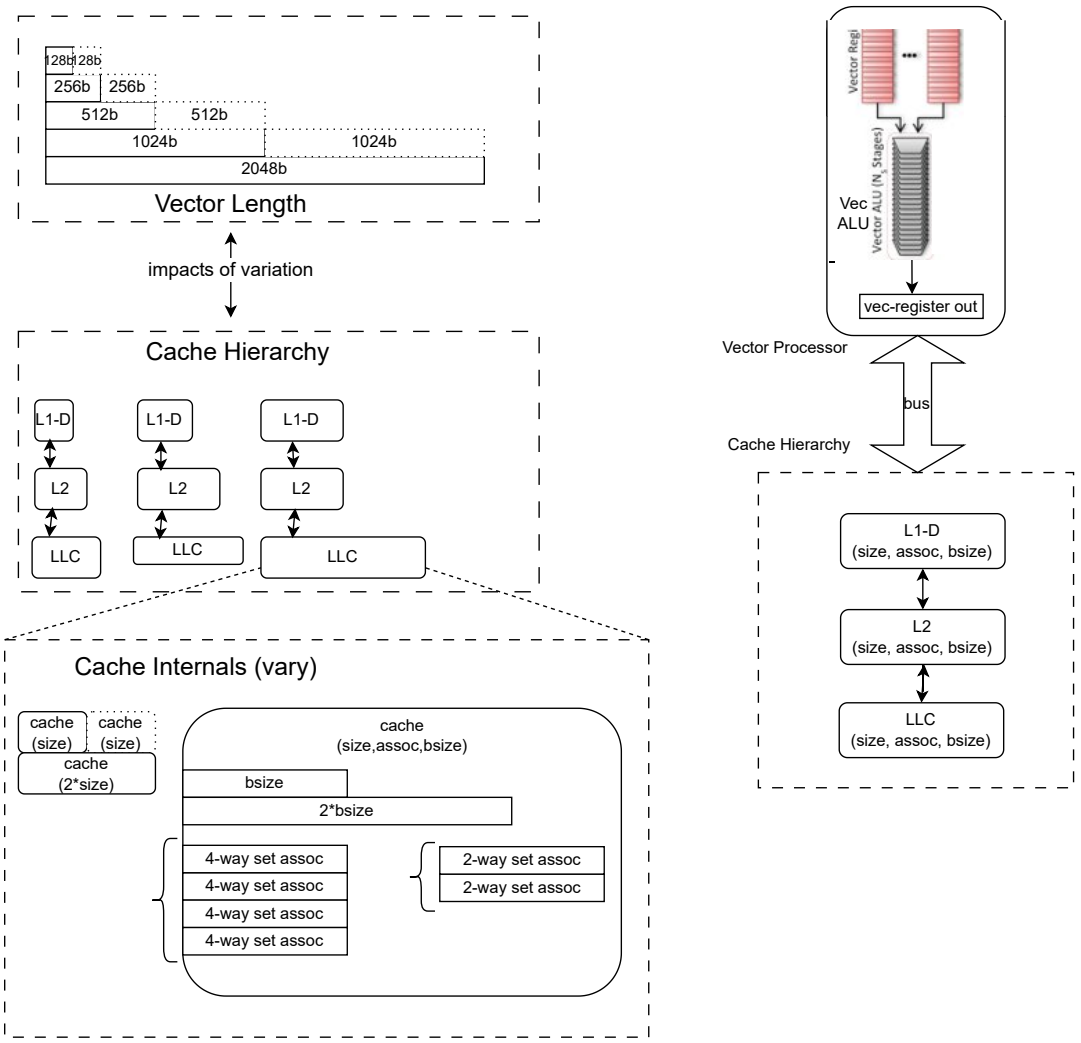
CHALMERS
UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG



Fast performance estimations for vector architectures using qemu-plugins

Master's thesis in Computer science and engineering

HARISH AMBALE GOPALAKRISHNA

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2025

Master's thesis 2025

Fast performance estimations for vector architectures using qemu-plugins

HARISH AMBALE GOPALAKRISHNA

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2025

Fast performance estimations for vector architectures using qemu-plugins
HARISH AMBALE GOPALAKRISHNA

Supervisor: Miquel Pericas, Department of Computer Science and Engineering
Examiner: Miquel Pericas, Department of Computer Science and Engineering

Master's Thesis 2025
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2025

Fast performance estimations for vector architectures using qemu-plugins
HARISH AMBALE GOPALAKRISHNA
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

New applications requiring high computations have arrived (AI, HPC, Cybersecurity etc..) leading to algorithm-hardware configuration exploration side-by-side. This has created the need for tools that can quickly navigate (and thereby prune) the wide design space to be explored by providing performance estimates in a short time-span. This thesis develops/extends a tool to provide performance estimates for an algorithm on a given hardware configuration having vector processing unit and cache hierarchy. It further validates the accuracy of results upon comparison with golden reference.

Keywords: Computer, science, computer science, engineering, project, thesis.

Acknowledgements

I thank my parents, my supervisor along with other professors of Computer Science and Engineering Department at Chalmers for their trust, guidance, discussions throughout the journey of my master studies.

Harish Ambale Gopalakrishna, Gothenburg, 2025-12-05

Contents

List of Figures **xi**

List of Tables **xiii**

1 Introduction **1**
1.1 Background . . . 1
1.2 Context . . . 2
1.3 Vector Length Agnostic Architectures . . . 3

2 Theory **5**
2.1 Aim . . . 5
2.2 Goals and Challenges . . . 5

3 Methods **7**
3.1 Compilation of benchmark(program) . . . 8
3.2 Verify vectorization (visual inspection) . . . 9
3.3 Overview of execution on `qemu-aarch64` and post-processing . . . 9
3.3.0.1 Definition of basic block . . . 9
3.3.1 `qemu-aarch64` program execution . . . 9
3.3.2 Idea of TCG plugin . . . 10
3.3.2.1 `qemu_plugin_register_vcpu_tb_trans_cb()` . . . 11
3.3.2.2 `qemu_plugin_register_vcpu_tb_exec_cb()` . . . 11
3.3.2.3 `qemu_plugin_register_vcpu_mem_cb()` . . . 11
3.3.2.4 `qemu_plugin_register_atexit_cb()` . . . 11
3.3.3 Post-processing `qemu-aarch64` outputs to obtain performance estimates . . . 12
3.4 IPC calculation . . . 13
3.4.1 Plugin for IPC calculation . . . 13
3.4.1.1 basic IPC calculation from `thesis-IPC-basic` . . . 14
3.4.1.2 IPC using `llvm-mca` from `thesis-IPC-llvm-mca` . . . 14
3.4.2 Post processing steps for IPC calculation . . . 14
3.4.2.1 `loop_detector` . . . 14
3.4.2.2 Create multiple assembly files using `make_bb_chunks` 15
3.4.2.3 `run_bb_using_parallel_chunks.sh` . . . 15
3.4.2.4 `aggregate_reports_chunks` . . . 15
3.5 Cache Misses Estimation . . . 15

3.5.1 Plugin for Reuse distance with SHARDS 17
3.5.2 Post processing for Cache miss-rate estimation 18
3.6 MSHR stalls, Bandwidth contention detection 19
3.6.1 Plugin for MSHR stalls, bandwidth contention detection . . . 20
3.6.2 Post processing steps for MSHR stalls and Bandwidth contention detection 21

4 Results **23**
4.1 Execution time 24
4.1.1 Execution time for IPC calculation 24
4.1.2 Execution time for Reuse distance histogram creation 25
4.2 Simulation results 25
4.2.1 IPC comparison 25
4.2.2 Cache misses comparison 29
4.2.3 Cache misses classification (GEMM) 39
4.2.4 Lines-Per-Instruction, MSHR stalls metric (GEMM) 41
4.3 Tabulation 43
4.3.1 IPC calculation results 43
4.3.2 Cache misses estimation results 44
4.4 Discussion 46
4.4.1 IPC calculations 46
4.4.2 Cache misses estimations 46
4.4.3 MSHR stalls and Bandwidth contention detection 47
4.5 Other tools tried 47

5 Conclusion **49**
5.1 Limitations and Future work 50
5.1.1 Limitations 50
5.1.2 Future work 50

Bibliography **53**

List of Figures

3.1 QEMU basic flow 10
3.2 QEMU Plugin concept 11
3.3 QEMU control flow with callbacks 12
3.4 Workflow big picture 13
3.5 Reuse distance with SHARDS approach sample JSON output file . . 18

4.1 GEMM IPC comparison with gem5(state-of-the-art) 26
4.2 SYRK IPC comparison with gem5(state-of-the-art) 27
4.3 Covariance IPC comparison with gem5(state-of-the-art) 28
4.4 MVT IPC comparison with gem5(state-of-the-art) 29
4.5 GEMM cfg1 cache misses comparison with gem5(state-of-the-art) . . 30
4.6 GEMM cfg2 cache-misses comparison with gem5(state-of-the-art) . . 30
4.7 GEMM cfg3 cache-misses comparison with gem5(state-of-the-art) . . 31
4.8 GEMM cfg4 cache-misses comparison with gem5(state-of-the-art) . . 32
4.9 SYRK cfg1 cache-misses comparison with gem5(state-of-the-art) . . . 32
4.10 SYRK cfg2 cache-misses comparison with gem5(state-of-the-art) . . . 33
4.11 SYRK cfg3 cache-misses comparison with gem5(state-of-the-art) . . . 34
4.12 SYRK cfg4 cache-misses comparison with gem5(state-of-the-art) . . . 34
4.13 Covariance cfg1 cache-misses comparison with gem5(state-of-the-art) 35
4.14 Covariance cfg2 cache-misses comparison with gem5(state-of-the-art) 36
4.15 Covariance cfg3 cache-misses comparison with gem5(state-of-the-art) 36
4.16 Covariance cfg4 cache-misses comparison with gem5(state-of-the-art) 37
4.17 MVT cfg1 cache-misses comparison with gem5(state-of-the-art) . . . 37
4.18 MVT cfg2 cache-misses comparison with gem5(state-of-the-art) . . . 38
4.19 MVT cfg3 cache-misses comparison with gem5(state-of-the-art) . . . 39
4.20 MVT cfg4 cache-misses comparison with gem5(state-of-the-art) . . . 39
4.21 Cold, Capacity, Conflict misses classification for L1 cache cfg2 of GEMM VL=512 40
4.22 Cold, Capacity, Conflict misses classification for L2 cache cfg2 of GEMM VL=512 40
4.23 Cold, Capacity, Conflict misses classification for LLC cache cfg2 of GEMM VL=512 41
4.24 Lines Per Instruction metric for GEMM cfg2 with VL=512 41
4.25 MSHR stalls metric for GEMM cfg2 with VL=512 42

List of Tables

4.1 cache configurations chosen for experiments 24
4.2 Execution Time for IPC calculations with format: (VL, Time) 24
4.3 Execution Time for Reuse distance histogram creation with format: (VL, Time) 25
4.4 IPC comparison with state-of-the-art (gem5) 43
4.5 GEMM (compute-bound) benchmark cache miss-rate comparison with state-of-the-art(gem5) 44
4.6 SYRK (compute-bound) benchmark cache miss-rate comparison with state-of-the-art(gem5) 44
4.7 Covariance (memory-bound) benchmark cache miss-rate comparison with state-of-the-art(gem5) 45
4.8 MVT (memory-bound) benchmark cache miss-rate comparison with state-of-the-art(gem5) 45

1

Introduction

1.1 Background

In the recent years, new applications that demand more compute (for e.g. Artificial Intelligence(AI) models, cryptography, High Performance Computing) have arrived, each targeting system metrics (like performance, energy efficiency etc..) unique for particular market segments (battery operated devices to data centers). To address this challenge, algorithmic exploration side-by-side with hardware configurations (termed as Co-design studies) are being performed, which opens up broader Design Space Exploration(DSE) options, leading to highly specialized, (domain specific) compute units (called accelerators). Incorporating these accelerators into existing computer systems are leading to increased diversity of computer architectures.

This creates the need for tools that can quickly navigate the wide design space and provide performance estimates in a short time-span. An example of codesign study is execution of YOLO on vector processors. Performing an algorithm-hardware exploration study using popular (and modular) system-architecture simulation platform like gem5[1] is limited by two main factors: Complexity and Speed. These simulations run at cycle-level, span from days to few months[2, section 5.3] just to check feasibility of the study. The complex nature of gem5 associates to steep learning curve of its internals for making new architectural changes while also making the process of debugging internal errors a complex task by itself.[3]. As an example let us consider the simulation of YOLO using gem5. YOLO (You Only Look Once) is a real-time object detection algorithm[4]. It has evolved to 8th version (YOLO-v8). YOLO-v3[5] was designed to be general-purpose object detection model and thus can be applied to many domains. The main idea of conducting design space explorations is to test multiple combinations of parameters, which results in an explosion of simulation time requirements. Executing YOLO-v3[5] algorithm on gem5 simulated RISC-V Vector Extension v1.0 hardware takes around 3 days. At times, the simulations fail due to internal errors that are cumbersome to debug leading to wasted time and effort. Becoming comfortable to use gem5 takes a long time[6, section 2.2] thus making research-time less productive.

Recently, vector processing has become more popular due to its improved energy efficiency, higher performance and as an alternative to Graphic Processing Units(GPUs). Examples of hardware platforms that support Vector processing include ARM Scalable Vector Extension (SVE) and RISC-V Vector Extension. These are Vector-

Length-Agnostic specifications meaning that it can have multiple implementations and each of which can target different performance metrics as per market needs. Having Vector-Length-Agnostic Instruction Set Architecture specification such as RISC-V Vector Extension, ARM Scalable Vector Extension (SVE) plays a vital role in codesign studies by:

- Conducting more fine-grained algorithm-architecture analysis which inturn translates to defining better requirements for future hardware architectures.
- Providing forward path for multiple real world implementations that target different performance metrics as per market needs.

This step is of pivotal importance due to end of Moore's law and Dennard scaling.

Investigating the behavior of a program (say General Element Matrix Multiply (GEMM)) with different configurations of vector length and cache hierarchy is an interesting co-design study that offers valuable insights. Currently, there are no available tools that can provide *quick* performance estimates with different levels of abstraction (e.g. Instructions Per Clock (IPC) at system level as observed by algorithm, individual hardware component metrics such as cache misses, bandwidth contention as observed from vector processing).

Thus a new (or existing) framework or tool(s) needs to be developed (or extended) for such studies.

1.2 Context

There are different ways of estimating performance. The most commonly used approaches are "binary instrumentation", "Roofline models" and "simulation models". Binary instrumentation involves addition/removal of new instructions to binary executable to extract instruction flow. These are fed to analytical models which calculate performance values by evaluating mathematical equations. Roofline models (original [7], cache-aware[8]) relates performance and ([7] considers only DRAM and [8] considers cache hierarchy as well as DRAM) memory traffic to calculate maximum theoretical upper bounds of performance a machine can reach through a graphical representation. It is more suitable for "throughput" oriented machines meaning they are developed under the assumption that the architecture provides abundant parallelism so that maximum performance can be achieved. Thus Roofline model should not be used when the system is processing latency bound workloads as the effect of latencies are not considered. Even then, roofline model offers valuable insight as to whether an application is memory-bound or compute-bound, which are crucial to determine and thereby plan the direction of future optimizations. Vector architectures belong to this category. Simulation models simulate the execution of instructions of the target processor and are classified based on granularity like cycle-level (SimEng[9], Sniper[10]) where timing is enforced at Instruction granularity meaning it does not model overlapping micro-events such as contentions, hazards, network-forwarding happenning at the same instant of time. Cycle-accurate simulations enforces timing at micro-architectural granularity by modelling every operations' start and completion

at same relative cycles (e.g. hazards, contentions, network-forwarding happenning during a single cycle) as it would on real hardware. Event driven (e.g. SST[11]) simulations depend on the granularity of the event and it can either be cycle-level if all activities happenning at *single clock cycle* are processed *together* or cycle-accurate if every activity at microarchitecture level is individually processed and then clock tick is advanced accordingly. Additionally, there are tools like LLVM-MCA (Machine Code Analysers)[12] that statically measure the performance of machine code in a specific CPU by using information available in LLVM scheduling models. In addtion to above, approaches like Multi level Simulation Approach (MUSA)[13] propose an end-to-end simulation methodology by having an instruction trace generation and simulation infrastructure. Simulation can be configured to be run at multiple levels ranging from coarse grained system level to cycle-accurate micro-architecture level.

One one side, none of the above approaches (other than Spike simulator [14]) currently support RISC-V Vector Extension v1.0 while on the other side even though, the state-of-the-art gem5[1] supports RISC-V Vector Extension 1.0, it is too slow for preliminary design-space-exploration studies. So, in summary, each approach has its strengths to model certain aspects of the system but not the specific combination of Vector-length and cache hierarchy at system level granularity needed for these types of codesign studies taking relatively less simulation time. This thesis will try to find novel ways of obtaining fast performance estimates for a system consisting of CPU, vector processor and a cache hierarchy which translates to finding a tool provides estimates at a speed X times faster than state-of-the-art(gem5[1]), while losing only Y% accuracy.

1.3 Vector Length Agnostic Architectures

In Vector Architectures, the size of the vector register also termed as the Vector Length is an important parameter that signifies how many bits/bytes of data the vector Arithmetic Logic Unit can work upon at a time to generate results. This directly translates to the amount of work accomplished every cycle thereby indicating the performance metric such as Instructions Per Clock or Total Clock cycles. Arm has introduced Scalable Vector Extension (SVE) where the specification defines a vector length from 128bits and doubling this to 512bits, 1024bits and finally to 2048bits. This gives the implementation (micro-architecture) freedom to choose any of the vector lengths based on market needs without recompilation of the user-program. Similarly, the RISC-V has defined RISC-V Vector Extension (RVV) that spefices vector lengths starting from 64bits/128bits upto 65536 bits. Refining this further, in the RVV, even though the micro-architecture implementer supports a certain vector length, the programmer(or compiler) has to specify the exact vector length and the size of each vector element in the program to use only parts of whole vector register. This provides very high flexibility and allows more tighter control for particular target application programs. Since the vector lengths are not hardcoded in specification but left to the micro-architecture implementer for Arm SVE or by the micro-architecture implementor as well as the programmer in RVV, these specifications are considered as Vector Length Agnostic specifications.

Differences include

- Key idea: achieve scalable performance across cores and workloads in case of Arm SVE; achieve portable vectorization across diverse RISC-V cores.
- Vector length : maximum of 2048bits for Arm SVE and must be multiple of 128bits; maximum of 65536bits and being multiple of 2 for RVV.
- In RVV, the programmer has to specify the vector length as well as size of each element (e.g. float=4 bytes per element, double=8 bytes per element); In Arm SVE, only the hardware vector length is specified by the micro-architecture implementor.
- Instruction encoding: Fixed-length (32bit) for Arm SVE and variable length for RVV. This makes decoding more complicated for RVV.
- Toolchain support: Based on experimentation done in this thesis to run a vectorized program, Arm SVE toolchain (compilers, intrinsics) are more mature, stable and RVV is still emerging.

2

Theory

2.1 Aim

Extend or create a tool that can provide quick performance estimates for a given hardware configuration. This helps in fast design space exploration of hardware-software codesign approach followed for developing new hardware.
(Currently) identified Performance parameters:

- Total Instructions, total clock cycles
- Instructions Per Clock (IPC)

  These strongly depends on

  - Cache misses for each level in the cache hierarchy
  - MSHR (Miss Status Handling Register) stalls at each level in the cache hierarchy
  - Bandwidth Contention (between processor and first level cache)

  So, measuring cache misses, MSHR stalls, Bandwidth contention are necessary (done by this thesis) and can be combined in some fashion to arrive at final IPC, total cycles. Due to limitation of time, combining the above results is not part of this thesis work.

2.2 Goals and Challenges

Primary goal is to execute any algorithm around 100x faster than gem5 simulation duration with an acceptable accuracy loss kept below 50% on ARM SVE supported (simulated) hardware. It involves selecting analytical or functional or event driven timing model(s) for faster simulation, supporting different vector lengths, cache hierarchy (L1, L2, LLC).

The research question(s) that I would like to seek answers include:

- Find out one fast way to estimate performance of vector architectures (i.e. system supporting vector extensions like ARM SVE, RISC-V Vector Extension v1.0) with an acceptable accuracy loss below 50%.

- How does changing cache hierarchy and vector lengths affect each other and their corresponding impacts on algorithm?

One of the main challenges is to support RISC-V Vector Extension v1.0 (RVV) as currently there are no baseline models and developing this is quite demanding and takes considerable time. Since ARM SVE and RVV are two different specifications having same end goals, it is very challenging to develop a tool that support both ARM SVE and RVV with minimal code differences. However, developing a tool that can cover both specifications will play a critical role and will offer applications a better contrast and thus facilitates making informed choices.
A new co-design tool/framework plays a crucial role to obtain initial estimates before using cycle-accurate (or cycle-level) simulators for detailed study. This thesis will either develop a new tool or extend an existing framework to aid co-design studies by providing fast and considerably accurate performance estimates.

3

Methods

The goal of this thesis is obtain quick performance estimates along with studying how vector length affects cache hierarchy and the final performance on a vectorized (auto/-manual) program. The program is compiled enabling auto/manual vectorization and then executed (on emulator) to obtain performance estimates. The execution step can be further broken down into actual execution on target-emulator/target-simulator to collect vital data and post-processing the collected data to obtain final performance numbers. The execution and post-processing steps are described one after another for each plugin as they are closely connected and each plugin is independent of one another.

Both Arm SVE and RVV are suitable for proposed methodology described in this chapter. However, **Arm SVE was chosen** due to

- stable, mature platform
- good toolchain support with ease of changing vector length using command line options
- existence of real-world hardware so that the results from thesis work can be validated by running on real hardware (This is not part of current thesis work).

Standard benchmark programs present in Polybench [15] are considered as the test programs. 4 benchmarks (2 compute bound, 2 memory bound) are chosen randomly:

1. **GEMM(General Matrix Multiply)** : Compute bound; used in almost every Deep Learning Algorithms
2. **SYRK(SYmmetric Rank-K update)** : Compute bound; used as a part of back-propagation in training Deep Neural Networks, linear algebra libraries used by PyTorch, TensorFlow etc..
3. **covariance** : Memory bound; used for Feature extraction, dimensionality reduction (Principal Component Analysis(PCA)[16]
4. **MVT(Matrix Vector Transpose)**: Memory bound; core building block in iterative solvers, feature projection steps (recommendation engines)

It should be noted that these are only sample benchmark programs that are considered randomly (2 compute bound and 2 memory bound). Other benchmark programs can

also be used. Once a benchmark program is chosen, the sequence/order of workflow is as follows:

1. Compilation of benchmark (program)
2. Verify vectorization (visual inspection)
3. Execution on `qemu-aarch64`
4. Post-processing `qemu-aarch64` outputs to obtain performance estimates

3.1 Compilation of benchmark(program)

Each benchmark is compiled for different vector lengths currently supported by `qemu-aarch64` [`128b, 256b, 512b`) by enabling auto-vectorization feature using the below command and setting `-msve-vector-bits` to one of `128, 256, 512`. `qemu-aarch64` limits having higheer vector-lengths even though they are supported by ARM SVE [17]. This creates the binary (ELF) that can be executed on a real or simulated ARM SVE hardware.

```
clang-18 -static -O3 -g --target=aarch64-linux-gnu -march=armv8-a+sve
    -mcpu=a64fx -msve-vector-bits=512 -DDATA_TYPE_IS_FLOAT
    -DLARGE_DATASET -I/PolyBenchC-4.2.1/utilities
    --sysroot=/usr/aarch64-linux-gnu
    PolyBenchC-4.2.1/utilities/polybench.c ludcmp.c -o
    ludcmp512b_float_large
```

where `-static` is used to inform compiler to create static binary (self-sufficient, no external linkages, dependencies).

`-O3` enables Optimization to level 3 (highest) and this is pre-requisite to enable auto-vectorization.

`-g` enables global symbols, useful for debugging

`-target=aarch64-linux-gnu` specifies the target architecture to generate the binary.

`-march=armv8-a+sve` enables SVE (Scalable Vector Extension) which uses SVE opcodes, registers etc.. in the generated binary.

`-mcpu=a64fx` is used to specify target CPU. `a64fx` supports 512b vectors and O3 execution.

`-msve-vector-bits` is used to select the Vector length `128,256,512,1024,2048`.

`-DDATA_TYPE_IS_FLOAT, -DLARGE_DATASET` are data-types to be used for the program compilation.

`-sysroot=/usr/aarch64-linux-gnu` specifies the path to cross-compiler installation

3.2 Verify vectorization (visual inspection)

Confirming code vectorization is a necessary step as the fundamental idea of this thesis is to aid codesign studies trying to incorporate vector hardware. Avoiding this step will defeat the purpose of this thesis as there are existing tools that already provide quick performance estimates. But, at the same time the methods developed in this thesis are generic enough and they can still be useful for systems not having any vector units. Vectorization is confirmed by manual inspection upon checking the presence of ARM SVE mnemonics by disassembling the binary (created in previous step). Disassembly command:

```
llvm-objdump-18 --arch=aarch64 -d ludcmp512b_float_large >
    ludcmp512b_float_large_disasm.txt
```

SVE mnemonics can be for example the predicate registers `p0..p15` or `z0..z31` registers, opcodes like `ld1b, st1b, ld1w, st1w, scvtf` etc..

3.3 Overview of execution on `qemu-aarch64` and post-processing

3.3.0.1 Definition of basic block

Basic block is sequence of consecutive instructions characterized by single entry, single exit without any branch instructions. It terminates when there is a branch/call/return instruction.

3.3.1 `qemu-aarch64` program execution

The basic principle of an emulator is to convert the guest instructions present in the binary into equivalent host instructions execute them on the host machine. Following the same principle, `qemu-aarch64` reads the input binary containing ARM aarch64 instructions. Each aarch64 instruction goes through 2 steps:

- Translation phase: Create multiple translation blocks (i.e. basic block) of host instructions.
  - Create multiple basic blocks of guest instructions.
  - Translate each guest instruction basic block into host instruction basic block and keep it ready for execution.
- Execution phase: Execute multiple host instruction basic blocks

What really happens is that when the program starts to execute, it checks if ther eare any host ISA basic blocks available for execution. If so, it executes each of the host ISA basic blocks and if there is no host ISA basic block, it will do a Just-In-Time (JIT) Translation phase on the go and create an Execution basic block and then, execute this basic block. This is depicted in Figure-3.1

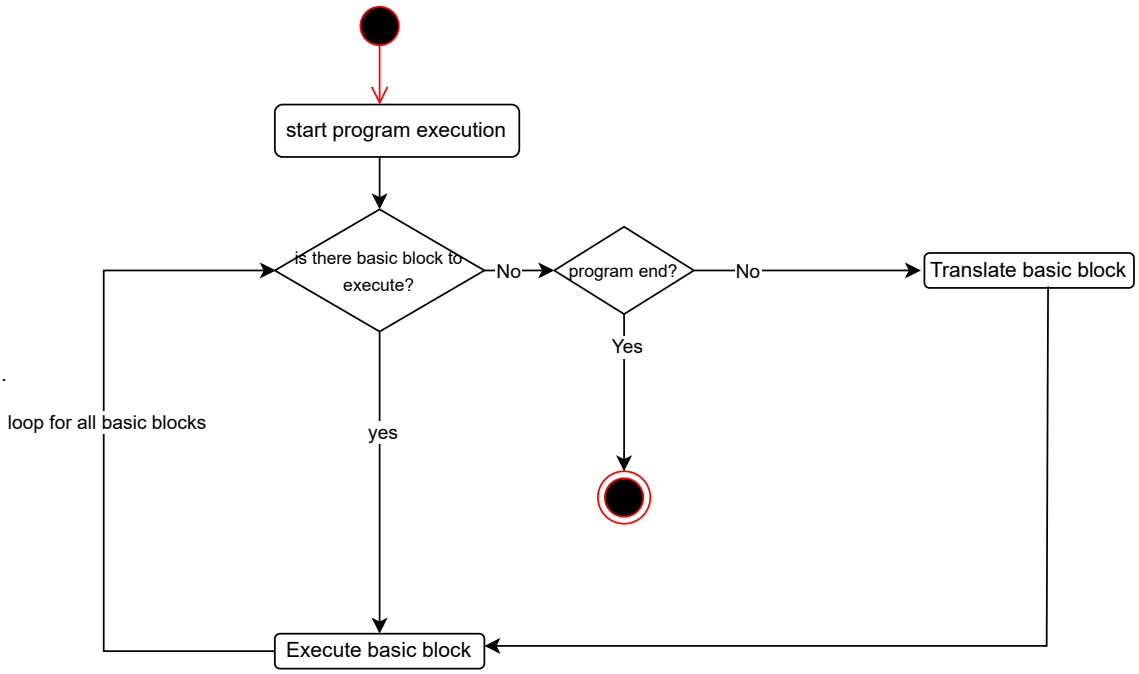


Figure 3.1: QEMU basic flow

3.3.2 Idea of TCG plugin

As the name suggests, "plug-in" is a function provided by the user (to a system) registering for a certain event so that the user receives a trigger/callback (from the system) upon the occurrence of that event. `qemu-aarch64`, reads the user plugin and inserts callbacks (to user-provided-functions) at translation phase. During execution phase, the callbacks invoke the user-provided-functions (containing custom logic). There are 2 levels at which a plugin can observe the program execution: per-tb-execution(`qemu_plugin_register_vcpu_tb_exec_cb()`), per-instruction-execution (`qemu_plugin_register_vcpu_insn_exec_cb()`, not used here). The important plugins used in this thesis work are briefly explained below. The basic idea of qemu plugin is as shown in Figure-3.2.

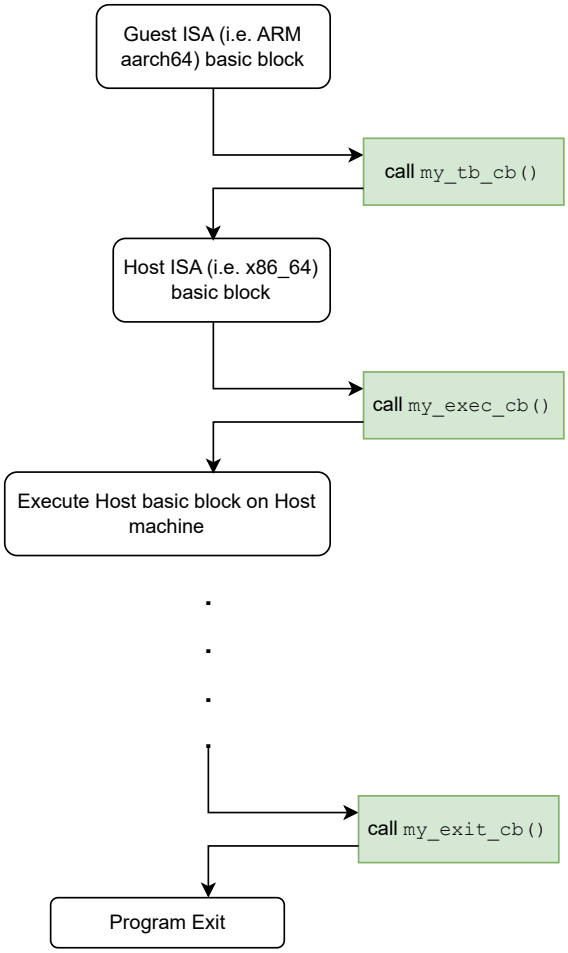


Figure 3.2: QEMU Plugin concept

3.3.2.1 `qemu_plugin_register_vcpu_tb_trans_cb()`

Calls the user-provided-function (at translation phase) just time after the creation of the guest instruction basic block and before translation into the host instruction basic block.

3.3.2.2 `qemu_plugin_register_vcpu_tb_exec_cb()`

Calls the user-provided-function (at execution phase) every time before a particular basic block starts execution.

3.3.2.3 `qemu_plugin_register_vcpu_mem_cb()`

Calls the user-provided-function (at execution phase) every time before a memory instruction (i.e. load/store) starts execution. This API provides the memory-address on which the load/store is about to be invoked.

3.3.2.4 `qemu_plugin_register_atexit_cb()`

Calls the user-provided-function (at execution phase) only once after all the guest instructions are finished executing (i.e. end of program execution) and just before the

`qemu-aarch64` finishes. This is useful to flush the data collected by user-provided-function into a file.

The qemu control flow when using plugins is as shown in Figure-3.3

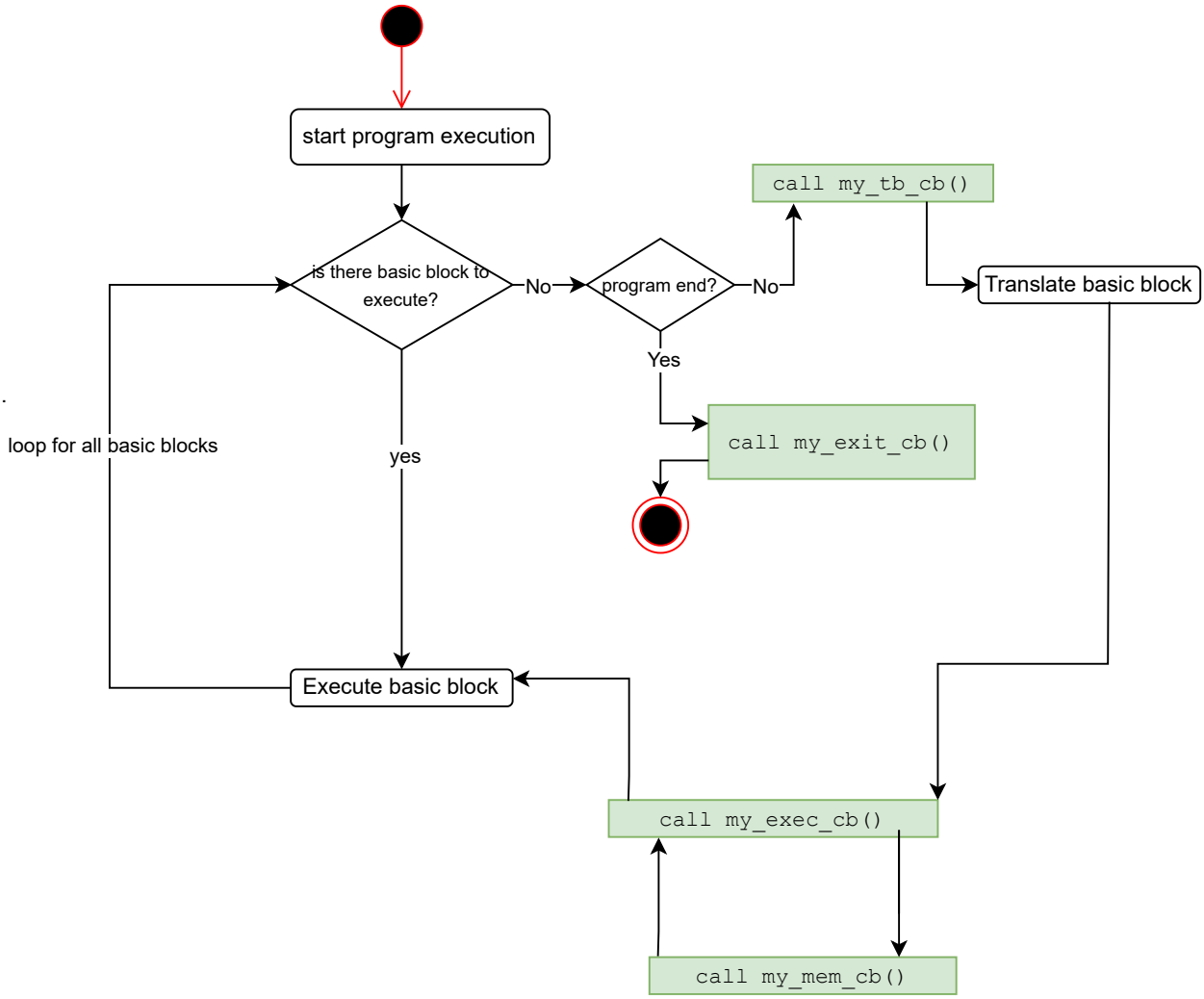


Figure 3.3: QEMU control flow with callbacks

3.3.3 Post-processing `qemu-aarch64` outputs to obtain performance estimates

This part uses data from the `qemu-aarch64` plugins (described below) to calculate intermediate data which are then aggregated to obtain final results.

The whole summary is depicted as a block diagram at Figure-3.4

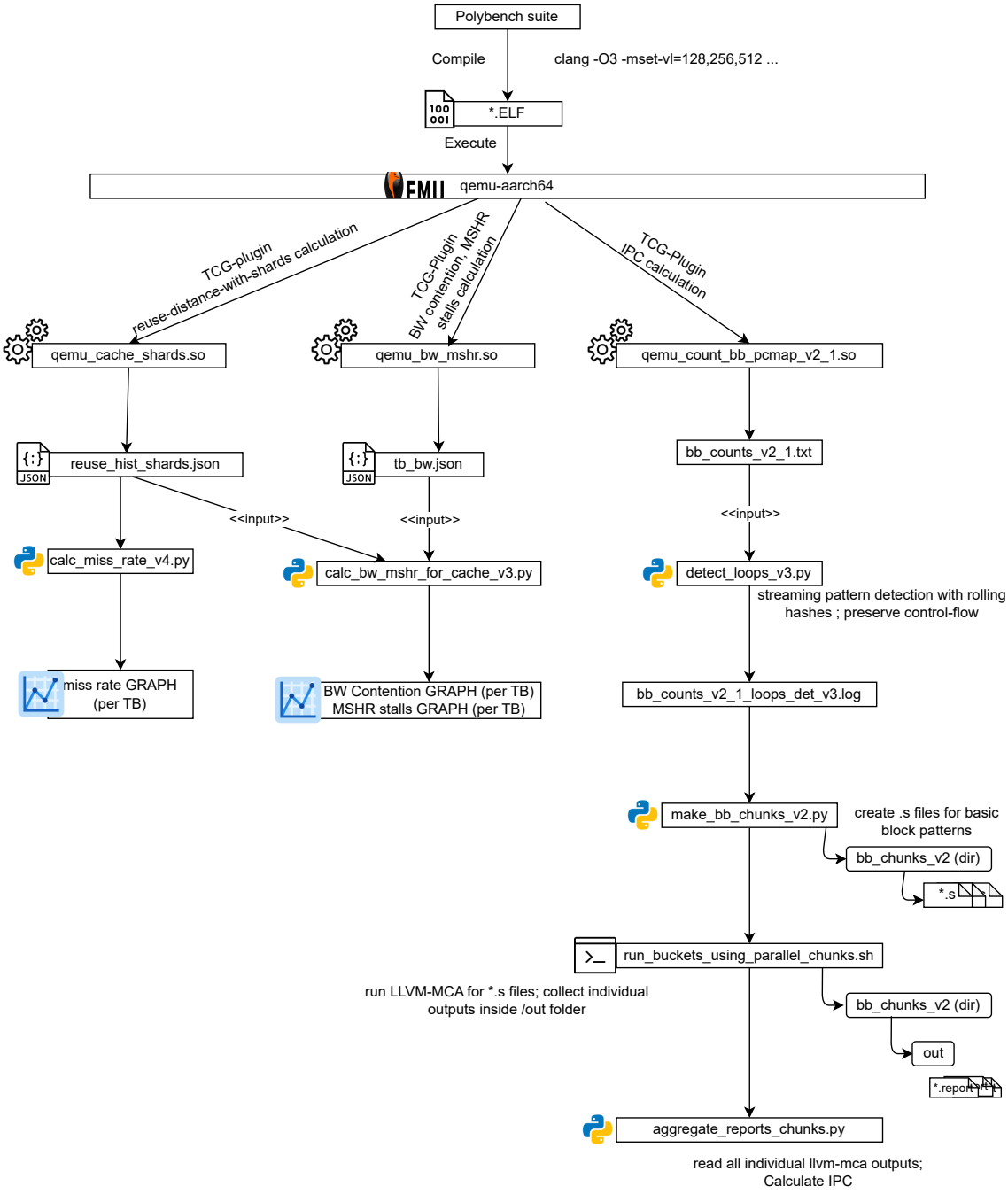


Figure 3.4: Workflow big picture

In the following sections, the execution of each plugin is described. It is immediately followed by the post-processing step. This is done for all 3 plugins.

3.4 IPC calculation

3.4.1 Plugin for IPC calculation

Two different approaches are explored here (i) basic IPC calculation (ii) IPC calculation using `llvm-mca`

3.4.1.1 basic IPC calculation from `thesis-IPC-basic`

As the program executes, increment an instruction counter for every instruction executed and then, count the cycles for every instruction by assigning an assumed latency (taken from **aarch64-instruction-profiler**) for every instruction and counting it till the end. At the program exit, the plugin calculates IPC by dividing the total instructions executed till that point to total cycles consumed till that point and writes it to a file. There is no post-processing step for this plugin.

3.4.1.2 IPC using `llvm-mca` from `thesis-IPC-llvm-mca`

As the program executes, record the control flow at basic-block level by storing the Program Counter and the corresponding count using
`qemu_plugin_register_vcpu_tb_exec_cb()` (refer 3.3.2.2). Try to flatten the loops by incrementing the same counter associated with basic block when the same basic block is executed multiple times in a sequential manner. Otherwise, add a new entry to the list with count as 1. Finally, at program exit, write to a text file using `qemu_plugin_register_atexit_cb()` (refer 3.3.2.4) as list of

```
Program Counter : count
```

This is then fed to the post-processing stage described shortly.

3.4.2 Post processing steps for IPC calculation

The output of `qemu-aarch64` IPC-calculation TCG plugin is a text file containing the start address of basic block and the count i.e. `Program Counter, Count` respecting the control flow of the program. The plugin manages to flatten the loops (i.e. same basic block executed consecutively multiple times). But still, many basic blocks might still be part of a bigger loop which are not easy to be identified inside the plugin considering the fact that it has to finish the program execution as quickly as possible. Hence, in the post-processing part, bigger loops containing multiple basic blocks are detected by a separate `loop_detector` program (written in python).

3.4.2.1 `loop_detector`

This python program detects the sequential occurrence of multiple blocks by using rolling hash technique[18]. At the end, this will create another output text file containing the basic blocks and the count in the format below and still preserving the control flow of the program execution.

```
Program Counter1, Program Counter2, ... , count.
```

This essentially tells that the sequence of basic blocks
`Program Counter1, Program Counter2, ...` is repeated for `count` iterations. In this way, loops are detected at 2 levels:

- single basic block repeating consecutively : identified by IPC-Calculation-TCG-Plugin (described in 3.4.1.2).
- multiple basic blocks repeating after a certain occurrence : identified by post-processing `loop_detector`

This output is provided to `make_bb_chunks` block for further post-processing.

3.4.2.2 Create multiple assembly files using `make_bb_chunks`

This python program reads the text file provided by `loop_detector` and creates assembly instructions contained inside every basic block. For each line present in the input file, if the `count` of a basic block is `1`, then

- combine multiple basic blocks untill a basic block with `count>1` is encountered OR
- wait until 100000 sequential basic blocks are encountered

and write to an assembly file by expanding each basic block by looking at the disassembly produced in section 3.2.
This creates multiple (10s to 100s of) assembly files and each assembly file internally preserves the control flow of the program.

The created assembly files are input to the next step.

3.4.2.3 `run_bb_using_parallel_chunks.sh`

This is a bash script which creates an output folder and runs `llvm-mca` on each assembly file (produced in previous step (refer 3.4.2.2)) in parallel and collects the output from individual `llvm-mca` executions (`*.report` file).

3.4.2.4 `aggregate_reports_chunks`

This python file reads all individual `llvm-mca` output files and calculates total instructions and total cycles consumed as per `llvm-mca`. Finally, the Instructions-Per-Cycle (`IPC`) is calculated by dividing the total instructions by total cycles consumed.

3.5 Cache Misses Estimation

A design space exploration for cache hierarchy involves parameters such as size, associativity, block size. The design space gets exploded as there can be different levels of caches forming the cache hierarchy. Thus, it is beneficial to come up with an approach where the results for any cache configuration involving the above parameters can be instantly calculated by post-processing an already collected data. The concept of reuse-distance-histogram perfectly fits here because once the reuse-distance histogram is generated, the cache misses for any cache hierarchy can be computed within a few seconds from this histogram. Reuse-distance[19] is defined as the number of unique cache lines accessed between two successive accesses to the

same cache line. The concept of SHARDS[20] extends the reuse-distance concept to be applicable to independent groups of cache lines by computing local reuse distance histograms for a small group of cache lines and then aggregating them together to form the final global reuse distance histogram. The whole idea is to run the simulation once and amortize it over every design space exploration of cache hierarchy.

3.5.1 Plugin for Reuse distance with SHARDS

Builds an online reuse-distance histogram with SHARDS[20] (using order-statistic-tree per TB) on cache-line-addresses. Reuse-distance[19](or stack distance) is the number of unique cache lines accessed between 2 memory accesses to the same cache-line. This is later post-processed to generate Miss-Ratio-Curves that are used to calculate cache-misses for a give cachesize, associativity, block-size. Overview of steps:

- check if the cache-line was seen before. If not, update the cold-miss. If yes, compute stack distance.
- Update the histogram for the corresponding basic block

As the program executes, record each cache-line touched by every basic block using `qemu_plugin_register_vcpu_mem_cb()` (refer 3.3.2.3). Choose a shard by hashing the calculated cache-line. If this memory instruction is a vector instruction (check by using opcodes), then calculate the total number of cache-line requests to be made by dividing vector-length with cache-line-size (provided as plugin arguments). For each new cache-line request, calculate reuse-distance again and update the old entry.

Calculate reuse-distance using SHARDS [20]. The idea is to group memory requests (by multiplying cache-line with a hash) into buckets. Then, the reuse distance is calculated for all the entries present inside this bucket and then aggregated together. This is a way of sampling the memory requests while still trying to maintain accuracy to speed up the whole process of capturing memory requests.
Finally, at program exit, write to a JSON file(easy to read and post-process than .txt file) using `qemu_plugin_register_atexit_cb()` (refer 3.3.2.4).

3.5.2 Post processing for Cache miss-rate estimation

Read the input JSON file from the reuse distance with SHARDS plugin (refer section 3.5.1). A sample output is shown in Figure-3.5.


```
{ // start of JSON file
"0x00424a90": [
    {
    "cold": 1,
    "hist": {
      "0": 1
      }
    },
    {
    "cold": 1,
    "hist": {
      "0": 3
      }
    }
  ],
"0x00400c34": [
    {
    "cold": 1215,
    "hist": {
      "0": 566781,
      "1": 14050,
      "24": 6333,
      "23": 14769,
      "22": 1631,
      "25": 244,
      "2": 193
      }
    },
    {
    "cold": 1217,
    "hist": {
      "0": 570121,
      "1": 13902,
      "25": 6559,
      "24": 15401,
      "23": 1711,
      "26": 305,
      "2": 128
      }
    },
    .
    .
  ],
    .
    .
    .
} // end of JSON file
```


Figure 3.5: Reuse distance with SHARDS approach sample JSON output file

For each basic block,

- Cold misses are already part of the JSON file and are just read.
- Capacity misses are calculated as those reuse distances that exceeds number of lines per SHARD.
- Conflict misses are the calculated as those reuse distances that lie between selected associativity (4,8,16 etc..) and number of lines per SHARD (i.e. remaining misses occurred in that particular SHARD)

The SHARDS[20] approach is considered to additionally incorporate conflict misses. If conflict misses are low or if fully associative caches are considered then the SHARDS approach can be replaced with a simpler reuse distance plugin (thus avoids grouping of cache lines). In a way, the SHARDS[20] approach tries to approximate the locality by random grouping. In real hardware, each cache line is mapped to a set which has a limited capacity which enforces strict constraints. So, the SHARDS[20] approach trades constraints to approximation. The cold, capacity, conflict misses are combined by averaging across all the SHARDS to obtain total cache-misses for a particular cache with associativity A, size S and block size B.

Finally, print cold, capacity, conflict misses as graphs at the basic block level for better visualization and print the total cache misses. The above calculation is performed for every cache (level) present in the hierarchy (e.g. L1, L2, LLC).

NOTE: L1-Instruction Cache is not modeled separately as we can have a single cache at every level in the hierarchy.

3.6 MSHR stalls, Bandwidth contention detection

Performing large computations often involves heavy interaction with memory. And modern compute architectures are inherently parallel and consists of non-blocking caches. These caches can simultaneously serve more than 1 request from the processor. One way of realizing (i.e. implementation) this type of cache is by using Miss Status Handling Registers (MSHR) that store metadata upon arrival of processor requests and updates them when particular processor request gets finished. In vector architectures, this is particularly important to measure as the memory requests made by vector unit span multiple elements. This can also result in contention if sufficient bandwidth is not available. Thinking in this direction, 2 new metrics are defined:

- Lines-Per-Instruction: Try to identify memory-intensive basic blocks and the particular level of cache that gets stressed by these memory-intensive basic blocks.
- MSHR stalls detection: Try to measure MSHR requirement per basic block.

By measuring the above metrics, the hardware elements necessary for sustaining the overall performance can be determined.

3.6.1 Plugin for MSHR stalls, bandwidth contention detection

As the program executes, for each basic block, record:

- no. of instructions executed inside this basic block using `qemu_plugin_register_vcpu_tb_trans_cb` (refer 3.3.2.1)
- no. of cache-lines requested using `qemu_plugin_register_vcpu_mem_cb` (refer 3.3.2.3)
- no. of times this basic block executed using `qemu_plugin_register_vcpu_tb_exec_cb` (refer 3.3.2.2)

Finally, at program exit, write to a JSON file (easy to read and post-process than .txt file) using `qemu_plugin_register_atexit_cb()` (refer 3.3.2.4).

3.6.2 Post processing steps for MSHR stalls and Bandwidth contention detection

Read the JSON files produced by reuse distance with SHARDS plugin (refer 3.5.1) and Bandwidth contention and MSHR stalls detection plugin (refer 3.6.1). Provide count of MSHR at each level of cache hierarchy and cache size (as number of cache-lines).

From the output of reuse distance with SHARDS plugin, calculate the overall cache misses for each level. The defined metric **Lines-Per-Instruction(LPE)** is calculated for each basic block as the ratio of number of cache-lines requested by that basic block to the number of instructions executed inside that basic block. Finally the top 10 basic blocks with highest LPE are plotted and visualized. Additionally, LPE for all the basic blocks are plotted for reference. Higher the relative value of LPI, the corresponding basic blocks most likely cause bandwidth contention which thereby introduces additonal latencies into the system. This is particularly useful for L1 caches as these directly receive the processor requests which might span multiple elements in case of vector units.
The **Misses-Per-Execution (MPE)** is determined for each basic block as the ratio of number of misses occurred for this basic block to number of times this basic block is executed. This is a measure of MSHR pressure caused by the workload (at basic blocks level).

Upon subtracting the number of MSHRs defined for this cache level with the MPE, we obtain the occurrence of MSHR stalls for a particular basic block on a particular cache level (e.g. L1, L2 or LLC). Ideally, the MSHR stalls must be 0 (or negative) so that there are no pipeline stalls introduced due to unavailablilty of MSHRs. A negative number indicates that there are more MSHRs available than the demands of that particular basic block. A positive number of MSHR stalls for a particular basic block indicate that additional MSHR are needed to avoid pipeline stalls (and additional delays) in the system.

Finally, the top 10 basic blocks having highest MSHR stalls are plotted for better visualization. Additionally, MSHR stalls for every basic block are also plotted for reference.

4

Results

This chapter describes the experimental setup, the host execution times for simulations and the results of simulations followed by a discussion of the obtained simulation results.

The experiments of this thesis are to simulate the execution of the auto-vectorized binaries for the benchmark programs described in section-3 on the `qemu-aarch64` execution environment described in section-3.2. The `qemu-aarch64` execution environment is run on host machine having configuration: AMD® Ryzen 7 7435hs × 16, 16GB RAM running Ubuntu 22.04 LTS operating system. The Each plugin is provided to `qemu-aarch64` execution environment as a command line parameter. The execution of benchmarks produces outputs that are specific to the plugin. These outputs are then post-processed as described in sections 3.4.1.2, 3.5.1 and 3.6.1. Finally, the results are depicted in the form of graphs or figures. The results are numbers for: IPC, cache misses (as %), host simulation time and graphs for MSHR stalls and Bandwidth contentions.

It must be noted that all of the simulations are executed in parallel on `qemu-aarch64` execution environment to collect initial plugin data. For post-processing, the IPC calculation depends on its own plugin; the cache-misses estimation depends on its own plugin; the post-processing step of MSHR stalls and Bandwidth contention detection depends on Resue-distance histogram generated from cache-misses plugin (refer section-3.5).

At first, the Table-4.1 provides the different cache configurations used in this work. Then, the execution time taken for each simulation are presented in Table-4.2 and Table-4.3 which will be referred for explaining the result of each simulation. Finally, the output of simulations are presented in the form of graphs, figures. The obtained results are discussed in 3 parts:

- the main result and its comparison to the gem5 tool.
- how the obtained results correlate to presented theory
- possible answers derived for speed of simulation and relaxation of accuracy.

cache-config	L1 {size, assoc, bsize}	L2 {size, assoc, bsize}	LLC {size, assoc, bsize}
cfg1	32KB, 8, 64B	256KB, 8, 64B	1MB, 16, 64B
cfg2	64KB, 8, 64B	512KB, 8, 64B	2MB, 16, 64B
cfg3	64KB, 8, 64B	1MB, 8, 64B	4MB, 16, 64B
cfg4	64KB, 8, 64B	2MB, 8, 64B	8MB, 16, 64B

Table 4.1: cache configurations chosen for experiments

4.1 Execution time

The following subsections tabulates the execution time taken on the host machine. It must be noted 2 or more gem5 simulations were run in parallel and there were also ubuntu related background tasks which might have affected the execution time presented below.

4.1.1 Execution time for IPC calculation

Benchmark	Thesis basic-IPC	Thesis llvm-mca-IPC	gem5ArmMinorCPU	gem5ArmO3CPU
GEMM	(128b, 9s) (256b, 31s) (512b, 22s)	(128b, 17s) (256b, 38s) (512b, 115s)	(128b, 5400s) (256, 3250s) (512, 2600s)	(128b, 22600s) (256b, 21000s) (512b, 21000s)
Covariance	(128b, 31s) (256b, 80s) (512b, 82s)	(128b, 160s) (256b, 230s) (512b, 208s)	(128b, 50000s) (256, 50000s) (512, 37000s)	(128b, 85000s) (256b, 106000s) (512b, 113000s)
SYRK	(128b, 16s) (256b, 25s) (512b, 18s)	(128b, 360s) (256b, 311s) (512b, 75s)	(128b, 16000s) (256, 8500s) (512, 3000s)	(128b, 24000s) (256b, 5500s) (512b, 1900s)
MVT	(128b, 0.4s) (256b, 0.4s) (512b, 0.4s)	(128b, 11s) (256b, 4s) (512b, 4s)	(128b, 180s) (256, 165s) (512, 159s)	(128b, 330s) (256b, 320s) (512b, 320s)

Table 4.2: Execution Time for IPC calculations with format: (VL, Time)

Since this thesis only uses the `qemu-aarch64` execution environment and only feeds `llvm-mca`, there is no finer control over the internal models of CPU and Vector units of `qemu-aarch64` and `llvm-mca`. This makes it very hard to reason the variation of execution times ranging different vector lengths. However, comparing the execution times achieved by this thesis with gem5, a very high speedup of around $100x$ can be observed from the numbers presented in Table-4.2.

4.1.2 Execution time for Reuse distance histogram creation

Benchmark	Thesis-reuse-dist-calc-time	gem5ArmMinorCPU	gem5ArmO3CPU
GEMM	(128b, 450s) (256b, 1620ss) (512b, 1620s)	(128b, 5400s) (256, 3250s) (512, 2600s)	(128b, 22600s) (256b, 21000s) (512b, 21000s)
Covariance	(128b, 3600s) (256b, 3600s) (512b, 3600s)	(128b, 50000s) (256, 50000s) (512, 37000s)	(128b, 85000s) (256b, 106000s) (512b, 113000s)
SYRK	(128b, 1500s) (256b, 1080s) (512b, 1080s)	(128b, 16000s) (256, 8500s) (512, 3000s)	(128b, 24000s) (256b, 5500s) (512b, 1900s)
MVT	(128b, 20s) (256b, 20s) (512b, 20s)	(128b, 180s) (256, 165s) (512, 159s)	(128b, 330s) (256b, 320s) (512b, 320s)

Table 4.3: Execution Time for Reuse distance histogram creation with format: (VL, Time)

As explained in 3.4.2.4, the simulation time provided in Table-4.3 quantifies the benefits of using reuse-distance-histogram approach to amortize the vast design space exploration of cache hierarchy. Now, the user can run multiple experiments changing each of the size, associativity, block size for cache level and multiple levels of caches to form cache hierarchy and still obtain instant results.

4.2 Simulation results

4.2.1 IPC comparison

The `thesis-IPC-basic` and `thesis-IPC-llvm-mca` models presented in this thesis work are compared against corresponding `ArmMinorCPU` and `ArmO3CPU` models of gem5. The reason for this comparison is that the `thesis-IPC-basic` represents the in-order, simple model being similar to gem5's `ArmMinorCPU`. On similar lines, the `thesis-IPC-llvm-mca` attempts to model out-of-order CPU as `llvm-mca` only supports Fujitsu A64FX CPU model [21] (and [22]). The Fujitsu A64FX is the only model that supports all the Vector Lengths `128b, 256b, 512b` which is exactly supported by `qemu-aarch64` at the moment.

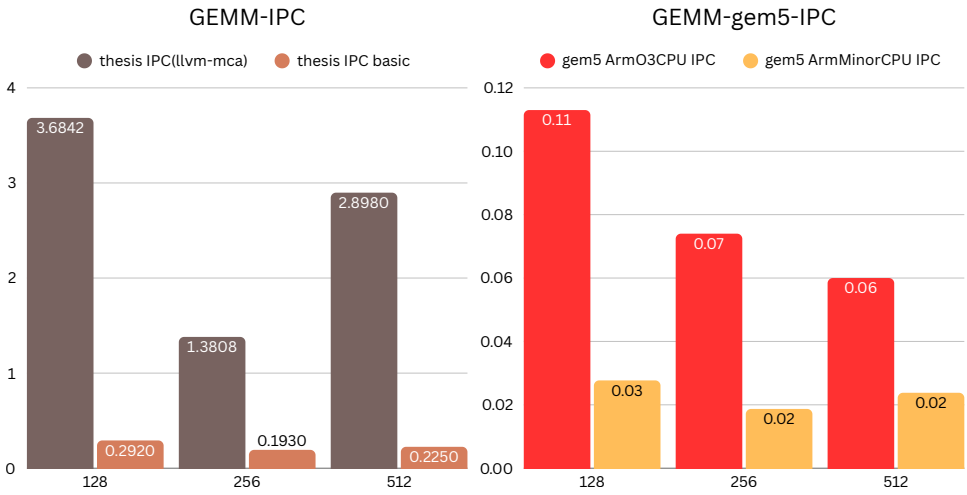


Figure 4.1: GEMM IPC comparison with gem5(state-of-the-art)

Looking at Figure-4.1, as vector length is increased, the IPC decreases slightly in both the `thesis-IPC-basic` and gem5 `ArmMinorCPU`. On the contrary, the IPC decreases from VL=128 to VL=256 and then increases from VL=256 to VL=512 for `thesis-IPC-llvm-mca` but decreases steadily for gem5 `ArmO3CPU`. Hence the `thesis-IPC-basic` method correlates well with gem5 `ArmMinorCPU`.
The `thesis-IPC-llvm-mca` method weakly correlates with gem5 `ArmO3CPU`.

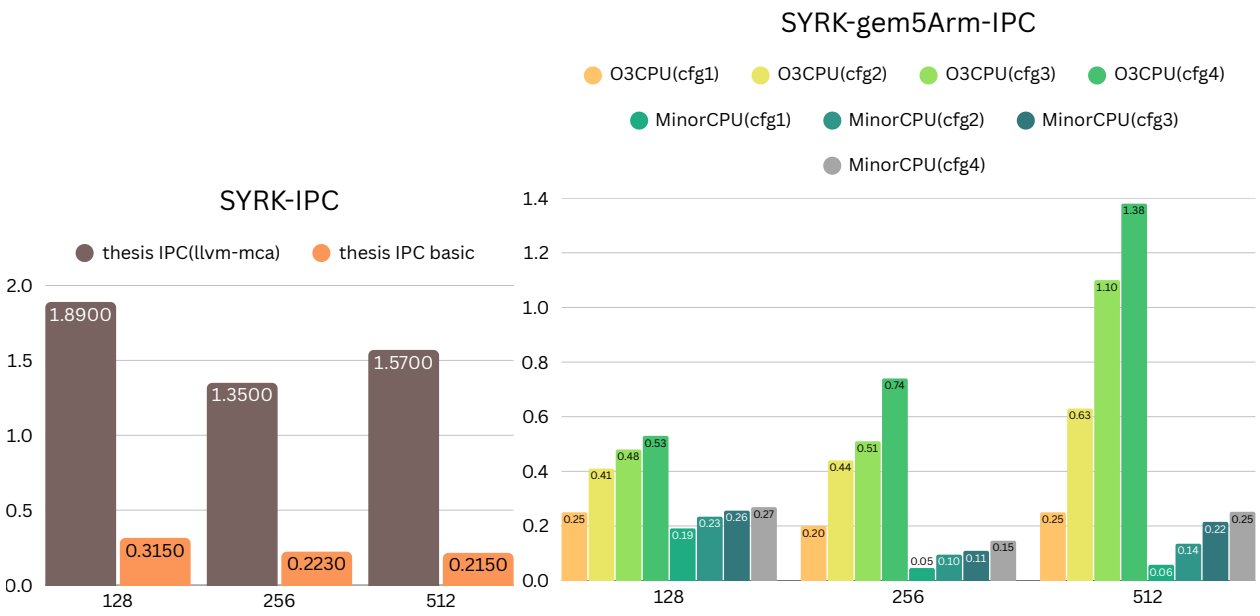


Figure 4.2: SYRK IPC comparison with gem5(state-of-the-art)

Looking at Figure-4.2, as vector length is increased, the IPC decreases slightly in both the `thesis-IPC-basic` and gem5 `ArmMinorCPU`. On the contrary, the IPC decreases from VL=128 to VL=256 and then increases from VL=256 to VL=512 for `thesis-IPC-llvm-mca` but increases steadily for gem5 `ArmO3CPU`. Hence the `thesis-IPC-basic` method correlates well with gem5 `ArmMinorCPU`. The `thesis-IPC-llvm-mca` method weakly correlates with gem5 `ArmO3CPU`.

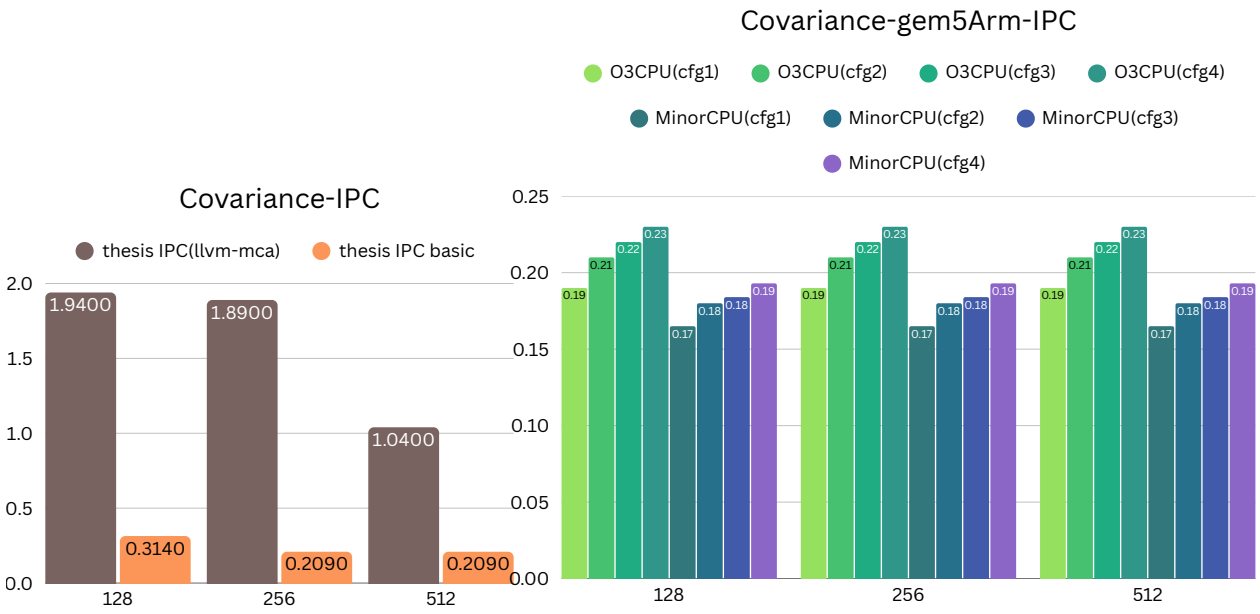


Figure 4.3: Covariance IPC comparison with gem5(state-of-the-art)

Looking at Figure-4.3, as vector length is increased, the IPC remains constant in both the `thesis-IPC-basic` and gem5 `ArmMinorCPU`. The IPC remains steady for VL=128 and VL=256 in both the `thesis-IPC-llvm-mca` and gem5 `ArmO3CPU` and strangely for VL=512, the `thesis-IPC-llvm-mca` reduces but gem5 `ArmO3CPU` remains steady as before. Hence the `thesis-IPC-basic` method correlates well with gem5 `ArmMinorCPU`. The `thesis-IPC-llvm-mca` method weakly correlates with gem5 `ArmO3CPU`.

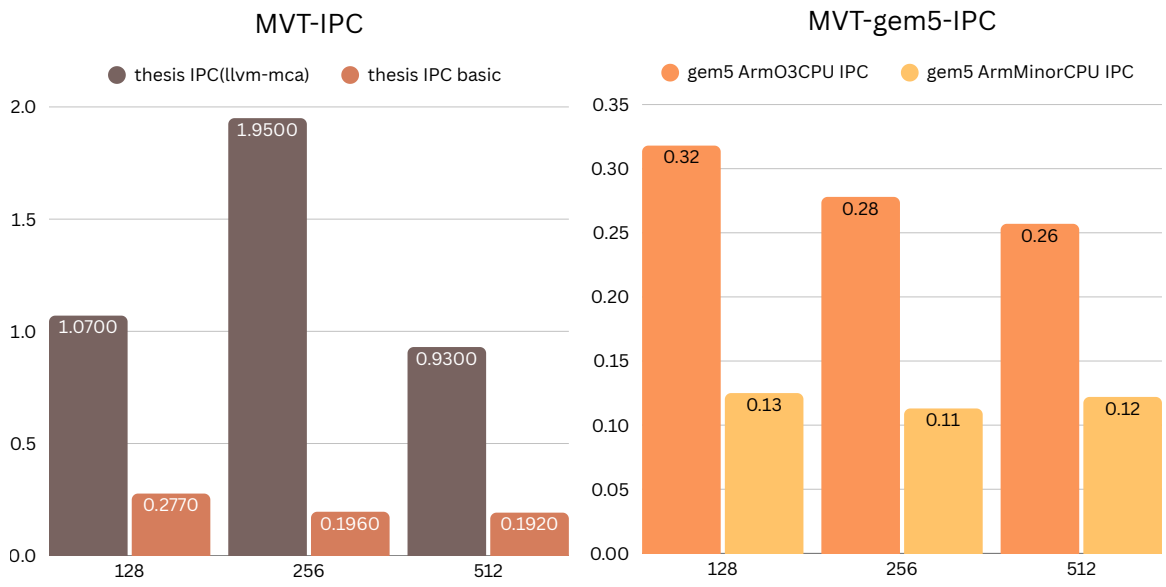


Figure 4.4: MVT IPC comparison with gem5(state-of-the-art)

Looking at Figure-4.4, as vector length is increased, the IPC remains constant in both the `thesis-IPC-basic` and gem5 `ArmMinorCPU`. The IPC remains steady for VL=128 and VL=512 in both the `thesis-IPC-llvm-mca` and gem5 `ArmO3CPU` and strangely for VL=256, the `thesis-IPC-llvm-mca` sharply increases but gem5 `ArmO3CPU` remains steady as before. Hence the `thesis-IPC-basic` method correlates well with gem5 `ArmMinorCPU`. The `thesis-IPC-llvm-mca` method weakly correlates with gem5 `ArmO3CPU`.

4.2.2 Cache misses comparison

Here each benchmark is ran with the cache configurations provided by Table-4.1. Each graph shows the scaling of vector length for particular configuration. Each graph is explained but a general conclusion is attempted to be derived in section-4.4.2.

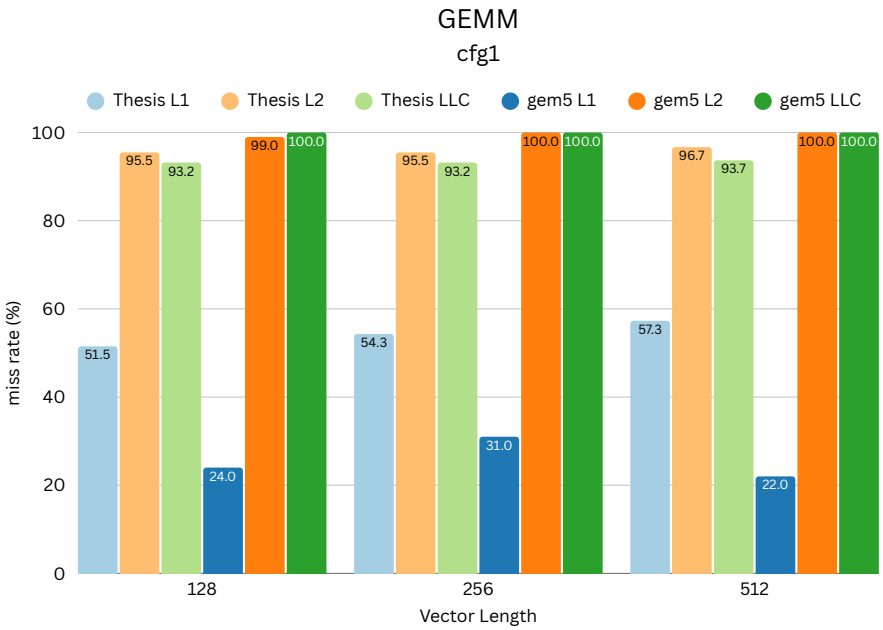


Figure 4.5: GEMM cfg1 cache misses comparison with gem5(state-of-the-art)

Observing figure-4.5, the estimated L1 miss rate seems to be almost double that of gem5, while the L2 and LLC miss rates match within 10% error compared to gem5. It is hard to draw a strong conclusion and a general conclusion is drawn in section-4.4.2

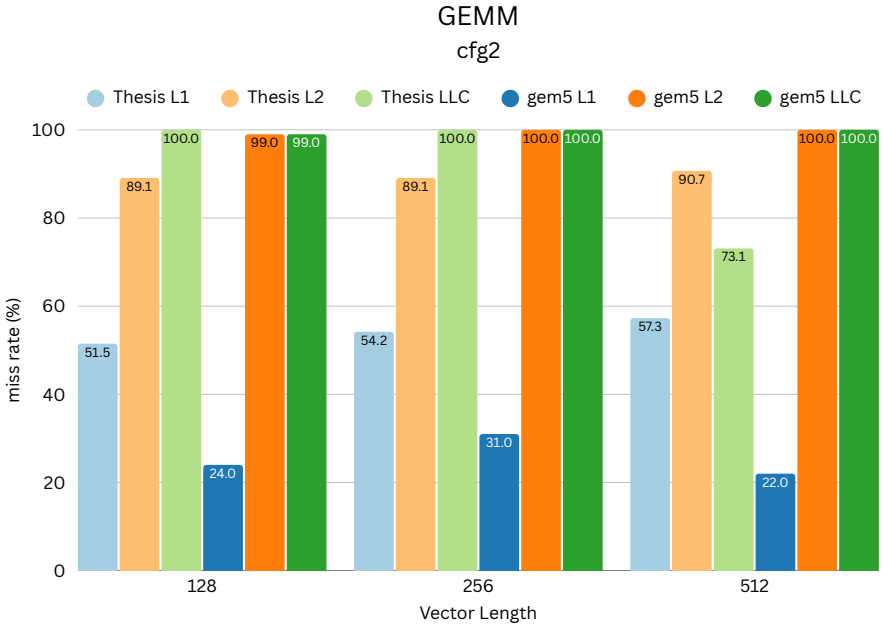


Figure 4.6: GEMM cfg2 cache-misses comparison with gem5(state-of-the-art)

Observing figure-4.6, the estimated L1 miss rate seems to be almost double that of gem5, while the L2 and LLC miss rates match within 10% error compared to gem5 except for VL=512 where thesis-LLC shows 25% error. It is hard to draw a strong conclusion and a general conclusion is drawn in section-4.4.2

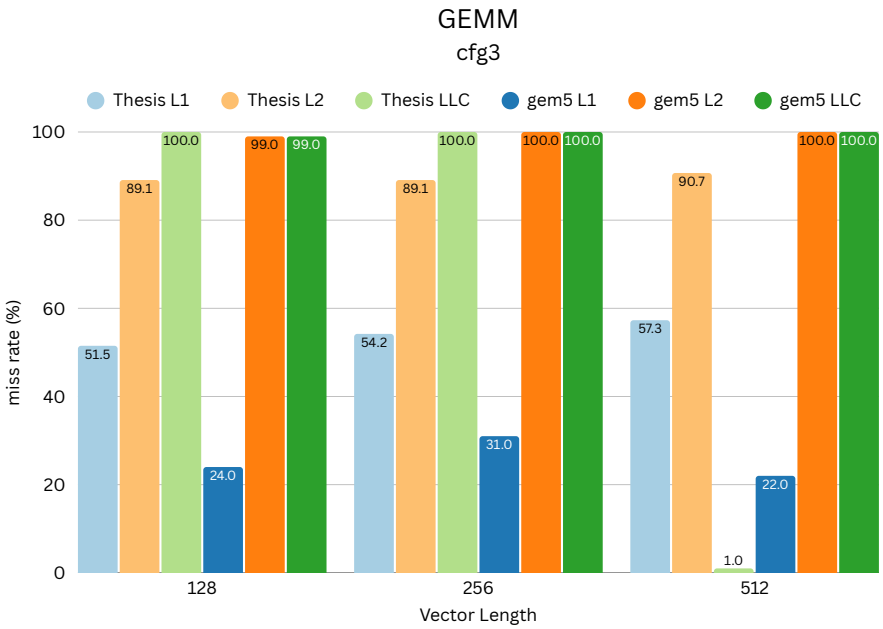


Figure 4.7: GEMM cfg3 cache-misses comparison with gem5(state-of-the-art)

Observing figure-4.7, the estimated L1 miss rate seems to be almost double that of gem5, while the L2 and LLC miss rates match within 10% error compared to gem5 except for VL=512 where thesis-LLC and gem5 contradict each other. It is hard to draw a strong conclusion and a general conclusion is drawn in section-4.4.2

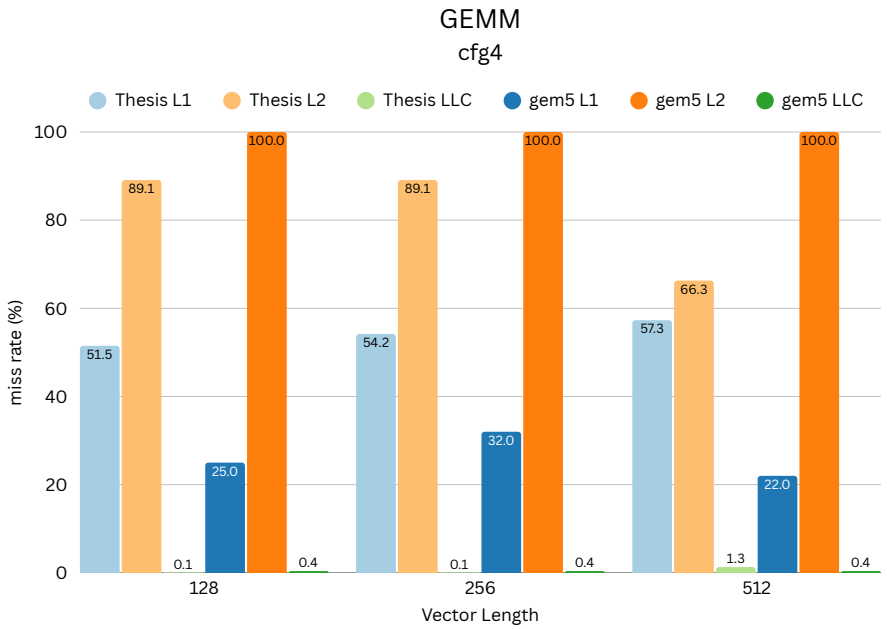


Figure 4.8: GEMM cfg4 cache-misses comparison with gem5(state-of-the-art)

Observing figure-4.8, the estimated L1 miss rate seems to be almost double that of gem5, while the L2 miss rates match within 10% error compared to gem5 except for VL=512 where thesis-L2 shows 35% error. Surprisingly, only for this configuration (cfg4), thesis-LLC matches very well with gem5. A general conclusion is drawn in section-4.4.2

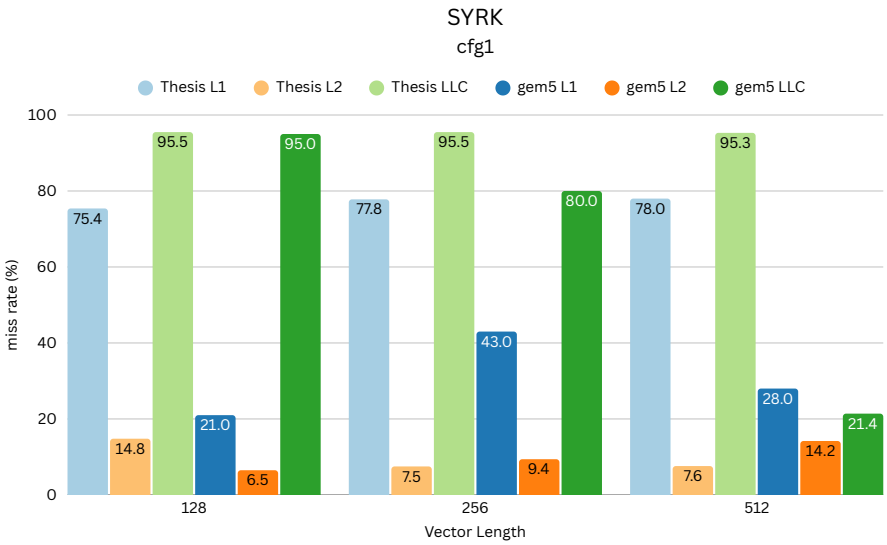


Figure 4.9: SYRK cfg1 cache-misses comparison with gem5(state-of-the-art)

Observing figure-4.9, the estimated thesis-L1 miss rate remains steady while gem5 almost doubles from VL=128 to VL=256 and again drops back to inital value for VL=512. The L2 and LLC miss rates match well between this thesis and gem5 except for VL=512 where gem5 -LLC shows strong reduction while thesis-LLC remains the same as other VL. It is hard to draw a strong conclusion and a general conclusion is drawn in section-4.4.2

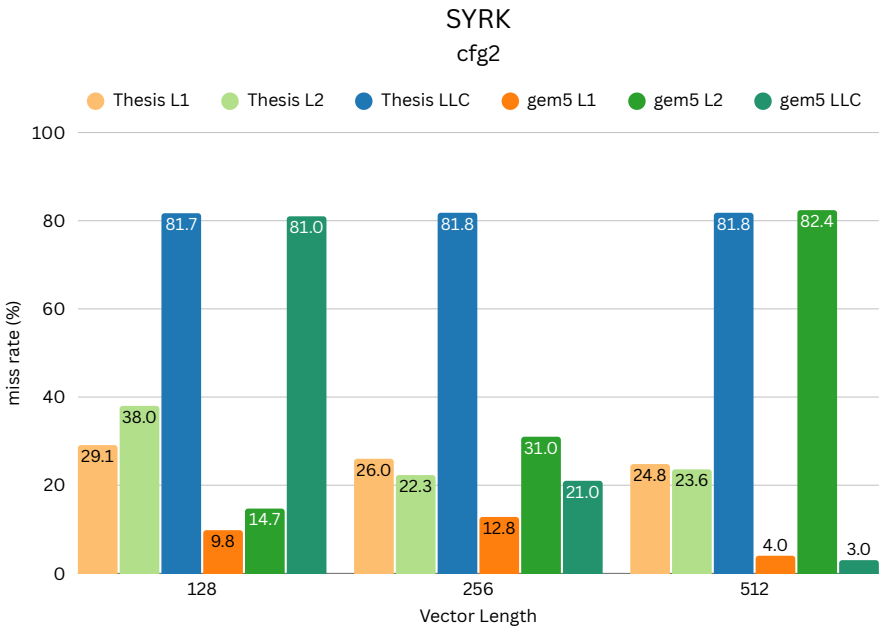


Figure 4.10: SYRK cfg2 cache-misses comparison with gem5(state-of-the-art)

Observing figure-4.10, the estimated thesis-L1 miss rate remains steady while gem5 estimates fluctuate slightly. For L2 misses, thesis-L2 slightly decreases with increasing VL but gem5-L2 doubles with increasing VL. On the contrary for LLC misses, thesis-LLC remains steady but gem5-LLC drops with increasing VL. Without further analysis, it is hard to draw a strong conclusion and so, a general conclusion is drawn in section-4.4.2

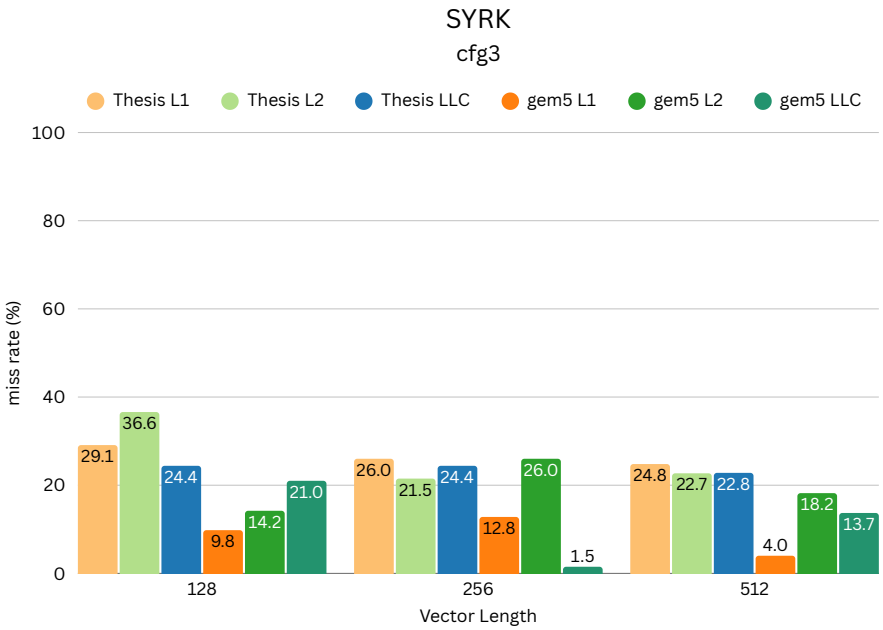


Figure 4.11: SYRK cfg3 cache-misses comparison with gem5(state-of-the-art)

Observing figure-4.11, the estimated thesis-L1 miss rate remains steady while the gem5 estimates fluctuate slightly. For L2 misses, thesis-L2 slightly decreases with increasing VL and gem5-L2 almost doubles from VL=128 to VL=256 and again drops back to inital value for VL=512. For LLC misses, thesis-LLC remains steady but gem5-LLC fluctuates with increasing VL. Without further analysis, it is hard to draw a strong conclusion and so, a general conclusion is drawn in section-4.4.2

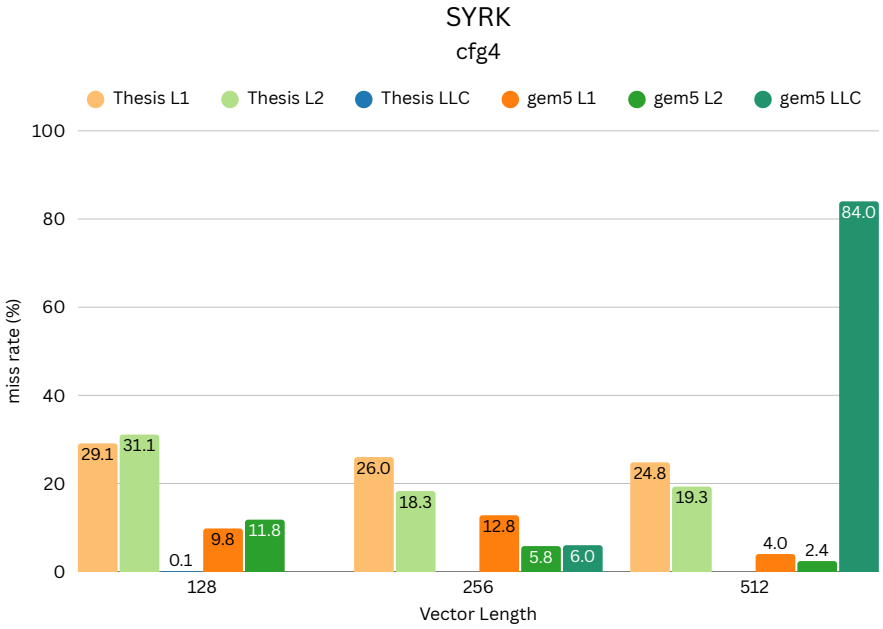


Figure 4.12: SYRK cfg4 cache-misses comparison with gem5(state-of-the-art)

Observing figure-4.12, the estimated thesis-L1 miss rate remains steady while gem5 estimates fluctuate slightly. For L2 misses, thesis-L2 and gem5-L2 decreases with increasing VL. For LLC misses, thesis-LLC remains steady but gem5-LLC increases

with increasing VL.

Observing figure-4.13,4.14,4.15,4.16, the gem5 misses remain constant with increasing VL, and the thesis estimated misses fluctuate slightly. A general conclusion is drawn in section-4.4.2.

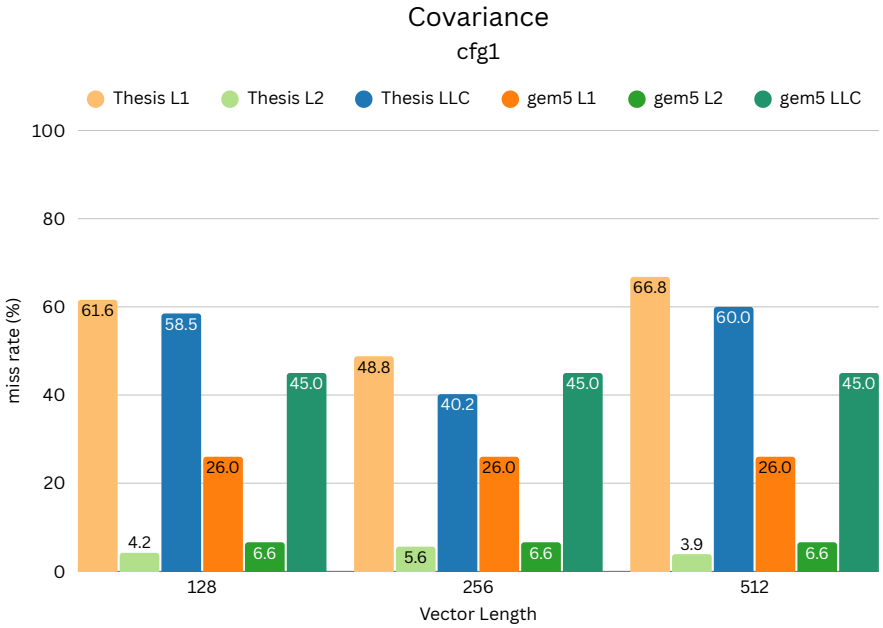


Figure 4.13: Covariance cfg1 cache-misses comparison with gem5(state-of-the-art)

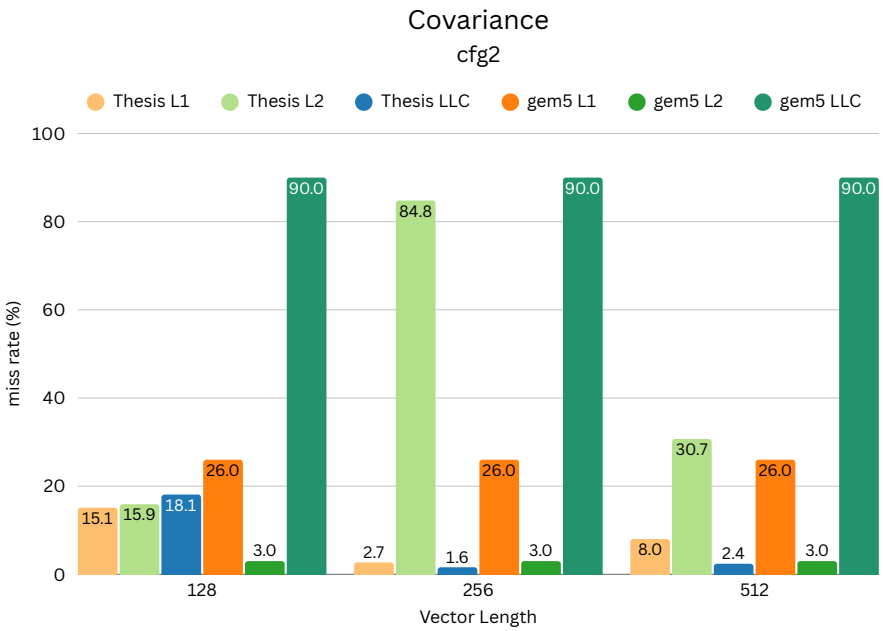


Figure 4.14: Covariance cfg2 cache-misses comparison with gem5(state-of-the-art)

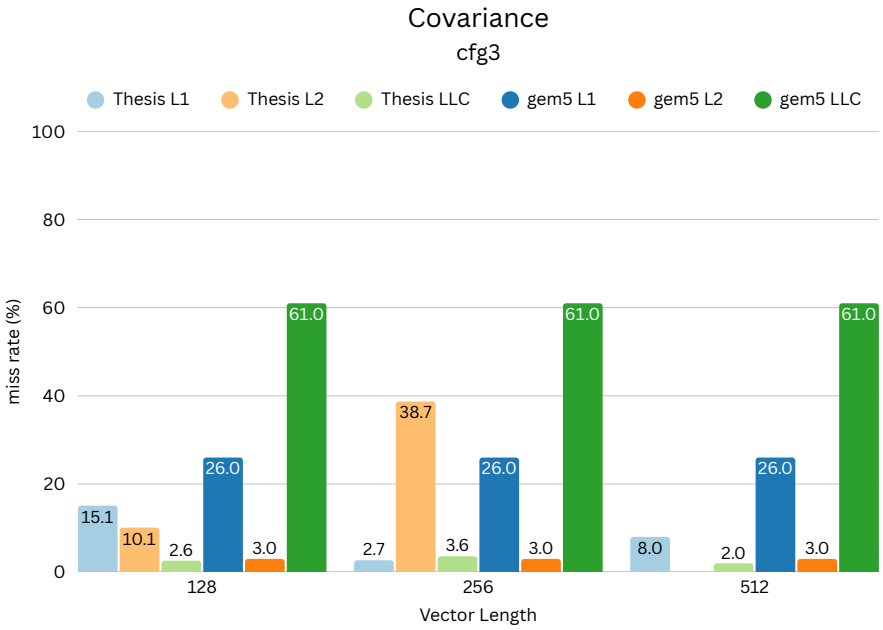


Figure 4.15: Covariance cfg3 cache-misses comparison with gem5(state-of-the-art)

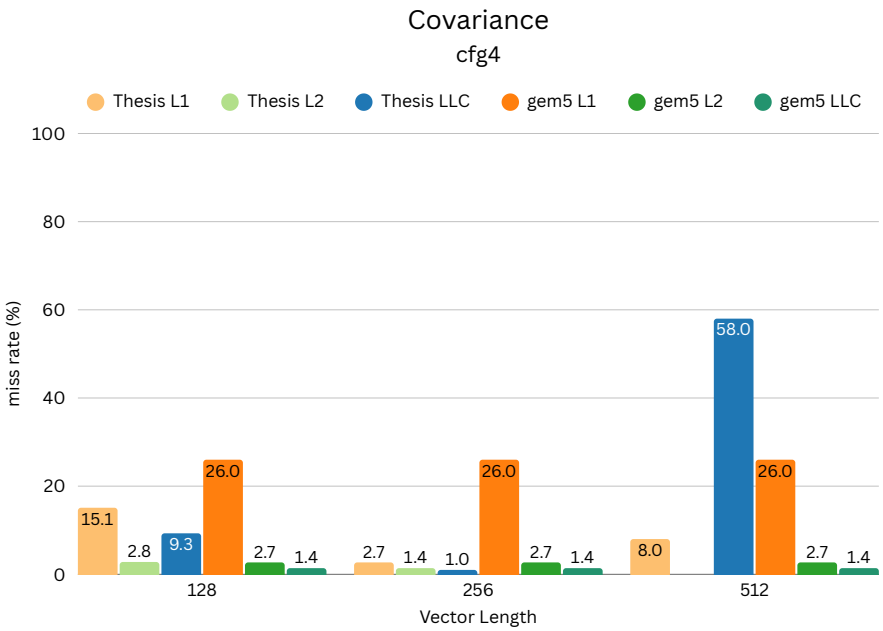


Figure 4.16: Covariance cfg4 cache-misses comparison with gem5(state-of-the-art)

Observing figures-4.17, 4.18, 4.19, 4.20, miss rates remains constant for increasing VL for each configuration. A general conclusion is drawn in section-4.4.2.

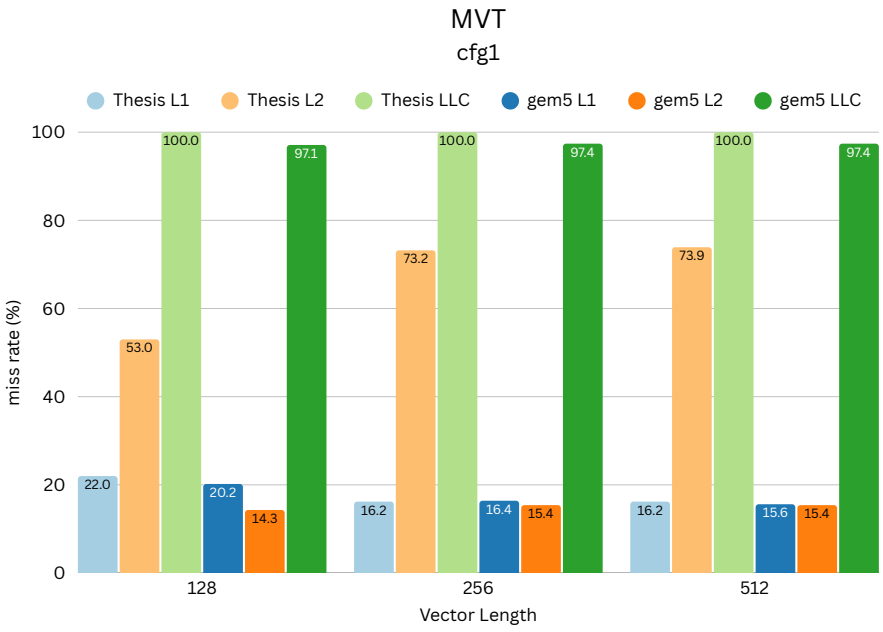


Figure 4.17: MVT cfg1 cache-misses comparison with gem5(state-of-the-art)

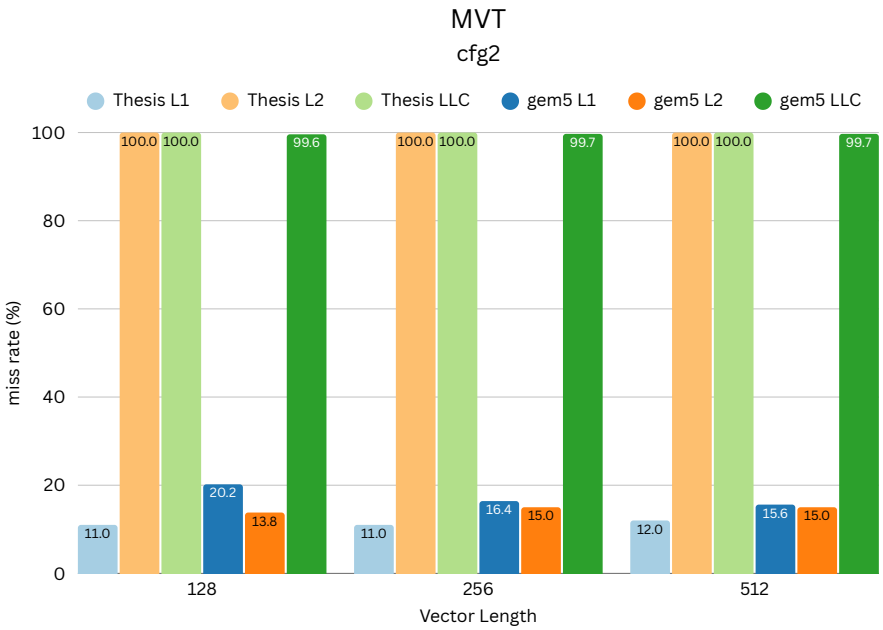


Figure 4.18: MVT cfg2 cache-misses comparison with gem5(state-of-the-art)

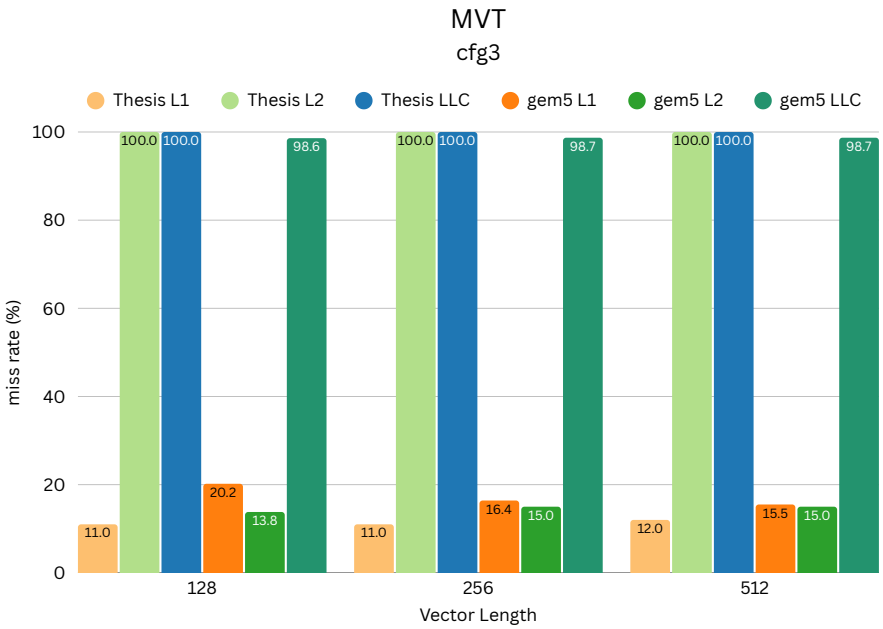


Figure 4.19: MVT cfg3 cache-misses comparison with gem5(state-of-the-art)

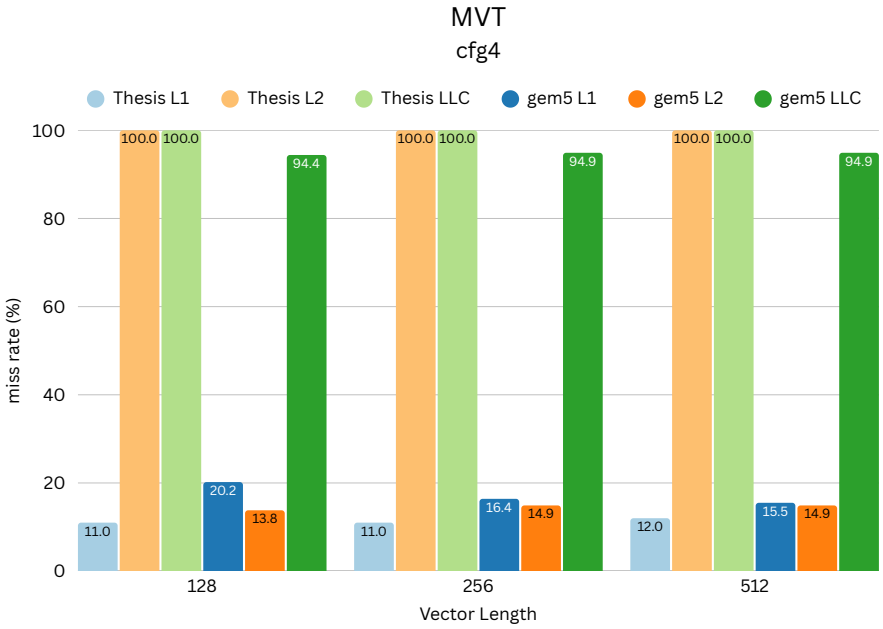


Figure 4.20: MVT cfg4 cache-misses comparison with gem5(state-of-the-art)

4.2.3 Cache misses classification (GEMM)

This section highlights the classification of cache misses (miss rate on Y axis) calculated from reuse distance histogram for each basic block (X axis). For simplicity, only GEMM benchmark having vector length of 512*b* with cache configuration `cfg2` shown in Table-4.1 is considered as an example. Some conflict and capacity misses can be observed for specific basic blocks as shown in figures 4.21, 4.22 and 4.23. While the classification is highly useful, further analysis is needed for better understanding.

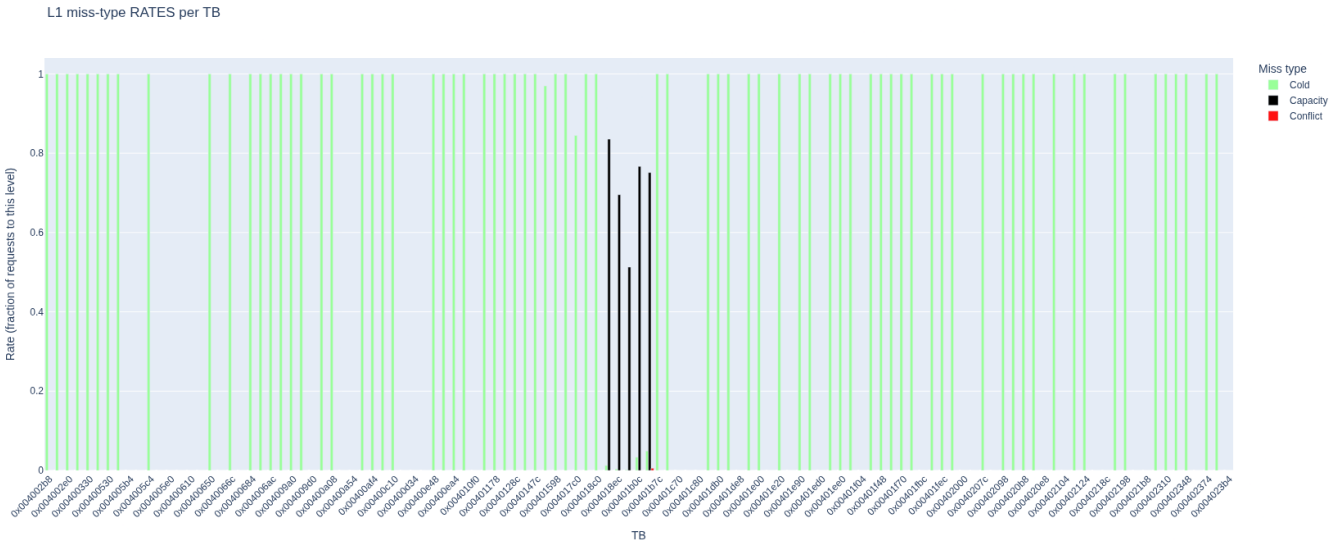


Figure 4.21: Cold, Capacity, Conflict misses classification for L1 cache cfg2 of GEMM VL=512

Figure 4.22: Cold, Capacity, Conflict misses classification for L2 cache cfg2 of GEMM VL=512

Figure 4.23: Cold, Capacity, Conflict misses classification for LLC cache cfg2 of GEMM VL=512

4.2.4 Lines-Per-Instruction, MSHR stalls metric (GEMM)

Figures 4.24, 4.25 highlights the top 10 memory intense basic blocks for particular levels of the cache. Their values needs to be validated for further use which can be considered as future work.

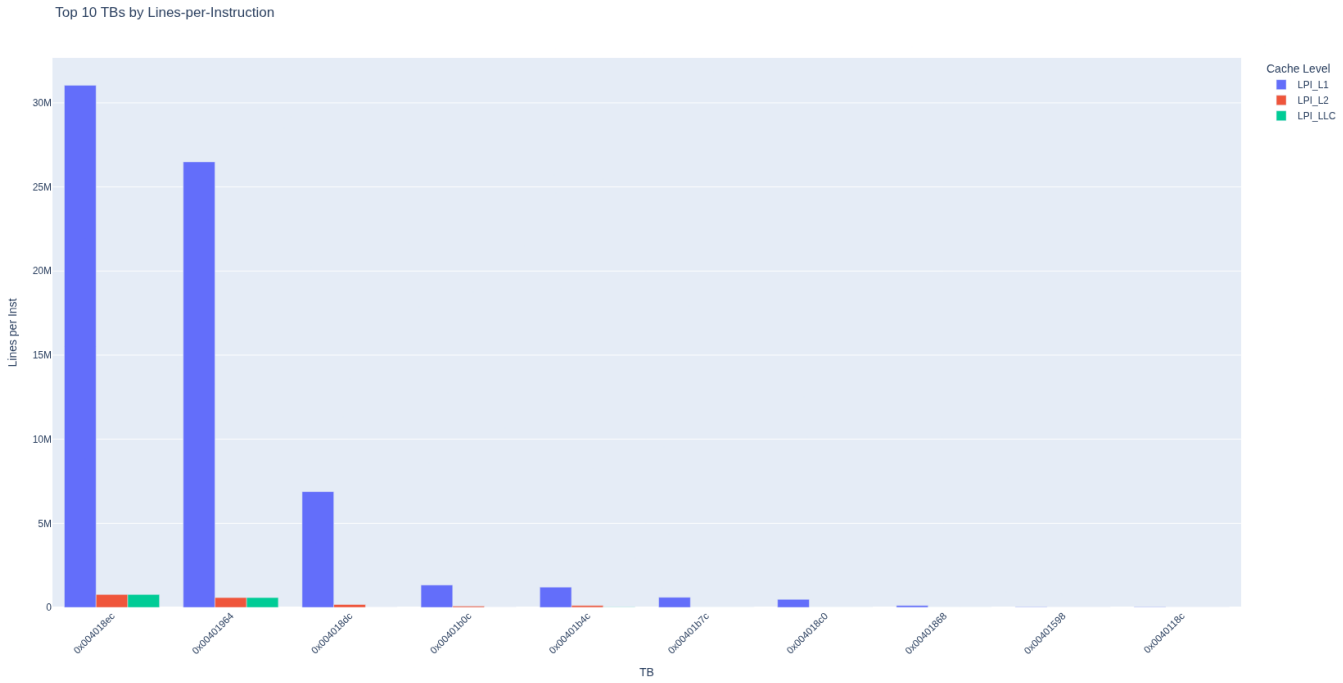


Figure 4.24: Lines Per Instruction metric for GEMM cfg2 with VL=512

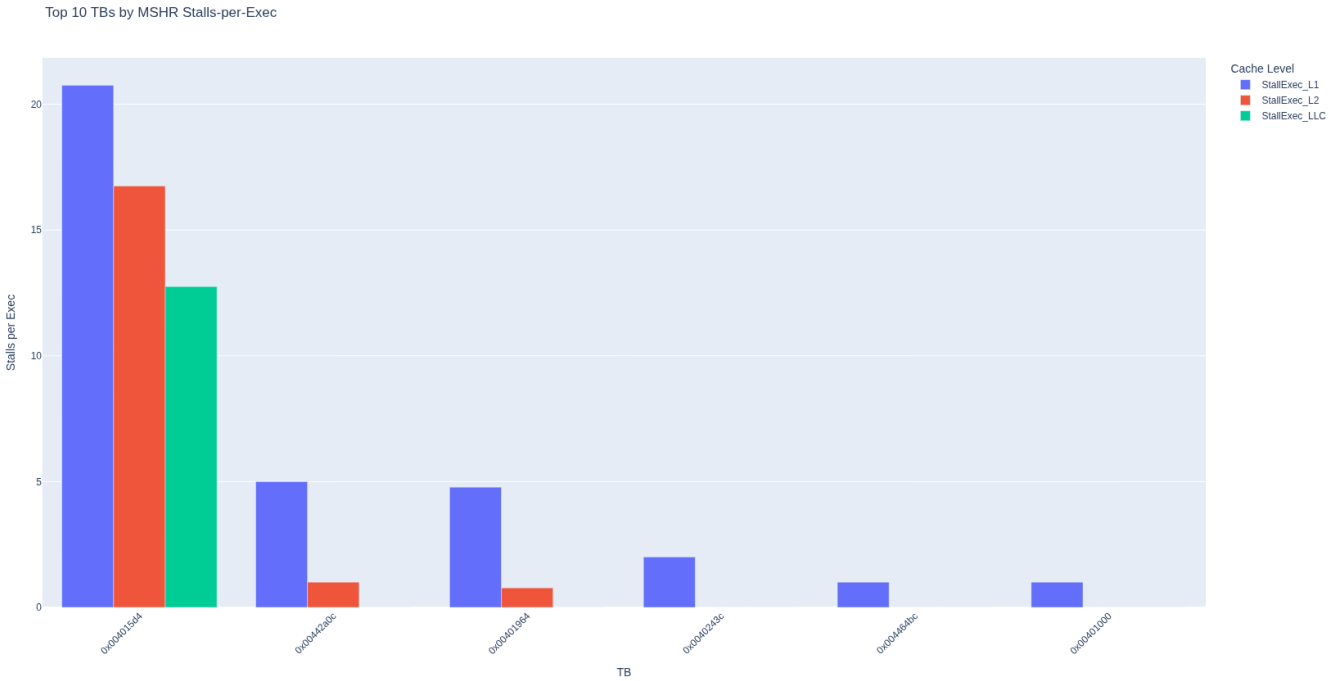


Figure 4.25: MSHR stalls metric for GEMM cfg2 with VL=512

4.3 Tabulation

Here the graphs are shown as absolute values just for easy reference. A general conclusion for all the values present in the tables are drawn in section-4.4

4.3.1 IPC calculation results

benchmark	type	Vector length	thesis IPC (llvm-mca)	thesis IPC (basic)	gem5 ArmO3CPU IPC	gem5 ArmMinorCPU IPC
GEMM	compute bound	128	3.6842	0.292	0.113	0.0277
GEMM		256	1.3808	0.193	0.074	0.0187
GEMM		512	2.8980	0.225	0.060	0.0238
SYRK	compute bound	128	1.89	0.315	[0.25, 0.41, 0.48, 0.53]	[0.191, 0.234, 0.256, 0.269]
SYRK		256	1.35	0.223	[0.20, 0.44, 0.51, 0.74]	[0.046, 0.095, 0.109 , 0.146]
SYRK		512	1.57	0.215	[0.25, 0.63, 1.10, 1.38]	[0.058, 0.135, 0.215, 0.252]
covariance	memory bound	128	1.94	0.314	[0.19, 0.21, 0.22, 0.23]	[0.165, 0.180, 0.184, 0.193]
covariance		256	1.89	0.209	[0.19, 0.21, 0.22, 0.23]	[0.165, 0.180, 0.184, 0.193]
covariance		512	1.04	0.209	[0.19, 0.21, 0.22, 0.23]	[0.165, 0.180, 0.184, 0.193]
mvt	memory bound	128	1.07	0.277	0.318	0.125
mvt		256	1.95	0.196	0.278	0.113
mvt		512	0.93	0.192	0.257	0.122

Table 4.4: IPC comparison with state-of-the-art (gem5)

4.3.2 Cache misses estimation results

cache-config	Vector-length	Thesis miss-rate (L1, L2, LLC)	gem5 miss-rate (L1, L2, LLC)
cfg1	128	0.515, 0.955, 0.932	0.24, 0.99, 1
cfg2		0.515, 0.891, 1	0.24, 0.99, 0.99
cfg3		0.515, 0.891, 1	0.24, 0.99, 0.99
cfg4		0.515, 0.891, 0.001	0.25, 1, 0.004
cfg1	256	0.543, 0.955, 0.932	0.31, 1, 1
cfg2		0.542, 0.891, 1	0.31, 1, 1
cfg3		0.542, 0.891, 1	0.31, 1, 1
cfg4		0.542, 0.891, 0.001	0.32, 1, 0.004
cfg1	512	0.573, 0.967, 0.937	0.22, 1, 1
cfg2		0.573, 0.907, 0.731	0.22, 1, 1
cfg3		0.573, 0.907, 0.001	0.22, 1, 1
cfg4		0.573, 0.663, 0.0013	0.22, 1, 0.004

Table 4.5: GEMM (compute-bound) benchmark cache miss-rate comparison with state-of-the-art(gem5)

cache-config	Vector-length	Thesis miss-rate (L1, L2, LLC)	gem5 miss-rate (L1, L2, LLC)
cfg1	128	0.754, 0.148, 0.955	0.21, 0.065, 0.95
cfg2		0.291, 0.380, 0.817	0.098, 0.147, 0.81
cfg3		0.291, 0.366, 0.244	0.098, 0.142, 0.21
cfg4		0.291, 0.311, 0.0011	0.098, 0.118, 0.006
cfg1	256	0.778, 0.075, 0.955	0.43, 0.094, 0.80
cfg2		0.260, 0.223, 0.818	0.128, 0.31, 0.21
cfg3		0.260, 0.215, 0.244	0.128, 0.26, 0.015
cfg4		0.260, 0.183, 0.0.001	0.128, 0.058, 0.06
cfg1	512	0.78, 0.076, 0.953	0.28, 0.142, 0.214
cfg2		0.248, 0.236, 0.818	0.04, 0.824, 0.030
cfg3		0.248, 0.227, 0.228	0.04, 0.182, 0.137
cfg4		0.248, 0.193, 0.0017	0.04, 0.024, 0.840

Table 4.6: SYRK (compute-bound) benchmark cache miss-rate comparison with state-of-the-art(gem5)

cache-config	Vector-length	Thesis miss-rate (L1, L2, LLC)	gem5 miss-rate (L1, L2, LLC)
cfg1	128	0.616, 0.042, 0.585	0.26, 0.066, 0.45
cfg2		0.151, 0.159, 0.181	0.26, 0.030, 0.90
cfg3		0.151, 0.101, 0.026	0.26, 0.03, 0.61
cfg4		0.151, 0.028, 0.093	0.26, 0.027, 0.014
cfg1	256	0.488, 0.056, 0.402	0.026, 0.066, 0.45
cfg2		0.027, 0.848, 0.016	0.26, 0.030, 0.90
cfg3		0.027, 0.387, 0.036	0.26, 0.03, 0.61
cfg4		0.027, 0.014, 1	0.26, 0.027, 0.014
cfg1	512	0.668, 0.039, 0.60	0.026, 0.066, 0.45
cfg2		0.08, 0.307, 0.024	0.26, 0.030, 0.90
cfg3		0.08, 0.2, 0.02	0.26, 0.03, 0.61
cfg4		0.08, 0.0007, 0.58	0.26, 0.027, 0.014

Table 4.7: Covariance (memory-bound) benchmark cache miss-rate comparison with state-of-the-art(gem5)

cache-config	Vector-length	Thesis miss-rate (L1, L2, LLC)	gem5 miss-rate (L1, L2, LLC)
cfg1	128	0.22, 0.53, 1	0.202, 0.143, 0.971
cfg2		0.11, 1, 1	0.202, 0.138, 0.996
cfg3		0.11, 1, 1	0.202, 0.138, 0.986
cfg4		0.11, 1, 1	0.202, 0.138, 0.944
cfg1	256	0.162, 0.732, 1	0.164, 0.154, 0.974
cfg2		0.11, 1, 1	0.164, 0.150, 0.997
cfg3		0.11, 1, 1	0.164, 0.150, 0.987
cfg4		0.11, 1, 1	0.164, 0.149, 0.949
cfg1	512	0.162, 0.739, 1	0.156, 0.154, 0.974
cfg2		0.12, 1, 1	0.156, 0.150, 0.997
cfg3		0.12, 1, 1	0.155, 0.150, 0.987
cfg4		0.12, 1, 1	0.155, 0.149, 0.949

Table 4.8: MVT (memory-bound) benchmark cache miss-rate comparison with state-of-the-art(gem5)

4.4 Discussion

This section tries to put together the observations made by analyzing the graphs. The subsequent section lists other tools that were tried but did not provide presentable results for the conclusion of this work.

4.4.1 IPC calculations

Observing the graphs for GEMM in Figure-4.1, for SYRK in Figure-4.2, for Covariance in Figure-4.3, for MVT in Figure-4.2, it is apparent that changing vector length did not significantly improve the IPC due to following reasons:

- `qemu-aarch64` and `llvm-mca` does not model micro-architectural vector pipelines. They are only functional models and they work at the Instruction Set Architecture (ISA) Level only. This makes the IPC values depend on Instruction Count alone and not the data-path width.
- Even though vectorization has been verified for all of the above benchmarks (steps highlighted in section-3.2), the `Clang-18` auto-vectorization is very poor.
- Since the tools developed in this thesis fundamentally depend on `qemu-aarch64` and `llvm-mca`, it merely reflects compiler-level limitations rather than micro-architectural performance.
- The `thesis-IPC-basic` method described in 3.4.1 strongly correlates with gem5 `ArmMinorCPU` whereas the `thesis-IPC-llvm-mca` method described in 3.4.1.1 weakly correlates with gem5 `ArmO3CPU`. This is because `llvm-mca` method tries to parallelize and run every basic block in isolation which is not the best approach but since `llvm-mca` do not support batch processing, no better way could be found at the moment. For my basic-ipc approach, most of the static latencies are assumed.

4.4.2 Cache misses estimations

Observing the miss-rates for different cache configurations for GEMM in Figures 4.5, 4.6, 4.7, 4.8, for SYRK in Figures 4.9, 4.10, 4.11, 4.12 , for Covariance in Figures 4.13, 4.14, 4.15, 4.16, for MVT in Figures 4.17, 4.18, 4.19, 4.20, it is apparent that changing vector length did not significantly reduce cache misses due to following reasons:

- The estimates for compute-bound workloads strongly correlate with gem5 whereas the memory-bound workloads weakly correlates with gem5.
- The classification of cache-misses (cold, capacity, conflict) have no references to cross-check and the classification helps to analyze workloads effectively.
- The reuse-distance approach is made for fully-associative caches. Considering set-associative caches might be less realistic.
- The reuse-distance histograms are insensitive to cache sizes in this range.

- Cache misses depend on memory access pattern of the workload and the selected workloads with the problem sizes might have been poorly auto-vectorized by compiler.
- Memory-Level-Parallelism is not considered.
- Approximations from SHARDS [20] might have introduced more errors.

4.4.3 MSHR stalls and Bandwidth contention detection

The newly introduced metrics: **Lines-Per-Instruction, Misses-Per-Execution** have no good references from any other tool which can be used to validate their results leading to future work.

4.5 Other tools tried

Before finalizing the use of qemu-TCG-plugins , vast number of tools were tried and each of them had one or the other drawbacks as explained below:

1. renode [23] and trace-based-model: performance numbers seemed inaccurate, trace-based-model was developed using python and running a hello-world took more time than the actual gem5 hello-world simulation.
2. spike and trace-based-model: additionally needs proxy-kernel which slows down the whole process even further.
3. DynamoRIO cross-compiled to 64bit RISC-V running on qemu-riscv64: Crashed and could not debug or draw conclusions.
4. DynamoRIO cross-compiled to ARM aarch64 running on qemu-aarch64: sporadic crashes, cannot change vector length. Upon asking this in public google groups, no responses were received leading to discontinuation of this work.
5. tried to come up with custom analytical model: This concept now manifests as the basic ipc-calculation model described in 3.4.1 but could not arrive at scatter-gather related operations. So, this can be considered as partial success.
6. LLVM-MCA alone: drawbacks: infinite L1 cache, control flow instructions are not considered, loops are executed only once as it doesn't support branching.
7. LLVM-MCAD[24]:works for small program sizes and hangs for large programs.
8. ArmIE : this can only be run on hardware containing real ARM CPU.
9. DineroIV[25] tool for cache misses: too slow but very accurate
10. Feeding qemu-aarch64 execution trace to llvm-mca[12]: llvm-mca doesn't do batch-processing, took 50GB of space but still didn't give any output
11. updating LLVM-MCA for batch processing : have to understand llvm internals which is very challenging and forms a separte thesis by itself.

5

Conclusion

This thesis makes an initial attempt to bridge the gap between microarchitecture-dependent vector lengths and access-pattern-dependent cache behavior, with a particular emphasis on achieving faster simulation. Addressing this problem is further challenged by the limited availability of suitable toolchains and infrastructure support. In this context, the thesis represents the humble beginnings of an infrastructure built upon the `qemu-aarch64` plugin framework. The work introduces custom plugins and post-processing tools to estimate key performance metrics such as IPC, supporting early-stage co-design studies for vector architectures. It also integrates the reuse-distance concept within QEMU to estimate cache misses, enabling broader exploration of cache hierarchy configurations. Finally, the thesis proposes preliminary performance metrics aimed at identifying potential bottlenecks and guiding future optimization efforts.

The following conclusions can be drawn for this thesis:

- The methodology by this thesis offers a functional framework for capturing, analyzing, visualizing performance metrics for vector architectures.
- The plugin framework defined by this thesis can be a first step to check if IPC varies by changing vector length, thereby serving as a diagnostic tool rather than a performance model.
- One of the aims of this thesis was to have a simulation speedup of 100x compared to gem5. This is achieved by having a plugin infrastructure working at Instruction Set Architecture (ISA) level. The downside of this approach as observed from the results is that the IPC remains more or less constant which means the vector length changes (or scaling) could not be effectively captured at the ISA level since it is a micro-architecture parameter thereby dropping the accuracy.
- The tradeoff between simulation speed and accuracy should be made by
  - identifying the performance characteristics to be measured (here: vector length, cache hierarchy)
  - identifying the level of abstraction that can effectively capture the variation of this performance characteristic (for e.g. vector length is a micro-architecture parameter hence choose micro-architecture modeling).

  - Doing so, will provide more judgment to make tradeoff between simulation speed and accuracy.

- For cache misses estimations, the speedup of 10x to 50x is achieved by using reuse-distance histogram approach with SHARDS. This amortizes the cost of generating the histogram by providing instant results for any cache hierarchy. The accuracy achieved is a mix as it holds good for some configurations and does not hold good for others. This is tricky to analyze due to presence of the whole hierarchy as the reuse distance is considered for individual caches.

5.1 Limitations and Future work

5.1.1 Limitations

- Vector Lengths : Only 128, 256 and 512 bit are currently supported by `qemu-aarch64`. To try different vector lengths, the internals of `qemu-aarch64` have to be modified.

- IPC Calculation: since `llvm-mca` is provided with multiple basic blocks that are executed in isolation (independently), it cannot re-build the control-flow-graph of the original program execution. This drops the accuracy of the approach. Also, `llvm-mca` does not consider the latency changes that occur with different cache hierarchies as it assumes infinite L1 cache and currently, the latencies are hardcoded inside target CPU specific `*.td` files.

- reuse distance with SHARDS: takes variable amount of time to create the reuse distance histogram and if cache-size is less than SHARDS, then, there are 1 or more lines per SHARD index which introduces more errors in the final cache miss-rate. The reuse-distance approach[19] is primarily applied to Fully-Associative caches but this thesis considers set-associative caches. The whole idea of reuse distance with SHARDS is only an approximate model and it will not model the real hardware behavior.

- Bandwidth contention and MSHR stalls detection: this is hard to validate with currently available tools and there is no straight-forward approach to obtain these insights by profiling a workload.

5.1.2 Future work

- try it for RISC-V. This will help to build up a vector-architecture exploration platform.

- optimize further loop detection further (e.g. different ways to flatten loops, detect patterns at multiple abstraction levels)

- new plugins to detect compiler optimization effects for vectorization. In particular, perform manual vectorization and try to quantify its effectiveness compared to auto-vectorization.

- Improvise IPC calculation by taking into account the memory latency, memory-level-parallelism etc..
- Add support to `llvm-mca` for batch-processing or processing of partial results (after analyzing dependencies between instructions).
- Methods to validate proposed metrics: Lines-Per-Instructions, Misses-Per-Execution.
- Run more benchmarks and try to analyse results.

Bibliography

[1] N. Binkert, B. Beckmann, G. Black, *et al.*, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, 2011. DOI: 10.1145/2024716.2024718.

[2] J. Domke, E. Vatai, B. Gerofi, *et al.*, "At the locus of performance: Quantifying the effects of copious 3d-stacked cache on hpc workloads," *ACM Trans. Archit. Code Optim.*, 2023. DOI: 10.1145/3629520.

[3] J. Lowe-Power, "Gem5 horrors and what we can do about it," in *Second gem5 User Workshop with ISCA*, 2015.

[4] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 779–788. DOI: 10.1109/CVPR.2016.91.

[5] J. Redmon and A. Farhadi, "Yolov3: An incremental improvement," *arXiv preprint arXiv:1804.02767*, 2018.

[6] J. Lowe-Power, A. M. Ahmad, A. Akram, *et al.*, "The gem5 simulator: Version 20.0+," *CoRR*, 2020. arXiv: 2007.03152.

[7] S. Williams, A. Waterman, and D. Patterson, "Roofline: An insightful visual performance model for multicore architectures," *Commun. ACM*, vol. 52, no. 4, pp. 65–76, 2009. DOI: 10.1145/1498765.1498785.

[8] A. Ilic, F. Pratas, and L. Sousa, "Cache-aware roofline model: Upgrading the loft," *IEEE Computer Architecture Letters*, vol. 13, no. 1, pp. 21–24, 2014. DOI: 10.1109/L-CA.2013.6.

[9] University of Bristol HPC Group, *Simeng*, https://github.com/UoB-HPC/SimEng, 2023.

[10] T. E. Carlson, W. Heirman, and L. Eeckhout, "Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulations," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2011, 52:1–52:12. [Online]. Available: https://dl.acm.org/doi/10.1145/2063384.2063454.

[11] A. F. Rodrigues, R. C. Murphy, P. Kogge, and K. D. Underwood, "The structural simulation toolkit: Exploring novel architectures," in *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing (SC '06)*, New York, NY, USA: ACM, 2006, 157–es. DOI: 10.1145/1188455.1188618.

[12] LLVM Project, *Llvm-mca: Machine code analyzer*, https://llvm.org/docs/CommandGuide/llvm-mca.html.

[13] T. Grass, C. Allande, A. Armejach, *et al.*, "Musa: A multi-level simulation approach for next-generation hpc machines," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '16)*, Salt Lake City, Utah: IEEE Press, 2016, ISBN: 9781467388153.

[14] RISC-V Foundation, *Spike: A risc-v isa simulator*, https://github.com/riscv-software-src/riscv-isa-sim, 2016.

[15] M. Reisinger, *Polybenchc 4.2.1*, https://github.com/MatthiasJReisinger/PolyBenchC-4.2.1.

[16] H. Abdi and L. J. Williams, "Principal component analysis," *WIREs Comput. Stat.*, vol. 2, no. 4, pp. 433–459, Jul. 2010, ISSN: 1939-5108. DOI: 10.1002/wics.101. [Online]. Available: https://doi.org/10.1002/wics.101.

[17] N. Stephens, S. Biles, M. Boettcher, *et al.*, "The arm scalable vector extension," *IEEE Micro*, vol. 37, no. 2, pp. 26–39, 2017. DOI: 10.1109/MM.2017.35.

[18] *Rolling hash technique*, https://en.wikipedia.org/wiki/Rolling_hash.

[19] E. Berg and E. Hagersten, "Statcache: A probabilistic approach to efficient and accurate data locality analysis," in *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2004, pp. 20–27. DOI: 10.1109/ISPASS.2004.1291352.

[20] C. A. Waldspurger, N. Park, A. Garthwaite, and I. Ahmad, "Efficient mrc construction with shards," in *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, ser. FAST'15, Santa Clara, CA: USENIX Association, 2015, pp. 95–110. [Online]. Available: https://dl.acm.org/doi/10.5555/2750482.2750490.

[21] S. Matsuoka, "Fugaku and a64fx: The first exascale supercomputer and its innovative arm cpu," in *2021 Symposium on VLSI Circuits*, 2021, pp. 1–3. DOI: 10.23919/VLSICircuits52068.2021.9492415.

[22] *Fujitsu a64fx microarchitecture manual*, https://www.stonybrook.edu/commcms/ookami/support/_docs/A64FX_Microarchitecture_Manual_en_1.3.pdf.

[23] AntMicro, *Renode: Co-design using trace-based simulation*, https://antmicro.com/blog/2023/07/risc-v-co-design-using-trace-based-simulation-with-renode-and-tbm/.

[24] M.-Y. ". Hsu, D. Gens, and M. Franz. "Mca daemon: Hybrid throughput analysis beyond basic blocks." (), [Online]. Available: https://llvm.org/devmtg/2022-05/slides/2022EuroLLVM-MCA-Deamon.pdf.

[25] J. Edler and M. D. Hill, *Dinero iv trace-driven uniprocessor cache simulator*, http://www.cs.wisc.edu/~markhill/DineroIV/.