Characterizing PBBS on single-core systems

Duain Chhabra Frederick Meneses Theodore Preduta Yoonie Jang

I. Introduction

Single-core systems provide a clear and manageable framework for understanding the performance characteristics of different workloads. In particular, using the gem5 simulator enables us to observe how key performance metrics change as CPU configuration parameters are varied, thereby offering valuable insights that can inform future hardware optimizations. These experiments not only reveal which workloads are best suited to specific hardware configurations but also offer a mechanistic understanding of the instructions driving performance improvements.

This study is structured into three main explorations. First, we examine how varying input sizes affects performance on a simple atomic timing model CPU by presenting key metrics and analyzing the instruction mix. Next, we compare the performance differences between in-order and out-of-order execution pipelines across a selected subset of workloads. Finally, we assess how different cache configurations impact system performance, highlighting improvements or degradations as the cache hierarchy is altered.

The Problem-Based Benchmark Suite (PBBS) is a publicly available set of benchmarks designed to test algorithms used to solve common problems [1]. The PBBS also provides scripts for generating input data, making this suite a convenient choice for running benchmarks on simulated computer systems. We chose a subset of the PBBS benchmarks to use for our investigation:

1) **SORT (comparisonSort):** Sorting is a fundamental operation that evaluates how a CPU pipeline handles numerical data processing.
2) **HIST (histogram):** This workload involves intensive memory interaction, making it ideal for assessing the impact of out-of-order execution and cache performance.
3) **BFS (breadthFirstSearch):** Graph traversal algorithms like BFS feature numerous control dependencies, which are critical for testing the CPU's control logic and branch prediction.
4) **MSF (minSpanningForest):** The minimum spanning forest problem stresses both computational and memory subsystems, allowing for an evaluation of the balance between processing speed and memory access efficiency.
5) **LRS (longestRepeatedSubstring):** String processing tasks, such as finding the longest repeated substring, are pivotal in many applications and test the processor's ability to manage complex pattern recognition and data manipulation.

II. Exploration 1

In this first exploration, we investigate how the benchmarks' performance and behavior change with varying input sizes, focusing particularly on the mix of operations. Our analysis confirms that the selected benchmarks exhibit a diverse range of operations, underscoring the importance of identifying a region of interest for detailed evaluation.

A. Experimental Setup

The experimental setup for this exploration comprises of two key components: the gem5 simulation configuration and the input data. In this section, we explain the rationale behind the configuration of both components.

Table I summarizes the input sizes used in our experiments. The largest input sizes were determined as the maximum values that allow simulations to complete in under 10 minutes. The smallest inputs were set to one-tenth of the largest, and intermediate inputs were generated such that their sizes increase incrementally by the smallest input size. For data generation, PBBS provides scripts that directly specify the desired size for graphs (used by BFS and MSF) and lists of numbers (used by HIST and SORT) [1]. For LRS, data was generated by taking prefixes of the required length from the provided DNA file.

TABLE I
Exploration 1 Input Sizes

Benchmark	Size Unit	Smallest Input	Largest Input
BFS	Vertices	10^2	10^3
HIST	List Length	10^4	10^5
LRS	Characters	10^4	10^5
MSF	Vertices	10^2	10^3
SORT	List Length	10^4	10^5

As for the gem5 simulation, the full gem5 script can be found in appendix A, the highlight of which, is the choice of CPU. We initially tested the results under two configurations, one making use of X86AtomicSimpleCPU and a more complex configuration making use of X86MinorCPU and several caches. Since we are varying the input size we had initially thought we might be able to observe cache effects on the X86MinorCPU configuration as input sizes grew, however we found that given our choice of inputs, no such effect could be meaningfully observed. In addition to the lack of extra insight the X86MinorCPU script provided, we were made aware of a bug in gem5 which cast doubts on the legitimacy of some of our results using that configuration [5]. So, for the purposes of this exploration we present the results generated using the X86AtomicSimpleCPU configuration.

B. Results and discussion

Because of our choice of simulated CPU, having no cache or pipeline, the main metrics of interest are the number of

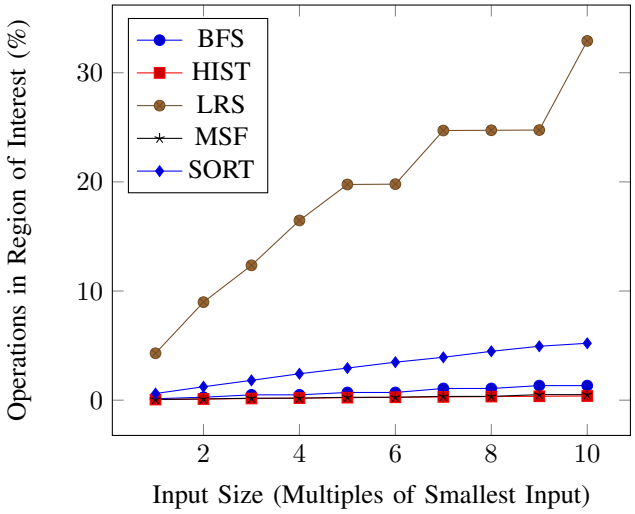

Fig. 1. Percentage of operations spent in the region of interest relative to the total number of operations.

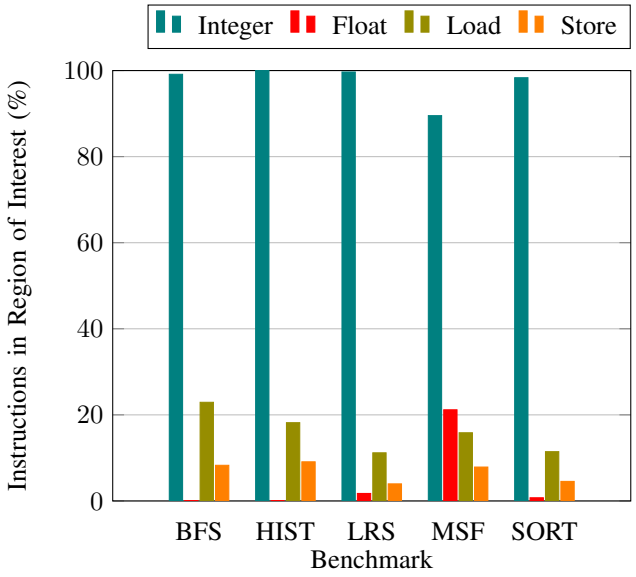

Fig. 2. Number of operations in the region of interest of each type, as a percentage of the total number of operations in the region of interest, using the largest input size from table I. Note that an instruction is classified as a certain type if it uses its respective component (e.g. ALU for Integer), hence why the instruction types do not add to 100%.

operations and the mix of operations. We now discuss each of these metrics and their implications.

First we consider the number of operations. As is expected, the total number of operations grows proportionally to the input size. In all cases we observe that the number of operations after the region of interest represent less than 0.5% of the total operations, so we will focus the rest of this discussion on the number of operations before and during the region of interest. As seen in figure 1, across all our tests a significant majority of operations are dedicated to before the region of interest. So if our goal is to measure how a certain algorithm behaves on a system, measuring total statistics for the entire benchmark is, at these input sizes, effectively measuring how input parsing behaves on a system. Hence, this is why we have a region of interest. In addition to this fact, in figure 1, as the input sizes increase, we can almost make out time complexity classes. The fact HIST and MSF are constant indicate that they grow directly proportionally to the size of the input, and so are in fact linear. Similarly we expect BFS, SORT, and LRS to be superlinear, which is what we see.

Moving on to the instruction mix, we found that across our input sizes, the relative instruction mix stayed constant. As we see in figure 2, we indeed find that the regions of interest for our choice of benchmarks is quite varied. In addition to this, we found the instruction mix before and after the region of interest, in the vast majority of cases, is within margin of error of the instruction mix of HIST in figure 2. The one exception to this is the instruction mix before the region of interest for MSF, which contains 18% floating point operations but is otherwise the same.

III. Exploration 2

Pipelined processors exploit instruction-level parallelism to improve the throughput of a processor. In-order pipelines are limited by how instructions must pass through them in program order, and therefore any stalls will affect all subsequent instructions. This does not occur in out-of-order pipelines, as instructions do not need to wait for preceding instructions to complete before being executed. This exploration aims to investigate the performance of in-order pipelined processors compared to out-of-order pipelined processors. We find that an out-of-order processor is associated with significant performance improvements, as expected.

A. Experimental setup

Unlike Exploration 1, this exploration did not vary the input data sets. Instead, only the configurations for the gem5 simulations were modified. The chosen PBBS benchmarks were run on four different processor configurations (Table II).

TABLE II
CPU configurations for Exploration 2

System	CPU	Execute width
1	MinorCPU (In-order)	1
2	MinorCPU (In-order)	2
3	MinorCPU (In-order)	4
4	O3CPU (Out-of-order)	N/A

The MinorCPU and O3CPU processors were chosen so that both in-order and out-of-order pipelines could be tested. The execute width parameter, which represents the number of instructions that may be executed at once, was varied to investigate whether modifying the width would have a measurable impact on performance. This value was set manually for systems 1 and 3, while system 2 used the default value of 2 for the MinorCPU class in gem5. Execute width is not a configurable parameter for the O3CPU and is thus not shown in the table.

The gem5 scripts used to simulate these systems can be found in appendix B. Note that while the CPU models were

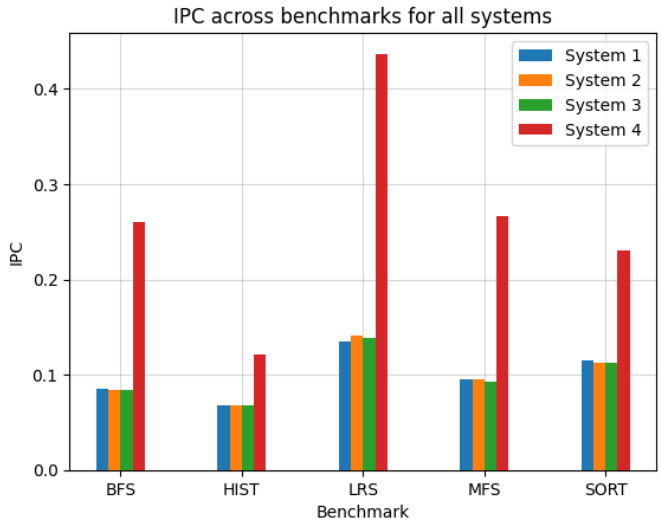

Fig. 3. IPC counts in the region of interest for all benchmarks, for each of the four systems.

varied between experiments, other parameters remained constant for all four configurations.

The inputs for the benchmarks were generated using the generation scripts provided with the benchmark suite. The input sizes used were 10,000 vertices (for the graph-based benchmarks, BFS and MSF) and 100,000 items for the remaining benchmarks (LRS, HIST, and SORT). These relatively large input sizes were chosen so that differences in metrics could be better identified.

B. Results and discussion

To determine the impact of pipeline order on performance, we examine the metrics that measure the areas where out-of-order pipelining will have the largest effect. Thus throughput (IPC) and the number of cycles the processor spent stalled (number of NOP instructions) were chosen as the metrics of interest. However, due to the bug in X86MinorCPU which made it impossible to get accurate instruction counts [5], the number of NOP instructions could unfortunately not be used as a metric.

As expected, IPC rises dramatically for system 4 (Figure 3), which uses the out-of-order pipeline. Avoiding stalls by executing instructions out-of-order allows the throughput to increase. This result highlights the effectiveness of an out-of-order pipeline for reducing the number of stalls. However, within the systems configured to use the in-order MinorCPU, increasing the execute width does not appear to result in significant performance improvements. This may be because it is difficult to achieve significant amounts of instruction-level parallelism without carefully configuring the rest of the processor. Therefore, increased execution widths may not necessarily corresponding to improved parallelism and performance [2].

The performance increases introduced by the out-of-order pipelines differ for each benchmark. Performance gains of a particular benchmark are influenced by their implementation and the amount of dependencies present in the algorithms; code with less dependencies is less likely to encounter hazards and be forced to stall in the in-order pipeline. In our observed IPC results, for instance, the HIST and SORT benchmarks experience the least performance increase from an in-order to out-of-order pipeline (roughly 78% and 103%, respectively, while the remaining benchmarks experienced performance increases closer to 200%). This may be because there are not as many dependencies in the HIST benchmark, unless multiple consecutive operations attempt to update the same bucket, which would have been unlikely given our input. Similarly, while the SORT benchmark may have many branching operations, consecutive instructions are unlikely to attempt to modify the same resource. These benchmarks likely experience less stalling in the in-order pipeline, so there may be less performance increases to be gained by using an out-of-order pipeline instead.

IV. Exploration 3

In this exploration we analyze how different cache parameters affect performance, focusing on cache size, input size and associativity. Specifically, we analyze the effect of sweeping the cache size and cache associativity on performance metrics such as L1D Cache Miss Rate and IPC (Instructions Per Cycle).

A. Experimental setup

The experimental setup for this exploration follows a similar methodology to Exploration 2, where we analyzed the effects of different CPU models (O3CPU and X86MinorCPU). However, in this case, our focus is on evaluating how varying L1 data cache sizes and different levels of associativity affect overall performance. We analyze the impact of cache size and associativity on performance metrics such as L1D cache miss rate and Instructions Per Cycle (IPC). To achieve this, we tested multiple cache configurations under different workloads to observe how changes in cache hierarchy influence execution behavior.

TABLE III
Cache Configurations for Exploration 3

System	L1D Cache Size	L1I Cache Size	Associativity
1	4KB	4KB	2-way
2	8KB	8KB	2-way
3	16KB	16KB	2-way
4	32KB	32KB	2-way
5	64KB	64KB	2-way
6	4KB	4KB	1-way
7	4KB	4KB	2-way
8	4KB	4KB	4-way
9	4KB	4KB	8-way

The MinorCPU and O3CPU were reused from exploration 2 and connected to cache configurations defined in Table III, with no L2 cache. The full script can be found in appendix 3.

B. Results and Discussion

Our experiments reveal significant variations in workload sensitivity to cache parameters and processor architectures,

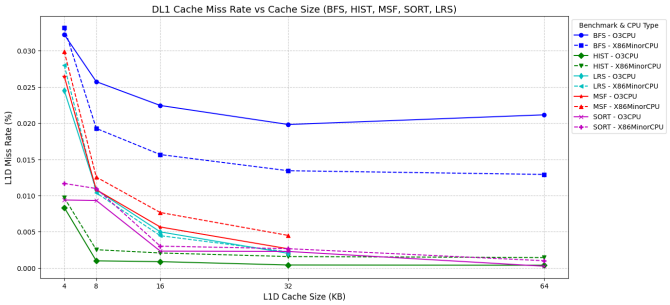

Fig. 4. DL1 Cache Miss Rates vs Cache Size for all cache configurations

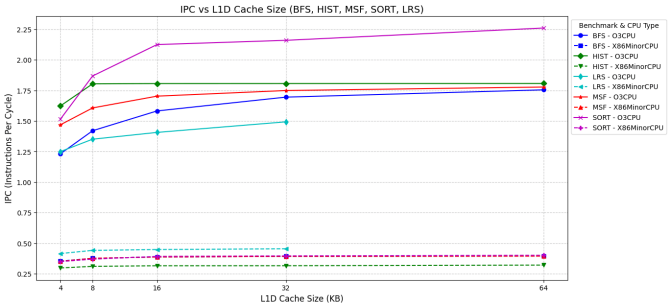

Fig. 5. IPC counts in the region of interest for all benchmarks, for all cache configurations

offering insights into cache hierarchy optimization. Below, we analyze these trends in the context of the posed research questions.

1) Workload Sensitivity to Cache Parameters: **Cache Size Sensitivity:** MSF and LRS showed the highest sensitivity to L1D size, with miss rates dropping 92% and 94% respectively when increasing from 4 KB to 32 KB (Fig. 4). MSF's irregular graph traversals and LRS's large string buffers explain this trend. Pipeline reordering (in-order vs. out-of-order) had minimal impact, indicating memory access patterns dominate over scheduling. In general, IPC is positively affected when increasing L1D cache sizes eventually leading to diminishing returns.

Associativity Sensitivity: HIST demonstrated the strongest sensitivity, with a 22% IPC increase in direct-mapped caches versus 2-way associative designs. HIST involves frequent updates to numerous small, non-sequential memory addresses (histogram bins). These bins often map to the same cache sets

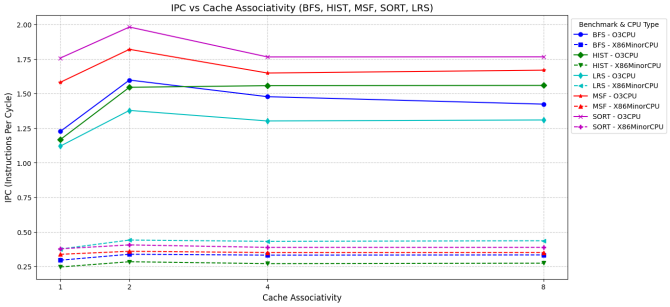

Fig. 6. IPC vs Associativity for all benchmarks, for all cache configurations

in low-associativity designs, causing thrashing. BFS showed moderate sensitivity (11% variation), while MSF and SORT saw negligible changes (1-2%) from sequential access patterns. In general, IPC is positively affected when increasing associativity up to a 2-way associative cache.

2) Processor Sensitivity to Cache Size: O3CPU exhibited greater IPC sensitivity to cache size than MinorCPU. Quadrupling L1D size improved O3CPU's IPC by up to 12.4% in SORT versus MinorCPU's 5-6% gain, attributed to O3CPU's ability to exploit memory-level parallelism (MLP) with cached data. O3CPU achieved lower miss rates in LRS, SORT, and HIST, while MinorCPU outperformed in BFS and MSF.

3) Implications of Input Scaling: Testing with small inputs at reduced cache sizes (4–64 KB) revealed trends despite simulation constraints. Real-world deployments require larger caches, yet MSF and LRS's sensitivity highlights persistent optimization challenges for edge devices with strict cache limits.

4) Summary of Key Findings:

1) **Most Sensitive Workloads:** - *Cache Size:* MSF (92% miss rate reduction) and LRS (94%) - *Associativity:* HIST (22% IPC variation)
2) **Most Sensitive Processor:** O3CPU, as its MLP potential is heavily cache-dependent.

These results emphasize workload-aware tuning: HIST benefits more from associativity, while MSF/LRS require larger caches. Future work should explore prefetchers and cache partitioning for edge optimization.

V. Conclusion

By running our set of benchmarks (BFS, HIST, LRS, MSF, and SORT) on a variety of differently configured simulated processors, we observed the effects of parameters such as input size, pipeline order and width, and cache configuration on performance. We confirm that out-of-order pipelines exhibit better performance than in-order pipelines, though varying the execution width of an in-order pipeline does not result in performance gains. We also observe that increasing cache sizes leads to better performance. Across both explorations, we find that different workloads react differently to variations in configuration parameters.

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.
[5] "gem5 MinorCPU bug(s)." Piazza. https://piazza.com/class/m4vp7hf1id4dm/post/82

APPENDIX A: SCRIPT 1

```
import m5
from m5.objects import *

import argparse

# Add "common" gem5 scripts to the path
m5.util.addToPath("/u/csc368h/winter/pub/scripts/gem5/")

# Set up and parse arguments
parser = argparse.ArgumentParser()
parser.add_argument('binary', type=str)
parser.add_argument('-a', '--binary_args')
args = parser.parse_args()

# System creation
system = System()

## gem5 needs to know the clock and voltage
system.clk_domain = SrcClockDomain()
system.clk_domain.clock = '200MHz'
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 atomic mode for memory modelling
system.mem_mode = 'atomic'

# CPU Setup
system.cpu = X86AtomicSimpleCPU()
system.cpu.icache_port = system.membus.cpu_side_ports
system.cpu.dcache_port = system.membus.cpu_side_ports

## 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]

## 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

```

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

APPENDIX B: SCRIPT 2

Below is a sample script used to simulate system 1 as described in section III A. The scripts used to simulate the other systems were different only in the lines indicated by comments.

```
import m5
from m5.objects import *
import argparse

# Add "common" gem5 scripts to the path
m5.util.addToPath("/u/csc368h/winter/pub/scripts/gem5/")

# import the common/SimpleOpts module
from common import SimpleOpts

parser = argparse.ArgumentParser()
parser.add_argument('binary', type=str)

# speicific argument for input
parser.add_argument('-a', '--binary_args')

## 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 = '200MHz'
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

system.mem_mode = 'timing'

# CPU Setup
system.cpu = X86MinorCPU()      # for system 4, this line was modified to use X86O3CPU()
    instead
system.cpu.icache_port = system.membus.cpu_side_ports
system.cpu.dcache_port = system.membus.cpu_side_ports

# setting width and instruction issue limit to 1, these lines were changed for system 3 and
    removed for systemd 2 and 4
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()

```

```
64 ## Use a full path to the binary
65 binary = args.binary
66 process.cmd = [binary, args.binary_args]
67
68 ## The necessary gem5 calls to initialize the workload and its threads
69 system.workload = SEWorkload.init_compatible(binary)
70 system.cpu.workload = process
71 system.cpu.createThreads()
72
73 # Start the simulation
74 root = Root(full_system=False, system=system) # must assign a root
75
76 m5.instantiate() # must be called before m5.simulate
77 m5.simulate()
```

APPENDIX C: SCRIPT 3

```
import m5
from m5.objects import *

import argparse

## Add the "binary" option to the script
DEFAULT_BINARY = '/u/csc368h/winter/pub/workloads/pbbsbench/benchmarks/breadthFirstSearch/
    backForwardBFS/BFS'

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')

# Add new arguments for CPU_CONFIG, CACHE_SWEEP, and ASSOCIATIVITY
parser.add_argument('--cpu_config', type=str, default='X86MinorCPU')
parser.add_argument('--cache_sweep', type=str, default='32kB')
parser.add_argument('--associativity', type=int, default=2)

## 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
cpu_class = globals()[args.cpu_config]
system.cpu = cpu_class()

## 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

# Cache setup
class L1ICache(Cache):
    assoc = args.associativity
    tag_latency = 1
    data_latency = 1
    response_latency = 1
    mshrs = 8
    tgts_per_mshr = 20
    size = args.cache_sweep

class L1DCache(Cache):
    assoc = args.associativity
    tag_latency = 1
    data_latency = 1
    response_latency = 1
    mshrs = 8
    tgts_per_mshr = 20
    size = args.cache_sweep

system.cpu.l1d = L1DCache()
```

```
system.cpu.l1d.mem_side = system.membus.cpu_side_ports
system.cpu.l1d.cpu_side = system.cpu.dcache_port

system.cpu.l1i = L1ICache()
system.cpu.l1i.mem_side = system.membus.cpu_side_ports
system.cpu.l1i.cpu_side = system.cpu.icache_port

# 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]

## 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()
```