

## Exercise 1:

### Software Optimizations

Starting from Exercise 2 of Lab 4, you are required to further speedup the benchmark (*my\_c\_benchmark*) .

For readability, provide the previously used configurations (Cut & Paste).

Parameters	Configuration 1	Configuration 2	Configuration 4	Configuration 5
<b>First changed parameter</b>	the_cpu.fetchWidth = 8	the_cpu.fetchWidth = 2	tag_latency = 1	tag_latency = 1
<b>Second changed parameter</b>	the_cpu.fetchBufferSize = 16		data_latency = 1	tgts_per_mschr = 5
<b>Third changed parameter</b>	the_cpu.fetchQueueSize = 32		response_latency = 1	
	the_cpu.decodeWidth = 4			

Original CPI (no hardware optimization): 2.083

	Configuration 1	Configuration 2	Configuration 4	Configuration 5
<b>CPI</b>	2.081	2.045	1.905	2.038
<b>Speedup (wrt Original CPI)</b>	1.001	1.019	1.094	1.022

Despite the hardware enhancements for increasing the CPU performance, remember that optimizing compilers for programs in high-level code also exist. The aim of optimizing compilers is to minimize or maximize some attributes of an executable computer program (code size, performance, etc.). They are also aware of hardware enhancements to perform very accurate optimizations.

Compilers can be your best friend (or worst enemy!). The more information you provide in your program, the better the optimized program will be.

You can compile your programs with different SW optimization strategies and/or additional features. In the *setup\_default* file:

```

ase_riscv_gem5_sim > $ setup_default
5
6 ##### CROSS COMPILER RISCV #####
7 #####   CROSS COMPILER RISCV   #####
8 #####
9 export CC="/mnt/d/gem5_simulator/riscv_toolchain/bin/riscv64-unknown-elf-gcc"
10 export CC_INSTALLATION_PATH="/mnt/d/gem5_simulator/riscv_toolchain/"
11 ## optimization flags for the compiler
12 export OPTIMIZATION_FLAGS="-O0 "
13

```

You can change the line 12.

Simulate the program for different optimization levels and collect statistics. You are required to change the OPTIMIZATION\_FLAGS variable in the *setup\_default*. O0 is the default value, you need to change the optimization value accordingly to the values in parenthesis in the following Table.

#### DO NOT CONFUSE -O3 WITH O3 PROCESSOR.

TABLE1: IPC for different compiler optimization levels and configurations

Optimization Configuration	Opt lvl 0 (-O0)	Opt lvl 1 (-O1)	Opt lvl 2 (-O2)	Opt size (-Os)	Opt lvl 3 (-O3)	Opt lvl 2 (-O2 --fast- math)
Original Configuration	0.480	0.396	0.444	0.415	0.444	0.459
Configuration 1	0.499	0.419	0.447	0.417	0.447	0.460
Configuration 2	0.480	0.397	0.443	0.414	0.443	0.459
Configuration 4	0.525	0.442	0.496	0.450	0.496	0.520
Configuration 5	0.480	0.414	0.447	0.416	0.447	0.460
Program Size [Bytes]	3228	3044	3032	3016	3032	3032

Regarding the Program Size (Code and Data!!), you can retrieve the size from:

```

~/ase_riscv_gem5_sim$ /opt/riscv-2023.10.18/bin/riscv64-unknown-elf-size -format=gnu
-radix=10 ./programs/my_c_benchmark/my_c_benchmark.elf

```

For brave and curious guys:

For visualize the enabled optimizations from the compiler perspective, you can run:

```

~/my_gem5Dir$ /opt/riscv-2023.10.18/bin/riscv64-unknown-elf-gcc -Q -O2 --help=optimizers

```

By changing the “-O2” parameter with the desired one, you will find the enabled/disabled optimizations.

Here are some possible types of optimizations:

- [https://en.wikipedia.org/wiki/Optimizing\\_compiler](https://en.wikipedia.org/wiki/Optimizing_compiler)
- <https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html>

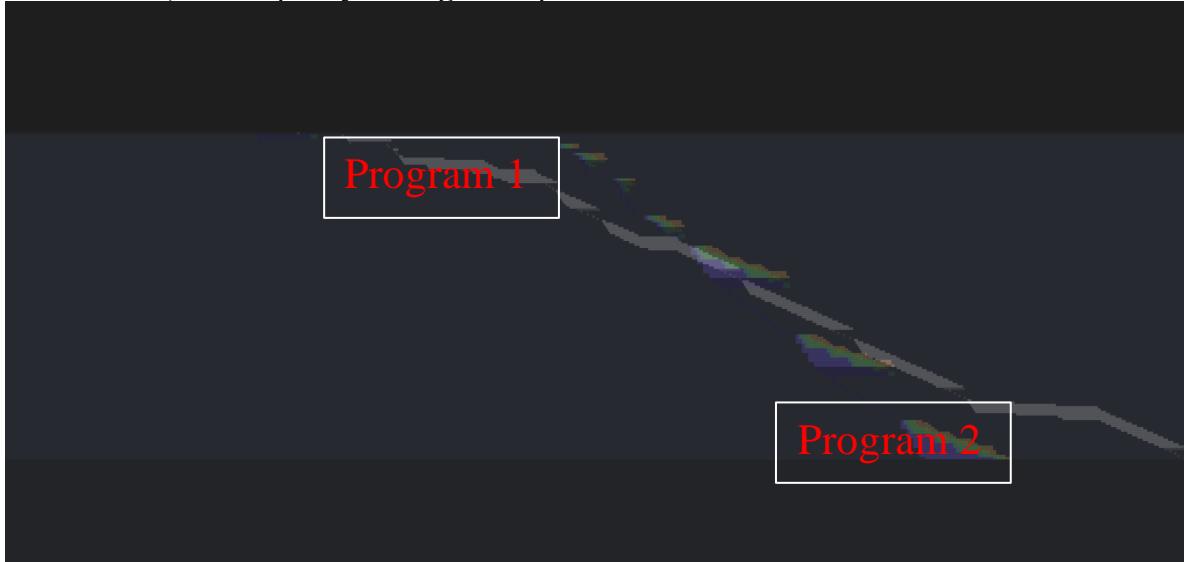
## Exercise 2:

Given your benchmark (`my_c_benchmark.c`), select the best optimization to obtain **your best angle of optimization**, compared to the baseline configuration (`riscv_o3_custom.py; -O0`).

1. Based on Table 1 (from Exercise 1), select the best optimization (for example, the green box corresponding to Configuration 1 with `-O2`).

Optimization Configuration	Opt lvl 0 (-O0)	Opt lvl 1 (-O1)	Opt lvl 2 (-O2)	Opt size (-Os)	Opt lvl 3 (-O3)	Opt lvl 2 (-O2 --fast- math)
Original Configuration	0.480	0.396	0.444	0.415	0.444	0.459
Configuration 1	0.499	0.419	0.447	0.417	0.447	0.460
Configuration 2	0.480	0.397	0.443	0.414	0.443	0.459
Configuration 4	0.525	0.442	0.496	0.450	0.496	0.520
Configuration 5	0.480	0.414	0.447	0.416	0.447	0.460
Program Size [Bytes]	3228	3044	3032	3016	3032	3032

2. By using **Konata**, overlap the two pipelines (the original obtained with `riscv_o3_custom.py` and the optimized corresponding to the best SW-HW combination) to compute your angle of optimization.



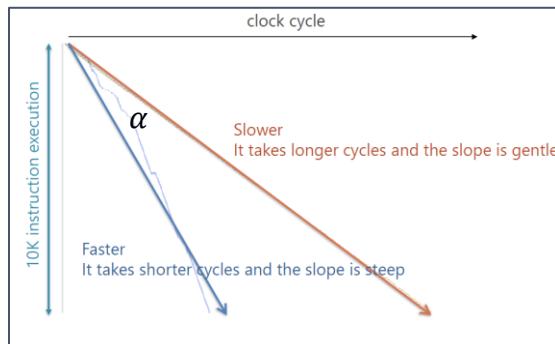
Compute the angle  $\alpha$  (named optimization angle) existing between the traces.

Hint: To load different traces in **Konata**, load them separately. Afterward, right-click in the pipeline visualizer and select “transparent mode”. You need to adjust the scale!



3. To compute the **angle of optimization  $\alpha$** :

$$\alpha = \arctan\left(\frac{\text{Clock Cycles}_{\text{baseline}}}{\text{Instructions}_{\text{baseline}}}\right) - \arctan\left(\frac{\text{Clock Cycles}_{\text{optimized}}}{\text{Instructions}_{\text{optimized}}}\right)$$



The angle of optimization is equal to:  $2.058^\circ$

4. Do you see any visual improvements (for example, a less discontinued pipeline)? Yes, why?  
No, why? What is happening? How they could be improved?

Nella configurazione ottimizzata i cambiamenti sono tutti localizzati nella cache di primo livello; il che significa che tutti i miglioramenti sono portati da minore latenza e tempo di risposta della cache. Avere una cache migliore permette di evitare stalli dovuti a più load consecutive e in generale di velocizzare tutti i processi che la riguardano (ad es. si sa prima se ci sarà un miss e si potrà cercare prima in quella di secondo livello)