Characterizing PBBS on single-core systems

Leyang Cai Edward Han Zekun Liu

I. Introduction

The Problem-Based Benchmark Suite (PBBS) provides diverse workloads, allowing us to analyze performance across different computational patterns [1]. This diversity allows us to select a variety of benchmarks, from memory vs compute intensive, regular vs irregular accesses, and integer vs floating point arithmetic. Following this, we selected the following five workloads: Serial radix integer sort, to demonstrate regular memory access pattern and memory-bound behavior, serial BFS, for irregular memory accesses, serial div sufsort suffix array, for integer heavy manipulations, parallel ck nBody, for floating point heavy computations and a compute bound benchmark, and oct tree KNN, which demonstrates vector arithmetic. These benchmarks allow us to explore the relationship between input size, processor pipelines, and cache configurations, which aligns with core computer architecture concepts, such as instruction-level parallelism (ILP), pipeline efficiency, and memory hierarchy optimization.

II. Exploration 1

In this exploration, we investigate how key metrics change across different input sizes for each phase of execution: setup, region of interest, and tear down. We will mainly focus on the region of interest, but will also investigate the significance of the other phases, namely the reason for reporting stats on from the region of interest. Before running these experiments, we hypothesized the following: As input size grows, the total number of instructions will increase during the region of interest, obviously. Additionally, we hypothesized that during the setup and tear down phase, it would also increase, but not as dramatically as during the region of interest.

A. Experimental setup

To investigate the effect of input sizes, we generate smaller input sizes using PBBS scripts. For serial radix integer sort, we used integer arrays of length 1k, 2k, 4k, 8k, 16k, 32k, 64k, and 128k. We found that these array lengths allowed us to capture a wide range of lengths while also keeping the gem5 runtime reasonable. For serial BFS, we generated graph inputs with n vertices and 20*n edges, with graph sizes varying from 100 vertices + 2000 edges, to 6.4k vertices + 128k. For serial div suffix array, we generated a varying number of strings, from 1k to 64k. For parallel CK nBody, we found that it ran especially slow, so we chose to investigate the benchmark with a list of 3d vectors ranging from 100 to 3200. All benchmarks ran on the same simulated system, meaning that all performances changes can be attributed to input size rather than hardware differences. Finally, we generated 1k to 128k 3d vectors as inputs for KNN. We used the X86MinorCPU at 75MHz in timing mode. The system has 8GB of DDR3_1600_8x8 DRAM, and a standard memory controller.

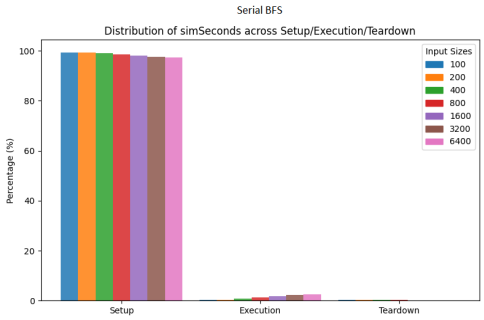

Fig. 1. Serial BFS - simSeconds across stages of execution, as percentages

The key metrics we track include: simInst and system.cpu.numCycles, which indicate the amount of work done by the system, simSeconds, which shows simulated execution time, system.cpu.cpi, which gives insight on how efficiently input scales (or doesn't), system.mem_ctrl.readReqs and system.mem_ctrl.writeReq, which shows how memory traffic scales with input sizing.

We also compare how the percentage of each execution region scale with input sizing, as a percentage.

B. Results and discussion

Firstly, we observe that the distribution of simSeconds across the three stages of execution during bfs is extremely skewed towards the setup phase, greater than 95%, seen in figure 1. This occurs not only on this metric, but other key metrics including read and write requests, and on other benchmarks, like integer sort. If we were to only report the entire execution's statistics, the "real" data from the region of interest would be completely drowned out by the large setup cost. By isolating the region of interest, we avoid misinterpretation of the data caused by aggregating statistics. Thus, for the rest of this exploration, we only report results from the region of interest. Unsurprisingly, we found that increasing the input size increased the simulation instructions, number of total cycles, simulation time, wall clock time, and both read and write requests in the region of interest. Figure 2 shows an example of simulated instructions in relation to input size. We notice that this figure appears to scale at an approximate O(nlogn), which is expected of a sorting algorithm. Note that not all benchmarks exhibit this behaviour, as their behaviours differ, but they certainly increase in relation to the input size.

Interestingly, we found no real correlation between input size and memory bandwidth, along with the CPI for the integer sort benchmark, despite write and read requests scaling with input size. For this, we turn to our BFS benchmark,

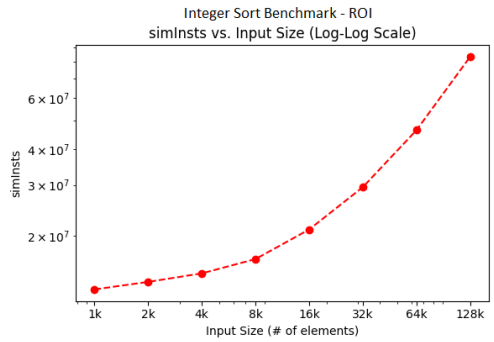

Fig. 2. Integer Sort Benchmark, simInst vs Input Sizes in ROI

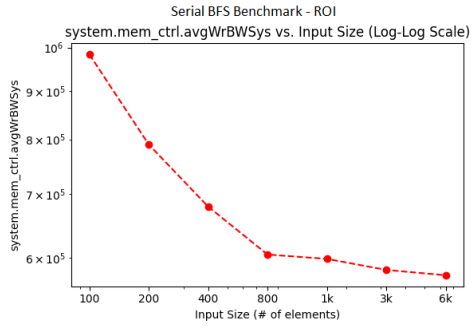

Fig. 3. Serial BFS Benchmark, Write Bandwidth vs Input Sizes in ROI

which incurs a significant penalty due to random memory accesses. We see that read bandwidth generally trends upward. This is expected, as we expect there to a greater number of memory access as the graph grows. We also note how the write bandwidth degrades with input size. This trend can be attributed to the fact that BFS writes ”visited” flags, and with more flags to write, the DRAM can become overloaded, causing write bandwidth to degrade, as seen in Figure 3.

If we investigate CPI, we notice that suffix array and nbody exhibit opposite behaviours - CPI decreases for larger inputs when running nbody, and increases for suffix array, as seen in Figure 4. This matches with our expectations, since suffix array is memory-bound and nbody is compute-bound. As more memory accesses are required, the system is forced to wait on DRAM, stalling the CPU. For larger nbody inputs, the CPU is

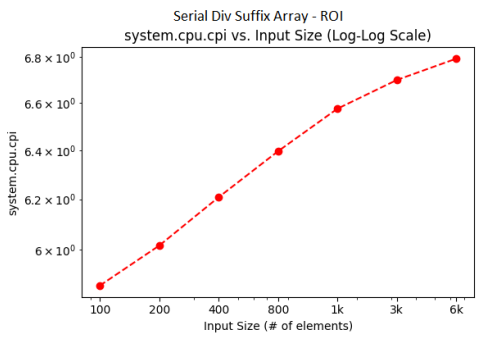

Fig. 4. Serial Div Suffix Array, CPI vs Input Sizes in ROI

busy executing floating-point operations, which amortizes the cost of waiting on DRAM, forcing CPI down.

Unfortunately, due to our reliance on MinorCPU, we were unable to faithfully capture the instruction mix due to bugs with gem5. We have opted not to present that portion of the data due to misleading conclusions.

III. Exploration 2

In this exploration, we will investigate the impact of different CPU architectures on PBBS benchmarks. We will compare in-order and out-of-order execution models using the MinorCPU (in-order) and O3CPU (out-of-order) implementations in the gem5 simulator. We can understand how architectural differences influence performance by analyzing key metrics across a subset of these benchmarks.

A. Experimental setup

For this experiment, we use the same system configuration as in the previous exploration, with the only changes being the CPU type: we test both the X86MinorCPU (in-order execution) and the X86O3CPU (out-of-order execution). All other system parameters remain constant, so that observed performance differences can be attributed only to CPU architecture. The system is configured as follows: 75MHz in timing mode, 8GB DDR3_1600_8x8 DRAM with a standard gem5 memory controller.

We evaluate the following benchmarks with their respective input sizes: For serial BFS, an adjacency graph with 300 nodes generated using randLocalGraph. For oct_tree KNN, A sequence of 100 two-dimensional points generated using randPoints. For radix sort, a sequence of 100 integers generated using randomSeq.

The key metrics tracked include: system.cpu.cpi and system.cpu.ipc, as they directly measure execution efficiency. We chose this because out-of-order pipelines tend to achieve higher instructions per cycle (IPC) by dynamically reordering instructions to reduce pipeline stalls, while in-order pipelines have to process instructions sequentially, resulting in lower IPC. Furthermore, system.cpu.branchPred.TakenMispredicted is important to measure how effectively each pipeline mitigates control-flow hazards. Branch mispredictions cause costly pipeline flushes, and out-of-order architectures often reduce these penalties through more advanced branch prediction and speculative execution.

However, due to an issue in MinorCPU regarding branch collection information, we were unable to collect branch data. Despite this limitation, we focus on the available metrics to understand the differences in execution behavior between in-order and out-of-order architectures.

In addition to the two CPU configurations, we also measured the performance difference across the X86MinorCPU with varying system.cpu.executeIssueLimit values.

B. Results and discussion

The differences in performance metrics, most notably CPI and IPC, between the in-order (X86MinorCPU) and out-of-order (X86O3CPU) configurations are directly tied to how

TABLE I
CPU CONFIGURATION AND BENCHMARK PERFORMANCE

CPU Configuration	Metric	Benchmarks		
		BFS	***KNN***	***ISORT***
X86O3CPU	CPI	2.871879	2.187114	2.389152
	IPC	0.348204	0.457223	0.418559
X86MinorCPU (limit=1)	CPI	7.995792	6.313829	8.823893
	IPC	0.125066	0.158382	0.113329
X86MinorCPU (limit=2)	CPI	7.832417	6.195678	8.770006
	IPC	0.127675	0.161403	0.114025
X86MinorCPU (limit=3)	CPI	7.853016	6.224563	8.768783
	IPC	0.127340	0.160654	0.114041
X86MinorCPU (limit=4)	CPI	7.831478	6.223995	8.768783
	IPC	0.127690	0.160669	0.114041

each benchmark is implemented. This is because of the inherent limitations and strengths of both CPU types.

Firstly, many of our chosen benchmarks contain sections where a significant portion of instructions are executed. The O3CPU has the ability to dynamically reorder instructions, allowing these critical regions to run more efficiently. If a benchmark's inner loop has several independent operations, the out-of-order CPU can exploit instruction-level parallelism (ILP), leading to a higher IPC and lower CPI. In contrast, the in-order pipeline must wait for each instruction to complete sequentially, even if some instructions are independent. This results in higher CPI and lower IPC.

For the serial BFS benchmark, the irregular memory access patterns and potential cache misses means that the CPU may have to wait for data to be fetched from memory. Interestingly, because the OC3CPU can schedule instructions while it waits for memory, it can effectively hide this latency. This is shown in the lower CPI for BFS on the O3CPU. The MinorCPU on the other hand has a higher CPI because it cannot easily overlap these memory delays with other useful work.

We also varied the system.cpu.executeIssueLimit for the in-order processor. The observation that performance (CPI and IPC) does not significantly change with different issue limits suggests that the bottleneck is not simply the issue width. Instead, it points to fundamental restrictions in the in-order execution model. Specifically, its inability to reorder instructions to overcome data and control hazards. This observation further connects the implementation's inherent characteristics (such as dependency chains and limited ILP) with the resulting performance under an in-order execution scheme.

The performance results in Table 1 reflect how each benchmark's implementation interacts with the underlying processor architecture. Benchmarks with high ILP, regular control flow, or predictable memory accesses benefit more from the out-of-order engine's ability to schedule instructions dynamically. Conversely, the in-order pipeline's strict sequential execution amplifies delays caused by memory latency and branch mispredictions. This analysis shows why the O3CPU has lower CPI and higher IPC, highlighting how loops, memory access patterns, and branch predictability affect performance.

IV. EXPLORATION 3

In this exploration, we investigate the impact of different cache configurations has on workload, as well as CPU performance. We executed 5 benchmark workloads on 4 different cache configurations to see which one is most and least affected by the cache configuration, then we compare the MinorCPU and the OP3CPU running the least volatile workload to see how CPU architecture interacts with the cache to affect performance.

A. Experimental setup

For consistency, we use the same system configuration as the previous explorations. 8GB of DDR3 1600 8x8 DRAM is allocated to both CPUs running at 75 MHz in timing mode. The 4 cache configurations are the following:

TABLE II
CACHE CONFIGURATIONS

Name	Specifications
Simple	4kB SimpleCache
Harvard	4kB L1 I, 4kB L1 D
Big L1D	4kB L1 I, 16kB L1 D
L2	4kB L1 I, 4kB L1 D, 32kB L2

These 4 configurations were selected as they presented a good balance of hierarchy as well as speed which will help us elucidate the varying levels of impact that cache might have.

We first sweep all combinations of workloads and cache configurations to find the least volatile workload (measured in SimTicks) in an attmept to control this variable, it should be noted that this performed on the X86O3 CPU. We then run that workload through all cache configurations to find the difference between the two CPUs.

B. Results and discussion

The result of the first phase is as follows:

TABLE III
PERFORMANCE IN DIFFERENT CACHE CONFIGURATIONS IN TICKS

Workload name	Simple	Harvard	Big L1D	L2
ISORT	5.623.E+10	2.002.E+10	1.856.E+10	2.031.E+10
BFS	5.583.E+10	1.976.E+10	1.834.E+10	2.011.E+10
SSA	5.559.E+10	1.970.E+10	1.829.E+10	2.005.E+10
NBODY	5.695.E+10	2.039.E+10	1.868.E+10	2.056.E+10
KNN	5.681.E+10	2.022.E+10	1.867.E+10	2.052.E+10

We can better see the difference through a graph (Fig 5), the results for simple cache is omitted since it's so above the others it does not provide useful insight. Through this, we see that the workload with the largest volatility is the NBODY benchmark, which is a floating point heavy benchmark and would likely work with very large variables requiring constant fetches. The least happens to be ISORT; we believe this is because since the sorting workload enjoys significant spatial locality, larger quantities of cache would have a reduced effect.

However, it is important to note that, in all workloads, a large L2 cache was actually detrimental to the performance.

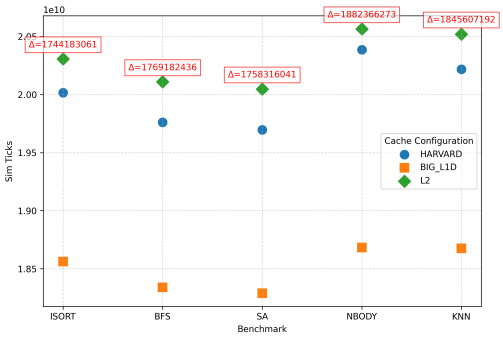

Fig. 5. Elapsed SimTick of CPUs under different cache configs

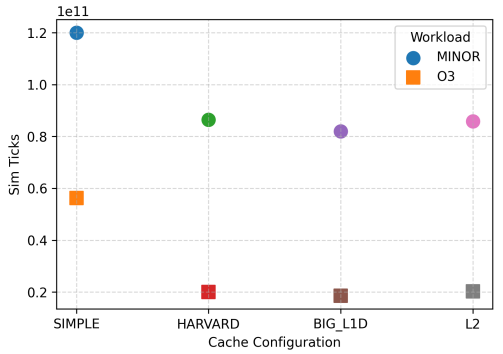

Fig. 6. Elapsed SimTick of workloads under different cache configs

We believe that this suggests that the benchmarks chosen do not feature enough cache / memory operations to the point that the benefits of accessing the slower L2 cache outweigh the overhead.

We can see in Fig.6 that there is a drastic difference between how the two CPUs are affected by the cache. The O3 CPU, likely due to its ability to execute instructions out of order, means that it can trade cache space to reduce stalling time. Again, we see that there is a large difference between a simple cache and a hierarchical cache. Although the difference shrinks between different configurations of hierarchical cache, suggesting that the additional space was not taken advantage of.

V. Conclusion

We investigated the effects of various input sizes, CPU architectures, and cache configurations on the PBBS benchmarks' performance through our set of tests. Our results emphasize a number of important findings:

First, the number of simulated instructions, execution cycles, and memory demands increased proportionately with increasing input size. However, different benchmarks displayed varied scaling characteristics, with DRAM bottlenecks causing noticeable performance reduction in memory-bound applications.

Second, on every benchmark, the out-of-order CPU fared better than the in-order CPU. Lower CPI and greater IPC were a result of O3CPU's capacity to dynamically reorder instructions and reduce stalls. The MinorCPU, on the other hand, was limited by its sequential execution model, leading to higher execution latencies, particularly for memory-bound workloads like BFS.

Third, memory-intensive benchmarks like NBODY were the most sensitive to cache hierarchy, according to our examination of various cache settings. Larger cache sizes helped workloads with great data locality, but they had little effect on workloads with erratic memory accesses. The best performance boost was consistently obtained using the L2 cache arrangement, indicating that DRAM latency is efficiently mitigated by an extra cache level.

All things considered, our findings show how crucial it is to modify system architectures to accommodate different workloads. Large L1 caches and out-of-order execution are advantageous for compute-bound applications like NBODY, but deeper cache hierarchies and better memory access patterns are advantageous for memory-bound workloads like BFS.

Despite our insights, certain limitations remain. For example, the inability to capture instruction mix due to gem5's MinorCPU bugs resulted in less effective data collection. Future research might examine hybrid workloads, improve cache hierarchy models, and look into other CPU configurations.

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: SCRIPT 1

data generation

```
./randomSeq -t int 1000 random_1k_int
./randomSeq -t int 2000 random_2k_int
./randomSeq -t int 4000 random_4k_int
./randomSeq -t int 8000 random_8k_int
./randomSeq -t int 16000 random_16k_int
./randomSeq -t int 32000 random_32k_int
./randomSeq -t int 64000 random_64k_int
./randomSeq -t int 128000 random_128k_int

./randLocalGraph -j -d 3 -m 1000 100 graph_100
./randLocalGraph -j -d 3 -m 2000 200 graph_200
./randLocalGraph -j -d 3 -m 4000 400 graph_400
./randLocalGraph -j -d 3 -m 8000 800 graph_800
./randLocalGraph -j -d 3 -m 16000 1600 graph_1600
./randLocalGraph -j -d 3 -m 32000 3200 graph_3200
./randLocalGraph -j -d 3 -m 64000 6400 graph_6400

./trigramSeq 1000 words_1k
./trigramSeq 2000 words_2k
./trigramSeq 4000 words_4k
./trigramSeq 8000 words_8k
./trigramSeq 16000 words_16k
./trigramSeq 32000 words_32k
./trigramSeq 64000 words_64k

./randPoints -d 3 100 nbody_100
./randPoints -d 3 200 nbody_200
./randPoints -d 3 400 nbody_400
./randPoints -d 3 800 nbody_800
./randPoints -d 3 1600 nbody_1600
./randPoints -d 3 3200 nbody_3200

./randPoints -d 3 1000 knn_1k
./randPoints -d 3 2000 knn_2k
./randPoints -d 3 4000 knn_4k
./randPoints -d 3 8000 knn_8k
./randPoints -d 3 16000 knn_16k
./randPoints -d 3 32000 knn_32k
./randPoints -d 3 64000 knn_64k
./randPoints -d 3 128000 knn_128k
```

run_pbbs.py

```
import m5
from m5.objects import *

import argparse

parser = argparse.ArgumentParser()
parser.add_argument(
    '-a', '--binary_args',
    default='random_10k_int',
    help=''
)
args = parser.parse_args()
```

```
binary_args = args.binary_args


# System creation
system = System()

## gem5 needs to know the clock and voltage
system.clk_domain = SrcClockDomain()
system.clk_domain.clock = '75MHz'
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

## 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

# switch out binaries when investigating other benchmarks
process.executable = '/u/csc368h/winter/pub/workloads/pbbsbench/benchmarks/integerSort/serialR
process.cmd = [process.executable] + binary_args.split()

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

print(f"\nbeginning simulation running {process.executable} {binary_args} ...")
exit_event = m5.simulate()
print("done because {}".format(exit_event.getCause()))
```

run_.py

```
import subprocess

# list of input files
input_files = [
    'knn_1k',
    'knn_2k',
    'knn_4k',
    'knn_8k',
    'knn_16k',
    'knn_32k',
    'knn_64k',
    'knn_128k'
]

gem5_bin = '/u/csc368h/winter/pub/bin/gem5.opt'

gem5_script = 'run_pbbs.py'

for infile in input_files:
    outdir_name = f"{infile}_out"

    cmd = [
        gem5_bin,
        f'--outdir={outdir_name}',
        gem5_script,
        '--binary_args', infile
    ]

    print(f"\nrunning gem5 with input file: {infile}")
    print("command: " + " ".join(cmd))

    subprocess.run(cmd, check=True)
```

plot_.py

```
    import os
import re
import matplotlib.pyplot as plt

KEY_METRICS = [
    "simInsts",
    "simOps",
    "system.cpu.numCycles",
    "simSeconds",
    "hostSeconds",
    "system.cpu.ipc",
    "system.cpu.cpi",
    "system.mem_ctrl.readReqs",
    "system.mem_ctrl.writeReqs",
    "system.mem_ctrl.avgRdBWSys",
```

```
    "system.mem_ctrl.avgWrBWSys"
]

OUTDIRS = [
    "nbody_100_out",
    "nbody_200_out",
    "nbody_400_out",
    "nbody_800_out",
    "nbody_1600_out",
    "nbody_3200_out"
]

INPUT_SIZES = [100, 200, 400, 800, 1600, 3200]

def parse_three_stats_dumps(stats_file):
    """
    read 'stats.txt' file which has three dumps
    returns list [setup_lines, execution_lines, teardown_lines],
    where each element is a list of lines for that region.
    """
    with open(stats_file, 'r') as f:
        lines = f.readlines()

    regions = []
    capturing = False
    current_block = []

    for line in lines:
        if "Begin Simulation Statistics" in line:
            capturing = True
            current_block = []
        elif "End Simulation Statistics" in line:
            capturing = False
            regions.append(current_block)
        elif capturing:
            current_block.append(line)

    return regions


def extract_key_metrics_from_lines(lines):
    """
    parse key metrics, returns a dict { metric_name: float_value }.
    """
    results = {}
    pattern = re.compile(r'^(\S+)\s+([\d.eE+\-NaN]+)\s+.*')

    for line in lines:
        match = pattern.match(line.strip())
        if match:
            key = match.group(1)
            val_str = match.group(2)
            if key in KEY_METRICS:
                # parse as float
                try:
                    results[key] = float(val_str)
                except ValueError:
                    results[key] = float('nan') if val_str == 'nan' else val_str
```

```
    return results


def main():
    data = {}

    # parse
    for outdir in OUTDIRS:
        stats_path = os.path.join(outdir, "stats.txt")
        if not os.path.isfile(stats_path):
            print(f"[ERROR] {stats_path} not found, skipping.")
            continue

        regions = parse_three_stats_dumps(stats_path)
        if len(regions) != 3:
            print(f"[ERROR] {stats_path} has {len(regions)} dumps")
            continue

        # only want execution tims
        execution_lines = regions[1]
        metrics = extract_key_metrics_from_lines(execution_lines)
        data[outdir] = metrics


    x_values = INPUT_SIZES

    for metric in KEY_METRICS:
        y_values = []
        valid_x = []

        for i, outdir in enumerate(OUTDIRS):
            if outdir in data and metric in data[outdir]:
                val = data[outdir][metric]
                y_values.append(val)
                valid_x.append(x_values[i])
            else:
                print(f"[WARNING] missing {metric} for {outdir}")


        if not y_values:
            continue

       #linear plot
        plt.figure(figsize=(6, 4))
        plt.plot(valid_x, y_values, marker='o', linestyle='--', color='b')
        plt.title(f"{metric} vs. Input Size (Linear Scale)")
        plt.xlabel("Input Size (# of elements)")
        plt.ylabel(metric)

        xtick_labels = []
        for val in valid_x:
            if val >= 1_000_000:
                xtick_labels.append(f"{val//1_000_000}M")
            elif val >= 1_000:
                xtick_labels.append(f"{val//1_000}k")
            else:
                xtick_labels.append(str(val))
        plt.xticks(valid_x, xtick_labels)
```

```
        plt.tight_layout()
        out_plot_linear = f"{metric}_linear.png"
        plt.savefig(out_plot_linear)
        plt.close()
        print(f"Saved plot: {out_plot_linear}")

        # log log plot
        plt.figure(figsize=(6, 4))
        plt.plot(valid_x, y_values, marker='o', linestyle='--', color='r')
        plt.title(f"{metric} vs. Input Size (Log-Log Scale)")
        plt.xlabel("Input Size (# of elements)")
        plt.ylabel(metric)
        plt.xscale('log')
        plt.yscale('log')

        plt.xticks(valid_x, xtick_labels, rotation=0)

        plt.tight_layout()
        out_plot_loglog = f"{metric}_loglog.png"
        plt.savefig(out_plot_loglog)
        plt.close()
        print(f"saved plot: {out_plot_loglog}")

    print("done")


if __name__ == "__main__":
    main()
```

plot_distribution.py

```
    #!/usr/bin/env python3

import os
import re
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.patches import Patch

KEY_METRICS = [
    "simInsts",
    "simOps",
    "system.cpu.numCycles",
    "simSeconds",
    "hostSeconds",
    "system.cpu.ipc",
    "system.cpu.cpi",
    "system.mem_ctrl.readReqs",
    "system.mem_ctrl.writeReqs",
    "system.mem_ctrl.avgRdBWSys",
    "system.mem_ctrl.avgWrBWSys"
]

OUTDIRS = [
    "words_1k_out",
    "words_2k_out",
    "words_4k_out",
```

```
    "words_8k_out",
    "words_16k_out",
    "words_32k_out",
    "words_64k_out",
]

INPUT_LABELS = ["1000", "2000", "4000", "8000", "16000", "32000", "64000"]

PHASES = ["Setup", "Execution", "Teardown"]


def parse_three_stats_dumps(stats_file):
    """
    read 'stats.txt' file which has three dumps
    returns list [setup_lines, execution_lines, teardown_lines],
    where each element is a list of lines for that region.
    """
    with open(stats_file, 'r') as f:
        lines = f.readlines()

    regions = []
    capturing = False
    current_block = []

    for line in lines:
        if "Begin Simulation Statistics" in line:
            capturing = True
            current_block = []
        elif "End Simulation Statistics" in line:
            capturing = False
            regions.append(current_block)
        elif capturing:
            current_block.append(line)

    return regions

def extract_key_metrics(lines):
    """
    parse key metrics, returns a dict { metric_name: float_value }.
    """
    results = {}
    pattern = re.compile(r'^(\S+)\s+([\d.eE+\-NaN]+)\s+.*')
    for line in lines:
        match = pattern.match(line.strip())
        if match:
            key = match.group(1)
            val_str = match.group(2)
            if key in KEY_METRICS:
                try:
                    results[key] = float(val_str)
                except ValueError:
                    results[key] = float('nan')
    return results

def main():
    # data[outdir][phase][metric] = value
    data = {}
```

```
for outdir in OUTDIRS:
    stats_path = os.path.join(outdir, "stats.txt")
    if not os.path.isfile(stats_path):
        print(f"[ERROR] {stats_path} not found, skipping.")
        continue

    regions = parse_three_stats_dumps(stats_path)
    if len(regions) != 3:
        print(f"[ERROR] {stats_path} has {len(regions)} dumps")
        continue

    data[outdir] = {}
    for i, phase_name in enumerate(PHASES):
        lines = regions[i]
        metrics_dict = extract_key_metrics(lines)
        data[outdir][phase_name] = metrics_dict

# one bar chart per metric
# create a color map for the bars of each input size
cmap = plt.cm.get_cmap("tab10")
num_inputs = len(OUTDIRS)
colors = [cmap(i % 10) for i in range(num_inputs)]

for metric in KEY_METRICS:
    # phase_percentages[phase][idx_outdir] = percentage for that outdir & phase
    phase_percentages = {ph: [] for ph in PHASES}

    for idx, outdir in enumerate(OUTDIRS):
        if outdir not in data:
            for ph in PHASES:
                phase_percentages[ph].append(0.0)
            continue

        val_setup = data[outdir]["Setup"].get(metric, 0.0)
        val_exec  = data[outdir]["Execution"].get(metric, 0.0)
        val_teard = data[outdir]["Teardown"].get(metric, 0.0)

        total = val_setup + val_exec + val_teard
        if total == 0:
            pct_setup, pct_exec, pct_teard = 0.0, 0.0, 0.0
        else:
            pct_setup = 100.0 * (val_setup / total)
            pct_exec  = 100.0 * (val_exec  / total)
            pct_teard = 100.0 * (val_teard / total)

        phase_percentages["Setup"].append(pct_setup)
        phase_percentages["Execution"].append(pct_exec)
        phase_percentages["Teardown"].append(pct_teard)

    fig, ax = plt.subplots(figsize=(8, 5))
    x_phase = np.arange(len(PHASES))  # [0,1,2]
    bar_width = 1.0 / (num_inputs + 1)

    # shift horizontally
    offset_base = - (num_inputs - 1) * bar_width / 2.0

    for i, outdir in enumerate(OUTDIRS):
```

```
            y_vals = [
                phase_percentages["Setup"][i],
                phase_percentages["Execution"][i],
                phase_percentages["Teardown"][i]
            ]
            x_positions = x_phase + (offset_base + i*bar_width)

            ax.bar(
                x_positions,
                y_vals,
                width=bar_width,
                color=colors[i],
                alpha=0.85
            )

        ax.set_xticks(x_phase)
        ax.set_xticklabels(PHASES)
        ax.set_ylabel("Percentage (%)")
        ax.set_title(f"Distribution of {metric} across Setup/Execution/Teardown")

        legend_patches = []
        for i in range(num_inputs):
            patch = Patch(color=colors[i], label=INPUT_LABELS[i])
            legend_patches.append(patch)

        ax.legend(handles=legend_patches, title="Input Sizes", loc="best")

        plt.tight_layout()
        outname = f"{metric}_phase_distribution.png"
        plt.savefig(outname)
        plt.close()

        print(f"saved: {outname}")

    print("done")

if __name__ == "__main__":
    main()
```

APPENDIX B: SCRIPT 2

run_pbbs.py

```
import m5
from m5.objects import *

import argparse

parser = argparse.ArgumentParser()
parser.add_argument(
    '-a', '--binary_args',
    help=''
)
parser.add_argument(
    '-c', '--cpu_type', type=str, default='o3cpu'
)
parser.add_argument(
    '-l', '--issue_limit', type=int, default='1'
)
args = parser.parse_args()
binary_args = args.binary_args
cpu_type = args.cpu_type
issue_limit = args.issue_limit

# System creation
system = System()

## gem5 needs to know the clock and voltage
system.clk_domain = SrcClockDomain()
system.clk_domain.clock = '75MHz'
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
if cpu_type == 'o3':
    system.cpu = X86O3CPU()
    print("Using o3cpu")
elif cpu_type == 'minor':
    system.cpu = X86MinorCPU()
    system.cpu.executeIssueLimit = issue_limit
    print("Using minorCPU with issue limit of: " + str(issue_limit))

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

# switch out binaries when investigating other benchmarks
process.executable = '/u/csc368h/winter/pub/workloads/pbbsbench/benchmarks/integerSort/serialR
process.cmd = [process.executable] + binary_args.split()

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

print(f"\nbeginning simulation running {process.executable} {binary_args} ...")
exit_event = m5.simulate()
print("done because {}".format(exit_event.getCause()))
```

run.py

```
    import subprocess

# list of input files
input_args = [['minor', '1'],['minor', '2'],['minor', '3'],['minor', '4'],['o3', '1']]

gem5_bin = '/u/csc368h/winter/pub/bin/gem5.opt'

gem5_script = 'run_pbbs.py'

input_file = 'data/sequenceInt'

for args in input_args:
    outdir_name = str(args[0]) + "-" + str(args[1]) + "-out"

    cmd = [
        gem5_bin,
        f'--outdir={outdir_name}',
        gem5_script,
        '--binary_args', input_file,
        '--cpu_type', args[0],
        '--issue_limit', args[1]
    ]
```

```
print(f"\nrunning gem5 with " + str(args[0]) + "-" + str(args[1]))
print("command: " + " ".join(cmd))

subprocess.run(cmd, check=True)
```

APPENDIX C: SCRIPT 3

part_1_run_pbbs.py

```
import m5
from m5.objects import *

import argparse

binaries = {
    "ISORT": "/u/csc368h/winter/pub/workloads/pbbsbench/benchmarks/integerSort/serialRadixSort
    "BFS": "/u/csc368h/winter/pub/workloads/pbbsbench/benchmarks/breadthFirstSearch/serialBFS/
    "SA": "/u/csc368h/winter/pub/workloads/pbbsbench/benchmarks/suffixArray/serialDivsufsort/S
    "NBODY": "/u/csc368h/winter/pub/workloads/pbbsbench/benchmarks/nBody/parallelCK/nbody",
    "KNN": "/u/csc368h/winter/pub/workloads/pbbsbench/benchmarks/nearestNeighbors/octTree/neig
}

parser = argparse.ArgumentParser()
parser.add_argument(
    '-b', '--binary',
    default='ISORT',
    help="""Usage: binary args specify the benchmarks to run.
            Options: ISORT, BFS, SA, NBODY, KNN"""
)
parser.add_argument(
    '-c', '--cache',
    default="SIMPLE",
    help="""Usage: specify how to configure the cache.
            Options: SIMPLE, HARVARD, BIG_L1D, L2"""
)
parser.add_argument(
    '--cpu',
    default="O3",
    help="""Usage: specify which CPU to use
            Options: MINOR, O3"""

)

args = parser.parse_args()
binary_path = binaries[args.binary]
cache_type = args.cache
cpu_type = args.cpu

system = System()

system.clk_domain = SrcClockDomain()
system.clk_domain.clock = '75MHz'
system.clk_domain.voltage_domain = VoltageDomain()

system.membus = SystemXBar()
system.system_port = system.membus.cpu_side_ports

system.mem_mode = 'timing'

if (cpu_type == 'MINOR'):
    system.cpu = X86MinorCPU()
else:
    system.cpu = X86O3CPU()
```

```
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

# set up cache

if (cache_type == "SIMPLE"):
    system.cache = SimpleCache(size="4kB")
    system.cpu.icache_port = system.cache.cpu_side
    system.cpu.dcache_port = system.cache.cpu_side
    system.cache.mem_side = system.membus.cpu_side_ports
elif (cache_type == "HARVARD"):
    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'
    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
elif (cache_type == "BIG_L1D"):
    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='16kB'
    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
```

```
else: # cache type L2
    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'
    # L1 data cache setup
    system.cpu.l1d = L1DCache()
    system.cpu.l1d.cpu_side = system.cpu.dcache_port

    # L1 instruction cache
    system.cpu.l1i = L1ICache()
    system.cpu.l1i.cpu_side = system.cpu.icache_port

    # L2 cache setup
    system.l2_cache = L2Cache()
    system.l2_bus = L2XBar()

    system.cpu.l1d.mem_side = system.l2_bus.cpu_side_ports
    system.cpu.l1i.mem_side = system.l2_bus.cpu_side_ports
    system.l2_cache.mem_side = system.membus.cpu_side_ports
    system.l2_cache.cpu_side = system.l2_bus.mem_side_ports

system.mem_ctrl = MemCtrl()
system.mem_ctrl.port = system.membus.mem_side_ports
system.mem_ctrl.dram = DDR3_1600_8x8()
address_ranges = [AddrRange('8GB')]
system.mem_ranges = address_ranges
system.mem_ctrl.dram.range = address_ranges[0]

process = Process()
process.executable = binary_path
process.cmd = [process.executable]
system.workload = SEWorkload.init_compatible(process.executable)
system.cpu.workload = process
system.cpu.createThreads()

root = Root(full_system=False, system=system)
m5.instantiate()
```

```
print(f"\nbeginning simulation running {process.executable}...")
exit_event = m5.simulate()
print("done because {}".format(exit_event.getCause()))
```

helper.py

```
import subprocess

gem5_bin = '/u/csc368h/winter/pub/bin/gem5.opt'
gem5_script = '/h/u8/c1/00/liuzeku4/csc368/hw1/memory_run.py'

binaries = [
    "ISORT",
    "BFS",
    "SA",
    "NBODY",
    "KNN"
]

cache_configs = [
    "SIMPLE",
    "HARVARD",
    "BIG_L1D",
    "L2"
]

for bin in binaries:
    for conf in cache_configs:
        outdir = f"{conf}_{bin}_out"
        cmd = [
            gem5_bin,
            f'--outdir={outdir}',
            gem5_script,
            '--binary', bin,
            '--cache', conf
        ]
        print(f"\n running gem5 with binary {bin} and cache conf {conf}")
        print("command: " + " ".join(cmd))

        subprocess.run(cmd, check=True)
```

part_2_run_pbbs.py

```
import subprocess

gem5_bin = '/u/csc368h/winter/pub/bin/gem5.opt'
gem5_script = '/h/u8/c1/00/liuzeku4/csc368/hw1/memory_run.py'

cpus = [
    "MINOR",
    "O3"
]

cache_configs = [
    "SIMPLE",
    "HARVARD",
    "BIG_L1D",
    "L2"
]
```

```
for cpu in cpus:
    for conf in cache_configs:
        outdir = f"{conf}_{cpu}_out"
        cmd = [
            gem5_bin,
            f'--outdir={outdir}',
            gem5_script,
            '--cpu', cpu,
            '--cache', conf
        ]
        print(f"\n running gem5 with cpu {cpu} and cache conf {conf}")
        print("command: " + " ".join(cmd))

        subprocess.run(cmd, check=True)
```