УНИВЕРЗИТЕТ У НОВОМ САДУ

ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА

Миодраг Ђукић

Ново решење компајлерске инфраструктуре за наменске процесоре

– ДОКТОРСКА ДИСЕРТАЦИЈА –

Ментор:

проф. др Мирослав Поповић

Нови Сад, 2014

Највећу захвалност дугујем ментору проф. др Мирославу Поповићу за сву помоћ, мотивацију, стрпљење и правилно усмеравање током рада на овој дисертацији.

Захваљујем се и доц. др Јелени Ковачевић за истрајну личну и стручну подршку, као и свесрдну логистичку помоћ при контактима са индустријом.

Посебно се захваљујем Радовану Обрадовићу за инспирацију и ширење видика, а мр Зорану Јовановићу за помоћ и увођење у професионалне аспекте развоја компајлера.

Значајну захвалност изражавам и Ненаду Четићу за честе и корисне консултације и сарадњу.

Захвалан сам и др Немањи Лукићу за дискусију и савете у вези са објективним мерама кода.

Како прављење компајлера представља захтеван тимски рад, захваљујем се и свим изузетним појединцима који су допринели стварању и потврђивању производа који је тема ове дисертације: Зорану Зарићу, Марку Крњетину, Ивану Поважану, Борису Спасојевићу, Момчилу Крунићу и Дејану Бокану.

На крају, велико хвала мојој породици чија подршка ми је била најзначајнија.

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**:		Миодраг Ђукић
Ментор, **МН**:		проф. др Мирослав Поповић
Наслов рада, **НР**:		Ново решење компајлерске инфраструктуре за наменске процесоре
Језик публикације, **ЈП**:		Српски
Језик извода, **ЈИ**:		Српски
Земља публиковања, **ЗП**:		Република Србија
Уже географско подручје, **УГП**:		Војводина
Година, **ГО**:		2014.
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)		7 поглавља / 74 стране / 47 цитата / 9 табела / 29 слика / 1 прилог
Научна област, **НО**:		Електротехника и рачунарство
Научна дисциплина, **НД**:		Рачунарска техника
Предметна одредница/Кqучне речи, **ПО**:		Дигитални сигнал процесор, наменски процесори, програмски алати, компајлери
УДК		
Чува се, **ЧУ**:		У библиотеци Факултета техничких наука, Нови Сад
Важна напомена, **ВН**:		
Извод, **ИЗ**:		Ова докторска теза описује и анализира приступ развоју Це компајлера за наменске процесоре. Такав компајлер захтева имплементацију посебних техника и алгоритама, претежно специфичних за нерегуларне процесорске архитектуре, да би генерисао ефикасан код, и при том је потребно да испуњава индустријске стандарде по питању робустности, разумљивости кода, могућности одржавања и проширивости. У ту сврху је предложена нова компајлерска инфраструктура над којом је имплементиран компајлер за Cirrus Coyote 32 ДСП. Квалитет генерисаног кода поређен је са квалитетом кода генерисног од стране већ постојећег компајлера за тај процесор. Уједно, одређени елементи организације компајлера су упоређени са популарним компајлерима отвореног кода GCC и LLVM.
Датум прихватања теме, **ДП**:		21.07.2014.
Датум одбране, **ДО**:		
Чланови комисије, **КО**:	Председник:	др Владимир Ковачевић, проф. емеритус
	Члан:	др Јован Ђорђевић, ред. проф.
	Члан:	др Миодраг Темеринац, ред. проф.
	Члан:	др Никола Теслић, ред. проф.
	Члан, ментор:	др Мирослав Поповић, ред. проф.

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**:		Miodrag Djukic
Mentor, **MN**:		Miroslav Popovic, PhD
Title, **TI**:		Novel solution for compiler infrastructure for embedded processors
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**:		2014.
Publisher, **PB**:		Author's reprint
Publication place, **PP**:		Novi Sad, Dositeja Obradovica sq. 6
Physical description, **PD**: (chapters/pages/ref./tables/pictures/graphs/appendixes)		7 chapters / 74 pages/ 47 references / 9 tables / 29 pictures / 1 appendix
Scientific field, **SF**:		Electrical Engineering
Scientific discipline, **SD**:		Computer Engineering, Engineering of Computer Based Systems
Subject/Key words, **S/KW**:		Digital signal processor, embedded system, software tools, compilers
UC		
Holding data, **HD**:		The Library of Faculty of Technical Sciences, Novi Sad, Serbia
Note, **N**:		
Abstract, **AB**:		This PhD thesis describes and analyses an approach to development of C language compiler for embedded processors. That kind of compiler requires implementation of special techniques and algorithms, mostly specific for irregular processor architectures, in order to be able to generate efficient code, whereas still meeting industrial strength standard by beeing robust, understandable, maintainable, and extensible. For this purpose the new compiler insfrastructure is proposed and on top of it a compiler for Cirrus Logic Coyote 32 DSP is built. Quality of the code generated by that compiler is compared with code generated by the previous compiler for the same processor architecture. Some elements of the compiler design are also compared to popular open source compilers GCC and LLVM.
Accepted by the Scientific Board on, **ASB**:		21.07.2014.
Defended on, **DE**:		
Defended Board, **DB**:	President:	dr Vladimir Kovačević, Professor Emeritus
	Member:	dr Jovan Đorđević, Professor
	Member:	dr Miodrag Temerinac, Professor
	Member:	dr Nikola Teslić, Professor
	Member, Mentor:	dr Miroslav Popovic, Professor

Menthor's sign

Садржај

Списак слика

Списак табела

СКРАЋЕНИЦЕ

МР	Међурепрезентација
ДСП	Дигитални сингал процесор
MAC	Multiply-accumulate
HD	High definition
ILP	Instruction level parallelism

ПОГЛАВЉЕ 1.

Увод

Када се у рачунарству говори о квалитету процесора, његове перформансе се морају ставити у контекст намене, трошкова производње и употребе. Трошкови развоја процесора састоје се од сразмерног дела и једноструког дела. Сразмерни део представља цена производње једног процесора, док једноструки део представља трошкове развоја, који су исти без обзира на број произведених процесора. Најчешћи мотив за прављење наменских процесора је могућност да се направи довољно добар процесор за одређену намену, али који је мањи, јефтинији за произвоњу и троши мање електричне енергије. Развој наменских процесора подразумева развој новог процесорског језгра, то јест нове процесорске архитектуре и због тога је скупљи од развоја процесора базираног на неком већ постојећем језгру (што је далеко чешћи случај код процесора опште намене). Значајан део трошкове развоја нове архитектуре представља развој системске програмске подршке, пре свега развојних алата. Са друге стране, иако мањи, наменски процесори успевају да одговоре на захтеве планиране употребе тако што уводе специјализоване елементе у физичку архитектуру. Ти специјализовани елементи су оријентисани ка томе да омогуће што ефикасније извршавање очекиваних програма, док се лакоћа програмирања за ту архитектуру ставља у други план. Последица тога је да писање ефикасних програма (који потпуно искоришћују могућности које процесор нуди) за наменске процесоре захтева од програмера далеко већу пажњу и способност. Тај

захтев се не односи само на људе програмере, него и на генераторе кода, пре свега компајлере који преводе програме са виших програмских језика. То доводи до оваквог следа: јефтинији процесор за задату намену има посебнију архитектуру; посебнија архитектура захтева већи напор од програмера, па је програмирање у асемблеру скупље, као и прављење преводиоца са неког вишег програмског језика, што диже цену развоја системске програмске подршке и писања програма, што на крају умањује почетну предност процесора у цени. Због тога је важно да се системски алати, а посебно компајлер, као најсложенији алат, могу направити брзо и поуздано, уз висок квалитет рада.

Класични компајлерски приступи не дају добре резултате код наменских система. Модерни правци стављају акценат на могућност лаког прилагођавања компајлера на различите архитектуре, али се мање пажње придаје компајлерским поступцима који доприносе прављењу бољег кода и начинима њихове имплементације, поготово за специјлизоване архитектуре које карактеришу наменске системе. Због тога се многи наменски процесори и даље програмирају ручним писањем асемблерског кода, [Fisher]. Током последње две деценије истраживања у области посебних компајлерских техника и алгоритама за наменске процесоре проширила су се знања о томе како да се боље компајлира код у тим случајевима. Међутим, од истраживања до индустријске примене потребно је начинити додатан корак, тако да компајлер који би могао бити у професионалној употреби мора још да буде и робустан, разумљив, проширив и лак за одржавање (примећено и у [Hamel]).

Неке од типичних посебности архитектуре за наменске процесоре јављају се код процесора за обраду дигиталних сигнала (ДСП – дигитални сигнал процесори). Те посебности архитектуре обухватају аритметику у непокретном зарезу, Харвард архитектуру, неортогонални скуп инструкција са израженом проточном структуром или хетерогеним паралелизмом на нивоу инструкција, адресне генераторе, физички подржане петље, физички подржан стек позива потпрограма, као и неке специјализоване инструкције (као што је MAC, енг. Mulitpy-ACcumulate).

Хипотеза ове докторске дисертације је да нова компајлерска инфраструктура која ће у дисертацији бити описана омогућава лакше

изражавање компајлерских поступака каратеристичних за наменске архитектуре, обезбеђујући уједно потребну робустност, проширивост и лакоћу одржавања. Прављење овакве инфраструктуре заснива се на истраживању у области програмских алата за наменске процесоре и искуству у прављењу компајлера индустријског квалитета. Као прва циљна платформа за предложну инфраструктуру изабран је дигитални сигнал процесор фирме Cirrus Logic, под назовм Coyote 32. За овај процесор већ постоји компајлерски преводилац са Це програмског језика и он ће се у овој дисертацији називати *полазни компајлер*. Тај компајлер је заснован на инфраструктури за процесоре опште намене, уз само неколико прилагођења за наменске процесоре. Прављењем новог компајлера, који је заснован на новој компајлерској инфраструктури, и његовим поређењем са полазним компајлером показаће се делотворност предложене инфраструктуре. Очекује се бољи резултат компајлера базираног на новој инфраструктури. Уједно се очекује и лакши рад са новом инфраструктуром.

Нова инфраструктура је резултат сталног рада на овој проблематици на Одсеку за рачунарску технику и рачунарске комуникације, Факултета техничких наука у Новом саду, и РТ-РК институту, а посебно током последњих пет година. У вези са појединачним аспектима инфраструктуре и компајлером заснованим на њој, објављено је током претходних пет година више радова на домаћим и међународним конференцијама, као и у међународним часописима. Ова дисертација сумира досадашњи учинак и даје преглед мотива, откривених проблема, изабраних решења и резултата који се тичу овог напора. У том смислу дисертација представља проширење рада [Djukic2], у којем је први пут изложена цела инфраструктура и урађено укупно вредновање компајлера заснованог на њој.

У наредном поглављу дат је преглед стања у области и описане су кључне особине компајлирања за наменске процесоре које га чине посебним у односу на компајлирање за процесоре опште намене и тиме га издвајају као посебну тему за истраживање. Затим је, у поглављу 3, детаљније описана физичка архитектура Coyote 32 процесора, за који је написан компајлер коришћењем инфраструктуре предложене у овој дисертацији. У поглављу 4 дат је преглед полазног компајлера, са посебним освртом на његова ограничења и проблеме који су се

јављали у његовом одржавању. Поглавље 5 је централно и даје детаљан опис нове компајлерске инфраструктуре и компајлера који је на њој заснован. Главни елементи инфраструктуре су објашњени, као и на који начин они олакшавају развој компајлера или побољшавају квалитет кода који компајлер генерише. Анализа перформанси новог компајлера и њихово поређење са перформансама полазног компајлера, дато је у поглављу 6. Уједно, приказане су неке основне мере сложености изворног кода полазног компајлера, новог компајлера и LLVM компајлера. Напослетку, у поглављу 7, сумирани су главни закључци и изложени даљи правци истраживања, од којих су неки већ започети.

ПОГЛАВЉЕ 2.

СТАЊЕ У ОБЛАСТИ И ПОСЕБНОСТИ КОМПАЈЛИРАЊА ЗА НАМЕНСКЕ ПРОЦЕСОРЕ

Досадашња истраживања у области посебних компајлерских техника и алгоритама за наменске процесоре донела су бројна сазнања о могућим начинима за побољшање компајлера у том смеру. Правци истраживања су се поставили према посебностима компајлирања за наменске процесоре. Аспекти разликовања компајлера за наменске процесоре од компајлера за процесоре опште намене сумирани су радовима [Laupers1], [Laupers2] и [Wolfe]. Прва тачка разликовања произилази из тога што је за наменске процесоре квалитет генерисаног кода далеко важнији (што чини прихватљивијим дуже превођење зарад добијања квалитетнијег кода, бар у завршним фазама развоја), док су сами програми у просеку мањи (што значи да се класичним техникама превођења могу превести брзо). Последица тога је већа могућности и учинковитост оптимизационих техника које се заснивају на вишеструким стратегијама превођења. Примери таквих оптимизација су варирање параметара превођења ([Fursin]), варирање самог улазног кода ([White]), као и варирање редоследа фаза превођења ([Kulkarni]). Осим вишеструких стратегија, и оптимизационе технике које претпостављају анализу целокупног програма добијају на значају услед повећања прихватљивог времена превођења и повећања релативне делотворности превођења.

Разлагање компајлера у фазе, или пролазе, има низ предности и представља уобичајену праксу у већини компајлерских архитектура. Међутим, те фазе најчешће нису међусобно независне, а и редослед фаза није непроменљив. Код компајлера за процесоре опште намене та међузависност у великој мери може бити занемарена, али код наменских процесора тим занемаривањем се значајније губи на квалитету кода. Одлуке донесене у једној фази утичу на друге фазе и познавање тог утицаја доприноси бољем доношењу одлука. Због тога, друга тачка разликовања компајлера за опште и наменске процесоре је управо већа потреба да се код компајлера за наменске процесоре повећа комуникација између различитих фаза, што се у литератури обично назива *здруживање фаза* (енг. phase coupling). Здруживање фазе избора инструкција, доделе регистара (и ресурса уопште) и распоређивања инструкција уочено је као најважније, пре свега због неортогоналности инструкционог скупа и паралелизма на нивоу инструкција. Рад [Bashford] даје преглед неколико приступа који здружују две или три од поменутих фаза. Примери скорашњијег истраживања у тој области су рад [Rajagopalan], у којем се анализира спој распоређивања и доделе регистара, затим [Eriksson] и [Grewal], који разматрају спајање све три фазе. Резултати увек приказују значајно побољшање квалитета кода, тако да се здруживање фаза сматра врло важним аспеком компајлера за наменске процесоре.

Здруживање фаза није ограничено на поменуте три. У раду [Choi] предложено је спајање фазе распоређивања и оптимизације која одређује адресе у меморији тако да погодују јединици за генерисање адреса. Уколико наменски процесори имају такву јединицу, овакав приступ добија на значају. Преглед ручних и компајлерских техника за повећање искоришћења јединице за генерисање адреса дат је у раду [Talavera].

У радовима [Laupers1] и [Laupers2] истиче се предност, у домену наменских процесора, избора инструкција над графом, у односу на избор инструкција над стаблом. Сложене инструкције, као што је MAC (енг. Multiply-ACcumlate, множи и акумулирај) захтевају сложене шаблоне који представљају проблем за технике избора инструкција над стаблом. Преглед неких од техника за избор инструкција над графом (и механизама за поплочавање графа графовским шаблоним) дат је у радовима [Bertin] и [Ebner].

У раду [Chen] је приказано како програмер може побољшати квалитет генерисаног кода тако што ће у изворни код унети одређене смернице упућене компајлеру. Неки елементи Це језика представљају компајлерске смернице. То су кључне речи ***restrict*** и ***register***, као и ***static*** у декларацији низа као параметра функције [CST99]. Компајлерске смернице описују особине кода које не мењају његову функционалност, али могу помоћи компајлеру да боље преведе код. Информације које се на овај начин саопштавају компајлеру обично је врло тешко, или чак немогуће, утврдити на други начин. Осим смерница које су део Це језика обично постоје и смернице коју су специфичне за одређен компајлер и одређен процесор. Такве смернице се изражавају помоћу прагми или атрибута.

Начин на који је улазни програм написан такође може утицати на квалитет излазног кода. За два функционално иста програма, али са мало различитим кодом, одређени компајлер може генерисати врло различите излазе, зато што су неке од оптимизација биле успешније са једом варијантом кода. Писање оптимизација које раде само са одређеним подскупом могућих начина изражавања неке функционалности у датом програмском језику или се ослањају на компајлерске смернице у изворном коду, најчешће је лакше од развијања и писања моћних оптимизација које успешно раде са великим простором улазних кодова исте функционалности.

У раду [Wulfe] је примећено да су програмери, посебно при раду са наменским процесорима, обично вољни да прилагоде свој код одређеном компајлеру. Повратна информација од компајлера о томе како би програмери могли да прилагоде конкретан код представљала би значајну олакшицу. Да би се то омогућило, потребно је оптимизацијама додати захтев да препознају случајеве и делове кода над којима нису успешно обављене. Та места у коду се затим пријављују програмеру, а пожељно је и да их прати савет како би се код можда могао променити. Ти савети се називају *програмерске смернице* и у последње време се почињу појављивати и код процесора опште намене, пре свега зато што су одређени елементи уобичајени за архитектуре наменских процесора делимично усвојени и у процесорима опште намене (као на пример векторизација и програмерске смернице код Intel компајелра, [Deilmann]).

ПОГЛАВЉЕ 3.

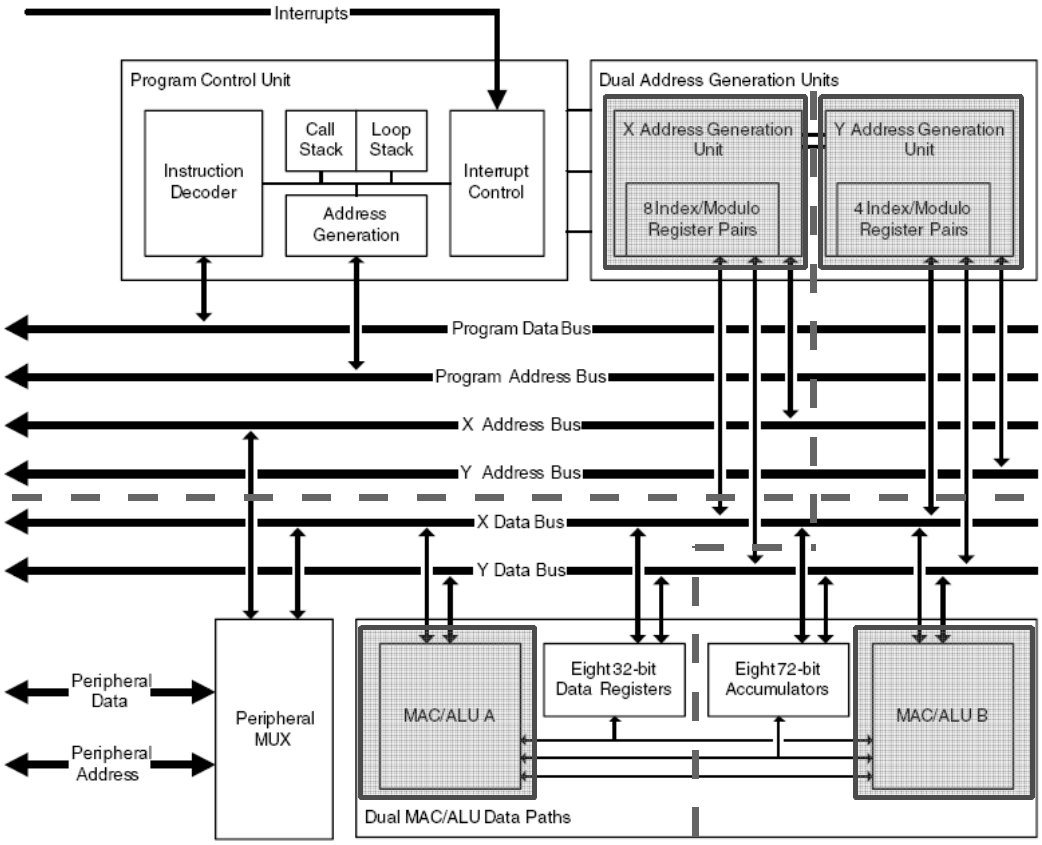
ОПИС ЦИЉНЕ ФИЗИЧКЕ АРХИТЕКТУРЕ

Може се рећи да је Coyote 32 језгро типичан представник класе процесора за дигиталну обраду сигнала. Основни елементи његове архитектуре сусрећу се и у осталим ДСП-овима у врло сличном облику. У том смислу закључци који се изводе на основу рада компајлера за ову архитектуру могу у великој мери да се прошире и на остале процесоре. Због тога опис Coyote 32 архитекуре који следи може служити и као побројавање архитектонских посебности које издвајају процесора за обраду дигиталних сигнала од процесора опште намене.

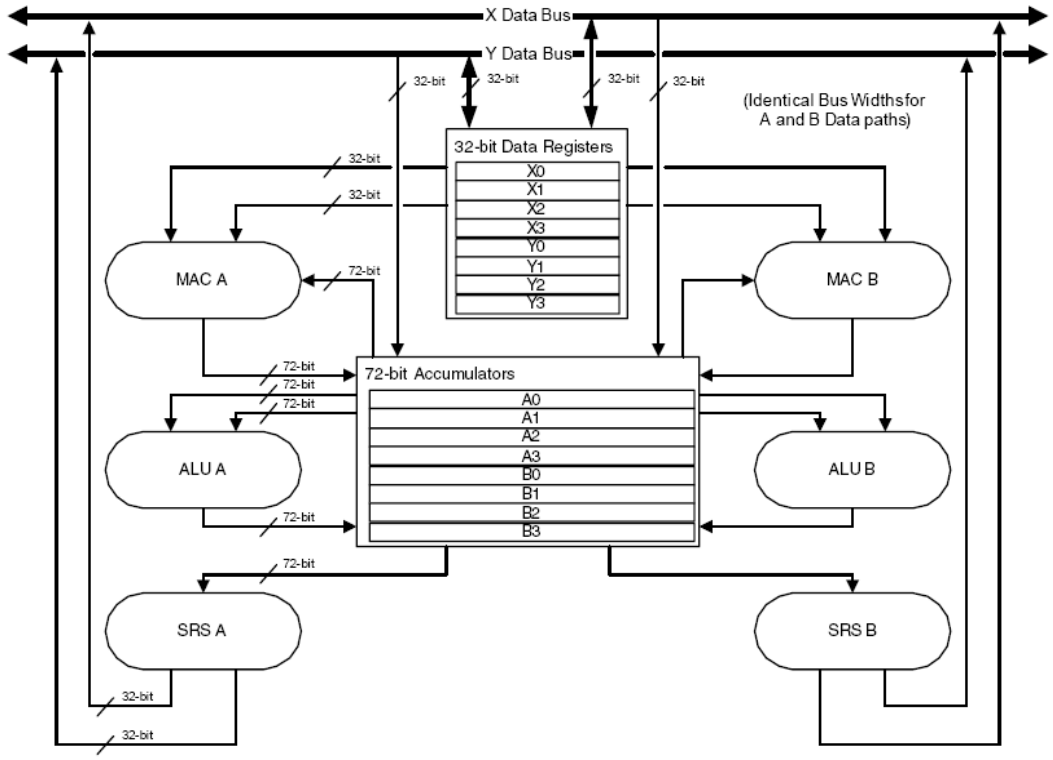
У основи Coyote 32 процесора је Харвард архитектура [Ковачевић2]. Меморијска зона за податке је додатно подељена на две зоне (X и Y), што је чест случај код ДСП-ова. Ширина све три меморије је иста и износи 32 бита, док меморије X и Y могу бити заједно третиране као једна меморије ширине 64 бита. Процесор поседује и три групе регистара:

- 8 акумулатора, дужине 72 бита
- 8 регистара података, дужине 32 бита
- 8 индексних регистара, дужине 16 бита; постоји заправо 12 индексних регистара али последња четири користи оперативни систем тако да су ефективно, са становишта програмера, на располагању само 8 регистара.

На слици 3.1 приказана је Coyote 32 архитектура, а на слици 3.2 дата је путања података кроз процесор.



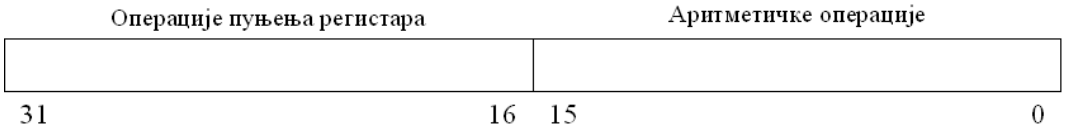
Слика 3.1 Илустрација Coyote 32 архитектуре



Слика 3.2 Путања података кроз Coyote 32 процесор

Проточна структура процесора је плитка и састоји се из три фазе: захватање, декодовање и извршење инструкције. Део проблема који настају услед међузависности фаза проточне обраде отклања сам процесор уметањем празног циклуса, али ипак већину проблема је сам програмер дужан да избегне. Међутим, због мале дубине проточне структуре оваквих случајева нема много. Напор који је потребно уложити да се асемблерски код добро сложи са аспекта проточне структуре, и по питању ваљаности и по питању ефикасности, није велики. Са друге стране, четири целине које су на слици 3.1 истакнуте сивим правоугаоницима представљају четири основне функционалне јединице процесора које могу обављати операције паралелно, током извршавања једне инструкције. Због тога се у инструкциону реч дужине 32 бита могу сместити од једне до чак четири операције. То сврстава овај процесор у ред процесора са паралелизмом на нивоу инструкција (или *инструкцијски паралелизам*, енг. instruction level parallelism, ILP) и значи да се искоришћење његових могућности ослања на статичко планирање операција у време превођења или програмирања.

Прва подела функционалних јединица процесора је према операцијама које обављају. Једна група функционалних јединица извршава операције преноса података између регистара и меморије, али делимично и између самих регистара. Саставни део ових функционалних јединица је и адресни генератор, који обавља аутоматско самоувећање или самоумањење одговарајућих индексних регистара. Друга група обухвата аритметичко-логичку јединицу и MAC (Multiply and ACcumulate) јединицу, и обавља аритметичке операције над акумулаторима и регистрима података. Имајући у виду слику 3.1 може се говорити о „хоризонталној“ подели (илустровано испрекиданом линијом). Горњих 16 бита инструкционе речи је резервисано за операције пуњења регистара, а доњих 16 за аритметичке операције (слика 3.3).



Слика 3.3 Подела инструкционе речи у Coyote 32 процесору

На слици 3.1 се види да свака од горе поменутих група поседује две функционалне јединице које обављају исте операције, само над различитим

скупом операнада. Из овога следи друга подела функционалних јединица: подела по групи операнада над којима врше операције. Референцирајући поново слику 3.1 сада можемо говорити о „вертикалној“ подели (исто илустровано испрекиданом линијом). Ова подела не узрокује даље рашчлањивање инструкционе речи, већ се подаци о две исте операције над различитим операндима пакују испреплетано у 16 бита који одговарају типу операције.

X меморија	Y меморија
X регистри података (x0-x3)	Y регистри података (y0-y3)
A акумулатори (a0-a3)	B акумулатори (b0-b3)
Индексни регистри (i0-i3)	Индексни регистри (i4-i7)

Табела 3.1 Подела регистара у подгрупе на основу функционалних јединица

Поделу функционалних јединица прати и подела регистара. Свака регистарска група се раздваја на два дела, при чему се операције над регистрима из једног дела могу извршавати паралелно са истом том операцијом над регистрима из другог дела. Подела ресурса почиње од меморије и аналогно се преноси на остале ресурсе (табела 3.1). Тиме четири групе ресурса добијају своје подгрупе. На пример, операција за пуњење регистра података из индексиране меморије има као излазни операнд један од регистара података, а као улазни операнд једну од меморијских зона индексирану произвољним индексом:

```
y1=xmem[i3]
```

Једна оваква операција заузима само горњих 16 бита инструкционе речи и она може постојати као засебна инструкција или се може ставити у паралелу са било којом аритметичком операцијом, која заузима доњих 16 бита. Међутим, могуће је и у истих тих горњих 16 бита упаковати две овакве операције, ако и само ако се у једној операцији у **x** регистар уписује податак из **x** меморије, а у другој операцији у **y** регистар податак из **y** меморије, док се **x** меморија мора индексирати регистрима i0 или i1, а **y** меморија регистрима i4 или i5:

```
x0=xmem[i1]; y1=ymem[i4]
```

Из овог примера се види да је подела индексних регистара различита у односу на друге ресурсе, јер код индексних регистара се ефективно разликују три групе регистара: i0-i1, i4-i5 и преостала четири индексна регистра.

Осим наведених операција, адресни генератори могу обавити самоувећање или самоумањење индексних регистара који се користе за приступ меморији, тако да горња инструкција може бити проширена на следећи начин:

```
x0=xmem[i1]; i1+=1; y1=ymem[i4]; i4-=1
```

Ове операције над индексним регистрима се такође кодују у горњих 16 бита, тако да доњих 16 бита остаје и даље слободно за аритметичке операције. Додавањем, на пример, две операције сабирања добија се инструкција са највише операција за Coyote 32 процесор:

```
x0=xmem[i1]; i1+=1; y1=ymem[i4]; i4-=1; a0=a1+b1; b0=b1+a1
```

Да би дате операције сабирања могле делити доњих 16 бита неопходно је да се један резултат смешта у **a** регистар, а друга у **b** регистар, али индекси тих регистара морају бити исти, као и индекси одговарајућих сабирака. Из свега наведеног произилази да избор регистарских операнада за неку операцију непосредно утиче на могућност паралелизације те операције са другим операцијама. Са аспекта преводиоца или програмера, ово је врло важна чињеница.

Циљна физичка архитектура је у стању да обавља неколико операција које имају више од два улазна операнда или више од једног излазног операнда, што су операције које се ређе сусрећу код процесора опште намене. Главни представник тих операција је MAC операција која множи два регистра података, међурезултат сабира са акумулатором и крајњи резултат смешта у акумулатор. Остали примери оваквих операција би биле операције учитавања из спојене XY меморије у пар одредишних регистара, као и постојање два одредишта код неких аритметичких операција.

Скуп операција циљне физичке архитектуре је такође врло неортогоналан, тако да се операције које је могуће обавити над једном врстом регистара веома разликују од операција које се могу обавити над другом врстом. Тако, на пример, сабирање са непосредним операндом је могуће само за индексне регистре, док сабирање два регистра ради само за акумулаторе; множење се може обавити само са регистрима података, а упоређивати се могу само акумулатори, итд.

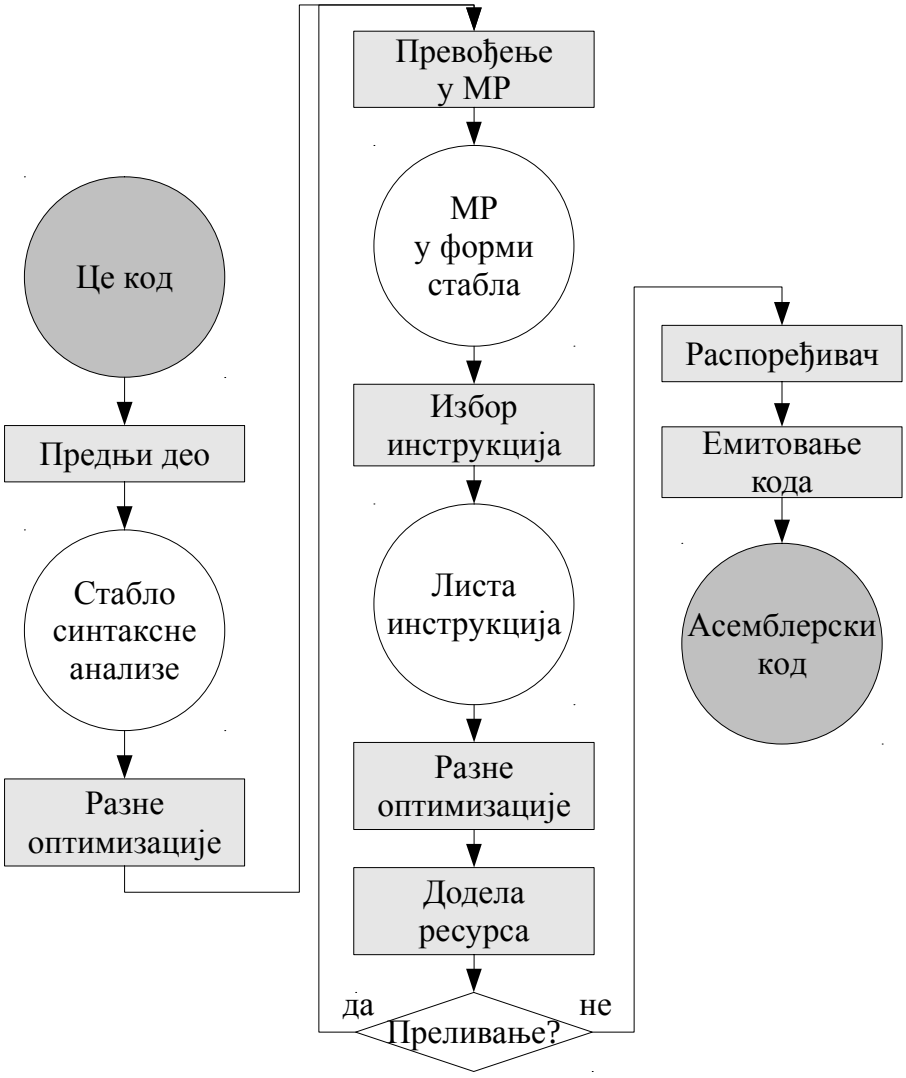
Још једна посебност циљне архитектуре је постојање физички подржаних петљи. Посебном инструкцијом се саопштава до које адресе се простире тело

петље и колико пута је то тело потребно извршити. Физичка архитектура ће се побринути да се тај блок инструкција изврши задати број пута, на тај начин уклањајући операције потребне за режију петље. Овај механизам на Coyote 32 платформи подржава веома добро формиране петље [Ковачевић1] (петље код којих је број понављања сталан и познат током превођења), али и добро формиране петље (петље код којих број понављања није познат у време превођења, али јесте познат при уласку у петљу током извршавања). Величина тела петље није ограничена, али број итерација јесте на 1024 ако се користи непосредни операнд (нула у коду операције означава 1024 итерација), а 65536 ако се користи индексни регистар.

ПОГЛАВЉЕ 4.

ОПИС ПОЛАЗНОГ КОМПАЈЛЕРА

Неколико кључних елемената полазног компајлера урађено је у складу са књигом [Appel]. Ток превођења организован је у пролазе (фазе) и приказан је на слици 4.1. Коришћен је комерцијлани предњи део компајлера, задужен са лексичку и синтаксну анализу, развијан од стране компаније Едисон дизајн груп (Edison Design Group). Предњи део је измењен да подржава проширење Це језика за уграђене системе (Embedded C [CSTEXT]) које уводи типове података са непокретним зарезом и квалификаторе за меморијске зоне. Стабло синтаксне анализе, које је излаз предњег дела, учитава се у задњи део компајлера. Одмах при учитавању претвара се у стабло међукода, независно од циљне архитектуре и полазног језика. Фаза избора инструкција конструисана је коришћењем IBURG алата ([Fraser1], [Fraser2]), на основу правила за поплочавање стабла и њихових цена. Избор инструкција обавља се над стаблом међукода тражењем најмање цене поплочавања целог стабла расположивим шаблонима. На основу таквог поплочавања генерише се низ операција циљног процесора који је функционално једнак стаблу међукода. За доделу регистара користи се алгоритам заснован на бојењу графа поједностављивањем са спајањем чворова (описан у [Appel]).



Слика 4.1 Ток превођења код полазног компајлера

На постојећу структуру додато је неколико проширења у циљу прилагођавања компајлера наменским процесорима за дигиталну обраду сигнала. На високом нивоу апстракције, док се још барата стаблом синтаксне анализе, додато је два пролаза: пролаз за откивање петљи које могу бити физички подржане и пролаз за искоришћење адресних генератора [Зарић]. На ниском нивоу, када се барата са листом инструкција, то јест операција, додат је пролаз за распоређивање инструкција заснован на слагању инструкција у листу (*слагајући распоређивач*, или *распоређивач листе*, енг. list scheduler), описан у [Живковић]. Фаза распоређивања открива могућности за остваривање паралелизма на нивоу инструкције, а уједно и обавља мутирање операција, уколико се за тиме јави потреба.

Додела регистара је благо измењена додавањем хеуристике која покушава да додели регистре тако да се у општем случају повећа потенцијал за паралелизам на нивоу инструкције [Ђукић]. Међутим, цео процес се обавља без било какве повратне информације из фазе распоређивања. Напослетку, додато је и неколико малих пролаза који обављају разне машински зависне ситне оптимизације. У литератури су такве оптимизације познате као *оптимизације прозорчића* (енг. peephole optimizations) јер оптимизују само врло мали блок узастопних инструкција (најчешће две или три), мада у овој конкретној изведби нису ограничене само на узастопне инструкције (као што је, рецимо, илустровано у књизи [Cooper]).

Полазни компајлер је дуже време коришћен за развој програма на неколико успешних комерцијланих пројеката. Квалитет генерисаног кода је био довољно добар у раним фазама развоја, када величина и брзина кода нису још увек толико важни. Ипак, сви програми су до краја пројекта бивали урађени скоро потпуно у ручно писаном асемблерском коду. Током тог периода коришћења компајлера уочено је неколико проблема при његовој употреби и одржавању. Алгоритми за откривање физички подржаних петљи и искоришћење адресних генератора су се показали као превише крути. Нису покривали све случајеве који су се сусретали у пракси, а програмери би то обично сазнали прилично касно у процесу развоја програма. Ниво паралелизма на нивоу инструкција није био превише висок у поређењу са ручно писаним кодом. Одржавање компајлера је такође било мукотрпно. Три различита међујезика повећавали су сложеност компајлерске структуре. Инжењер који ради на одржавању морао би познавати рад са сва три међујезика (два права међујезика и једна апстрактна синтакса), и разумети њихову програмску организацију и семантику.

Показало се да је део компајлера који барата са стаблом синтаксне анализе и стаблом међукода много тежи за отклањање грешака и контролисано извршавање (дебаговање), што је уочено и у раду [Johnson]. Фазу избора инструкција је било посебно тешко одржавати. Потребно је било знање додатног језика за опис правила за поплочавање стабла са којим IBURG алат ради. Свака промена скупа правила захтевала је поновно покретање IBURG алата и генерисање кода који обавља избор инструкција. Осим што ово успорава рад,

чињеница да је код аутоматски генерисан отежавала је његово разумевање и поправљање. На крају, коришћене међурепрезентације су биле ограничене на функције и нису изражавале међупроцедуралне зависности. Због тога би сваки покушај додавања глобалних оптимизација захтевао значајне промене у целом компајлеру, па због тога тако нешто никад није ни спроведено у дело.

ПОГЛАВЉЕ 5.

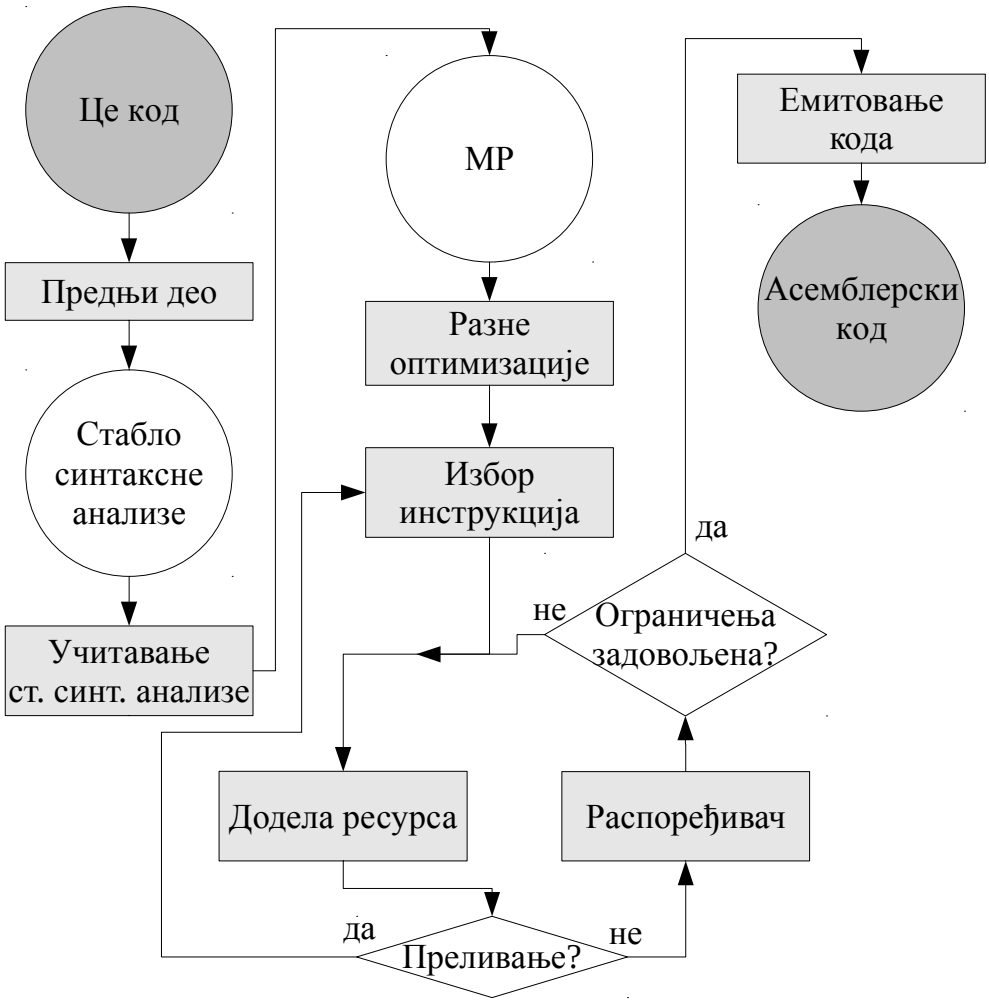
ОПИС НОВЕ КОМПАЈЛЕРСКЕ ИНФРАСТРУКТУРЕ

Основна намера при започињању развоја новог компајлера била је превазилажење недостатака полазног компајлера, али такође и испуњење четири циља наведена у уводу. Одлучено је да се прво развије компајлерска инфраструктура, а затим да се на основу ње изгради Це компајлер за Coyote ДСП. Иако иницијално није било намере да се инфраструктура искористи за више од једног компајлера ипак је изабран овај приступ, и то из два разлога. Један разлог је уверење да је пројектовање компајлера на основу неке инфраструктуре систематичније и стога лакше за одржавање. Други разлог је претпоставка да ће се инфраструктура моћи искористити и за неке друге алате осим компајлера, као што је на пример компајлирани симулатор [Djukic1]. До данас је, ипак, ова компајлерска инфраструктура употребљена и за два додатна компајлера. Један је рађен у истраживачке сврхе, и циљна платформа му је MIPS процесор [Povazan2], док је други, чија је циљна платформа Starkey Sonnet ДСП, још у развоју.

Компајлерска инфраструктура је организована као библиотека класа, насупрот коцепата аутоматског генерисања кода и доменског моделовања [Guilan]. Због тога програмер који жели да направи компајлер ослањајући се на ову инфраструктуру сам пише Це++ код који управља током превођења, користећи елементе понуђење у библиотеци. За неке активности током превођења у библиотеци је понуђено само једно решење, док је за неке понуђено

више решења од којих програмер бира најповољније за конкретну циљу архитектуру и намену. Уједно, програмер је слободан да у било ком тренутку и за било коју функционалност напише сопствени код, занемарујући елементе библиотеке који се за ту функционалност нуде. То омогућава велику флексибилност у пројектовању, али самим тим пребацује већу одговорност на програмера.

У новом компајлеру искоришћен је исти предњи део за Це програмски језик, као и код полазног компајлере. Исто стабло синтаксне анализе које је излаз из предњег дела учитава се у задњи део, међутим, током учитавања стабло синтаксне анализе се одмах преводи у међурепрезентацију. Тиме се стабло синтаксне анализе скоро потпуно уклања из компајлера (слика 5.1). У идеалним условима део кода који учитава стабло синтаксне анализе никада више не би требало мењати.



Слика 5.1 Ток превођења код новог компајлера

5.1 Међурепрезентација

У новој инфраструктури се користе две међурепрезентације, једна ближа полазном језику (али ипак независна од њега), и друга ближа циљној процесорској архитектури (и тесно зависна од ње). Обе међурепрезентације, међутим, имају исту програмску структуру. То значи да се у компајлеру практично користи једна међурепрезентација која може да садржи два нивоа апстракције: виши (HL, енг. high level) и нижи (LL, енг. low level). Једина разлика између та два нивоа је што нижи ниво садржи операције које одговарају операцијама циљног процесора (асемблерски код се емитује директно из МР ниског нивоа), док се код вишег нивоа користе машински независне операције. Операције се у инфраструктури називају инструкцијама, али је тај појам различит од машинске инструкције, која у себи може садржати више од једне операције. Када се буде причало о распоређивању, направиће се прецизнија разлика тих појмова. За сада, под појмом инструкција мисли се на објекат међурепрезентације који моделује једну операцију, а у случају другог смисла тог појма биће наглашено да је у питању машинска инструкција. Детаљнији опис елемената међурепрезентације дат је на слици 5.1.1.

Сами операнди инструкција су такође исти на оба нивоа апстракције. Сваки операнд је одређен типом и врстом. Типови операнада осликавају типове полазног језика, у овом случају Цеа, и допуњују смисао операција високог нивоа. Могуће врсте опаранда су:

- константни операнд (чија тачна бројна вредност је позната током превођења)

- адресни операнд (исто константа, али која је везана за адресу неког објекта, па као таква није позната током превођења)

- симболички операнд (слично као адресни, само није везан за адресу неког објекта)

- офсет операнд (изражава померај у односу на неку адресу; састоји се од једног адресног операнда и целог броја)

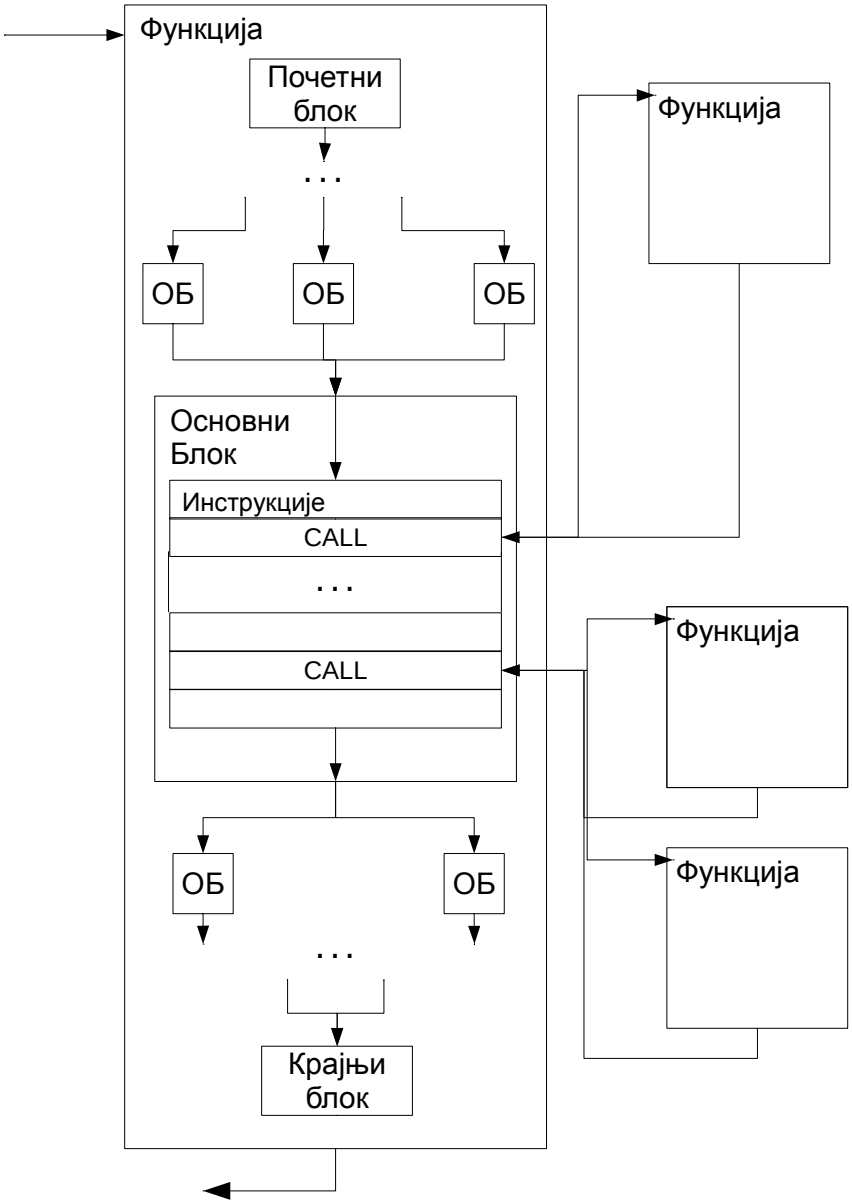
- ресурсни операнд (представља променљиву вредност којој мора бити додељен неки ресурс током извршавања)

У зависности од врсте, операнди имају још неке особине. Најважнија је особина ресурсних операнада која одређује класу ресурса која му је придружена, као и конкретан ресурс из те класе. На почетку тока превођења, док је међурепрезентација на вишем нивоу апстракције, тип операнда је кључна особина од интереса, али како се међурепрезентација спушта на нижи ниво, тако важнији постају ресурси придружени ресурсним операндима. Међутим, као што је већ речено, операнди стално садрже све поменуте особине и исти су у оба нивоа апстракције.

Оваква организација међурепрезентације чини много лакшим развој и одржавање компајлера. Комапајлерском инжењеру је довољно да познаје само један скуп поступака за руковање међурепрезентацијом и њеним елементима. Осим тога, током превођења међурепрезентација може садржати мешавину елемената високог и ниског нивоа. То значи да се може обављати делимично спуштање кода, али и да се стање међурепрезентације може на једноставан и униформан начин записати у датотеку у било ком тренутку, што је јако повољно приликом отклањања грешака.

Слика 5.1.1 Неформални опис основног садржаја међурепрезентације

Међурепрезентација је хибридне организације. На нивоу основних блокова организована је линеарно, док су сами основни блокови увезани у граф тока управљања који одговара функцији. Функције су затим увезане у граф позива. Међупроцедуралне зависности су саставни део међурепрезентације, што омогућава лакше изражавање глобалних оптимизација. За сваку функцију наведено је које друге функција она може звати, и на којим местима се то дешава, као и од стране којих функција она може бити позвана (слика 5.1.2).



Слика 5.1.2 Сликовит приказ огранизације међурепрезентације

За граф тока управљања задужен је модул за анализу тока управљања. Он ради над функцијом и поправља постојећи граф тока управљања. Заправо, на самом почетку превођења, када се учитава стабло синтаксне анализе и формира међурепрезентација, свака функција је представљена графом тока који има само један основни блок у који су смештене све инструкције. Тек првим позивом анализе тока управљања добија се исправан граф. Зато ако током превођења дође до промене инструкција, није потребно водити рачуна о изгледу графа. Довољно је убацити или избацити инструкције у одговарајући основни блок, а накнадни позив анализе тока управљања средиће граф. Саставни део међурепрезентације је

и граф тока података, за чије формирање је задужен модул за анализу тока података. И овај модул се може позвати у било ком тренутку да поправи постојећи граф. Оба ова модула раде над међурепрезентацијом без обзира на ниво апстракције операција.

Још једна важна имплементациона карактеристика међурепрезентације је и употреба сакупљача смећа за управљање динамички заузетим објектима који се користе као елементи међурепрезентације. На овај начин се програмер растерећује од размишљања о динамички заузетим објектима, чиме се додавање и уклањање елемената међурепрезентације значајно поједностављује. Сакупљач ради техником „обележи и почисти“, то јест у одређеним тренуцима обележи све објекте до којих се може доћи крећући од коренских објеката, а затим пролази кроз све објекте и уклања оне који нису обележени. Код оваквог приступа режија прављења нових објеката остаје готово иста, а сакупљање смећа се може обавити у одабраним повољним тренуцима тока превођења. Употреба неког програмског језика који подразумева постојање сакупљача смећа би уклонила потребу за експлицитним увођењем сакупљача, али писање индустријског компајлера употребом језика који нису Це или Це++ представља искорак који у овој области тек треба да се направи.

5.2 Моделовање циљне архитектуре

Циљна процесорска архитектура је моделована ручно, пошто не постоји формалан опис архитектуре. Компајлерска инфраструктура очекује да особине циљног процесора буду описане у неколико класа. Одговарајуће класе се наслеђују и дефинишу се потребни атрибути и методе. Преглед кључних класа дат је на слици 5.2.1.

Ресурси циљног процесора се обично први моделују, наслеђивањем класе ***CTargetResource***. Инфраструктура обезбеђује механизам за описивање основних група ресурса (различите меморијске зоне, различите групе регистара) и њихових подгрупа. Постојање подгрупа је врло важно за изражавање ресурсних ограничења која се јављају у вези са пралелизмом на нивоу инструкција (чему се посвећује пажња у каснијим фазама превођења). Да би опис ресурса био потпун, потребно је наследити и класу ***CTargetPreAllocator***, у којој се одређује

механизам доделе основних ресурсних група ресурсним операндима. Ова додела не мора бити коначна. У компајлеру се у било ком тренутку може донети другачија одлука, али ова класа се користи како би се осигурало да ће сваком ресурсном операнду бити придружена нека ресурсна група, тј. да неће бити операнда без придружене групе.

Слика 5.2.1 Преглед класа за опис циљне процесорске архитектуре у новој инфраструктури. Класе су груписане по областима које описују.

Операциони кодови инструкција су излистани у два набројива типа, један за виши ниво MP, други за нижи. Сваком операционом коду мора бити придружен репрезентациони знаковни низ (стринг), који ће бити коришћен за запис MP у датотеку. Репрезентациони знаковни низ не мора неопходно бити искоришћен за завршно записивање асемблерског кода јер једна операција нема увек исту знаковну репрезентацију у свим контекстима. Код Coyote 32 процесора знаковна репрезентација највише зависи од тога да ли је операција у паралели са неком другом. Завршни емитер асембелерског кода мора да води рачуна о таквим посебним случајевима. Све појединости о знаковном запису дефинишу се у класи наслеђеној од класе ***CTargetFormater***.

Дефиниција инструкција не садржи податке о томе које врсте операнада одређена операција може имати. Одговорност је на програмеру компајлера да води рачуна о томе на местима где се инструкције међурепрезентације праве, то

јест да обезбеди да се инструкција направе са одговарајућим врстама операнада.

Осим репрезентационог знаковног низа још неке апстрактне особине морају бити придружене инструкцији, нпр. да ли је у питању инструкција гранања, записа у меморију или читања, и слично. Те особине омогућавају да се многе уобичајене компајлерске анализе и оптимизације (као што су анализа животног века, уклањање мртвог кода, анализа тока извршавања, уклањање заједничких подизраза, прављење графа тока зависности, анализа показивача итд.) обављају над међурепрезентацијом без обзира које операције она садржи (високог или ниског нивоа, било које циљне платформе). Класа чијим наслеђивањем се те особине дефинишу је ***CTargetInstructionSet***.

Још је потребно дефинисати и особине инструкција са становишта проточне структуре. Кашњења инструкција приликом читања операнада и писања резултата морају се описати у класи наслеђеној од ***CTargetLatencies***, док се у класи наслеђеној од ***CTargetHazards*** описује постојање посебних случајева сметњи у проточној структури који се не смеју сусрести у коду. Паралелизам на нивоу инструкције, ако постоји код неке циљне архитектуре, потребно је описати у класи наслеђеној од ***CTargetParallelConditions***, али о томе ће више бити речи у поглављу о распоређивачу. Напослетку, неопходно је дефинисати и позивну конвенцију наслеђивањем класе ***CTargetCallingConvention***.

Подаци о величини основних Це типова у битима, и њиховој величини у бајтима (меморијским речима) за различите меморије (ширине различитих меморијских зона не морају бити исте: код Coyote 32 процесора XY зона је широка 64 бита, док су остале зоне широке 32 бита) не налазе се у класи. Те информације су потребне и предњем делу, а оба предња дела која су до овога тренутка прилагођена за употребу са овом инфраструктуром написана су у Цеу, а не Це++-у. Одлучено је да се такве информације дефинишу само на једном месту и да се користе од стране и предњег и задњег дела. Због тога је направљено посебно заглавље у којем се налазе симболи који одређују величине типова у битима и табела која одређује колико речи (бајтова) ти типови заузимају у различитим меморијским зонама.

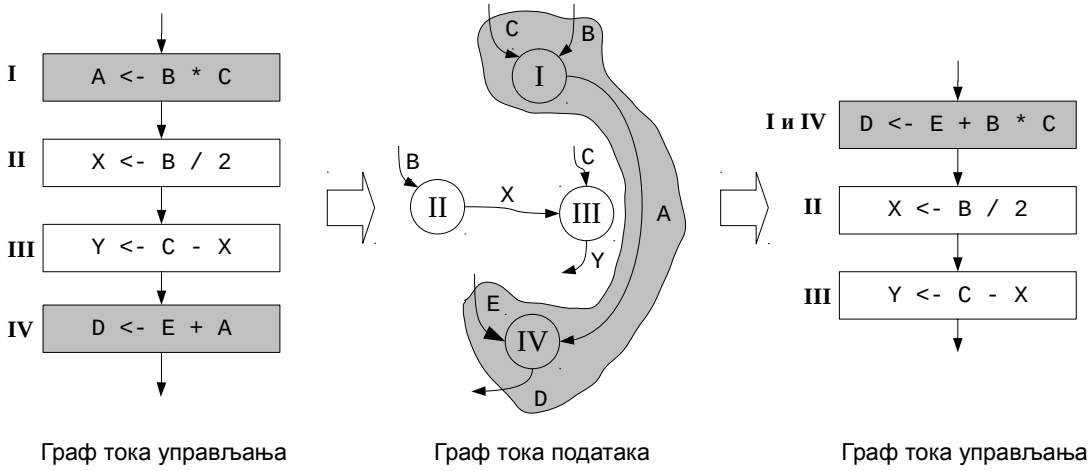
Све што је до сада наведено представља основни опис циљне архитектуре и посебности одређеног компајлера који се заснива на овој инфраструктури. Осим

описа паралелизма на нивоу инструкције, све остало је неопходно за правилан рад компајлера и генерисање ваљаног кода. Међутим, то је још далеко од компајлера који генерише и квалитетан код. У том смислу, многе оптимизације захтевају додатне податке и упутства како да правилно и делотворно раде са одређеном циљном платформом. То се обично обавља наслеђивањем класе која представља одговарајућу оптимизацију или анализу, и попуњавањем потребних метода или атрибута да би та класа радила и са новом платформом. Овде је важно подсетити да је компајлерска инфраструктура организована као библиотека класа и да је ово у складу са тим концептом. Због тога се од програмера компајлера очекује још и формулисање комплетног поступка превођења позивањем одговарајућих модула у складу са жељеним током превођења.

5.3 Избор инструкција - Преписивачка правила

Нова компајлерска инфраструктура укључује подршку за изражавања и примену *преписивачких правила* (енг. rewrite rules) као средства за трансформацију међурепрезентације и избор инструкција над графом. Свако правило је описано једном Це++ класом која наслеђује базну класу правила. Објекат те класе представља само правило и користи се за примену правила. Класа сваког правила садржи две међурепрезентације: улазни и излазни шаблон. Улазни шаблон описује међукод који се може заменити, а излазни шаблон представља сменатички еквивалентан део међукода који ће бити искоришћен да замени код који одговара улазном шаблону. И улазни и излазни шаблон се конструишу као делови међурепрезентације, што значи да је познавање рада са међурепрезентацијом довољно за писање правила. Међукод који се користи за шаблоне је линеаран, али се интерпретира као граф када се упарује, јер се упаривање обавља над графом зависности података (граф тока података), као што је илустровано примером на слици 5.3.1. За сваки шаблон конструише се граф зависности података, а у графу зависности података главне међурепрезентације (оне на коју се примењују правила) траже се делови који су пандан улазном шаблону. При упаривању шаблона са делом главне међурепрезентације потребно је да се слажу чворови и ивице графа зависности,

што подразумева да се и операнди придружени ивицама слажу. Као што је описано у поглављу о међурепрезентацији (5.1), операнд међурепрезентације је одређен својом врстом (ресурсни операнд, константа итд.), типом и, у зависности од врсте, неком особином као што су: вредност, класа ресурса, величина у меморији итд. У сваком правилу могуће је специфицирати до које мере се операнди морају подударати. На пример, у неким правилима само тип операнда се мора слагати, док класа ресурса не мора бити иста.



Слика 5.3.1 Илустрација примене преписивачког правила. Шаблон правила се изражава као низ инструкција, али се упарује на графу тока података. Дат је хипотетички пример правила за избор MAC (Multiply ACcumulate) инструкције.

Ако се улазни шаблон успешно упари са неким делом међукода то за већину правила значи да се оно може применити. Међутим, у неким случајевима додатни услови морају бити задовољени (нпр. да се нека константа налази у одређеном опсегу и сл.). У том случају се виртуелна метода правила која обавља упаривање може преклопити и у тој преклопљеној методи се могу обавити додатне провере. За неке уобичајеније додатне услове, као што је, рецимо, горепоменута провера опсега константе, дате су предефинисане функције у инфраструктури да би олакшале посао програмерима, али, као што је наглашено, не постоји било какво ограничење у погледу врсте додатних услова који се могу придружити правилу.

```
(define_insn "*addsi3"
  [(set (match_operand:GPR 0 "register_operand" "=d")
   (plus:GPR (match_operand:GPR 1 "register_operand" "d")
   (match_operand:GPR 2 "register_operand" "d")))]
  ""
  "addu\t%0,%1,%2"
  [(set_attr "alu_type" "add")
   (set_attr "mode" "<MODE>")])
```

а) GCC правило

```
class ArithLogicR<string opstr, RegisterOperand RO,
                  bit isComm = 0,
                  InstrItinClass Itin = NoItinerary,
                  SDPatternOperator OpNode = null_frag>:
  InstSE<(outs RO:$rd), (ins RO:$rs, RO:$rt),
         !strconcat(opstr, "\t$rd, $rs, $rt"),
         [(set RO:$rd, (OpNode RO:$rs, RO:$rt))],
         Itin, FrmR, opstr>
{
  let isCommutable = isComm;
  let isReMaterializable = 1;
}

def ADD : MMRel,
          ArithLogicR<"add", CPURegsOpnd, 1, IIAlu, add>,
          ADD_FM<0, 0x20>;
```

б) LLVM правило

```
accumulator: PLUS_NO(accumulator, accumulator) = 103 (1);

CAsmOperand* InstSelect::Reduce(NODEPTR_TYPE p, int nonterm)
{
  int rulenumber = burm_rule(STATE_LABEL(p), nonterm);
  short* nts = burm_nts[rulenumber];
  NODEPTR_TYPE kids[10];

  burm_kids(p, rulenumber, kids);

  switch (rulenumber)
  {
    case 103:
      ...
      CAsmOperand *src1, *src2, *dst;
      ...
      src1 = Reduce(kids[0], nts[0]);
      src2 = Reduce(kids[1], nts[1]);
      ...
      return dst;
  }
  ...
}
```

в) IBURG правило

```
class CLowerAddNumType : public CSingleInstTypeMatchRule
{
public:
  CLowerAddNumType(CRewriter* rewriter)
  : CSingleInstTypeMatchRule(rewriter)
  {
    m_ruleName = "CLowerAddNumType";

    CBaseType* numType
      = (CBaseType*)m_types->createNumType();

    COperand* dest = m_ir->createResOp(numType, ACCUM);
    COperand* src1 = m_ir->createResOp(numType, ACCUM);
    COperand* src2 = m_ir->createResOp(numType, ACCUM);
    COperand* flag
      = m_globals->getGlobOp(CGlobOps::ARITH_FLAG);

    MAKE_PATTERN(ADD_OC, DEST(dest), SRC(src1, src2));
    MAKE_TEMPLATE(OPKIND_ADD_ACC_ACC, DEST(dest, flag),
      SRC(src1, src2));
  }
};
```

г) Правило код новог компајлер

Слика 5.3.2 Четири облика преписивачког правила за сабирање целих бројева

Главна предност оваквог приступа опису преписивачких правила произилази из чињенице да се правила изражавају директно у Це++-у, а не у неком додатном језику. Није потребно додатно учење синтаксе пошто је знање рада са међурепрезентацијом довољно за писање правила. Правила се брзо преводе јер нема међукорака који генерише код из неког друго језика. Из истог разлога и отклањање грешака је лакше јер правила могу бити анализирана и контролисано извршавана у свом изворном облику. Негативна страна је то што су овако написана правила тесно везана за конкретну инфраструктуру, али ионако је реткост да се правила и опис циљне архитектуре уопште, преносе из једне инфраструктуре у другу. Осим тога, постојање посебног језика за правила омогућило би можда већу флаксибилност и изражајност. Међутим, приликом имплементације Coyote 32 Це компајлера коришћењем ове инфраструктуре показало се да је изражајност довољно добра за ту намену.

Слика 5.3.2 показује пример једног правила. Правило које поклапа целобројно сабирање дато је за четири различита компајлера. Прва три облика правилу су, редом, за GCC (преузето из GCC MIPS прилагођења), за LLVM (преузето из LLVM MIPS прилагођења) и за IBURG (из полазног компајлера). На

четвртом месту дат је облик правила који се користи у новој компајлерској инфраструктури. Иако приказана правила не циљају исти процесор ипак се могу упоредити јер имају исти смисао, само различит контекст који произилази из различитих компајлерских инфраструктура. Важно је напоменути да правила за GCC и LLVM садрже и делимичне информације о знаковној и бинарној репрезентацији инструкција циљне платформе. У полазном и новом компајлеру информације о знаковном облику инструкције се похрањују на другом месту, док се бинарни формат уопште не емитује. Од приказаних правила само правило из нове компајлерске структуре је у потпуности написано у Це++-у и не захтева никакво посебно претпроцесирање (MAKE_PATTERN и MAKE_TEMPLATE су једноставни Це++ макрои и њихово значење је прилично јасно).

Чињеница да се јединствена међурепрезентација користи кроз цео задњи део компајлера омогућава употребу преписивачких правила за разне трансформације кода, не само за избор инструкција. У Coyote 32 компајлеру преписивачка правила се користе и за смањење комплексности, алгебарска поједностављивања, неке трансформације на вишем нивоу које су специфичне за циљни процесор, као и оптимизације прозорчића на нижем нивоу. Укупно има 1032 правила за избор инструкција и 108 правила за друге сврхе.

Свака инструкција поседује листу имена свих правила која су довела до њеног стварања. Та листа се користи при отклањању грешака када је потребно утврдити ток настајања инструкције. Када се правило примењује, нека **I** буде улазни скуп инструкција (инструкције које су упарене са улазним шаблоном), а **O** скуп излазних инструкција (инструкције које су генерисане на основу излазног шаблона). Начин формирања листе имена правила за излазне инструкције описан је формулом (1).

$$\forall o \in O, list(o) \leftarrow \sum_{i \in I} list(i) + r \qquad (1)$$

У формули (1) сабирање над листама представља њихово надовезивање, а **r** име правила које се примењује. Гледањем листа имена правила могуће је у било

ком тренутку утврдити редослед примене правила који је довео до одређене инструкције. Изглед исписа ових листи је дат на слици 5.3.3.

```
x0 = a0h; a0 = a0 – a1  # RULES: (CLowerMulInt, CMoveIntRegIntAcc), (CLowerSubSimilarNumType)
x1 = a1h  # RULES: (CLowerMulIntegers, CMoveIntRegIntAcc)
x0 = a0h; a1 = x0 * x1  # RULES: (CLowerSubSimilarNumType, CMoveIntRegIntAcc), (CLowerMulInt)
a1 = a1 >> 1  # RULES: (CLowerMulInt)
AnyReg(a1h, a1l)  # RULES: (CLowerMulInt)
a1l = (0x0)  # RULES: (CLowerMulInt)
x1 = a1h  # RULES: (CLowerMulInt, CMoveIntRegIntAcc)
a0 = x0 * x1  # RULES: (CLowerMulInt)
a0 = a0 >> 1  # RULES: (CLowerMulInt)
AnyReg(a0h, a0l)  # RULES: (CLowerMulInt)
a0l = (0x0)  # RULES: (CLowerMulInt)
ret  # RULES: (CRetRule)
```

Слика 5.3.3 Илустрација исписа листи правила чијом применом су настале инструкције

5.4 Распоређивач

До овог момента у процесу компајлирања није узиман у обзир паралелизам на нивоу инструкција, као ни конфликти у проточној структури. Једна операција је представљена једном инструкцијом међукода и претпоставља се да је кашњење сваке инструкције 1. Задатак модула распоређивача је да уреди код у складу са проточном структуром и да искористи паралелизам на нивоу инструкције. Уређивање кода у складу са проточном структуром је обично неопходно да би генерисани код уопште био ваљан, док паралелизација операција само доприноси квалитету кода (ово је тачно под претпоставком да на циљној архитектури свака операција може постојати сама у инструкцији). Coyote 32 циљни процесор има плитку проточну структуру и само неколико случајева који могу довести до конфликта. Због тога главни посао распоређивача у овом случају је паралелизовање операција.

Уместо распоређивача заснованог на слагању инструкција у листу (слагајућег распоређивача), који се користи у полазном компајлеру, у новом компајлеру користи се перколациони распоређивач [Nicolau] (међутим, важно је напоменути да је овај распоређивач само један модул библиотеке класа које чине ову инфраструктуру и у том смислу не представља једини могући начин на који се распоређивање може обавити). Слагајући распоређивач уређује операције по неком приоритету, а затим пролази кроз тако уређену листу и бира операције које ће ставити у исти циклус, то јест у исту машинску инструкцију. За разлику од њега, перколациони распоређивач пролази кроз операције редом којим се налазе у програму и покушава да их помери на највишу или најнижу могућу позицију у току извршавања. Када помера операцију перколациони распоређивач

тражи циклус у основном блоку где је може преместити, узимајући у обзир све зависности података и тока извршавања, као и кашњења у проточној структури. Циклус у који се покушава преместити операција може бити празан, али и може већ садржати неку операцију. У том случају операција може бити премештена у тај циклус само ако по условима инструкционог паралелизма може ићи са том операцијом у паралели. Распоређивач, дакле, мора поседовати знање о условима за паралелизам на нивоу инструкције и то знање је изражено као скуп правила која су врло слична преписивачким правилима. Свако паралелизационо правило садржи информације о једном случају паралелизације: листу операција које се могу извршити у паралели и листу ресурсних ограничења која морају бити задовољена да би таква паралелизација била могућа. Уколико постоји много случајева пралелизације онда овај приступ захтева писање великог броја паралелизационих правила, али са друге стране омогућава директније и прецизније одређивање шта може ићи у пралелу, а шта не.

Један пример пралелизационих услова на Coyote 32 процесору је дупли меморијски упис (врло сличан примеру датом у поглављу 2), који је могућ само ако су задовољени следећи услови:

а) Одредишта су две различите меморијске зоне: X и Y зона

б) Оба изворишта су акумулаторски регистри

в) Обе меморије се адресирају индексним регистрима

г) Изворište које се уписује у X зону мора бити један од прва 4 акумулатора, а изворište које се уписује у Y зону мора бити један од друга 4 акумулатора

д) Идексни регистар који адресира X зону мора бити регистар 0 или 1, а индексни регистар који адресира Y зону мора бити регистар 4 или 5.

Сви горенаведени услови, осим а), могу бити изражени скупом ресурсних ограничења која емитује распоређивач. Услов а) је већ индиректно одређен у улазном програму путем квалификатора за меморијске зоне коју су део Embedded C проширења Це језика [CSTEXT]. Ресурсна ограничења могу терати операнд у одређену ресурсну групу или подгупу, изражавати да два операнда морају бити у истој, или различитој, ресурсној групи, да два операнда морају имати исте индексе унутар две различите ресурсне групе итд. Свако ограничење

поседује приоритет који може имати разне вредности, али у суштини две су главне групе ограничења: обавезна и опциона. Скуп свих ресурсних ограничења представља улаз у фазу доделе ресурса.

Ресурсна ограничења се могу емитовати из било ког компајлерског модула, али три су главне тачке у току превођења из којих ограничења потичу. То су избор инструкција, распоређивач и конструкција графа сметњи. Избор инструкција и конструкција графа сметњи емитују само обавезна ограничења. Избор инструкција емитује ограничења која се тичу природе инструкција (неке инструкције раде само ако су одређени операнди одређене врсте ресурса), док се приликом конструкције графа сметњи емитују ограничења која представљају те сметње (за сваки пар операнада који су у сметњи емитује се по једно ограничење које изражава да та два операнда морају имати различите индексе, тј. адресе, унутар исте групе ресурса). Распоређивач, као што је већ речено, емитује ограничења која, ако су задовољена, омогућавају да одређене операције буду у паралели у једној машинској инструкцији. Додељивач ресурса није увек у стању да задовољи сва ограничења, шта више, неретко је скуп ограничења и теоретски незадовољив. Због тога након доделе ресурса распоређивач мора поново анализирати код, предложити нови распоред операција и проследити нова ограничења додељивачу ресурса. Ова петља се може поновити више пута, све док сва ограничења не буду задовољена. То представља везу распоређивача и додељивача ресурса у новој компајлерској инфраструктури. Начин на који се ограничења генеришу и задовољавају може се одрадити на разне начине, али само постојање подршке за такав механизам комуникације ових фаза превођења представља значајан елемент инфраструктуре.

5.5 Искоришћење физички подржаних петљи и адресних генератора

Два уобичајена елемента архитектуре ДСП-ова су физички подржане петље и адресни генератори. Препознавање петљи које могу бити реализоване као физички подржане и одређивање ефикасног искоришћења адресних генератора представљају тешке задатке у општем случају. Да би се ти елементи архитектуре успешно искористили за широк опсег могућих улазних програма морају се

користи врло сложени алгоритми, као на пример алгоритам заснован на апстрактној интерпретацији, програмским исечцима и политопским моделима [Lokucijewski], а чак и тада се могу десити случајеви који неће бити покривени. По питању физички подржаних петљи нови компајлер покрива само случајеве петљи које се могу пронаћи у DSP Stone скупу тестова [Zivojnovic]. Овај скуп тестова је направљен за мерење перформанци дигиталних сигнал процесора и њихове пропратне системске програмске подршке, и као такав садржи неке типичне имплементације учесталих операција у том домену. Показало се да је алгоритам који покрива случајеве који се у DSP Stone тестовима срећу релатино једноставан.

За сваку петљу у улазном Це коду која није преведана у физички подржану емитује се порука која саопштава програмеру зашто се то десило. Ако програмер зна да та петља ипак има особине које су потребне за превођење у физички подржану (да је добро, или веома добро, формирана петља [Ковачевић1]) онда на основу те поруке може знати како треба да промени код да би и компајлер препознао петљу и превео је у физички подржану. Овај приступ представља компромис између сложености оптимизације и напора корисничког програмера. Компромис је прихватиљив уколико нема превише случајева када се од програмера очекује додатни напор. Ова претпоставка је потврђена за нови Coyote 32 компајлер јер приликом компајлирања пет апликација за обраду HD (енд. high definition) звука (погледати поглавље са резултатима) све петље у коду су преведене у физички подржане, осим оних петљи које свакако по својој природи не би могле бити физички подржане.

Врло сличан приступ је примењен и за искоришћење адресних генератора. Употребљен је једноставнији алгоритам који покрива само неке случајеве (исто изведене из DSP Stone скупа тестова), али за сваки приступ меморији за који се не користи адресни генератор у излазном коду емитује се порука са одговарајућим информацијама. Од програмера се у тим случајевима обично очекује да промени приступ низу индексирањем у приступ преко показивача или да промени редослед приступа. За разлику од препознавања физички подржаних петљи, ова оптимизације није постигла искоришћење адресног генератора у

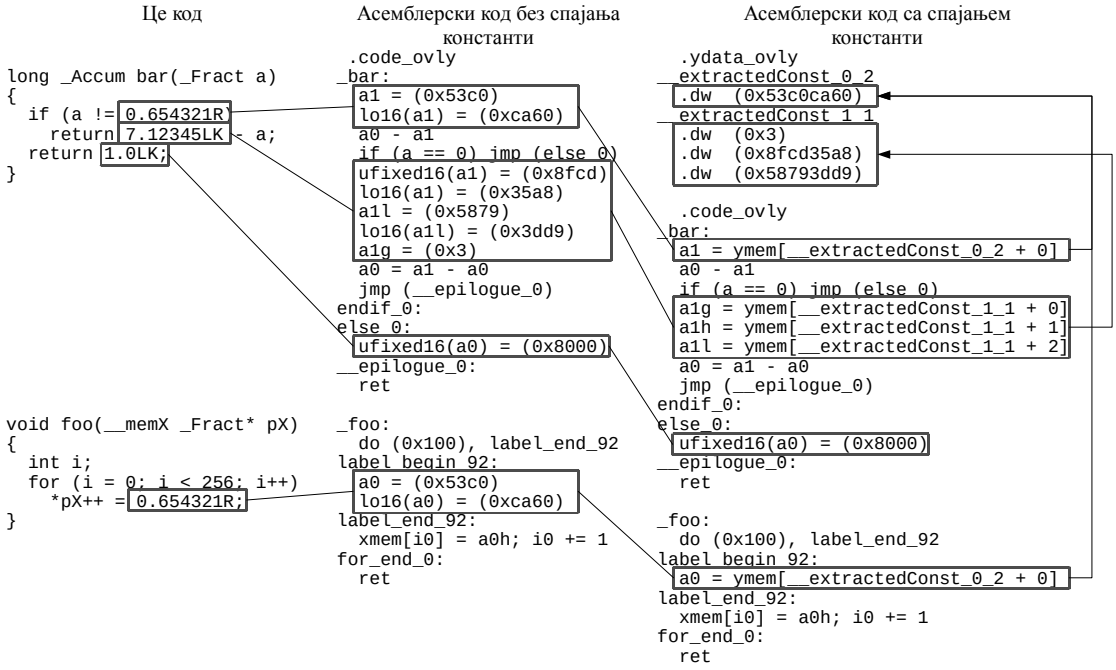
седам случајева који се срећу у поменутих пет апликација. Међутим, напор да се програмски код промени у складу са емитованим порукама је био прихватљив.

5.6 Глобалне оптимизације

Нови компајлер обавља и три глобалне оптимизације. Прва је аутоматско уграђивање функција, где компајлер бира функције чији ће позив бити уграђен у позивајућу функцију [Спасојевић1]. Уграђивање има за циљ да уклони препреку за друге оптимизације која настаје услед позива функције. Уграђивањем се та препрека потпуно уклања. Када уграђивање позива није повољно (због величине функције и њеног коришћења на више места) тада друга глобална оптимизација покушава да смањи цену позива помоћу међупроцедуралне доделе регистара, уколико је доступан код и позивајуће и позване функције. Ове оптимизације је било једноставно извести због добре представе међупроцедуралних односа у компајлерској инфраструктури

Трећа глобална оптимизација је спајање константи. Оптимизација пописује све константе које се користе у програму и за сваку различиту вредност формира јединствену меморијску локацију којој ће се приступати на сваком месту на којем се та вредност користи. Ова оптимизације је корисна на оваквим врстама циљне архитектуре јер је у општем случају ефикасније учитати вредност у регистар из меморије него користити непосредне операнде у инструкцијама. На пример, постављање 32битне вредности у регистар коришћењем непосредних операнада захтева две инструкције, од којих једна поставља горњих 16, а друга доњих 16 бита, док је за ту исту ствар довољна једна инструкција која чита једну реч из меморије. Дакле, ако се константа постави у меморију за податке, а не у програмску меморију, заузеће програмске меморије ће бити мање за једну инструкцију чак и ако се конкретна константа користи само на једном месту. Доборбит је још већа у случајевима већих константи, рецимо 64-битних или 72-битних. Корисник може да бира у коју меморију ће бити смештене константе. Пример рада ове оптимизације дат је на слици 5.6.1. Види се добробит коју оптимизација доноси у случају 72-битне константе (литерала), и у случају 32-битне константе која се користи на два места (када се прави само један простор у меморији за саму вредност који се користи на два различита места). Видљиво је

и да константа неће бити похрањена у меморију ако ју је могуће учитати у једној инструкцији (литерал 1.0LK као повратна вредност функције ***bar***). Величина програмског кода је у датом примеру смањења за 5 инструкција, али је убрзање извршавања кода приметније јер се тело петље у функцји ***foo*** смањило за једну инструкцију.



Слика 5.6.1 Илустрација рада глобалне оптимизације за спајање константи.

Све глобалне оптимизације ће дати боље резултате ако се позову над целим програмом, мада раде и ако се програм преводи део по део (датотеку по датотеку). При једном позиву компајлера могуће је проследити више датотека са изворним кодом као улаз и оне ће све заједно бити преведене. Тако је могуће проследити цео програм, чиме би се глобалне оптимизације максимално искористиле. Наравно, на овај начин у глобалне оптимизације није могуће укључити део програма који се налази у библиотекама или додатним објектним датотекама (у случају мешања Це и асемблерских функција). Смештање међурепрезентације у бинарном облику у библиотеке (које су у својој суштини скупови објектних датотека) и довршавање компајлирања током повезивања би премостило ову препреку за случајеве када библиотеке настају од Це кода. Међутим, код наменских система библиотеке готово увек потичу од асемблерског кода, па би корист од оваквог приступа била мала. Са друге стране, препрека која настаје на прелазу између асемблерске и Це функције може бити

премоштена саопштавањем потребних информација о асемблерској функцији кроз неки механизам прихватљив компајлеру. У овој инфраструктури изабран је механизам прагми. Сваком прототипу може бити придружена прагма која описује како та функција користи ресурсе циљне платформе, а та информација може бити искоришћена приликом глобалних оптимизација. Наравно, програмер је у потпуности одговоран за тачност података наведених у прагми.

На слици 5.6.2 илустрована је разлика у генерисаном коду у зависности од постојања поменуте прагме. Када прагма не постоји компајлер претпоставља основну позивну конвенцију, по којој се не гарантује очување регистра **i0** приликом позива функције ***bar***. Пошто се, исто по позивној конвенцији, први аргумент показивачког типа прослеђује у регистру **i0**, генерисани код смешта вредност из **i0** у **i2** јер је **i2** регистар чија вредност ће, по конвенцији, увек бити сачувана. Због тога и функција ***foo*** мора сачувати **i2**, што укупно даје додатне 4 инструкције. На десној страни слике се види исти тај код али у којем се прагмом саопштава да функција мења само вредности у **x0** и **i5**. Са тим знањем компајлер може безбедно да користи **i0** у функцији и код је краћи за 4 инструкције.

```
void bar();

__memY _Fract y;

void foo(__memX _Fract* pX)
{
  int i;
  for (i = 0; i < 256; i++)
  {
    bar();
    *pX++ = y;
  }
}
```

```
_foo:
  xmem[i7] = i2; i7 += 1
  i2 = i0
  do (0x100), label_end_92
label_begin_92:
  call (_bar)
  a0 = ymem[_y + 0]
label_end_92:
  xmem[i2] = a0h; i2 += 1
for_end_0:
  i7 -= 1
  i2 = xmem[i7]
  ret
```

```
#pragma ASM_FUNC_CLOBBERED_REGS("x0,i5")
void bar();

__memY _Fract y;

void foo(__memX _Fract* pX)
{
  int i;
  for (i = 0; i < 256; i++)
  {
    bar();
    *pX++ = y;
  }
}
```

```
_foo:
  do (0x100), label_end_92
label_begin_92:
  call (_bar)
  a0 = ymem[_y + 0]
label_end_92:
  xmem[i0] = a0h; i0 += 1
for_end_0:
  ret
```

Слика 5.6.2 Пример употребе прагме за саопштавање ресурса које функција мења

5.7 Вишеструко превођење

Компајлерска инфраструктура поседује модул за клонирање међурепрезентације у било ком тренутку. Прављење копије међурепрезентације је врло корисно за испробавање различитих стратегија превођења [Спасојевић2]. У новом Coyote 32 компајлеру свака функција се клонира у одређеном тренутку у задњем делу. Различите оптимизационе стратегије се примењују на различитим клоновима, а на крају се задржава само најбољи резултат. Овај механизм омогућава једноставно и лако решење у случајевима када је потребно донети неке компликоване одлуке у оптимизационим алгоритмима. Уместо да се алгоритми усложњавају деловима који покушавају да унапред одреде која би била најбоља одлука (што често није ни могуће), много је лакше једноставно пробати више могућности и видети која је најбоља. Тренутно се цело превођење одвија као један програмски процес, али би се у будућности испробавање сваке различите стратегије могло одвојити у независне задатке који би се могли извршавати паралелно на вишејезгарским системима.

ПОГЛАВЉЕ 6.

РЕЗУЛТАТИ И ДИСКУСИЈА

Да би се квантификовало побољшање обављено је неколико мерења на полазном компајлеру и новом компајлеру. Главна мера квалитета кода је његова величина, док брзина извршавања није мерена. Но, имајући у виду да ниједна оптимизација, ни код једног компајлера, не даје предност брзини кода на рачун величине (као рецимо одмотавање петљи) може се закључити да су величина и брзина кода добро корелирани. Следећих пет апликација за обраду HD звука је употребљено за мерења: две апликације за обраду јачине звука, две вишеканалске виртуализационе апликације и једна апликација сложене обраде (која садржи неколико различитих алгоритама за обраду).

Апликације	Полазни компајлер	Нови компајлер	Релативно побољшање
Обрада јачине звука 1	1329	954	30%
Виртуализациона 1	549	467	18%
Виртуализациона 2	3259	2791	17%
Контрола јачине звука 2	4428	3751	18%
Сложена обрада	4733	3975	19%

Табела 6.1 Величине преведеног кода и процентуално смањење

Табела 6.1 приказује величине резултујућег кода за свих пет апликација када су све оптимизације укључене у оба компајлера. Очевидно је да за већину апликација нови компајлер производи око 18% мањи код, са максимумом од 30% за прву апликацију.

Апликације	Без глобалних оптимизација	Са глобалним отимизацијама	Релативни допринос
Обрада јачине звука 1	997	954	5%
Виртуализациона 1	492	467	5%
Виртуализациона 2	2895	2791	4%
Контрола јачине звука 2	3801	3751	1%
Сложена обрада	4129	3975	4%

Табела 6.2 Допринос глобалних оптимизација смањењу величине кода

Табела 6.2 приказује допринос глобалних оптимизација: међупроцедуралне доделе регистара, аутоматског уграђивања функција и спајања констати, са смештањем у меморију за податке. Допринос је исказан релативно у односу на величину кода када су укључене све оптимизације осим глобалних оптимизација. Резултати показују да глобалне оптимизације смањују величину кода за 4 до 5%. Код друге апликације за обраду јачине звука допринос је само 1%. То се приписује чињеници да та апликација има мали број функција и констати у односу на своју величину.

Апликације	Без оптимизација преписивачким правилима	Са оптимизацијама преписивачким правилима	Релативно побољшање
Обрада јачине звука 1	1067	954	12%
Виртуализациона 1	493	467	6%
Виртуализациона 2	3012	2791	8%
Контрола јачине звука 2	4166	3751	11%
Сложена обрада	4353	3975	9%

Табела 6.3 Допринос малих оптимизација које су изведене преписивачким правилима

У табели 6.3 се може видети колико се величина кода смањује због оптимизација које су имплементиране коришћењем преписивачких правила. Те оптимизације обухватају смањење сложености операција, алгебарска поједностављивања и неке оптимизационе трансформације међурепрезентације које су специфичне за циљну архитектуру. Допринос тих малих оптимизација је исто исказан релативно у односу на величину кода када су укључене све

оптимизације осим оних чији допринос се мери. Оптимизациона правила смањују величину кода за 6 до 12%. Када се овај резултат сагледа у контексту напора уложеног у писање тих оптимизација може се закључити да реализовање ових оптимизација кроз механизам правила представља добар однос доприноса и уложеног напора.

Апликације	Стратегија 1	Стратегија 2	Боља од две стратегије за сваку функцију
Обрада јачине звука 1	998	1008	954
Виртуализациона 1	487	469	467
Виртуализациона 2	2831	2803	2791
Контрола јачине звука 2	3792	3801	3751
Сложена обрада	4020	4010	3975

Табела 6.4 Детаљни приказ резултата за пример коришћења вишеструког превођења

Допринос вишеструког превођења анализиран је посебно. Компајлер је подешен да користи само две различите стратегије. Разлика је била у фази доделе ресурса. Табела 6.4 приказује три резултата за сваку апликацију: први резултат представља величину кода у случају када се за сваку функцију примењује стратегија 1, други резултат представља величину при примени стратегије 2 за сваку функцију, а трећи резултат се добија када се за сваку функцију изабере мањи од два кода произведена применом те две стратегије. Ови резултати показују корисност овог приступа, али постоји још простора да се додају нове стратегије и то ће бити један од задатака даљег унапређења компајлера.

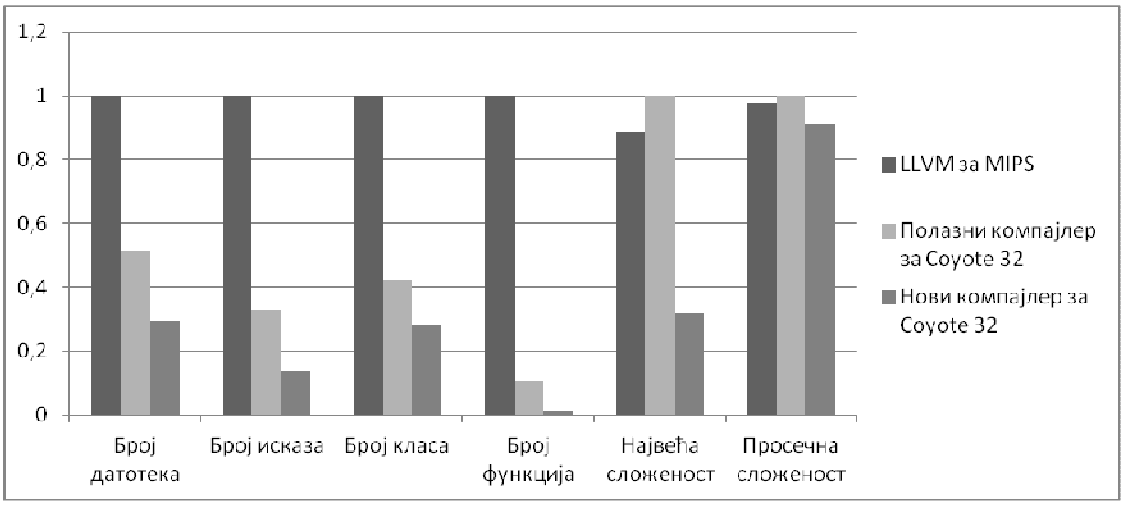
Апликације	Полазни компајлер	Нови компајлер	Просечни проценат паралелних инструкција
Обрада јачине звука 1	8%	14%	15%
Виртуализациона 1	10%	13%	
Виртуализациона 2	15%	19%	20%
Контрола јачине звука 2	13%	18%	
Сложена обрада	17%	21%	нема

Табела 6.5 Удео паралелних инструкција

Напослетку је дата анализа паралелизма на нивоу инструкција у излазном коду. У ту сврху мерен је удео паралелних инструкција (инструкција које имају више од једне операције) у свакој апликацији. У раду [Povazan1] иста методологија је примењена и на великом скуп апликација које су ручно писане у асемблерском језику. Ова анализа је показала да постоји разлика у просечном уделу паралелних инструкција између различитих група апликација. Рецимо, показује се да апликације које обрађују јачину звука имају врло близак проценат паралелних инструкција, али који је различит од процента паралелних инструкција код, рецимо, виртуализационих апликација. У табели 6.5 је у прве две колоне дат измерени проценат паралелних инструкција за полазни и нови компајлер. У трећој колони су дати просечни проценти паралелних инструкција у ручно писаном асемблерском коду за те врсте апликација (не постоји једино мера за сложене апликације, јер оне нису анализиране у [Povazan1]). Из приказаног се види да је нови компајлер не само побољшао искоришћење инструкционог паралелизма у односу на полазни компајлер, већ и да се тесно приближио ручно писаном коду. Уочава се и да је највећи напредак у паралелизацији остварен код прве апликације за обраду јачине звука, што је главни разлог за значајно веће укупно смањење величине кода приказано у табели 6.1.

Наведени резултати потврђују да компајлер заснован на новој инфраструктури генерише квалитетнији код и боље одговара захтевима циљне физичке архитектуре. Осим тога, циљ је био и да нови компајлер буде мањи и једноставнији за разумевање и одржавање. У табели 6.6 и на слици 6.1 приказане

су неке мере везане за изворни код полазног компајлера, новог компајлера, али и LLVM компајлера. Анализирани су само задњи делови компајлера, а наведене су величине изворног кода изражене у броју наредби, као и број датотека изворног кода, број класа и функција. Дате су још и максималне и средње вредности Мек Кејбове сложености (енг. McCabe complexity, cyclomatic complexity), у складу са [McCabe]. Табела 6.6 садржи конкретне вредности, док слика 6.1 илуструје односе тих вредности између три компајлера.



Слика 6.1 Приказ односа мера комплетног изворног кода три компајлера. Вредности су изражене релативно у односу на највећу вредност код сваке метрике.

Компајлери	Број датотека	Број исказа	Број класа	Број функција	Највећа Мек Кејбова сложеност	Просечна Мек Кејбова сложеност
LLVM за MIPS	1591	375.054	3396	3997	330	3,32
Полазни компајлер за Coyote 32	814	123.579	1443	419	371	3,41
Нови компајлер за Coyote 32	470	51.620	953	52	118	3,10

Табела 6.6 Мере комплетног изворног кода три компајлера.

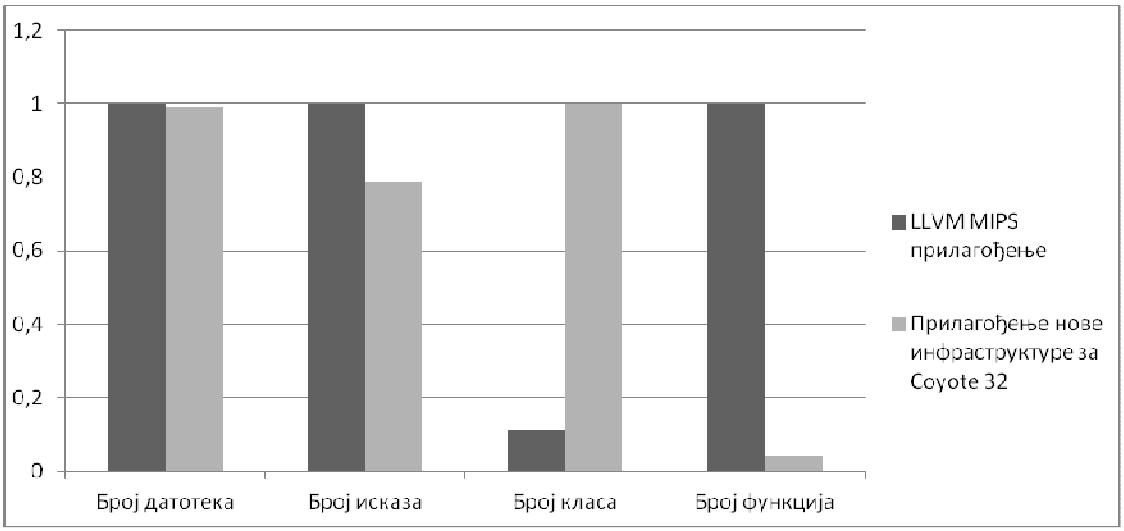
Мере су добијене коришћењем бесплатног програмског алата SourceMonitor. При пребројавању датотека у обзир су узимане .cpp и .h датотеке. Код LLVM-а су у урачунате и .td датотеке, које садрже опис циљне архитектуре и правила за избор инструкција, као што је код полазног компајлера урачуната .brg датотека са описом шаблона за поплочавање стабла међукода и кодом за

прављење излазне листе инструкција. Међутим, то не утиче много на крањи резултат јер постоји 21 .td датотека, и свега једна .brg датотека. Под бројем исказа мисли се на Це++ исказе, док је приликом тумачења кода написаног на посебним језицима за опис архитектуре и избора инструкција (који се налази у .td и .brg датотекама) иместо броја исказа у обзир узиман број линија кода (не рачунајући празне линије). Дакле, крајњи резултат је збир броја Це++ исказа и броја линија кода написаног на тим посебним језицима. Као што се може претпоставити на основу броја дататотека које садрже код на посебним језицима, ово не утиче много на крајњи резултат. Мек Кејбова сложеност се, поједностављено речено, формулише за једну функцију или методу као број наредби гранања у њеном телу, плус 1 (јер код без иједног гранања, по овој дефиницији, има сложеност 1). Интересантно је напоменути да се и на број исказа, датотека, класа и слично, исто може гледати као на мере сложености [Stein].

Из резултата у табели 6.6. и на слици 6.1 се види да је од сва три компајлера, нови компајлер, заснован на новој инфрастуктури, најмање обимности. И по броју линија и по броју исказа, око двоструко је мањи од полазног компајлера, а вишеструко мањи од LLVM-а. По броју класа су резултати мало приближнији, али и даље нови компајлер има за трећину мање класа од полазног и три пута мање класа од LLMV-а. Слично је и по питању броја функција, мада је тешко те резултате коментарисати јер је то делимично питање стила, где је код новог компајлера била изражена тенденција ка малој употреби функција. Просечна Мек Кејбова комплексност је врло приближна код три посматрана компајлера, али је код новог компајлера ипак најмања. По [McConnell] свака вредност испод 5 сматра се добром, али то се односи на појединачне методе или функције. Из податка о максималној сложености види се да неке методе или функције у полазном компајлеру и LLVM-у имају сложеност и преко 300, док је код новог компајлера максимална сложеност 118. Високе вредности Мек Кејбове комплексности говоре да би се већа пажња морала посветити тој методи или функцији, посебно са спекта испитивања њене исправности.

У директном поређењу са полазним компајлером, види се да је нови компајлер неспорно једноставнији и мањи, док даје боље резултате по питању квалитета генерисаног кода. Поређење новог компајлера са LLVM-ом не може бити директно јер, осим што не циљају исте архитектуре (па није могуће говорити о томе који генерише бољи код), ова два компајлера немају исти скуп функционалности. Ипак, из добијених резултата се може са великом поузданошћу закључити да је нови компајлер једноставнији од LLVM-а, што је врло повољно са становишта писања новог компајлера, јер једноставнија инфраструктура подразумева једноставнију употребу, одржавање и слично.

Део ових метрика примењен је и посебно на платформско прилагођење компајлера (део кода који описује процесорску архитектуру и користи инфраструктуру на начин који одговара циљној платформи). Код полазног компајлера постоји тесна веза између платформски независног дела и прилагођења, који ни приликом иницијалног развоја полазног компајлера нису плански раздвајани, па је тешко одредити који део изворног кода представља прилагођење. Због тога су на овај начин анализирани само нови компајлер и LLVM. Резултати су дати у табели 6.7 и на слици 6.2.

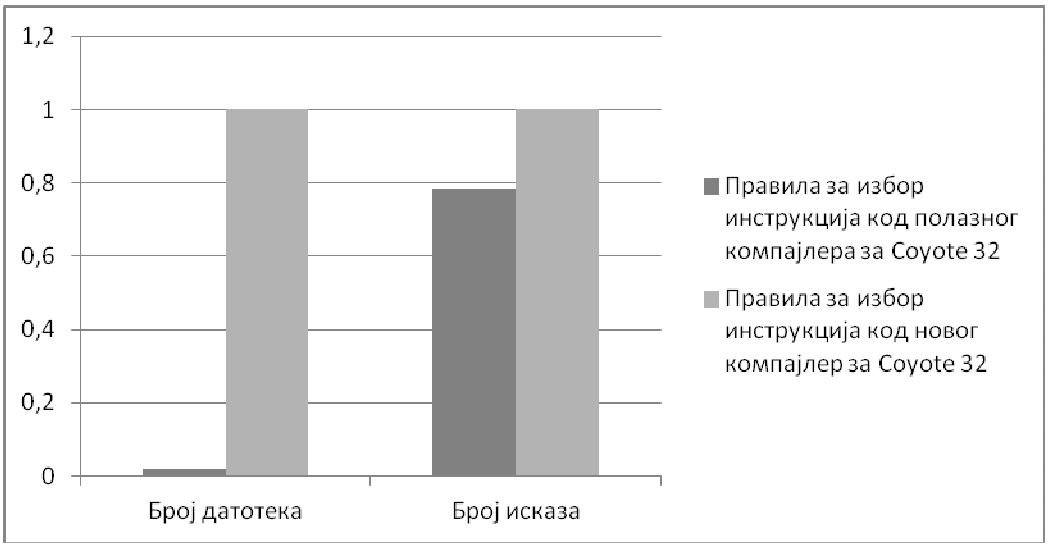


Слика 6.2 Приказ односа мера изворног кода делова за платформско прилагођење компајлера. Вредности су изражене релативно у односу на највећу вредност код сваке метрике.

Компајлери	Број датотека	Број исказа	Број класа	Број функција
LLVM за MIPS	113	31.747	83	196
Нови компајлер за Coyote 32	112	24.984	739	8

Табела 6.7 Мере изворног кода делова за платформско прилагођење компајлера.

Као што је већ споменуто, део LLVM компајлера за MIPS платформу су и 21 .td датотека написана у посебном језику за опис архитектуре и избор инструкција. Те датотеке су део платформског прилагођења и урачунате су у мерења. Упадљиво је да по броју класа нови компајлер одскаче од LLVM-а. То је због тога што се скоро свако преписивачко правило у новом компајлеру гради прављењем нове класе. Укупно има 711 класа за преписивачка правила, а само 28 класа за остала прилагођења. Ако се то има у виду, онда је закључак да је код за прилагођење мањи код новог компајлера по свим метрикама. Интересантно је размотрити да ли то значи да нова компајлерска структура омогућава краћи запис потребних информација за прилагођење, и поред тога што LLVM поседује посебан језик за опис архитектуре и избор инструкција. Циљне платформе нису исте и самим тим количина потребних инфромација не мора бити иста. Пошто обе архитектуре поседују 32-битну инструкциону реч, могло би претпоставити да је и количина потребних инфморација приближно слична. Међутим, у LLVM-у се у коду за прилагођење описују и бинарни облици инструкција, а и поред основног MIPS инструкционог скупа покривена су и нека проширења, као што су MicroMISP и MipsDSP. Осим тога, део прилагођења се односи и на конкретизацију одређених компајлерских анализа, тако да ако LLVM подржава више анализа онда очекује и више информација за прилагођење. Због тога се не може олако тврдити да нова инфраструктура омогућава краћи запис, али се ипак намеће закључак да поред предности које има запис правила у новој инфраструктури (види поглавље 5.3), концизност записа није значајно мања него када се користи посебан језик. Ову тезу потврђује и поређење само правила за избор инструкција код полазног и новог компајлера. Издвајање правила за избор инструкција се код полазног компајлера могло јасно одрадити јер су правила записана у посебној датотеци и у посебном језику који разуме IBURG алат. Поређење је приказано у табели 6.8 и на слици 6.3.



Слика 6.3 Приказ односа мера изворног кода који описује правила за избор инструкција. Вредности су изражене релативно у односу на највећу вредност код сваке метрике.

Компајлери	Број датотека	Број исказа
Полазни компајлер за Coyote 32	1	10.301
Нови компајлер за Coyote 32	53	12.127

Табела 6.8 Мере изворног кода који описује правила за избор инструкција.

У овом случају циљна платформа је иста и количина информација које је потребно саопштити се може сматрати једнаком за оба компајлера. Резултат показује да је запис правила коришћењем посебног језика краћи за око 20%, што се може сматрати релативно малом разликом. Овакви резултати су у складу и са правилима приказаним на слици 5.3.1, где се види да су величине правила врло сличне.

ПОГЛАВЉЕ 7.

ЗАКЉУЧАК

Општи закључак је да је предложена нова компајлерска инфраструктура успешно одговорила на постављене захтеве. Инфраструктура је недвосмислено олашала изражавање компајлерских поступака каратеристичних за наменске архитектуре. То је постигнуто уграђивањем у саму инфраструктуру следећих елемената:

- механизма за изражавање везе између фаза помоћу ресурсних ограничења, што је важно за добро искоришћење паралелизма на нивоу инструкције

- могућности да се при моделовању циљне архитектуре искажу сложени односи различитих ресурсних група и подгрупа, што повећава изражајност ресурсних ограничења

- механизма преписивачких правила, који омогућава избор инструкција над графом и једноставно изражавање малих оптимизација

- могућности вишеструког превођења, које омогућава проверу више различитих стратегија у случајевима када је тешко у напред оценити квалитет оптимизационих одлука

- механизма за емитовање повратних информација о учинку оптимизација, што отвара простор за употребу једноставнијих оптимизација пребацивањем дела посла на програмера

- механизма изражавања паралелизма на нивоу инструкције помоћу скупа паралелизационих случајева и перколационог распоређивача, који не захтева формулисање критеријума распоређивања

Лакоћи одржавања допринели су следећи елементи:

- постојање јединственог програмског модела међурепрезентације, што омогућава употребу истих знања и поступака у свим фазама превођења

- изражавање преписивачких правила директно у Це++-у, што убрзава превођење компајлера и олакшава проналажење грешака

- придруживање свакој инструкцији листе имена правила чија примена је довела до њеног формирања

- постојање једноставног механизма за запис тренутног стања међурепрезентације у датотеку

Квалитет генерисаног кода је потврђен кроз поређење са резултатима полазног компајлера, док је једноставност компајлера заснованог на новој инфраструктури потврђена кроз поређење објективних мера сложености са полазним комапјлером и LLVM компајлером. На тај начин потврђена је и полазна хипотеза, јер је направљн бољи компајлер, једноставнијег кода, а доказан и у индустријској употреби.

Додатну потврду исправности ове инфраструктуре представља чињеница да је у току њена примена на још једну циљну платформу из класе процесора за дигиталну обраду сигнала. За ту нову циљну платформу такође већ постоји компајлер писан од стране реномиране фирме из те области, тако да за неко наредно истраживање остаје поређење перформанси тог компајлера са перформансама компајлера написаног коришћењем ове инфраструктуре. Резултати изнети у овој докторској дисертацији представљају потврду да предложена инфраструктура нуди предности у односу на компајлерске технике за процесоре опште намене, али ти нови резултати би представљали проверу и у односу на друге компајлере који користе технике за наменске процесоре. Такође, било би интересантно проверити и примену ове компајлерске инфраструктуре код компајлера за процесоре опште намене. У том смислу, довршавање прилагођења за MIPS процесор омогућило би упоређивање са LLVM-ом и GCC-ом.

Један од праваца даљег побољшања инфраструктуре био би увођење додатних модула за анализу кода. Неке статичке анализе кода које су корисне код наменских процесора описане су у [Kienle]. Остали правци обухватају даље унапређење споја фаза распоређивања и доделе ресурса, са могућим укључивањем и фазе избора инструкција, као и додавање нових стратегија у режиму вишеструког превођења. Осим тога, остаје да се анализира и у пракси потврди употребљивост предложене инфраструктуре за додавање у компајлер могућности да помогне корисницима да обављају неке од уобичајених доменских трансформација кода, као што је код дигиталне обраде сигнала трансфорамција аритметике са покретном зарезу у аритметику са непокретним зарезом [Barleanu].

Литература

[Aho] A. V. Aho, M. S. Lam, R. Sethi, J. D. Ullman, Compilers: principles, techniques, & tools, Second edition, Addison-Wesley, 2007.

[Appel] A. W. Appel. Modern compiler implementation in C, Cambridge University Press, 2004.

[Barleanu] A. Barleanu, V. Baitoiu, A. Stan, "Digital filter optimization for C language," Advances in Electrical and Computer Engineering, Vol. 11, no. 3, pp. 111-114, 2011, doi: 10.4316/AECE.2011.03018

[Bashford] S. Bashford, R. Laupers, “Phase-Coupled Mapping of Data Flow Graphs to Irregular Data Paths”, Design Automation for Embedded Systems, Volume 4, Issue 2-3, pp. 119–165, 1999, doi: 10.1023/A:1008966522714

[Bertin] V. Bertin, J. Daveau, P. Guillaume, T. Lepley, D. Pilat, C. Richard, M. Santana, T. Thery, “FlexCC2: An Optimizing Retargetable C Compiler for DSP Processors”, Lecture Notes in Computer Science Volume 2491, pp. 382-398, 2002. doi: 10.1007/3-540-45828-X_28

[Chen] G. Chen, M. Kandemir, “Optimizing embedded applications using programmer-inserted hints”, Design Automation Conference, 2005. Proceedings of the ASP-DAC 2005. Asia and South Pacific, Vol. 1, pp. 157-160, 2005, doi: 10.1109/ASPDAC.2005.1466149

[Choi] Y. Choi, T. Kim, “Address assignment in DSP code generation - an integrated approach”, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 22, Issue 8, pp. 976-984, 2003, doi: 10.1109/TCAD.2003.814955

[CLDSP] Cirrus Logic 32-bit DSP Assembly Programmer’s Guide, September 2013, Available: http://www.cirrus.com/en/pubs/manual/32bit_DSP_Assembly_Guide_UM11.pdf

[Cooper] K. D. Cooper, L. Torczon. Engineering a compiler, Morgan Kaufmann, 2004.

[CST99] JTC1/SC22/ WG14/N1256, Programming languages - C, Technical report, ISO/IEC, 2007.

[CSTEXT] JTC1/SC22/WG14, Programming languages - C - Extensions to support embedded processors, Technical report, ISO/IEC, 2006.

[Deilmann] M. Deilmann, “A Guide to Vectorization with Intel C++ Compilers”, Available: http://download-software.intel.com/sites/default/files/m/d/4/1/d/8/CompilerAutovectorizationGuide.pdf

[Djukic1] M. Djukic, N. Cetic, R. Obradovic, M. Popovic, “An approach to instruction set compiled simulator development based on a target processor C compiler back-end design”, Innovations in Systems and Software Engineering: Volume 9, Issue 3, pp. 135-145, 2013, doi: 10.1007/s11334-013-0220-0

[Djukic2] M. Djukic, M. Popovic, N. Cetic, I. Povazan, “Embedded system oriented compiler infrastructure”, Advances in Electrical and Computer Engineering, Vol. 14, No. 3, pp. 123-130, 2014, doi: 10.4316/AECE.2014.03016

[Ebner] D. Ebner, F. Brandner, B. Scholz, A. Krall, P. Wiedermann, A. Kadlec, “Generalized instruction selection using SSA-graphs”, ACM SIGPLAN Notices - LCTES '08, Vol. 43, Issue 7, pp. 31-40, 2008, doi: 10.1145/1379023.1375663

[Eriksson] M. Eriksson, C. Kessler, “Integrated Code Generation for Loops”, ACM Transactions on Embedded Computing Systems, Vol. 11S, Issue 1, No. 19, 2012, doi: 10.1145/2180887.2180896

[Fisher] J. A. Fisher, P. Faraboschi, C. Young. Embedded computing: A WLIW approach to architecture, compilers, and tools. Morgan Kaufmann, 2005.

[Fraser1] C. W. Fraser, R. R. Henry, T. A. Proebsting, “BURG – Fast optimal instruction selection and tree parsing”, ACM SIGPLAN Notices, Volume 27, Issue 4, pp. 68-76, 1992, doi: 10.1145/131080.131089

[Fraser2] C. W. Fraser, D. R. Hanson, T. A. Proebsting, “Engineering a simple, efficient code generator generator”, ACM Letters on Programming Languages and Systems, Volume 1, Issue 3, pp. 213-226, 1992, doi: 10.1145/151640.151642

[Fursin] G. Fursin, O. Temam, “Collective optimization: a practical collaborative approach”, ACM Transactions on Architecture and Code Optimization, Volume 7, No. 4, Article 20, 2010, doi: 10.1145/1880043.1880047

[Grewal] G. W. Grewal, C. T. Wilson, “Mapping reference code to irregular DSPs within the retargetable, optimizing compiler COGEN(T)”, Proceedings 34th ACM/IEEE International Symposium on Microarchitecture, pp. 192-202, 2001, doi: 10.1109/MICRO.2001.991118

[Guilan] D. Guilan, Z. Suqing, T. Jinlan, J. Weidu, “A Study of Compiler Techniques for Multiple Targets in Compiler Infrastructures”, ACM SIGPLAN Notices, Volume 37, Issue 6, pp. 45-51, 2002, doi: 10.1145/571727.571735

[Hamel] L. H. Hamel, “Industrial strength compiler construction with equations”, ACM SIGPLAN Notices, Volume 27, Issue 8, pp. 43-50, 1992, doi: 10.1145/142137.142145

[Johnson] T. A. Johnson, S. I. Lee, L. Fei, A. Basumallik, G. Upadhyaya, R. Eigenmann, S. P. Midkiff, "Experiences in Using Cetus for Source-to-Source Transformations", Languages and Compilers for High Performance Computing, pp. 1-14, 2005, doi: 10.1007/11532378_1

[Kienle] H. M. Kienle, J. Kraft, Thomas Nolte, “System-specific static code analyses: a case study in the complex embedded systems domain”, Software Quality Journal, Vol. 20, Issue 2, pp. 337-367, 2012, doi: 10.1007/s11219-011-9138-7

[Kulkarni] P. A. Kulkarni, D. B. Whalley, G. S. Tyson, J. W. Davidson, “Practical Exhaustive Optimization Phase Order Exploration and Evaluation”, ACM Transactions on Architecture and Code Optimization, Vol. 6, Issue 1, Article 1, 2009, doi: 10.1145/1509864.1509865

[Leupers1] R. Leupers, “Code generation for embedded processors”, in Proc. 13th International Symposium on System Synthesis (ISSS'00), Madrid, 2000, doi: 10.1109/ISSS.2000.874046

[Leupers2] R. Leupers, “Compiler design issues for embedded processors”, IEEE Design & Test of Computers, Volume 19, Issue 4, pp. 51-58, 2002, doi: 10.1109/MDT.2002.1018133

[Lokuciejewski] P. Lokuciejewski, D. Cordes, H. Falk, P. Marwedel, "A Fast and Precise Static Loop Analysis Based on Abstract Interpretation, Program Slicing and Polytope Models", Proceedings of the 7th annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '09), Washington, pp. 136-146, 2009, doi: 10.1109/CGO.2009.17

[McCabe] T. J. McCabe, “A complexity measure”, IEEE Transactions on Software Engineering, SE-2, vol. 4, pp. 308-320, 1976.

[McConnell] S. C. McConnell, “Code complete: A practical handbook of software construction”, 2nd edition, Microsoft Press, 2004.

[Nicolau] A. Nicolau, R. Potasman, “Realistic scheduling: compaction for pipelined architectures”, MICRO 23 Proceedings of the 23rd annual workshop and symposium on Microprogramming and microarchitecture, Orlando, pp. 69-79, 1990, doi: 10.1145/255237.255252

[Povazan1] I. Považan, M. Popovic, M. Đukic, and M. Krnjetin, “Measuring the quality characteristics of an assembly code on embedded platforms”, Telfor Journal, Volume 4, No. 1, 2012, doi: 10.1109/TELFOR.2011.6143798

[Povazan2] I. Povazan, M. Popovic, M. Djukic, N. Cetic, “A retargetable C compiler for embedded systems”, ECBS-EERC 2013, Budapest, Hungary, 2013. doi: 10.1109/ECBS-EERC.2013.15

[Rajagopalan] S. Rajagopalan, S. P. Rajan, S. Malik, S. Rigo, G. Araujo, K. Takayama, “A retargetable VLIW compiler framework for DSPs with instruction-level parallelism”, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 20, Issue 11, pp. 1319-1328, 2001, doi: 10.1109/43.959861

[Stein] C. Stein, G. Cox, L. Etzkorn, “Exploring the relationship between cohesion and complexity”, Journal of Computer Science 1(2), pp. 137-144, 2005.

[Talavera] G. Talavera, M. Jayapala, J. Carrabina, F. Catthoor, “Address Generation Optimization for Embedded High-Performance Processors: A Survey”, Journal of Signal Processing Systems, Vol. 53, Issue 3, pp. 271-284, 2008, doi: 10.1007/s11265-008-0165-y

[White] D. R. White, A. Arcuri, J. A. Clark, “Evolutionary Improvement of Programs”, IEEE Transactions on Evolutionary Computation, Vol. 15, No. 4, pp. 515-538, 2011, doi: 10.1109/TEVC.2010.2083669

[Wolfe] M. Wolfe, “How compilers and tools differ for embedded systems”, Proc. 2005 international conference on Compilers, architectures and synthesis for embedded systems, New York, 2005. doi: 10.1145/1086297.1086298

[Zivojnovic] V. Zivojnovic, J.M. Velarde, C. Schlager, H. Meyer, "DSP-stone: A DSP-oriented benchmarking methodology", Proceedings of International Conference on Signal Processing Applications and Technology, 1994, pp. 715-720.

[Ђукић] М. Ђукић, В. Ковачевић, „Једно решење модификације Це компајлера са линијском структуром у циљу повећања искоришћења инструкционог паралелизма“, Зборник радова 51. конференције ЕТРАН, Херцег Нови – Игало, Јун 2007.

[Живковић] М. Живковић, В. Ковачевић, Д. Мишковић, „Једно решење распоређивача иструкција преводиоца за процесор за дигиталну обраду сигнала“, Зборник радова 51. конференције ЕТРАН, Херцег Нови – Игало, Јун 2007.

[Зарић] З. Зарић, „Један концепт оптимизационе технике за искоришћење адресне јединице у Це компајлеру“, дипломски-мастер рад, ФТН, Нови Сад, октобар 2008.

[Ковачевић1] В. Ковачевић, М. Поповић, „Системска програмска подршка у реланом времену 1“, ФТН издаваштво, Нови Сад 2011.

[Ковачевић2] В. Ковачевић, М. Поповић, М. Темеринац, Н. Теслић, „Архитектуре и алгоритми дигиталних сигнал процесора 1“, ФТН Издаваштво, Нови Сад 2005.

[Спасојевић1] Б. Спасојевић, М. Ђукић, З. Зарић, Ј. Ковачевић, „Анализа корисности интерпоцедуралних компајлерских оптимизација код програма за дигиталне сигнал процесоре“, Зборник радова 55. конференције ЕТРАН, Бања Врућица, Јун 2011.

[Спасојевић2] Б. Спасојевић, М. Ђукић, З. Зарић, М. Крунић, М. Поповић, „Једно решење претраживања простора одлука при синтези кода у компајлеру“, Зборник радова 19. конференције ТЕЛФОР, Београд, Новембар 2011. doi: 10.1109/TELFOR.2011.6143851

Додатак А

Преглед записа стања међурепрезентације у одабраним тренуцима током једног тока превођења

У овом додатку биће дат приказ тока превођења кроз увид у исписе стања међурепрезентације у одабраним тренуцима. Овај приказ има два циља. Први циљ је да илуструје поступак превођења и понашање неких компоненти и оптимизација. Други циљ је да покаже способност компајлера да у било ком тренутку једноставно запише међурепрезентацију, што је омогућено постојањем јединствене програмске структуре репрезентације кроз цео задњи део компајлера.

Це код програма који ће бити коришћен за овај приказ дат је на слици А.1 и представља скаларни производ два вектора познате дужине. Аритметика се обавља у непокреном зарезу.

```
long _Accum foo(__memX _Fract* pX, __memY _Fract* pY)
{
     int i;
     long _Accum sum = 0.0lk;
     for (i = 0; i < 256; i++)
          sum += *pX++ * *pY++;

     return sum;
}
```

Слика А.1 Пример Це кода за скаларни производ два вектора дужине 256

Пре преласка на стандардни запис међурепрезентације приказан је излаз из посебног модула који за међурепрезнтацију генерише преводиви Це код истог значења. Превођењем тог Це кода компајлером за платформу домаћина добија се извршиви програм. На тај начин се испитни циклус (преведи – изврши - провери резултат) може обавити само над делом компајлера, то јест само над делом тока превођења, јер није потребно у потпуности добити асемблерски код за циљну платформу да би се код могао извршити. Пошто овај модул ради само за МР високог нивоа овај поступак може се употребити за проверу трансформамција на високом нивоу. Прављењем модула који ово ради и са инструкцијама ниског нивоа добио би се се компајлирани симулатор (што је за Cirrus Coyote 32 практично изведено и приказано у раду [Djukic1]). Код који поменути модул генерише за прво стање међурепрезентације у задњем делу компајлера дат је у наставку:

```
long__Accum _foo(_Fract* rs2, _Fract* rs4);

long__Accum _foo(_Fract* rs2, _Fract* rs4)
{
long__Accum rs1;
_Fract* rs3;
_Fract* rs5;
int rs6;
long__Accum rs7;
int rs8;
_Fract* rs9;
_Fract rs10;
long__Accum rs11;
_Fract* rs12;
_Fract rs13;
```

```
long__Accum rs14;
_Fract* rs15;
_Fract* rs16;
int rs17;
int rs18;
  _foo:
    rs3 = rs2;
    rs5 = rs4;
    rs7 = 0;
    rs6 = 0x0;
  for_0:
    rs8 = rs6 < 0x100;
    if (!(rs8)) goto for_end_0;
    rs9 = rs3;
    rs10 = *rs9;
    rs12 = rs5;
    rs13 = *rs12;
    rs11 = rs10 * rs13;
    rs14 = rs7 + rs11;
    rs7 = rs14;
    rs15 = rs3 + 0x1;
    rs3 = rs15;
    rs16 = rs5 + 0x1;
    rs5 = rs16;
  init_latch_label_0:
    rs17 = rs6;
    rs18 = rs6 + 0x1;
    rs6 = rs18;
    goto for_0;
  for_end_0:
    rs1 = rs7;
    goto __epilogue_0;
  __epilogue_0:
    return rs1;
}
```

Иако генерисани код садржи аритметику у непокретном зарезу, могао би се превести и на платформи домаћину која нема подршку за те типове увођењем емулационих класа и превођењем Це++ компајлером.

Први запис стања међурепрезентације у задњем делу компајлера приказан је на слици А.2 и представља међурепрезентацију после неколико једноставних трансформација у задњем делу. Ресурсни операнди су најбројнији у међурепрезентацији и карактерише их „rs” на почетку назива. У овом примеру још су видљиви и симболички операнди (лабеле) и литерали (константе). Могуће је укључити и испис типа сваког од операнда, као и ресурсе који су им додељени (што се деси тек касније у току превођења). Свака инструкција међурепрезентације (која представља једну операцију, за разлику од машинске инструкције која може садржати неколико операција) је у посебној линији. У коментару везаном за сваку инструкцију дате су информације о броју Це линије од које је инструкција потекла, циклусу у оквиру основног блока (попуњава распоређивач и у почетку је постављено на -1), као и скуп преписивачких правила која су довела до те инструкције. Види се да је за сада на неколико места примењено само правило са називом ***CpointerIntOperationRule*** које код сабирање интеџера и показивача уводи множење са величином типа на који показивач показује. У наредним записима, зарад једноставности, информације из коментара биће приказане само ако је потребно.

```
   .code_ovly
_foo:        /* LN: 2 | CYCLE: -1 | RULES: () */
  rs4584 = rs2        /* LN: 2 | CYCLE: -1 | RULES: () */
  rs4586 = rs4        /* LN: 2 | CYCLE: -1 | RULES: () */
  rs4590 = 0:0        /* LN: 5 | CYCLE: -1 | RULES: () */
  rs4588 = 0x0        /* LN: 6 | CYCLE: -1 | RULES: () */
for_0:        /* LN: 6 | CYCLE: -1 | RULES: () */
  rs4592 = rs4588 < 0x100        /* LN: 6 | CYCLE: -1 | RULES: () */
  if not (rs4592) jmp for_end_0        /* LN: 6 | CYCLE: -1 | RULES: () */
  rs4593 = rs4584        /* LN: 7 | CYCLE: -1 | RULES: () */
  rs4594 = mem[rs4593]        /* LN: 7 | CYCLE: -1 | RULES: () */
  rs4596 = rs4586        /* LN: 7 | CYCLE: -1 | RULES: () */
  rs4597 = mem[rs4596]        /* LN: 7 | CYCLE: -1 | RULES: () */
  rs4595 = rs4594 * rs4597        /* LN: 7 | CYCLE: -1 | RULES: () */
  rs4598 = rs4590 + rs4595        /* LN: 7 | CYCLE: -1 | RULES: () */
  rs4591 = rs4598        /* LN: 7 | CYCLE: -1 | RULES: () */
  rs4603 = 0x1        /* LN: 7 | CYCLE: -1 | RULES: (CPointerIntOperationRule) */
  rs4604 = rs4603 * 0x1        /* LN: 7 | CYCLE: -1 | RULES: (CPointerIntOperationRule) */
  rs4599 = rs4584 + rs4604        /* LN: 7 | CYCLE: -1 | RULES: (CPointerIntOperationRule) */
  rs4585 = rs4599        /* LN: 7 | CYCLE: -1 | RULES: () */
  rs4605 = 0x1        /* LN: 7 | CYCLE: -1 | RULES: (CPointerIntOperationRule) */
  rs4606 = rs4605 * 0x1        /* LN: 7 | CYCLE: -1 | RULES: (CPointerIntOperationRule) */
  rs4600 = rs4586 + rs4606        /* LN: 7 | CYCLE: -1 | RULES: (CPointerIntOperationRule) */
  rs4587 = rs4600        /* LN: 7 | CYCLE: -1 | RULES: () */
init_latch_label_0:        /* LN: 7 | CYCLE: -1 | RULES: () */
  rs4602 = rs4588 + 0x1        /* LN: 6 | CYCLE: -1 | RULES: () */
  rs4589 = rs4602        /* LN: 6 | CYCLE: -1 | RULES: () */
  rs4584 = rs4585        /* LN: 6 | CYCLE: -1 | RULES: () */
  rs4586 = rs4587        /* LN: 6 | CYCLE: -1 | RULES: () */
  rs4588 = rs4589        /* LN: 6 | CYCLE: -1 | RULES: () */
  rs4590 = rs4591        /* LN: 6 | CYCLE: -1 | RULES: () */
  jmp for_0        /* LN: 6 | CYCLE: -1 | RULES: () */
for_end_0:        /* LN: 6 | CYCLE: -1 | RULES: () */
  rs1 = rs4590        /* LN: 9 | CYCLE: -1 | RULES: () */
  jmp __epilogue_0        /* LN: 9 | CYCLE: -1 | RULES: () */
__epilogue_0:        /* LN: 10 | CYCLE: -1 | RULES: () */
  ret        /* LN: 10 | CYCLE: -1 | RULES: () */
```

Слика А.2 Први запис међурепрезентације

Наредни запис је направљен након пропагације константи и срачунавања константних израза. Резултат је поједностављивање кода који рачуна померај показивача. Запис је приказан на слици А.3 упоредо са претходним записом да се боље истакне промена која је настала.

```
   .code_ovly                          .code_ovly
_foo:                               _foo:
  rs4584 = rs2                        rs4584 = rs2
  rs4586 = rs4                        rs4586 = rs4
  rs4590 = 0:0                        rs4590 = 0:0
  rs4588 = 0x0                        rs4588 = 0x0
for_0:                              for_0:
  rs4592 = rs4588 < 0x100             rs4592 = rs4588 < 0x100
  if not (rs4592) jmp for_end_0       if not (rs4592) jmp for_end_0
  rs4593 = rs4584                     rs4593 = rs4584
  rs4594 = mem[rs4593]                rs4594 = mem[rs4593]
  rs4596 = rs4586                     rs4596 = rs4586
  rs4597 = mem[rs4596]                rs4597 = mem[rs4596]
  rs4595 = rs4594 * rs4597            rs4595 = rs4594 * rs4597
  rs4598 = rs4590 + rs4595            rs4598 = rs4590 + rs4595
  rs4591 = rs4598                     rs4591 = rs4598
  rs4603 = 0x1
  rs4604 = rs4603 * 0x1
  rs4599 = rs4584 + rs4604            rs4599 = rs4584 + 0x1
  rs4585 = rs4599                     rs4585 = rs4599
  rs4605 = 0x1
  rs4606 = rs4605 * 0x1
  rs4600 = rs4586 + rs4606            rs4600 = rs4586 + 0x1
  rs4587 = rs4600                     rs4587 = rs4600
init_latch_label_0:                 init_latch_label_0:
  rs4602 = rs4588 + 0x1               rs4602 = rs4588 + 0x1
  rs4589 = rs4602                     rs4589 = rs4602
  rs4584 = rs4585                     rs4584 = rs4585
  rs4586 = rs4587                     rs4586 = rs4587
  rs4588 = rs4589                     rs4588 = rs4589
  rs4590 = rs4591                     rs4590 = rs4591
  jmp for_0                           jmp for_0
for_end_0:                          for_end_0:
  rs1 = rs4590                        rs1 = rs4590
  jmp __epilogue_0                    jmp __epilogue_0
__epilogue_0:                       __epilogue_0:
  ret                                 ret
```

Слика А.3 Други запис међурепрезентације – после пропагације константи и срачунавања константних израза

На слици А.4 дат је запис након фазе уклањања заједничких подизраза и непотребних инструкција премештања (енг. move). Указано на једну трансформацију кода која се десила. Анализом кода, а полазећи од инструкција обележених бројевима 1, 2 и 3, закључује се да је **rs4585** једнак операнду **rs4599** (инструкција 1), **rs4584** једнак **rs4585** (инструкција 2), а **rs4593** једнак **rs4584** (инструкција 3); из чека следи да се и **rs4585** и **rs4584** и **rs4593** могу заменити

операндом **rs4599**. Иста трансформација се обавља и у вези са операндом **rs4600**. Након трансформације код је краћи и садржи мање операнада.

```
   .code_ovly                          .code_ovly
_foo:                               _foo:
   rs4584 = rs2                        rs4599 = rs2
   rs4586 = rs4                        rs4600 = rs4
   rs4590 = 0:0                        rs4598 = 0:0
   rs4588 = 0x0                        rs4602 = 0x0
for_0:                              for_0:
   rs4592 = rs4588 < 0x100             rs4592 = rs4602 < 0x100
   if not (rs4592) jmp for_end_0       if not (rs4592) jmp for_end_0
3  rs4593 = rs4584
   rs4594 = mem[rs4593]                rs4594 = mem[rs4599]
   rs4596 = rs4586
   rs4597 = mem[rs4596]                rs4597 = mem[rs4600]
   rs4595 = rs4594 * rs4597            rs4595 = rs4594 * rs4597
   rs4598 = rs4590 + rs4595            rs4598 = rs4598 + rs4595
   rs4591 = rs4598
   rs4599 = rs4584 + 0x1               rs4599 = rs4599 + 0x1
1  rs4585 = rs4599
   rs4600 = rs4586 + 0x1               rs4600 = rs4600 + 0x1
   rs4587 = rs4600
init_latch_label_0:                 init_latch_label_0:
   rs4602 = rs4588 + 0x1               rs4602 = rs4602 + 0x1
   rs4589 = rs4602
2  rs4584 = rs4585
   rs4586 = rs4587
   rs4588 = rs4589
   rs4590 = rs4591
   jmp for_0                           jmp for_0
for_end_0:                          for_end_0:
   rs1 = rs4590                        rs1 = rs4598
   jmp __epilogue_0                    jmp __epilogue_0
__epilogue_0:                       __epilogue_0:
   ret                                 ret
```

Слика А.4 Трећи запис међурепрезентације – после уклањања заједничких подизраза и непотребних инструкција премештања

Следећи запис међурепрезентације је настао након анализе петљи и проналажења физички подржаних петљи. На упоредном приказу са леве стране су подвучене инструкције које чине контролу петље и мењају њен бројач. На десној страни је уочљиво да је компајлер препознао да се петља извршава 256 пута и направио је физички подржану петљу. Ово је први тренутак у току превођења када се јавља мешавина инструкција високог нивоа и инструкција ниског нивоа, јер је инструкција за физички подржану петљу везана за циљну архитектуру. Осим увођења физички подржане петље, уклоњени су и неки сувишни скокови.

```
  .code_ovly
_foo:
  rs4599 = rs2
  rs4600 = rs4
  rs4598 = 0:0
  rs4602 = 0x0
for_0:
  rs4592 = rs4602 < 0x100
  if not (rs4592) jmp for_end_0
  rs4594 = mem[rs4599]
  rs4597 = mem[rs4600]
  rs4595 = rs4594 * rs4597
  rs4598 = rs4598 + rs4595
  rs4599 = rs4599 + 0x1
  rs4600 = rs4600 + 0x1
init_latch_label_0:
  rs4602 = rs4602 + 0x1
  jmp for_0
for_end_0:
  rs1 = rs4598
  jmp __epilogue_0
__epilogue_0:
  ret
```

```
  .code_ovly
_foo:
  rs4599 = rs2
  rs4600 = rs4
  rs4598 = 0:0


  do (0x100), label_end_92
label_begin_92:
  rs4594 = mem[rs4599]
  rs4597 = mem[rs4600]
  rs4595 = rs4594 * rs4597
  rs4598 = rs4598 + rs4595
  rs4599 = rs4599 + 0x1
  rs4600 = rs4600 + 0x1
label_end_92:


for_end_0:
  rs1 = rs4598


  ret
```

Слика А.5 Четврти запис међурепрезентације – после откривања физички подржаних петљи и уклањања непотребних скокова

Након откривања физички подржаних петљи почиње делимично спуштање кода, то јест избор инструкција. Примењују се групе правила која замењују инструкције високог нивоа инструкцијама ниског нивоа. На слици А.6, за разлику од досадашњих слика, није дат и претходни запис, али су у коментарима дате листе правила која су примењена. Правила примењена у овом кораку подвучена су (***CPointerIntOperationRule*** правило је примењено још пре првог записа међукода, види слику А.2). Код неких инструкција гледањем само у запис није одмах виљиво да је дошло до трансформације јер је синтакса асемблера за Coyote 32 таква да, рецимо, множење и сабирање се записују инфиксно и личе на природан запис тих операција који се користи за инструкције вишег нивоа. Међутим, начин на који су дати коментари са додатним информацијама говори о ком нивоу апстракције се ради: ако је у питању Це синтакса коментара (са **/*** и ***/**) онда је то инструкција вишег нивоа, а ако је линијски коментар који почиње знаком **#**, онда је то инструкција ниског нивоа.

```
  .code_ovly
_foo:    /* RULES: () */
  rs4599 = rs2    /* RULES: () */
  rs4600 = rs4    /* RULES: () */
  rs4598 = 0:0    /* RULES: () */
  do (0x100), label_end_92    # RULES: ()
label_begin_92:    /* RULES: () */
  rs4594 = mem[rs4599]    /* RULES: () */
  rs4597 = mem[rs4600]    /* RULES: () */
  rs4595 = rs4594 * rs4597    # RULES: (CLowerMulFracts)
  rs4598 = rs4598 + rs4595    # RULES: (CLowerAddSimilarNumericType)
  rs4599 += 1    # RULES: (CPointerIntOperationRule, CLowerPointerIncSpecial)
  rs4600 += 1    # RULES: (CPointerIntOperationRule, CLowerPointerIncSpecial)
label_end_92:    # RULES: ()
for_end_0:    /*  RULES: () */
  rs1 = rs4598    /* RULES: () */
  ret    # RULES: (CRetRule)
```

Слика А.6 Пети запис међурепрезентације – након примене неких преписивачких правила

Наредни запис, дат на слици А.7, илуструје примену преписивачког правила које радни над инструкцијама ниског нивоа и генерише инструкције ниског нивоа. У питању је ***CMacPlus1*** правило, које од одговарајућих инструкција сабирања и множења прави MAC инструкцију. До сада је, дакле, илустрована примена три врсте правила:

- Правила чије су улаз инструкције високог нивоа, а излаз исто високог нивоа. Користе се за разне машински независне трансформације. Пример је ***CPointerIntOperationRule*** са лике А.2.

- Правила чији су улаз инструкције високог нивоа, а излаз инструкције ниског нивоа. Обављају избор машинских инструкција, то јест спуштање међурепрезентације. Има их највише, а примери њихове примене дати су на сликама А.6 и А.8.

- Правила чији су улаз инструкције ниског нивоа, а излаз исто инструкције ниског нивоа. Служе за машински зависне оптимизације и разне оптимизације прозорчића. Пример је ***CMacPlus1***, илустрован на слици А.7.

```
  .code_ovly
_foo:    /* RULES: () */
  rs4599 = i0    /* RULES: () */
  rs4600 = i1    /* RULES: () */
  rs4598 = 0:0    /* RULES: () */
  do (0x100), label_end_92    # RULES: ()
label_begin_92:    /* RULES: () */
  rs4594 = mem[rs4599]    /* RULES: () */
  rs4597 = mem[rs4600]    /* RULES: () */
  rs4598 += rs4594 * rs4597    # RULES: (CLowerAddSimilarNumericType, CLowerMulFracts, CMacPlus1)
  rs4599 += 1    # RULES: (CPointerIntOperationRule, CLowerPointerIncSpecial)
  rs4600 += 1    # RULES: (CPointerIntOperationRule, CLowerPointerIncSpecial)
label_end_92:    # RULES: ()
for_end_0:    /* RULES: () */
  a0 = rs4598    /* RULES: () */
  ret    # RULES: (CRetRule)
```

Слика А.7 Шести запис међурепрезетације – након примене правила за комбиновање сабирања и множења у MAC инструкцију

Запис на слици А.8 настао је након примене остатка правила за спуштање кода. Сада су све инструкције ниског нивоа, осим лабела, које се не спуштају и увек се третирају као да су високог нивоа (што се види по облику коментара). Једна од промена које се могу уочити је везана за инструкције читања из меморије. Сада јасно стоји из које меморије се подаци читају, прва чита из **X** меморије, друга из **Y**. Карактеристична је још и инструкција ниског нивоа за започињање физички подржане петље, пошто за њу није везано правило од којег је настала. То је зато што њу уводи модул за откривање физички подржаних петљи, као што је описао у тексу уз слику А.5. Осим примене преосталих правила за спуштање кода, функција је још прилагођења позивној конвенцији. Сада уместо обичних ресурсних операнада за улазне аргументе и повратну вредност у запису видимо конкретне регистре i0 и i1, за аргументе, и a0, за повратну вредност, јер тако прописује позивна конвенција.

```
   .code_ovly
 _foo:    /* RULES: () */
   rs4599 = i0    # RULES: (CMovePointerToPointer)
   rs4600 = i1    # RULES: (CMovePointerToPointer)
   rs4598 = 0    # RULES: (CFpConstToLongAccum)
   do (0x100), label_end_92    # RULES: ()
 label_begin_92:    /* RULES: () */
   rs4594 = xmem[rs4599]    # RULES: (CLowerLoadFractFromXMemToData)
   rs4597 = ymem[rs4600]    # RULES: (CLowerLoadFractFromYMemToData)
   rs4598 += rs4594 * rs4597    # RULES: (CLowerAddSimilarNumericType, CLowerMulFracts, CMacPlus1)
   rs4599 += 1    # RULES: (CPointerIntOperationRule, CLowerPointerIncSpecial)
   rs4600 += 1    # RULES: (CPointerIntOperationRule, CLowerPointerIncSpecial)
 label_end_92:    # RULES: ()
 for_end_0:    /* RULES: () */
   a0 =+ rs4598    # RULES: (CMoveSameLongAccumAccSameLongAccumAcc)
   ret    # RULES: (CRetRule)
```

Слика А.8 Седми запис међурепрезентације – након потпуног спуштања кода и прилагођења позивној конвенцији

Следећи запис је настао након првог пролаза кроз распоређивач. Распоређивач је преуредио неке инструкције и предложио њихове циклусе у основним блоковима. Види се да су четири инструкције из основног блока који чини тело петље подигнуте изнад MAC инструкције и обележене да деле исти циклус. То значи да се распоређивач предлаже да се оне нађу у паралели, а то ће бити могуће ако буду испуњена ресурсна ограничења која је распоређивач емитовао.

```
  .code_ovly
_foo:
  rs4599 = i0
  rs4600 = i1
  rs4598 = 0
  do (0x100), label_end_92
label_begin_92:
  rs4594 = xmem[rs4599]
  rs4597 = ymem[rs4600]
  rs4598 += rs4594 * rs4597
  rs4599 += 1
  rs4600 += 1
label_end_92:
for_end_0:
  a0 =+ rs4598
  ret
```

```
  .code_ovly
_foo:    /* CYCLE: 0 */
  rs4599 = i0    # CYCLE: 0
  rs4600 = i1    # CYCLE: 1
  rs4598 = 0    # CYCLE: 2
  do (0x100), label_end_92    # CYCLE: 3
label_begin_92:    /* CYCLE: 0 */
  rs4594 = xmem[rs4599]    # CYCLE: 0
  rs4599 += 1    # CYCLE: 0
  rs4597 = ymem[rs4600]    # CYCLE: 0
  rs4600 += 1    # CYCLE: 0
  rs4598 += rs4594 * rs4597    # CYCLE: 1
label_end_92:    # CYCLE: 1
for_end_0:    /* CYCLE: 0 */
  a0 =+ rs4598    # CYCLE: 0
  ret    # CYCLE: 1
```

Слика А.9 Осми запис међурепрезентације - након предлагања редоследа инструкција и паралелизма од стране распоређивача

На наредној слици (А.10) дата су два записа, напоредо са претходним записом. Та два записа илуструју поступност доделе ресурса. Први следећи запис показује да су додељени ресурсни свим операндима који се тичу потенцијално паралелних операција. То је зато што додељивач ресурса прво покушава да задовољи необавезна ресурсна ограничења. Види се да операнду који није обухваћен неким од необавезних ограничења (већ се на њега односе само сметње, које су обавезна ограничења) у том првом кораку није додељен ресурс. Тек у другом кораку, чији резулат се види на последњем запису на слици А.10, ресурс ће бити додељен и том операнду. У том другом кораку алгоритам за доделу ресурса покушава да смањи број потребних инструкција премештања (eng. move), тако што ће покушати доделити исти ресурс и левом и десном операнду код што више операција премештања. Задатак је у овом примеру кода врло лак, тако да ће преосталом операнду бити додељен ресурс a0. Сада претпоследња инструкција функције може бити уклоњена, што се види на слици А.11 поређењем два записа између којих је позван модул за уклањање инструкција премештања.

```
  .code_ovly
_foo:
  rs4599 = i0
  rs4600 = i1
  rs4598 = 0
  do (0x100), label_end_92
label_begin_92:
  rs4594 = xmem[rs4599]
  rs4599 += 1
  rs4597 = ymem[rs4600]
  rs4600 += 1
  rs4598 += rs4594 * rs4597
label_end_92:
for_end_0:
  a0 =+ rs4598
  ret
```

```
  .code_ovly
_foo:
  i0 = i0
  i4 = i1
  rs4598 = 0
  do (0x100), label_end_92
label_begin_92:
  x0 = xmem[i0]
  i0 += 1
  y0 = ymem[i4]
  i4 += 1
  rs4598 += x0 * y0
label_end_92:
for_end_0:
  a0 =+ rs4598
  ret
```

```
  .code_ovly
_foo:
  i0 = i0
  i4 = i1
  a0 = 0
  do (0x100), label_end_92
label_begin_92:
  x0 = xmem[i0]
  i0 += 1
  y0 = ymem[i4]
  i4 += 1
  a0 += x0 * y0
label_end_92:
for_end_0:
  a0 =+ a0
  ret
```

Слика А.10 Девети и десети запис међурепрезентације – током и након процеса доделе ресурса

```
  .code_ovly
_foo:
  i0 = i0
  i4 = i1
  a0 = 0
  do (0x100), label_end_92
label_begin_92:
  x0 = xmem[i0]
  i0 += 1
  y0 = ymem[i4]
  i4 += 1
  a0 += x0 * y0
label_end_92:
for_end_0:
  a0 =+ a0
  ret
```

```
  .code_ovly
_foo:

  i4 = i1
  a0 = 0
  do (0x100), label_end_92
label_begin_92:
  x0 = xmem[i0]
  i0 += 1
  y0 = ymem[i4]
  i4 += 1
  a0 += x0 * y0
label_end_92:
for_end_0:

  ret
```

Слика А.11 Једанаести запис међурепрезентације – након позива модула за улањање инструкција премештања код којих је исти ресурс са обе стране

Напослетку, на слици А.12, дат је асемблерски код који је излаз из компајлера и упоређен је са последњим записом међурепрезентације пред емитовање кода. Уочљиве су две разлике. Прва је изглед паралелних операција Оне сада чине једну машинску инструкцију и емитоване су на начин који очекује асемблер циљне платформе. Друга разлика је позиција лабеле ***label_end_92***. До сада је она представљала граничник тела физички подржане петље, али у ствари физичка петља ради тако што укључује у тело и инструкцијау на адреси која је прослеђена као операнд. До самог краја ова лабела је држана иза последње иструкције, јер лабела у међурепрезентацији условљава прављење новог основног блока у графу тока управљања. На овај начин током превођења не

формира се нови непотребни основни блок и смисао лабеле је другачији, и тек при емитовању се лабела исписује на одговарајућем месту.

```
   .code_ovly                  .code_ovly
 _foo:                       _foo:
   i4 = i1                     i4 = i1
   a0 = 0                      a0 = 0
   do (0x100), label_end_92    do (0x100), label_end_92
 label_begin_92:             label_begin_92:
   x0 = xmem[i0]               x0 = xmem[i0]; i0 += 1; y0 = ymem[i4]; i4 += 1
   i0 += 1
   y0 = ymem[i4]
   i4 += 1                   label_end_92:
   a0 += x0 * y0               a0 += x0 * y0
 label_end_92:
 for_end_0:                  for_end_0:
   ret                         ret
```

Слика А.12 Излазни асемблерски код упоређен са једанаестим записом међурепрезентације

```
   .code_ovly
 _foo:
   i4 = i1
   x0 = xmem[i0]; i0 += 1; y0 = ymem[i4]; i4 += 1; a0 = 0
   do (0x100), label_end_92
 label_begin_92:
 label_end_92:
   x0 = xmem[i0]; i0 += 1; y0 = ymem[i4]; i4 += 1; a0 += x0 * y0
 for_end_0:
   ret
```

Слика А.13 Оптимални асемблерски код за дату функцију

Из излазног асемблерског кода се види да је, на овом једноставном примеру, компајлер генерисао врло квалитетан код искористивши паралелизам на нивоу инструкције, физички подржане петље, адресне генераторе и носећи се добро са неортогоналним скупом инструкција. Међутим, код може бити још бољи и за то би требало у компајлер додати модул за љуштење петље (енг. loop peeling). У случају постојања тог модула излазни код би могао изгледати као на слици А.13.