Ref. Ares(2022)2413213 - 31/03/2022

D6.2

Simulation infrastructure and early architectural explorations
v1.0

Document information

Work package:	WP6: System Simulation and FPGA Emulation
Contract number:	956702
Project website:	www.eprocessor.eu
Author(s)	Ioannis Mavroidis (EXAP)
Contributors	BSC: Julian Pavon Rivera, Ivan Vargas EXAP: Iakovos Mavroidis, Dimitri Mavroidis
Reviewer(s)	Vassilis Papaefstathiou (FORTH)
Dissemination Level	PU
Nature	R - Report
Contractual deadline:	31/3/2022

This document may contain proprietary material of certain eProcessor contractors. The commercial use of any information contained in this document may require a license from the proprietor of that information.

EuroHPC Joint Undertaking

This project has received funding from the European High-Performance Computing Joint Undertaking Joint Undertaking (JU) under grant agreement No 956702. The JU receives support from the European Union's Horizon 2020 research and innovation programme and Spain, Sweden, Greece, Italy, France, and Germany.

Change Log

Version	Author(s)	Comments and Description of change
0.1	Ioannis Mavroidis (EXAPSYS)	Initial version
0.2	Ioannis Mavroidis (EXAPSYS)	Added FP, speed-ups
0.3	Ioannis Mavroidis (EXAPSYS), Ivan Vargas (BSC)	Updated default parameter values to better match Cortus core. Added vector section.
0.4	Vassilis Papaefstathiou (FORTH)	Reviewed version
1.0	Ioannis Mavroidis (EXAPSYS)	Final version for submission

Contents

Contents 2
1. Summary 3
2. The gem5 simulator 3
3. Simulations for Scalar Core 3
3.1 Benchmark Suite 3
3.2 Methodology 4
3.3 Results 7
3.3.1 L1 Caches 7
3.3.2 L2 Cache 13
3.3.3 L1 and L2 Caches Line Size 16
3.3.4 Integer Unit 18
3.3.5 Floating Point Unit 21
3.3.6 Load-Store Buffers 24
3.3.7 Reorder Buffers 25
3.3.8 Instruction Mix 26
4. Simulation for Vector Accelerators 27
4.1 Methodology 27
4.2 Experimental Results 29
5. Concluding Remarks and Next Steps 30
6. List of Abbreviations 31

1. Summary

This deliverable reports on the results from the first stage of "Task 6.1: Architectural simulations" in the context of "WP6: System Simulation and FPGA Emulation", in which we develop the simulation infrastructure required to drive the architectural design decisions for the eProcessor chip. For this purpose we use the state-of-the-art gem5 simulator and several microbenchmarks from the Bioinformatics and HPC domains.

Part of our work also involves simulation of the RISC-V vector ISA. However, since gem5 does not include a flexible and customizable vector architecture, such as one that could evaluate different implementations including short (around 512-bit), medium (around 4,096-bit), and large (16,384-bit or more) vectors, we extended gem5 by adding our own parameterizable Vector Processing Unit (VPU).

The microbenchmarks are all run in a multitude of hardware configurations, each one varying the size or other features of specific architectural blocks, and examine how each configuration affects the total runtime. Our results will be used in microarchitecture component sizing decisions.

2. The gem5 simulator

For our simulations we use gem5[1]. The gem5 simulator has been established as the de facto simulator used for computer architecture research. It is utilized both in academia and in industry by companies such as ARM, AMD, Google, Micron, HP, and Samsung.

Some of its features are the following:

- Open-source, community-supported
- Event-driven, cycle accurate simulation
- 2 different modes of operation: Full-system emulation, and syscall emulation
- Modular design consisting of components that can be easily parameterized, extended or replaced.
- Models for multiple components: CPUs, caches, memory controllers, buses, etc.
- Multiple CPUs ranging in simulation detail: atomic, timing-simple, in-order, detailed out-of-order
- Support for multiple ISAs: AMD, ARM, SPARC, MIPS, POWER, RISC-V, x86
- Components written in C++ for higher simulation speed, but configurable via Python for greater ease of use.

3. Simulations for Scalar Core

3.1 Benchmark Suite

The Benchmarks that we use for the early architectural explorations of eProcessor, fall into the Bioinformatics and HPC domains. "WP3: Software Applications Use Cases, Specifications and Evaluation" develops and provides the RISC-V assembly code and linker scripts for a total of 93 microbenchmarks. Each microbenchmark implements a function of well-known benchmark suites

[1] https://www.gem5.org/

such as the NAS Parallel Benchmarks, Wavefront alignment algorithm, Smith-Waterman etc. All microbenchmarks start with some initialization code and then run the selected function in an infinite loop. Each microbenchmark comes with a "weight" number signifying its relative importance within the original benchmark. In order to speed up our simulations we use the top 22 microbenchmarks with respect to their "weight". These are shown in the following table:

Domain	Benchmark	Function
Bioinformatics	BSW_B100_L1K_N1K	sw_compute_banded
Bioinformatics	BSW_B10_L100_N10K	sw_compute_banded
Bioinformatics	BSW_B200_L1K_N1K	sw_compute_banded
Bioinformatics	BSW_B20_L100_N10K	sw_compute_banded
Bioinformatics	BSW_B300_L1K_N1K	sw_compute_banded
Bioinformatics	BSW_B30_L100_N10K	sw_compute_banded
Bioinformatics	SW_L100_N10K	sw_compute
Bioinformatics	SW_L1K_N1K	sw_compute
Bioinformatics	WFA_L100K_N100_E10	affine_wavefronts_compute_next
Bioinformatics	WFA_L100K_N100_E1	affine_wavefronts_compute_next
Bioinformatics	WFA_L10K_N1K_E10	affine_wavefronts_compute_next
Bioinformatics	WFA_L10K_N1K_E1	affine_wavefronts_compute_next
Bioinformatics	WFA_L1K_N10K_E10	affine_wavefronts_compute_next
HPC	NAS_CG_S	conj_grad
HPC	NAS_CG_W	conj_grad
HPC	NAS_EP_S	vranlc
HPC	NAS_EP_W	vranlc
HPC	NAS_FT_S	fftz2
HPC	NAS_IS_W	rank
HPC	NAS_MG_W	resid
HPC	NAS_SP_S	adi
HPC	NAS_SP_W	adi

Table 1: Microbenchmarks used

3.2 Methodology

The gem5 simulator can be parameterized with several parameters. In order to study the effect of various architectural features of the RISC-V scalar core, we needed to create a set of gem5 configurations for each such feature, varying the feature-related parameters, while keeping all other parameters at some reasonable default values, matching the currently favored eProcessor architecture as provided by Cortus.

In this way, a total of 101 different gem5 configurations are created and each was run against the 22 microbenchmarks of the previous section, for a total of 2222 simulations.

The following table lists all gem5 parameters that we use, along with their default values.

Parameter name	Default value	Description
--numROBEntries	64	Number of reorder buffer entries
--numPhysIntRegs	128	Number of physical integer registers
--LQEntries	8	Load Queue entries
--SQEntries	8	Store Queue entries
--fetchWidth	4	Fetch width

--fetchBufferSize	32	Fetch buffer size in bytes
--dispatchWidth	4	Dispatch width
--commitWidth	4	Commit width
--wbWidth	6	Writeback width
--decodeWidth	4	Decode width
--issueWidth	4	Issue width
--renameWidth	4	Rename width
--squashWidth	64	Squash width
--cache_line_size	64	Cache line size in bytes
--l1ic_size	16kB	L1 Instr cache size
--l1ic_assoc	4	L1 Instr cache associativity
--l1ic_mshrs	4	L1 Instr cache MSHRs (max outstanding requests)
--l1ic_latency	1	L1 Instr cache latency
--l1dc_size	16kB	L1 Data cache size
--l1dc_assoc	4	L1 Data cache associativity
--l1dc_mshrs	4	L1 Data cache MSHRs (max outstanding requests)
--l1dc_latency	1	L1 Data cache latency
--l2c_size	256kB	L2 cache size
--l2c_assoc	8	L2 cache associativity
--l2c_latency	12	L2 cache latency
--rdwr_count	2	Number of read-write ports
--ALU_count	4	Number of Integer ALUs
--ALU_opLat	1	ALU latency
--ALU_pipelined	True	ALU pipelined
--mult_count	1	Number of Integer Multipliers
--mult_opLat	4	Integer Multiplier latency
--mult_pipelined	True	Integer Multiplier pipelined
--div_count	1	Number of Integer Dividers
--div_opLat	9	Integer Divider latency
--div_pipelined	True	Integer Divider pipelined
--FP_ALU_count	2	Number of FP ALUs
--FP_ALU_opLat	4	FP ALU latency
--FP_ALU_pipelined	True	FP ALU pipelined
--FP_mult_count	1	Number of FP Multipliers
--FP_mult_opLat	8	FP Multiplier latency
--FP_mult_pipelined	True	FP Multiplier pipelined
--FP_div_count	1	Number of FP Dividers
--FP_div_opLat	16	FP Divider latency
--FP_div_pipelined	True	FP Divider pipelined

Table 2: gem5 configuration parameters

Following is an example of the command used to compile one of the microbenchmarks:

```
# riscv64-unknown-linux-gnu-gcc -O0 -O3 -g -no-pie -march=rv64g -
fno-builtin -fno-pic -nostdlib -o
binaries/Bioinformatics.BSW_B100_L1K_N1K.0953_sw_compute_banded_168.
exe -T microbenchmarks-
main/CPU/Bioinformatics/BSW_B100_L1K_N1K/0953_sw_compute_banded_168/
0953_sw_compute_banded_168.lds -T microbenchmarks-
main/CPU/microbenchmark_orignal_env.lds microbenchmarks-
main/CPU/Bioinformatics/BSW_B100_L1K_N1K/0953_sw_compute_banded_168/
0953_sw_compute_banded_168.s
```

And below we show an example of the command that is used to run the resulting executable with gem5 for one of the configurations:

```
# gem5/build/RISCV/gem5.opt --debug-flags=Exec configs/eprocessor.py
--l1ic_assoc=1 --l1dc_assoc=1 --l1ic_size=4kB --l1dc_size=4kB --
l1ic_mshrs=2 --l1dc_mshrs=2 --
cmd=binaries/Bioinformatics.BSW_B100_L1K_N1K.0953_sw_compute_banded_
168.exe > /dev/null 2>/dev/null
```

The gem5 configuration file configs/eprocessor.py seen in the above command, configures a minimal hardware system consisting of one out-of-order single-core CPU of RISC-V ISA, with separate Level 1 instruction and data caches (L1 caches), a unified Level 2 cache (L2 cache), and a 1GB DDR3 main memory. It also configures the system to run in bare-metal, i.e. in syscall emulation mode without any OS code, until a total of 10 million instructions have been committed, for each of the benchmarks that we ran in this study. The selection of 10 million instructions is inline with guidance from the WP3 benchmark developers and is adequate to capture the important activity of the benchmarks and associated cache effects.

At the end of each simulation run, gem5 automatically dumps all recorded statistics into file m5out/stats.txt. The following screenshot shows the beginning of one such m5out/stats.txt file:

```
---------- Begin Simulation Statistics ----------
simSeconds                                   0.003946                       # Number of seconds simulated (Second)
simTicks                                   3945989500                       # Number of ticks simulated (Tick)
finalTick                                  3945989500                       # Number of ticks from beginning of simulation (
simFreq                                  1000000000000                       # The number of ticks per simulated second ((Ti
hostSeconds                                     49.96                       # Real time elapsed on the host (Second)
hostTickRate                                 78987959                       # The number of ticks simulated per host second
hostMemory                                    1163216                       # Number of bytes of host memory used (Byte)
simInsts                                     10000002                       # Number of instructions simulated (Count)
simOps                                       10000002                       # Number of ops (including micro ops) simulated
hostInstRate                                   200167                       # Simulator instruction rate (inst/s) ((Count/Se
hostOpRate                                     200165                       # Simulator op (including micro ops) rate (op/s)
system.clk_domain.clock                           500                       # Clock period in ticks (Tick)
system.clk_domain.voltage_domain.voltage            1                       # Voltage in Volts (Volt)
system.cpu.numCycles                          7891980                       # Number of cpu cycles simulated (Cycle)
system.cpu.numWorkItemsStarted                      0                       # Number of work items this cpu started (Count)
system.cpu.numWorkItemsCompleted                    0                       # Number of work items this cpu completed (Count
system.cpu.instsAdded                        12237149                       # Number of instructions added to the IQ (exclud
system.cpu.instsIssued                       12114724                       # Number of instructions issued (Count)
system.cpu.squashedInstsIssued                  49840                       # Number of squashed instructions issued (Count)
system.cpu.squashedInstsExamined              2237104                       # Number of squashed instructions iterated over
system.cpu.squashedOperandsExamined           1239463                       # Number of squashed operands that are examined
system.cpu.numIssuedDist::samples             7886677                       # Number of insts issued each cycle (Count)
system.cpu.numIssuedDist::mean               1.536100                       # Number of insts issued each cycle (Count)
system.cpu.numIssuedDist::stdev              1.605578                       # Number of insts issued each cycle (Count)
system.cpu.numIssuedDist::underflows                0      0.00%      0.00% # Number of insts issued each cycle (Count)
system.cpu.numIssuedDist::0                   3277376     41.56%     41.56% # Number of insts issued each cycle (Count)
system.cpu.numIssuedDist::1                   1011133     12.82%     54.38% # Number of insts issued each cycle (Count)
system.cpu.numIssuedDist::2                   1307847     16.58%     70.96% # Number of insts issued each cycle (Count)
system.cpu.numIssuedDist::3                   1012350     12.84%     83.80% # Number of insts issued each cycle (Count)
system.cpu.numIssuedDist::4                    983235     12.47%     96.26% # Number of insts issued each cycle (Count)
system.cpu.numIssuedDist::5                    250509      3.18%     99.44% # Number of insts issued each cycle (Count)
system.cpu.numIssuedDist::6                     44227      0.56%    100.00% # Number of insts issued each cycle (Count)
```

Figure 1: Screenshot of gem5 statistics file

All m5out/stats.txt files, from all 2222 simulations, are copied over and kept in a separate folder where they are available for post-processing. A Perl script was then used to post-process these files, extract the values of the statistics we wanted to measure and plot them in the graphs of the following sections.

3.3 Results

In our simulations we study the following architectural blocks: L1 caches, L2 cache, Integer Unit (ALU), FP Unit (ALU, multiplier), Load-Store buffers, and Reorder Buffers. In the following sections we display the configurations we run for each block and the results from the simulations.

3.3.1 L1 Caches

For the L1 caches the parameters that we vary are: associativity, size, and number of MSHRs. Specifically:

- The values that we try for associativity are: 1 (i.e. direct-mapped), 2 (i.e. 2-way set associative), and 4 (i.e. 4-way set associative).
- The values that we try for size are: 4 kBytes, 8 kBytes, and 16 kBytes.
- The values that we try for the number of MSHRs are: 2, 4, and 8.

Taking all different combinations for these values results in a total of 3 x 3 x 3 = 27 configurations for the L1 caches.

We use the following naming convention for these configurations:

l1c_[associativity]_[size]_[# of MSHRs]

For example, configuration l1c_1_8kB_4 represents a L1 cache that is direct-mapped, has a size of 8 kBytes, and has 4 MSHRs. Configuration l1c_4_16kB_2 represents a L1 cache that is 4-way set associative, has a size of 16 kBytes, and has 2 MSHRs.

Each microbenchmark is represented with one colored line in the following figures. Notice that since the figures in this report are restricted by a small width, including in all figures the names of the microbenchmarks corresponding to each colored line would make them unreadable. For this reason, the color-mapping of the microbenchmarks is shown once in the following figure, and all microbenchmark names have been omitted from the remaining figures of this report.

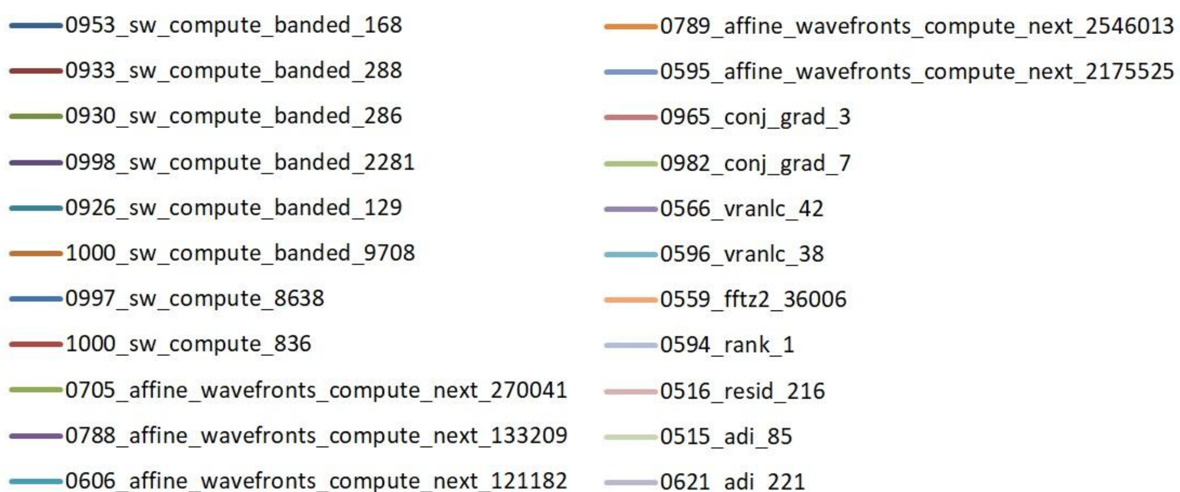

Figure 2: Color-mapping of microbenchmarks (used in all Figures)

The following Figure 3 shows the running time (as expressed in millions of clock cycles), for each microbenchmark across all 27 L1 cache configurations.

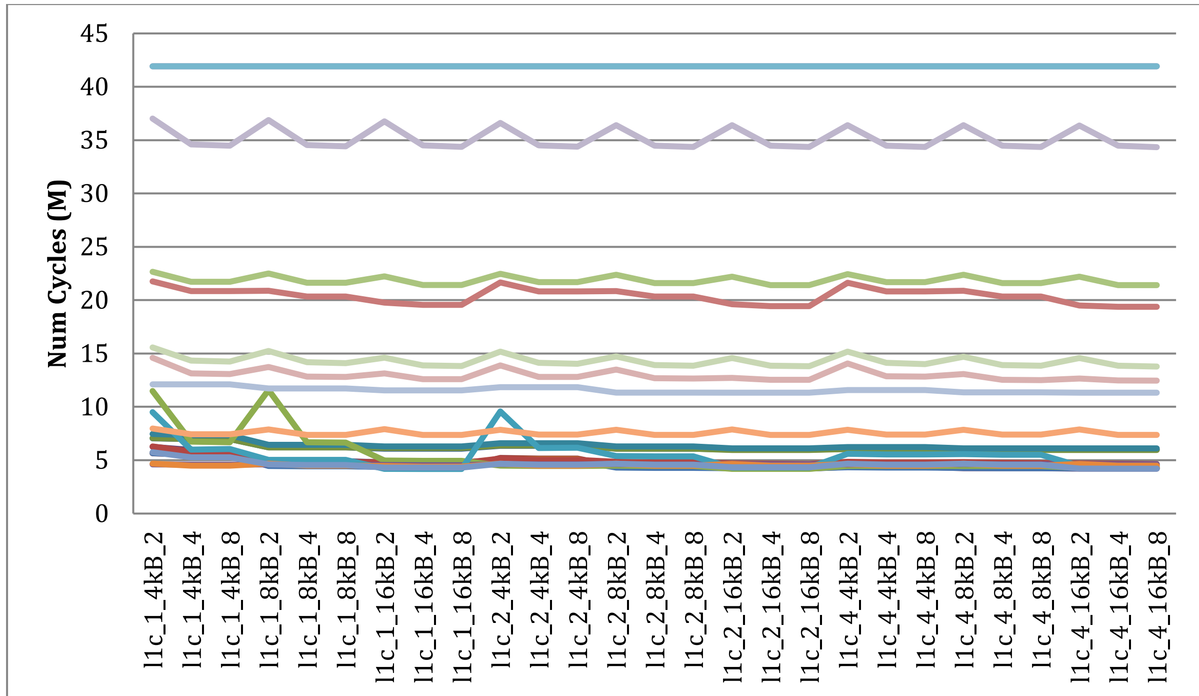

Figure 3: Running time for different L1 cache configurations

The following Figure 4 shows the L1 data cache miss rate (for all microbenchmarks across all L1 cache configurations), as defined by:

Miss Rate 1 = L1 misses / L1 accesses

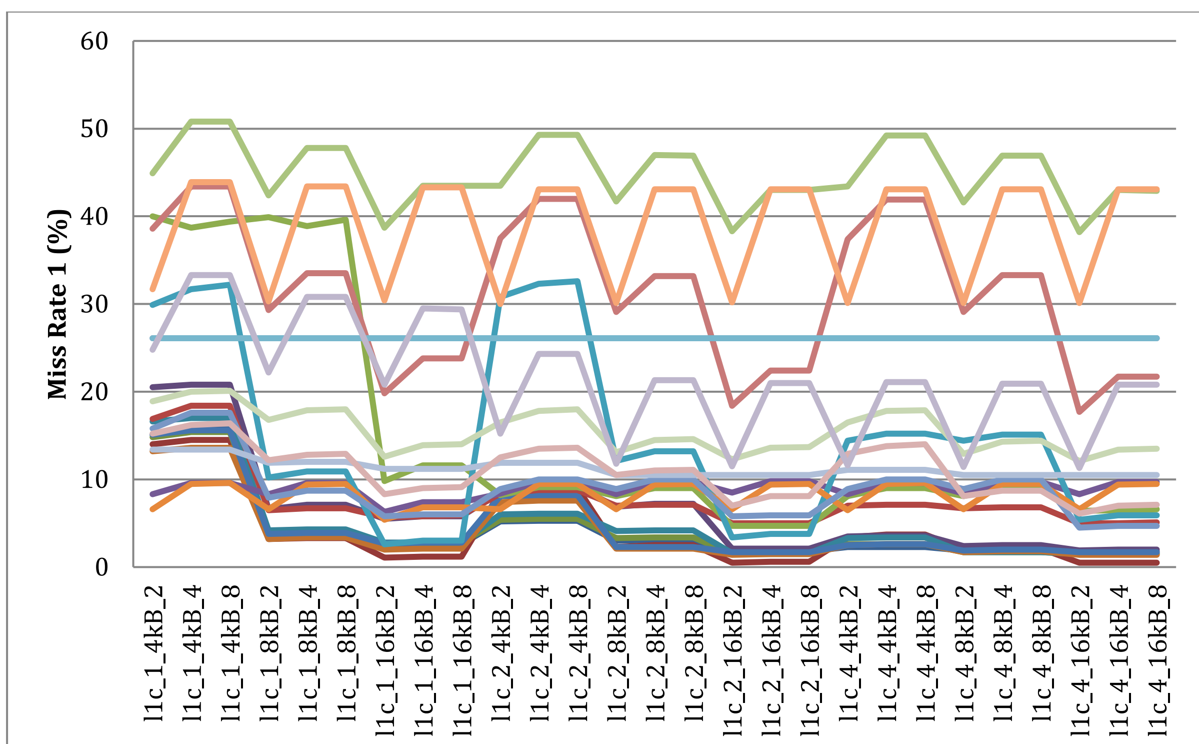

Figure 4: L1 data cache "Miss Rate 1" for different configurations

However, there are cases where this graph can be misleading. For example, we notice cases where increasing the number of MSHRs resulted in an increased miss rate even though, as we can see in the previous Figure, the overall runtime decreases. The reason is that an increased number of MSHRs leads to increased ILP, and in cases where multiple memory instructions that all access the same cache line happen to run in parallel, multiple cache misses will be recorded (notice that this is just an artifact of how gem5 counts misses, since misses to the same cache line should normally be counted as a single miss). This is in contrast to having a smaller number of MSHRs where subsequent memory instructions will be stalled and not allowed to cause a cache miss. At a closer look, the multiple cache misses for the same cache line that can be recorded with a higher number of MSHRs will all result in a single L2 cache access. For this reason, a better metric for the L1 miss rate could be the one expressed as:

Miss Rate 2 = L2 accesses / L1 accesses

The following Figure 5 shows this metric:

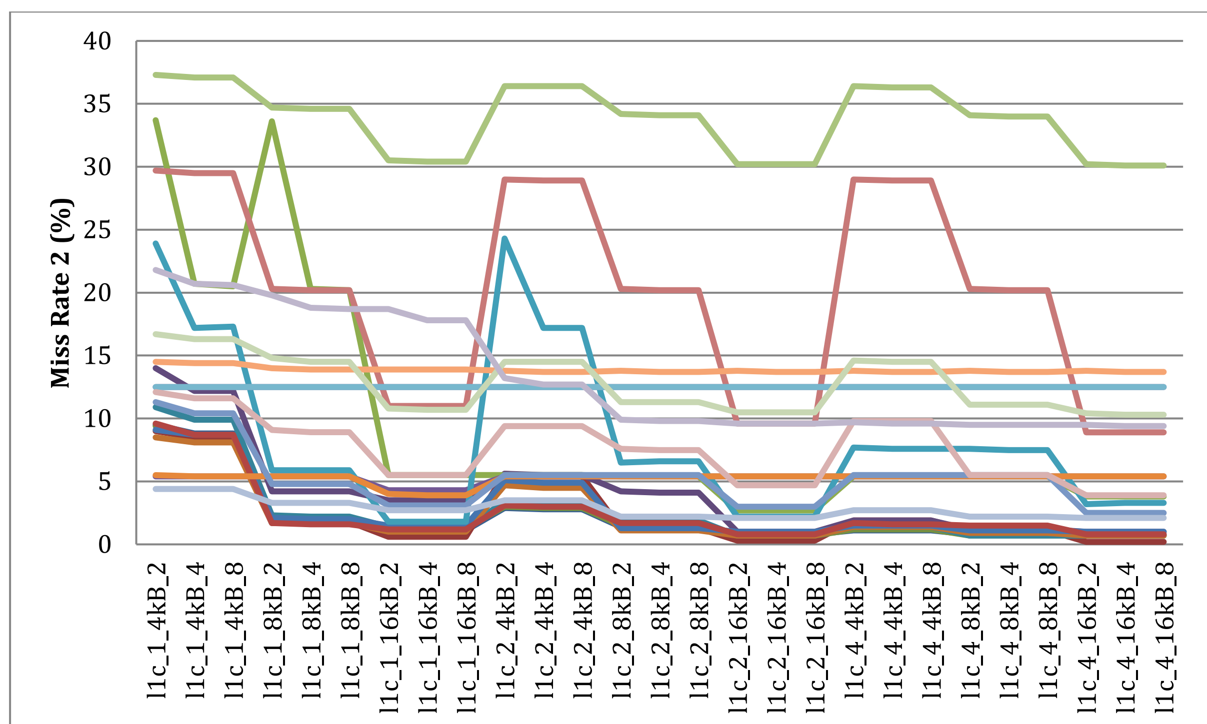

Figure 5: L1 data cache "Miss Rate 2" for different configurations

Here, we can see that increasing the number of MSHRs, does indeed lead to fewer L2 accesses, and thus overall decreased runtime.

Since the total of 27 configurations make the above diagrams a little hard to read and understand, here are the subset of these diagrams where only the configurations with 16 kBytes cache size are displayed, in Figures 6, 7 and 8.

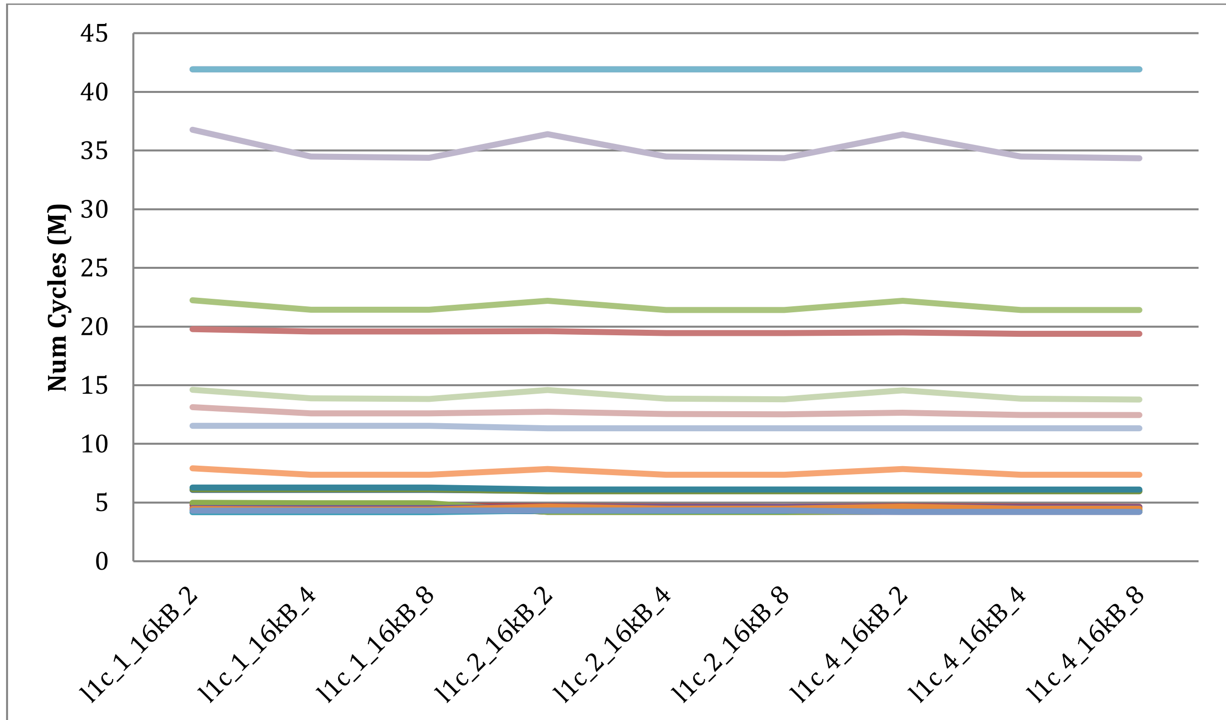

Figure 6: Running time for 16 kBytes L1 cache configurations

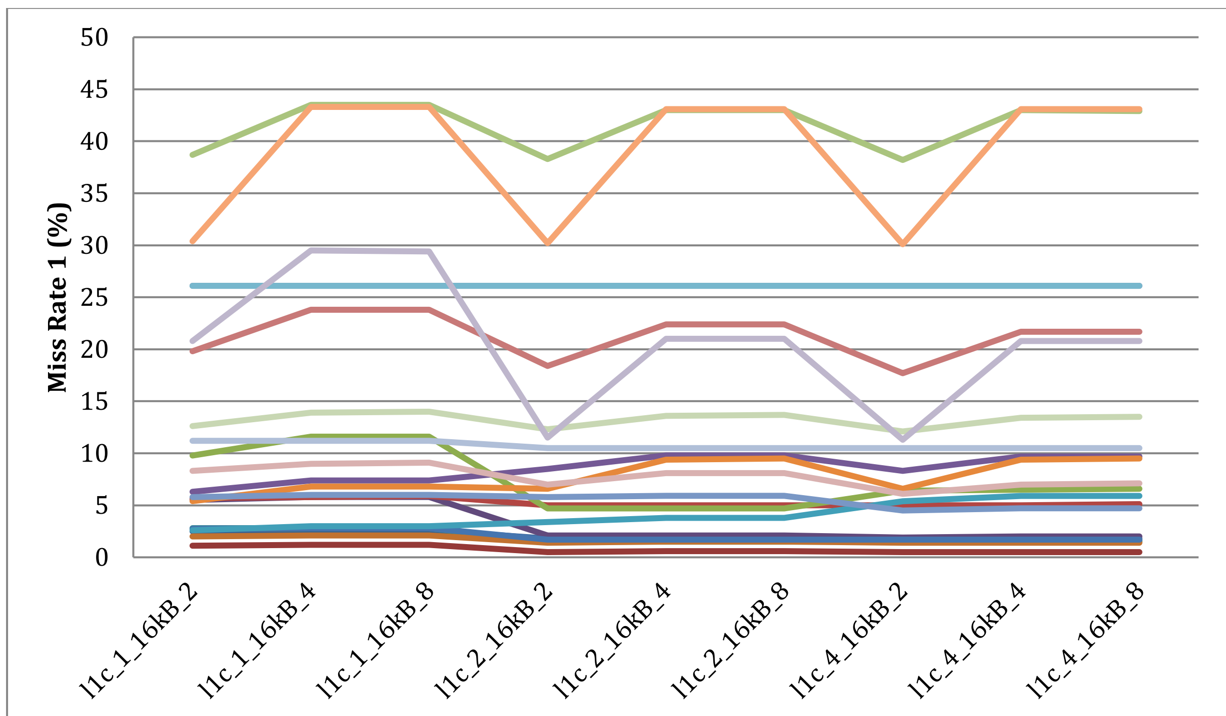

Figure 7: L1 data cache “Miss Rate 1” for 16 kBytes configurations

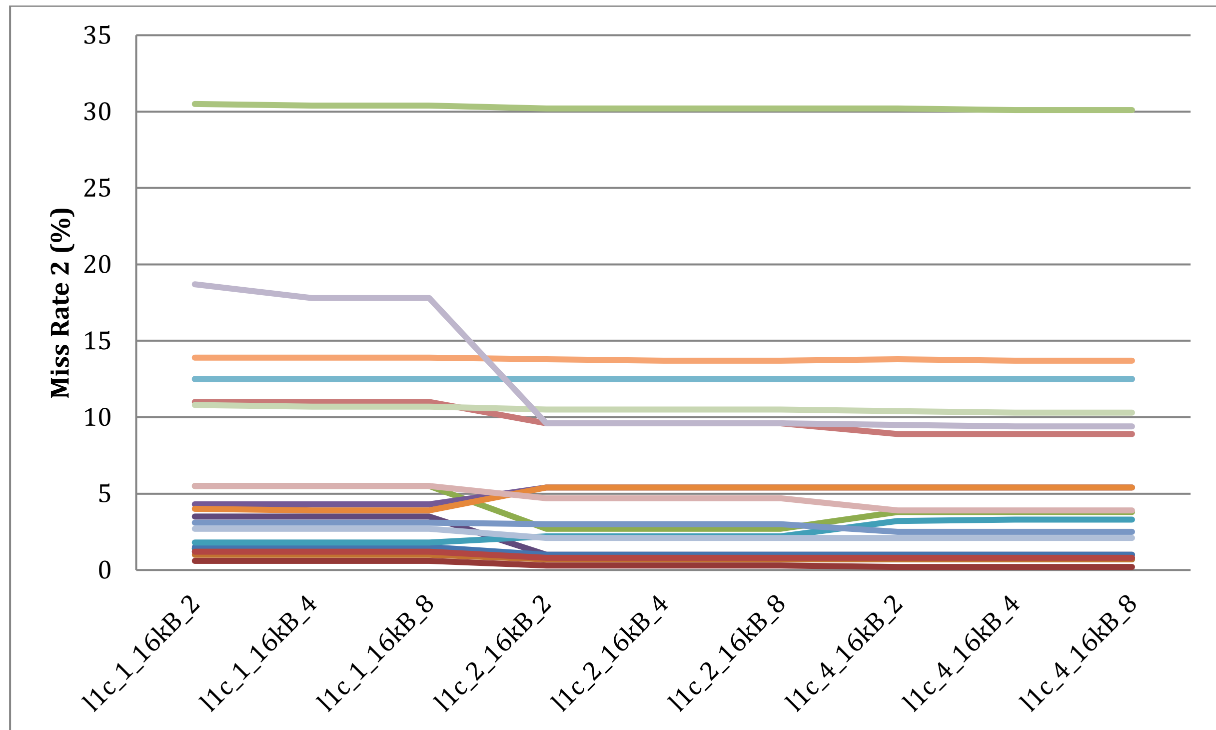

Figure 8: L1 data cache "Miss Rate 2" for 16 kBytes configurations

Speed-ups:

Going from direct-mapped to 2-way set associativity results in the following average and maximum speed-ups (across the 22 microbenchmarks) for the different L1 cache configurations (min speed-up is 0, i.e. no speed-up, and is not shown).

Notice: The speedups in this report are expressed in percentages, e.g. if time[configuration 1] = 150 and time[configuration 2] = 100, then the reported speedup going from configuration 1 to configuration 2 is given by the expression speedup = round(100*(150/100-1), 1) = 50.0 (%)

L1 Cache Configuration	Direct-mapped => 2-way Average Speedup (%)	Max Speedup (%)
4 kBytes, 2 MSHRs	13.9	156.1
4 kBytes, 4 MSHRs	7.4	50.6
4 kBytes, 8 MSHRs	7.4	49.7
8 kBytes, 2 MSHRs	8.3	158.5
8 kBytes, 4 MSHRs	3.2	49.6
8 kBytes, 8 MSHRs	3.1	49.3
16 kBytes, 2 MSHRs	2.1	18.6
16 kBytes, 4 MSHRs	1.9	17.7
16 kBytes, 8 MSHRs	1.9	17.7

Table 3: Speed-ups when L1 cache associativity changes from direct-mapped to 2-way

Going from 2-way to 4-way set associativity results in the following average and maximum speed-ups:

L1 Cache Configuration	2-way => 4-way Average Speedup (%)	Max Speedup (%)
4 kBytes, 2 MSHRs	7.1	70.6
4 kBytes, 4 MSHRs	4.2	14.6
4 kBytes, 8 MSHRs	4.2	14.6
8 kBytes, 2 MSHRs	1.0	12.2
8 kBytes, 4 MSHRs	0.9	11.8
8 kBytes, 8 MSHRs	0.9	11.8
16 kBytes, 2 MSHRs	0.0	2.6
16 kBytes, 4 MSHRs	0.0	2.9
16 kBytes, 8 MSHRs	0.0	2.9

Table 4: Speed-ups when L1 cache associativity changes from 2-way to 4-way

Going from 4 kBytes to 8 kBytes results in the following average and maximum speed-ups:

L1 Cache Configuration	4 kBytes => 8 kBytes Average Speedup (%)	Max Speedup (%)
Direct-mapped, 2 MSHRs	13.9	88.7
Direct-mapped, 4 MSHRs	8.8	23.9
Direct-mapped, 8 MSHRs	8.8	23.9
2-way set assoc., 2 MSHRs	7.7	77.9
2-way set assoc., 4 MSHRs	4.4	16.3
2-way set assoc., 8 MSHRs	4.4	16.3
4-way set assoc., 2 MSHRs	1.4	7.6
4-way set assoc., 4 MSHRs	1.0	3.4
4-way set assoc., 8 MSHRs	1.0	3.4

Table 5: Speed-ups when L1 cache size changes from 4 kBytes to 8 kBytes

Going from 8 kBytes to 16 kBytes results in the following average and maximum speed-ups:

L1 Cache Configuration	8 kBytes => 16 kBytes Average Speedup (%)	Max Speedup (%)
Direct-mapped, 2 MSHRs	9.3	132.5
Direct-mapped, 4 MSHRs	4.3	35.3
Direct-mapped, 8 MSHRs	4.3	35.1
2-way set assoc., 2 MSHRs	3.7	24.5
2-way set assoc., 4 MSHRs	3.2	24.4
2-way set assoc., 8 MSHRs	3.3	24.4
4-way set assoc., 2 MSHRs	2.8	25.1
4-way set assoc., 4 MSHRs	2.4	24.0
4-way set assoc., 8 MSHRs	2.4	24.0

Table 6: Speed-ups when L1 cache size changes from 8 kBytes to 16 kBytes

Going from 2 to 4 MSHRs results in the following average and max speed-ups:

L1 Cache Configuration	2 MSHRs => 4 MSHRs Average Speedup (%)	Max Speedup (%)
Direct-mapped, 4 kBytes	9.6	70.9
Direct-mapped, 8 kBytes	5.4	73.6
Direct-mapped, 16 kBytes	1.5	7.5
2-way set assoc., 4 kBytes	4.8	55.6
2-way set assoc., 8 kBytes	1.8	6.5

2-way set assoc., 16 kBytes	1.4	6.8
4-way set assoc., 4 kBytes	2.1	9.4
4-way set assoc., 8 kBytes	1.7	6.2
4-way set assoc., 16 kBytes	1.3	6.6

Table 7: Speed-ups when L1 cache changes from using 2 MSHRs to 4 MSHRs

Going from 4 to 8 MSHRs results in the following average and maximum speed-ups:

L1 Cache Configuration	4 MSHRs => 8 MSHRs Average Speedup (%)	Max Speedup (%)
Direct-mapped, 4 kBytes	0.1	0.7
Direct-mapped, 8 kBytes	0.1	0.5
Direct-mapped, 16 kBytes	0.1	0.4
2-way set assoc., 4 kBytes	0.1	0.6
2-way set assoc., 8 kBytes	0.1	0.4
2-way set assoc., 16 kBytes	0.1	0.4
4-way set assoc., 4 kBytes	0.1	0.8
4-way set assoc., 8 kBytes	0.1	0.5
4-way set assoc., 16 kBytes	0.1	0.5

Table 8: Speed-ups when L1 cache changes from using 4 MSHRs to 8 MSHRs

3.3.2 L2 Cache

For the unified L2 cache we vary the following parameters: associativity, and cache size. Specifically:

- The values that we try for associativity are: 2, 4, and 8
- The values that we try for size are: 64 kBytes, 128 kBytes, 256 kBytes, and 512 kBytes

The number of MSHRs is kept constant at 20. Taking all different combinations for these values results in a total of 3 x 4 = 12 configurations for the L2 cache.

We use the following naming convention for these configurations:

l2c_[associativity]_[size]

For example, configuration l2c_2_128kB represents a L2 cache that is 2-way set associative and has a size of 128 kBytes. Configuration l2c_4_256kB_2 represents a L2 cache that is 4-way set associative and has a size of 256 kBytes.

The following Figure 9 shows the running time (as expressed in millions of clock cycles), for each microbenchmark across all 12 L2 cache configurations.

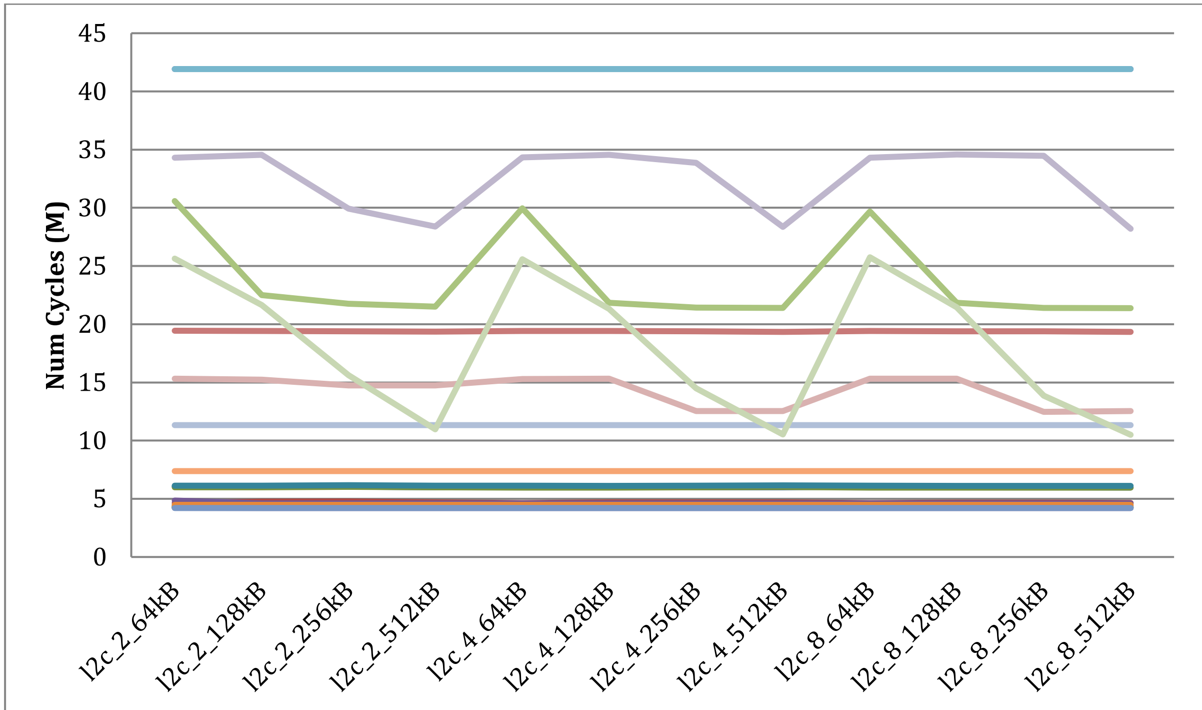

Figure 9: Running time for different L2 cache configurations

The following Figure 10 shows the L2 cache miss rate, as defined by:

Miss Rate = L2 misses / L2 accesses

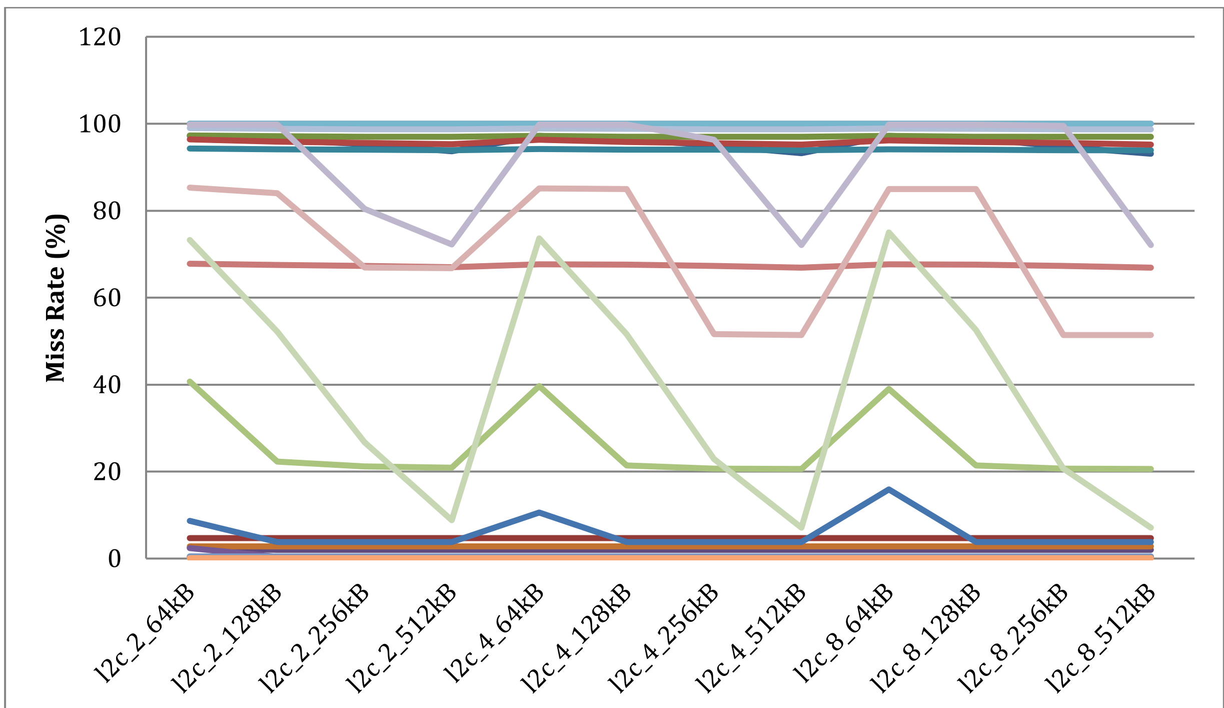

Figure 10: L2 cache Miss Rate for different configurations

We also vary the latency of the L2 cache from 12 up to 24 clock cycles, while keeping all other parameters at their default values (i.e. size=256 kBytes, associativity=8). The following Figure 11 shows the running time for the different latency settings:

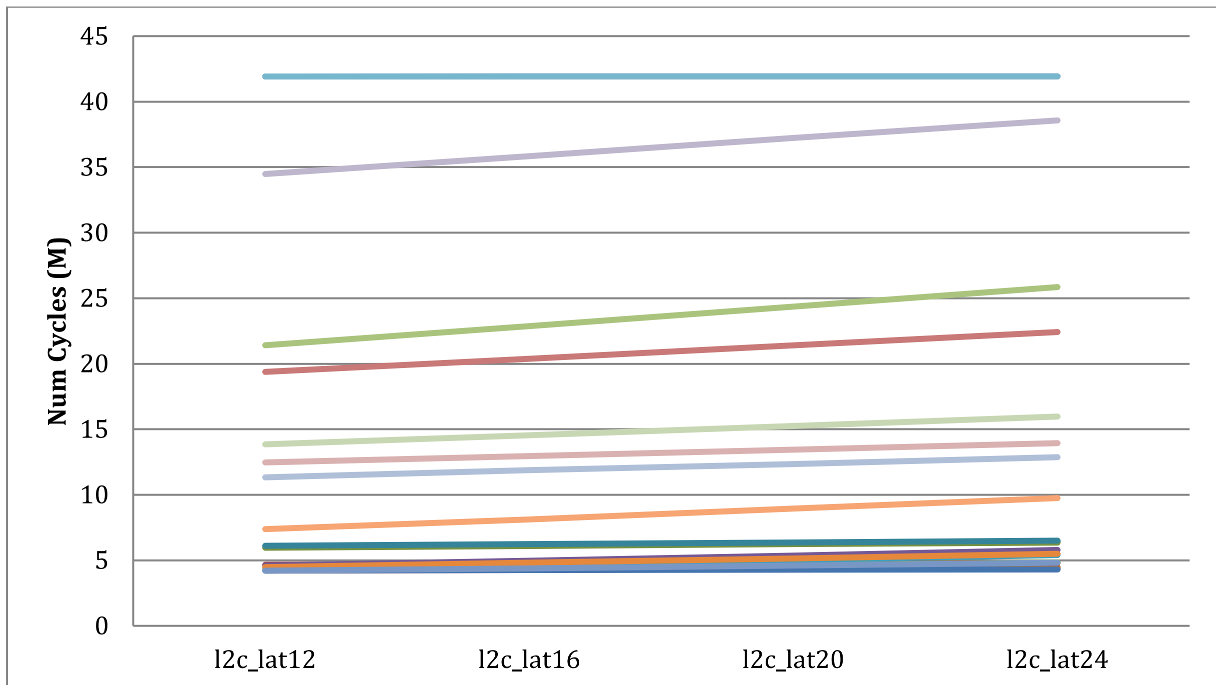

Figure 11: Running time for different L2 cache latencies (256 kBytes, 8-way set assoc.)

Speed-ups:

Going from 2-way to 4-way set associativity results in the following average and maximum speed-ups:

L2 Cache Configuration	2-way => 4-way Average Speedup (%)	Max Speedup (%)
64 kBytes	0.4	5.4
128 kBytes	0.3	3.0
256 kBytes	0.9	17.6
512 kBytes	1.0	17.6

Table 9: Speed-ups when L2 cache associativity changes from 2-way to 4-way

Going from 4-way to 8-way set associativity results in the following average and maximum speed-ups:

L2 Cache Configuration	4-way => 8-way Average Speedup (%)	Max Speedup (%)
64 kBytes	0.0	0.9
128 kBytes	0.0	0.4
256 kBytes	0.2	4.7
512 kBytes	0.2	1.3

Table 10: Speed-ups when L2 cache associativity changes from 4-way to 8-way

Going from 64 kBytes to 128 kBytes results in the following average and maximum speed-ups:

L2 Cache Configuration	64 kBytes => 128 kBytes Average Speedup (%)	Max Speedup (%)
2-way set assoc.	2.7	35.9
4-way set assoc.	2.6	37.2
8-way set assoc.	2.6	36.0

Table 11: Speed-ups when L2 cache size changes from 64 kBytes to 128 kBytes

Going from 128 kBytes to 256 kBytes results in the following average and maximum speed-ups:

L2 Cache Configuration	128 kBytes => 256 kBytes Average Speedup (%)	Max Speedup (%)
2-way set assoc.	2.7	38.6
4-way set assoc.	3.3	47.0
8-way set assoc.	3.7	54.9

Table 12: Speed-ups when L2 cache size changes from 128 kBytes to 256 kBytes

Going from 256 kBytes to 512 kBytes results in the following average and maximum speed-ups:

L2 Cache Configuration	256 kBytes => 512 kBytes Average Speedup (%)	Max Speedup (%)
2-way set assoc.	2.4	42.5
4-way set assoc.	2.6	37.6
8-way set assoc.	2.5	31.9

Table 13: Speed-ups when L2 cache size changes from 256 kBytes to 512 kBytes

Changing the L2 cache latency from 24 down to 12 cycles results in the following average and maximum speed-ups (for a 256 kBytes, 8-way set assoc. L2 cache):

256 kBytes, 8-way s.a. L2 cache	Average Speedup (%)	Max Speedup (%)
Latency going from 24 to 20 cycles	3.6	9.1
Latency going from 20 to 16 cycles	3.7	10.1
Latency going from 16 to 12 cycles	3.6	10.0

Table 14: Speed-ups when L2 cache latency changes from 24 cycles down to 12 cycles

3.3.3 L1 and L2 Caches Line Size

The following Figures 12, 13 and 14 show how varying the cache line size from 16 up to 64 bytes, affects the running time and the L1 data cache and L2 cache miss rates.

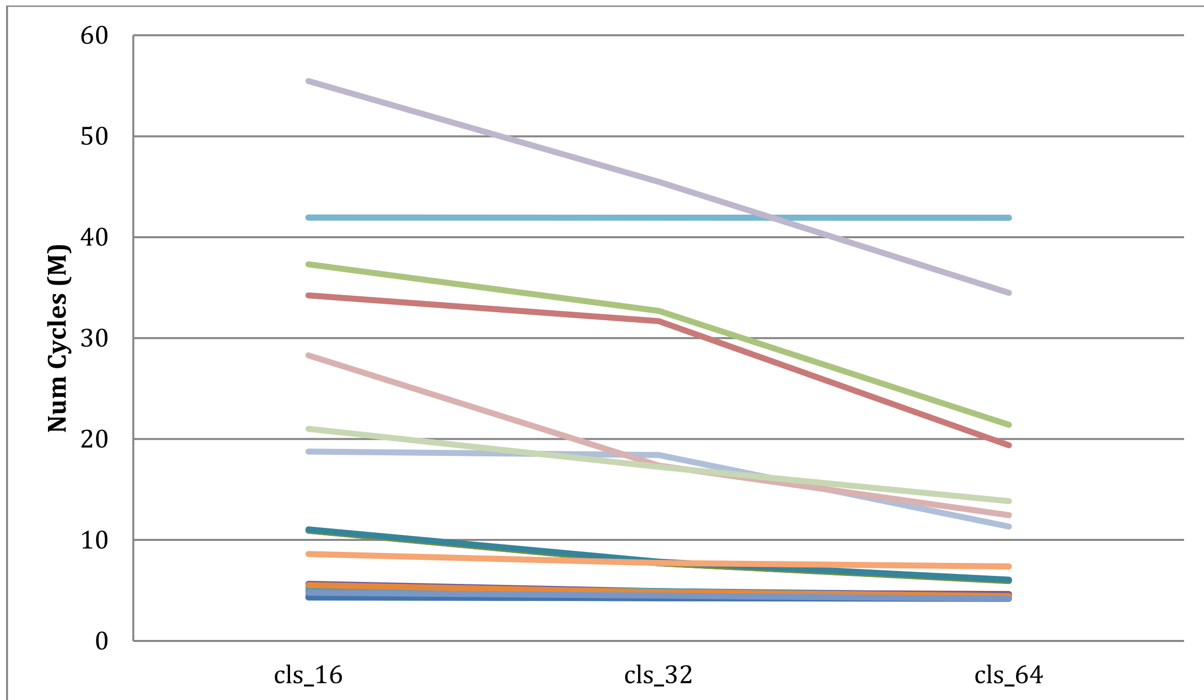

Figure 12: Running time for different cache line sizes

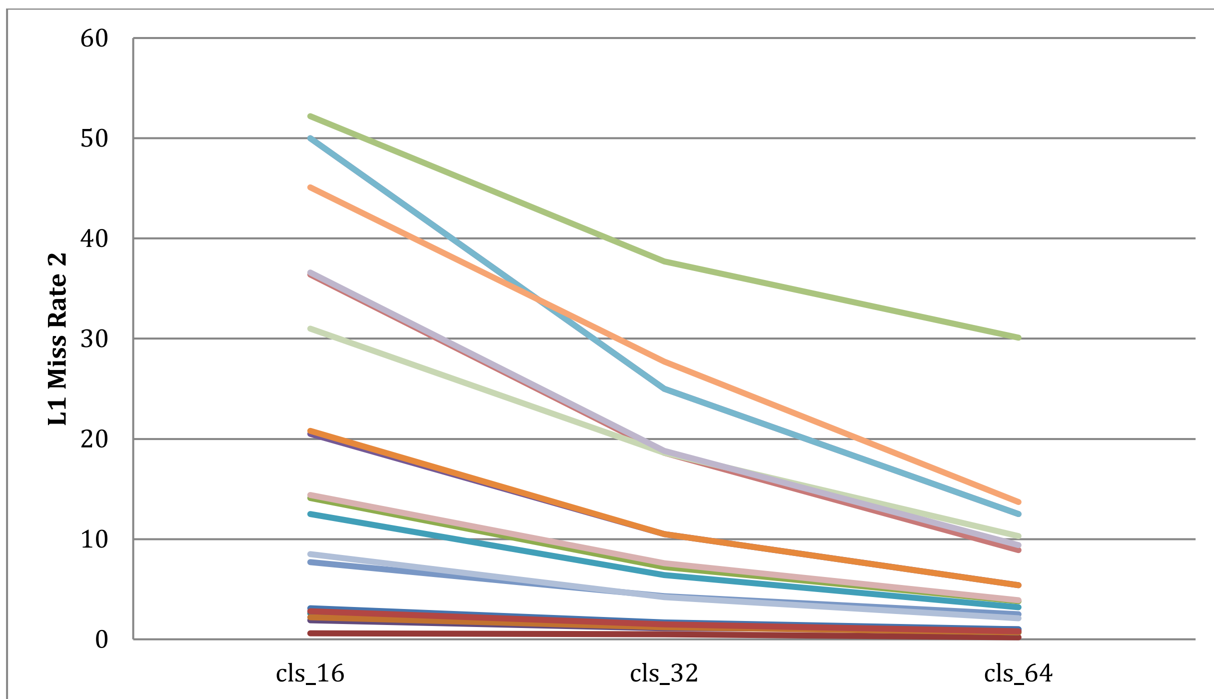

Figure 13: L1 data cache “Miss Rate 2” for different cache line sizes

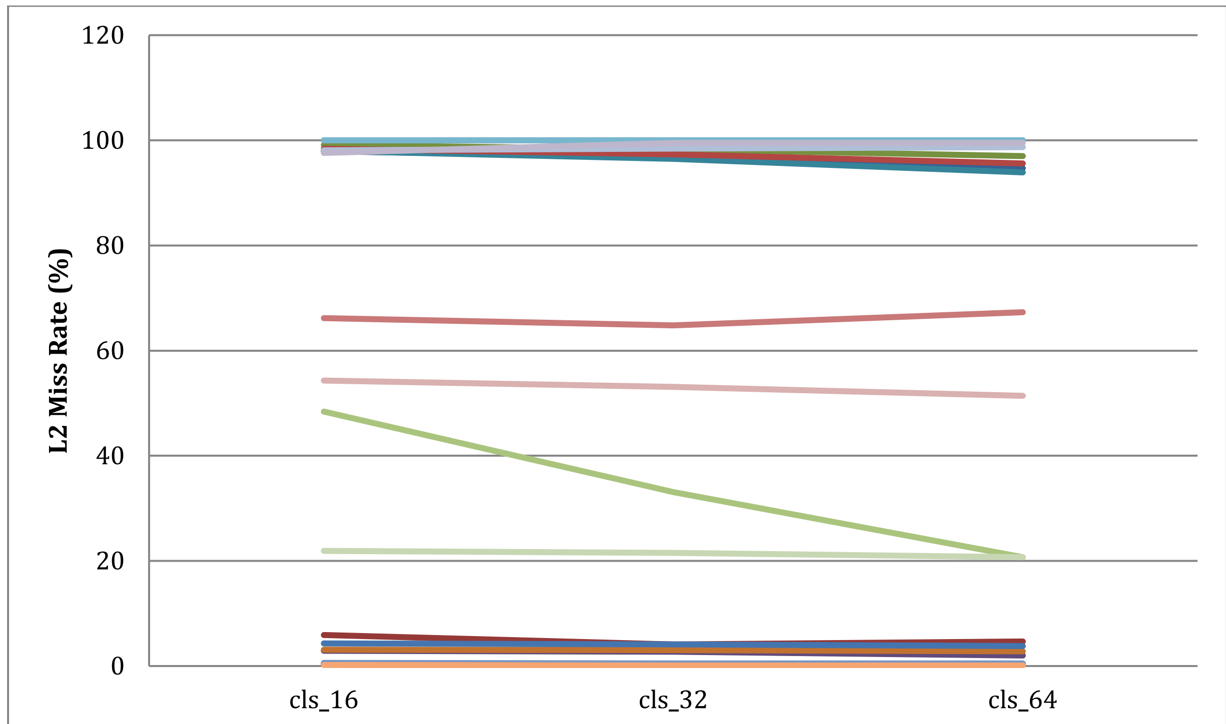

Figure 14: L2 cache Miss Rate for different cache line sizes

Speed-ups:

Going from 16 to 32 bytes results in the following speed-ups: average 15.2, maximum 63.0
Going from 32 to 64 bytes results in the following speed-ups: average 19.2, maximum 63.5

3.3.4 Integer Unit

The Integer unit comprises of 3 types of units:

- Integer multipliers (for Integer multiplications)
- Integer divisors (for Integer divisions)
- Integer ALUs (for all remaining Integer operations)

Since the available microbenchmarks do not perform any (at least, significant) number of integer multiplications or divisions, we did not examine the Integer multipliers or divisors.

For the Integer ALUs, the parameters that we vary are: the number of ALUs, latency, and whether they are pipelined or not. Specifically:

- The values that we try for the number of ALUs are: 1, 2, and 4.
- The values that we try for latency are: 1, 2, and 4 clock cycles.
- The values that we try for pipelined are: false, and true (in case of latencies higher than 1).

Taking all different combinations for these values results in a total of 3 + (3 x 2 x 2) = 15 configurations for the Integer ALUs.

We use the following naming convention for these configurations:

ALU_c[# of ALUs]_l[latency]_p[1 if pipelined, 0 if not]

For example, ALU_c1_l4_p0 represents a configuration with 1 ALU that has a latency of 4 cycles and is not pipelined. Similarly ALU_c4_l2_p1 represents a configuration with 4 ALUs that have a latency of 2 cycles and are pipelined.

The following Figure 15 shows the running time for each microbenchmark across all 15 Integer ALU configurations.

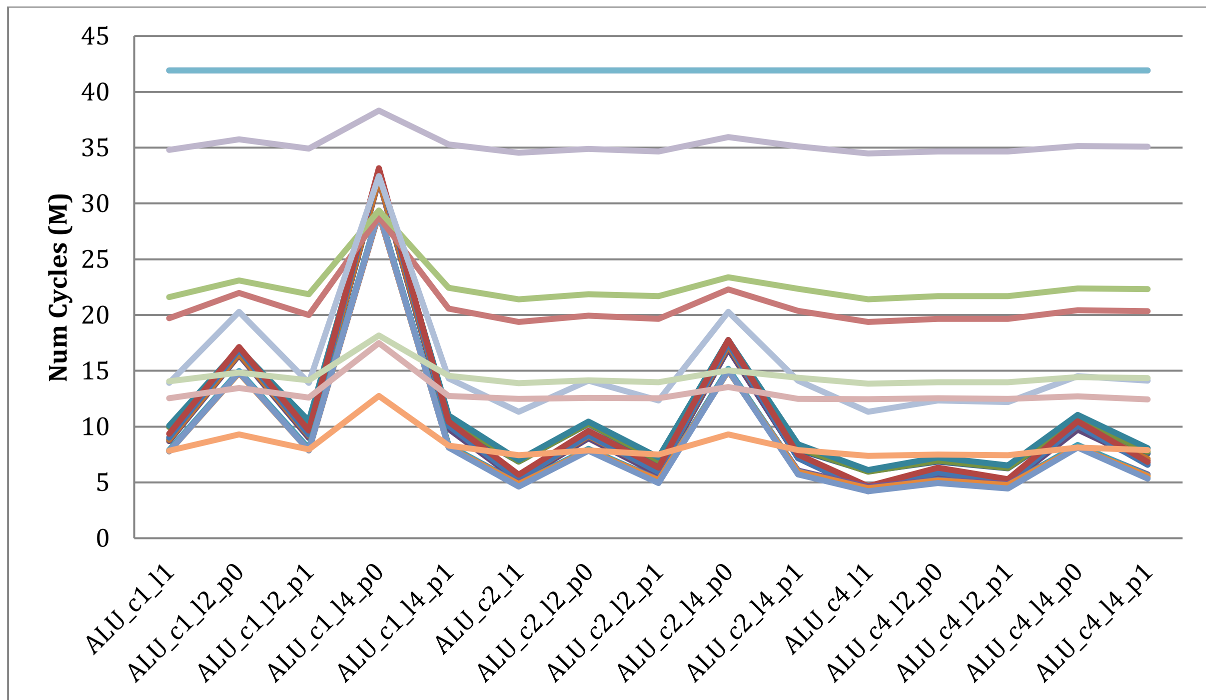

Figure 15: Running time for different ALU configurations

Assuming most designs nowadays use a pipelined integer ALU, the following Figure 16 shows only the subset of this figure containing the pipelined ALU configurations.

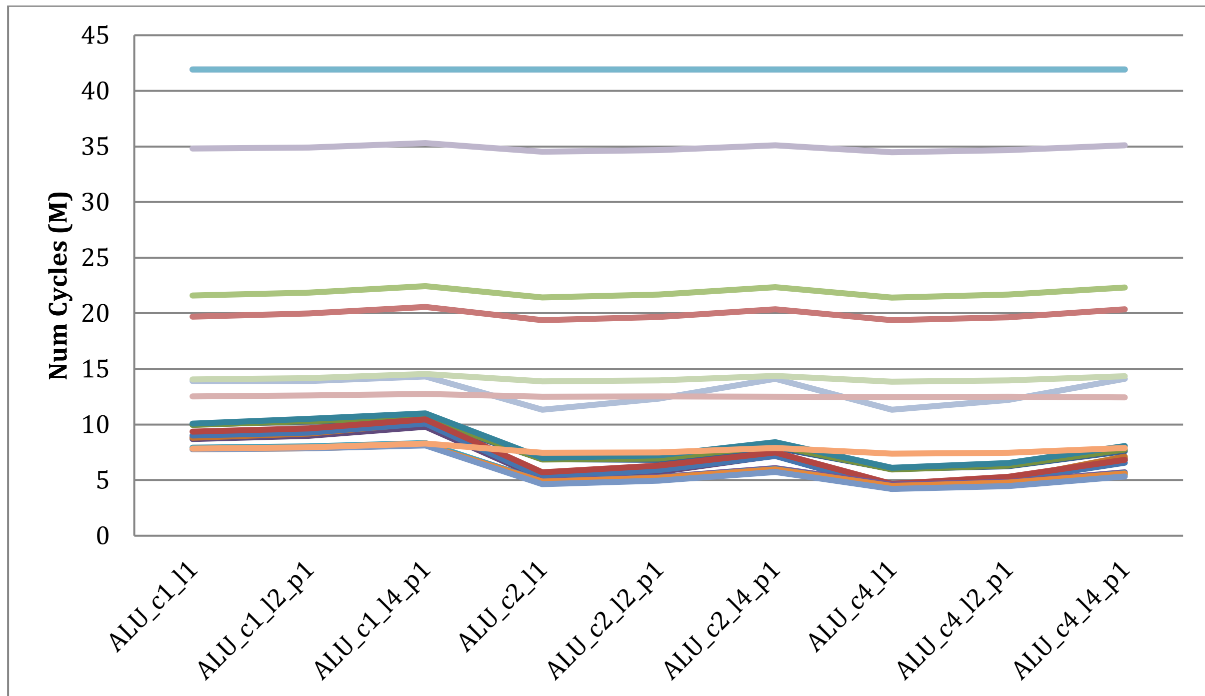

Figure 16: Running time for pipelined ALU configurations

The following Figure 17 shows the parameter "FU Busy" which represents the number of attempts that were made to use an ALU when none was available.

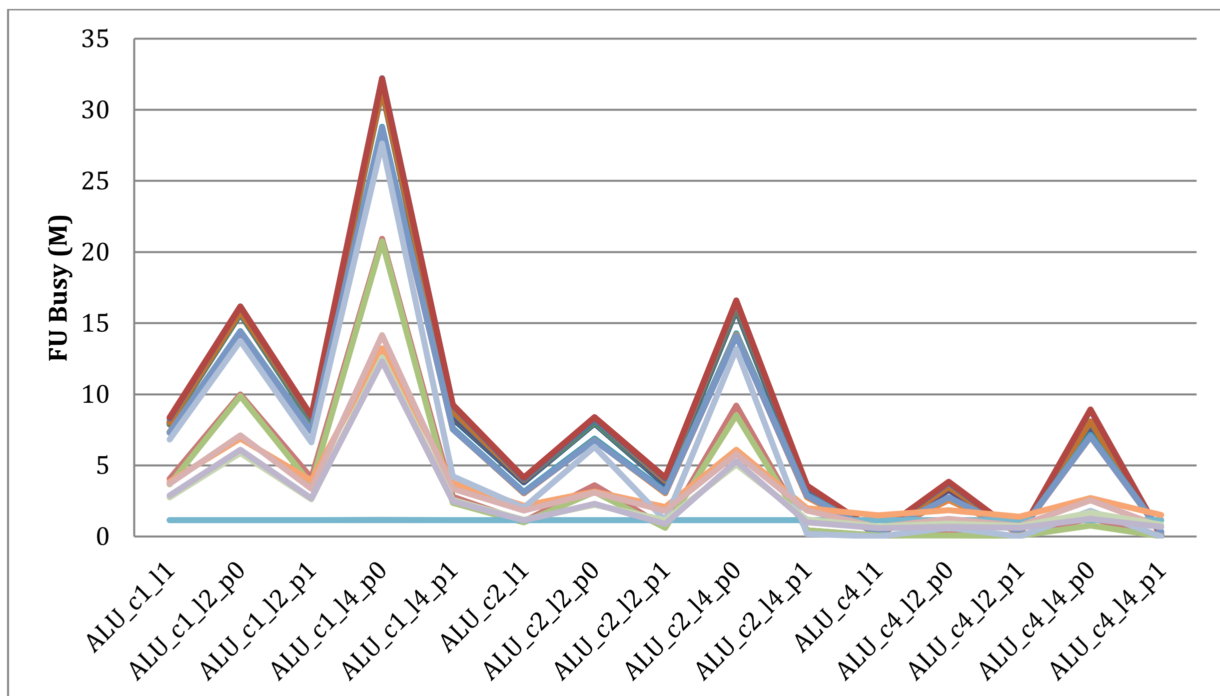

Figure 17: ALU non-availability for different ALU configurations

Speed-ups:

Going from 1 to 2 ALUs results in the following average and maximum speed-ups:

ALUs Configuration	1 Int ALU => 2 Int ALUs Average Speedup (%)	Max Speedup (%)
Latency: 1	37.3	70.6
Latency: 2, Pipelined: false	51.4	89.7
Latency: 2, Pipelined: true	32.6	59.3
Latency: 4, Pipelined: false	62.1	94.2
Latency: 4, Pipelined: true	22.1	41.7

Table 15: Speed-ups when number of Integer ALUs changes from 1 to 2

Going from 2 to 4 ALUs results in the following average and maximum speed-ups:

ALUs Configuration	2 Int ALUs => 4 Int ALUs Average Speedup (%)	Max Speedup (%)
Latency: 1	9.4	24.5
Latency: 2, Pipelined: false	32.5	59.0
Latency: 2, Pipelined: true	7.8	19.5
Latency: 4, Pipelined: false	47.6	84.6
Latency: 4, Pipelined: true	3.8	9.7

Table 16: Speed-ups when number of Integer ALUs changes from 2 to 4

3.3.5 Floating Point Unit

The FP unit comprises of 3 types of units:

- FP multipliers (for single or double precision FP multiplications)
- FP divisors (for single or double precision FP divisions)
- FP ALUs (for all remaining FP operations)

Since the available microbenchmarks do not perform any (at least, significant) number of FP divisions, we did not examine the FP divisors.

For the FP ALUs, the parameters that we vary are: the number of ALUs, latency, and whether they are pipelined or not. Specifically:

- The values that we try for the number of ALUs are: 1, 2, and 4.
- The values that we try for latency are: 3, 4, and 5 clock cycles.
- The values that we try for pipelined are: false, and true.

Taking all different combinations for these values results in a total of 3 x 3 x 2 = 18 configurations for the FP ALUs.

The naming convention that we use for these configurations is similar to the one for the Integer ALUs:

FP_ALU_c[# of ALUs]_l[latency]_p[1 if pipelined, 0 if not]

The following Figure 18 shows the running time for each microbenchmark across all 18 FP ALU configurations.

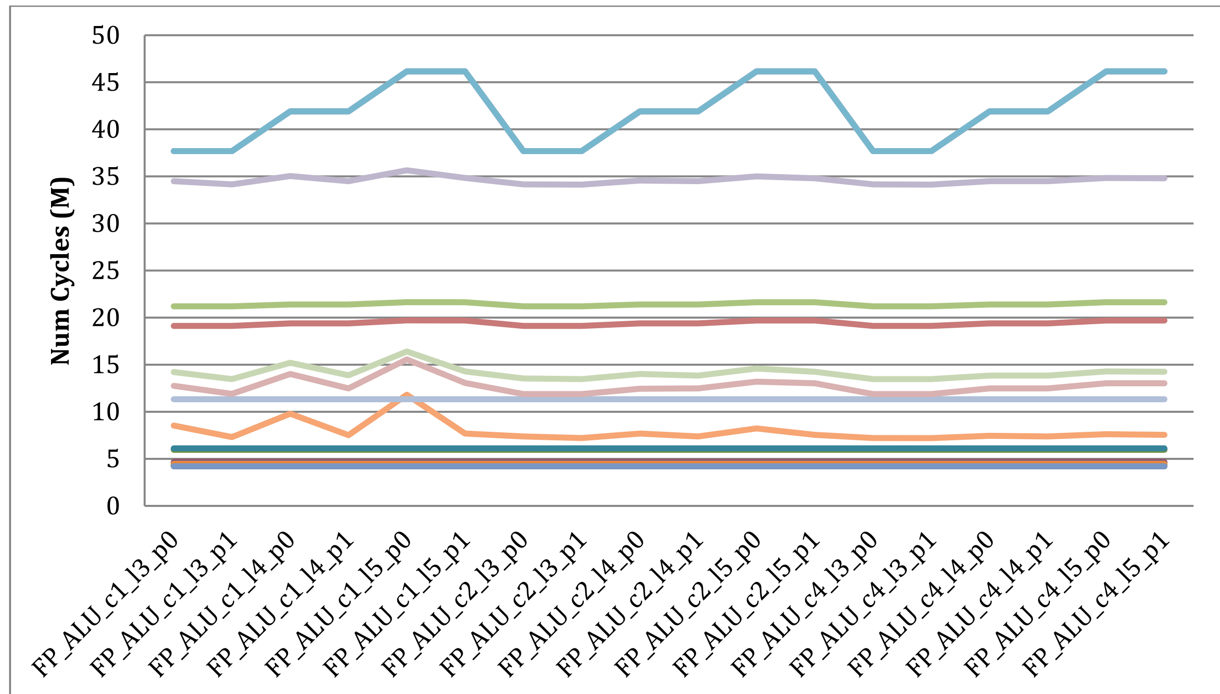

Figure 18: Running time for different FP ALU configurations

For the FP Multipliers, the parameters that we vary are: the number of Multipliers, latency, and whether they are pipelined or not. Specifically:

- The values that we try for the number of Multipliers are: 1, 2.
- The values that we try for latency are: 6, 8, and 10 clock cycles.
- The values that we try for pipelined are: false, and true.

Taking all different combinations for these values results in a total of 2 x 3 x 2 = 12 configurations for the FP Multipliers.

The naming convention that we use for these configurations is similar to the one for the FP ALUs:

FP_mult_c[# of ALUs]_l[latency]_p[1 if pipelined, 0 if not]

The following Figure 19 shows the running time for each microbenchmark across all 12 FP Multipliers configurations.

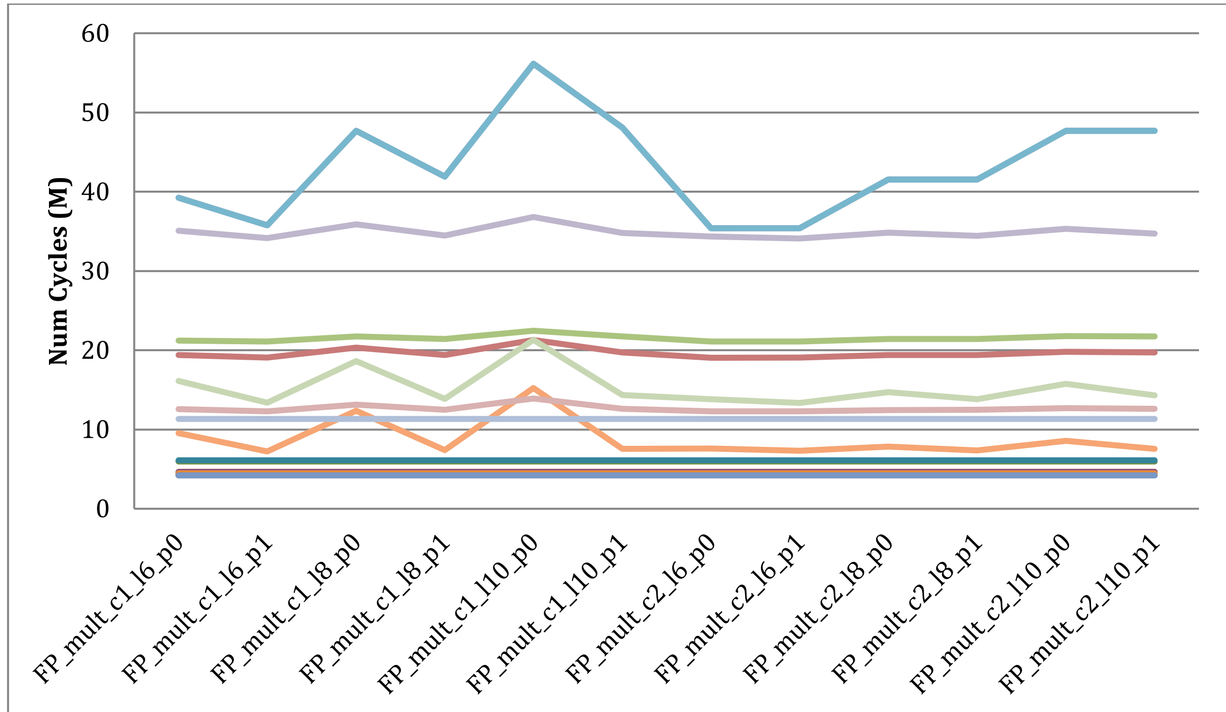

Figure 19: Running time for different FP Multipliers configurations

Speed-ups:

Going from 1 to 2 FP ALUs results in the following average and maximum speed-ups:

FP ALUs Configuration	1 FP ALU => 2 FP ALUs Average Speedup (%)	Max Speedup (%)
Latency: 3, Pipelined: false	1.3	15.4
Latency: 3, Pipelined: true	0.1	1.5
Latency: 4, Pipelined: false	2.3	27.2
Latency: 4, Pipelined: true	0.1	1.6
Latency: 5, Pipelined: false	3.4	43.3
Latency: 5, Pipelined: true	0.1	1.6

Table 17: Speed-ups when number of FP ALUs changes from 1 to 2

Going from 2 to 4 FP ALUs results in the following average and maximum speed-ups:

FP ALUs Configuration	2 FP ALU => 4 FP ALUs Average Speedup (%)	Max Speedup (%)
Latency: 3, Pipelined: false	0.1	2.2
Latency: 3, Pipelined: true	0.0	0.1
Latency: 4, Pipelined: false	0.2	3.2
Latency: 4, Pipelined: true	0.0	0.0
Latency: 5, Pipelined: false	0.6	8.4
Latency: 5, Pipelined: true	0.0	0.3

Table 18: Speed-ups when number of FP ALUs changes from 2 to 4

Going from 1 to 2 FP multipliers results in the following average and maximum speed-ups:

FP Multipliers Configuration	1 FP mult => 2 FP mult Average Speedup (%)	Max Speedup (%)
Latency: 6, Pipelined: false	3.2	25.6
Latency: 6, Pipelined: true	0.1	1.1
Latency: 8, Pipelined: false	5.9	57.7
Latency: 8, Pipelined: true	0.1	0.9
Latency: 10, Pipelined: false	7.9	77.7
Latency: 10, Pipelined: true	0.1	0.8

Table 19: Speed-ups when number of FP Multipliers changes from 1 to 2

Changing the latency of FP multipliers from 10 to 8 clock cycles, results in the following average and maximum speed-ups:

FP Multipliers Configuration	Latency 10 => Latency 8 Average Speedup (%)	Max Speedup (%)
# of Multipliers: 1, Pipelined: false	4.1	23.3
# of Multipliers: 1, Pipelined: true	1.9	14.7
# of Multipliers: 2, Pipelined: false	2.4	14.8
# of Multipliers: 2, Pipelined: true	1.9	14.8

Table 20: Speed-ups when FP Multipliers latency changes from 10 to 8 cycles

Changing the latency of FP multipliers from 8 to 6 clock cycles, results in the following average and maximum speed-ups:

FP Multipliers Configuration	Latency 8 => Latency 6 Average Speedup (%)	Max Speedup (%)
# of Multipliers: 1, Pipelined: false	4.6	29.6
# of Multipliers: 1, Pipelined: true	2.1	17.2
# of Multipliers: 2, Pipelined: false	2.3	17.4
# of Multipliers: 2, Pipelined: true	2.0	17.4

Table 21: Speed-ups when FP Multipliers latency changes from 8 to 6 cycles

3.3.6 Load-Store Buffers

The following Figure 20 shows how varying the number of load and store buffers from 4 up to 32, affected the running time. Each configuration contains the same number of load and store buffers, for example, configuration lsq_4 contains 4 load buffer entries and 4 store buffer entries.

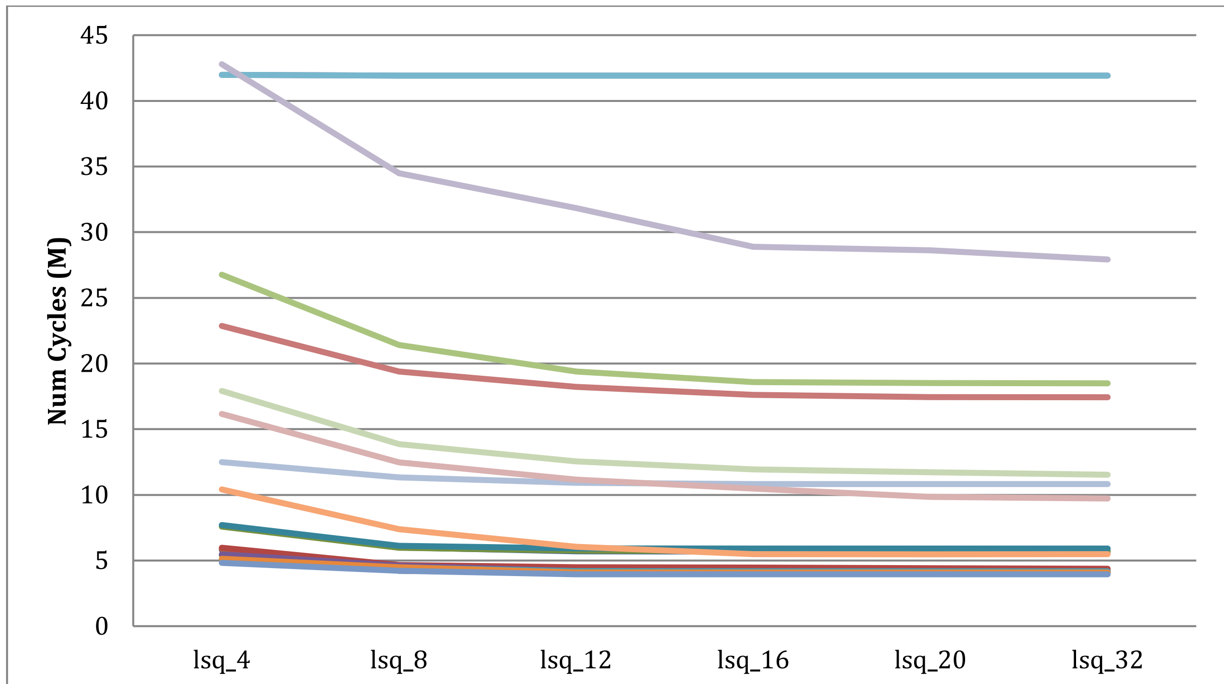

Figure 20: Running time for different Load-Store buffers configurations

Speed-ups:

Changing the number of load & store buffer entries results in the following average and maximum speed-ups:

Change	Average Speedup (%)	Max Speedup (%)
Going from 4 to 8 load & store buffer entries	23.4	41.1
Going from 8 to 12 load & store buffer entries	6.2	22.0
Going from 12 to 16 load & store buffer entries	2.1	10.4
Going from 16 to 20 load & store buffer entries	0.5	6.4
Going from 20 to 32 load & store buffer entries	0.3	2.5

Table 22: Speed-ups when number of Load & Store Buffer Entries changes from 4 up to 32

3.3.7 Reorder Buffers

The following Figure 21 shows how varying the number of reorder buffer entries from 32 up to 256, affects the running time.

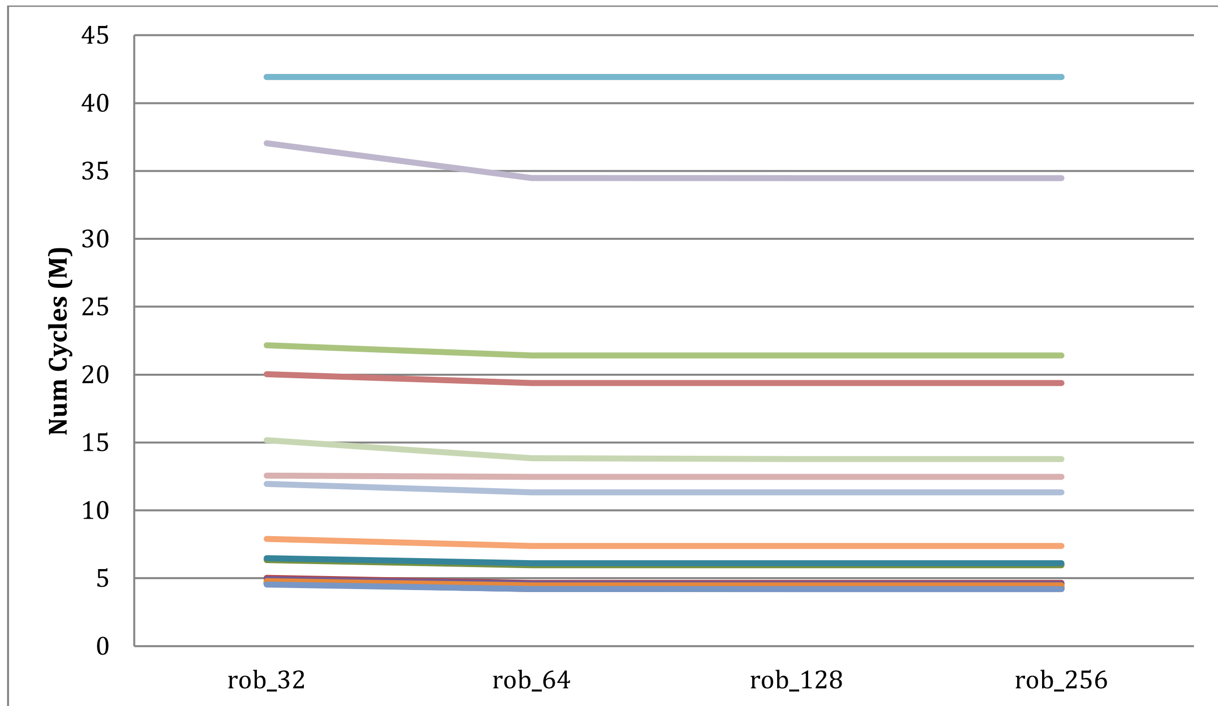

Figure 21: Running time for different Reorder buffers configurations

Speed-ups:

Changing the number of reorder buffer entries results in the following average and maximum speed-ups:

Change	Average Speedup (%)	Max Speedup (%)
Going from 32 to 64 ROBs	6.4	11.5
Going from 64 to 128 ROBs	0.1	0.5
Going from 128 to 256 ROBs	0.0	0.0

Table 23: Speed-ups when number of Reorder Buffer Entries changes from 32 up to 256

3.3.8 Instruction Mix

The following Figure 22 shows the instruction mix, in millions of executed instructions across all microbenchmarks (for a specific configuration, in this case l1c_1_4kB_2). The number of commited instructions is fixed at 10 million for all the simulations that we run. However, the number of executed instructions ("num_insts" in the figure) can be higher due to branch mispredictions and speculative execution.

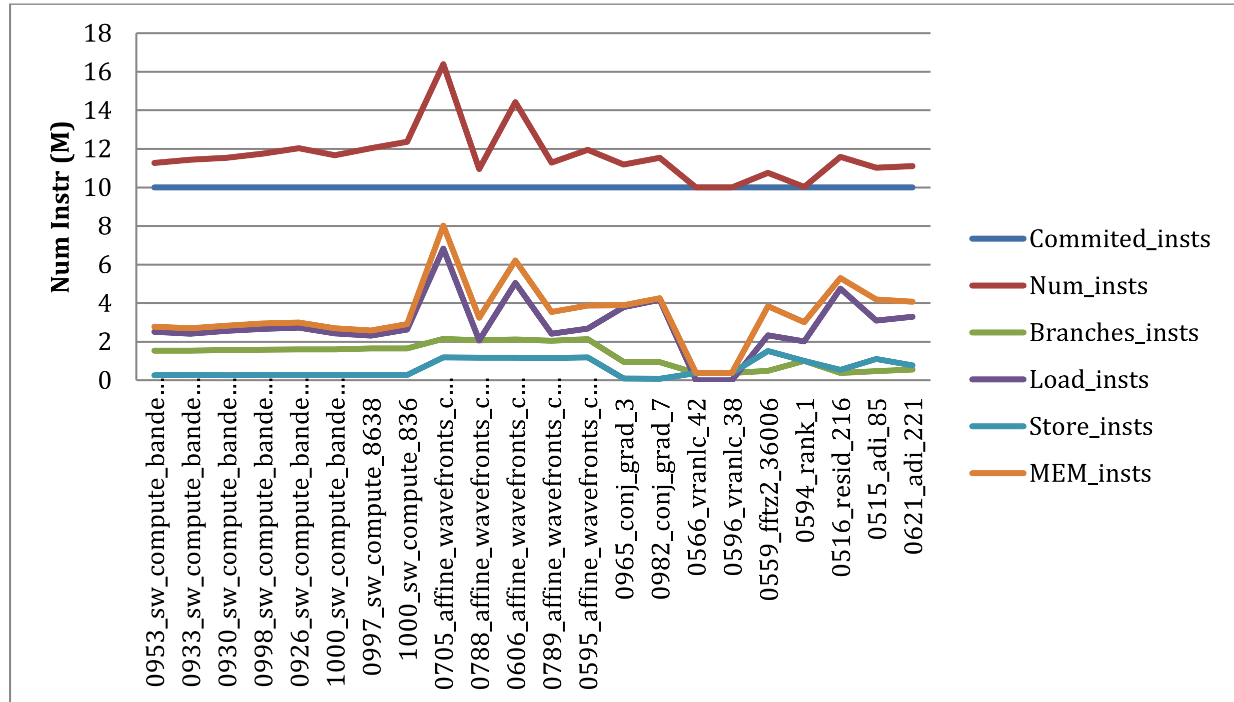

Figure 22: Instruction mix

4. Simulation for Vector Accelerators

This section provides the results in the integration of the Out of Order core with a VPU. In this deliverable we develop the first approach to build the simulation infrastructure required to take the architectural decisions for the design of the eProcessor chip and the VPU integration. For this purpose we used gem5, a simulator for computer architecture research, widely accepted in both academia and industry.

4.1 Methodology

To simulate a single RISC-V core processor on the gem5 simulator we use an out of order CPU microarchitecture model and a memory model that resembles an ARM A72. The parameters used for this core are described in detail in Table 3.

Parameter name	Default value	Description
--numROBEntries	128	Number of reorder buffer entries
--numPhysIntRegs	128	Number of physical integer registers
--LQEntries	48	Load Queue entries
--SQEntries	48	Store Queue entries
--fetchWidth	2	Fetch width
--fetchBufferSize	16	Fetch buffer size in bytes
--dispatchWidth	2	Dispatch width
--commitWidth	8	Commit width

--wbWidth	8	Writeback width
--decodeWidth	2	Decode width
--issueWidth	8	Issue width
--renameWidth	3	Rename width
--squashWidth	12	Squash width
--cache_line_size	64	Cache line size in bytes
--l1ic_size	48Kb	L1 Instr cache size
--l1ic_assoc	3	L1 Instr cache associativity
--l1ic_mshrs	4	L1 Instr cache MSHRs (max outstanding requests)
--l1ic_latency	1	L1 Instr cache latency
--l1dc_size	32Kb	L1 Data cache size
--l1dc_assoc	2	L1 Data cache associativity
--l1dc_mshrs	16	L1 Data cache MSHRs (max outstanding requests)
--l1dc_latency	2	L1 Data cache latency
--l2c_size	256kB	L2 cache size
--l2c_assoc	16	L2 cache associativity
--l2c_latency	12	L2 cache latency
--rdwr_count		Number of read-write ports
--ALU_count	4	Number of Integer ALUs
--ALU_opLat	1	ALU latency
--ALU_pipelined	True	ALU pipelined
--mult_count	1	Number of Integer Multipliers
--mult_opLat	4	Integer Multiplier latency
--mult_pipelined	True	Integer Multiplier pipelined
--div_count	1	Number of Integer Dividers
--div_opLat	9	Integer Divider latency
--div_pipelined	True	Integer Divider pipelined
--FP_ALU_count	3	Number of FP ALUs
--FP_ALU_opLat	1	FP ALU latency
--FP_ALU_pipelined	True	FP ALU pipelined
--FP_mult_count	2	Number of FP Multipliers
--FP_mult_opLat	3	FP Multiplier latency
--FP_mult_pipelined	True	FP Multiplier pipelined
--FP_div_count	2	Number of FP Dividers
--FP_div_opLat	64	FP Divider latency
--FP_div_pipelined	True	FP Divider pipelined

Table 24: gem5 configuration parameters

Following is an example of the standard command used to compile the benchmarks:

```
$ $LLVM/bin/clang++ --target=riscv64-unknown-elf -march=rv64g -mepi -O2 -
DUSE_RISCV_VECTOR -o kernel.e kernel.cc
```

And the next is an example of the command that was used to run the resulting executable with gem5 for one of the configurations:

```
$ ./build/RISCV/gem5.opt configs/example/se.py -c $APPS/kernel.e --cpu-type
DerivO3CPU  --caches  --l2cache  --cacheline_size  64  --l1i_size  32kB  --
l1d_size 32kB --l2_size 256kB --l2_assoc 16 --l1i_assoc 4 --l1d_assoc 4 --
num-l3caches 0 --sys-clock 2GHz
```

Since the simulations in gem5 take a long time to execute, we reduced the L2 cache size in order to force the simulations to stress the memory system and thus we can obtain meaningful results even for medium size datasets. This means that instead of simulating an input of 256 millions of elements to stress the memory, we can simulate only 2 millions to get our results.

4.2 Experimental Results

We perform our first simulations on a model of an out of order RISC-V core. During this deliverable we perform the Axpy kernel from the RIVEC benchmarks[2], changing the type of data from float to integer. This change was required because the support for floating point instructions is still work in progress. For this analysis we use three sizes of datasets, from 256 thousand data to 2,56 millions of integer numbers, and we cover vector lengths from 8 to 64 elements per vector. Since Axpy is a memory-bound kernel, a marginal speed-up increase is obtained as the vector lane size increases.

The results are shown next, in Figure 23:

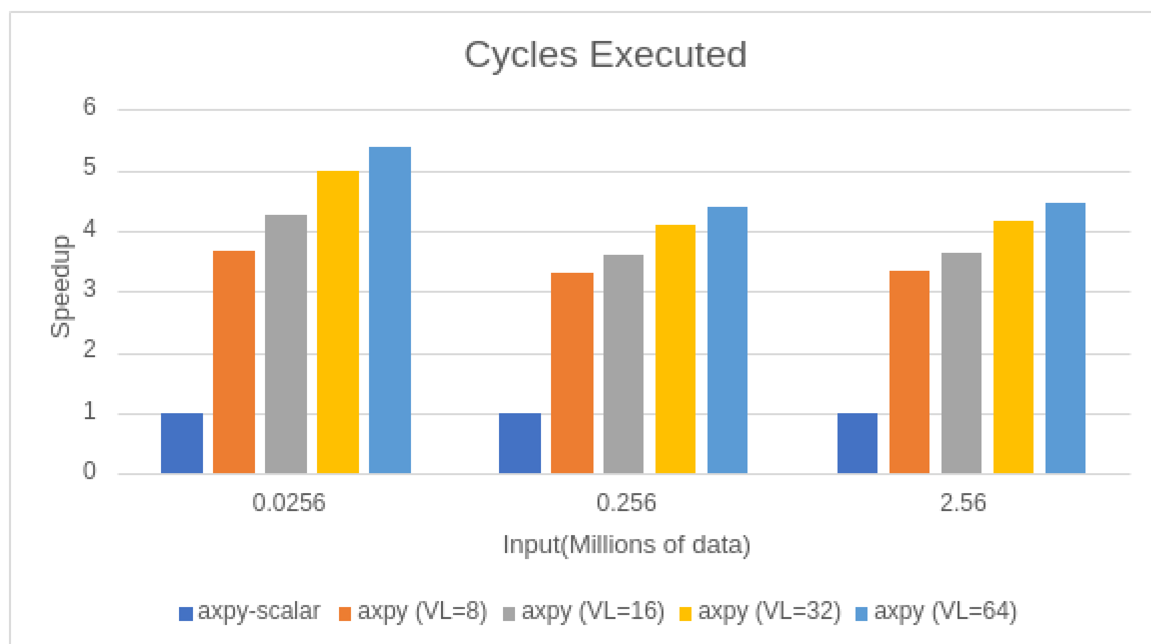

Figure 23: Number of cycles executed for scalar and for different configurations of vector lengths 8, 16, 32 and 64.

[2] "Rivec benchmark suite," https://github.com/RALC88/riscv-vectorizedbenchmark-suite, accessed: 2022-03-30.

In Figure 24, we show the number of executed instructions for the axpy kernel. We take the number of instructions executed for scalar axpy as a baseline and then we present the number of times that the number of cycles are reduced. For example, for the VL = 8 the number of executed instructions reduces by 6 times, for VL = 16 the number of executed instructions reduces by 16 times, for VL = 32 the number of executed instructions reduces by 24 times and finally for VL = 64 the number of executed instructions reduces by 48 times.

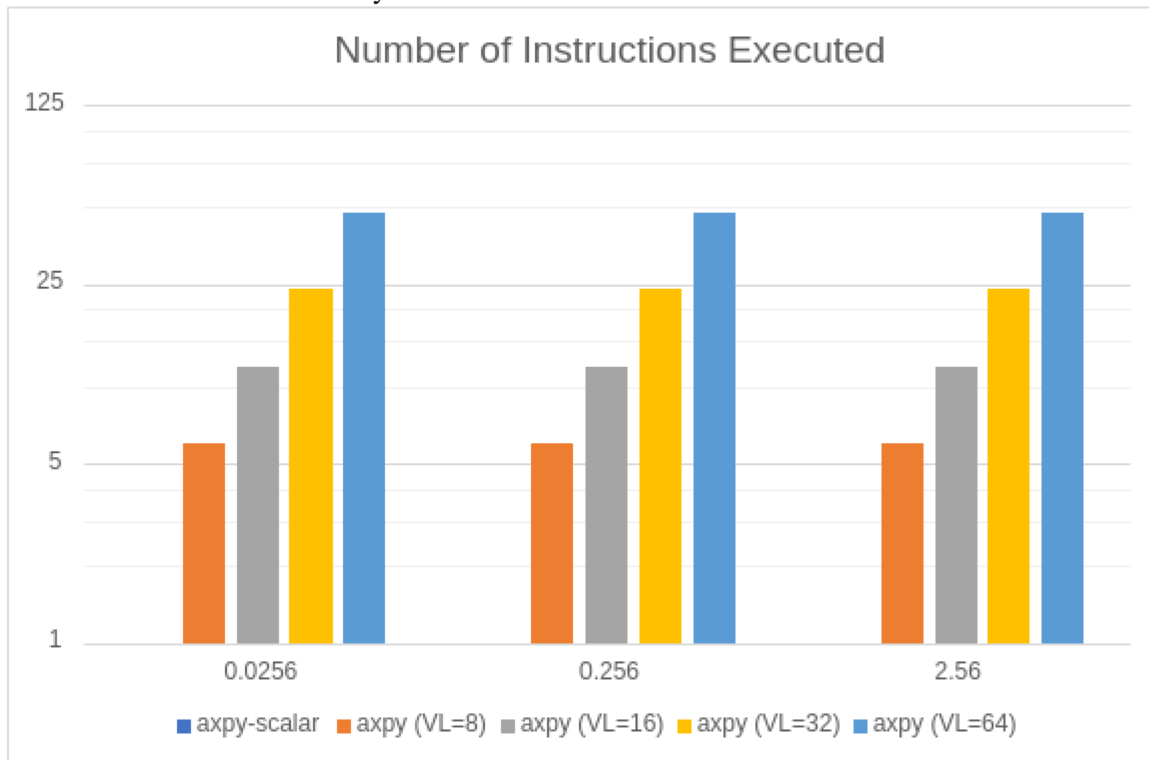

Figure 24: Number of executed instructions taking the scalar number of executed instructions as baseline.

5. Concluding Remarks and Next Steps

In this report we presented the simulation infrastructure that we developed, as well as the results that we received from the simulations that will be used to drive the architectural design decisions for the eProcessor chip.

Concerning the simulations of the scalar core, we explored how different microarchitecture component sizing decisions affect the overall runtime of a set of representative microbenchmarks.

Concerning the simulations of the vector accelerators, this first approach of the integration of the VPU to the OoO RISC-V established the first step to the integration of the whole RISC-V vector extension. The integration of the VPU will allow better architectural decisions in the design of both the VPU and the core. The main important achievements in this deliverable are listed below:

1. Support for the decoding of vector instructions.
2. Support for basic vector memory operations such as loads and stores with unit strided.
3. Support for fundamental vector arithmetic operations such as multiplication, addition, etc.
4. Add templates to include more core models besides the already supported DerivO3, Atomic and Minor.

Upcoming tasks for the scalar core simulations include:

1. Use a wider range of microbenchmarks from more domains
2. Run some simulations in Full-system emulation mode, i.e. with OS code
3. Explore more architectural components, such as branch predictor and others
4. Enhance gem5 configuration to better match the evolving core from Cortus

Upcoming tasks for the VPU simulations include:

1. Extend the vector decoder to cover the RISCV vector extension.
2. Support for more types of instructions.
3. Add the eProcessor core model as the OoO model integrated to the VPU.
4. Execute and Analyze more benchmarks from the RIVEC suit.

6. List of Abbreviations

ALU: Arithmetic-Logic Unit
Atomic: gem5 core model used to generate checkpoints and simulate in functional level.
DerivO3: gem5 core model of a generic Out of Order core.
FP: Floating Point
HPC: High Performance Computing
ISA: Instruction Set Architecture
Minor: gem5 core model of a generic in order core.
MSHR: Miss Status Holding Register
NAS: NASA Advanced Supercomputing
OoO: Out of Order.
RIVEC: is a suite of benchmarks that focuses on vector microarchitectures; nevertheless, it can be used as well for Multimedia SIMD microarchitectures.
SIMD: Single Instruction Multiple Data.
VL: Vector Length
VPU: Vector Processing Unit.