, San Jose, 2012. doi: 10.1145/2259016.

[25] N. P. Johnson, H. Kim, P. Prabhu, A. Zaks, D. I. August, "Speculative separation for privatization and reductions," *SIGPLAN Notices - PLDI '12*, pp. 359-370, 2012. doi: 10.1145/2345156.2254107.

[26] T. Oh, S. R. Beard, N. P. Johnson, S. Popovych, D. I. August, "A Generalized Framework for Automatic Scripting Language Parallelization," in *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques (PACT '17)*, Portland, 2017. doi: 10.1109/PACT.2017.28.

[27] C. Wang, X. Li, J. Zhang, X. Zhou, X. Nie, "MP-Tomasulo: A Dependency-Aware Automatic Parallel Execution Engine for Sequential Programs," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 10, no. 2, pp. 9:1-9:26, 2013. doi: 10.1145/2459316.2459320.

[28] Y. Dou, J. Zhou, G.-M. Wu, J.-F. Jiang, Y.-W. Lei, S.-C. Ni, "A unified co-processor architecture for matrix decomposition," *Journal of Computer Science and Technology*, vol. 25, no. 4, pp. 874-885, 2010.

[29] M. Dali, A. Guessoum, R. M. Gibson, A. Amira, N. Ramzan, "Efficient FPGA Implementation of High-Throughput Mixed Radix Multipath Delay Commutator FFT Processor for MIMO-OFDM," *Advances in Electrical and Computer Engineering,* vol. 17, no. 1, pp. 27-38, 2017. doi: 10.4316/AECE.2017.01005.

[30] D. Capko, A. Erdeljan, M. Popovic, S. G., "An optimal initial partitioning of large data model in utility management systems," *Advances in Electrical and Computer Engineering,* vol. 11, no. 4, pp. 41-46, 2011.

[31] D. Capko, A. Erdeljan, G. Svenda, P. M., "Dynamic repartitioning of large data model in distribution management systems," *Electronics and Electrical Engineering: System Engineering, Computer Technology*, vol. 4, no. 120, pp. 83-88, 2012.

[32] G. Karypis, V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM Journal of Scientific Computing*, vol. 20, no. 1, pp. 359-392, 1998.

[33] A. Hormati, Y. Choi, M. Kudlur, R. Rabbah, T. Mudge, S. Mahlke, "Flextream: Adaptive compilation of streaming applications for heterogeneous architectures," in *The 18th Int. Conf. on Parallel Arch. and Compilation Techn.*, Raleigh, North Carolina, 2009.

[34] W. Amme, P. Braun, F. Thomasset, E. Zehendner, "Data dependence analysis of assembly code," *International Journal of Parallel Programming*, vol. 28, no. 5, pp. 431-467, 2000.

[35] G. Matheou, P. Evripidou, "Verilog-based simulation of hardware support for data-flow concurrency on multicore systems," in *Proceedings - 2013 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation, IC-SAMOS 2013*, Samos, 2013.

[36] A. Aho, M. Lam, R. Seth, J. Ullman, Compilers, principles, techniques & tools, second edition, Pearson Education, Inc, 2007.

[37] D. Sweetman, See MIPS run second edition, The Morgan Kaufmann Series in Computer Architecture and Design, 2007.

[38] A. Bernstein, "Analysis of programs for parallel processing," *IEEE Transactions on Electronic Computers,* Vols. EC-15, no. 5, pp. 757-763, 1966.

[39] S. Debray, R. Muth, M. Weippert, Alias analysis of executable code, Tucson, U.S.A: Department of Computer Science, University of Arizona, 1996.

[40] G. Karypis, V. Kumar, Multilevel k-way hypergraph partitioning, Minneapolis, MN, Department of Computer Science and Engineering, Army HPC Research Center, University of Minnesota.

[41] C. Wimmer, F. M., Linear scan register allocation on SSA Form, Irvine: Department of Computer Science University of California.

[42] M. Puletto, V. Sarkar, "Linear Scan Register Allocation," *ACM Transactions on Programming Languages and Systems*, vol. 21, no. 5, pp. 895-913, 1999.

[43] M. Djukic, M. Popovic, N. Cetic, I. Povazan, "Embedded Processor Oriented Compiler Infrastructure," *Advances in Electrical and Computer Engineering*, vol. 14, no. 3, pp. 123-130, 2014. doi: 10.4316/AECE.2014.03016.

Биографија

Владимир Б. Маринковић је рођен 2. марта 1986. године у Новом Саду. Школовао се у Новом Саду. Факултет техничких наука у Новом Саду уписао је 2005. године. Дипломирао је 2010. године на смеру Рачунарска техника и рачунарске комуникације са просечном оценом 9.88 (девет и 88/100), одбранивши дипломски рад "Једно решење физичке архитектуре за оцену квалитета видео садржаја високе дефиниције у реалном времену", са оценом 10. Докторске студије је уписао школске 2010/2011 године на истом факултету, на смеру Електротехника и рачунарство, студијски програм Рачунарство и аутоматика.

Истраживачки интерес Владимира Маринковића усмерен је ка паралелизацији програма за извршавање на вишејезгарним процесорима и вишепроцесорским системима, као и на програмске преводиоце за овакве системе.

Био је стипендиста Министарства за науку и технолошки развој од школске 2010/2011 до школске 2014/2015. године. У 2011. години је изабран у звање Истраживач сарадник на Истраживачко-развојном институту РТ-РК за системе засноване на рачунарима. 2015. године је изабран у звање Асистент мастер на Факултету техничких наука у Новом Саду. Од школске 2010. године активно учествује у извођењу наставе на предметима Системска програмска подршка у реалном времену 1 и 2, Архитектуре и алгоритми ДСП 1 и Линукс програмирање у реалном времену.

Аутор је или коаутор 2 рада у истакнутим међународним часописима, једног рада у часопису међународног значаја, 19 радова на међународним конференцијама и 9 радова на националним конференцијама. Коаутор је 4 техничка решења, као и 2 патента.

Од страних језика течно говори енглески.