ECE 6373: Advanced Computer Architecture

Course Project

Simple Scalar
Sim-Outorder Model

Ali Sura Ozdemir

Abstract— In this project work, an evaluation of the CPU performance was analyzed according to the number of basic cache parameters, such as the L1 and L2 cache size, associativity, block sizes. All functional data was collected with simulators from the Alpha version of the Simple Scalar toolset, version 3.0. Various benchmarks (eon, bzip2 etc.) from the SPEC2000 suite were used to test. The results show the differences between usage of various benchmarks. For interval cache data, the simulators were modified to print stats every 100 million executed instructions. A combination of Perl and Tcsh scripts were used to launch, manage, and process the results of these simulations.

Index Terms—SimpleScalar, Sim-outorder, SPEC2000.

1 Introduction

Computer architects are constantly researching new techniques in microprocessor design. Instead of spending time and money on implementing the new designs in hardware, computer architects first test these techniques on simulators. SimpleScalar simulator toolkit, which was developed by SimpleScalar LLC. In computer terms, a benchmark is a special program that is used to characterize a computer system's, or, in this case, a microprocessor's, performance in executing the program. The theory is that a benchmark program is representative of real-world applications, so a measure of how well a system performs in the execution of the benchmark is indicative of what the system's performance with actual applications will be like. Different applications tax computers in different ways, so there is no way that a single benchmark could predict the performance of a computer system on all applications. For example, one benchmark might provide reasonable statistics on how well a computer system would run a word processor, but this data could be meaningless when attempting to predict the performance of the system when playing a graphically intensive computer game.

SIMPLE SCALAR v3.0 is an open source tool used to simulate real programs to obtain and analyze performance information of simulated microarchitectures. SimpleScalar was originally developed by Todd Austin at the University of Wisconsin Madison[1]. The SimpleScalar toolset provides an infrastructure for simulation and architectural modeling. The toolset can model a variety of platforms ranging from simple unpipelined processors to detailed dynamically scheduled microarchitectures with multiple-level memory hierarchies. For users with more individual needs, SimpleScalar offers a documented and well-structured design, which simplifies extending the toolset to accomplish most architectural modeling tasks. SimpleScalar simulators reproduce computing device operations by executing all program instructions using an interpreter. The toolset's instruction interpreters support several popular instruction sets, including Alpha, Power PC, x86, and ARM.[2].

The SPEC CPU2000 benchmark suite (http://www.spec.org/osg/cpu2000) is a collection of 26 compute-intensive, non-trivial programs used to evaluate the performance of a computer's CPU, memory system, and compilers[3]. In this project 10 SPEC2000 benchmarks were used in the analysis.

2 Architecture and Methodology

2.1 Architecture

SimpleScalar can simulate Alpha and PISA (Portable ISA). Others are being added to the SimpleScalar. The PISA instruction set is a simple MIPS-like instruction set maintained primarily for instructional use. The tool set takes binaries compiled for the SimpleScalar architecture and simulates their execution on one of several provided processor simulators. The machine running SimpleScalar is called the Host machine or Host while the ISA that one is targeting such as Alpha or PISA is called Target. Gcc cross-compiler for PISA is available on the internet. Cross-compiler creates a sort of illusion that the underlying machine where PISA instructions are executed is a PISA machine while it is not. The following figure 1 illustrates this notion.

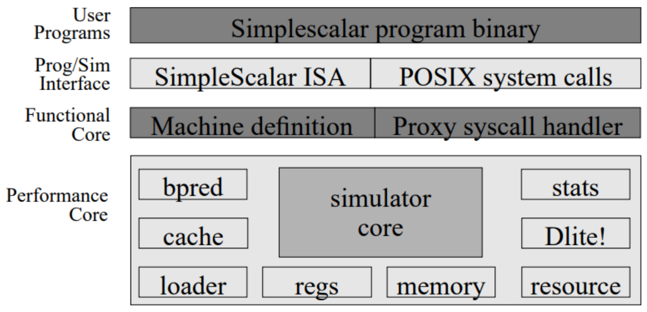


Figure 1: Simulator Structure [5]

There are several available simulator models in SimpleScalar Architecture:
sim-safe: slightly slower instruction interpreter, as it checks for memory alignment and memory access permission on all memory operations. This simulator can be used if the simulated program causes sim-fast to crash without explanation.
sim-profile: instruction interpreter and profiler. This simulator keeps track of and reports dynamic instruction counts, instruction class counts, usage of address modes, and profiles of the text and data segments.
sim-cache: memory system simulator. This simulator can emulate a system with multiple levels of instruction and data caches, each of which can be configured for different sizes and organizations. This simulator is ideal for fast cache simulation if the effect of cache performance on execution time is not needed.
sim-bpred: branch predictor simulator. This tool can simulate difference branch prediction schemes and reports results such as prediction hit and miss rates. Like sim-cache, this does not simulate accurately the effect of branch prediction on execution time.
sim-outorder: detailed microarchitectural simulator. This tool models in detail and out-of-order microprocessor with all of the bells and whistles, including branch prediction, caches, and external memory. This simulator is highly parameterized and can emulate machines of varying numbers of execution units.[4].

In this project sim-outorder model was used.
In terms of benchmarks, CFP2000 contains 14 applications (6 Fortran-77, 4 Fortran-90 and 4 C) that are used as benchmarks, in this project 5 of them were used:

Name	Ref Time	Remarks
168.wupwise	1600	Quantum chromodynamics
171.swim	3100	Shallow water modeling
177.mesa	1400	3D Graphics library
178.galgel	2900	Fluid dynamics: analysis of oscillatory
183.equake	1300	Finite element simulation; earthquake modeling

CINT2000 and CFP2000 are based on compute-intensive applications provided as source code. CINT2000 contains eleven applications written in C and 1 in C++ (252.eon) that are used as benchmarks, in this project 5 of them were used as below.

Name	Ref Time	Remarks
176.gcc	1100	C compiler
181.mcf	1800	Minimum cost network flow solver
186.crafty	1000	Chess program
252.eon	1300	Ray tracing
256.bzip2	1500	Data compression utility

So, total of 10 Benchmarks were used.

2.2 Methods

For the actual characterization, the sim-outorder simulator from the toolkit was used. The compilation was done with the SimpleScalar toolkit on Ubuntu Linux and configured it to simulate programs using the Alpha instruction set. Compiling the benchmarks enabled them to be simulated by the SimpleScalar simulator. To compile the default Makefiles was utilized which came with the benchmarks. Sometimes, the conversion of the cache configurations that was needed to use into command line parameters for sim-outorder were implemented in the .config file. Several shell scripts were experimented to automate the simulation of benchmarks. After that a benchmark could be simulated by sim-outorder without problems, scripts were also used to simulate the benchmark with sim-outorder and the .config files. After each simulation run, sim-outorder outputted the results to a different file. When all the runs were finished, the output was parsed to the files for the statistics that is needed into a form that enabled to easily copy and paste them into an Excel spreadsheet. Finally, depicting the values from the numbers to graph were implemented and analyzed the data.
After copy files in Simulator and Benchmark; below, there is a sample command to run the sim-order for the benchmarks:

- run the sim-outorder configuration for the bzip benchmark and save the output to the text file:

```
./sim-outorder -config ../default.cfg -redir:sim outputs1.txt -fastfwd
100000000 -max:inst 100000000 bzip200.peak.ev6 input.source 58
```

3 Experimental Results

Shell script was run for each configuration on ten different SPEC2000 benchmarks. The benchmarks used were bzip, crafty, eon, equake, galgel, gcc, mcf, mesa, swim and wupwise.
As outputs of the experiments, since there are tens of parameters to be compared, here 7 of them are selected for each question. They are:

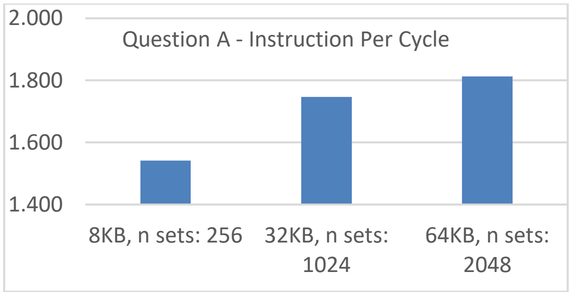
- Instructions per cycle : sim_IPC
- Total number of loads executed : sim_total_loads
- Total number of stores executed: sim_total_stores
- Instruction miss rate (i.e., misses/ref) : il1.miss_rate
- Data miss rate (i.e., misses/ref): dl1.miss_rate
- Unified miss rate : ul2.miss_rate
- Total simulation time in cycles: sim_cycle

All graphs show the average of 10 pre-defined Benchmarks as stated above.

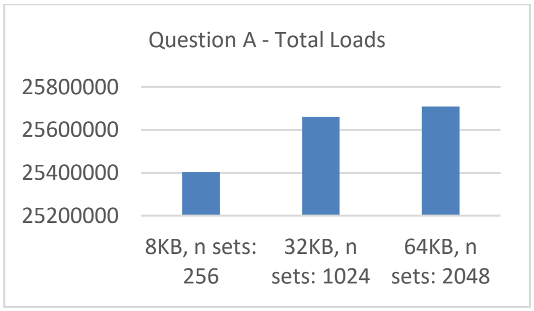
3.1 Analysis – Question A

Comparison of Instruction cache sizes (8KB, 32 KB and 64KB, block size and associativity)

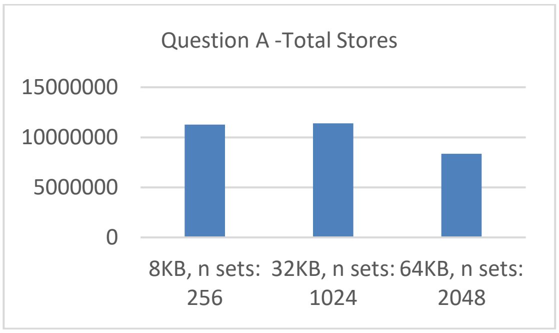
Below the graph result shows the average of 10 benchmark values depicting as Instruction cache size increase then Instruction per cycle (IPC) also increases.



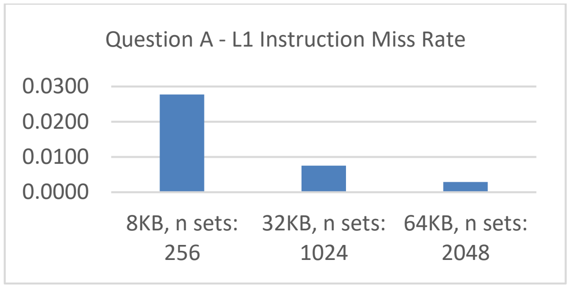
As L1 instruction cache size increase, loads executed increase by a negligible amount.



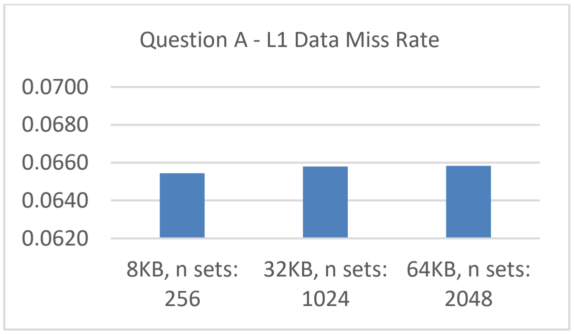
As L1 instruction cache size increase, stores executed decrease by a negligible size.



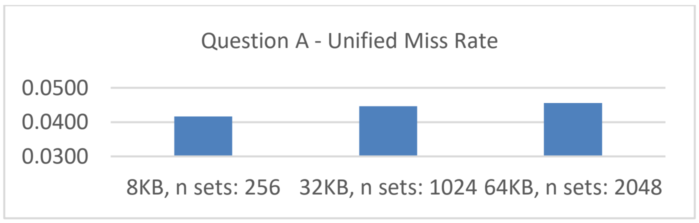
As L1 instruction cache size increase, L1 instruction cache miss rate reduces below.



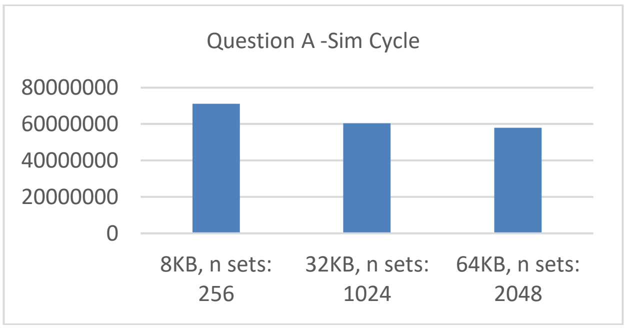
As L1 instruction cache size increase, L1 Data cache miss rate almost stay constant except a bit increase once passing to 32KB as below.



As L1 instruction cache size increase, L2 Unified cache miss rate slightly increase but in general stay constant as below.



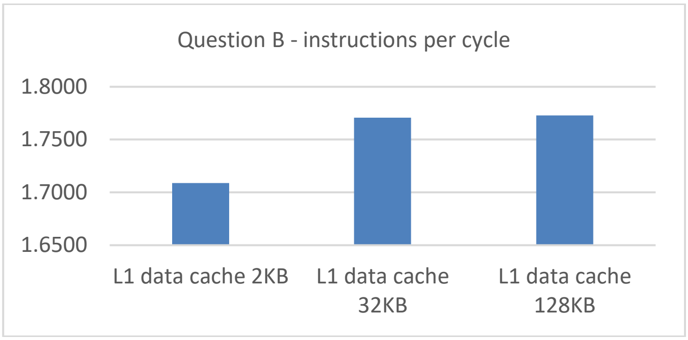
As L1 instruction cache size increase, total simulation time cycle decreases as below.



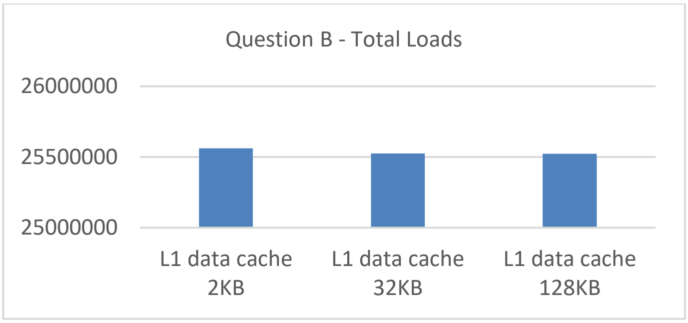
3.2 Analysis – Question B

Data cache size (compare 2KB, 32KB, and 128KB, all 4-way set associative, block size)

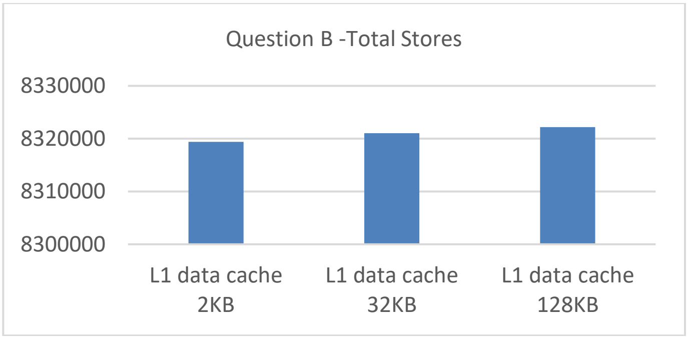
As L1 data cache size increase, there is a slightly IPC increase from 2KB to 32KB then remains constant.



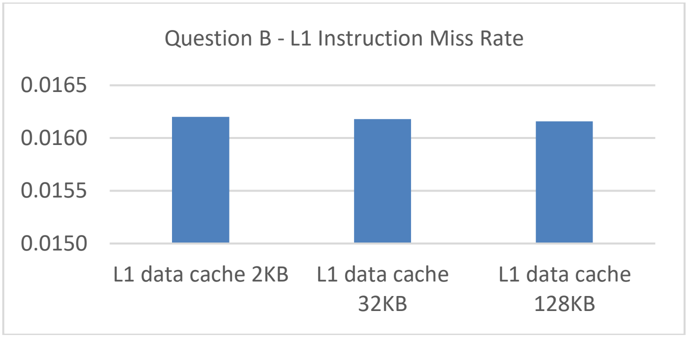
As L1 data cache size increase, loads executed stays constant.



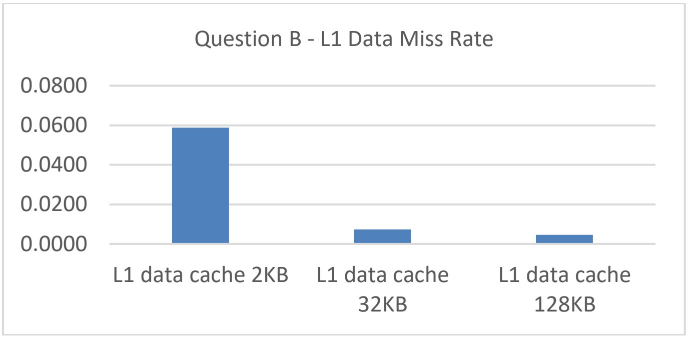
As L1 data cache size increase, stores executed stays constant.



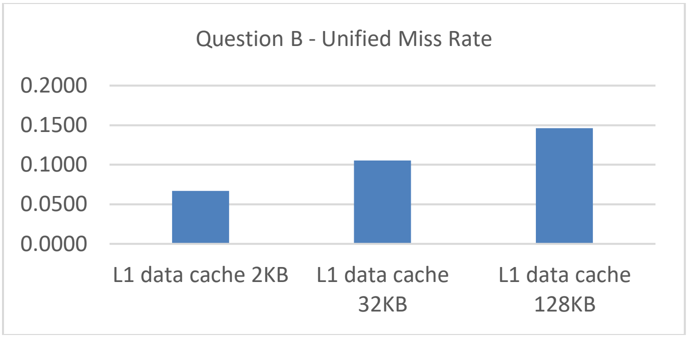
As L1 data cache size increase, L1 instruction cache miss rate stays constant as below.



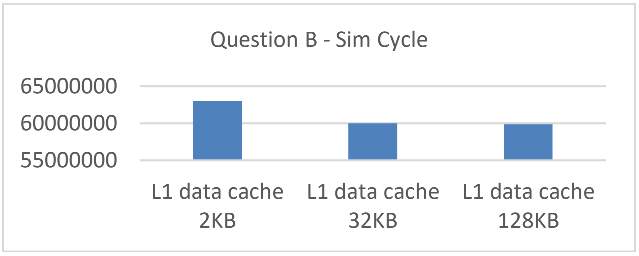
As L1 data cache size increase, L1 data cache miss rate decrease.



As L1 data cache size increase, Unified L1-L2 miss rate increase.



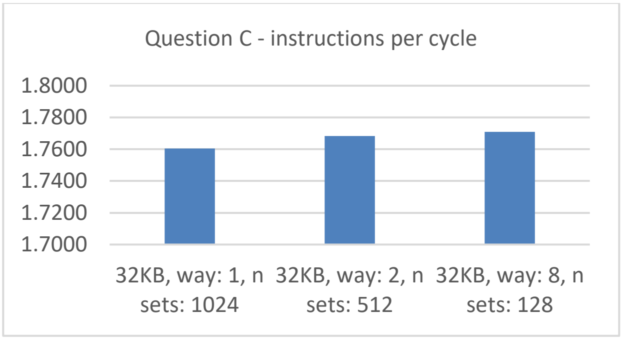
As L1 instruction cache size increase, total simulation time cycle decreases as below.



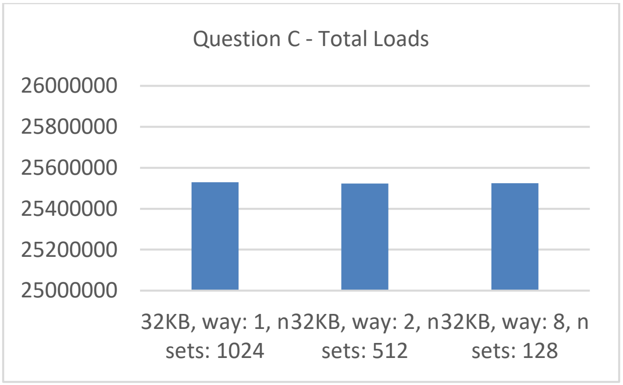
3.3 Analysis – Question C

Data cache associativity (compare 1-way, 2-way, 8-way, all 32KB, block size)

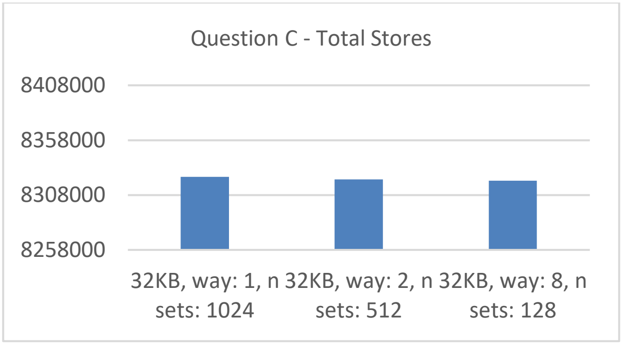
As L1 data cache associativity increases, instruction per cycle stays constant.



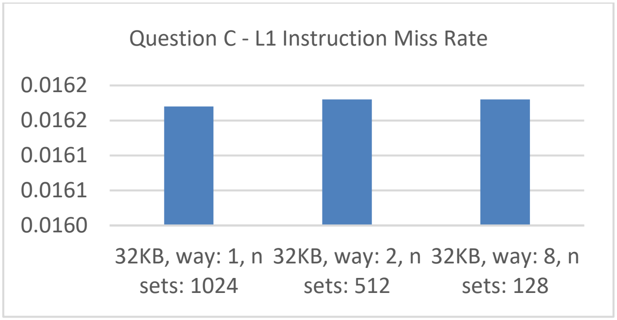
As L1 data cache associativity increases, Loads executed remains constant.



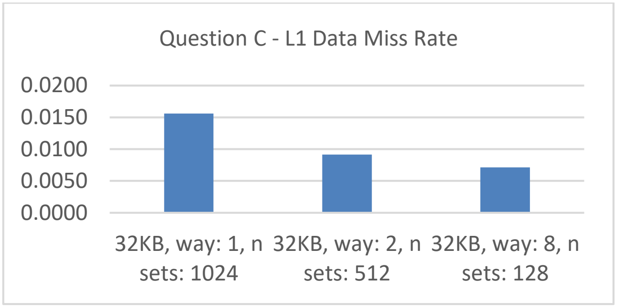
As L1 data cache associativity increases, stores executed remains constant.



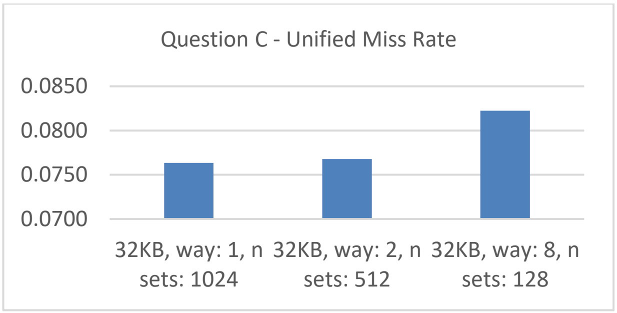
As L1 data cache associativity increases, L1 instruction cache miss rate stays constant.



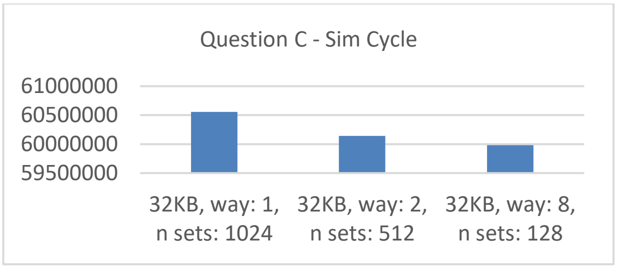
As L1 data cache associativity increases, L1 data cache miss rate reduces.



As L1 data cache associativity increases, unified miss rate increases.



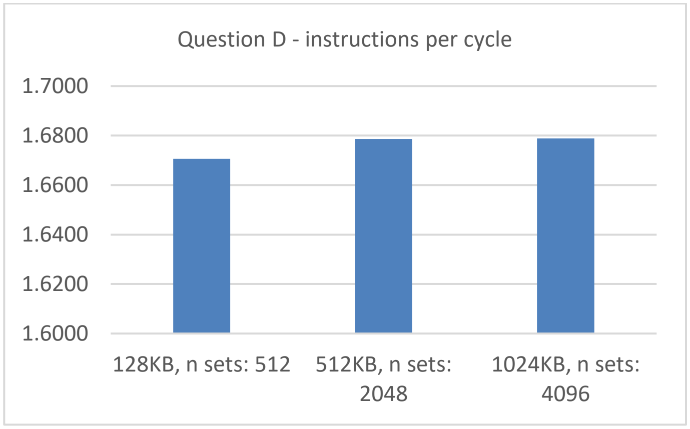
As L1 data cache associativity increases, total simulation time cycle slightly reduces.



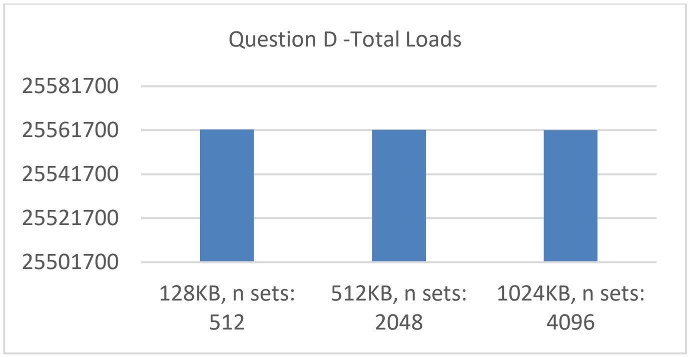
3.4 Analysis – Question D

L2 cache size (compare 128KB, 512KB, and 1MB, block size and associativity)

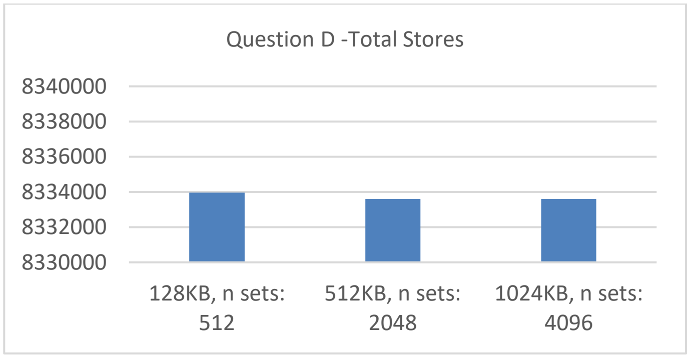
As L2 data cache size increases, instruction per cycle insignificantly slightly increase from 128KB to 512KB and then stays constant.



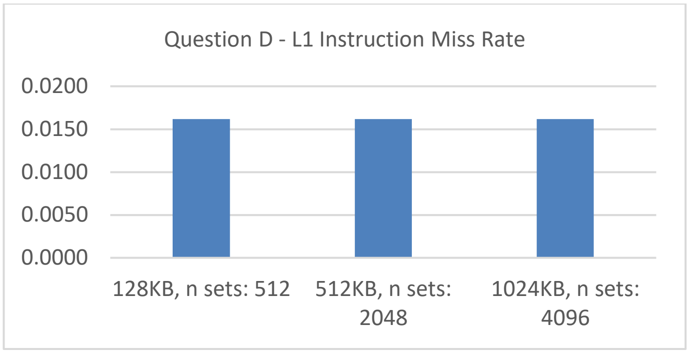
As L2 data cache size increases, total loads of execution stay constant.



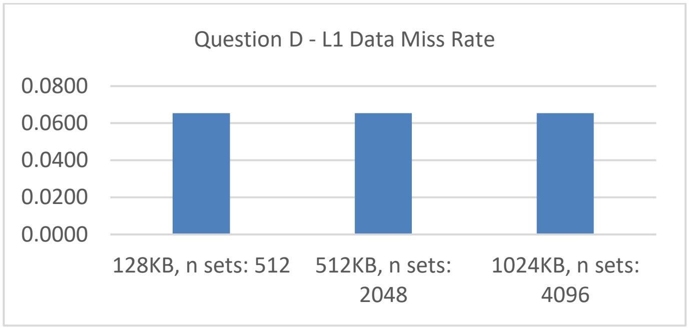
As L2 data cache size increases, total stores of execution stay constant.



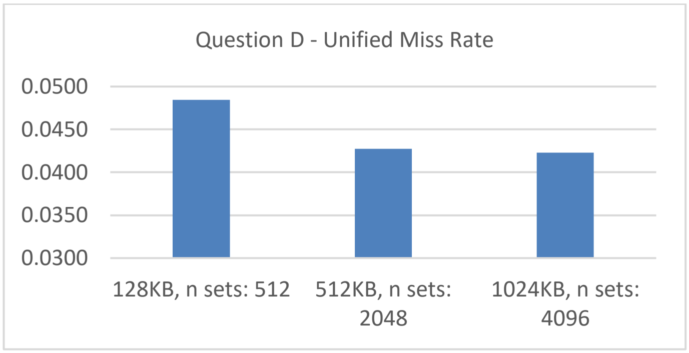
As L2 data cache size increases, L1 instruction miss rate stays constant.



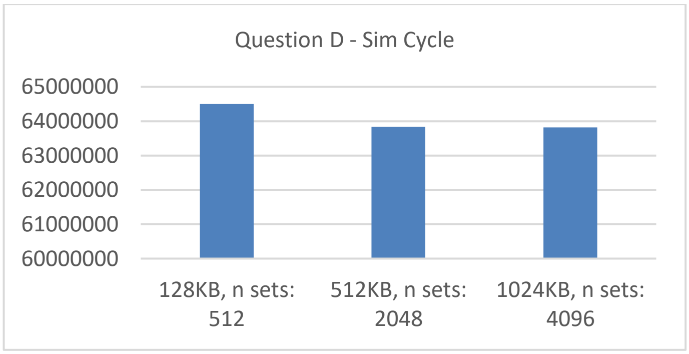
As L2 data cache size increases, L1 data miss rate stays constant.



As L2 data cache size increases, Unified miss rate slightly decreases from 128KB to 512KB.



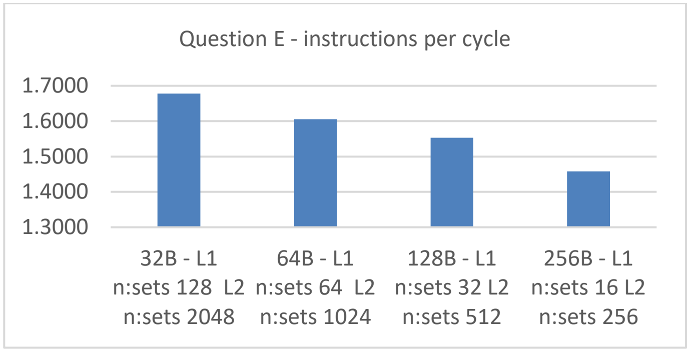
As L2 data cache size increases, total simulation time cycle insignificantly slightly decrease from 128KB to 512KB and then stays constant.



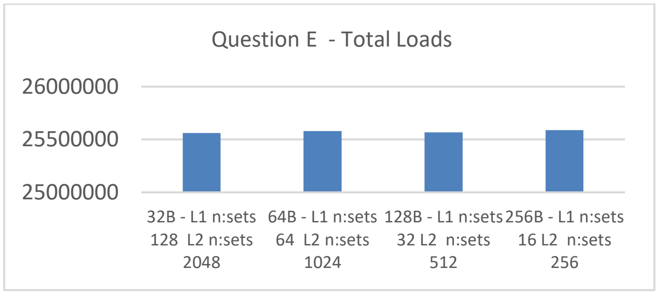
3.5 Analysis – Question E

L1 and L2 block size (compare 32B, 64B, 128B, and 256B, cache size and associativity)

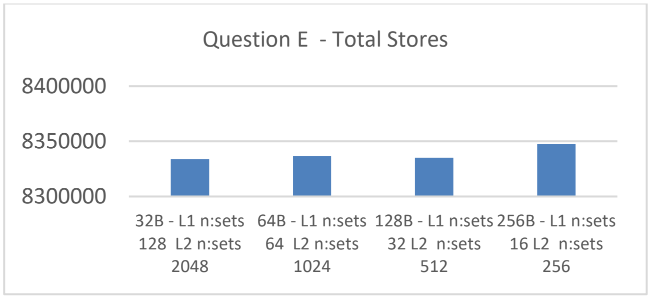
As block size increase, instruction per cycle reduce.



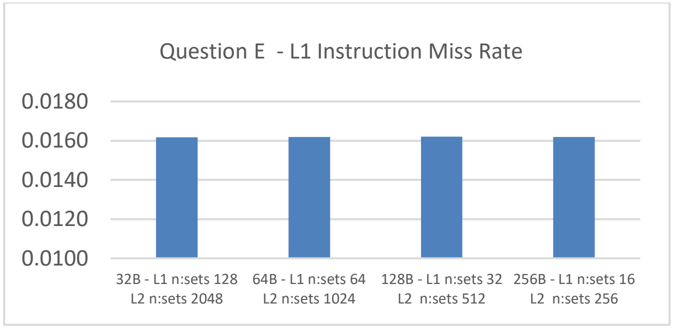
As block size increase, total loads executed remains constant.



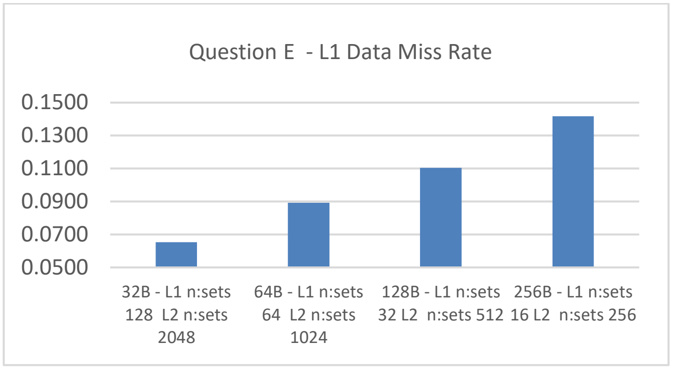
As block size increase, total stores executed remains constant except slightly increase on 256B.



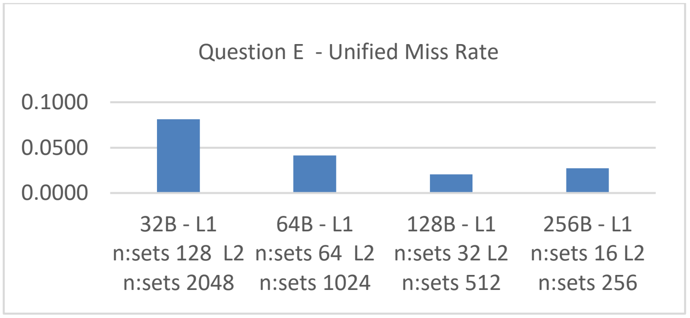
As block size increase, L1 instruction miss rate stays constant.



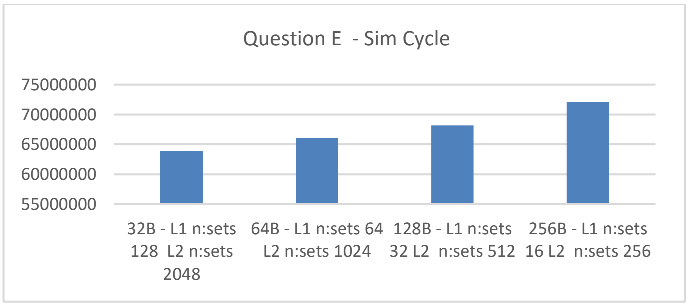
As block size increase, L1 data miss rate increase.



As block size increase, unified miss rate increase.



As block size increase, total simulation time cycle increase.



REFERENCES

[1] Doug Burger and Todd M Austin. The SimpleScalar tool set, version 2.0, ACM SIGARCH Computer Architecture New 25.3 1997. pp. 13-25.

[2] Todd Austin, Eric Larson, Dan Ernst. Infrastructure for Computer System Modeling, publication at: https://www.researchgate.net/publication/2955585 SimpleScalar: March 2002

[3]Jason F. Cantin, Cache Performance for SPEC CPU2000 Benchmarks https://research.cs.wisc.edu/multifacet/misc/spec2000cache-data/old_index.html

[4]SimpleScalar http://www.ecs.umass.edu/ece/koren/architecture/Simplescalar/SimpleScalar_introduction.htm