Characterizing PBBS on single-core systems

Henry Chen, Runyi Li, Nicholas Pedro, Julie Tang

I. Introduction

The Problem Based Benchmark Suite (PBBS) is a collection of benchmark problems defined in terms of I/O specification with varying algorithms and implementations [1]. This paper aims to provide a workload characterization of the PBBS benchmark suite for single-core systems. We explore different configurations of input set, we contrast in-order and out-of-order pipelining, and we investigate different cache configurations.

In this study, we follow similar methodologies to previous works which have characterized OLTP Workloads [3], and explored Instruction-level Parallelism [2]. Thus, we selected from the PBBS library a subset of 5 benchmarks to focus our exploration on, which is outlined in Table 1. We evaluate comparison sort to emphasize the importance of implementation when exploring in-order versus out-of-order execution. Word count, breadth first search, and Delaunay Triangulation represent fairly uniquely structured workloads, and were selected with the aim of providing a fair assessment that reflects a variety of practical workloads.

TABLE I
Benchmarks Measured

Benchmark Name	**Benchmark Statistics**	
	Instruction Count	***Operations***[1]
Word Count (serial)	29080069	63816235
Breadth First Search (serial)	86938715	159422813
Comparison Sort (sample)	77372890	132986160
Comparison Sort (serial)	55484925	94321435
Delaunay Triangulation (serial)	196717076	349350858

[1]Including micro ops.

II. Exploration 1

In this section, we sweep input sizes to evaluate the relationship between input sizes and key metrics in all three regions of execution: before, after, and the region of interest. Across varying benchmarks, we expect that increasing input size should increase the number of cycles and instructions executed, whereas the CPI should remain mostly constant in all three regions. We also attempt to explain the importance of the region of interest and why we (usually) only report stats from there. We propose that the stats from the before and after regions are either negligible or overwhelming.

A. Experimental setup

Simulations are conducted on each benchmark multiple times with varying input sizes on the same gem5 configuration to ensure fairness. The system uses the MinorCPU to model an in-order processor, 8GB of DDR3_1600_8x8 main memory, a processor frequency of 75Mhz, and executes one instruction at a time (See Appendix B).

Inputs to the benchmarks were generated for values of $n \in \{2^{11}, 2^{12}, 2^{13}, 2^{14}\}$ using the input generation scripts included with PBBSbench (see Appendix A). We sweep these fixed values of n across benchmarks. For both Comparison Sort benchmarks, n represents the number of integers generated. For Delaunay Triangulation, n represents the number of 2-D points. For BFS, n represents the number of vertices and a quarter of the number of edges. Finally, for Word Count, n represents the number of characters in the trigram string.

All benchmarks were run using gem5, by supplying gem5 with a direct path to each compiled binary file and the appropriate arguments (See Appendix C). For each benchmark, all arguments except the input file were fixed.

By using the same system configuration and arguments across different sweeps, we intend that all variability results from the size of the inputs. This ensures that any trends we observe in the stats are correlated to the input size. After analyzing benchmark performance, it was found that the MinorCPU configuration with gem5 does not correctly report instruction mixes; hence, we refrain from analyzing this characteristic.

B. Results and discussion

The number of cycles is a key metric, as it correlates to the amount of time and resources exhausted during the benchmark's execution. In Figure 1 (See appendix F), we see that the number of cycles in the region of interest and after region approximately doubles for each double in input size (a linear trend). However, in the before region, that pattern is still observed for more complex algorithms such as Delaunay and BFS, while less complex algorithms exhibit an attenuated increase. This tells us that the algorithms most likely utilize resources efficiently, as increased input size did not introduce new cost-overheads, since the number of simulated cycles increases linearly with respect to the input size. Moreover, we also see that the number of cycles in the before region is significantly greater than in the other two regions. This suggests that if the number of cycles was reported as the total of the three regions, the inclusion of the initialization stage would inflate cycle count and dilute the contribution of the algorithm itself. Hence, to properly evaluate the performance of the algorithm itself, it is necessary to only examine the cycle count in the region of interest.

The CPI is another key metric, since it measures the average number of cycles required to complete each instruction, which translates to how efficiently the processor handles each instruction of the algorithm. Because processor frequency is held constant across benchmarks, to achieve the desired

characteristic of decreased simulation time, we must decrease CPI. In Figure 2 (See appendix F), we see that the CPI in all regions of execution remains mostly constant for all benchmarks and input sizes. This suggests that the input size has a negligible impact on the rate at which instructions are processed, increase input size does not introduce new overheads. Figure 2 also shows that each stage has a different CPI; thus, the CPI of the entire benchmark execution would be a weighted average based on the number of cycles each stage took. From Figure 1 (See appendix F), it is clear that this weighted average could be skewed towards the before or after regions. Thus, examining CPI across the entire benchmark execution is not a good representation of the algorithm's actual CPI.

All in all, the ability of gem5 to separate execution stats into before, after, and the region of interest is critical for accurate algorithm analysis. Key metrics such as CPI and cycle count can be misrepresented if the total execution is taken into account. Consequently, it is advantageous to only report the region of interest.

III. Exploration 2

In this section, we contrast the performance of using an in-order pipeline against an out-of-order pipeline. Across varying benchmarks, we expect the key metric of CPI to decrease when utilizing out-of-order execution. Moreover, the extent of performance gain by using out-of-order execution should vary between benchmarks depending on its implementation.

A. Experimental setup

To better understand the impact on performance of in-order and out-of-order processors, we use a simulation based methodology across varying benchmarks. In this section, we describe the processor and system model, the input generation procedure, and the simulation infrastructure.

Simulations are conducted on each benchmark separately using two different virtual systems constructed within the gem5 simulator. One system uses the MinorCPU, which models an in-order pipeline processor. The other system uses the O3CPU, which models an out-of-order pipeline processor. Other characteristics are equivalent between the two system to ensure fairness in evaluation. Namely, each system uses 8GB of DDR3_1600_8x8 main memory, each processor is clocked at 75Mhz, and each processor is configured to execute one instruction at a time (See Appendix B).

Inputs to the benchmarks were generated when necessary using the input generation scripts included with PBBSbench (see Appendix A). Inputs were held constant between simulations using the in-order system and the out-of-order system to ensure fairness. As discussed in Exploration 1, input sizes were chosen so that simulations could be completed within 2-3 minutes each.

Finally, all benchmarks were run using gem5, by supplying gem5 with a direct path to each compiled binary file and the appropriate arguments (See Appendix C). For each benchmark, the supplied arguments were held fix, and only the system binary file was changed to run each of the two systems

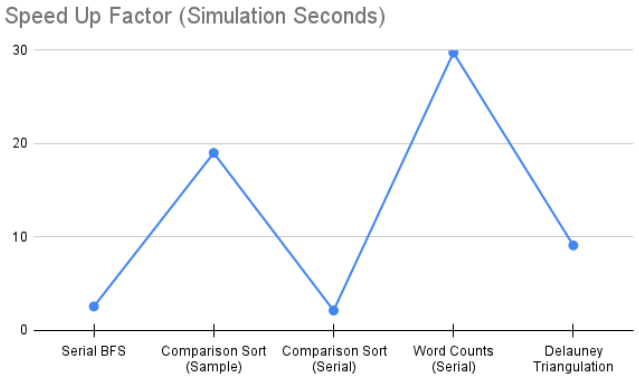


Fig. 1. Speed Up Factor In Simulation Time.

once. This study makes a simplifying assumption, namely that results should (more-or-less) not change between different passes of the simulation using gem5 when supplied with the exact same binary files and arguments. After running the gem5 simulation of the benchmark, results were saved in the stats.txt file. In this file, we examined the second dump as the region of interest, as we are concerned with the system's performance on the benchmark itself, which occurs after initialization has ended, but before de-initialization begins.

B. Results and discussion

In this section, we discuss the overall trend of out-of-order execution surpassing the performance of in-order execution, and the variance in performance gains between different workloads.

In figure 2, we see that using out-of-order execution achieved speed-up in simulation time by a factor of 2 to 30 depending on the workload. Moreover, Fig. 3 illustrates that processor CPI decreases across the board when using an out-of-order pipeline (Appendix D). CPI is the key metric of comparison between these two systems, as reduced CPI directly translates to reduced execution time. Since processor frequency is constant between both systems, spending less clock cycles per instruction results in more instructions being executed per unit of time, improving overall system performance. This decrease in CPI can be explained by out-of-order execution allowing independent instructions following a stalled instruction to potentially still be executed, reducing wasted cycles spent stalling. Hence, out-of-order execution provides the possibility of executing more instructions where an in-order pipeline would stall completely, allowing for potentially decreased CPI, which is confirmed by our observations across all benchmarks. Naively, this suggests that out-of-order execution is always preferable to in-order execution across our workloads.

However, this conclusion does not take into consideration the (potentially significant) increase in cost and complexity to implement an out-of-order pipeline in hardware. Thus, the choice of which system to choose would need to weigh the increased cost against performance gains which varied across benchmarks.

Hence, based on the intended workload, accurate estimations of the anticipated speed-up achievable by using an out-of-order pipeline must be obtained to make a well-informed decision. Curiously, our observations showed significant variance in speed-up between different workflows, even between different implementations of the same sorting algorithm.

For instance, figure 2 suggests that using an out-of-order processor for the serial version of comparison sort yielded significantly lower speed-up gains than the sample version of comparison sort. Examining the source code, the key difference between the two implementations is that comparison sort in series is implemented in-place, where as the sample version is not. A potential explanation is that by sorting in-place, we increase the likelihood of register conflicts when reading and writing, leaving less potential for instructions following a stalled instruction to themselves not be in conflict. Consequently, we are not able to leverage the advantages of out-of-order execution to the same extent as the non-in-place implementation.

IV. Exploration 3

In this section, we will be comparing the performance between different cache configurations, by looking at two different sweeps: one for cache size, and the other for cache associativity. We expect a lower miss rate (higher hit rate) to caches with a larger size and a higher associativity. Additionally, the results in performance should vary between benchmarks due to its implementation.

V. Experimental Setup

To evaluate how **cache size** and **cache associativity** impact performance, we conduct two separate "cache sweeps" in the gem5 simulator. All simulations employ a single system configured with an out-of-order `O3CPU`, and the remaining system parameters (e.g., memory latency, CPU clock rate) match those described in Exploration 2. Unless otherwise stated, the cache settings follow the baseline Python script shown in Listing 1:

For the *size sweep*, we create three variants by modifying the `size` parameter in these classes:

- A **base system** that matches the default script (`4kB` L1, `32kB` L2).
- A **doubled-L1 system**, where both L1 instruction and data caches have their `size` doubled (`8kB`), while the L2 remains `32kB`.
- A **doubled-L2 system**, where the L2 is set to `64kB`, and both L1 caches remain `4kB`.

For the *associativity sweep*, we instead adjust the `assoc` parameter. One configuration sets associativity to **1** (direct mapped cache), while the other uses **fully associative** caches, computed by:

$$\text{assoc} = \frac{\text{Cache Size}}{\text{Cache Line Size}}.$$

Hence, for a `4kB` L1 with a `64B` line size, the number of lines is $\frac{4096}{64} = 64$, yielding a fully associative L1. The same principle is applied to the L2: its associativity is set to $\frac{\text{L2 Size}}{\text{Cache Line Size}} = 512$. We run each benchmark with these two extremes—**direct mapped** and **fully associative**—across the same battery of simulations.

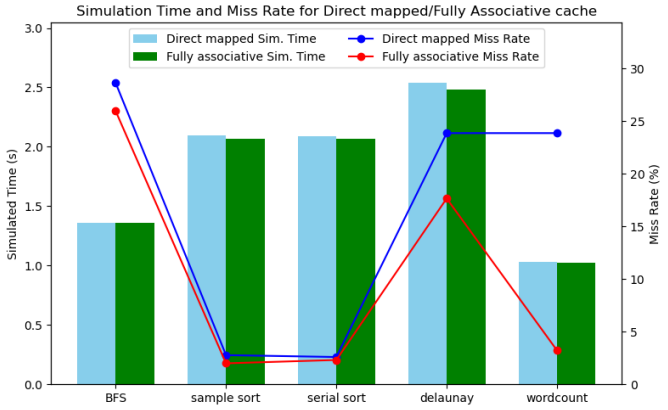


Fig. 2. Simulation Time and Miss Rate for Direct mapped/Fully Associative cache.

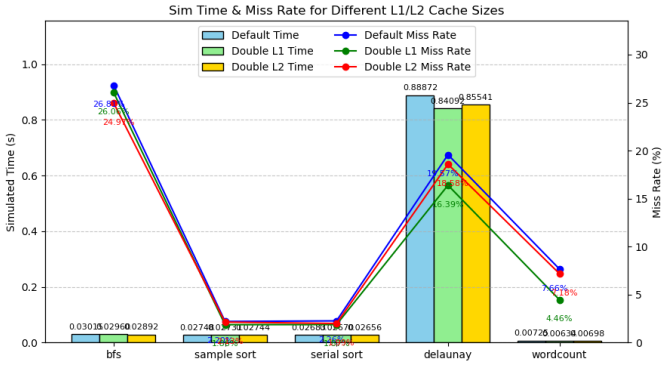


Fig. 3. Simulation Time and Miss Rate for Different Cache Sizes.

After each simulation completes, we gather statistics from the *second gem5 stats dump*, focusing on the *miss rate* (i.e., total misses ÷ total accesses) across the L1 instruction/data caches and the L2 cache. These statistics let us quantify how changes in cache size or associativity affect overall hits, misses, and simulated runtime for every benchmark.

You may see all scripts used in https://github.com/LatinScribe/CSC368-A1. //

A. Results and discussion

The results in Figure 4 and Table II in Appendix E show two main observations about direct-mapped caches (associativity = 1) and fully associative (associativity = number of lines). First, transitioning from direct-mapped to fully associative cache designs consistently reduces miss rates across all evaluated benchmarks, though the magnitude of improvement depends on the workload of benchmarks. Notably, serial BFS and serial wordcount demonstrate substantial reductions - serial BFS decreases from 29% to 26% while wordcount achieves a dramatic drop from 24% to 3% - suggesting that these applications experience frequent conflict misses under direct-mapped configurations. In contrast, the sample sort and the serial sort maintain low miss rates below 3% even with direct

mapping, resulting in minimal improvement (under 0.5%) when switching to full associativity.

Despite these miss rate improvements, the corresponding execution time reductions remain negligible for most workloads. Serial BFS exhibits a 2.6% absolute miss rate reduction (28. 6% to 25. 97%) but maintains identical execution times of 1.36 s in both configurations. Similarly, the improvement in the absolute miss rate of the serial word count 21% only reduces the runtime by 0.01s (1.03s to 1.02s). This indicates that when miss penalties cannot be fully hidden or overlapped, reducing conflict misses has a clearer impact on overall performance.

However, select applications demonstrate measurable performance gains. The delaunay (triangulation) benchmark shows a reduction in the absolute miss rate 6% (24% to 18%) accompanied by a 2. 4% run-time improvement (2.54 to 2.48 s). This implies that workloads with insufficient instruction-level parallelism or limited memory concurrency derive greater benefit from conflict miss reduction.

The observed variations align with theoretical principles of cache associativity: direct-mapped caches impose strict block-to-line mappings that induce conflict misses for data with temporal locality but non-sequential access patterns, while fully associative designs eliminate such constraints through flexible block placement.

As summarized in Figure5 and Table III in appendix E, enlarging the L1 or L2 caches impacts miss rates and execution time in different ways, depending heavily on the memory access patterns of the benchmarks. BFS experiences a gradual decrease in the miss rate (26. 8% 26,. 1% 25. 0%) and a modest reduction in runtime (0.03015 s 0.02960 s 0.02892 s) as we move from default to doubled L1 and doubled L2, indicating that while BFS can benefit from added capacity, its overall performance remains limited by other factors in the pipeline or memory system.

In contrast, sample sort and serial sort both exhibit very low baseline miss rates (around 2%); consequently, doubling L1 leads to only minor improvements (e.g., 2.20% $\rightarrow$ 1.86% in sample sort) and essentially negligible changes in execution time (around 0.027–0.028s). Notably, sample sort sees its miss rate dip when L1 is enlarged but rise slightly when only the L2 is doubled, implying that its working set benefits more from a larger L1 than from additional L2 capacity.

For delaunay, however, an enlarged L1 yields a pronounced advantage: the miss rate drops from 19.57% to 16.39%, and the runtime improves from 0.8887s to 0.8409s. Doubling L2 alone also helps, but less so (18.58% miss rate, 0.8554s). This behavior suggests that conflict or capacity misses in the L1 are significant for delaunay, and alleviating them leads to a more visible overall speedup. Meanwhile, wordcount stands out for seeing a strong miss-rate improvement from doubling the L1 (7.66% $\rightarrow$ 4.46%) but an unexpected deterioration when only the L2 is enlarged (18.37%), underscoring that a larger L2 may not always align well with certain access patterns.

In summary, while larger caches generally reduce misses—especially at the L1 level—these results underscore that each benchmark's behavior is highly context-dependent. Some (e.g., BFS, delaunay) clearly benefit from more capacity in terms of both miss rate and runtime; others (e.g., sample sort, serial sort) are already fairly optimized for the default sizes, so additional capacity yields marginal gains. Wordcount demonstrates that an enlarged L2 can even introduce performance anomalies depending on how data is reused. Hence, these findings reinforce that "bigger is not always better," and that a careful balance between L1 and L2 sizing, along with each application's memory footprints, is crucial for optimal performance.

VI. Conclusion

Overall, characterizing the Problem Based Benchmark Suite for single-core configured systems using the gem5 simulator provided valuable insight into the effects of input size and the importance of the region of interest. It also exhibited the generalized benefit of out-of-order pipelining over in-order pipelines, while emphasizing the importance of analyzing source code implementation when estimating potential speed-up gains. Finally, the workload characterization showed that while larger caches often decreased misses, the best solution is a balance between L1 and L2 sizing depending on the memory footprint of the intended workload.

References

[1] Anderson, Daniel, et al. "The problem-based benchmark suite (PBBS), v2." Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. 2022.

[2] Wall, David W. "Limits of instruction-level parallelism." Proceedings of the fourth international conference on Architectural support for programming languages and operating systems. 1991.

[3] Keeton, Kimberly, et al. "Performance characterization of a quad Pentium Pro SMP using OLTP workloads." Proceedings of the 25th annual international symposium on Computer architecture. 1998.

[4] Sánchez, Friman, et al. "Performance analysis of sequence alignment applications." 2006 IEEE International Symposium on Workload Characterization. IEEE, 2006.

APPENDIX A: INPUT SCRIPTS

The following scripts call PBBS supplied input generators, which is used to generate the data provided as inputs to the benchmarks (you may need `trigrams.txt` to be located in the directory you run these commands in):

- `/u/csc368h/winter/pub/workloads/pbbsbench/testData/sequenceData/randomSeq -t int -r 512 2048 randomSeq_512_2K.txt`
- `/u/csc368h/winter/pub/workloads/pbbsbench/testData/geometryData/randPoints -d 2 2048 randPoints2D_2K.txt`
- `/u/csc368h/winter/pub/workloads/pbbsbench/testData/graphData/randLocalGraph -j -d 3 -m 8192 2048 randomAdjGraph_8K_2K.txt`
- `/u/csc368h/winter/pub/workloads/pbbsbench/testData/sequenceData/trigramString 2048 trigramString_2K.txt`
- `/u/csc368h/winter/pub/workloads/pbbsbench/testData/sequenceData/randomSeq -t int -r 1024 4096 randomSeq_1K_4K.txt`
- `/u/csc368h/winter/pub/workloads/pbbsbench/testData/geometryData/randPoints -d 2 4096 randPoints2D_4K.txt`
- `/u/csc368h/winter/pub/workloads/pbbsbench/testData/graphData/randLocalGraph -j -d 3 -m 16384 4096 randomAdjGraph_16K_4K.txt`
- `/u/csc368h/winter/pub/workloads/pbbsbench/testData/sequenceData/trigramString 4096 trigramString_4K.txt`
- `/u/csc368h/winter/pub/workloads/pbbsbench/testData/sequenceData/randomSeq -t int -r 2048 8192 randomSeq_2K_8K.txt`
- `/u/csc368h/winter/pub/workloads/pbbsbench/testData/geometryData/randPoints -d 2 8192 randPoints2D_8K.txt`
- `/u/csc368h/winter/pub/workloads/pbbsbench/testData/graphData/randLocalGraph -j -d 3 -m 32768 8192 randomAdjGraph_32K_8K.txt`
- `/u/csc368h/winter/pub/workloads/pbbsbench/testData/sequenceData/trigramString 8192 trigramString_8K.txt`
- `/u/csc368h/winter/pub/workloads/pbbsbench/testData/sequenceData/randomSeq -t int -r 4096 16384 randomSeq_4K_16K.txt`
- `/u/csc368h/winter/pub/workloads/pbbsbench/testData/geometryData/randPoints -d 2 16384 randPoints2D_16K.txt`
- `/u/csc368h/winter/pub/workloads/pbbsbench/testData/graphData/randLocalGraph -j -d 3 -m 65536 16384 randomAdjGraph_65K_16K.txt`
- `/u/csc368h/winter/pub/workloads/pbbsbench/testData/sequenceData/trigramString 16384 trigramString_16K.txt`

This script can be downloaded at: https://github.com/LatinScribe/CSC368-A1/blob/main/Pipelined%20processors/commands.txt

APPENDIX B: SYSTEM SCRIPTS

The following is the gem5 system configuration in python (inorder.py) with an in-order pipeline:

```
import m5
from m5.objects import *

import argparse

## Add the "binary" option to the script
DEFAULT_BINARY = '/u/csc368h/winter/pub/workloads/hello'

parser = argparse.ArgumentParser()
parser.add_argument('binary', type=str, default=DEFAULT_BINARY)
parser.add_argument('-a', '--binary_args')
parser.add_argument('-f', '--frequency', type=str, default='75MHz')

## Parse command-line arguments
args = parser.parse_args()
```

```
# System creation
system = System()

## gem5 needs to know the clock and voltage
system.clk_domain = SrcClockDomain()
system.clk_domain.clock = args.frequency
system.clk_domain.voltage_domain = VoltageDomain() # defaults to 1V

## Create a crossbar so that we can connect main memory and the CPU (below)
system.membus = SystemXBar()
system.system_port = system.membus.cpu_side_ports

## Use timing mode for memory modelling
system.mem_mode = 'timing'

# CPU Setup
system.cpu = X86MinorCPU()
system.cpu.icache_port = system.membus.cpu_side_ports
system.cpu.dcache_port = system.membus.cpu_side_ports

system.cpu.executeInputWidth = 1
system.cpu.executeIssueLimit = 1

## This is needed when we use x86 CPUs
system.cpu.createInterruptController()
system.cpu.interrupts[0].pio = system.membus.mem_side_ports
system.cpu.interrupts[0].int_requestor = system.membus.cpu_side_ports
system.cpu.interrupts[0].int_responder = system.membus.mem_side_ports

# Memory setup
system.mem_ctrl = MemCtrl()
system.mem_ctrl.port = system.membus.mem_side_ports

## A memory controller interfaces with main memory; create it here
system.mem_ctrl.dram = DDR3_1600_8x8()

## A DDR3_1600_8x8 has 8GB of memory, so setup an 8 GB address range
address_ranges = [AddrRange('8GB')]
system.mem_ranges = address_ranges
system.mem_ctrl.dram.range = address_ranges[0]

# Process setup
process = Process()

## Use a full path to the binary
binary = args.binary
process.cmd = [binary] + args.binary_args.split()

## The necessary gem5 calls to initialize the workload and its threads
system.workload = SEWorkload.init_compatible(binary)
system.cpu.workload = process
system.cpu.createThreads()

# Start the simulation
root = Root(full_system=False, system=system) # must assign a root

m5.instantiate() # must be called before m5.simulate
```

```
m5.simulate()
```

This script can be downloaded at: Python script for in-order processor: https://github.com/LatinScribe/CSC368-A1/blob/main/Pipelined%20processors/inorder.py

The following is the gem5 system configuration in python (out-of-order.py) with an out-of-order pipeline:

```
import m5
from m5.objects import *

import argparse

## Add the "binary" option to the script
DEFAULT_BINARY = '/u/csc368h/winter/pub/workloads/hello'

parser = argparse.ArgumentParser()
parser.add_argument('binary', type=str, default=DEFAULT_BINARY)
parser.add_argument('-a', '--binary_args')
parser.add_argument('-f', '--frequency', type=str, default='75MHz')

## Parse command-line arguments
args = parser.parse_args()

# System creation
system = System()

## gem5 needs to know the clock and voltage
system.clk_domain = SrcClockDomain()
system.clk_domain.clock = args.frequency
system.clk_domain.voltage_domain = VoltageDomain() # defaults to 1V

## Create a crossbar so that we can connect main memory and the CPU (below)
system.membus = SystemXBar()
system.system_port = system.membus.cpu_side_ports

## Use timing mode for memory modelling
system.mem_mode = 'timing'

# CPU Setup
system.cpu = X86O3CPU()
system.cpu.icache_port = system.membus.cpu_side_ports
system.cpu.dcache_port = system.membus.cpu_side_ports

#system.cpu.executeInputWidth = 1
# system.cpu.executeIssueLimit = 1

## This is needed when we use x86 CPUs
system.cpu.createInterruptController()
system.cpu.interrupts[0].pio = system.membus.mem_side_ports
system.cpu.interrupts[0].int_requestor = system.membus.cpu_side_ports
system.cpu.interrupts[0].int_responder = system.membus.mem_side_ports

# Memory setup
system.mem_ctrl = MemCtrl()
system.mem_ctrl.port = system.membus.mem_side_ports

## A memory controller interfaces with main memory; create it here
system.mem_ctrl.dram = DDR3_1600_8x8()
```

```
## A DDR3_1600_8x8 has 8GB of memory, so setup an 8 GB address range
address_ranges = [AddrRange('8GB')]
system.mem_ranges = address_ranges
system.mem_ctrl.dram.range = address_ranges[0]

# Process setup
process = Process()

## Use a full path to the binary
binary = args.binary
process.cmd = [binary] + args.binary_args.split()

## The necessary gem5 calls to initialize the workload and its threads
system.workload = SEWorkload.init_compatible(binary)
system.cpu.workload = process
system.cpu.createThreads()

# Start the simulation
root = Root(full_system=False, system=system) # must assign a root

m5.instantiate() # must be called before m5.simulate
m5.simulate()
```

APPENDIX C: BENCHMARK SCRIPTS

The following are the commands to run all the benchmarks on all the input sizes (should be ran in the same directory that the input files were created):

- /u/csc368h/winter/pub/bin/gem5.opt system.py /u/csc368h/winter/pub/workloads/pbbsbench/benchmarks/comparisonSort/sampleSort/sort --binary_args "-o output.txt -r 1 randomSeq_512_2K.txt"
- /u/csc368h/winter/pub/bin/gem5.opt system.py /u/csc368h/winter/pub/workloads/pbbsbench/benchmarks/comparisonSort/serialSort/sort --binary_args "-o output.txt -r 1 randomSeq_512_2K.txt"
- /u/csc368h/winter/pub/bin/gem5.opt system.py /u/csc368h/winter/pub/workloads/pbbsbench/benchmarks/delaunayTriangulation/incrementalDelaunay/delaunay --binary_args "-o output.txt -r 1 randPoints2D_2K.txt"
- /u/csc368h/winter/pub/bin/gem5.opt system.py /u/csc368h/winter/pub/workloads/pbbsbench/benchmarks/breadthFirstSearch/serialBFS/BFS --binary_args "-o output.txt -r 1 randomAdjGraph_8K_2K.txt"
- /u/csc368h/winter/pub/bin/gem5.opt system.py /u/csc368h/winter/pub/workloads/pbbsbench/benchmarks/wordCounts/serial/wc --binary_args "-o output.txt -r 1 trigramString_2K.txt"
- /u/csc368h/winter/pub/bin/gem5.opt system.py /u/csc368h/winter/pub/workloads/pbbsbench/benchmarks/comparisonSort/sampleSort/sort --binary_args "-o output.txt -r 1 randomSeq_1K_4K.txt"
- /u/csc368h/winter/pub/bin/gem5.opt system.py /u/csc368h/winter/pub/workloads/pbbsbench/benchmarks/comparisonSort/serialSort/sort --binary_args "-o output.txt -r 1 randomSeq_1K_4K.txt"
- /u/csc368h/winter/pub/bin/gem5.opt system.py /u/csc368h/winter/pub/workloads/pbbsbench/benchmarks/delaunayTriangulation/incrementalDelaunay/delaunay --binary_args "-o output.txt -r 1 randPoints2D_4K.txt"
- /u/csc368h/winter/pub/bin/gem5.opt system.py /u/csc368h/winter/pub/workloads/pbbsbench/benchmarks/breadthFirstSearch/serialBFS/BFS --binary_args "-o output.txt -r 1 randomAdjGraph_16K_4K.txt"

- /u/csc368h/winter/pub/bin/gem5.opt system.py /u/csc368h/winter/pub/workloads/pbbsbench/benchmarks/wordCounts/serial/wc --binary_args "-o output.txt -r 1 trigramString_4K.txt"
- /u/csc368h/winter/pub/bin/gem5.opt system.py /u/csc368h/winter/pub/workloads/pbbsbench/benchmarks/comparisonSort/sampleSort/sort --binary_args "-o output.txt -r 1 randomSeq_2K_8K.txt"
- /u/csc368h/winter/pub/bin/gem5.opt system.py /u/csc368h/winter/pub/workloads/pbbsbench/benchmarks/comparisonSort/serialSort/sort --binary_args "-o output.txt -r 1 randomSeq_2K_8K.txt"
- /u/csc368h/winter/pub/bin/gem5.opt system.py /u/csc368h/winter/pub/workloads/pbbsbench/benchmarks/delaunayTriangulation/incrementalDelaunay/delaunay --binary_args "-o output.txt -r 1 randPoints2D_8K.txt"
- /u/csc368h/winter/pub/bin/gem5.opt system.py /u/csc368h/winter/pub/workloads/pbbsbench/benchmarks/breadthFirstSearch/serialBFS/BFS --binary_args "-o output.txt -r 1 randomAdjGraph_32K_8K.txt"
- /u/csc368h/winter/pub/bin/gem5.opt system.py /u/csc368h/winter/pub/workloads/pbbsbench/benchmarks/wordCounts/serial/wc --binary_args "-o output.txt -r 1 trigramString_8K.txt"
- /u/csc368h/winter/pub/bin/gem5.opt system.py /u/csc368h/winter/pub/workloads/pbbsbench/benchmarks/comparisonSort/sampleSort/sort --binary_args "-o output.txt -r 1 randomSeq_4K_16K.txt"
- /u/csc368h/winter/pub/bin/gem5.opt system.py /u/csc368h/winter/pub/workloads/pbbsbench/benchmarks/comparisonSort/serialSort/sort --binary_args "-o output.txt -r 1 randomSeq_4K_16K.txt"
- /u/csc368h/winter/pub/bin/gem5.opt system.py /u/csc368h/winter/pub/workloads/pbbsbench/benchmarks/delaunayTriangulation/incrementalDelaunay/delaunay --binary_args "-o output.txt -r 1 randPoints2D_16K.txt"
- /u/csc368h/winter/pub/bin/gem5.opt system.py /u/csc368h/winter/pub/workloads/pbbsbench/benchmarks/breadthFirstSearch/serialBFS/BFS --binary_args "-o output.txt -r 1 randomAdjGraph_65K_16K.txt"
- /u/csc368h/winter/pub/bin/gem5.opt system.py /u/csc368h/winter/pub/workloads/pbbsbench/benchmarks/wordCounts/serial/wc --binary_args "-o output.txt -r 1 trigramString_16K.txt"

This script can be downloaded at: https://github.com/LatinScribe/CSC368-A1/blob/main/Pipelined%20processors/commands.txt

APPENDIX D: CHARTS FOR EXPLORATION 2

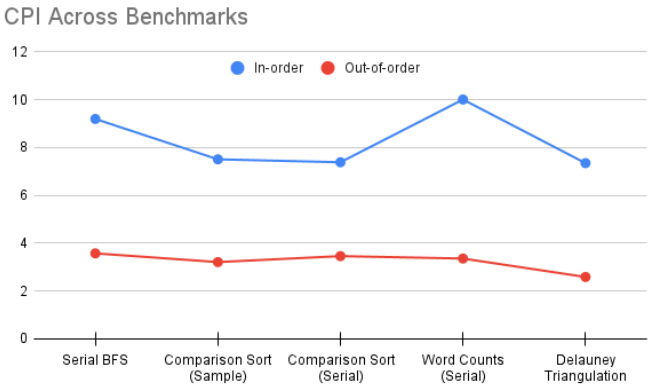


Fig. 4. CPI Across Benchmarks.

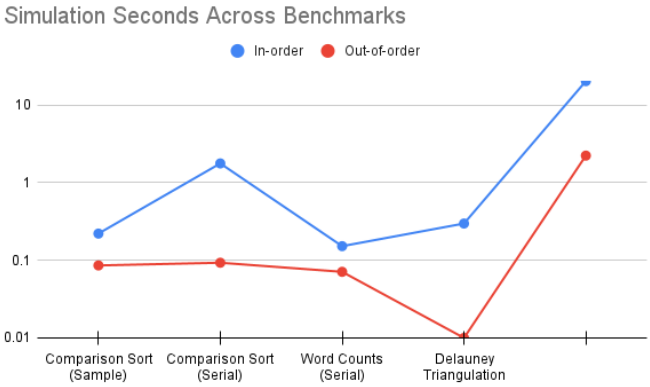


Fig. 5. Simulation Seconds Across Benchmarks.

APPENDIX E: EXPLORATION 3 RESULT

TABLE II
CACHE STATISTICS ON DIFF. ORGANIZATION

Benchmark	Assoc.	Sim. Time (s)	Total Hits	Total Misses	Miss Rate (%)
serialBFS	**1**	1.360939	1,261,940	506,447	28.64%
	Full	1.361544	1,206,741	423,297	25.97%
sample sort	**1**	2.095822	1,531,452	43,741	2.78%
	Full	2.065584	1,533,412	31,318	2.00%
serialsort	**1**	2.088718	1,130,282	30,128	2.60%
	Full	2.065095	1,126,306	26,700	2.32%
delaunay	**1**	2.536819	38,723,414	12,147,927	23.88%
	Full	2.481602	40,189,083	8,624,443	17.67%
wordcount serial	**1**	1.031422	386,181	48,553	23.88%
	Full	1.023477	411,526	13,742	3.23%

[a]Assoc. refers to direct mapped cache(1) or fully associative cache(full).

TABLE III
CACHE STATISTICS FOR DIFFERENT L1/L2 CONFIGURATIONS

Benchmark	Size	Sim. Time (s)	Total Hits	Total Misses	Miss Rate (%)
serial BFS	**default**	0.030151	1,222,193	447,773	26.81%
	L1*2	0.029596	1,213,448	427,680	26.06%
	L2*2	0.028924	1,246,056	414,622	24.97%
sample sort	**default**	0.027484	1,534,089	34,474	2.20%
	L1*2	0.027313	1,533,907	29,065	1.86%
	L2*2	0.027440	1,534,909	33,374	2.13%
serial sort	**default**	0.026831	1,126,134	26,096	2.26%
	L1*2	0.026704	1,126,786	21,521	1.87%
	L2*2	0.026565	1,127,069	22,899	1.99%
delaunay	**default**	0.888724	39,915,084	9,714,796	19.57%
	L1*2	0.840916	40,524,675	7,946,341	16.39%
	L2*2	0.855410	40,254,536	9,184,507	18.58%
wordcount serial	**default**	0.007252	397,215	32,967	7.66%
	L1*2	0.006338	401,212	18,739	4.46%
	L2*2	0.006981	41,052,963	9,236,213	18.37%

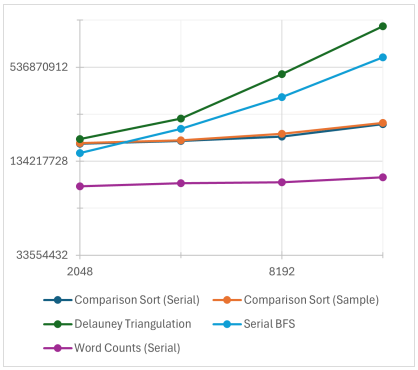
[a]"default" uses the baseline cache size (as in Tutorial 3).
[b]"L1" doubles the L1 caches; "L2" doubles the L2 cache.

Listing 1. Default Cache Configuration in gem5 Python Script

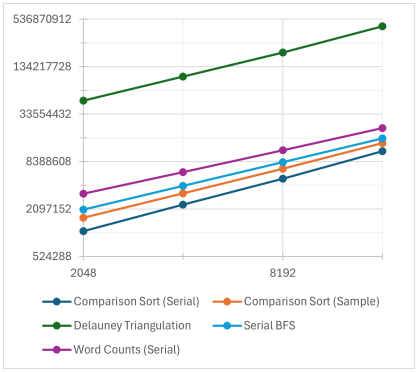
```
class L1ICache(Cache):
    assoc = 2
    tag_latency = 1
    data_latency = 1
    response_latency = 1
    mshrs = 8
    tgts_per_mshr = 20
    size = '4kB'

class L1DCache(Cache):
    assoc = 2
    tag_latency = 1
    data_latency = 1
    response_latency = 1
    mshrs = 8
    tgts_per_mshr = 20
    size = '4kB'

class L2Cache(Cache):
    assoc = 8
    tag_latency = 8
    data_latency = 8
    response_latency = 1
    mshrs = 8
    tgts_per_mshr = 20
    size = '32kB'
```

Appendix F: Charts for Exploration 1

(a) Before



(b) Region of Interest



(c) After

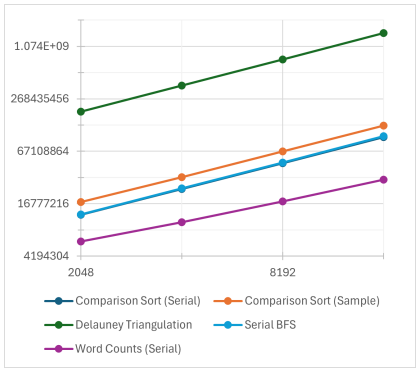
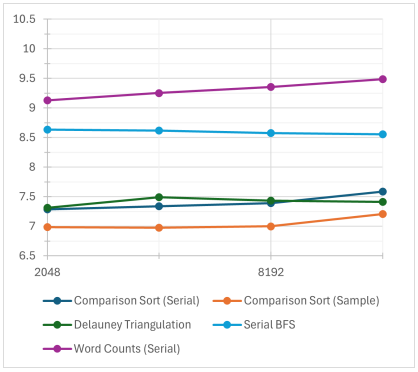
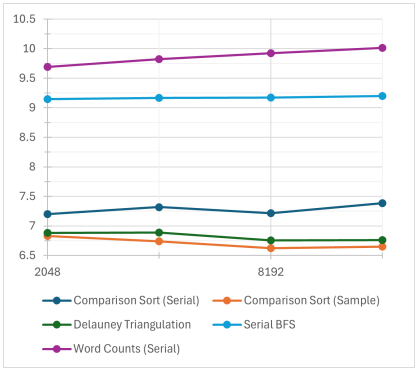


Fig. 6. Number of cycles vs. input size

(a) Before



(b) Region of Interest



(c) After

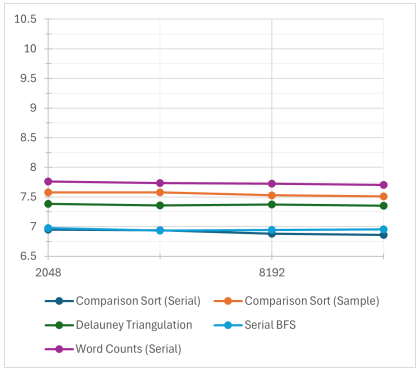


Fig. 7. CPI vs. input size