Characterizing PBBS: Speculation in the memory and pipeline

Edward Han Leon Cai Matthew Liu

I. Introduction

The Problem-Based Benchmark Suite (PBBS) provides diverse workloads, allowing us to analyze performance across different computational patterns. 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 impact of speculation and branch prediction in computer processors and observe the impact and usecases they may have.

II. Exploration 1

In this exploration, we investigate how prefetchers attached to L1 instruction and data caches impact performance across a subset of PBBS benchmarks. We evaluated both spatial locality exploitation (via sequential prefetching) and stride-based patterns, measuring their effects on execution time, cache efficiency, and memory traffic. Our analysis focuses on distinguishing workloads with regular access patterns (e.g., sorting algorithms) from those with irregular memory behavior (e.g., graph traversals).

A. Experimental setup

We simulate three PBBS benchmarks—radix_sort, BFS, and KNN, on a gem5 configuration with an X86O3CPU at 75 MHz and 8GB DDR3 memory. For each benchmark, we performed the following runs: a run without any prefetching as a baseline, a run with a TaggedPrefetcher installed on the L1 instruction cache, a run with a StridePrefetcher installed on the L1 data cache, and finally a run with a TaggedPrefetcher installed on the L1 data cache. Both cache sizes were set to 4 KB for all runs and the prefetchers were configured to be of degree 2.

We chose to observe the following statistics:

- `system.cpu.ipc`: Measures improvements in pipeline efficiency due to prefetching.

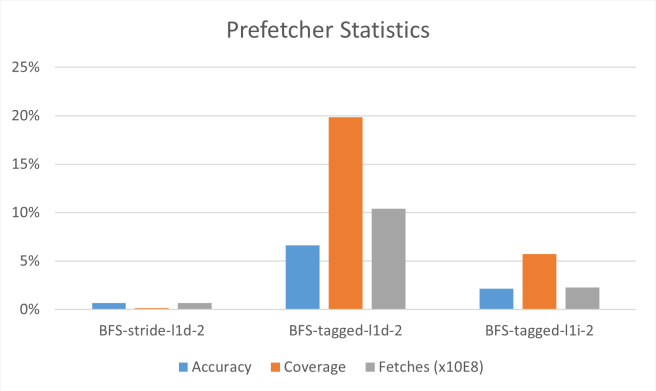

Fig. 1. Performance statistics of different prefetchers on BFS

- `system.cpu.l1d/i.overallMissRate`: Measures the impact of prefetching on cache miss rates.
- `system.cpu.l1d/i.prefetcher.pfIssued`, `system.cpu.l1d/i.prefetcher.accuracy`, and `system.cpu.l1d/i.prefetcher.coverage`: Measure the effectiveness of the prefetcher.

B. Results and discussion

We find that for the two graph traversal-based benchmarks, KNN and BFS, installing a prefetcher had little to negative benefit on the IPC measure. In fact, for BFS installing a prefetcher worsened the IPC by up to 7.6%. On the other hand, prefetching consistently improved radix sort IPC by up 5.3%. The differences can be explained by the nature of the workload itself, it is very difficult to consistent predict the subsequent node during graph traversal, resulting in the prefetcher being very inaccurate, and sparse. On the other hand, radix sort enjoys a lot of temporal and spatial locality, thus the prefetcher has a much better performance as well: up to 34% in accuracy and 49% in coverage.

For BFS, we note that the usage of the prefetcher is inversely correlated with the performance of the cache. This result can be explained by the very poor accuracy of the prefetcher. L1D tagged, which issued far more fetches than the other two had an accuracy of 7% while L1D stride and L1I tagged suffered an abysmal 1% and 2% respectively. The inverse performance gain can therefore be explained as the prefetcher polluting the cache and occupying bandwidth without providing much benefit.

Similar to BFS, our prefetchers had little positive impact on the efficiency of the processor during KNN bench-

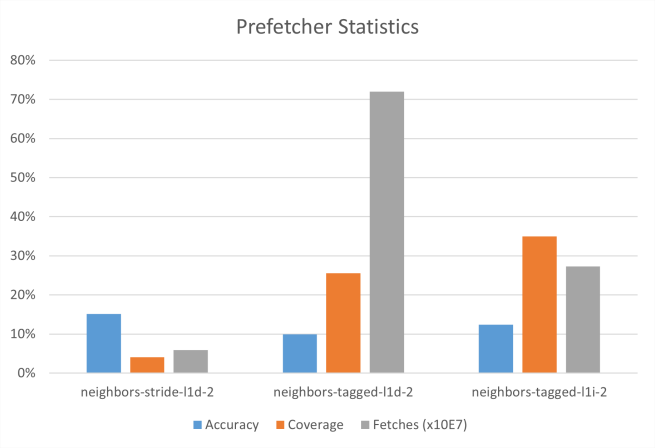

Fig. 2. Performance statistics of different prefetchers on KNN

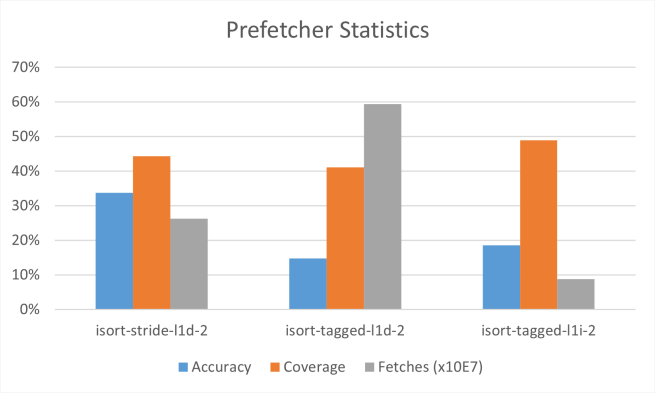

Fig. 3. Performance statistics of different prefetchers on Radix Sort

marks. However, in contrast to BFS two prefetchers: L1D stride and L1I tagged actually had a positive impact on the efficiency proportional to their usage. Although both prefetchers increased the number of accesses, they had an accuracy of 10%, which is enough to offset the extra bandwidth cost. The fact that the two caches benefitted from different types of prefetchers is interesting as it can be extrapolated that KNN operates on data that enjoys spatial locality while the instructions benefits from temporal locality instead.

Finally, radix sort is a benchmark with high locality, and this can be observed through the relatively high accuracy and coverage of each prefetcher. In particular, the stride prefetcher on L1D had an accuracy of 34% and a coverage of 44%. This was sufficient to reduce the cache miss percentage by about 25% compared to no prefetching. Again, comparing the stride and tagged prefetchers on the data cache reveals that this benchmark operates on spatially local data. Another interesting observation is that despite the relatively poor accuracy of the tagged L1D prefetcher, and despite issuing about twice as many fetches, it has a similar memory access count to the stride prefetcher, and did not increase the amount of misses by much. Digging deeper into the data, we find that while the tagged prefetcher had more than 3 times the number of unused fetches as stride (729396 v 187649), it had an almost identical count of useful fetches (874479 : 884873). Thus, it can be concluded that the benefits of a successful fetch out weighs the cost of the extra bandwidth load. This result may be different however on a more constrained memory system.

As a final point of discussion, though the IPC improvements for the instruction cache prefetchers were all small relative to data cache prefetchers, we note that the L1I prefetcher issued far fewer fetches. Normalizing the IPC improvement by the fetch count demonstrates that the L1I prefetcher resulted improvements on par if not better than the L1D prefetchers. The small number of fetches is likely due to the relative size difference between the data the algorithms operate on and the size of the code the algorithm is written in. It will be interesting to benchmark the performance of prefetchers on very large software like a database or operating system.

III. Exploration 2

In this exploration, we investigate how branch prediction accuracy changes depending on the benchmark and branch predictor configuration. Focusing on the region of interest, we predict that in general, as predictor size and the number of bits used in each saturating counter (ctrbits) increase, the accuracy will increase, though it may vary based on the benchmark. We also investigate how the size of the tournament branch predictor impacts accuracy. We predict that increasing size of local, global, and choice predictor size and ctrbits will also increase accuracy.

A. Experimental setup

We chose a system configuration with an X8603CPU, at 75MHz and 1V, L1I and L1D cache with no prefetching. We found more interesting results with the X86O3CPU as opposed to a simple cpu. Frequency and voltage were chosen for consistency with the previous assignment, and prefetching was removed to observe only the accuracy of branch prediction. To investigate branch prediction accuracy, we chose a simple LocalBP and varied its localPredictorSize and localCtrBits parameters to localPredictorSize of (512, 2048, 8196) and localCtrBits of 2, and localPredictorSize of 2048 and localCtrBits of (1, 2, 4) to test the effect of each. We then chose the worst performing benchmark, and attempt to use TournamentBP with varying configurations to achieve a high branch prediction accuracy.

B. Results and discussion

Figure 4 shows minimal accuracy changes when varying localPredictorSize with localCtrBits fixed at 2, with a maximum increase of 0.28%. The key observation is in Figure 5, where increasing localCtrBits (with localPredictorSize fixed at 2048) significantly improves accuracy,

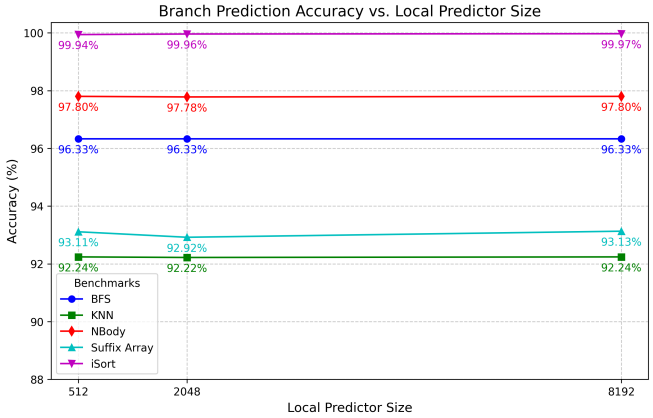

Fig. 4. Accuracy for varying localPredictorSize, localCtrBits = 2, with varying benchmarks

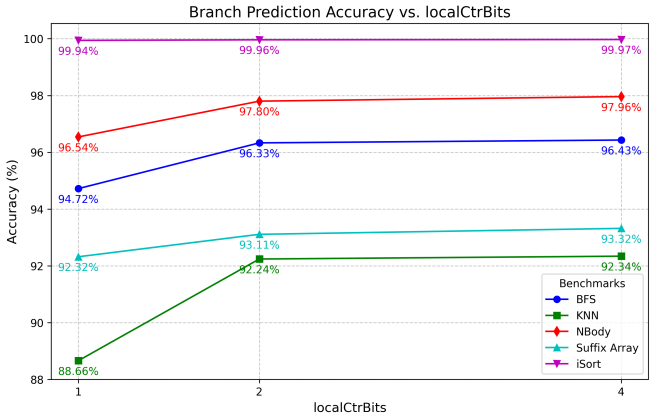

Fig. 5. Accuracy for varying localCtrBits, localPredictorSize = 2048, with varying benchmarks

particularly in KNN, which sees a 4% gain. We also note that there seems to be a varying degree of branch predictor accuracy depending on the benchmark, with iSort performing nearly perfectly. This disparity arises because LocalBP() indexes into a table of counters using the PC. In find_leaf function in benchmarks/nearestNeighbours/octTree/k_nearest_neighbors.h, we see the condition `if(lookup_bit(searchInt, current -> bit) == 0)`, which lacks pattern for whether the call to lookup_bit returns 0 or 1, so the branch predictor frequently mispredicts. Another key source of mispredictions is the recursive tree travel found in `k_nearest_rec`, which decides which subtree to explore based on a bounding box. The condition `within_epsilon_box(sqrt(max_distance)` is unpredictable as `sqrt(max_distance)` updates when more neighbours are found. Additionally, there are nested conditions comprising of three additional checks. These checks depend on dynamically changing node states (is_leaf(), Left(), etc) which makes it challenging for the saturating counters to identify a stable pattern.

On the reverse side, iSort performs exceptionally well

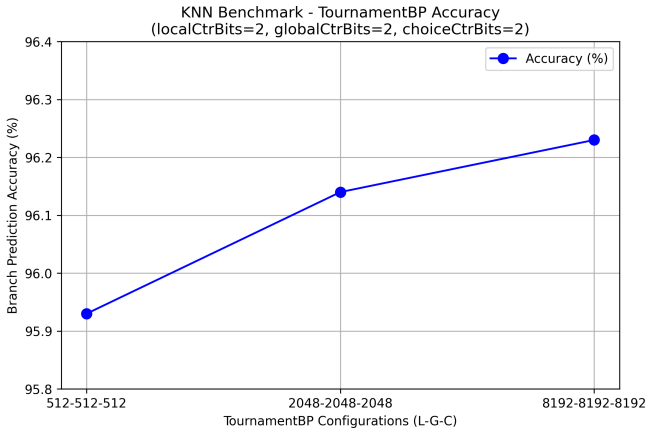

Fig. 6. Accuracy for varying predictor size, CtrBits = 2, with KNN

with LocalBP(). In `seq_count_sort_` (parlaylib/include/parlay/internal/counting_sort.h, lines 82 to 92), the code runs simple for loopsfor a consistent number of iterations with no early breaks. Since all loops occur at the same PC across iterations, the LocalBP learns to predict these branches as taken. In addition, we notice there are no data-dependent branch skips, unlike KNN, meaning consistent control flow across loop bodies. Additionally, the while loop in `seq_radix_sort` is deterministic and highly predictable, which contributes to the high accuracy. In `seq_write_down` lines 71-74, the loop condition `j >= 0`, results in the branch being always taken until `j == -1`, allowing the counter to quickly settle on "taken." Overall, we see that the code lends itself well to the LocalBP(), as there is little variation in taken/not taken for each PC.

Next, we observe how the size of a tournament branch predictor impacts accuracy. We set local, global, and choice counter bits to the default - 2 - and vary the local, global, and choice predictor sizes. We notice that with little configuration, the branch predictor accuracy is already roughly 3% better compared to LocalBP(). We also observe that increasing the size of the predictor has a minor but positive impact on the accuracy of the branch predictor. We see the same pattern when keeping local, global, and choice predictor size constant, and varying the number of saturating counter bits. We also notice the logarithmic shape of Figure 7, suggesting that there is some inherent unpredictability, even with a tournament predictor.

The sudden improvement when switching to a tournament BP is primarily due to the inclusion of a global predictor and choice predictor. In previously mentioned functions, the branching behavior depends on the data structure, which a local predictor fails to see. The global predictor can recognize high level patterns, which when paired with the choice predictor, can "learn" which is better for each branch. For example, some branches like `if (bit == 0 || n < node_cutoff)` depend on the local struc-

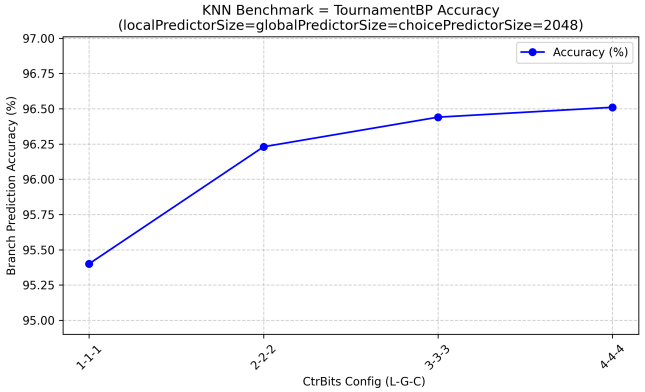

Fig. 7. Accuracy for varying ctrBits, predictorSize = 2048, with KNN

ture, whereas branches like `if(distance(T->Left()) < distance(T->Right()))` rely on previous behavior, which makes global history more effective.

IV. Conclusion

Our investigation explored the impact of prefetching, and the impact of branch prediction strategies on execution performance. Our test results emphasized a number of key findings:

The effectiveness of prefetching depends on the workloads involved. When a workload does not respond well to prefetching, the extra costs to the memory system hamper performance. However, generally the increased costs to the memory bandwidth seems to be worthwhile for the IPC improvements.

Secondly, branch predictor accuracy generally seems to improve with predictor size and number of saturating counter bits. However, the effectiveness of this varies depending on the benchmark. Irregular control flows, like KNN saw substantial improvements when switching from LocalBP to TournamentBP, which leveraged global history for more accurate predictions. Benchmarks with predicatable control flows, like iSort, performed exceptionally well even with LocalBP.

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] Smith, Alan Jay. "Sequential program prefetching in memory hierarchies." Computer 11.12 (1978): 7-21.

[3] Chen, Tien-Fu, and Jean-Loup Baer. "Effective hardware-based data prefetching for high-performance processors." IEEE transactions on computers 44.5 (1995): 609-623.

[4] Yeh, Tse-Yu, and Yale N. Patt. "Two-level adaptive training branch prediction." Proceedings of the 24th annual international symposium on Microarchitecture. 1991.

[5] Bienia, Christian, et al. "The PARSEC benchmark suite: Characterization and architectural implications." Proceedings of the 17th international conference on Parallel architectures and compilation techniques. 2008.

Appendix A: Script 1

exploration_one_run_pbbs.py

```
import m5
from m5.objects import *

import argparse

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'

parser = argparse.ArgumentParser()
parser.add_argument(
    '-a', '--binary_args',
    help=''
)
parser.add_argument(
    '-b', '--binary',
    help='Path to the binary executable'
)
parser.add_argument(
    '-p', '--prefetcher',
    help='Type of prefetcher to use (tagged or stride)'
)
parser.add_argument(
    '-c', '--cache',
    help='Type of cache to use (l1i or l1d)'
)
parser.add_argument(
    '-d', '--degree',
    help='Degree of prefetching'
)

args = parser.parse_args()
binary_args = args.binary_args
binary = args.binary
prefetcher = args.prefetcher
attached_cache = args.cache
prefetch_degree = int(args.degree)

# 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()

## 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
system.cpu.l1d = L1DCache()
system.cpu.l1d.mem_side = system.membus.cpu_side_ports
system.cpu.l1d.cpu_side = system.cpu.dcache_port
if attached_cache =='l1d':
    if prefetcher == 'tagged':
        system.cpu.l1d.prefetcher = TaggedPrefetcher(degree=prefetch_degree)
    elif prefetcher == 'stride':
        system.cpu.l1d.prefetcher = StridePrefetcher(degree=prefetch_degree)

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

if attached_cache == 'l1i':
    if prefetcher == 'tagged':
        system.cpu.l1i.prefetcher = TaggedPrefetcher(degree=prefetch_degree)
    elif prefetcher == 'stride':
        system.cpu.l1i.prefetcher = StridePrefetcher(degree=prefetch_degree)


# 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 = binary
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()))
```

exploration_one_run.py

```
import subprocess

# list of input files
input_args = [
    ['tagged', 'l1i', '1'], ['tagged', 'l1i', '2'], ['tagged', 'l1i', '4'], ['tagged', 'l1i', '8'],
    ['tagged', 'l1d', '1'], ['tagged', 'l1d', '2'], ['tagged', 'l1d', '4'], ['tagged', 'l1d', '8'],
    ['stride', 'l1i', '1'], ['stride', 'l1i', '2'], ['stride', 'l1i', '4'], ['stride', 'l1i', '8'],
    ['stride', 'l1d', '1'], ['stride', 'l1d', '2'], ['stride', 'l1d', '4'], ['stride', 'l1d', '8'],
]

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

binary = '/u/csc368h/winter/pub/workloads/pbbsbench/benchmarks/integerSort/serialRadixSort/isort'

gem5_script = 'run_pbbs.py'

input_file = 'data/sequenceInt'

for args in input_args:

    outdir_name = '-'.join(args)
    outdir_name = f"{binary.split('/')[-1]}-{outdir_name}"

    cmd = [
        gem5_bin,
        f'--outdir={outdir_name}',
        gem5_script,
        '--binary', binary,
        '--binary_args', input_file,
        '--prefetcher', args[0],
        '--cache', args[1],
        '--degree', args[2]
    ]
    print(' '.join(cmd))
```

```
print(f"\nrunning gem5 with " + outdir_name)
print("command: " + " ".join(cmd))

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

Appendix B: Script 2

data generation

```
./randomSeq -t int 64000 random_64k_int
./randLocalGraph -j -d 3 -m 32000 3200 graph_3200
./trigramSeq 32000 words_32k
./randPoints -d 3 1600 nbody_1600
./randPoints -d 3 16000 knn_16k
```

run_pbbs_knn.py

```
    import m5
from m5.objects import *

import argparse

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'


parser = argparse.ArgumentParser()
parser.add_argument(
    '-a', '--binary_args',
    default='',
    help='Command-line arguments to pass to the binary (default: random_10k_int)'
)
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.branchPred = LocalBP(localPredictorSize=512, localCtrBits=2) #switch here


## 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
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

process.executable = '/u/csc368h/winter/pub/workloads/pbbsbench/benchmarks/nearestNeighbors/octTree/ neighb
process.cmd = [process.executable, "-k", "1", "-d", "3"] + 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 with X86O3CPU, running {process.executable} {binary_args} ...")
exit_event = m5.simulate()
print("Exiting @ tick {} because {}"
```

```
      .format(m5.curTick(), exit_event.getCause()))
```

also run_pbbs_bfs.py, run_pbbs_knn.py, run_pbbs_nbody.py, run_pbbs_isort.py, run_pbbs_sa.py" with just the executable switched out

run.py

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

import subprocess

benchmarks = [
    ("run_pbbs_bfs.py", "graph_3200"),
    ("run_pbbs_knn.py", "knn_16k"),
    ("run_pbbs_nbody.py", "nbody_1600"),
    ("run_pbbs_isort.py", "random_64k_int"),
    ("run_pbbs_sa.py", "words_32k")
]

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


for script, input_data in benchmarks:
    outdir = f"{script[:-3]}_{input_data}_out"

    cmd = [
        gem5_bin,
        f'--outdir={outdir}',
        script,
        '--binary_args', input_data
    ]

    print(f"\nRunning gem5 for {script} with data={input_data}")
    print("Command: " + " ".join(cmd))

    subprocess.run(cmd, check=True)

print("\nAll benchmarks completed.")
```

plot.py

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

import os
import re
import matplotlib.pyplot as plt


KEY_METRICS = [
    "system.cpu.branchPred.condPredicted",
    "system.cpu.branchPred.condIncorrect"
]


OUTDIRS = {
    "run_pbbs_bfs_graph_3200_out": 2048,
    "run_pbbs_knn_knn_16k_out": 2048,
    "run_pbbs_nbody_nbody_1600_out": 2048,
    "run_pbbs_sa_words_32k_out": 2048,
```

```
    "run_pbbs_isort_random_64k_int_out": 2048,
}


BENCHMARK_NAMES = {
    "run_pbbs_bfs_graph_3200_out": "BFS (graph_3200)",
    "run_pbbs_knn_knn_16k_out": "KNN (knn_16k)",
    "run_pbbs_nbody_nbody_1600_out": "NBody (nbody_1600)",
    "run_pbbs_sa_words_32k_out": "Suffix Array (words_32k)",,
    "run_pbbs_isort_random_64k_int_out": "iSort (random_64k_int)",
}


def parse_three_stats_dumps(stats_file):
    """ Reads 'stats.txt' file which has three dumps (setup, execution, teardown). """
    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):
    """ Extracts key branch prediction metrics from the given lines of text. """
    pattern = re.compile(r'^(\S+)\s+([\d.eE+\-NaN]+)\s+.*')
    results = {}

    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')

    if "system.cpu.branchPred.condPredicted" in results and "system.cpu.branchPred.condIncorrect" in result
        cpred = results["system.cpu.branchPred.condPredicted"]
        cincorr = results["system.cpu.branchPred.condIncorrect"]
        results["bpAccuracy"] = 1.0 - (cincorr / cpred) if cpred > 0 else 0.0

    return results
```

```
def main():
    data = {}

    for outdir, predictor_size in OUTDIRS.items():
        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:
            continue

        execution_lines = regions[1]
        metrics = extract_key_metrics_from_lines(execution_lines)
        data[outdir] = metrics
        data[outdir]["predictorSize"] = predictor_size


    sorted_data = sorted(data.items(), key=lambda x: (BENCHMARK_NAMES[x[0]], x[1]["predictorSize"]))

    print("\nBranch Prediction Accuracy Results for Different ‘localPredictorSize‘ Values:\n")
    print(f"{’Benchmark’:<35} {’Predictor Size’:<15} {’Cond. Predicted’:<20} {’Cond. Incorrect’:<20} {’Accu
    print("=" * 120)

    last_benchmark = None


    for outdir, metrics in sorted_data:
        name = BENCHMARK_NAMES.get(outdir, outdir)
        predictor_size = metrics.get("predictorSize", "Unknown")

        cond_pred = int(metrics.get("system.cpu.branchPred.condPredicted", 0))
        cond_incorr = int(metrics.get("system.cpu.branchPred.condIncorrect", 0))
        accuracy = metrics.get("bpAccuracy", 0) * 100


        if last_benchmark and last_benchmark != name:
            print("-" * 120)

        print(f"{name:<35} {predictor_size:<15} {cond_pred:<20} {cond_incorr:<20} {accuracy:.2f}%")

        last_benchmark = name

    print("=" * 120)


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

Appendix C: Appendix 1 Data

Name	# Cycles	IPC	L1D Acc	D Miss	L1I Acc	I Miss
BFS-none-none-0	76746377	1.8219	58891837	0.0966	23067591	0.0242
BFS-stride-l1d-2	76640243	1.7776	57371702	0.1012	22466446	0.0277
BFS-stride-l1i-2	76746377	1.8219	58891837	0.0966	23067591	0.0242
BFS-tagged-l1d-2	76848380	1.6828	54639556	0.1149	21427987	0.0303
BFS-tagged-l1i-2	76761510	1.7892	57997086	0.0893	22545169	0.0541
neighbors-none-none-0	77275923	1.7551	49054263	0.0952	21138859	0.0462
neighbors-stride-l1d-2	77239861	1.7678	49361970	0.0924	21284829	0.0462
neighbors-stride-l1i-2	77275923	1.7551	49054263	0.0952	21138859	0.0462
neighbors-tagged-l1d-2	77567455	1.7056	47974978	0.1016	20730032	0.0455
neighbors-tagged-l1i-2	77248466	1.8037	50184848	0.0957	21815734	0.0483
isort-none-none-0	76810811	1.5309	46351252	0.0739	1716757	0.1868
isort-stride-l1d-2	76711810	1.6165	48889002	0.0538	1798056	0.1875
isort-stride-l1i-2	76810811	1.5309	46351252	0.0739	1716757	0.1868
isort-tagged-l1d-2	76909406	1.6094	48892907	0.0555	1789791	0.1882
isort-tagged-l1i-2	76821227	1.5454	46773473	0.0746	1773207	0.1556

TABLE I
Basic Performance Metrics

Name	D Issued	D Unused	D Useful	D Acc	D Cov	I Issued	I Unused	I Useful	I Acc	I Cov
BFS-none-none-0	0	0	0	0	0	0	0	0	0	0
BFS-stride-l1d-2	675871	116293	4502	0.0067	0.0015	0	0	0	0	0
BFS-stride-l1i-2	0	0	0	0	0	0	0	0	nan	0
BFS-tagged-l1d-2	10383615	2824639	688463	0.0663	0.1983	0	0	0	0	0
BFS-tagged-l1i-2	0	0	0	0	0	2267000	880072	48661	0.0215	0.0571
neighbors-none-none-0	0	0	0	0	0	0	0	0	0	0
neighbors-stride-l1d-2	593783	64139	90036	0.1516	0.0411	0	0	0	0	0
neighbors-stride-l1i-2	0	0	0	0	0	0	0	0	nan	0
neighbors-tagged-l1d-2	7197879	2433541	712452	0.0990	0.2553	0	0	0	0	0
neighbors-tagged-l1i-2	0	0	0	0	0	2729026	662435	337367	0.1236	0.3497
isort-none-none-0	0	0	0	0	0	0	0	0	0	0
isort-stride-l1d-2	2622921	187649	884873	0.3374	0.4429	0	0	0	0	0
isort-stride-l1i-2	0	0	0	0	0	0	0	0	nan	0
isort-tagged-l1d-2	5940509	729396	874479	0.1472	0.4111	0	0	0	0	0
isort-tagged-l1i-2	0	0	0	0	0	874688	214687	162226	0.1855	0.4889

TABLE II
Prefetcher Performance Metrics