



УНИВЕРЗИТЕТ У НОВОМ САДУ
ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА У
НОВОМ САДУ



Душан Стојковић

**Један приступ за реверзибилни инжињеринг
перформанси RISC-V микро-архитектура**

МАСТЕР РАД

Нови Сад, 2026.



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 :	Дипломски – мастер рад
Аутор, АУ :	Душан Стојковић
Ментор, МН :	Проф. др. Мирослав Поповић
Наслов рада, НР :	Један приступ за реверзибилни инжињеринг перформанси RISC-V микро-архитектура
Језик публикације, ЈП :	Српски / латиница
Језик извода, ЈИ :	Српски
Земља публиковања, ЗП :	Република Србија
Уже географско подручје, УГП :	Војводина
Година, ГО :	2026.
Издавач, ИЗ :	Ауторски репринт
Место и адреса, МА :	Нови Сад; Трг Доситеја Обрадовића 6
Физички опис рада, ФО : (поглавља/страница/ цитата/табела/слика/графика/прилога)	7/99/37/33/10/0/0
Научна област, НО :	Електротехника и рачунарство
Научна дисциплина, НД :	Рачунарска техника
Предметна одредница/Кључне речи, ПО :	RISC-V, профилисање, софтверски алати
УДК	
Чува се, ЧУ :	У библиотеци Факултета техничких наука, Нови Сад
Важна напомена, ВН :	
Извод, ИЗ :	Овај рад представља један начин да се мере перформансе RISC-V микро-архитектура. Циљ је експериментално пронаћи ваљане вредности за цене инструкција како би се оптимизације софтвера вршиле на микро-архитектуралном нивоу. Развијено је окружење за профилисање које примењује методологију која минимизује грешку у мерењима. Тестови су генерисани хијерархијом претпроцесорских макроа. Покретање и филтрирање тестова је урађено коришћењем googletest окружења. Евалуација решења је спроведена на процесору SracemIT-X60 и резултати су поређени са јавно доступним вредностима цена инструкција по метрикама кашњења и проточности.
Датум прихватања теме, ДП :	
Датум одбране, ДО :	
Чланови комисије, КО :	Председник: ванр. проф. др Јелена Радић
	Члан: доц. др Миодраг Ђукић
	Члан, ментор: ред. проф. др Мирослав Поповић
	Потпис ментора



KEY WORDS DOCUMENTATION

Accession number, ANO :	
Identification number, INO :	
Document type, DT :	Monographic publication
Type of record, TR :	Textual printed material
Contents code, CC :	Master Thesis
Author, AU :	Dusan Stojkovic
Mentor, MN :	PhD Miroslav Popovic
Title, TI :	One approach to reverse-engineering of the performance of RISC-V micro-architectures
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 :	2026.
Publisher, PB :	Author's reprint
Publication place, PP :	Novi Sad, Trg Dositeja Obradovica 6
Physical description, PD : (chapters/pages/ref./tables/pictures/graphs/appendixes)	7/99/37/33/10/0/0
Scientific field, SF :	Electrical Engineering
Scientific discipline, SD :	Computer Engineering, Engineering of Computer Based Systems
Subject/Key words, S/KW :	RISC-V, profiling, software tools
UC	
Holding data, HD :	The Library of Faculty of Technical Sciences, Novi Sad, Serbia
Note, N :	
Abstract, AB :	This paper presents one way to measure the performamnce of RISC-V micro-architectures. The goal is to find proper values of instruction costs experimentally so that optimizations of software can be done on the micro-arhictectural level. An environment is presented for profiling which applies a metodology which minimizes measurement error. Tests are generated using a hierarchy of nested pretprocessor macros. Running and filtering of tests is achieved using googletest framework. Evaluation of the solution is conducted on the SpacemiT-X60 processor and the results are compared against publically avaiable data on latency and throughput metrics.
Accepted by the Scientific Board on, ASB :	
Defended on, DE :	
Defended Board, DB :	President: PhD Jelena Radic
	Member: PhD Miodrag Djukic
	Member, Mentor: PhD Miroslav Popovic
	Menthor's sign



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

Број:

ЗАДАТАК ЗА ЗАВРШНИ РАД

Датум:

(Погајке уноси предметни наставник - ментор)

Студијски програм:	Рачунарска техника и рачунарске комуникације – Софтвер за потрошачку електронику		
Студент:	Душан Стојковић	Број	E54/2023
Степен и врста студија:	Мастер академске студије		
Област:	Електротехника и рачунарство		
Ментор:	Проф. др Мирослав Поповић		
НА ОСНОВУ ПОДНЕТЕ ПРИЈАВЕ, ПРИЛОЖЕНЕ ДОКУМЕНТАЦИЈЕ И ОДРЕДБИ СТАТУТА ФАКУЛТЕТА ИЗДАЈЕ СЕ ЗАДАТАК ЗА ЗАВРШНИ РАД, СА СЛЕДЕЋИМ ЕЛЕМЕНТИМА:			
- проблем – тема рада; - начин решавања проблема и начин практичне провере резултата рада, ако је таква провера неопходна;			

НАСЛОВ ЗАВРШНОГ РАДА:

Један приступ за реверзибилни инжињеринг перформанси RISC-V микро-архитектура

ТЕКСТ ЗАДАТКА:

У оквиру овог дипломског рада потребно упознати се са микро-архитектуром RISC-V SpacemiT-X60. Затим анализирати недостатке претходних метода профилисања микро-архитектура на RISC-V SpacemiT-X60 процесору. На основу анализе неопходно је израдити пројекат унапређеног алата микро-архитектуралних тестова. По узору на израђен пројекат имплементирати алат. Верификовати алат тако што се резултати мерења упореде са резултатима постојећих решења где год је то могуће. Дискутовати резултате евалуације и уочити предности и мане алата и напослетку написати текст дипломског рада.

Руководилац студијског програма:	Ментор рада:

Примерак за: - Студента; - Ментора



УНИВЕРЗИТЕТ У НОВОМ САДУ • ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА

21000 НОВИ САД, Трг Доситеја Обрадовића 6

ИЗЈАВА О НЕПОСТОЈАЊУ СУКОБА ИНТЕРЕСА

Образац Q2.НА.04-03 - Издање 1

Изјављујем да нисам у сукобу интереса у односу ментор – кандидат и да нисам члан породице (супружник или ванбрачни партнер, родитељ или усвојитељ, дете или усвојеник), повезано лице (крвни сродник ментора/кандидата у правој линији, односно у побочној линији закључно са другим степеном сродства, као ни физичко лице које се према другим основама и околностима може оправдано сматрати интересно повезаним са ментором или кандидатом), односно да нисам зависан/на од ментора/кандидата, да не постоје околности које би могле да утичу на моју непристрасност, нити да стичем било какве користи или погодности за себе или друго лице било позитивним или негативним исходом, као и да немам приватни интерес који утиче, може да утиче или изгледа као да утиче на однос ментор-кандидат.

У Новом Саду, дана _____

Ментор

Кандидат

Захвалност

Пре свега желим да се захвалим својој породици на њиховој непрекидној подршци кроз читаво моје образовање. Такође, желим да се захвалим проф. Поповићу за његову посвећеност, стрпљење и све савете које су ми биле звезде водиле кроз израду овог рада. Напоселтку, желим да се захвалим свим колегама из РТ-РК који верују у отвореност изворног кода и који су сарађивали са мном на побољшању RISC-V софтверског екосистема.

САДРЖАЈ

1. Увод.....	1
2. Теоријске основе и сродни радови.....	3
2.1.1 Рани дигитални рачунари.....	3
2.1.2 Прва интегрисана кола и појава сложених рачунарских архитектура.....	4
2.1.3 Појава персоналних рачунара и прве RISC архитектуре.....	5
2.2 RISC-V.....	7
2.2.1 Преглед главних карактеристика RISC-V.....	9
2.2.2 Архитектуралне карактеристике скаларних RISC-V микро-архитектура....	10
2.2.3 Архитектуралне карактеристике векторских RISC-V копроцесора.....	11
2.2.4 Архитектура SparcemiT-X60 RISC-V процесора базирана на његовим јавно доступним карактеристикама.....	12
2.2.5 Методологија мерења перформанси.....	15
2.2.6 Метрике прикупљане perf Linux алатом.....	16
2.2.7 Избор метрика корелације резултата.....	17
2.2.7.1 Корелација базирана на Кендаловом коефицијенту корелације рангова.....	18
2.2.7.2 Средња апсолутна процентуална грешка (MAPE).....	19
2.2.7.3 Геометријска средина односа (GMR).....	19
2.3 Сродни радови.....	20
2.3.1 Један начин за профилисање Linux perf алатом аутовекторизације програмског преводиоца на RISC-V [11].....	20
2.3.2 Решење за профилисање RISC-V инструкција rvv-bench [12].....	21

2.3.3 Практично PMU профилисање и анализа крова на RISC-V платформама независна од хардвера [13].....	21
2.3.4 uops.info: категоризација кашњења, проточности и заузетости портова на Intel x86 процесорима [14].....	22
3. Концепт решења	23
3.1 Окружење за извршавање микро-архитектуралних тестова.....	24
3.2 Генератор микро-архитектуралних тестова.....	25
3.2.1 Приступне макро функције.....	25
3.2.2 Генерисање C++ функција путем макро претпроцесора за микро-архитектуралне тестове.....	27
3.2.3 Експанзија сродних макроа по типу теста.....	29
3.3 Омотач око Linux perf алата – PerfWrapper.....	30
3.4 Скрипта за извршавање алата за мерење микро-архитектура.....	31
3.5 Организација stake листе за превођење.....	32
3.6 Логички ток алата за мерење микро-архитектура.....	32
3.6.1 Језгро извршног окружења.....	35
4. Програмско решење	39
4.1 Реализација окружења за извршавање тестова.....	39
4.1.1 Датотека run_pipeline_benchmarks.cpp.....	40
4.1.1.1 Класа PipelineBenchmarkGapPerf.....	41
4.1.1.2 Класа PipelineBenchmarkLatencyPerf.....	41
4.1.1.3 Класа PipelineBenchmarkRooflinePerf.....	42
4.1.1.4 Класа PipelineBenchmarkThroughputPerf.....	42
4.1.1.5 Именски простор anonymous_namespace{run_pipeline_benchmarks.cpp}	42
4.1.2 Датотека util.cpp.....	43
4.1.3 Датотека util.h.....	43
4.1.3.1 Структура BenchmarkConfig.....	45
4.2 Реализација претпроцесорског генератора microbenchmark_generator.h.....	45
4.2.1 Структура benchmark_core.....	45
4.2.2 Структура instruction_benchmark.....	46
4.2.3 Макрои генератора.....	46
4.3 Реализација омотача око Linux perf алата.....	67
4.3.1 Датотека perf_config.h.....	67
4.3.1.1 Структура perf_config.....	67

4.3.2 Датотека perf_collector.h.....	67
4.3.2.1 Класа PerfCollector.....	67
4.3.3 Датотека perf_group.h.....	68
4.3.3.1 Структура perf_hw_event.....	68
4.3.3.2 Класа PerfGroup.....	69
4.3.3.3 Класа PerfWrapper::anonymous_namespace{perf_group.h}::PerfCounter..	70
4.3.3.4 Структура PerfWrapper::group_read_data.....	71
4.3.3.5 Структура PerfWrapper::read_value.....	71
4.3.4 Именски простор PerfWrapper.....	72
4.4 Реализација рачунице сложених метрика у заглављу metric_calculations.h.....	73
4.4.1 Структура derived_metric.....	74
4.4.2 Именски простор MetricCalculations.....	75
4.5 Реализација скрипте за извршавање алата.....	76
5. Евалуација	78
5.1 Дискусија резултата.....	85
5.1.1 Дискусија векторских резултата.....	90
5.1.2 Дискусија резултата скаларног кашњења и проточности.....	91
6. Закључак.....	93
7. Литература.....	96

СПИСАК СЛИКА

Слика 1 - Наследство RISC-V, прецртана на основу оригиналног истраживања.....	9
Слика 2: Преглед SracemIT-X60 језгра.....	14
Слика 3: Преглед одозго арахитектуре решења.....	23
Слика 4: Граф језгра претроцесорског макро генератора тестова.....	25
Слика 5: Архитектура CMakeLists.txt pipeline_benchmarks алата.....	32
Слика 6: Блок дијаграм тока превођења алата за мерење микро-архитектура.....	34
Слика 7: Блок дијаграм тока извршавања алата за мерење микро-архитектура.....	36
Слика 8: Овај рад у поређењу са gvv-bench за сваку векторску инструкцију.....	88
Слика 9: Слика 9: Геометријска средина односа овај рад / gvv-bench по вредности LMUL, по групама инструкција и укупно.....	88
Слика 10: Дијаграмим упарених тачака LLVM модела и измерених вредности за сваку класу скаларних инструкција; лево: кашњење, десно: проточност.....	89

СПИСАК ТАБЕЛА

Табела 1: Карактеристике SparcemiT-X60 процесора.....	14
Табела 2: Пример генераторског LATENCY_BENCH теста.....	26
Табела 3: Пример генераторског THROUGHPUT_BENCH теста.....	26
Табела 4: Пример макро функције која генерише C++ функцију.....	27
Табела 5: Пример макро функције која генерише тело C++ функције.....	28
Табела 6: Основни макрои спајања макроа.....	29
Табела 7: Пример макро дефиниције асемблерске инструкције.....	29
Табела 8: Алгоритам естимирања J.....	37
Табела 9: Алгоритам извршавања једног теста.....	37
Табела 10: Садржај датотеке run_pipeline_benchmarks.cpp.....	40
Табела 11: Садржај именског простора anonymouse_namespace{run_pipeline_benchmarks.cpp}.....	42
Табела 12: Садржај датотеке util.cpp.....	43
Табела 13: Садржај датотеке util.hpp.....	43
Табела 14: Садржај структуре BenchmarkConfig.....	45
Табела 15: Садржај структуре benchmark_core.....	45
Табела 16: Садржај структуре instruction_benchmark.....	46
Табела 17: Сви макрои генератора.....	46
Табела 18: Садржај структуре perf_config.....	67
Табела 19: Садржај класе PerfCollector.....	67
Табела 20: Садржај датотеке perf_group.h.....	68
Табела 21: Садржај структуре perf_hw_event.....	68
Табела 22: Садржај класе PerfGroup.....	69

Табела 23: Садржај класе PerfCounter.....	70
Табела 24: Садржај структуре PerfWrapper::group_read_data.....	71
Табела 25: Садржај структуре PerfWrapper::read_value.....	71
Табела 26: Садржај именског простора PerfWrapper.....	72
Табела 27: Садржај структуре derived_metric.....	74
Табела 28: Садржај именског простора MetricCalculations.....	75
Табела 29: Сва поља скрипте run.sh.....	76
Табела 30: Све функције скрипте run.sh.....	77
Табела 31: Поређење цена векторских инструкција rvv-bench наспрам pipeline_benchmarks	80
Табела 32: Поређење цена скаларних инструкција LLVM распоређивача наспрам pipeline_benchmarks.....	83
Табела 33: Резултат по групи инструкција t-b, MAPE, GMR, метрика евалуације.....	85

СКРАЋЕНИЦЕ

CPU	- Central Processing Unit – Централна процесорска јединица
ISA	- Instruction Set Architecture – Архитектура скупа инструкција
RISC	- Reduced Instruction Set Computing, Рачунар са смањеним скупом инструкција
RISC-V	- Reduced Instruction Set Computing - Version Five, Рачунар са смањеним скупом инструкција – верзија пет
SIMD	- Single Instruction, Multiple Data – Једна инструкција, више података обрађених
VLIW	- Very Long Instruction Word – Инструкција врло дуге речи
PMU	- Performance Metric Unit – Јединица за мерење перформанси процесора
IPC	- Instructions Per Cycle – Инструкције по циклусу
API	- Application Program Interface – Апликацијски програмски интерфејс
SOC	- System on Chip – Систем на чипу
ASLR	- Address Space Layout Randomization – Насумично распоређивање адресног простора
CPI	Cycles per Instruction – Циклуси по инструкцији
MAPE	Mean Absolute Percentage Error - Средња апсолутна процентуална грешка
GMR	Geometric Mean Radius – Геометријска средина односа

SEW	Set Element Width – Постављена дужина елемента
LMUL	...Vector Register Group Multiplier – Множител векторских група

1. Увод

Рачунарске архитектуре су кроз историју развоја дигиталних рачунарских система евоулирале значајно. Почевши са вакумским цевима до прве употребе транзистора и интегрисаних кола па све до првих персоналних рачунара, рачунарске системи су значајно унапредовали.

Временом се и ISA (енгл. Instruction Set Architecture) – скуп *инструкција које рачунарска архитектура подржава* трансформисао како би рачунар могао да обави све комплексније захтеве разних програма и апликација. Један од кључних момената је појава RISC – *архитектура са редукованим скупом инструкција* рачунара који су омогућили мобилне рачунаре какве знамо данас.

RISC-V – *рачунарска архитектура са редукованим скупом инструкција верзије њеј* се појавила 2010 године. Постала је прва модерна потпуно отвореног кода RISC (енгл. Reduced Instruction Set Computer) архитектура која је прерасла академске кругове и била прихваћена од стране индустрије. Језгро RISC-V ISA спецификације је мало и релативно једноставно да нови пројекти процесора испоштују али га је могуће проширити са великим бројем ратификованих проширења које га праве равноправном конкуренцијом другим модерним ISA (нпр. ARM или x86) по цени али и перформансама.

Данас је увелико на снази Муров закон, достигли смо границе Денардовог скалирања као и исплативост паралелизације Амдаловог закона. Једино што преостаје је вешто оптимизовање софтвера за дубоко наменски хардвер како би брзина извршавања програма могла да расте [6].

Овај рад се конкретно бави мерењима кашњења и проточности инструкција RISC-V процесора у Linux радном окружењу. Демонстрирано је како, у условима са великом количином кашњења током рада оперативног система коришћењем хардверских бројача инструкција, је могуће измерити врло прецизно време извршавања сваке инструкције у

разним сценаријима употребом ваљане методологије обраде података. Приказан је један начин за генерисање као и извршавање великог броја тестова. Овакви подаци су незаобилазни у свету програмских преводиоца који распоређују инструкције специјализовано за циљану микро-архитектуру ако је доступна.

Рад се састоји из седам поглавља.

Друго поглавље поставља теоријске темеље који су неопходни за разумевање остатка рада.

У трећем поглављу је представљен концепт решења окружења за евалуацију RISC-V микро-архитектура.

Четврто поглавље је детаљна документација програмског решења. Ту се наводе програмске структуре имплементиране у алату за процену перформанси RISC-V микро-архитектура.

У петом поглављу је садржана евалуација програмског решења, показане су његове могућности као и недостаци и обављена је функционална верификација.

Напоследку је дат списак литературе који овај рад цитира.

2. Теоријске основе и сродни радови

Еволуција рачунарских архитектура је прошла кроз многе итерације, при чему сваку од њих карактеришу специфична побољшања и иновације. Пут од раних рачунара до модерних персоналних рачунара је поплочан значајним открићима који су обликовали данашње рачунарске архитектуре.

У овом поглављу је изложена историја рачунарских архитектура све до појаве RISC-V и затим су објашњене главне особине RISC-V архитектуре за овај рад.

2.1.1 Рани дигитални рачунари

Прича о модерним процесорима почиње још давно пре него што су се појавили први Intel 4004 процесори који су базирани на транзисторима (1971). Пре него што је идеја централне процесорске јединице која је програмибилна произвољним програмима у машинском језику изведена употребом транзистора први процесори су били прављени коришћењем механичких релеја који служе као прекидачи.

Зачетници процесора су једноставна логичка кола која раде основне математичке операције: сабирање, одузимање, можење и дељења бинарних бројева у Буловој алгебри. Први прототип струјног кола које извршава операцију сабирања био је Model K који је радио тако што је користио два механичка релеја и две сијалице. Наредни модели су били први рачунари који су радили као калкулатори са првим моделом у серији који је имао око 460 релеја [1].

Друга идеја која је променила парадигму у развоју процесора је прелазак са механичких на дигиталне релеје. Кључна технологија која је допринела овом прелазу је технологија вакуумских цеви. Специфично, вакуумске електронске триоде које су, за

разлику од механичких релеја, проводиле струју емитовањем електрона од катоде до аноде. Елиминисањем механичких делова је прелазак из затвореног у отворено коло и обрнуто убрзан са реда величине у милисекундама на микросекунде [2]. Управо је употребом 18000 вакумских цеви и релеја 1500 направљен први програмабилан рачунар ENIAC (1946) са радним тактом од 5 kHz [3].

Иако модерни рачунари нису базирани на вакумским електронским триодама рачунари развијани на овај начин су поставили основе развоја дигиталних логичких кола. Са појавом полупроводничких транзистора (Белове лабораторије 1947) настају нови рачунари који у значајно мање простора садрже више логичких кола него претходне технологије.

2.1.2 Прва интегрисана кола и појава сложених рачунарских архитектура

Временом је софистицираност производње полупроводника унапредовала. Открићем планарног процеса 1959. године (Роберт Нојс, Robert Nojs у компанији Fairchild Semiconductor), постало је могуће израђивати транзисторе директно у силицијумским плочицама (вејферима, енгл. wafer), На силицијумским плочицама израђује се велики број интегрисаних кола, која се након процеса производње издвајају, повезују са спољним контактима и пакују у заштитно кућиште. Логичка кола која проистичу из датог процеса су програмабилни процесори.

Као што је процес производње процесора напредовао тако је и његова намена. У почетку је рачунар био коришћен у војне и академске сврхе. Временом је рачунар постепено почео да се користи у све ширем спектру области. Зато је филозофија око пројектовања процесора прерасла почетне оквире и адаптирала се на:

1. Обраду података за бизнисе,
2. Раст важности гледања система у целовитости, посебно улазно излазне аспекте,
3. Универзалну употребу асемблерских програма, преводиоца и других мета-програма,
4. Развој магнетних трака за снимање података,
5. Повећање стоструко способности складиштења података,
6. Адаптацију у системе базиране на реалном времену.

Током овог периода (1960-тих) већина нових рачунарских модела, из перспективе логичке структуре, представљала је унапређене, проширене или редизајниране верзије рачунара из 1950-их година, реализоване применом нових технолошких поступака.

Већ од 50-тих година прошлог века у IBM-у се почео користити термин рачунарске архитектуре. У раду „*Arhitecture of the IBM System/360*“ се формално уводи концепт по

дефиницији као опис атрибута система онако видљивих од стране програмера, то јест, концептуална структура и функционално понашање, различито од организације тока података и контрола, логичког пројектовања и физичке имплементације [4].

Данас је извесно да је прикладнији термин ISA, јер скуп инструкција које рачунарска архитектура подржава је формални начин да се опишу могућности процесора: величина и формат инструкција, регистри доступни програмеру, операнди и типови података подржани као и меморијски модел. Ефективно, ISA би се могла гледати као линија контакта између софтверског и хардверског инжињера.

Такође је битно напоменути да су у овом периоду већ увелико заступљене породице рачунарских архитектура, наиме, рачунара који фундаментално деле исту архитектуру али постоје разлике у имплементацији које се огледају у: проточној структури, паралелизму на нивоу инструкција, редоследу извршавања инструкција, предикцији гранања и слично. Овакве разлике се обично називају микро-архитектуралне разлике и постаће релевантне у даљем тексту рада.

2.1.3 Појава персоналних рачунара и прве RISC архитектуре

Већ 70-тих и почетком 80-тих година прошлог века су почели да се појављују први персонални рачунари. Сасвим нова тржишта су се отворила за рачунаре тога времена, тржишта која су захтевала другачије особине од рачунара. Са растом тржишта на којима се користе рачунари је порастао и број фирми које производе рачунаре.

Већински су се у овом периоду изазови задовољавања потреба нових тржишта превазилазили проширивањем ISA, на пример, са потребом да се на неки нови начин обраде подаци, уводиле су се нове специјализоване рачунарске инструкције. Временом су се почели појављивати проблеми у овом приступу. Велика прекретница у даљем развоју рачунарских архитектура је моменат препознавања да се са оваквим поступком стварају све сложеније рачунарске архитектуре – CISC (енгл. Complex Instruction Set Computer).

У наставку се кратко наводе неки од разлога зашто се пројектују све сложеније рачунарске архитектуре.

Брзина меморије наспрам брзине процесора – процесори су генерално значајно бржи у обради него у приступу радној меморији. Испоставило се да се потпрограми процесора које извршавају више инструкција у блоковима који могу да послуже више пута (често називане псеудо-инструкције) понашају спорије него уграђене, недељиве инструкције. Тако је био случај да су се увођењем инструкција са покретним зарезом у скуп инструкција рачунарске архитектуре перформансе рачунара драстично побољшале.

Технолошке иновације у пројектовању процесора су резултирале у исплативом премештању потпрограма у саму архитектуру процесора.

Величина програма – рани рачунари су оскудевали у меморији. Стога је у многим случајевима, поготово у онима где се нека операција користи учестано, било исплативије да се уведе нека нова инструкција у пројекту процесора него да се кроз комбинацију две или више инструкције нека сложена операција изврши. Ово је резултовало мањим програмима, што је суштински омогућавало да се са мање меморије уради више обраде.

Маркетиншка стратегија – лако је представити да имати нечега више је нужно и боље. Тако су и произвођачи тога времена пласирали своје рачунаре на тржиште говорећи да су њихови рачунари супериорнији јер подржавају тај или онај начин обраде директно у хардверу. У том духу се и иновација пласирала као додавање нових могућности обраде. Нове сложеније архитектуре имају ту тенденцију да само додају нове инструкције у свој скуп, ретко су се старе инструкције изостављале у новијим генерацијама архитектура.

Пораст сложености програмских захтева – како би се пружила ефективнија програмска подршка програмским језицима на вишем нивоу апстракције, аргументовало се да ће сложенија рачунарска архитектура бити потпора новим захтевима сложенијег софтвера.

У капиталном научном раду, *Patterson, D. A.; Ditzel, D. R. (1980). The Case for the Reduced Instruction Set Computer* [5], аутори препознају ове аргументе произвођача рачунара и у сваком виде недостатке. Као решење препоручују прелазак на развој рачунара са редукованим скупом инструкција; RISC архитектуре.

У наставку се дају контрааргументи за горенаведене аргументе за CISC архитектуре.

Брзина меморије наспрам брзине процесора – са развојем полупроводничке меморије се и брзина меморије значајно побољшала. Такође, опала је и цена меморији. Читав аргумент додавања нових инструкција у процесор постаје дискутабилан када се оне ретко користе. Показало се да и програмски преводиоци ретко искоришћавају пун скуп инструкција које ISA пружа.

Величина програма – на основу доказа које су *Patterson, D. A.* и *Ditzel, D. R.* приложили испоставља се да се паметним бирањем инструкција у спецификацији ISA комбинованим са проширивањем радне меморије доступне процесору већ тада су се могли већина проблема везаних за величину програма у меморији превазићи и решити.

Пораст сложености програмских захтева – Испоставља се да, у случају оперативних система са дељењем времена извршавања на процесору, за сложеније рачунарске архитектуре је врло често теже сачувати стање неке нити у регистар сенке и прекључати другу. Овај проблем се погоршава ако постоје компликовани режими адресирања.

Ирационалне имплементације – сложене архитектуре су плодно тло за ирационалне одлуке у реализацији ISA. Дешавало се да сложене инструкције у одређеним сценаријима буду спорије него низ еквивалентних елементарних инструкција. Примера ради, на IBM 370, научници су пронашли да је низ од до четири инструкције учитавања јефтинији него псеудо-инструкција која учитава вредности из меморије у до четири регистра.

Тежина пројектовања CISC архитектура – сложеност у пројектовању CISC архитектура су инхерентне у имену; сложени системи захтевају више времена да се развију и валидирају. Додатно, настајање и отклањање грешака у сложеним системима није једноставан подухват.

Ово су само неки од аргумената које су навели *Patterson, D. A.* и *Ditzel*. Иако су технике развоја хардвера напредовале и CISC архитектуре опстале, аргументи и данас стоје. Данас RISC процесора има све више око нас, и тај тренд не делује као да ће стати у скорије време.

2.2 RISC-V

Након објаве првог рада који спомиње RISC архитектуре осамдесетих, започиње велика дебата у свету рачунарских архитектура која ће трајати наредне три деценије. Ипак, истраживачи Универзитета у Калифорнији, Берклија и Универзитета у Станфорду су започели радове на првим прототиповима процесора који ће обликовати све будуће рачунарске архитектуре. Њихови прототипи, први сасвим RISC типа, били су по многим карактеристикама бржи од тадашњих Intel, IBM и других комерцијално доступних процесора [6].

RISC-V рачунарска архитектура је развијена од стране научника са Универзитета у Калифорнији, Берклија. На основу самог имена очевидно је да је у питању пети у низу пројеката развоја процесора RISC архитектуре.

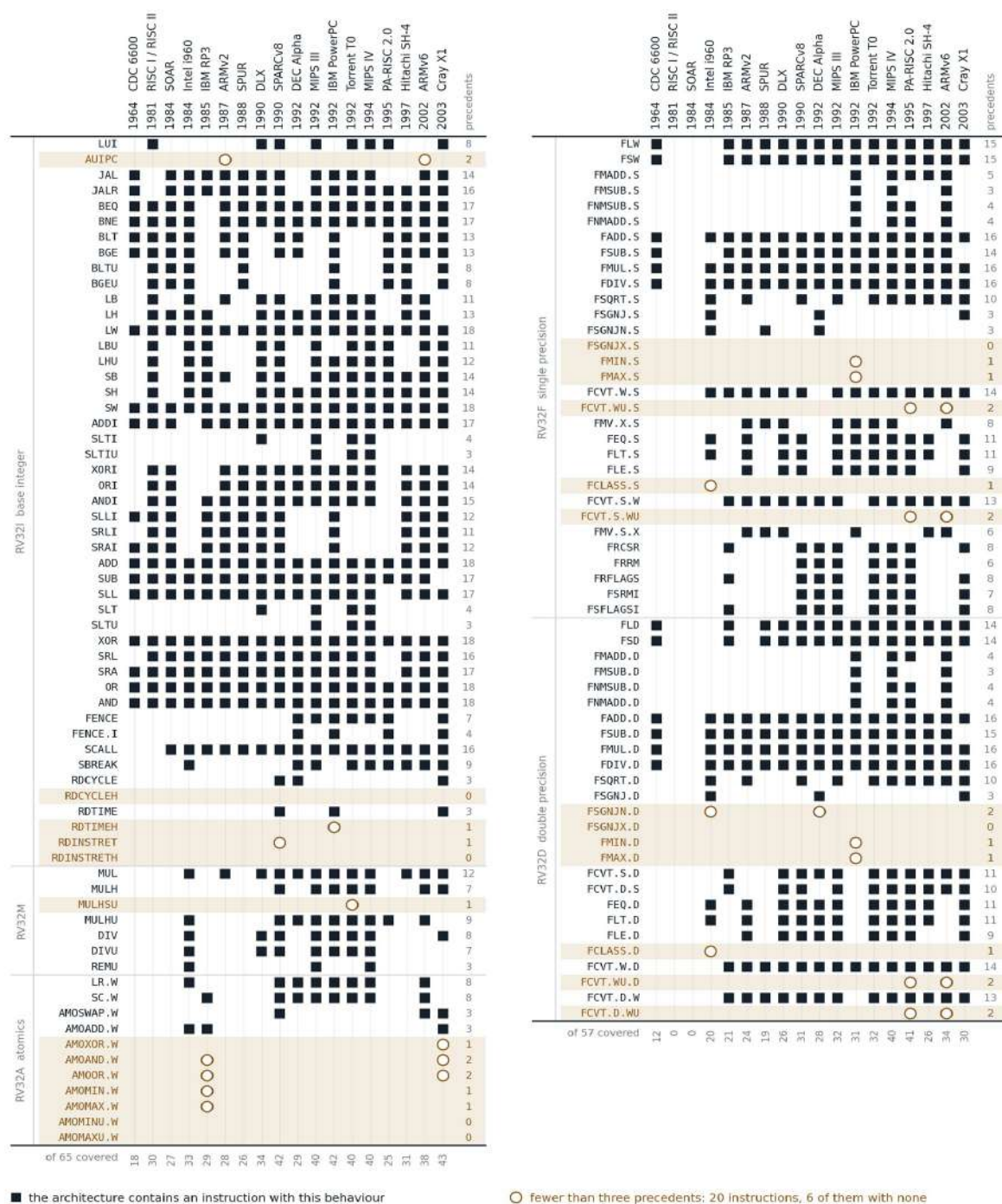
Постанак RISC-V архитектуре је последица недостатка рачунарске архитектуре која је довољно напредна да може да се употреби у Беркли лабораторији за развој рачунарских архитектура за паралелну обраду (енгл. *Parallel Computing Laboratory - Par Lab*) као основа за прототип новог процесора. Група истраживача која је укључивала Krste Asanović, Andrew Watermana и Yunsup Leea је уз подршку David Pattersona отпочела тро месечни пројекат за развој новог процесора за који се испоставило да ће трајати пет година [15].

Развој иницијалне ISA спецификације је трајао пет година јер су истраживачи желели да почну од нуле како би могли применити поуке многих грешака које су претходне RISC архитектуре направиле. Тако је RISC-V заправо колаж могућности из бар 18 RISC ISA. Од

иницијалних 122 инструкција RISC-V G ISA спецификације (G означава скуп могућности процесора над скаларним регистрима) 98 инструкција се појављује у барем три историјске RISC ISA спецификације. Слика 1 у себи садржи детаље о томе одакле је RISC-V ISA све позајмила инструкције [7].

ARM архитектура је убрзо одбијена као кандидат због скупих лиценци повезаних са прибављањем права на пројекте процесора који користи њен ISA. Током развоја основне архитектуре, интересовања за ову архитектуру су већ почела да се појављују из академије као и индустрије тако да је истраживачком тиму на Универзитету у Берклију постало очигледно да постоји недостатак напредне ISA спецификације доступне у свету отвореног кода.

Када су творци RISC-V званично почели да га пласирају у свет 2014. архитектура је већ почела да сазрева. Тадашњи прототипи су већ били упоредиви у перформансама као ARM Cortex-A5 породица процесора са значајно мањом површином у силицијуму. Овим су успешно показали да не постоји добар технички разлог да ISA буде пословна тајна, нити да се исплати скривати детаље имплементације од заједнице која ће користити ISA. Демонстрирали су предности отворености: мања цена, бржи развој и раније откривање грешака [8].



Data from T. Chen and D. A. Patterson, "RISC-V Genealogy", Tech. Rep. UCB/ECS-2016-6, EECS Dept., UC Berkeley, 2016. Redrawn; grouping and layout differ from the original.

Слика 1 - Наследство RISC-V, прецртана на основу оригиналног истраживања

2.2.1 Преглед главних карактеристика RISC-V

У периоду када је настао RISC-V, сасвим нове ISA спецификације просто нису биле успешне већ тридесет година уназад; ни у свету CISC нити архитектуре врло дуге инструкцијске речи (енгл. VLIW, Very Long Instruction Word) није било успешних потпуно

нових ISA [8]. RISC-V је променио постојеће стање тако што је спојио неке старе и неке нове концепте и иновирао у свету рачнарских архитектура на следеће начине [6]:

1. **Једноставност** – Оригинална ISA је далеко мања од, рецимо ARM.
2. **Пројекат отпочет од нуле** – RISC пројекат отпочет 25 година касније од иницијалне идеје; тако да је научено на грешкама претходних архитектура и избегнуто поистовећивање са технологијама које су специфичне за неку микро-архитектуру.
3. **Модуларност** – Основни ISA је мали, тако да је већина потребних проширења необавезна. Ово омогућује велику прилагодљивост и фино кројен наменски пројекат.
4. **Подршка за специјализацију** – Велики део простора у машинском коду је остављен за ужу специјализацију.
5. **Пројектован у складу са заједницом** – Свако ново проширење пролази кроз процес јавног дискутовања који може трајати изузетно дуго пре него што ново проширење ISA бива одобрено од стране RISC-V Фондације.
6. **RISC-V Фондација проширује ISA искључиво из техничких разлога** – мотиви приватних група могу бити тајне, маркетинг или чиста добит, док кроз транспарентан рад RISC-V Фондације овакви мотиви су неизводљиви.

2.2.2 Архитектуралне карактеристике скаларних RISC-V микро-архитектура

За анализу перформанси неког процесора неопходно је дубоко познавање детаља имплементације извршавања инструкција у самом хардверу. На примеру потпуно отворене RISC-V архитектуре – *Rocket* [16] развијани су многи процесори. У наставку овог поглавља ће се увести концепти микро-архитектура релевантни за тумачење резултата представљених у овом раду на примеру једног од кључних процесора који је утемељио принципе на којима RISC-V стоји данас [9].

Сваки модеран процесор ефективно има проточну структуру за извршавање инструкција која варира у броју стања кроз које инструкције пролазе, поседују сложене хијерархије скривене меморије и хардвер за предикцију гранања у машинском коду.

Када је у питању редослед којим се инструкције извршавају два решења су најраспрострањенија: у току у коме инструкције пристижу и спекулативан (ван редоследа у коме инструкције пристижу). У овом раду ће се разматрати микро-архитектура које извршавају инструкције проточно у редоследу у коме пристижу у проточну структуру.

Спекулативни процесори, пружају значајно веће перформансе али не гарантују да ће се инструкције извршити у редоследу датом у машинском коду. Спекулација утиче на то

како се разматрају резултати експерименталних мерења микро-архитектура, јер ће процесор оптимизовати како извршава инструкције, што утиче на метрике базиране на циклусима извршавања.

Процесор дат у примеру извршава инструкције у редоследу у којем су пристигле у проточну структуру. Поседује проточну структуру која подржава двоструко извршавање скаларних инструкција. Ово повећава паралелизам на нивоу извршавања инструкција и један је од стандардних начина како се постижу веће перформансе.

Такође, поседује заобилазни механизам (енгл. *bypass*) који омогућује да се резултат једне инструкције прослеђује другој зависној инструкцији без да се тај резултат упише у регистар. Ово утиче на измерену вредност за паралелизам процесора у зависности од сценарија. Процесор поседује и L1 скривену меморију који се дели између два процесорска језгра.

2.2.3 Архитектурале карактеристике векторских RISC-V копроцесора

Значајно је напоменути да је пример процесора из претходног поглавља први RISC-V процесор који је демонстрирао агилну методологију развоја RISC-V чиплет архитектура са функционалним векторским убрзивачем [9]. Модерни RISC-V процесори су усвојили принцип употребе векторског копроцесора као убрзивача обраде података у HPC (енгл. *high performance computing*) контекстима као што су: обрада сигнала, мултимедија, вештачка интелигенција и слично. Суштинска разлика је у томе како данашњи векторски копроцесори раде; нису *Cray* стила, већ SIMD. Супер рачунари *Cray* стила реализују векторску обраду података коришћењем дубоких проточних структура којима су се ланчале векторске инструкције [17].

RISC-V векторској екстензији ISA су много сродније SSE, AVX, или NEON архитектуре (x86 и ARM) које подржавају векторску реч непромењиве дужине, примера ради 256-бита. Разлика је у томе што је векторска реч у случају SSE, AVX или NEON архитектура, непромењиве дужине што значи да свака обрада на векторској јединици мора бити над 256 бита док је у случају RISC-V подесива по питању дужине векторских речи (енгл. *vector length* - VLEN), примера ради, ако је најдужа векторска реч VLEN 256-бита, RISC-V векторска јединица ће подржавати од 8 до 256-бита конфигурабилно.

Сваки RISC-V векторски копроцесор од стандарда RVV 0.7 па после се конфигурише на исти начин *vsetvl* инструкцијом. Инструкција *vsetvl* утиче на следеће глобалне параметре векторске јединице:

- Дужина вектора (енгл. vector length – vl), суштински је број елемената над којим ће се наредне векторске операције извршавати.
- Постављена величина елемената (енгл. set element width – SEW) над којим се врше наредне векторске операције, подржани су векторски елементи од e8, e16, e32 и e64 бита.
- Множитељ група векторских регистара (енгл. group multiplier – LMUL) служи за бирање броја векторских регистара у једној групи операција на векторској јединици. Стандардне вредности за LMUL групације су mf8, mf4, mf2, m1, m2, m4, m8, где су *mf* разломљене конфигурације у којима се групе елемената смештају у један векторски регистар. На пример, ако је VLEN 128 бита за величину елемената SEW од 16, LMUL може бити mf4 или више јер за векторске регистре од 64 бита је могуће 4 векторска елемената од 16 бита спаковати у један векторски регистар.

Формула по којој се максимални број елемената у вектору по векторској операцији који тражена конфигурација подржава одређује:

$$VLMAX = \frac{LMUL * VLEN}{SEW}$$

На пример, за VLEN = 128, SEW = 32 и LMUL = 4 VLMAX = 16, дакле, 16 елемената од 32 бита се могу обрадити са овом конфигурацијом на векторској јединици одједном.

Векторска јединица поседује и друге глобалне регистре који могу да се конфигуришу, али су овде представљени само они који су релевантни за могуће конфигурације векторских инструкција које се евалуирају у овом раду. Све могућности векторских јединица се налазе у ISA спецификацији [37].

2.2.4 Архитектура SрасеmіT-X60 RISC-V процесора базирана на његовим јавно доступним карактеристикама

У овом раду је главни фокус профилисање SрасеmіT-X60 (понекад спомињан као K1) процесора. Зарад разумевања могућности SрасеmіT-X60 процесора, приказаних на слици 2 у виду апстрактног дијаграма, основни појмови хетерогених RISC-V процесора се испод наводе:

На нивоу језгра:

- Кластер (енгл. Cluster) – је група од два или више процесорских језгра који деле ресурсе.
- Два различита X60 кластера (X60 и X60 Boost) други поседује АИ убрзивач (енгл. AI accelerator).

- Приступна тачка малој меморији језгру процесора (енгл. Scratchpad pool access port). Користи је АИ убрзивач за добављање података за обраду.
- Језгро процесора RISC-V (енгл. RISC-V 64GCVB core). GCVB означава проширења која процесор садржи:
 - G = IMAFD – генералне скаларне инструкције, по реду слова са десне стране једнакости: целобројне, множење и дељење, атомске, са покретним зарезом од 32-бита и са покретним зарезом од 64-бита.
 - C – смањене инструкције, верзије G инструкција које су мање од 64 бита у машинском коду. Постигинуто смањивањем скупа дозвољених регистара за операнде инструкција.
 - V – стандардно векторско проширење.
 - B – проширење за манипулацију бита и бајтова у регистрима.
- Скривена меморија за инструкције и податке (енгл. I-cache / D-cache) по сваком језгру процесора.
- Јединица за транспорт команди до процесора од модула за отклањање грешки (енгл. debug transport unit DTU).

На нивоу између процесора:

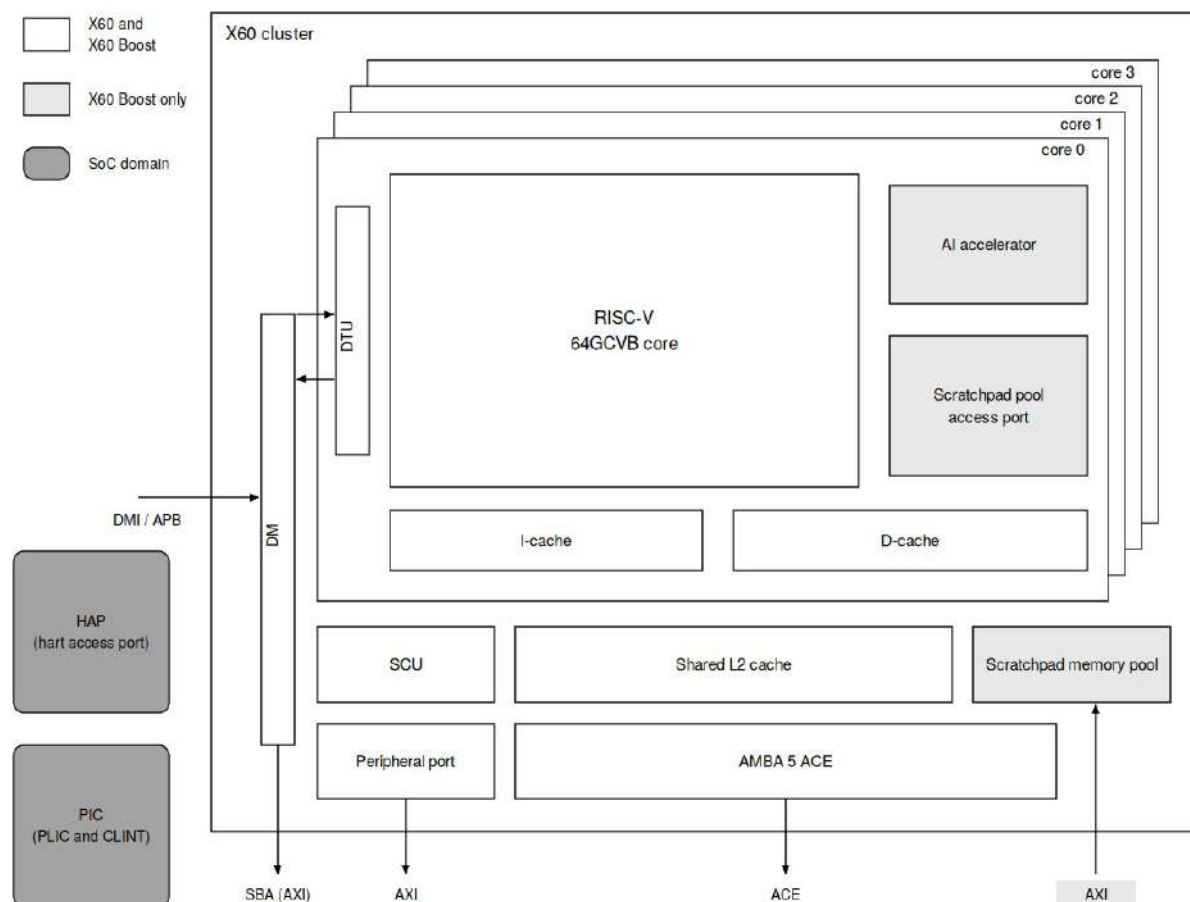
- Дељена скривена меморија другог нивоа, L2 (енгл. Shared L2 cache)
- Јединица за одржавање редовног стања дељене скривене меморије L2 између процесорских језгра (енгл. snoop control unit SCU).
- Базен мале меморије за АИ убрзавач (енгл. Scratchpad memory pool).

Портови који напуштају сваки кластер:

- Магистрала која повезује кластере са остатком компоненти система на чипу (енгл. AMBA 5 ACE to ACE), као што је DRAM (енгл. Dynamic Random Access Memory), динамичка радна меморија.
- Засебна магистрала за читање из регистара и приступ уређајима (енгл. Peripheral port to AXI).
- Додатна магистрала за модул за отклањање грешки (енгл. SBA - AXI) која се користи за писање и читање из меморије.
- Директна путања за друге компоненте система на чипу за писање у малу меморију (енгл. AXI to scratchpad memory pool).

Изван кластера:

- Приступна тачка језгрима (енгл. hart access port HAP) за отклањање грешки.
- Контролери за системске прекиде базиране на софтверским тајмерима као и спољним изворима (енгл. PLIC and CLINT).



Слика 2: Преглед SracemiT-X60 језгра

Главне карактеристике процесора од значаја за овај рад се могу видети у следећој табели [18]:

Табела 1: Карактеристике SracemiT-X60 процесора

SracemiT-X60	Могућност
Број кластера	2
Број језгра по кластеру	4
Величина L1 скривене меморије по језгру	32K
Величина дељене L2 скривене меморије	512K
Величина речи векторске јединице	256-бита који се извршавају на 128-бита x2 дужини магистрале
Убрзивач за AI до 2.0 TOPS (енгл. Trillion Operations Per Second)	На нултом кластеру
Спецификације RISC-V ISA	RISC-V 64GCVB и RVA22 стандарди
Подршка за PMU	Има

--	--

Овде се завршавају јавне информације о процесору. Као што се може видети, информације о томе како су реализоване скаларни и векторски део процесора су оскудне. Детаљи неопходни за дубоке микро-архитектуралне оптимизације, као што се могу видети у 2.2.2, нису јавно познати.

Како би се извршавао оптималан машински код на процесору, поготово када проточна структура извршава инструкције у реду у коме долазе, потребно је спознати специфичности микро-архитектуре и користити их ваљано. У оваквим сценаријима је једини начин експериментално доћи до микро-архитектуралних појединости.

2.2.5 Методологија мерења перформанси

Мерења у овом раду су спроведена над Linux оперативном систему. Апликативни софтвер који се извршава у Linux окружењу је природно случајан. Извршавање програма је непредвидиво и не важи централна гранична теорема из статистичке анализе [10]. Пошто ни са бесконачно мерења понашање случајних променљивих по инструкцији мереној неће бити униформно распоређено, неопходно је увести методологију која ће минимизирати грешку у неком прихватљивом временском оквиру.

У основи, проблем се може представити следећом формулом [10]:

$$\begin{aligned}
 T &= t_Q + \epsilon \\
 &= t_{Q_0} + \sum_j X_Q^{(j)} \tau^{(j)} + \epsilon \\
 &= n t_{P_0} + \sum_j \sum_{i=1}^{nk} X_Q^{[i](j)} \tau^{(j)} + \epsilon \quad (1)
 \end{aligned}$$

Где су:

- j – калибрациони број покретања тестова. Добијен Алгоритмом 1, представљеним у концепту решења.
- i – број инструкција у једном тесту.
- T – укупно време извршавања програма који се мери.
- t_Q - теоретско извршавање програма, укључујући грешку.
- ϵ - грешка настала због непрецизности мерача.
- t_{Q_0} - теоријско извршавање програма у идеалним условима.
- $\sum_j X_Q^{(j)} \tau^{(j)}$ - сума случајних променљивих да се грешка догоди и њена цена на перформансе. Ефективно кашњење у систему.
- $n t_{P_0}$ - Идеално време извршавање програма P_0 n пута.

- $\sum_j \sum_{i=1}^{nk} x_Q^{[i](j)} \tau^{(j)}$ - Сума кашњења у систему по понављању истог програма мереног по свакој вероватноћи да се кашњење деси на свакој инструкцији у програму P_0 .

Свако извршавање програма се своди на очекиван резултат и грешку коју мерач и случајни догађаји као што су: системски прекиди, миграције на процесору, промашаји у кешу и сл. уносе. Експериментално мерење извршавања програма $\hat{t}_{P_0}$, ће варирати значајно између мерења. Зато је неопходно одабрати од свих j мерења најмању аритметичку средину по редном броју мерења [10]:

$$\begin{aligned} \hat{t}_{P_0} &= \min\left(\frac{T_1}{1}, \dots, \frac{T_j}{j}\right) \\ &= t_{P_0} + \min(E_1, \dots, E_j) \end{aligned} \quad (2)$$

Одабирањем најмањег мерења из скупа свих мерења T , ефективно се приближава идеалном мерењу t_{P_0} јер је последица бирања минималне аритметичке средине по редном броју мерења што минимизира грешку E . Примена формула наведених у овом поглављу се налазе у поглављу 3.2.

Последица оваквог одабирања $\hat{t}_{P_0}$ је да ће се сваки тест покренути:

$$j_{P_0} = \frac{j(j-1)}{2} \quad (3)$$

пута. j_{P_0} је укупан број покретања тестова по калибрационом броју j који је последица Алгоритма 2.

2.2.6 Метрике прикупљане perf Linux алатом

У претходном поглављу се увела формална методологија на којој се робусна мерења у овом раду добијају. У оригиналном раду о робусним мерењима користили су се софтверски тајмери за прикупљање мерења [10]. Надоградња у овом раду је примена робустне методологије на PMU бројаче процесора који се добијају употребом perf Linux алата. Сам PMU је суштински софтверски омотач који је део платформског софтвера који належе над процесорском екстензијом [19].

Linux perf је саставни део системске програмске подршке Linux кернела који употребом системских позива, конфигурише и чита из PMU регистара. PMU је саставни део модерних процесора и у себи садржи велики број могућих догађаја на процесору који могу да се броје. Неке од метрика које модерни процесори чине доступнима [20]:

- Број протеклих циклуса на процесору,

- Број инструкција које су успешно прошле кроз проточну структуру. Неки процесори подржавају и финије детаље по типу инструкције која је успешно прошла кроз проточну структуру,
- Промашаји читања скривене меморије; по нивоу скривене меморије,
- Број инструкција које су блокирале проточну структуру на улазу,
- Број инструкција које су блокирале проточну структуру на излазу (услед заузетости свих приступних тачки).

Метрике које су од значаја за профилисање микро-архитектура се ефективно могу поделити у две групе: основне (M) и изведене (D). Горенаведене метрике се класификују као основне јер их омогућује директно базна програмска подршка. Сложене метрике су функције које за улаз имају основне метрике, а излаз су нове метрике:

$$D = f(M_1, M_2, \dots, M_N) \quad (4)$$

Пример једне сложене метрике је број успешно (engl. Retired) извршених инструкција по циклусу процеса IPC. Рачуна се као однос броја успешно извршених инструкција и циклуса рада процесора. Остале сложене метрике коришћене у овом раду се могу видети у 3.3.

Профилисање извршавања програма Linux perf алатом се може прикупљати на два нивоа: мерење извршавања независне апликације и инструментацијом директно у изворном коду. Предност профилисања апликације је у томе што се могу пронаћи критичне тачке, тј. функције које су најзахтевније у датом сценарију на процесору. Такође, perf поседује команде као што су *stat*, *record* и *report* које генеришу извештаје који садрже у себи и основне и сложене метрике на глобалном нивоу и на нивоу функција.

Са инструментацијом директно у изворном коду могуће је прикупити метрике директно на нивоу неког блока кода што је идеално за микро-архитектурална мерења јер минимизира грешке у мерењима тако што смањује кашњења која производи остатак система јер се не посматра.

2.2.7 Избор метрика корелације резултата

Радови који пореде моделе перформанси на нивоу инструкција и основних блокова на хардверу често наводе две врсте мерења валидности експерименталних резултата: меру грешке и корелацију рангова. Сродни радови који презентују алате за x86 архитектуру као што су Bhive [32], DiffTune [33] и uiCA [34] се ослањају на неке мере грешке и корелације рангова. Овај рад прати овај начин представљања резултата додавањем одступања базираног на геометријској средини односа. Зато се приказ резултата своди на опис метрика

корелације изабраних за анализу датих резултата као и аргументе зашто ове метрике имају смисла у контексту у коме се примењују.

2.2.7.1 Корелација базирана на Кендаловом коефицијенту корелације рангова

Кендралов τ (tau) коефицијент корелације мери редоследну асоцијацију између два низа мерења и тиме даје степен сличности по рангу сваке од вредности која се посматра. Како би се добио Кендралов τ (tau) коефицијент потребно је дефинисати сагласност и несагласност у подацима.

Формула за Кендралов τ (tau) коефицијент је:

$$\tau_b = \frac{n_c - n_d}{\sqrt{(n_0 - n_1)(n_0 - n_2)}} \quad (5)$$

Где су:

- **Сагласни** n_c ако било која два пара резултата (x_i, y_i) и (x_j, y_j) ако важи $x_i > x_j$ и $y_i > y_j$ или $x_i < x_j$ и $y_i < y_j$.
- **Несагласни** n_d ако $x_i > x_j$ и $y_i < y_j$ или $x_i < x_j$ и $y_i > y_j$.
- А коефинцијенти у корену:

$$n_0 = \frac{N(N-1)}{2}, \quad n_1 = \frac{\sum_k t_k(t_k-1)}{2}, \quad n_2 = \frac{\sum_l u_l(u_l-1)}{2} \quad (6)$$

Где су:

- t_i - број истих вредности у i -тој групи x_i
- u_j - број истих вредности у j -тој групи y_j

Кендралов τ (tau) коефицијент је број од $[-1, 1]$ који се интерпретира као:

- **1** – резултати имају исти редослед, савршено рангирање.
- **0** – не постоји веза; редослед је насумичан.
- **-1** – редослед је сасвим супротан.

Оваква метрика је корисна за естимацију цена инструкција зато што се распоређивачи инструкција морају одлучити коју прво да изврше. Стога је распоређивачком моделу и другим оптимизацијама програмских преводиоца битнији релативни поредак цена него њихове апсолутне величине [32], а Кендралов коефицијент ово веродостојно представља.

2.2.7.2 Средња апсолутна процентуална грешка (MAPE)

Средња апсолутна процентуална грешка је статистичка метода која за сваку инструкцију срачуна грешку између јавно доступне вредности и експериментално измерене вредности у овом раду. Представља се као:

$$MAPE = \frac{\sum_{i=1}^N |b_i - a_i|}{m_i} \quad (7)$$

Где су:

- $m_i = a_i$ за скаларна и векторкса мерења експериментални резултат.
- b_i јавно доступни резултати са којима се експериментални резултати пореде.

Слабост ове методе је њена асиметричност. Примера ради, у случају `vadd.vv` инструкције ако се као $m_i = a_i$ узме `rvv-bench` или експериментална мерења овог рада утиче на MAPE вредност за дату инструкцију:

$$\frac{|3.93 - 1.00|}{1.00} = 293 \% \quad \text{наспрам} \quad \frac{|1.00 - 3.93|}{3.93} = 75 \%$$

Ипак, ова метрика се представља зато што је стандардан начин за представљање сличних модела експеримената [32, 33, 34]. Да би се премостио овај недостатак уводи се још једна метрика која се не ослања на апсолутну вредност и зато боље моделује систематско одступање.

2.2.7.3 Геометријска средина односа (GMR)

Идеја је да се за сваку инструкцију срачуна однос $r = b / a$ између јавно доступне вредности и експериментално добијене и онда да се пронађе средња вредност овог односа.

У случају када је једно мерење 2x веће од модела ($r = 2$) и једно у коме је упола мање ($r = 0.5$), аритметичка средина ова два односа би дала $(2 + 0.5) / 2 = 1.25$ што наговештава да је мерење 25% изнад нормале што је погрешно. Зато се рачуна средња вредност логаритама:

- $\ln(2) = +0.69$
- $\ln(0.5) = -0.69$

Просечна вредност је 0, и $e^0 = 1$, што значи да нема одступања.

Стога се уводи формула:

$$GMR = \exp\left(\frac{1}{N} \sum_{i=1}^N \ln r_i\right) \quad (8)$$

Ово је основни начин на који SPEC, најсвеобухвативнији скуп алгоритама за изивршавање на процесорима и оцењивање његових перформанси у реалним сценаријима, извештава о резултатима мерења [35]. GMR ће у евалуацији решења послужити као показатељ смера грешке MAPE; у зависности да ли је изнад или испод један знамо да ли је вредност измерена овим решењем већа или мања од јавно доступних података, респективно.

2.3 Сродни радови

Значајан утицај на овај рад су били радови који ће у наставку бити описани.

2.3.1 Један начин за профилисање Linux perf алатом аутовекторизације програмског преводиоца на RISC-V [11]

Рад који представља рани прототип овог рада бави се профилисањем извршавања апликација како би се идентификовали недостаци у искоришћењу векторских јединица на процесорима. Посматране векторске инструкције се преводе из виших програмских језика, обично у петљама у којима постоји велики потенцијал за паралелизацију, коју програмски преводиоца препозна и преведе. Овако генерисан векторски код се назива аутовекторизован.

Уз помоћ Linux perf алата се врше мерења на нивоу апликације. Методологија процењивања степена аутовекторизације и њеног утицаја на укупне перформансе представљена у раду се заснива на корелацији најзахтевнијих функција са степеном аутовекторизације која се рачуна као проценат петљи које су преведене у машински код употребом векторских инструкција.

Фундаментална разлика је ниво на коме се врши профилисање. Ефективно, методологија профилисања аутовекторизације је независна од микро-архитектуре. Програми који су релевантни за овакву студију поседују алгоритме који се имплементирају у вишим програмским језицима за обраду великих количина података (обраде слике, сигнала, мултимедија, вештачке интелигенције и сл.). Резултат студије је потенцијални простор у коме се ручним писањем векторског кода или додатним оптимизовањем програмског преводиоца може више перформанси извући из постојећег хардвера.

За дубоко профилисање микро-архитектуралних специфичности сваке инструкције доступне на процесору, мерења на нивоу апликације нису довољна. Ипак, обрада извештаја Linux perf алата који садржи метрике сродне онима из 2.2.6. инспирисао је овај рад и начин посматрања основних метрика.

2.3.2 Решење за профилисање RISC-V инструкција rvv-bench [12]

Алат rvv-bench [21] је прихваћен од стране заједнице као ауторитет за микро-архитектурална мерења процесора коришћеног у овом раду. Ослања се на директно читање из *rdcycle* инструкције која служи као бројач циклуса, али може да употреби и Linux perf системске позиве ако платформа то подржава.

Недостатак је ослањање искључиво на број циклуса извршавања програма и узимање чисте аритметичке средине по укупном броју инструкција извршених. Ако се блок кода профилише у окружењу у коме влада оперативни систем, чиста аритметичка средина не гарантује стабилна мерења [10]. Такође, rvv-bench не поседује стандардан начин за додавање нових метрика базираних на PMU бројачима по произвољном типу програма који се мери.

Репозиторијум је коришћен за темељна мерења разних RISC-V процесора и резултати су јавно доступни тако да, где год су информације доступне, резултати мерења овог рада ће се упоредити са онима добијеним rvv-bench алатом.

2.3.3 Практично PMU профилисање и анализа крова на RISC-V платформама независна од хардвера [13]

А. Баташев је у свом раду дефинисао алат *mperf* који је један начин да се профилишу перформансе RISC-V микро-архитектура употребом могућности PMU процесора. Концепт дефинисан у раду се заснива на динамичкој инструментацији инструкција које се генеришу око захтевних делова изворног кода уз помоћ проширења међу репрезентације кода програмског преводиоца LLVM. Додају се позиви функција који мере перформансе употребом Linux perf API позива, као и у овом раду. Сличности између овог рада и алата *mperf* су начин инструментације Linux perf бројача и дефинисање софтверског алата који конфигурабилно подиже микро-архитектурале тестове које мери.

Недостатак који се надомешћује у овом раду је примена методологије робустног мерења. Начин на који су мерења приказана у раду А. Баташева не узима у обзир низ типова кашњења који утичу на прецизност мерења апликативног софтвера (види 2.2.5.). Стога,

доводи се у питање валидност експерименталних резултата. Највећи изазов код профилисања микро-архитектура је доћи до поузданих експерименталних резултата [10].

2.3.4 uops.info: категоризација кашњења, проточности и заузетости портова на Intel x86 процесорима [14]

Алат uops.info се разликује већински од алата представљеног у овом раду. За почетак у питању је сасвим друга архитектура процесора. Процесори профилисани са uops.info алатом су углавном процесори који извршавају инструкције ван редоследа којим пристижу у проточну структуру процесора, што захтева промену у основним дефиницијама метрика.

Зато је методологија на основу које дефинишу микро-архитектуралне тестове са којима мере кашњење и проточност по инструкцији није сасвим компатибилна са оном представљеном у концепту решења овог рада. Ипак, показује универзалност проблема са којима се овај рад суочава; проналажење тачних вредности о појединостима микро-архитектуре употребом математичког модела на основу кога се извршавају тестови по циљаној инструкцији тако да се минимизира грешка са довољним бројем понављања и пронађе права вредност цене микро-архитектуре.

3. Концепт решења

У овом поглављу се објашњава идеја иза самог решења. Концепт је подељен блоковски, тако да су најбитнији делови сваког модула издвојени. На следећој слици је приказан преглед одозго архитектуре решења:



Слика 3: Преглед одозго арахитектуре решења

Важно је напоменути значење иза избора облика за сваки блок у дијаграму. Облим правоугаоницима су представљени концепти који групишу функционалности програмског решења: датотеке, C++ простори имена (енгл. `namespace`) или основни концепт иза неког објекта. Правоугаоницима правих ивица су представљене класе, методе или функције које су саставни део програмског решења. У наставку ће сваки блок бити описан са лева на десно.

3.1 Окружење за извршавање микро-архитектуралних тестова

Леви блок садржи у себи окружење у коме се тестови микро-архитектура извршавају. Омотач који омогућује динамичко бирање тестова за извршавање и механизме провере тачности резултата је Google оквир за тестирање gtest писан у C++ [22]. Оквир за тестирање gtest ради тако што проширује главну функцију main C++ програма. Уз помоћ макро функција TEST_P се сакупљају тестови генерисани, а INSTANTIATE_TESTSUITE_P прави окружење за извршавање свих тестова које је филтер обухватио. По сваком типу теста (о чему ће бити више речи у делу о генератору) дефинише се класа која наслеђује од gtest testing::Test класе. На овај начин се дефинишу изведене класе:

- *LatencyPerf* – агрегатор класа која дефинише начин на који се извршавају микро-архитектурални тестови који мере кашњење инструкција.
- *ThroughputPerf* – агрегатор класа која дефинише начин на који се извршавају микро-архитектурални тестови који мере проточност структуре по инструкцији. Поред кашњења, проточност је најосновнија метрика по инструкцији. Ефективно, проточност јесте степен паралелизма.

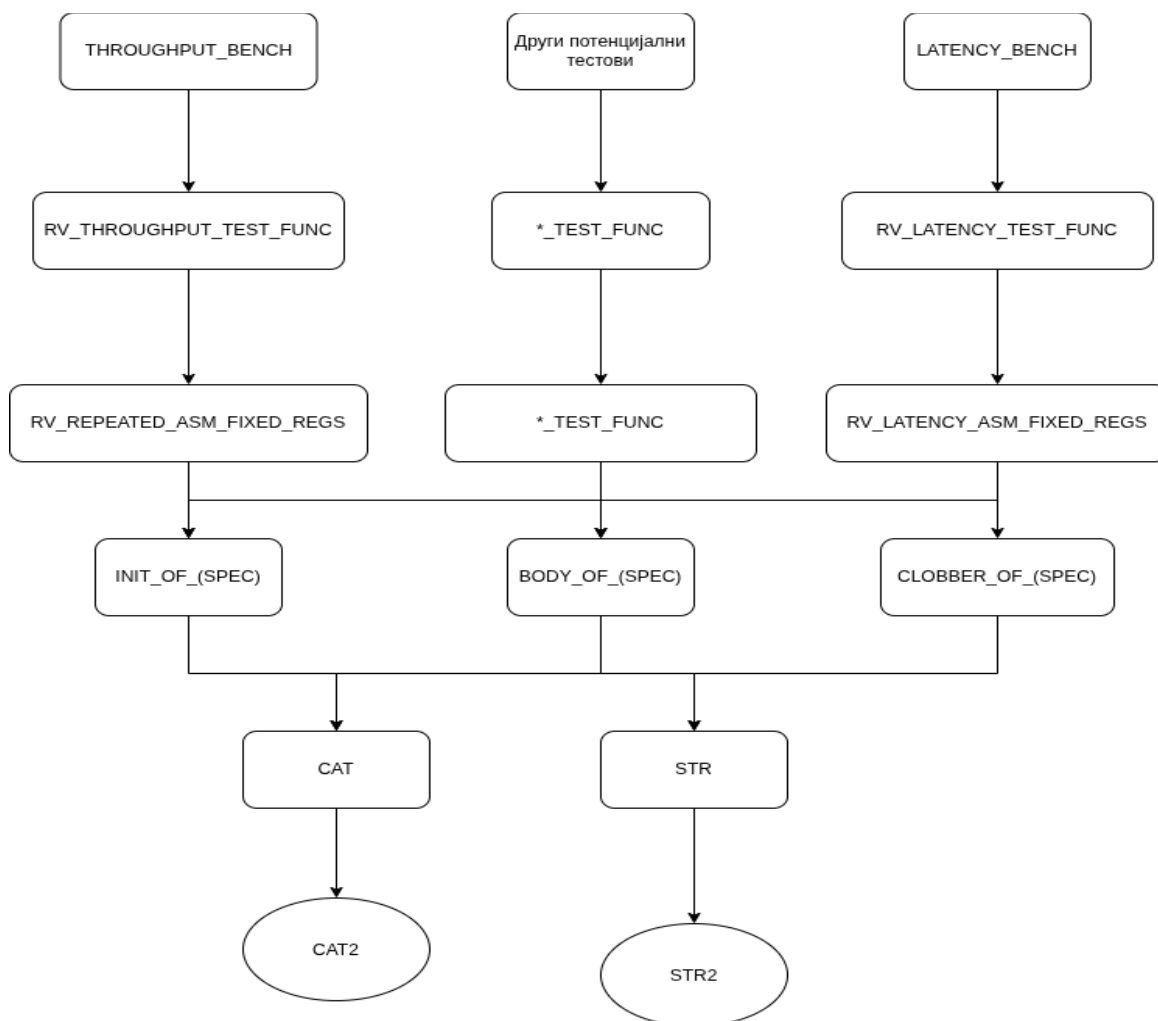
Окружење за извршавање тестова осим што агрегира тестове и групише их по класи извршења за задатак има и да одради помоћне прорачуне и извести о резултатима тестова. Следећи блокови помажу у реализацији помоћних функционалности окружења за извршавање тестова:

- *RunWithPerf* – дељена функција између изведених класа за извршавање тестова која на основу типа теста бира конфигурацију и извршава тест. Суштински имплементира извршни део појединачних тестова логичког тока извршавања програма.
- *CalibrateJ* – служи за проналажење потребног J броја итерација са којима ће се елиминисати грешка унесена у експериментална мерења бројачима. Напомене се да постоји разлика између потребног и довољног. J. J. Chen, J. Revels, и A. Edelman напомињу да се у J мерења статистички гарантује да је кашњење минимизирано; али J је обично доста веће него што је потребно. Стога се за каснија покретања истог теста може применити пророчка функција *v* која налази довољно J [10]. У овом раду се овај корак прескаче, користи се потребно J.
- *utilities* – је скуп шаблонски функција које служе за парсирање специјалних аргумената командне линије који нису део gtest окружења, али и исписивање резултата мерења по тесту на конзолу. Окружење за извршавање тестова ће коначни извештај сместити у .json датотеку.

3.2 Генератор микро-архитектуралних тестова

Генератор микро-архитектуралних тестова је специфичан по томе што ради у времену превођења изворног кода зато што је базиран на претпроцесорским макро функцијама. Излаз из генератора су функције чији су потписи у С формату а тела чист RISC-V асемблерски код.

Следећи дирекциони ациклични граф концептуално показује како претпроцесор генерише микро-архитектуралне тестове:



Слика 4: Граф језгра претроцесорског макро гененратора тестова

Ово је репрезентативни део претпроцесорских макроа, специфична имена макроа су груписана у класе које представља мања група чворова ради читљивости архитектуре генератора. Даље објашњење се заснива на опису сваког слоја графа, одозго на доле.

3.2.1 Приступне макро функције

На врху се налазе приступне макро функције које диференцирају која врста теста ће бити генерисана. Приступне макро функције се сакупљају у листе коришћењем X-макро

технике. X-макро је техника коришћења макроа тако да се прикупљају у низове [23]. Дате листе се сакупљају из *.inc* датотека. Груписање макроа у низове који садрже све инстанце тестова по некој групи је функционалност која се показала као довољна за филтрирање тестова по имену употребом *gtest* извршног окружења. Улазни макрои су уједно и врсте тестова које алат генерише:

- LATENCY_BENCH – макро функција која генерише тестове кашњења по инструкцији у виду C функције са асемблерским телом.
- THROUGHPUT_BENCH – макро функција која генерише тестове проточности по групама инстукција у виду C функција са асемблерским телом.

Следе примери асемблерског кода за сваку од група тестова које алат генерише:

Табела 2: Пример генераторског LATENCY_BENCH теста

```
add_latency8192:
    li t0, 1
    li t1, 3
    .rept 8192
    add t0, t0, t1
    ret
```

Табела 3: Пример генераторског THROUGHPUT_BENCH теста

```
add_throughput_x6:
    li a0, 1
    li t0, 2
    li a1, 3
    li t1, 4
    li a2, 5
    li t2, 6
    li a3, 7
    li t3, 8
    li a4, 9
    li t4, 10
    li a5, 11
    li t5, 12
    .rept 8192
    add a0, a0, t0
```

```

add a1, a1, t1
add a2, a2, t2
add a3, a3, t3
add a4, a4, t4
add a5, a5, t5
.endr
ret

```

Као што се може видети из претходних табела, тестови су реализовани у форми: иницијализација регистра, покретање тестова који се формирају као низ асемблерских инструкција поновљених уз помоћ `.rept` асемблерске директиве.

Сви тестови се сакупљају коришћењем X-макро архитектуре у низове по врсти теста. Скуп свих тестова се налази у програмском решењу.

3.2.2 Генерисање C++ функција путем макро претпроцесора за микро-архитектуралне тестове

Други и трећи нивои генератора тестова се могу посматрати као средњи. Служе да кажу проширивачу (енгл. *expander*) како да формира нове макрое у зависности од текстуалног уноса параметара X-макроа. Сваки X-макро користи макро функцију типа теста са суфиксом `_TEST_FUNC` који формира потпис C++ функције. Пример једне макро функције за генерисање C++ функције је приказан испод; за остале типове тестова се C++ функције слично генеришу.

Табела 4: Пример макро функције која генерише C++ функцију

```

define RV_THROUGHPUT_TEST_FUNC(FUNC_NAME, ITERS, INIT, BODY, ...) \
extern "C" __attribute__((noinline, aligned(64)))                \
void FUNC_NAME()                                                 \
{                                                                    \
    RV_REPEATED_ASM_FIXED_REGS(                                  \
        ITERS,                                                    \
        INIT,                                                      \
        BODY,                                                      \
        __VA_ARGS__                                                \
    );                                                            \
}

```

}

Затим је тело функције описано још једном угњежђеном макро функцијом која је срж генератора; на овом нивоу се формира тело теста коришћењем уграђене функције `asm volatile()` програмског преводиоца којом се у C++ код усађују машинске инструкције у асемблеру:

Табела 5: Пример макро функције која генерише тело C++ функције.

```
#define RV_REPEATED_ASM_FIXED_REGS(ITERS, INIT, BODY, ...) \
asm volatile ( \
    ".option push\n" \
    ".option norvc\n" \
    INIT \
    ".rept " STR(ITERS) "\n" \
    BODY \
    ".endr\n" \
    ".option pop\n" \
    : \
    : \
    : __VA_ARGS__ \
)
```

Приви део аргумената `asm volatile` функције је текст асемблерског кода који садржи инструкције али користи и директиве за преводилац асемблера као што су `.option` које дефинишу услове превођења C++ функције и `.rept` које кажу преводиоцу колико пута да понови инструкције у угњежђеном асемблерском блоку.

Напоследку су параметри који наговештавају програмском преводиоцу које променљиве, литерали и показивачи или у овом случају сирови регистри се користе у асемблерском блоку (енгл. *clobbered*).

Како би се у потпуности генерисале C++ функције претпроцесор ради експанзију макро функција (енгл. *expansion*). Аргументи макро функција које генеришу тело C++ функције имају генеричка имена `INIT`, `BODY` и `__VA_ARGS__` али су заправо макро функције које се формирају припајањем префикса и суфикса у зависности од циљаног теста. Следеће поглавље описује експанзију макроа све до елементарне RISC-V инструкције.

3.2.3 Експанзија сродних макроа по типу теста

Свака C++ функција теста садржи три градивна блока:

1. Иницијализацију регистра на неке тестне вредности.
2. Садржи тело теста које се састоји од i инструкција.
3. Преноси повратну информацију о томе који регистри су коришћени у телу функције.

Како би се постигла генерализација градивних блокова уводи се експанзија сродних макроа у последњем делу генератора. Концептуално се ова генерализација постиже употребом спајања имена макроа. Како би се из два имена формирало треће валидно име за макро потребно је да макро са трећим именом постоји у изворном коду. Техника спајања макроа налаже на следећим основним макроима:

Табела 6: Основни макрои спајања макроа

```
#define STR2(x) #x
#define STR(x) STR2(x)
#define CAT2(a, b) a##b
#define CAT(a, b) CAT2(a, b)
```

Макрои CAT и STR припајају текст у два различита случаја: CAT припаја имена макроа датих у изворном коду, док STR припаја литерале асоциране са дефиницијом макроа. Употребом спајања могуће је дефинисати генеричке макрое за иницијализацију, коришћење регистра и извршавање асемблерског кода. У графу на слици 4 су приказани чворови INIT_OF_(SPEC), BODY_OF_(SPEC) и CLOBBERS_OF_(SPEC) који употребом технике спајања макроа се трансформишу у наменске макрое иницијализације, коришћења регистра и извршног дела теста. Генерисање макроа иницијализације и извршавања теста користи и друге макрое који формирају инструкције и прослеђују им регистре:

Табела 7: Пример макро дефиниције асемблерске инструкције

```
#define ALU_OP2(OP, RD, RD_VAL) \
    (ALU_OP2, OP, RD, RD_VAL)
```

Слагањем макро дефиниција асемблерских инструкција добијају се сложени макрои иницијализације и извршавања тестова којима се путем технике експанзије макроа могу генерички користити. На овај начин се постиже велики степен поновне искористивости постојећег кода за додавање нових тестова.

3.3 Омотач око Linux perf алата – PerfWrapper

Коришћење омотача око Linux perf алата је неопходно јер, у зависности од теста, је потребно конфигурисати процесорске бројаче за различите метрике и то је основна функција простора имена PerfWrapper. PerfWrapper простор имена у себи садржи класе које директно руководе: конфигурацијом, груписањем, сакупљањем и обрадом метрика прикупљених коришћењем Linux perf C API. Сваки модул се у наставку описује:

PerfGroup – језгро омотача које служи да групише perf бројаче у шаблонску класу по броју основних метрика. Користи низ PerfCounter објеката који су скривени (видљиви на нивоу датотеке). Сваки PerfCounter региструје бројач на PMU употребом perf_event_open системског позива [24]. Низ бројача који су успешно отворени се затим групише са једним бројачем који је вођа групе. На тај начин се даје до знања perf распоређивачу да се дати бројачи у истом тренутку одабирају ако је то могуће. Под условом да процесор поседује довољно независних регистара распоредиће све бројаче одједном. Ако овај услов није задовољен, време активног (енгл. active) бројања постаје проценат времена које је бројач провео спреман (енгл. ready) чекајући да се ослободи регистар [25].

Осим груписања бројача, PerfGroup омогућује и започињање, заустављање и читавање мерења по сваком бројачу. Специфично је читавање мерења. Због тога што се мерења врше понављањем истог теста J_p пута, а бира се минимална аритметичка средина између сваког теста, читање групе подржава динамичко ажурирање најбољег резултата мерења по некој хеуристици која је дата као показивач на функцију. За потребе овог рада хеуристика је бирање минималне аритметичке средине.

PerfCollector – шаблонска класа по броју основних метрика која која складишти оптималне резултате мерења по свакој основној метрици. Директно је спрегнута са PerfGroup класом јер сирови резултати које PerfGroup произведе су обрађени током извршавања теста и складиштени употребом PerfCollector класе. Функција хеуристике коју PerfGroup користи је пријатељска функција (у C++ friend функција) PerfCollector класе.

PerfConfig – шаблонска структура која у времену превођења дефинише скуп основних метрика и сложених метрика асоцираних по тесту. Метрике које су неопходне типу теста су познате током превођења, тако да се дефинишу као inline constexpr на глобалном нивоу простора имена (у C++ ће се овако дефинисане променљиве наћи са подацима у машинском коду програма). Скуп основних метрика је информација неопходна да се у времену превођења зна какви PerfGroup и PerfCollector шаблони се корисите по типу теста.

metric_calculations – скуп изведених метрика које PerfConfig користи. Дефинише шаблонску структуру derived_metrics. Шаблон derived_metrics даје име свакој функцији

која рачуна сложене метрике. Функције за рачунање сложених метрика, `metric_fn` све имају повратну вредност `std::optional<double>` јер не постоји гаранција да ће за дати улаз бити могуће израчунати тачан излаз. Аргумент `metric_fn` је `std::span<const uint64_t>` јер је ово оптималан начин за прослеђивање шаблона `std::array` типа низа по референци; исечак низа `span` овде омогућује варијабилан број аргумената где се сваком бројачу приступа преко `enum class Events` индекса. Све подржане сложене метрике се налазе у програмском решењу.

Увођење шаблона у `perf` омотач се ради јер су детаљи тестова познати током времена превођења. Ипак, сваки део омотача је током времена извршавања уско спрегнут тако да је последица увођења конфигурисања бројача по типу теста током времена превођења била шаблонско стварање групе бројача као и сакупљача резултата извршавања тестова. Напоследку, цео омотач је шаблонски реализован.

3.4 Скрипта за извршавање алата за мерење микро-архитектура

Скрипта `run.sh` је помоћна приступна тачка извршавању алата. Решава проблем стандардног начина покретања, јер за исправна мерења постоје многи фактори који утичу на тачност мерења између покретања. Главне могућности скрипте за покретање алата су:

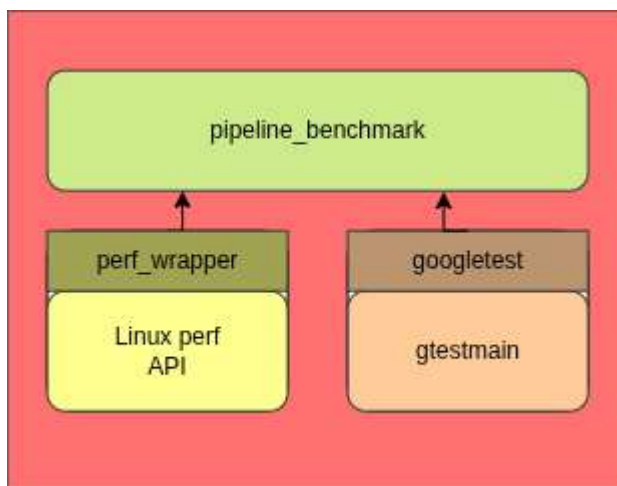
- Могућност инструментације на нивоу апликације – корисно ако је сама апликација микро-архитектурални тест
- Фиксирање језгра на коме се алат покреће – битно јер елимише кашњења услед миграције програма са једног на други процесор [25].
- ASLR (engl. Address Space Layout Randomization) – Насумично распоређивање адресног простора за неки програм је механизам заштите у Linux-у од статичког закључивања критичних делова неког програма. У контексту покретања микро-архитектуралних тестова ASLR делује неповољно јер уводи непредвидиво кашњење услед непредвидивости распореда програмског адресног простора које утиче на промашаје у скривеној меморији и предикцији гранања при сваком извршавању програма [28].
- Контрола пермисија у кернелу – неки `perf` догађаји су резервисани само за кернел, скрипта за покретање тестова може омогућити максимални ниво пермисија апликативном програму у контексту коришћења PMU.
- Филтрирање по тесту – директно прослеђивање `gtest` окружењу шаблонских филтера који омогућују висок ниво контроле над избором теста за покретање [26].
- Наменско бирање параметера `J` – ако корисник унапред зна `J` за које жели да тестира. Подразумевано `J` рачуна алат ако вредност није дата.

3.5 Организација стаке листе за превођење

Извршна датотека алата представљеног у овом алату се зове `pipeline_benchmarks`. Како би се `pipeline_benchmarks` направио користи се стаке окружење за превођење. Како би се C++ програм превео у стаке окружењу неопходно је дефинисати `CMakeLists.txt` датотеку која поседује у себи три модула неопходна за превођење алата:

- Путању до `googletest` репозиторијума, модул који се повлачи из изворног кода је `gtestmain`.
- `perf_wrapper` статичка библиотека која је део програмског решења, дефинисана је као засебан модул ради ефективније дистрибуције функционалности инструментације `perf` Linux у изворном коду у даљим радовима.
- `pipeline_benchmarks` – извршна датотека алата. Повлачи `perf_wrapper` и `gtestmain` и формира извршно окружење алата.

Следећи дијаграм приказује архитектуру по којој се `pipeline_benchmark` преводи:



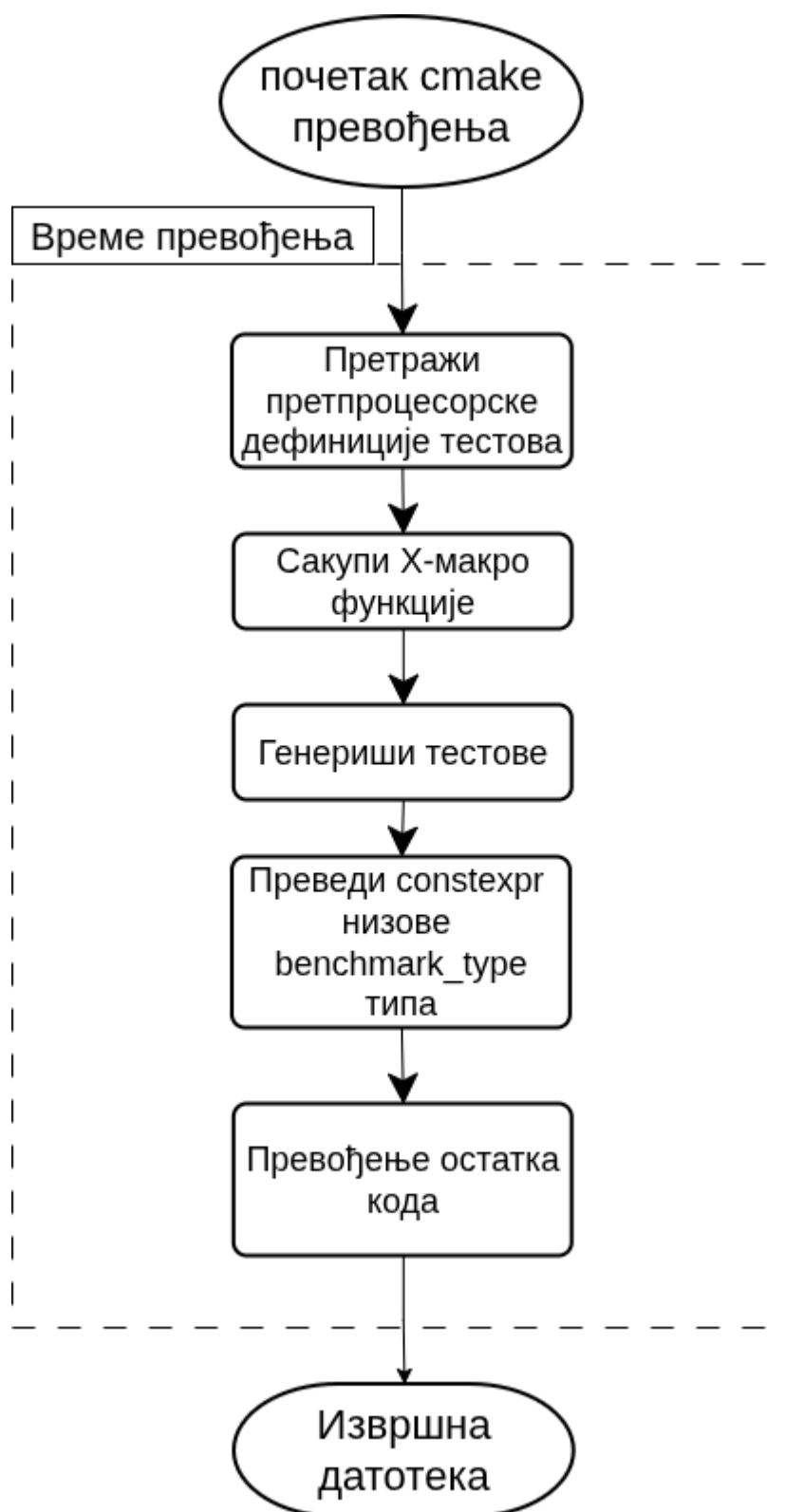
Слика 5: Архитектура `CMakeLists.txt`
`pipeline_benchmarks` алата

3.6 Логички ток алата за мерење микро-архитектура

У овом поглављу се приказује ток извршавања програмског решења овог рада. Ток извршавања се може поделити на време превођења и време извршавања. Изузев главних алгоритама који омогућују робусна мерења они су приказани одвојено. Следећи дијаграми представљају ток извршавања обе фазе коришћења алата за мерење микро-архитектура.

Ток превођења алата у основи је релевантан за претпроцесорски генератор тестова. Главни кораци су:

1. Претражи макро дефиниције – у овој фази се формирају табеле симбола макроа.
2. Сакупи X-макрое – Издвојено специфично јер је у графу позива макроа улазна тачка сваки од X-макроа.
3. Генериши микро-архитектуралне тест функције (учињено у језгру претпроцесора из поглавља 3.2.) – користе механизам проширивања макроа базиран на спајању имена макроа како би се генеричка имена макроа у дефиницији претворила у наменски макро тест.
4. Превођење низова који сакупљају све генерисане тестове по типу теста. Овим низовима је познат број елемената у времену превођења јер је број генерисаних тестова (X-макроа) познат у времену превођења.
5. Остатак превођења; прво шаблонски део решења па остатак извршног окружења.



Слика 6: Блок дијаграм тока превођења алата за мерење микро-архитектура

3.6.1 Језгро извршног окружења

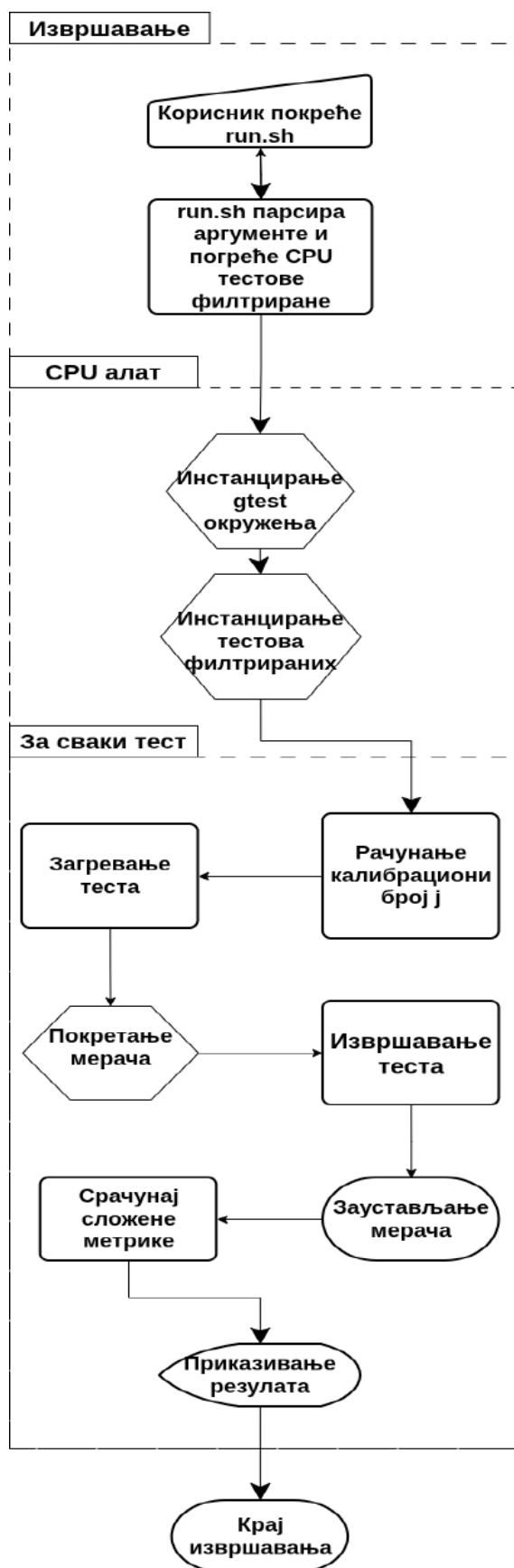
Извршно окружење служи да изврши тестове. Сваки тест се извршава на исти начин. У наставку овог поглавља су представљени дијаграм тока извршавања, алгоритми бирања калибрационог броја и извршавања тестова.

Ток извршавања алата се може поделити у три фазе:

1. Парсирање аргумената – класично С парсирање аргумената се у овом алату изводи из два пута: први пут за аргументе који су прослеђени алату коришћењем `run.sh` скрипте и други пут се прослеђују остали аргументи `gtest` окружењу.
2. Подизање окружења за покретање тестова `gtest` – ову фазу покрива само извршно окружење и једини део релевантан за овај рад су инстанцирање намеских тестова микро-архитектуре употребом изведених класа `gtest` окружења и макро функција које интегришу низове тестова у `gtest` окружење.
3. Извршавање сваког теста као и извештавање о резултатима. Језгро извршног окружења.

Улаз у свако извршавање теста је калибрациони број j . Калибрациони број j се добија тако што се пронађе аритметичка средина празних мерења. Улаз у извршавање сваког теста је заправо улаз у функцију калибрације а то су: број понављања мерења E и идеална прецизност мерача. Идеална прецизност је узета да буде 1 циклус процесора, то јест, идално би било да сваки циклус процесора је могуће измерити. Ово није реалистично због разних кашњења тако да нам експериментална прецизност мерача постаје калибрациони број j . Овај број представља број довољних итерација мерења теста за елиминацију грешке услед увођења мерача у датом контексту на процесору.

Извршавање тестова је урађено коришћењем две функције приказане у алгоритму 2. Подела је била неопходна због тога што су конфигурације тестова познате у времену превођења и као такве бирање адекватних метрика за примену мора бити урађено у изворном коду статички. Спољна функција на основу врсте теста подиже `perf` сакупљач резултата метрика на основу статичке конфигурације. Затим се `perf` догађаји креирају у унутрашњој функцији, загрева се тест и извршава j_{p_0} пута и резултати сакупљају по датој хеуристици (минимална аритметичка средина). Напослетку се у спољној функцији рачунају сложене метрике над сакупљеним резултатима основних метрика и корисник бива обавештен о резултату исписом на терминалу али и записом у `.json` датотеку (покривено `gtest` функционалностима).



Слика 7: Блок дијаграм тока извршавања алата за мерење микро-архитектура

Табела 8: Алгоритам естимирања J

```

function CalibrateJ()
  Initialize hw_events with a CPU cycle hardware event
  Initialize perf_group with label "TAU_CALIBRATION" and hw_events
  E ← 100000
  tau_acc_cycles ← 0

  for i ← 0 to E - 1 do
    perf_group.start_measuring()
    perf_group.stop_measuring()
    tau_acc_cycles ← tau_acc_cycles + perf_group.read_group_by_index(0)
  end for

  tau_cycles ← tau_acc_cycles / E
  tau_precision ← 1.0
  calculated_j ← ⌊tau_cycles / tau_precision⌋

  g_benchmark_config.runs = calculated_j
end function

```

Табела 9: Алгоритам извршавања једног теста

```

function ExecuteBench(benchmark, hw_events, collector)
  event_n ← hw_events.size
  perf_group ← InitializePerfGroup("test", hw_events)

  // Загревање теста
  benchmark.function()

  // Извршавање теста
  for k ← 0 to g_benchmark_config.runs - 1 do
    perf_group.start_measuring()
    for m ← 0 to k - 1 do

```

```
        benchmark.function()
    end for
    perf_group.stop_measuring()
    collector.record_measurements(perf_group, k)
end for
end function

function RunWithPerf(benchmark)
    switch benchmark.kind do
        case gap, latency, throughput
            config <- g_latency_config
            event_n <- config.hw_events.size
            collector <- InitializeCollector()

            ExecuteBench(benchmark, config.hw_events, collector)

            ReportCollection(benchmark.name, config.hw_events, collector)
            ReportMetrics(benchmark.name, config.hw_events, collector,
config.derived_metrics)
            break
        case roofline:
            config <- g_latency_config
            event_n <- config.hw_events.size
            collector <- InitializeCollector()

            default:
                Error("Unsupported benchmark kind")
            end switch
    end switch
end function
```

4. Програмско решење

У даљем тексту ће бити описана програмска реализација концепта решења. Програмско решење је, као што је већ споменуто, реализовано у C++ програмском језику са помоћном скриптом за извршавање алата. Програмско решење се ослања на C претпроцесор, C++ шаблонске класе и функције као и функционалности развојног окружења gtest. Документација решења је подељена на функционалне целине описане у концепту решења: окружење за извршавање, генератор микро-архитектуралних тестова, омотач за профилисање Linux perf алатом и извршна скрипта run.sh. Ово поглавље обухвата само делове gtest развојног окружења који су коришћени, остатак није релевантан за овај алат.

4.1 Реализација окружења за извршавање тестова

У овом поглављу ће бити описане компоненте окружења за извршавање тестова које су саставни део решења. Даље излагање је подељено у два дела, главну извршну датотеку алата и помоћно заглавље за исписивање резултата.

4.1.1 Датотека `run_pipeline_benchmarks.cpp`

Главна извршна датотека `pipeline_benchmarks` окружења спаја функционалности `gtest` окружења за извршавање тестова и `perf_wrapper` омотача Linux `perf`.

Класе: `PipelineBenchmarkLatencyPerf`, `PipelineBenchmarkThroughputPerf`, `PipelineBenchmarkGapPerf`, `PipelineBenchmarkRooflinePerf`, `PerfGroup`, `perf_config`

Табела 10: Садржај датотеке `run_pipeline_benchmarks.cpp`

Врста	Члан	Опис
Конструктор	<code>TEST_P(PipelineBenchmarkLatencyPerf, Run)</code>	Инстанца класе намењена за извршавање тестова кашњења.
Класа	<code>TEST_P(PipelineBenchmarkThroughputPerf, Run)</code>	Класа намењена за извршавање тестова проточности.
Метода	<code>INSTANTIATE_TEST_SUITE_P(ScalarLatency, PipelineBenchmarkLatencyPerf, ::testing::ValuesIn(scalar_latency_benchmarks), GetNameFromBenchmark<instruction_benchmark>)</code>	Метода служи за филтрирање тестова кашњења агрегираних у <code>scalar_latency_benchmarks</code> низу. Окружење <code>gtest</code> ће пронаћи методу и покренути је.
Метода	<code>INSTANTIATE_TEST_SUITE_P(ScalarThroughput, PipelineBenchmarkThroughputPerf, ::testing::ValuesIn(scalar_throughput_benchmarks), GetNameFromBenchmark<instruction_benchmark>)</code>	Метода служи за филтрирање тестова проточности агрегираних у <code>scalar_throughput_benchmarks</code> низу. Окружење <code>gtest</code> ће пронаћи методу и

Врста	Члан	Опис
		покренути је.
Функција	<pre>template<std::size_t EventCount> void ExecuteBench(const benchmark_core &benchmark, const std::array<perf_hw_event, EventCount> &hw_events, PerfCollector<EventCount> &collector)</pre>	Функција извршавања једног микро-архитектуралног теста по броју PMU догађаја мерених.
Функција	<pre>void RunWithPerf(const benchmark_core &benchmark)</pre>	Функција која статички диференцира конфигурације по тесту.
Функција	<pre>int main(int argc, char **argv)</pre>	Улазна функција алата. Подиже gtest окружење и извршава све сакупљене тестове.

4.1.1.1 Класа PipelineBenchmarkGapPerf

Наслеђује: `private public::testing::Test, private public::testing::WithParamInterface<gap_benchmark>`

Нема чланова.

Служи да увеже микро-архитектуралне типове тестова `gap_benchmark` са gtest извршним окружењем.

4.1.1.2 Класа PipelineBenchmarkLatencyPerf

Наслеђује: `private public::testing::Test, private public::testing::WithParamInterface<instruction_benchmark>`

Нема чланова.

Служи да увеже микро-архитектуралне типове тестова `instruction_benchmark` са `gtest` извршним окружењем.

4.1.1.3 Класа `PipelineBenchmarkRooflinePerf`

Наслеђује: `private public::testing::Test`, `private public::testing::WithParamInterface<roofline_benchmark>`

Нема чланова.

Служи да увеже микро-архитектуралне типове тестова `roofline_benchmark` са `gtest` извршним окружењем.

4.1.1.4 Класа `PipelineBenchmarkThroughputPerf`

Наслеђује: `private public::testing::Test`, `private public::testing::WithParamInterface<instruction_benchmark >`

Нема чланова.

Служи да увеже микро-архитектуралне типове тестова са `gtest` извршним окружењем.

4.1.1.5 Именски простор

`anonymous_namespace{run_pipeline_benchmarks.cpp}`

Табела 11: Садржај именског простора

`anonymous_namespace{run_pipeline_benchmarks.cpp}`

Врста	Члан	Опис
Функција	<code>template<typename Benchmark> std::string GetNameFromBenchmark(const ::testing::TestParamInfo<Bench mark> &info)</code>	Шаблонска функција за добављање имена тестова употребом шаблонских класа из <code>gtest</code> оквира.
Функција	<code>unsigned int CalibrateJ()</code>	Калибрациона функција за одабир броја

Врста	Члан	Опис
		извршавања теста j.
Променљива	<code>BenchmarkConfig g_benchmark_config</code> <code>= { 0 }</code>	Глобална променљива која чува број извршавања теста j.

4.1.2 Датотека `util.cpp`

Табела 12: Садржај датотеке `util.cpp`

Врста	Члан	Опис
Функција	<code>std::size_t ParseRuns(</code> <code>std::string_view value)</code>	Парсира корисников захтев за наменски калибрациони број.
Функција	<code>void ParsePerf(</code> <code>std::string_view value)</code>	Парсира корисников захтев за контролу над Linux perf омотачем. У случају да постоји разлог да се угаси.

4.1.3 Датотека `util.h`

Класе: `BenchmarkConfig`, `PerfCollector`, `perf_hw_event`,
`derived_metric`

Табела 13: Садржај датотеке `util.hpp`

Врста	Члан	Опис
Функција	<code>template<typename T></code> <code>void ReportMetricAndPrint(</code> <code>std::string name,</code> <code>const T &value)</code>	Шаблонска функција која исписује генерички тип метрике на конзолу.
Функција	<code>void ParseBenchmarkArguments(</code> <code>int argc, char **argv,</code>	Наменско парсирање аргумената за алат. Служи

Врста	Члан	Опис
	BenchmarkConfig &benchmark_cfg)	да прикупи калибрациони број ако га је корисник експлицитно проследио.
Функција	<pre>template<std::size_t EventCount> void ReportCollection(std::string_view name, std::span< const perf_hw_event> hw_events, PerfCollector<EventCount> collection)</pre>	Шаблонска функција која служи да извести о резултатима експерименталних мерења. Испис се дешава на конзоли али и у .json датотеку у оквиру gtest окружења.
Функција	<pre>template<std::size_t EventCount, PerfEvent... Events> void ReportMetric(std::string_view benchmark_name, std::span<const perf_hw_event> hw_events, const PerfCollector<EventCount> &collector, const derived_metric< Events... > &metric)</pre>	Шаблонска функција која извештава о једној сложеној метрици Linux perf омотача. Испис се дешава на конзоли али и у .json датотеку у оквиру gtest окружења.
Функција	<pre>template<std::size_t EventCount, typename... Metrics> void ReportMetrics(std::string_view benchmark_name, std::span<const perf_hw_event> hw_events, const PerfCollector<EventCount> &collector, const std::tuple<Metrics...></pre>	Приступна шаблонска функција која позива ReportMetric ради исписа сложених метрика. Ослања се на std::apply методу која за све резултате мерења примењује одговарајућу функцију сложене

Врста	Члан	Опис
	<code>&metrics)</code>	метрике.

4.1.3.1 Структура `BenchmarkConfig`

Табела 14: Садржај структуре `BenchmarkConfig`

Врста	Члан	Опис
Поље	<code>std::size_t runs = -1</code>	Број покретања сваког теста, калибрациони број j.

4.2 Реализација претпроцесорског генератора `microbenchmark_generator.h`

Претпроцесорски генератор је, као што је у концепту решења објашњено, сачињен од великог броја макро функција. У овом поглављу ће се представити садржај `microbenchmark_generator.h` заглавља.

Почеће се са основним структурама које X-макрои користе за сакупљање микро-архитектуралних тестова.

4.2.1 Структура `benchmark_core`

Табела 15: Садржај структуре `benchmark_core`

Врста	Члан	Опис
Поље	<code>const std::string_view name</code>	Име теста.
Поље	<code>const isa_type isa</code>	Тип инструкције која се гледа.
Поље	<code>const benchmark_type kind</code>	Тип теста.
Поље	<code>const int iterations</code>	Број итерација теста.
Поље	<code>bench_fn function</code>	Тестна функција.

4.2.2 Структура `instruction_benchmark`

Наслеђује: `public benchmark_core`

Табела 16: Садржај структуре `instruction_benchmark`

Врста	Члан	Опис
Поље	<code>const std::string_view instruction</code>	Име инструкције.

4.2.3 Макрои генератора

Табела 17: Сви макрои генератора

Врста	Члан	Опис
Макро	<code>STR2(x) #x</code>	Спаја два текстуална литерала.
Макро	<code>STR(x) STR2(x)</code>	Омотач за <code>STR2(x)</code> .
Макро	<code>CAT2(a, b) a##b</code>	Спаја два макроа у трећи.
Макро	<code>CAT(a, b) CAT2(a, b)</code>	Омотач за <code>CAT2(a, b)</code> .
Макро	<code>INSN_OP3(OP, RD, RS1, RS2) STR(OP) " " STR(RD) ", " STR(RS1) ", " STR(RS2) "\n"</code>	Генерише инструкцију са три операнда.
Макро	<code>INSN_OP2(OP, RD, RS1) STR(OP) " " STR(RD) ", " STR(RS1) "\n"</code>	Генерише инструкцију са два операнда.
Макро	<code>INSN_LOAD(OP, RD, OFFSET, BASE) STR(OP) " " STR(RD) ", " STR(OFFSET) "(" STR(BASE) " \n"</code>	Генерише инструкцију за учитавања вредности из меморије у регистре.
Макро	<code>INIT_GPR(REG, VAL) "li " STR(REG) ", " STR(VAL) "\n"</code>	Помоћни макро за иницијализацију скаларних регистара.

Врста	Члан	Опис
Макро	INIT_FPD(REG, VAL) "fmv.d.x " STR(REG) ", " STR(VAL) "\n"	Иницијализација регистра покретног зареза на 32-битне вредности.
Макро	INIT_FPS(REG, VAL) "fmv.w.x " STR(REG) ", " STR(VAL) "\n"	Иницијализација регистра покретног зареза на 64-битне вредности.
Макро	ALU_OP3(OP, RD, RS, RD_VAL, RS_VAL) (ALU_OP3, OP, RD, RS, RD_VAL, RS_VAL)	Генерише Аритметичко-логичку инструкцију са три операнда.
Макро	ALU_OP2(OP, RD, RD_VAL) (ALU_OP2, OP, RD, RD_VAL)	Генерише Аритметичко-логичку инструкцију са два операнда.
Макро	FPD_OP3(OP, RD, RS) (FPD_OP3, OP, RD, RS)	Генерише инструкцију са покретним зарезом са три операнда 64-битних вредности.
Макро	FPS_OP3(OP, RD, RS) (FPS_OP3, OP, RD, RS)	Генерише инструкцију са покретним зарезом са три операнда 32-битних вредности.
Макро	GPR_LOAD(OP, RD, BASE, OFFSET, BASE_VAL) (GPR_LOAD, OP, RD, BASE, OFFSET, BASE_VAL)	Помоћни макро за учитавање у скаларне регистаре.
Макро	INIT_OF(SPEC) INIT_OF_ SPEC	Шаблонски макро који припајањем формира иницијализацију микро-архитектуралних тестова.
Макро	INIT_OF_(KIND, ...) CAT(INIT_OF_, KIND) (__VA_ARGS__)	Шаблонски макро који припајањем формира иницијализацију микро-архитектуралних тестова са

Врста	Члан	Опис
		варијабилним бројем аргумената.
Макро	BODY_OF(SPEC) BODY_OF_ SPEC	Шаблонски макро који припајањем формира тело микро-архитектуралних тестова.
Макро	BODY_OF_(KIND, ...) CAT(BODY_OF_, KIND) (__VA_ARGS__)	Шаблонски макро који припајањем формира тело микро-архитектуралних тестова са варијабилним бројем аргумената.
Макро	CLOBBERS_OF(SPEC) CLOBBERS_OF_ SPEC	Шаблонски макро који припајањем формира скуп коришћених регистара микро-архитектуралних тестова.
Макро	CLOBBERS_OF_(KIND, ...) CAT(CLOBBERS_OF_, KIND) (__VA_ARGS__)	Шаблонски макро који припајањем формира скуп коришћених регистара микро-архитектуралних тестова са варијабилним бројем аргумената.
Макро	NAME_OF(SPEC) NAME_OF_ SPEC	Шаблонски макро који припајањем формира текст.
Макро	NAME_OF_(KIND, OP, ...) STR(OP)	Шаблонски макро са варијабилним бројем аргумената који припајањем формира текст.
Макро	INIT_OF_ALU_OP3(OP, RD, RS, RD_VAL, RS_VAL) INIT_GPR(RD, RD_VAL) \	Припојен макро иницијализације АЛУ инструкције за тестне

Врста	Члан	Опис
	INIT_GPR(RS, RS_VAL)	функције.
Макро	BODY_OF_ALU_OP3(OP, RD, RS, RD_VAL, RS_VAL) INSN_OP3(OP, RD, RD, RS)	Припојен макро АЛУ инструкције тела тестне функције.
Макро	CLOBBERS_OF_ALU_OP3(OP, RD, RS, RD_VAL, RS_VAL) STR(RD), STR(RS)	Припојен макро коришћења АЛУ регистара са три операнда.
Макро	INIT_OF_ALU_OP2(OP, RD, RD_VAL) INIT_GPR(RD, RD_VAL)	Припојен макро иницијализације АЛУ операције са два операнда.
Макро	BODY_OF_ALU_OP2(OP, RD, RD_VAL) INSN_OP2(OP, RD, RD)	Припојен макро тела АЛУ операције са два операнда.
Макро	CLOBBERS_OF_ALU_OP2(OP, RD, RD_VAL) STR(RD)	Припојен макро коришћења АЛУ регистра са два операнда.
Макро	INIT_OF_FPD_OP3(OP, RD, RS) INIT_FPD(RD, zero) \ INIT_FPD(RS, zero)	Припојен макро иницијализације покретног зареза 64-бита тестних функција.
Макро	BODY_OF_FPD_OP3(OP, RD, RS) INSN_OP3(OP, RD, RD, RS)	Припојен макро тела покретног зареза 64-бита тестних функција.
Макро	CLOBBERS_OF_FPD_OP3(OP, RD, RS) STR(RD), STR(RS)	Припојен макро коришћења покретног зареза 64-бита тестних функција.
Макро	INIT_OF_FPS_OP3(OP, RD, RS) INIT_FPS(RD, zero) \ INIT_FPS(RS, zero)	Припојен макро иницијализације покретног зареза 32-бита тестних функција.

Врста	Члан	Опис
	STR(RD) ", 0(" STR(BASE) ") \n"	
Макро	CLOBBERS_OF_MEM_OP(OP, RD, BASE, MEM_VAL) STR(RD), STR(BASE)	Припојен макро коришћења регистра за рад са меморијом у тестовима.
Макро	IMM_OP2(OP, RD, IMM) (IMM_OP2, OP, RD, IMM)	Припојен макро инструкције са два операнда: регистра и константе.
Макро	INIT_OF_IMM_OP2(OP, RD, IMM) INIT_GPR(RD, 0)	Припојен макро иницијализације теста скаларног регистра на константу.
Макро	BODY_OF_IMM_OP2(OP, RD, IMM) STR(OP) " " STR(RD) ", " STR(IMM) "\n"	Припојен макро тела теста са скаларним регистром и константом.
Макро	CLOBBERS_OF_IMM_OP2(OP, RD, IMM) STR(RD)	Припојен макро коришћења регистра у тесту са константом.
Макро	RAW_OP(NAME, INIT, BODY, ...) (RAW_OP, NAME, INIT, BODY, __VA_ARGS__)	Сирова инструкција са произвољним бројем аргумената. Сирово се овде користи као литерал.
Макро	INIT_OF_RAW_OP(NAME, INIT, BODY, ...) INIT	Припојена сирова иницијализација за тест.
Макро	BODY_OF_RAW_OP(NAME, INIT, BODY, ...) ".option push\n\t" \ "option arch, +xspacemit\n\t" \ BODY	Припојено сирово тело тестова. Фиксирано на xspacemit.

Врста	Члан	Опис
	<code>"\n\t" \ ".option pop\n"</code>	
Макро	<code>CLOBBERS_OF_RAW_OP(NAME, INIT, BODY, ...) __VA_ARGS__</code>	Припојени макро коришћења сирових регистра за тестове.
Макро	<code>INIT_ADD_FADD_D_BODY INIT_REGS_D(ADD_FADD_INIT_REGS)</code>	Припојен макро иницијализације регистара АЛУ и покретног зареза.
Макро	<code>RV_LATENCY_ASM_FIXED_REGS(ITER S, INIT, BODY, ...) asm volatile (\ ".option push\n" \ ".option norvc\n" \ INIT \ ".rept " STR(ITER) "\n" \ BODY \ ".endr\n" \ ".option pop\n" \ : \ : \ : __VA_ARGS__ \)</code>	Функција генерисања садржаја тестова кашњења. Сакупља иницијализацију, тело и коришћене регистре у <code>__VA_ARGS__</code> специјалном макроу.
Макро	<code>RV_LATENCY_TEST_FUNC(FUNC_NAME , ITERS, INIT, BODY, ...) extern "C" __attribute__((noinline, aligned(64))) \ void FUNC_NAME() \ { \ RV_LATENCY_ASM_FIXED_REGS(\ ITERS, \ INIT, \ BODY, \ __VA_ARGS__ \); \ }</code>	Функција генерисања дефиниције C функције теста кашњења.

Врста	Члан	Опис
Макро	LATENCY_BENCH(FUNC_NAME, ITERS, SPEC) RV_LATENCY_TEST_FUNC(\ FUNC_NAME, \ ITERS, \ INIT_OF(SPEC), \ BODY_OF(SPEC), \ CLOBBERS_OF(SPEC), "memory" \)	Улазна функција X-макроа за генерисање тестова кашњења.
Макро	ALU_OP3_THROUGHPUT(OP, ...) (ALU_OP3_THROUGHPUT, OP, __VA_ARGS__)	Проширена функција коришћења АЛУ инструкција за тестове проточности.
Макро	VA_COUNT(...) VA_COUNT_I(__VA_OPT__(__VA_ARGS__,) \ 16, 15, 14, 13, 12, 11, 10, 9, \ 8, 7, 6, 5, 4, 3, 2, 1, 0 \)	Специјални макрои су делови C 20 стандарда и служе да прошире макро у режиму петље n пута где је n у овом случају 16.
Макро	VA_COUNT_I(_1, _2, _3, _4, _5, _6, _7, _8, _9, _10, _11, _12, _13, _14, _15, _16, COUNT, ...) COUNT	Помоћна функција проширивања неког макроа 16 пута.
Макро	PARENS ()	Празан макро. Служи за итеративно припајање макроа
Макро	EXPAND(...) EXPAND1(EXPAND1(EXPAND1(EXPAND1(_VA_ARGS__))))	Макро експанзије припајањем варијабилним макроима.
Макро	EXPAND1(...) EXPAND2(EXPAND2(EXPAND2(EXPAND2(_VA_ARGS__))))	Макро експанзије припајањем варијабилним макроима.

Врста	Члан	Опис
	2(__VA_ARGS__))))	
Макро	EXPAND2(. . .) EXPAND3(EXPAND3(EXPAND3(EXPAND3(__VA_ARGS__)))))	Макро експанзије припајањем варијабилним макроима.
Макро	EXPAND3(. . .) EXPAND4(EXPAND4(EXPAND4(EXPAND4(__VA_ARGS__)))))	Макро експанзије припајањем варијабилним макроима.
Макро	EXPAND4(. . .) __VA_ARGS__	Макро експанзије припајањем варијабилним макроима.
Макро	OPS_PER_ITER_OF(SPEC) OPS_PER_ITER_OF_ SPEC	Помоћни макро за експанзију операције по итерацији .
Макро	OPS_PER_ITER_OF_(KIND, . . .) CAT(OPS_PER_ITER_OF_, KIND) (__VA_ARGS__)	Помоћни макро за експанзију операција по итерацији са варијабилним бројем променљивих.
Макро	OPS_PER_ITER_OF_ALU_OP3_THROUGH HPU T(OP, . . .) VA_COUNT(__VA_ARGS__)	Помоћни макро са варијабилним бројем променљивих за експанзију АЛУ инструкција који се користи за тестове проточности.
Макро	FOR_EACH(MACRO, FIXED, . . .) __VA_OPT__(\ EXPAND(FOR_EACH_HELPER(\ MACRO, FIXED, __VA_ARGS__)) \)	Помоћни макро који за сваки MACRO примењује технику експанзије.
Макро	FOR_EACH_HELPER(MACRO, FIXED, . . .)	Унутрашњи помоћни макро

Врста	Члан	Опис
	<pre> FIRST, ...) MACRO(FIXED, FIRST) \ __VA_OPT__(\ FOR_EACH_AGAIN PARENS \ (MACRO, FIXED, __VA_ARGS__) \) </pre>	петље. Понавља макро n пута.
Макро	<pre> FOR_EACH_AGAIN() FOR_EACH_HELPER </pre>	Унутрашњи помоћни макро петље.
Макро	<pre> FOR_EACH_COMMA(MACRO, FIXED, ...) __VA_OPT__(\ EXPAND(\ FOR_EACH_COMMA_HELPER(MACRO, FIXED, __VA_ARGS__ \) \) \) </pre>	Унутрашњи помоћни макро петље који генерише зарезе на адекватним местима.
Макро	<pre> FOR_EACH_COMMA_HELPER(MACRO, FIXED, FIRST, ...) MACRO(FIXED FIRST) \ __VA_OPT__(\ , FOR_EACH_COMMA_AGAIN PARENS \ (MACRO, FIXED, __VA_ARGS__) \) </pre>	Унутрашњи помоћни макро петље који генерише зарезе на адекватним местима.
Макро	<pre> FOR_EACH_COMMA_AGAIN() FOR_EACH_COMMA_HELPER </pre>	Унутрашњи помоћни макро петље који генерише зарезе на адекватним местима.
Макро	<pre> OP3_CHAIN(RD, RS, RD_VAL, RS_VAL) (RD, RS, RD_VAL, RS_VAL) </pre>	Операција са три операнда која може да се уланча.
Макро	<pre> OP3_CHAIN_RD(CHAIN) OP3_CHAIN_RD_I CHAIN </pre>	Операција са три операнда која може да се уланча.
Макро	<pre> OP3_CHAIN_RD_I(RD, RS, RD_VAL, RS_VAL) RD </pre>	Операција са три операнда која може да се уланча.
Макро	<pre> OP3_CHAIN_RS(CHAIN) OP3_CHAIN_RS_I CHAIN </pre>	Операција са три операнда која може да се уланча.

Врста	Члан	Опис
Макро	OP3_CHAIN_RS_I(RD, RS, RD_VAL, RS_VAL) RS	Операција са три операнда која може да се уланча.
Макро	OP3_CHAIN_RD_VAL(CHAIN) OP3_CHAIN_RD_VAL_I CHAIN	Операција са три операнда која може да се уланча.
Макро	OP3_CHAIN_RD_VAL_I(RD, RS, RD_VAL, RS_VAL) RD_VAL	Операција са три операнда која може да се уланча.
Макро	OP3_CHAIN_RS_VAL(CHAIN) OP3_CHAIN_RS_VAL_I CHAIN	Операција са три операнда која може да се уланча.
Макро	OP3_CHAIN_RS_VAL_I(RD, RS, RD_VAL, RS_VAL) RS_VAL	Операција са три операнда која може да се уланча.
Макро	INIT_ONE_ALU_OP3_CHAIN(OP, CHAIN) INIT_GPR(OP3_CHAIN_RD(CHAIN), \ OP3_CHAIN_RD_VAL(CHAIN)) \ INIT_GPR(OP3_CHAIN_RS(CHAIN), \ OP3_CHAIN_RS_VAL(CHAIN))	Проширење ланчане иницијализације АЛУ инструкција.
Макро	BODY_ONE_ALU_OP3_CHAIN(OP, CHAIN) INSN_OP3(\ OP, \ OP3_CHAIN_RD(CHAIN), \ OP3_CHAIN_RD(CHAIN), \ OP3_CHAIN_RS(CHAIN))	Проширење ланчаног генерисања тела АЛУ инструкција.
Макро	CLOBBERS_ONE_ALU_OP3_CHAIN(OP, CHAIN) STR(OP3_CHAIN_RD(CHAIN)), \ STR(OP3_CHAIN_RS(CHAIN))	Проширење ланчаног генерисања коришћења регистра АЛУ инструкција.

Врста	Члан	Опис
Макро	INIT_OF_ALU_OP3_THROUGHPUT(OP, ...) FOR_EACH(\ INIT_ONE_ALU_OP3_CHAIN, \ OP, \ __VA_ARGS__)	Проширење ланчане иницијализације АЛУ инструкција које служи за тестове проточности са варијабилним бројем параметара.
Макро	BODY_OF_ALU_OP3_THROUGHPUT(OP, ...) FOR_EACH(\ BODY_ONE_ALU_OP3_CHAIN, \ OP, \ __VA_ARGS__)	Проширење ланчаног тела теста проточности са АЛУ инструкцијама са три операнда.
Макро	CLOBBERS_OF_ALU_OP3_THROUGHPUT(OP, ...) FOR_EACH_COMMA(\ CLOBBERS_ONE_ALU_OP3_CHAIN, \ OP, \ __VA_ARGS__)	Проширење ланчаног коришћења регистра теста проточности са АЛУ инструкцијама са три операнда.
Макро	FPD_OP3_THROUGHPUT(OP, GPR1, VAL1, GPR2, VAL2, ...) (FPD_OP3_THROUGHPUT, OP, GPR1, VAL1, GPR2, VAL2, __VA_ARGS__)	Проширење ланчаног коришћења регистра теста проточности са инструкцијама покретног зареза 64-бита са три операнда.
Макро	OPS_PER_ITER_OF_FPD_OP3_THROUGHPUT(OP, GPR1, VAL1, GPR2, VAL2, ...) VA_COUNT(__VA_ARGS__)	Проширење ланчаног коришћења регистра теста проточности са инструкцијама покретног зареза 64-бита са три операнда варијабилног броја параметара.
Макро	INIT_ONE_FPD_OP3_CHAIN(OP, CHAIN)	Проширење ланчане иницијализације са

Врста	Члан	Опис
	INIT_FPD(OP3_CHAIN_RD(CHAIN), \ OP3_CHAIN_RD_VAL(CHAIN)) \ INIT_FPD(OP3_CHAIN_RS(CHAIN), \ OP3_CHAIN_RS_VAL(CHAIN))	Инструкцијама покретног зареза 64-бита са три операнда.
Макро	BODY_ONE_FPD_OP3_CHAIN(OP, CHAIN) INSN_OP3(\ OP, \ OP3_CHAIN_RD(CHAIN), \ OP3_CHAIN_RD(CHAIN), \ OP3_CHAIN_RS(CHAIN))	Проширење ланчаног тела са инструкцијама АЛУ са три операнда.
Макро	CLOBBERS_ONE_FPD_OP3_CHAIN(OP, CHAIN) STR(OP3_CHAIN_RD(CHAIN)), \ STR(OP3_CHAIN_RS(CHAIN))	Проширење ланчаног коришћења регистра са инструкцијама покретног зареза 64-бита са три операнда.
Макро	INIT_OF_FPD_OP3_THROUGHPUT(OP, GPR1, VAL1, GPR2, VAL2, ...) \ INIT_GPR(GPR1, VAL1) \ INIT_GPR(GPR2, VAL2) \ FOR_EACH(\ INIT_ONE_FPD_OP3_CHAIN, \ OP, \ __VA_ARGS__)	Проширење ланчане иницијализације теста проточности са инструкцијама покретног зареза 64-бита и АЛУ са три операнда.
Макро	BODY_OF_FPD_OP3_THROUGHPUT(OP, GPR1, VAL1, GPR2, VAL2, ...) \ FOR_EACH(\ BODY_ONE_FPD_OP3_CHAIN, \ OP, \ __VA_ARGS__)	Проширење ланчаног тела теста проточности са инструкцијама покретног зареза 64-бита са три операнда.
Макро	CLOBBERS_OF_FPD_OP3_THROUGHPUT	Проширење ланчаног

Врста	Члан	Опис
	(OP, GPR1, VAL1, GPR2, VAL2, ...) STR(GPR1), STR(GPR2), \ FOR_EACH_COMMA(\ CLOBBERS_ONE_FPD_OP3_CHAIN, \ OP, \ __VA_ARGS__)	коришћења регистара тестова проточности са инструкцијама покретног зареза 64-бита са три операнда.
Макро	FPS_OP3_THROUGHPUT(OP, GPR1, VAL1, GPR2, VAL2, ...) (FPS_OP3_THROUGHPUT, OP, GPR1, VAL1, GPR2, VAL2, __VA_ARGS__)	Проширење инструкције покретног зареза проточности са инструкцијама покретног зареза 64-бита са три операнда.
Макро	OPS_PER_ITER_OF_FPS_OP3_THROUG HPUT (OP, GPR1, VAL1, GPR2, VAL2, ...) VA_COUNT(__VA_ARGS__)	Проширење ланчаног коришћења инструкција покретног зареза 32-бита за тестове проточности.
Макро	INIT_ONE_FPS_OP3_CHAIN(OP, CHAIN) INIT_FPS(OP3_CHAIN_RD(CHAIN), \ OP3_CHAIN_RD_VAL(CHAIN)) \ INIT_FPS(OP3_CHAIN_RS(CHAIN), \ OP3_CHAIN_RS_VAL(CHAIN))	Проширење ланчане иницијализације инструкција покретног зареза 32-бита.
Макро	BODY_ONE_FPS_OP3_CHAIN(OP, CHAIN) INSN_OP3(\ OP, \ OP3_CHAIN_RD(CHAIN), \ OP3_CHAIN_RS(CHAIN), \ OP3_CHAIN_RS_VAL(CHAIN))	Проширење ланчаног тела инструкција покретног зареза 32-бита.

Врста	Члан	Опис
	OP3_CHAIN_RD(CHAIN), \ OP3_CHAIN_RS(CHAIN))	
Макро	CLOBBERS_ONE_FPS_OP3_CHAIN(OP, CHAIN) STR(OP3_CHAIN_RD(CHAIN)), \ STR(OP3_CHAIN_RS(CHAIN))	Проширење ланчаног коришћења регистара инструкција покретног зареза 32-бита.
Макро	INIT_OF_FPS_OP3_THROUGHPUT(OP, GPR1, VAL1, GPR2, VAL2, ...) INIT_GPR(GPR1, VAL1) \ INIT_GPR(GPR2, VAL2) \ FOR_EACH(\ INIT_ONE_FPS_OP3_CHAIN, \ OP, \ __VA_ARGS__)	Проширење ланчане иницијализације инструкција покретног зареза 32-бита за тест проточности.
Макро	BODY_OF_FPS_OP3_THROUGHPUT(OP, GPR1, VAL1, GPR2, VAL2, ...) FOR_EACH(\ BODY_ONE_FPS_OP3_CHAIN, \ OP, \ __VA_ARGS__)	Проширење ланчаног тела теста инструкција покретног зареза 32-бита за тестове проточности.
Макро	CLOBBERS_OF_FPS_OP3_THROUGHPUT (OP, GPR1, VAL1, GPR2, VAL2, ...) STR(GPR1), STR(GPR2), \ FOR_EACH_COMMA(\ CLOBBERS_ONE_FPS_OP3_CHAIN, \ OP, \ __VA_ARGS__)	Проширење ланчаног коришћења регистра инструкција покретног зареза 32-бита за тестове проточности.
Макро	RV_REPEATED_ASM_FIXED_REGS(ITE RS, INIT, BODY, ...)	Функција генерисања садржаја тестова проточности. Сакупља

Врста	Члан	Опис
	<pre>asm volatile (\ ".option push\n" \ ".option norvc\n" \ INIT \ ".rept " STR(ITERS) "\n" \ BODY \ ".endr\n" \ ".option pop\n" \ : \ : \ : __VA_ARGS__ \)</pre>	иницијализацију, тело и коришћене регистре у <code>__VA_ARGS__</code> специјалном макроу.
Макро	<pre>RV_THROUGHPUT_TEST_FUNC(FUNC_NAME, ITERS, INIT, BODY, ...) extern "C" __attribute__((noinline, aligned(64))) void FUNC_NAME() \ { RV_REPEATED_ASM_FIXED_REGS(\ ITERS, \ INIT, \ BODY, \ __VA_ARGS__ \); \ }</pre>	Функција генерисања дефиниције C функције теста проточности.
Макро	<pre>THROUGHPUT_BENCH(FUNC_NAME, ITERS, SPEC) RV_THROUGHPUT_TEST_FUNC(\ FUNC_NAME, \ ITERS, \ INIT_OF(SPEC), \ BODY_OF(SPEC), \ CLOBBERS_OF(SPEC), \ "memory" \)</pre>	Улазна функција X-макроа за генерисање тестова проточности.
Макро	<pre>INIT_VTYPE_E64M1 "vsetvli t5,</pre>	Сирова инцијализација

Врста	Члан	Опис
	x0, e64, m1, ta, ma\n" \ "slli t5, t5, 3\n"	векторског копроцесора.
Макро	VEC_OP(OP, VD) (OP, VD)	Основна векторска инструкција
Макро	VEC_OP_OP(CH) VEC_OP_OP_I CH	Основна векторска инструкција
Макро	VEC_OP_OP_I(OP, VD) OP	Основна векторска инструкција
Макро	VEC_OP_VD(CH) VEC_OP_VD_I CH	Основна векторска инструкција
Макро	VEC_OP_VD_I(OP, VD) VD	Основна векторска инструкција
Макро	BODY_ONE_VEC_COMPUTE(FIXED, CHAIN) BODY_ONE_VEC_COMPUTE_I(CHAIN)	Тело обраде једне инструкције векторског теста.
Макро	BODY_ONE_VEC_COMPUTE_I(CHAIN) STR(VEC_OP_OP(CHAIN)) " " STR(VEC_OP_VD(CHAIN)) ", " \ STR(VEC_OP_VD(CHAIN)) ", " STR(VEC_OP_VD(CHAIN)) "\n"	Тело ланчане обраде векторским инструкцијама.
Макро	VEC_COMPUTE_THROUGHPUT(...) (VEC_COMPUTE_THROUGHPUT, __VA_ARGS__)	Помоћни макро теста проточности векторских инструкција.
Макро	OPS_PER_ITER_OF_VEC_COMPUTE_TH ROUGHPUT(...) VA_COUNT(__VA_ARGS__)	Помоћни макро теста проточности векторских инструкција који служи за варијабилно ланчање макроа.
Макро	INIT_OF_VEC_COMPUTE_THROUGHPUT (...) INIT_VTYPE_E64M1 \ "li t4, 0x3ff5555555555555\n" \ FOR_EACH(INIT_ONE_VEC_BCAST,	Иницијализација теста проточности векторских инструкција базирана на варијабилним макроима.

Врста	Члан	Опис
	0, __VA_ARGS__)	
Макро	INIT_ONE_VEC_BCAST(FIXED, CHAIN) "vmv.v.x STR(VEC_OP_VD(CHAIN)) ", t4\n"	Иницијализација теста проточности векторских инструкција.
Макро	BODY_OF_VEC_COMPUTE_THROUGH HPUT(...) FOR_EACH(BODY_ONE_VEC_COMPUTE, 0, __VA_ARGS__)	Тело теста проточности векторских инструкција базирана на варијабилним макроима.
Макро	CLOBBERS_OF_VEC_COMPUTE_THROUGH HPUT (...) "t4", "t5", FOR_EACH_COMMA(CLOBBERS_ONE_VEC C_RE G, 0, __VA_ARGS__)	Коришћени регистри теста проточности векторских инструкција варијабилним макроима.
Макро	CLOBBERS_ONE_VEC_REG(FIXED, CHAIN) STR(VEC_OP_VD(CHAIN))	Коришћен регистар теста проточности векторских инструкција.
Макро	BODY_ONE_VEC_LOAD(BASE, CHAIN) BODY_ONE_VEC_LOAD_I(BASE, CHAIN)	Помоћни макро који у телу генерише једно учитавање у векторски регистар.
Макро	BODY_ONE_VEC_LOAD_I(BASE, CHAIN) STR(VEC_OP_OP(CHAIN)) " " STR(VEC_OP_VD(CHAIN)) ", (" STR(BASE) ")\n" \ "add " STR(BASE) ", " STR(BASE) ", t5\n"	Помоћни макро који генерише једно векторско учитавање у телу теста.
Макро	VEC_STREAM_THROUGHPUT(BUFSYM,	Помоћни макро за тестове

Врста	Члан	Опис
	<code>MASK, ...)</code> <code>(VEC_STREAM_THROUGHPUT,</code> <code>BUFSYM, MASK, __VA_ARGS__)</code>	проточности учитавања из меморије у векторски копроцесор.
Макро	<code>OPS_PER_ITER_OF_VEC_STREAM_THR</code> <code>OUGH</code> <code>PUT(BUFSYM, MASK, ...)</code> <code>VA_COUNT(__VA_ARGS__)</code>	Помоћни макро ланчаних операција проточности учитавања из меморије у векторски копроцесор.
Макро	<code>INIT_OF_VEC_STREAM_THROUGHPUT(</code> <code>BUFSYM, MASK, ...)</code> <code>INIT_VTYPE_E64M1 \ "la t2, "</code> <code>STR(BUFSYM) "\n" \ "mv t0,</code> <code>t2\n" \</code> <code>"li t3, " STR(MASK) "\n"</code>	Иницијализација теста проточности учитавања из меморије у векторски копроцесор.
Макро	<code>BODY_OF_VEC_STREAM_THROUGHPUT(</code> <code>BUFS</code> <code>YM, MASK, ...)</code> <code>FOR_EACH(BODY_ONE_VEC_LOAD,</code> <code>t0,</code> <code>__VA_ARGS__) \ "sub t1, t0,</code> <code>t2\n"</code> <code>\ "and t1, t1, t3\n" \ "add t0,</code> <code>t2,</code> <code>t1\n"</code>	Тело теста проточности учитавања из меморије у векторски копроцесор.
Макро	<code>CLOBBERS_OF_VEC_STREAM_THROUGH</code> <code>PUT(</code> <code>BUFSYM, MASK, ...)</code> <code>"t0", "t1", "t2", "t3", "t5",</code> <code>FOR_EACH_COMMA(CLOBBERS_ONE_VE</code> <code>C_RE</code> <code>G, 0, __VA_ARGS__)</code>	Коришћени регистри теста проточности векторског копроцесора када учитава из меморије.

Врста	Члан	Опис
Макро	ITER_TWO_POW_EIGHT 256	Број итерација за .pert асемблерску директиву.
Макро	ITER_TWO_POW_NINE 512	Број итерација за .pert асемблерску директиву.
Макро	ITER_TWO_POW_TEN 1024	Број итерација за .pert асемблерску директиву.
Макро	ITER_TWO_POW_THIRTEEN 8192	Број итерација за .pert асемблерску директиву.
Макро	ITER_TWO_POW_NINETEEN 524288	Број итерација за .pert асемблерску директиву.
Макро	SCALAR_LATENCY_BENCHMARK(FUNC_NAME, ISA, SPEC, ITERS) LATENCY_BENCH(FUNC_NAME, ITERS, SPEC)	Улазни макро за X-макро теста кашњења који генерише тестове кашњења перформанси процесора.
Макро	SCALAR_LATENCY_BENCHMARK(FUNC_NAME, ISA, SPEC, ITERS) { \ STR(FUNC_NAME), \ ISA, \ benchmark_type::latency, \ ITERS, \ FUNC_NAME, \ NAME_OF(SPEC) \ },	X-макро теста кашњења.
Макро	SCALAR_THROUGHPUT_BENCHMARK(FUNC_NAME, ISA, SPEC, ITERS) THROUGHPUT_BENCH(FUNC_NAME, ITERS, SPEC)	Улазни макро за X-макро теста проточности који генерише тестове проточности перформанси процесора.

Врста	Члан	Опис
Макро	SCALAR_THROUGHPUT_BENCHMARK(FUNC_NAME, ISA, SPEC, ITERS) { \STR(FUNC_NAME), \ ISA, \ benchmark_type::throughput, \ ITERS, \ FUNC_NAME, \ NAME_OF(SPEC) \ },	X-макро теста проточности.
Макро	L1_WORKING_SET 16383	Величина L1 скривене меморије.
Макро	L2_WORKING_SET 262144	Величина L2 скривене меморије.
Макро	DRAM_WORKING_SET 33554432	Величина DRAM скривене меморије.
Псеудоним типа	using bench_fn = void(*)()	Помоћни псеудоним.
Набрајање	enum class isa_type	Вредности: RV64I, RV64M, RV64A, RV64F, RV64D, RV64C, RV64V, RV64B.
Набрајање	enum class benchmark_type	Вредности: gap, latency, throughput, roofline.
Промењлива	instruction_benchmark scalar_latency_benchmarks[]	Низ који агрегира тестове кашњења.
Промењлива	instruction_benchmark scalar_throughput_benchmarks[]	Низ који агрегира тестове проточности.
Промењлива	unsigned char g_mem_buffer[64 *1024 *1024]	Низ коришћен за меморијске тестове.

4.3 Реализација омотача око Linux perf алата

У наставку ће бити описане софтверске компоненте омотача Linux perf алата.

4.3.1 Датотека perf_config.h

Реализација структуре конфигурација у времену превођења се налази у овој датотеци.

4.3.1.1 Структура perf_config

```
template<typename Events, typename Metrics>
struct perf_config
```

Табела 18: Садржај структуре perf_config

Врста	Члан	Опис
Поље	Events hw_events	Хардверске основне метрике.
Поље	Metrics derived_metrics	Сложене метрике.

4.3.2 Датотека perf_collector.h

4.3.2.1 Класа PerfCollector

```
template<std::size_t N> PerfCollector
```

Табела 19: Садржај класе PerfCollector

Врста	Члан	Опис
Конструктор	PerfCollector()	Основни конструктор.
Метода	void record_measurements(PerfGroup<N> &perf_group, int k)	Метода за снимање основних метрика. Користи се методама PerfGroup методама за добављање мерења системским

Врста	Члан	Опис
		позивима.
Поље	<pre>std::array<uint64_t, N> data = [] { std::array<uint64_t, N> result{}; result.fill(UINT64_MAX); return result; }()</pre>	Низ који складишти резултате мерења. Почетна вредност му је највећи неозначени цео број зато што алгоритам одабирања увек узима најмању аритметичку средину.

4.3.3 Датотека perf_group.h

Класе: PerfWrapper::read_value, PerfWrapper::group_read_data, PerfWrapper::perf_hw_event, PerfWrapper::anonymous_namespace{perf_group.h}::PerfCounter, PerfWrapper::PerfGroup

Табела 20: Садржај датотеке perf_group.h

Врста	Члан	Опис
Макро	MAX_EVENTS 16	Максимални број основних метрика дозвољених по групи метрика.

4.3.3.1 Структура perf_hw_event

Табела 21: Садржај структуре perf_hw_event

Врста	Члан	Опис
Поље	std::string_view name	Име основне метрике.
Поље	uint32_t type	Тип основне метрике. Обично софтверска или хардверска.
Поље	uint32_t counter	Јединствени бројач асоциран са PMU.

4.3.3.2 Класа PerfGroup

```
template<std::size_t N> PerfGroup
```

Табела 22: Садржај класе PerfGroup

Врста	Члан	Опис
Псеудоним типа	<code>using heuristic_fn = void(*) (std::span< uint64_t > res, const double scale, const uint64_t k, std::span<const read_value> data)</code>	Псеудоним хеуристичке функције која служи за одабирање минималне аритметичке средине микро-архитектуралних мерења.
Конструктор	<code>PerfGroup(std::string_view name, const std::array< perf_hw_event, N> &hw_events)</code>	Конструктор perf групе који креира основне бројаче по метрикама.
Метода	<code>void start_measuring()</code>	Метода користи вођу групе, прву метрику, да започне мерења.
Метода	<code>void stop_measuring()</code>	Метода користи вођу групе, прву метрику да оконча мерења.
Метода	<code>uint64_t read_group_by_index(int index)</code>	Помоћна метода за читање вредности са само једног бројача.
Метода	<code>void read_group(std::span< uint64_t > res, const uint64_t k, heuristic_fn hfn)</code>	Помоћна метода за читање вредности свих основних метрика са могућношћу одабирања само корисних на основу хеуристичке функције.
Приватна променљива	<code>std::string_view group_name</code>	Име групе.
Приватна	<code>std::array<std::optional<</code>	Бројачи алоцирани за групу.

Врста	Члан	Опис
променљива	<code>PerfCounter>, N></code> <code>perf_counters</code>	<code>std::optional</code> овде служи да нагласи да ће статички број елемената низа зависити од путање у коду која ће бити извршена (тип теста, утиче на број метрика).
Приватна променљива	<code>int leader_fd</code>	Дескриптор датотеке отворене за <code>perf</code> догађај вође групе.

4.3.3.3 Класа

`PerfWrapper::anonymous_namespace{perf_group.h}::PerfCounter`

Табела 23: Садржај класе `PerfCounter`

Врста	Члан	Опис
Конструктор	<code>PerfCounter()</code> <code>noexcept=default</code>	Празан конструктор.
Конструктор	<code>PerfCounter(</code> <code>const uint32_t &type,</code> <code>const uint32_t &config,</code> <code>const int &group_fd=-1)</code>	Конструктор који користи <code>perf_event_open</code> API и у зависности од <code>group_fd</code> дескриптора конфигурише бројач вође или обичан бројач.
Деструктор	<code>~PerfCounter()</code>	Служи да затвори дескриптор датотеке <code>fd</code> .
Метода	<code>std::size_t start()</code>	Започиње мерење једне метрике.
Метода	<code>std::size_t stop()</code>	Зауставља мерење једне метрике.
Метода	<code>uint64_t read_value()</code>	Чита вредност једне метрике.
Метода	<code>int get_fd() const noexcept</code>	Враћа дескриптор датотеке.
Приватна метода	<code>long perf_event_open(</code> <code>struct perf_event_attr</code>	Омотачка метода која користи системски позив <code>perf_event_open</code> да добава дескриптор датотеке <code>fd</code> .

Врста	Члан	Опис
	<code>&hw_event,</code> <code>pid_t pid,</code> <code>int cpu,</code> <code>int group_fd,</code> <code>unsigned long flags)</code>	намењен за читање вредности процесорских метрика са PMU.
Приватно поље	<code>int fd</code>	Дескриптор преко кога се добављају подаци о основним метрикама са PMU.

4.3.3.4 Структура `PerfWrapper::group_read_data`

Табела 24: Садржај структуре `PerfWrapper::group_read_data`

Врста	Члан	Опис
Поље	<code>uint64_t nr</code>	Број чланова групе
Поље	<code>uint64_t time_enabled</code>	Време које је метрика провела спремна.
Поље	<code>uint64_t time_running</code>	Време које је метрика провела активна.
Поље	<code>struct read_value</code> <code>values[MAX_EVENTS]</code>	Очитане вредности по члану групе.

4.3.3.5 Структура `PerfWrapper::read_value`

Табела 25: Садржај структуре `PerfWrapper::read_value`

Врста	Члан	Опис
Поље	<code>uint64_t value</code>	Вредност основне метрике датог мерења.
Поље	<code>uint64_t id</code>	Идентификатор метрике измерене.

4.3.4 Именски простор PerfWrapper

Класе: `group_read_data`, `perf_config`, `perf_hw_event`, `PerfCollector`, `PerfGroup`, `read_value`

Табела 26: Садржај именског простора PerfWrapper

Врста	Члан	Опис
Набрајање	<code>enum class PerfEvent : std::size_t</code>	Вредности: Cycles, Instructions.
Функција	<code>void keep_min(std::span<uint64_t> res, const double scale, uint64_t k, std::span<const read_value> data)</code>	Дефиниција хеуристичке функције за задржавање аритметичког минимума мерења.
Променљива	<code>auto g_latency_config = perf_config {.hw_events = std::array{ perf_hw_event{"cycles", PERF_TYPE_HARDWARE, PERF_COUNT_HW_CPU_CYCLES, perf_hw_event{"instr", PERF_TYPE_HARDWARE, PERF_COUNT_HW_INSTRUCTIONS } }, .derived_metrics = std::tuple{ derived_metric< PerfEvent::Instructions, PerfEvent::Cycles>{ .name = "IPC",</code>	Статичка декларација глобалне променљиве која садржи у себи конфигурације основних и сложених метрика за извршавање тестова кашњења.

Врста	Члан	Опис
	<code>.mfncalculator = MetricCalculations::CalculateRatio } } }</code>	
Променљива	<code>auto g_throughput_config = perf_config { .hw_events = std::array{ perf_hw_event{"cycles", PERF_TYPE_HARDWARE, PERF_COUNT_HW_CPU_CYCLES, perf_hw_event{"instr", PERF_TYPE_HARDWARE, PERF_COUNT_HW_INSTRUCTIONS } }, }, .derived_metrics = std::tuple{ derived_metric< PerfEvent::Instructions, PerfEvent::Cycles>{ .name = "IPC", .mfncalculator = MetricCalculations::CalculateRatio } } }</code>	Статичка декларација глобалне променљиве која садржи у себи конфигурације основних и сложених метрика за извршавање тестова протоčnosti.

4.4 Реализација рачунице сложених метрика у заглављу `metric_calculations.h`

У оквиру овог заглавља је реализован механизам по коме се рачунају сложене метрике. Следећа поглавља садрже програмску реализацију рачунице

сложених метрика реализовану путем наменских структура и шаблонских метода.

4.4.1 Структура `derived_metric`

```
template<PerfEvent... Events> derived_metric
```

Табела 27: Садржај структуре `derived_metric`

Врста	Члан	Опис
Метода	<code>double Calculate(std::span<const MetricValue> collected_values) const</code>	Рачуна сложену изведену метрику на основу прикупљених основних метрика. Користи <code>MetricValue</code> као индекс по коме приступа сакупљеним вредностима.
Поље	<code>std::string_view name</code>	Име сложене метрике.
Поље	<code>metric_fn mfn</code>	Функција која рачуна сложену метрику.
Статичко поље	<code>static std::array< std::size_t, sizeof... (Events)> indexes { ToIndex(Events)... }</code>	Низ основних метрика које се користе за дату сложену метрику. Свака комбинација основних метрика дефинише нову структуру сложене метрике на основу шаблона.

Врста	Члан	Опис
Набрајање	<code>enum class PerfEvent</code>	Сви основни догађаји из којих се сложене метрике могу извести.

4.4.2 Именски простор MetricCalculations

Табела 28: Садржај именског простора MetricCalculations

Врста	Члан	Опис
Псеудоним типа	<code>using metric_fn = std::optional<double>(* (std::span< const std::uint64_t >)</code>	Псеудоним на основу кога се дефинишу све сложене метрике. Улаз је низ произвољне величине сакупљених резултата мерења.
Псеудоним типа	<code>using MetricValue = std::uint64_t</code>	Мапирање стандардног типа метрика.
Функција	<code>static std::optional<double> CalculateRatio(std::span<const std::uint64_t> values)</code>	Сложена метрика рачунања односа.
Функција	<code>static std::optional<double> CalculatePercentage(std::span< const std::uint64_t> values)</code>	Сложена метрика рачунања процента једне основне метрике у другој.
Функција	<code>static std::optional<double> CalculateSum(std::span< const std::uint64_t > values)</code>	Сложена метрика рачунања суме основних метрика.
Функција	<code>std::size_t ToIndex(PerfEvent event) noexcept</code>	Помоћна функција која трансформише набрајање у индекс. Користи се за

Врста	Члан	Опис
		добављање вредности основних метрика.

4.5 Реализација скрипте за извршавање алата

У оквиру овог решења једноставна `bash` скрипта је написана како би олакшала конфигурисање и извршавање алата `run.sh`. Следеће табеле представљају програмско решење скрипте за извршавање алата:

Табела 29: Сва поља скрипте `run.sh`

Поље	Подразумевана вредност	Сврха
<code>SCRIPT_DIR</code>	<code>"\$(cd "\$(dirname "\${BASH_SOURCE[0]}")" && pwd)"</code>	Путања до директоријума са скриптама.
<code>CPU</code>	<code>3</code>	Фиксирано језгро на коме ће се алат покренути.
<code>RUNS</code>	<code>0</code>	Број покретања. Наменски калибрациони број. Када је <code>0</code> , игнорисан је.
<code>SHOULD_PERF</code>	<code>0</code>	Апликативно профилисање <code>perf</code> алатом ће бити упаљено ако је вредност <code>1</code> .
<code>OUTPUT_DIR</code>	<code>"perf_results/\$(date +%Y%m%d_%H%M%S)"</code>	Излазна датотека у коју ће се исписати апликативне метрике.
<code>BINARY</code>	<code>"</code>	Неопходно је проследити ову променљиву скрипти како би се изабрао програм

		за покретање.
ASLR	<code>\$(cat /proc/sys/kernel/randomize_va_space)</code>	Чита и уписује вредност у датотеку за паљење или гашење ASLR.
TESTS	<code>""</code>	Садржи филтер текст за gtest.

Следећа табела садржи функције које run.sh поседује:

Табела 30: Све функције скрипте run.sh

Име	Опис
<code>configure_aslr()</code>	На основу уноса корисника пали или гаси ASLR.
<code>prepare_filter_list()</code>	Омогућује филтер листи да користи специјалне карактере као што је * у Unix системима да конфигурише TESTS променљиву.
<code>run_perf_stat()</code>	Извршава апликативни perf stat на захтев корисника над програмом BINARY.
<code>config_perf_level()</code>	Конфигурише ниво инструментације Linux perf, апликативни или на нивоу изворног кода.

5. Евалуација

Евалуација је обављена на развојној плочи BananaPi BPI F3 која поседује SpacemiT-X60 процесор [31]. Сви микро-архитектурални тестови су покренути над истим језгром процесора на једној развојној плочи. Испитивање настоји да покаже:

- Резултате микро-архитектуралних тестова векторских инструкција као поређење rvv-bench резултата [29] и резултата постигнутих у овом раду везаних за кашњење.
- Проточност и кашњење скаларних инструкција се пореди са ценама модела распоређивања SpacemiT-X60 процесора у LLVM колекцији програмских преводиоца [30].

Важно је напоменути како су срачунате метрике кашњења и проточности у овој евалуацији. Алат `pipeline_benchmark` представљен у овом раду рачуна кашњење и проточност као однос измереног броја циклуса подељеног са измереним бројем успешно извршених инструкција (енгл. *cycles per instruction CPI*). LLVM не рачуна проточност инструкција директно у свој модел распоређивања, већ посматра коју врсту јединице инструкција користи и колико дуго; тако да је проточност однос броја јединица које инструкција користи (обично једна) подељено са бројем јединица доступних на процесору за извршавање дате инструкције. Примера ради, ако постоје две аритметичко-

логичке јединице у језгру процесора, проточност би била једнака два за сваку инструкцију којој је једна јединица довољна да се изврши.

Такође, представљен је само репрезентативни скуп инструкција по сваком типу инструкције. Ово је учињено по узору на програмске преводиоце, као што ће се видети у примеру скаларних резултата LLVM групише цене инструкција по одређеним класама зато што, архитектурално говорећи, неке инструкције су врло сродне па се зато може одабрати једна инструкција као репрезентативни члан групе.

У дискусији се приказују табеле и графици који приказују стопу подударана јавно доступних информација о микро-архитектури SparceMiT-X60 са експерименталним резултатима овог рада. Под стопом се подразумева корелација Кендраловим коефицијентом τ -b, срења процентуална грешка MAPE и геометријска средња вредност GMR са којом се добија смер MAPE. Теоријске основе о метрикама евалуације се налазе у поглављу 2.2.7.

Следеће две табеле представљају поређење мерења јавно доступних резултата микро-архитектуралних цена SparceMiT-X60 са резултатима експерименталних мерења добијених у овом раду. У табели 31 и 32 свака ћелија у себи садржи две вредности, прва вредност је јавно доступна информација по инструкцији, а друга експериментални резултати добијени у овом раду. Пошто би табеле са мерењима биле идентичне по структури садржај је здружен у ћелијама ради веће прегледности.

Табела 31 представља вредности кашњења векторских инструкција у свим могућим конфигурацијама SEW i LMUL које су могуће за VLEN od 256 бита. Више детаља о томе како се конфигурише векторски RISC-V копроцесор се налазе у теоријским основама, поглавље 2.2.3.

Табела 32 у себи садржи поређења модела цена инструкција за SparceMiT-X60 процесор уграђен у LLVM наспрам експерименталних резултата добијених у овом раду. Због великог броја инструкција које процесор садржи исте су груписане тако да је за сваки ред одабрана једна инструкција која је репрезентативан пример, а инструкције у тој групи имају исту цену по LLVM моделу распоређивања. Име групе по LLVM моделу распоређивања се налази у

колони: LLVM модел цена. Јединица је име у LLVM интерној репрезентацији за део језгра који извршава дати тип инструкција. Последње две колоне представљају кашњење и проточност по LLVM моделу цена за процесор наспрам експерименталних резултата у овом раду, респективно.

Табела 31: Поређење цена векторских инструкција `gcv-bench` наспрам `pipeline_benchmarks`

Инструкција	SEW	mf8	mf4	mf2	m1	m2	m4	m8
<code>vaadd.vv</code>	e8	1.00 / 3.93	1.00 / 3.93	1.00 / 3.93	1.00 / 3.93	2.00 / 3.93	4.00 / 4.91	8.00 / 7.83
	e16		1.00 / 3.92	1.00 / 3.92	1.00 / 3.93	2.00 / 3.93	4.00 / 4.90	8.00 / 7.82
	e32			1.00 / 3.92	1.00 / 3.93	2.00 / 3.93	4.00 / 4.90	8.00 / 7.82
	e64				1.00 / 3.91	2.00 / 3.91	4.00 / 4.88	8.00 / 7.78
<code>vaaddu.vv</code>	e8	1.00 / 3.93	1.00 / 3.93	1.00 / 3.93	1.00 / 3.93	2.00 / 3.93	4.00 / 4.91	8.00 / 7.83
	e16		1.00 / 3.92	1.00 / 3.92	1.00 / 3.93	2.00 / 3.93	4.00 / 4.90	8.00 / 7.82
	e32			1.00 / 3.92	1.00 / 3.93	2.00 / 3.93	4.00 / 4.90	8.00 / 7.82
	e64				1.00 / 3.91	2.00 / 3.91	4.00 / 4.88	8.00 / 7.78
<code>vadd.vv</code>	e8	1.00 / 3.93	1.00 / 3.93	1.00 / 3.93	1.00 / 3.93	2.00 / 3.93	4.00 / 4.91	8.00 / 7.83
	e16		1.00 / 3.92	1.00 / 3.92	1.00 / 3.93	2.00 / 3.93	4.00 / 4.90	8.00 / 7.82
	e32			1.00 / 3.92	1.00 / 3.93	2.00 / 3.93	4.00 / 4.90	8.00 / 7.82
	e64				1.00 / 3.91	2.00 / 3.91	4.00 / 4.88	8.00 / 7.78
<code>vand.vv</code>	e8	1.00 / 3.93	1.00 / 3.93	1.00 / 3.93	2.00 / 3.93	4.00 / 3.93	8.00 / 7.82	16.00 / 15.54
	e16		1.00 / 3.92	1.00 / 3.92	2.00 / 3.93	4.00 / 3.93	8.00 / 7.82	16.00 / 15.52
	e32			1.00 / 3.92	2.00 / 3.93	4.00 / 3.93	8.00 / 7.82	16.00 / 15.54
	e64				2.00 / 3.91	4.00 / 3.91	8.00 / 7.78	16.00 / 15.46
<code>vasub.vv</code>	e8	1.00 / 3.93	1.00 / 3.93	1.00 / 3.93	1.00 / 3.93	2.00 / 3.93	4.00 / 4.91	8.00 / 7.83
	e16		1.00 / 3.92	1.00 / 3.92	1.00 / 3.93	2.00 / 3.93	4.00 / 4.90	8.00 / 7.82
	e32			1.00 / 3.92	1.00 / 3.93	2.00 / 3.93	4.00 / 4.90	8.00 / 7.82
	e64				1.00 / 3.91	2.00 / 3.91	4.00 / 4.88	8.00 / 7.78
<code>vasubu.vv</code>	e8	1.00 / 3.93	1.00 / 3.93	1.00 / 3.93	1.00 / 3.93	2.00 / 3.93	4.00 / 4.91	8.00 / 7.83
	e16		1.00 / 3.92	1.00 / 3.92	1.00 / 3.93	2.00 / 3.93	4.00 / 4.90	8.00 / 7.82
	e32			1.00 / 3.92	1.00 / 3.93	2.00 / 3.93	4.00 / 4.90	8.00 / 7.82
	e64				1.00 / 3.91	2.00 / 3.91	4.00 / 4.88	8.00 / 7.78
<code>vdiv.vv</code>	e8	12.23 / 11.68	13.33 / 11.68	13.68 / 11.68	27.47 / 21.25	55.54 / 41.91	110.32 / 81.52	219.73 / 151.92
	e16		12.05 / 11.67	12.57 / 11.67	25.00 / 21.23	50.07 / 41.87	100.28 / 80.87	199.94 /

Инструкција	SEW	mf8	mf4	mf2	m1	m2	m4	m8
								153.46
	e32			11.80 / 11.67	23.58 / 21.25	47.37 / 41.87	95.20 / 81.30	190.94 / 152.04
	e64				22.41 / 21.12	44.79 / 41.64	89.44 / 80.51	178.53 / 151.79
vdivu.vv	e8	11.86 / 10.72	12.43 / 10.72	13.39 / 10.72	26.72 / 19.36	53.11 / 38.17	107.07 / 74.11	214.05 / 139.16
	e16		11.27 / 10.71	11.69 / 10.71	23.49 / 19.32	46.92 / 38.13	93.69 / 74.05	188.94 / 139.65
	e32			11.11 / 10.71	22.13 / 19.34	44.25 / 38.13	88.28 / 74.12	178.47 / 139.29
	e64				20.54 / 19.22	41.32 / 37.89	82.56 / 73.98	164.86 / 138.82
vfabs.v ¹	e16		1.00 / 3.92	1.00 / 3.92	2.00 / 3.93	4.00 / 3.93	8.00 / 7.82	16.00 / 15.53
	e32			1.00 / 3.92	2.00 / 3.93	4.00 / 3.93	8.00 / 7.82	16.00 / 15.53
	e64				2.00 / 3.91	4.00 / 3.91	8.00 / 7.78	16.00 / 15.44
vfadd.vv	e16		1.00 / 3.92	1.00 / 3.92	1.00 / 3.93	2.00 / 3.93	4.00 / 4.90	8.00 / 7.82
	e32			1.00 / 3.92	1.00 / 3.93	2.00 / 3.93	4.00 / 4.90	8.00 / 7.82
	e64				1.00 / 3.91	2.00 / 3.91	4.00 / 4.88	8.00 / 7.78
vfclass.v	e16		1.00 / 3.92	1.00 / 3.92	2.00 / 3.93	4.00 / 3.93	8.00 / 7.82	16.00 / 15.52
	e32			1.00 / 3.92	2.00 / 3.93	4.00 / 3.93	8.00 / 7.82	16.00 / 15.54
	e64				2.00 / 3.91	4.00 / 3.91	8.00 / 7.78	16.00 / 15.44
vfdiv.vv	e16		15.07 / 13.60	15.06 / 13.60	30.06 / 25.02	60.37 / 49.11	121.06 / 94.38	241.22 / 178.63
	e32			19.05 / 19.33	38.43 / 36.26	76.54 / 70.67	153.10 / 133.00	307.05 / 241.84
	e64				40.01 / 37.89	80.05 / 73.79	160.17 / 139.28	320.34 / 250.11
vfмасс.vv	e16		1.00 / 4.90	1.00 / 4.90	1.00 / 4.90	2.00 / 4.90	4.00 / 4.91	8.00 / 7.83
	e32			1.00 / 4.90	1.00 / 4.90	2.00 / 4.90	4.00 / 4.91	8.00 / 7.83
	e64				1.00 / 5.84	2.00 / 5.84	4.00 / 6.82	8.00 / 7.79
vfмadd.vv	e16		1.00 / 4.90	1.00 / 4.90	1.00 / 4.90	2.00 / 4.90	4.00 / 4.91	8.00 / 7.82
	e32			1.00 / 4.90	1.00 / 4.90	2.00 / 4.90	4.00 / 4.91	8.00 / 7.83
	e64				1.00 / 5.84	2.00 / 5.84	4.00 / 6.82	8.00 / 7.79
vfmax.vv	e16		1.00 / 3.92	1.00 / 3.92	1.00 / 3.93	2.00 / 3.93	4.00 / 4.91	8.00 / 7.82
	e32			1.00 / 3.92	1.00 / 3.93	2.00 / 3.93	4.00 / 4.90	8.00 / 7.82
	e64				1.00 / 3.91	2.00 / 3.91	4.00 / 4.88	8.00 / 7.79
vfmin.vv	e16		1.00 / 3.92	1.00 / 3.93	1.00 / 3.93	2.00 / 3.93	4.00 / 4.90	8.00 / 7.82

Инструкција	SEW	mf8	mf4	mf2	m1	m2	m4	m8
	e32			1.00 / 3.92	1.00 / 3.93	2.00 / 3.93	4.00 / 4.90	8.00 / 7.82
	e64				1.00 / 3.91	2.00 / 3.91	4.00 / 4.88	8.00 / 7.79
vfmsac.vv	e16		1.00 / 4.90	1.00 / 4.90	1.00 / 4.90	2.00 / 4.90	4.00 / 4.91	8.00 / 7.83
	e32			1.00 / 4.90	1.00 / 4.90	2.00 / 4.90	4.00 / 4.91	8.00 / 7.83
	e64				1.00 / 5.84	2.00 / 5.84	4.00 / 6.82	8.00 / 7.79
vfmsub.vv	e16		1.00 / 4.90	1.00 / 4.90	1.00 / 4.90	2.00 / 4.90	4.00 / 4.91	8.00 / 7.83
	e32			1.00 / 4.90	1.00 / 4.90	2.00 / 4.90	4.00 / 4.91	8.00 / 7.83
	e64				1.00 / 5.84	2.00 / 5.84	4.00 / 6.82	8.00 / 7.79
vfmul.vv	e16		1.00 / 3.92	1.00 / 3.92	1.00 / 3.93	2.00 / 3.93	4.00 / 4.90	8.00 / 7.82
	e32			1.00 / 3.92	1.00 / 3.93	2.00 / 3.93	4.00 / 4.90	8.00 / 7.82
	e64				1.36 / 4.88	2.78 / 4.88	5.46 / 4.88	10.79 / 7.79
vfrec7.v	e16		1.00 / 3.92	1.00 / 3.92	2.00 / 3.93	4.00 / 3.93	8.00 / 7.82	16.00 / 15.54
	e32			1.00 / 3.92	2.00 / 3.93	4.00 / 3.93	8.00 / 7.82	16.00 / 15.53
	e64				2.00 / 3.91	4.00 / 3.91	8.00 / 7.78	16.00 / 15.44
vfrsqrt7.v	e16		1.00 / 3.92	1.00 / 3.92	2.00 / 3.93	4.00 / 3.93	8.00 / 7.82	16.00 / 15.52
	e32			1.00 / 3.92	2.00 / 3.93	4.00 / 3.93	8.00 / 7.82	16.00 / 15.52
	e64				2.00 / 3.91	4.00 / 3.91	8.00 / 7.78	16.00 / 15.44
vfsgnj.vv	e16		1.00 / 3.92	1.00 / 3.92	2.00 / 3.93	4.00 / 3.93	8.00 / 7.82	16.00 / 15.53
	e32			1.00 / 3.93	2.00 / 3.93	4.00 / 3.93	8.00 / 7.82	16.00 / 15.53
	e64				2.00 / 3.91	4.00 / 3.91	8.00 / 7.78	16.00 / 15.44
vfsgnjn.vv	e16		1.00 / 3.92	1.00 / 3.92	2.00 / 3.93	4.00 / 3.93	8.00 / 7.82	16.00 / 15.54
	e32			1.00 / 3.92	2.00 / 3.93	4.00 / 3.93	8.00 / 7.82	16.00 / 15.54
	e64				2.00 / 3.91	4.00 / 3.91	8.00 / 7.78	16.00 / 15.44
vfsgnjx.vv	e16		1.00 / 3.92	1.00 / 3.92	2.00 / 3.93	4.00 / 3.93	8.00 / 7.82	16.00 / 15.53
	e32			1.00 / 3.92	2.00 / 3.93	4.00 / 3.93	8.00 / 7.82	16.00 / 15.52
	e64				2.00 / 3.91	4.00 / 3.91	8.00 / 7.78	16.00 / 15.44
vfsqrt.v	e16		13.57 / 13.60	13.98 / 13.60	28.17 / 25.02	56.62 / 49.25	110.80 / 94.55	223.70 / 176.85
	e32			16.88 / 19.33	33.73 / 36.29	68.51 / 70.47	136.41 / 133.10	273.49 / 239.41
	e64				33.77 / 37.89	65.22 / 73.85	138.13 / 138.81	263.65 / 249.35
vfsub.vv	e16		1.00 / 3.92	1.00 / 3.92	1.00 / 3.93	2.00 / 3.93	4.00 / 4.90	8.00 / 7.82
	e32			1.00 / 3.92	1.00 / 3.93	2.00 / 3.93	4.00 / 4.90	8.00 / 7.82
	e64				1.00 / 4.88	2.00 / 4.88	4.00 / 4.88	8.00 / 7.79

Инструкција	SEW	mf8	mf4	mf2	m1	m2	m4	m8
vor.vv	e8	1.00 / 3.93	1.00 / 3.93	1.00 / 3.93	2.00 / 3.93	4.00 / 3.93	8.00 / 7.82	16.00 / 15.54
	e16		1.00 / 3.92	1.00 / 3.92	2.00 / 3.93	4.00 / 3.93	8.00 / 7.82	16.00 / 15.53
	e32			1.00 / 3.92	2.00 / 3.93	4.00 / 3.93	8.00 / 7.82	16.00 / 15.54
	e64				2.00 / 3.91	4.00 / 3.91	8.00 / 7.78	16.00 / 15.44
vrem.vv	e8	13.26 / 12.64	13.82 / 12.64	14.80 / 12.64	29.58 / 23.16	59.07 / 45.46	117.88 / 87.67	237.69 / 165.03
	e16		13.03 / 12.64	13.41 / 12.64	27.03 / 23.14	53.94 / 45.46	108.00 / 88.06	216.15 / 163.26
	e32			12.89 / 12.63	25.75 / 23.14	51.51 / 45.46	102.79 / 88.22	205.08 / 164.08
	e64				24.31 / 22.99	48.81 / 45.26	97.76 / 87.99	195.15 / 163.26
vremu.vv	e8	12.73 / 11.68	13.73 / 11.68	14.28 / 11.68	28.67 / 21.25	56.78 / 41.83	115.18 / 81.08	229.98 / 152.95
	e16		12.29 / 11.67	12.77 / 11.67	25.38 / 21.23	50.62 / 41.83	101.54 / 80.94	204.40 / 151.80
	e32			12.09 / 11.67	23.94 / 21.25	48.00 / 41.83	96.94 / 81.53	191.86 / 152.42
	e64				22.63 / 21.12	45.14 / 41.53	89.82 / 80.66	180.34 / 151.29
vxor.vv	e8	1.00 / 3.93	1.00 / 3.93	1.00 / 3.93	2.00 / 3.93	4.00 / 3.93	8.00 / 7.82	16.00 / 15.54
	e16		1.00 / 3.92	1.00 / 3.92	2.00 / 3.93	4.00 / 3.93	8.00 / 7.82	16.00 / 15.52
	e32			1.00 / 3.92	2.00 / 3.93	4.00 / 3.93	8.00 / 7.82	16.00 / 15.52
	e64				2.00 / 3.91	4.00 / 3.91	8.00 / 7.78	16.00 / 15.46

Табела 32: Поређење цена скаларних инструкција LLVM распоређивача
наспрам pipeline_benchmarks

Тип	Инструкција	LLVM модел цена	Јединица	Кашњење	Проточност
and	and, or, xor	WriteIALU	IEU	1 / 0.516675	0.5 / 0.50437
sll	sll, srl, sra	WriteShiftReg	IEU	1 / 1.01	0.5 / 0.50437
slt	slt, sltu	WriteIALU	IEU	1 / 1.00864	0.5 / 0.50437
mul	mul, mulh[s][u]	WriteIMul	IEU	6 / 4.98144	0.5 / 2.99
mul	mulw	WriteIMul32	IEU	3 / 2.99	0.5 / 1.00194
clmul	clmul, clmulh, clmulr	WriteCLMUL	IEU	2 / 1.99951	0.5 / 0.504208
divw	divw, divuw, remw, remuw	WriteIDiv32/IRem32	IEUA	12 / 3.99017	12 / 2.9866

Тип	Инструкција	LLVM модел цена	Јединица	Кашњење	Проточност
div	div, divu, rem, remu	WriteIDiv/IRem	IEUA	20 / 3.9898	20 / 2.99
sh1add	sh1add, sh2add, sh3add	WriteSHXADD	IEU	2 / 2.00158	0.5 / 0.504208
rotate	rol, ror, rori	WriteRotateReg/Imm	IEU	1 / 1.0118	0.5 / 0.504208
min	min	WriteIMinMax	IEU	1 / 1.01351	0.5 / 0.50522
minu	minu	WriteIMinMax	IEU	1 / 1.0118	0.5 / 0.504208
maxu	maxu	WriteIMinMax	IEU	1 / 1.01375	0.5 / 0.504208
max	max	WriteIMinMax	IEU	1 / 1.01338	0.5 / 0.504208
zicond	czero.eqz, czero.nez	WriteIALU	IEU	1 / 1.01363	0.5 / 0.504208
fadd_s	fadd.s, fsub.s	WriteFAdd32	FP	4 / 3.98749	1 / 0.670657
fadd_d	fadd.d, fsub.d	WriteFAdd64	FP	5 / 4.97998	1 / 0.67098
fmul_s	fmul.s	WriteFMul32	FP	4 / 3.98689	1 / 0.670657
fmul_d	fmul.d	WriteFMul64	FP	5 / 4.9801	1 / 0.83632
fdiv_s	fdiv.s	WriteFDiv32	FP	15 / 9.91781	15 / 8.91089
fdiv_d	fdiv.d	WriteFDiv64	FP	22 / 9.91865	22 / 8.91612
load	lb ... ld	WriteLDB ... WriteLDD	LS	4 / 2.00063	0.5 / 1.00194
fload	flh, flw, fld	WriteFLD16/32/64	LS	4 / 1.25666	0.5 / 1.25044
store	sb ... sd	WriteSTB ... WriteSTD	LS	4 / 1.00961	0.5 / 1.00178
fstore	fsh, fsw, fsd	WriteFST16/32/64	LS	4 / 1.50341	0.5 / 1.49726
clz	clz, clzw	WriteCLZ	IEU	1 / 1.00961	0.5 / 0.504213
ctz	ctz, ctzw	WriteCTZ	IEU	1 / 1.00961	0.5 / 0.504213
cpop	cpop, cpopw	WriteCPOP	IEU	2 / 1.99891	0.5 / 0.504213
move	mv (addi rd, rs, 0)	WriteIALU	IEU	1 / 1.00973	0.5 / 0.504208
li	li (addi / lui)	WriteIALU	IEU	1 / 1.23954	0.5 / 0.504217
auipc	auipc	WriteIALU	IEU	1 / 1.51271	0.5 / 1.50081
fmadd_s	fmadd.s, fmsub.s, fnm*.s	WriteFMA32	FP	5 / 4.97877	1 / 1.00194
fmadd_d	fmadd.d, fmsub.d, fnm*.d	WriteFMA64	FP	6 / 5.96861	1 / 1.00226
mtc	fmv.w.x / fmv.d.x	WriteFMovI32ToF32/ I64ToF64	IEU	4 / 3.9864	0.5 / 1.00162
mfc	fmv.x.w / fmv.x.d	WriteFMovF32ToI32/ F64ToI64	IEU	(6 / 4) / (5.97357 / -)	0.5 / 1.9979

Тип	Инструкција	LLVM модел цена	Јединица	Кашњење	Проточност
fsqrt_s	fsqrt.s	WriteFSqrt32	FP	15 / 9.92285	15 / 8.92347
fsqrt_d	fsqrt.d	WriteFSqrt64	FP	22 / 9.92285	22 / 8.92044
fcvt_i2f	fcvt.{s,d}.{w,l}[u]	WriteFCvtI*ToF*	IEU	4 / 3.98713	0.5 / 1.00146
fcvt_f2i	fcvt.{w,l}[u].{s,d}	WriteFCvtF*ToI*	IEU	6 / 5.97417	0.5 / 1.9979
fcvt	fcvt.s.d, fcvt.d.s	WriteFCvtF*ToF*	FP	4 / 3.98713	1 / 1.00194
fmove	fmv.s / fmv.d (fsgnj)	WriteFSGNJ32/64	FP	(4 / 5) / (3.98701 / -)	1 / 1.00194
fcmp	feq, flt, fle (.s/.d)	WriteFCmp32/64	FP	6 / 5.97248	1 / 1.99725

5.1 Дискусија резултата

Добијене метрике евалуације, уведене у теоријским основама, су примењене на вредности из табела 31 и 32 па су приказане у табели 33. Инструкције су додатно груписане у класе чија су имена дата у колони поређења, број груписаних инструкција у једну класу је дат бројем N, а остале колоне по метрикама евалуације су примењене формуле описане у поглављу 2.2.7.

Табела 33: Резултат по групи инструкција t-b, MAPE, GMR, метрика евалуације

Поређење	N	Кендалов t-b	MAPE	GMR
Векторске	534	0.80	106 %	1.59
целобројне ALU	176	0.71	134 %	1.95
целобројно дељење/остатак	88	0.91	15 %	0.84
FP једноставне	180	0.73	106 %	1.69
FP FMA	60	0.77	203 %	2.45
FP дељење/корен	30	0.94	11 %	0.92
Скаларне, кашњење	42	0.79	46 %	0.80
Скаларне, проточност	42	0.50	53 %	1.11

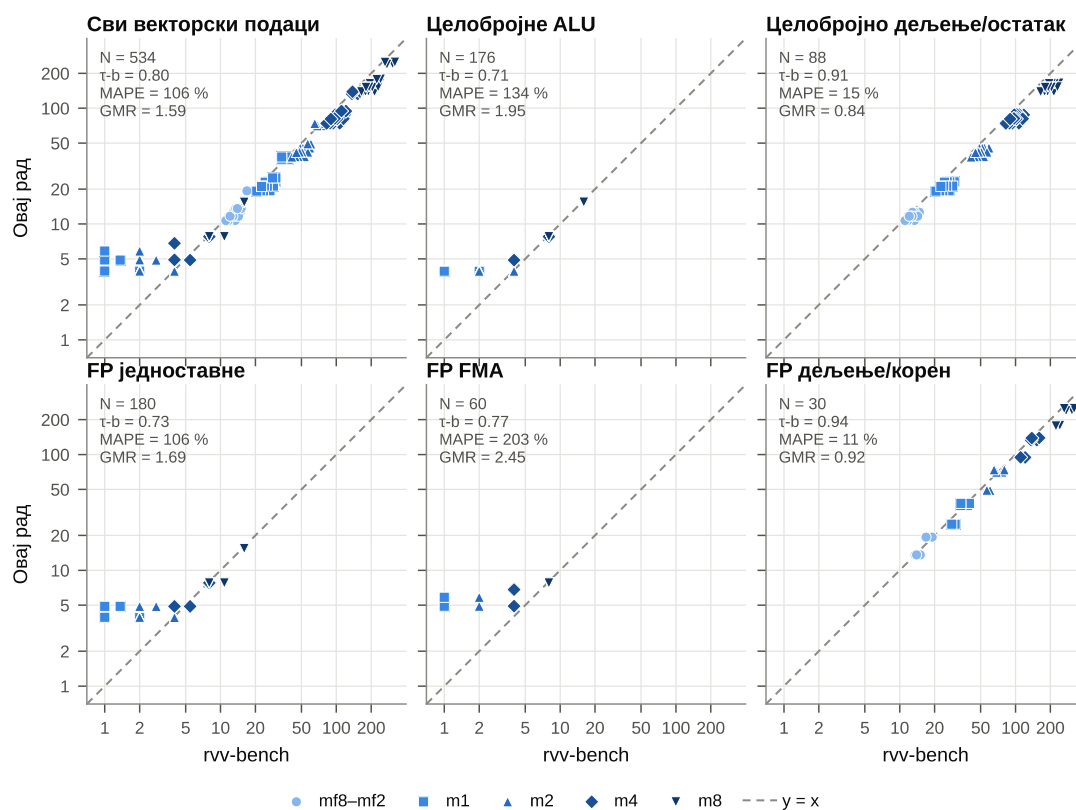
Како би се лакше интерпретирали резултати, у наставку су представљене слике за дискусију резултата. Сlike у наставку су направљене спајањем података из табела 31 и 32 са резултатима примене метрика евалуације из табеле 33.

Слика 8 представља поређења векторских резултата из табеле 31 на логаритамском графику. Вредности на x оси су добијене алатом `gvv-bench`, док су вредности на y оси експериментални резултати овог рада. Обе осе су у циклусима по инструкцији - CPI. Вредности које су на испрекиданој прави $x = y$ су идентичне што је идеално. Различити облици по типу вредности представљају LMUL конфигурације за инструкције, јер се из табеле 31 јасно види да избор LMUL значајно утиче на кашњење векторских инструкција. Свака група векторских инструкција из табеле 33 има свој график, док су на првом графику све групе векторских инструкција представљене заједно.

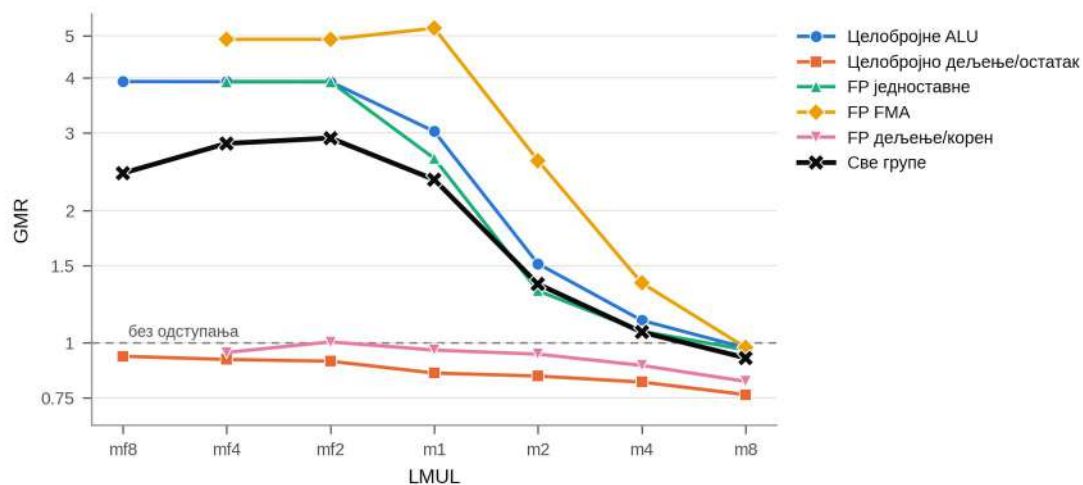
Слика 9 представља исте векторске податке али по смеру одступања срачунатом коришћењем GMR метрике на логаритамском графику. Вредности представљене су по посматраним групама из табеле 33. Када је права ближе вредности $GMR = 1$ то означава да за ту групу векторских инструкција постоји велико поклапање између јавно доступних и експерименталних података. Када су вредности испод 1 одступају тако да су вредности измерене овим радом мање него јавно доступни подаци `gvv-bench`. Одступање изнад 1 означава да су овим радом измерене вредности за дату групу инструкција веће него вредности добијене коришћењем `gvv-bench`. Као и на слици 8, јасно се види утицај на кашњење конфигурације LMUL параметара на векторској јединици.

На слици 10 права на хоризонталној оси показује величину одступања између кашњења (прва половина слике) и проточности (друга половина слике) LLVM модела цена инструкција наспрам вредности експериментално добијене алатом представљеним у овом раду. Вертикална оса представља репрезенте групу инструкција из табеле 32. На врху обе половине слике се налазе резултати евалуационих метрика за кашњење и проточност респективно.

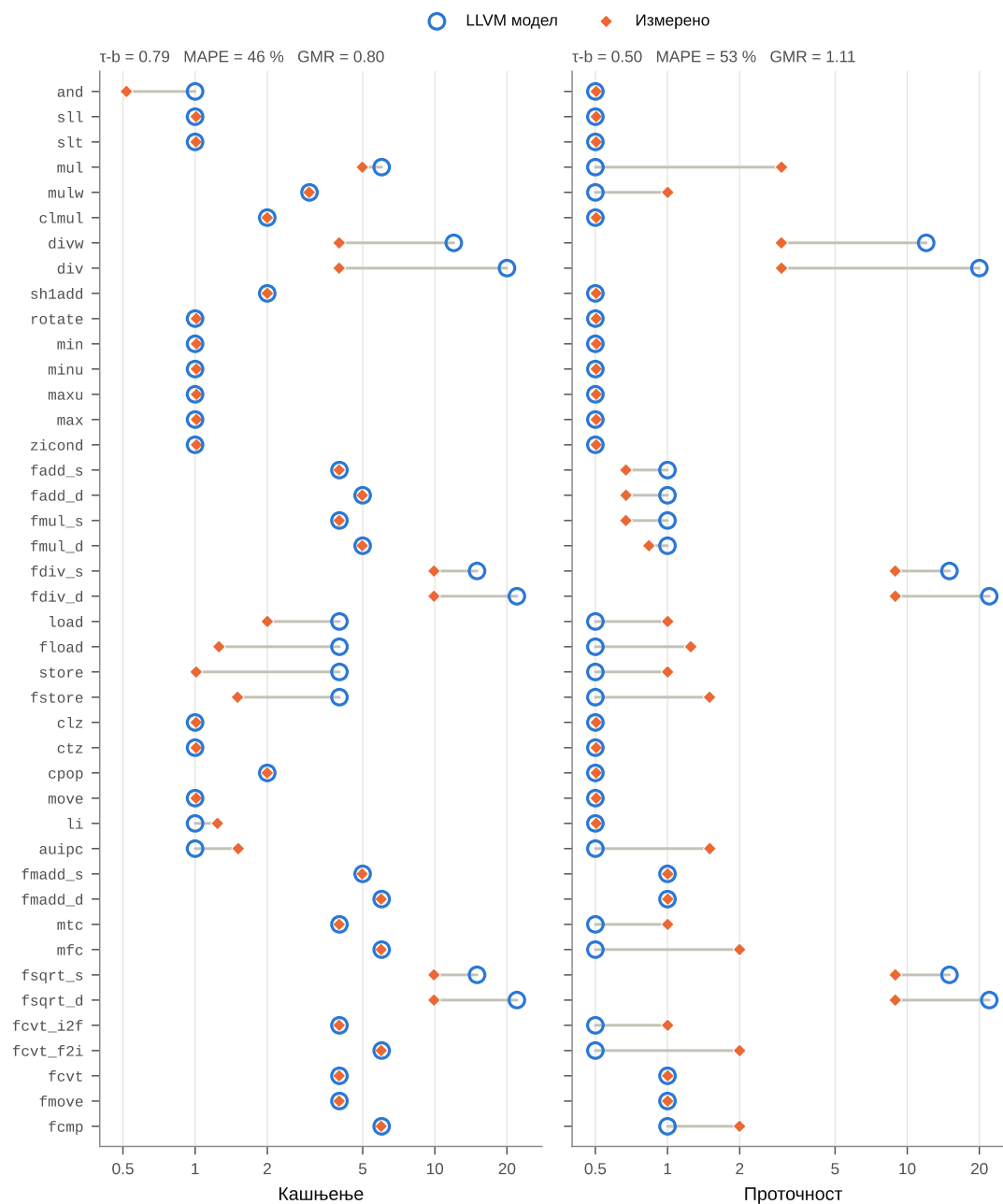
Идеалан случај је када се ромб и круг налазе у истој тачки јер то означава поклапање у вредностима LLVM модела распоређивача инструкција и експерименталних мерења овог рада по посматраној инструкцији.



Слика 8: Овај рад у поређењу са rvv-bench за сваку векторску инструкцију.



Слика 9: Геометријска средина односа овај рад / rvv-bench по вредности LMUL, по групама инструкција и укупно.



Слика 10: Дијаграмим упарених тачака LLVM модела и измерених вредности за сваку класу скаларних инструкција; лево: кашњење, десно: проточност.

Даља интерпретација се заснива на посматрању резултата метрика евалуације из табеле 33 у контексту слика 8, 9 и 10.

5.1.1 Дискусија векторских резултата

Када се посматрају резултати евалуације векторских инструкција, може се видети велики степен корелације Кендраловог коефицијента, просечно 0.80. Ово значи да је ранг инструкција између скупа вредности `rvv-bench` и експерименталних резултата овог рада врло сличан. У пракси, ово значи да ће разлика у перформансама програма преведеног програмским преводиоцем који поседује вредности једног или другог модела кашњења бити само у специфичним сценаријима где се ранг инструкција које су у програму између два модела разликује. На основу MAPE од 106% види се да постоји значајно одступање у релативним вредностима по индивидуалним инструкцијама. GMR од 1.59 означава да су ова одступања последица тога да су вредности добијене експериментално у овом раду значајно веће по инструкцији.

Дубљи увид у укупна одступања векторских инструкција примећеним у табели 33 се може видети на сликама 8 и 9. За мале вредности LMUL инструкције у групама: целобројне ALU, једноставне инструкције са покретним зарезом (енгл. floating point) FP и множење са акумулацијом покретног зареза (engl. floating point multiply accumulate) FMA приметно је значајно одступање. Највероватније је да за ове групе векторских инструкција постоји драстична разлика у поствци тестних функција наспрам `rvv-bench`.

У случају група векторских инструкција множења и дељења је супротан случај, добијене су мање вредности независно од промене у конфигурацији LMUL. Дељење и множење су специјалан случај за моделовање јер ће на кашњење више утицати улазне вредности у инструкцију него поставка тестног случаја.

Делом се одступања у резултатима могу описати недовољним поравнањем између тестова који служе за естимацију кашњења инструкција између `rvv-bench` и тестова имплементираних у овом раду.

5.1.2 Дискусија резултата скаларног кашњења и проточности

Резултати скаларног кашњења су приказани табелом 33 и сликом 10. За кашњење је Кендралов коефицијент корелације значајно бољи него за проточност, 0.79 наспрам 0.50. У случају кашњења на слици 10 се види да су одступања углавном последица мањих експерименталних резултата по инструкцији него што су забележени у LLVM моделу кашњења за распоређивач. Овим се боље очувава ранг инструкција по Кендраловом коефицијенту него за проточност и самим тиме ће разлика у моделима мање утицати на само распоређивање инструкција у машинском коду. Проточност има доста лошији Кендралов коефицијент. На слици 10 се види и зашто: за инструкције читавања и писања у меморију је у овом раду измерена значајно већа вредност за проточност него што је дата у LLVM моделу, док су вредности инструкција множења, дељења и кореновања значајно мање. Комбинација ових одступања нарушава сличност у ранговима експерименталних мерења овог рада и изведеног LLVM модела проточности.

Док су за векторске инструкције одступања по Кендраловом коефицијенту корелације била доста мања, а по MAPE и GMR значајно већа, у случају скаларних инструкција је прича обрнута. Вредности за кашњења измерена у овом раду су просечно мало мања са GMR 0.89 и са мањом грешком од него у случају векторских инструкција MAPE 46%. Вредности за проточност измерене у овом раду су мало веће са GMR 1.11 и грешком од 53% у поређењу са LLVM вредностима. Ови резултати наговештавају да су тестови врло слични и да у резултатима има доста подударња.

Слично као са векторским тестовима постоји велики степен одступања у инструкцијама које имају варијабилно кашњење у зависности од вредности у

регистрима: дељење, множење, квадратни корен и сл. Извесно је да би се побољшањем у вредностима које се користе у тестним случајевима ово одступање отклонило.

6. Закључак

У овом раду је представљен један начин да се процене вредности инструкција микро-архитектура RISC-V. Експерименти су извршени на SrasemiT-X60 процесору који поседује велики број скаларних проширења али и векторско проширење RISC-V ISA али је отворено питање које су тачне вредности цена инструкција. Резултати показују велики степен подударања са јавно доступним вредностима из сродних радова.

Алат представљен у овом раду поседује сложен систем макроа који генерише микро-архитектуралне тестове који ефективно моделују: кашњење и проточност перформанси процесора. Како би прикупила мерења ослања се на инструментацију PMU регистара Linux perf системским позивима у изворном коду [24]. Алгоритам одабирања експерименталних мерења настоји да емпиријски минимизује грешку услед непредвидивости извршавања програма у окружењу са оперативним системом [10]. Динамичко филтрирање великог броја тестова зарад селективног покретања је реализовано интеграцијом googletest развојног окружења у пројекат [22].

Предности оваквог решења су стабилност у резултатима мерења. Способност генерисања тестова базирана на претпроцесорским макроима омогућује брзо додавање великог броја сличних тестова. Филтрирање по тесту помаже код селективног покретања тестова што је за велики број микро-архитектуралних тестова које овај алат поседује неопходно. Ослањање на Linux perf за добављање информација о раду процесора отвара могућности за

профилисање рада разних аспеката процеса због великог броја догађаја које подржава. Избор смештања великог дела функционалности алата у време превођења гарантује високе перформансе због могућности коришћења оптимизујућих програмских преводиоца.

Смештање великог дела функционалности у време превођења је уједно и мана јер иницијално превођење алата у машински код може да потраје. Срећом, стаке је окружење које подржава инкрементално превођење тако да једном када су сви тестни случајеви преведени, на даље ће се инкрементално укључивати у превођење. Иако генерисање тестова великим бројем угњеждених макроа омогућује врло брзо укључивање стотине нових тестова у окружење, врло је сложено за одржавање и проширивање. У програмском решењу се налази огроман број макроа који су потенцијално дупликати и валидно је рећи да је само мали подскуп ових макроа неопходан за успешно генерисање тестова.

Метрике које су узете за евалуацију резултата су преузете из сродних радова [32, 33, 34], са проширењем увођења GMR у дискусији овог рада. Предност GMR је та што даје повратну информацију о смеру грешке, то јест, да ли су експериментална мерења просечно већа или мања од скупа вредности са којима се пореде. Зато се, за разлику од претходних радова, резултати у овом раду представљају као комбинација поређења мерења и примене Кендраловог кофицијента τ -b, MAPE и GMR метрика.

Евалуација је спроведена на нивоу инструкција али је немогуће рећи које вредности су ваљаније када се пореде резултати само на нивоу инструкција микро-архитектуре. Један од начина како би се одабрале најбоље вредности за цене инструкција би био имплементирање модела цена микро-архитектуре за распоређивач инструкција GCC или LLVM колекције програмских преводиоца и са системском програмском подршком која користи нови модел цена спровести мерења перформанси процесора SPECINT 2017 [36] скупом тестова. Поређења између два модела распоређивања по перформансама алгоритама из SPECINT 2017 би поуздано издвојило ваљаније цене по инструкцијама.

Током евалуације овог рада примећено је да увођење различитог броја основних метрика у `perf` групе утиче на калибрациони број j . Са већим бројем основних метрика које требају да се прибележе по тесту у једној групи расте и калибрациони број j . Ово откриће је у складу са мултиплексирањем које `perf`

распоређивач мора да уради због ограниченог броја PMU регистара [25]. Добијање робусних мерења за велики калибрациони број драстично успорава извршавање теста. Како би се овај проблем превазишло, један могући начин је генерисати велике тестове који ће бити за ред величине већи по броју инструкција од калибрационог броја j .

Дубље истраживање порекла одступања за мале вредности груписања LMUL код векторских инструкција у поређењу `rvv-bench` и експерименталних резултата у овом раду је тема даљег рада и усавршавања генератора микро-архитектуралних тестова.

У овом раду су представљене само две врсте микро-архитектуралних тестова. Даљи рад се може заснивати на додавању нових као што је тест процепа који меша две или више група инструкција, као композицију, рецимо АБА. Овакав тест може показати ефекат мешања инструкција на различитим јединицама обраде процесора на загушења на улазу и излазу из проточне структуре. Такође, сродни рад А. Баташева показује употребу PMU догађаја у анализи крива перформанси процесора [13]. Увођење оваквог теста и евалуирање резултата на сличан начин као у овом раду је један пут за даљи развој алата.

7. Литература

- [1] IRVINE, M. M. "Early Digital Computers at Bell Telephone Laboratories." *IEEE Annals of the History of Computing*, vol. 23, no. 3, 2001, pp. 22–42. ISSN 1058-6180.
- [2] ECCLES, W. H.; JORDAN, F. W. "A Trigger Relay Utilising Three-Electrode Thermionic Vacuum Tubes." *The Radio Review*, vol. 1, no. 3, 1919, pp. 143–146.
- [3] Goldstine, H. H.; Goldstine, Adele . (1946). The Electronic Numerical Integrator and Computer (ENIAC). *Mathematical Tables and Other Aids to Computation*, 2(15), 97–.doi:10.2307/2002620
- [4] AMDAHL, G. M.; BLAAUW, G. A.; BROOKS, F. P. "Architecture of the IBM System/360." *IBM Journal of Research and Development*, vol. 8, no. 2, 1964, pp. 87–101.
- [5] Patterson, D. A.; Ditzel, D. R. (1980). The Case for the Reduced Instruction Set Computer. *ACM SIGARCH Computer Architecture News*, 8(6), 25–33. [doi:10.1145/641914.641917](https://doi.org/10.1145/641914.641917)
- [6] Hennessy, John L. and Patterson, David A. (2019). A new golden age for computer architecture. *Commun. ACM*, jan 48–60. <https://doi.org/10.1145/3282307>
- [7] Chen, T., & Patterson, D. A. (2016). RISC-V geneology (Technical Report No. UCB/EECS-2016-6). Electrical Engineering and Computer Sciences, University of California, Berkeley.
- [8] K. Asanović and D. A. Patterson, "Instruction sets should be free: The case for RISC-V," Dept. Elect. Eng. Comput. Sci., Univ. California, Berkeley, Berkeley, CA, USA, Tech. Rep. UCB/EECS-2014-146, Aug. 2014. [Online]. Доступно: <https://www2.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-146.html>

- [9] Y. Lee, A. Waterman, R. Avizienis, H. Cook, C. Sun, V. Stojanović, and K. Asanović, "A 45nm 1.3GHz 16.7 double-precision GFLOPS/W RISC-V processor with vector accelerators," in *ESSCIRC 2014 — 40th European Solid State Circuits Conference (ESSCIRC)*, Venice Lido, Italy, Sep. 2014, pp. 199–202.
- [10] J. Chen, J. Revels, and A. Edelman, "Robust Benchmarking in Noisy Environments," in *Proc. 20th IEEE High Performance Extreme Computing Conference (HPEC)*, 2016. 10.48550/arXiv.1608.04295.
- [11] D. Stojkovic and M. Popovic, "A Way to Identify Potential Functions for Vectorization for RISC-V," in *2025 12th International Conference on Electrical, Electronic and Computing Engineering (IcETRAN)*, 2025, pp. 1-6.
- [12] Јавно доступан извештај (приступљен у септембру 2026.): https://camel-cdr.github.io/rvv-bench-results/spacemit_x60/index.html
- [13] A. Batashev, "Dissecting RISC-V Performance: Practical PMU Profiling and Hardware-Agnostic Roofline Analysis on Emerging Platforms," 2025. 10.48550/arXiv.2507.22451.
- [14] Andreas Abel and Jan Reineke. 2019. uops.info: Characterizing Latency, Throughput, and Port Usage of Instructions on Intel Microarchitectures. In *2019 Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)*, April 13–17, 2019.
- [15] Babbage, RISC-V - Part 1: Origins and Architecture, The Chip Letter 2023. [Online] Доступно: <https://thechipletter.substack.com/p/risc-v-part-1-origins-and-architecture> (приступљено 17. 09. 2026.)
- [16] Berkeley Architecture Research: "3.2. Rocket Core", Chipyard [Online]: Доступно: <https://chipyard.readthedocs.io/en/stable/Generators/Rocket.html> (приступљено 17. 09. 2026.)
- [17] R. Espasa, M. Valero, and J. E. Smith, "Vector architectures: Past, present and future," *Proc. 12th Int. Conf. Supercomputing (ICS '98)*, Melbourne, Australia, 1998, pp. 425–432, doi: 10.1145/277830.277935.
- [18] Banana Pi, "SpacemiT K1 Datasheet," *BPI-F3 Documentation*. [Online]. Доступно: https://docs.banana-pi.org/en/BPI-F3/SpacemiT_K1_datasheet (приступљено 17. 09. 2026.)
- [19] RISC-V International, "Sscofpmf" Extension for Count Overflow and Mode-Based Filtering, Version 1.0, RISC-V Privileged Architecture Specification, Version 20260120. Доступно: <https://docs.riscv.org/reference/isa/v20260120/priv/sscofpmf.html> (приступљено 17. 09. 2026.)
- [20] RISC-V International, Performance Monitoring Unit Extension (EID #0x504D55 "PMU"), RISC-V Supervisor Binary Interface (SBI) Specification, Version 3.0.

- Доступно: <https://docs.riscv.org/reference/sbi/v3.0/ext-pmu.html> (приступљено 17. 09. 2026).
- [21] camel-cdr, *rvv-bench: A collection of RISC-V Vector (RVV) benchmarks to help developers write portably performant RVV code*, GitHub repository. Доступно: <https://github.com/camel-cdr/rvv-bench> (приступљено 17. 09. 2026).
- [22] Google, "Testing Reference," GoogleTest Documentation. Доступно: <https://google.github.io/googletest/reference/testing.html> (приступљено 17. 09. 2026).
- [23] R. Meyers, "The New C: X Macros," *C/C++ Users Journal*, May 2001. Доступно: <https://jacobfilipp.com/DrDobbs/articles/CUJ/2001/0105/meyers/meyers.htm> (приступљено 17. 09. 2026).
- [24] The Linux man-pages project, "perf_event_open(2) — Linux manual page." Доступно: https://man7.org/linux/man-pages/man2/perf_event_open.2.html (приступљено 17. 09. 2026).
- [25] B. Gregg, *Systems Performance: Enterprise and the Cloud*, 2nd ed. Boston, MA, USA: Addison-Wesley, 2021.
- [26] Google, "Advanced GoogleTest Topics," GoogleTest Documentation. Доступно: <https://google.github.io/googletest/advanced.html> (приступљено 17. 09. 2026).
- [27] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh, "On the effectiveness of address-space randomization," у *Proc. 11th ACM Conf. Computer and Communications Security (CCS '04)*, Washington, DC, USA, 2004, pp. 298–307, doi: 10.1145/1030083.1030124.
- [28] C. Curtsinger and E. D. Berger, "STABILIZER: Statistically sound performance evaluation," у *Proc. 18th Int. Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS '13)*, Houston, TX, USA, 2013, pp. 219–228, doi: 10.1145/2451116.2451141.
- [29] camel-cdr, "RVV benchmark SpacemiT X60 (Banana Pi BPI-F3, SpacemiT K1)," *rvv-bench-results*. Доступно: https://camel-cdr.github.io/rvv-bench-results/spacemit_x60/index.html (приступљено 19. 09. 2026).
- [30] M. Larabel, "RISC-V LLVM Scheduler Tuning For SpacemiT-X60 On Clang Yields 4~18% Speedups," *Phoronix*, 6. 05. 2025. Доступно: <https://www.phoronix.com/news/SpacemiT-X60-LLVM-Scheduler> (приступљено 19. 09. 2026).
- [31] Banana Pi, "BPI-F3 Introduction," BPI-F3 Documentation. Доступно: https://docs.banana-pi.org/en/BPI-F3/BananaPi_BPI-F3 (приступљено 19. 09. 2026).

- [32] Y. Chen, A. Brahmakshatriya, C. Mendis, A. Renda, E. Atkinson, O. Sýkora, S. Amarasinghe and M. Carbin, "BHive: A benchmark suite and measurement framework for validating x86-64 basic block performance models," in *Proc. IEEE Int. Symp. on Workload Characterization (IISWC)*, 2019.
- [33] A. Renda, Y. Chen, C. Mendis and M. Carbin, "DiffTune: Optimizing CPU simulator parameters with learned differentiable surrogates," in *Proc. 53rd IEEE/ACM Int. Symp. on Microarchitecture (MICRO)*, 2020, pp. 442–455.
- [34] A. Abel and J. Reineke, "uiCA: Accurate throughput prediction of basic blocks on recent Intel microarchitectures," in *Proc. 36th ACM Int. Conf. on Supercomputing (ICS)*, 2022.
- [35] Standard Performance Evaluation Corporation, "SPEC CPU® 2017 Overview / What's New?," Q15. [Online]. Available: <https://www.spec.org/cpu2017/Docs/overview.html>. [Accessed: Sep. 20, 2026].
- [36] Standard Performance Evaluation Corporation, "SPEC CPU®2017 Overview / What's New?," 2020. Доступно: <https://www.spec.org/cpu2017/Docs/overview.html> (приступљено 20. 09. 2026).
- [37] RISC-V International, "V" Standard Extension for Vector Operations, Version 1.0, RISC-V ISA Manual, RISC-V Ratified Specifications Library. Доступно: <https://docs.riscv.org/reference/isa/v20260120/unpriv/v-st-ext.html> (приступљено 21. 09. 2026).