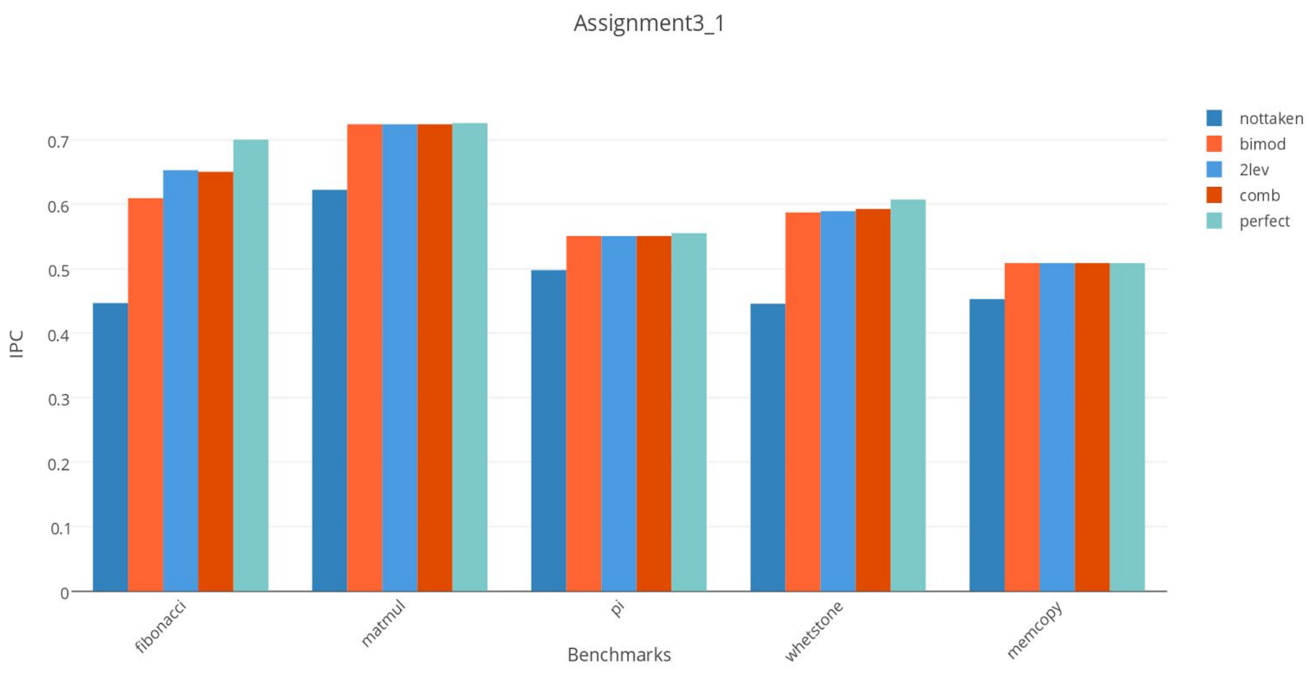
Assignment-3

Rajendra Bogati (376628), Dhananjaya Kittur(376629), Niroj Pokhrel(376630)

1. Task 1 (Branch Prediction Algorithm)

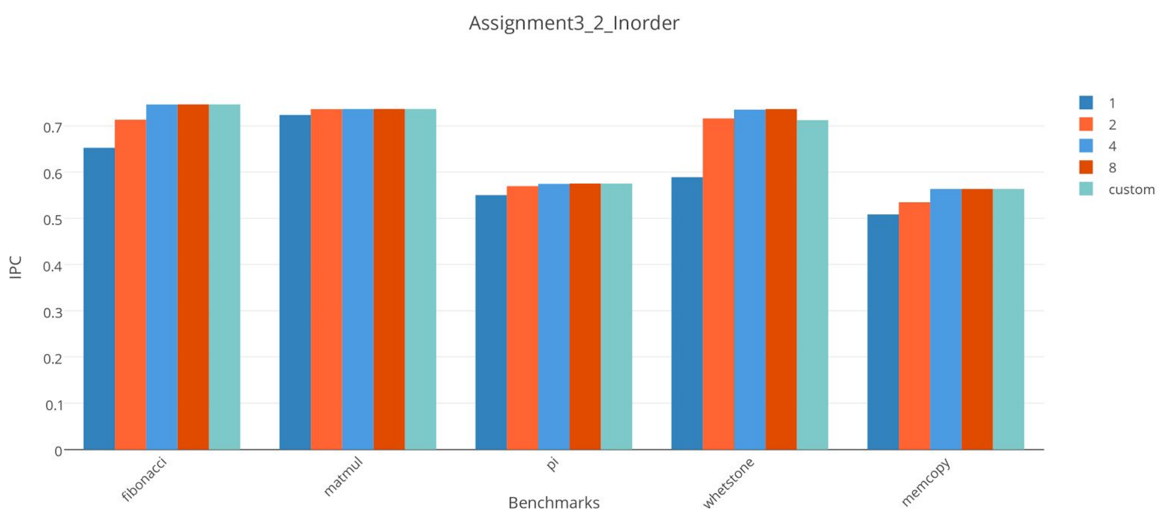
Bar graph for the benchmarks measuring the performance of branch prediction algorithms nottaken, bimod, 2lev, comb and perfect.



2. Task2 (Increased width)

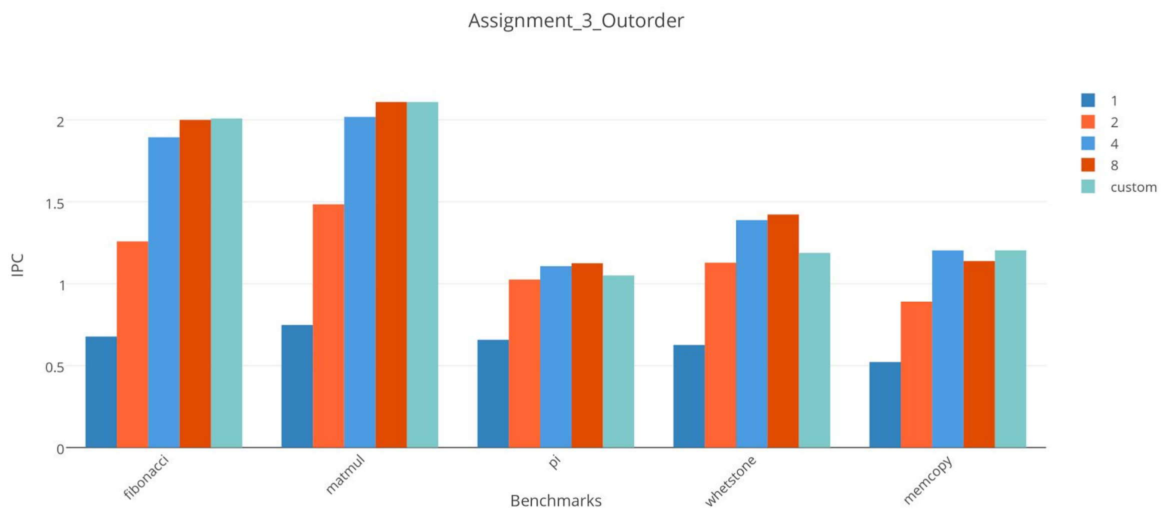
A. Inorder

The bar graph for increasing the size of fetch, decode, issue, commit and resources to the value of 1, 2, 4, 8 and custom. In the custom, processor width is 8 but resources is default. The order of execution is inorder.



B. Outorder

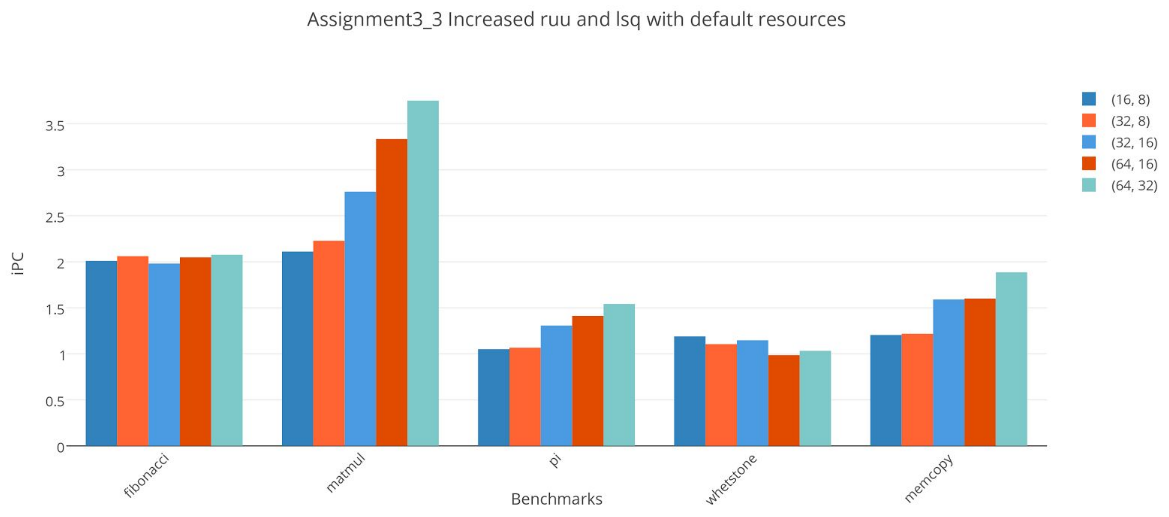
The bar graph for increasing the size of fetch, decode, issue, commit and resources to the value of 1, 2, 4, 8 and custom. In the custom, processor width is 8 but resources is default. The order of execution is outorder.



3. Increased size of ruu and lsq

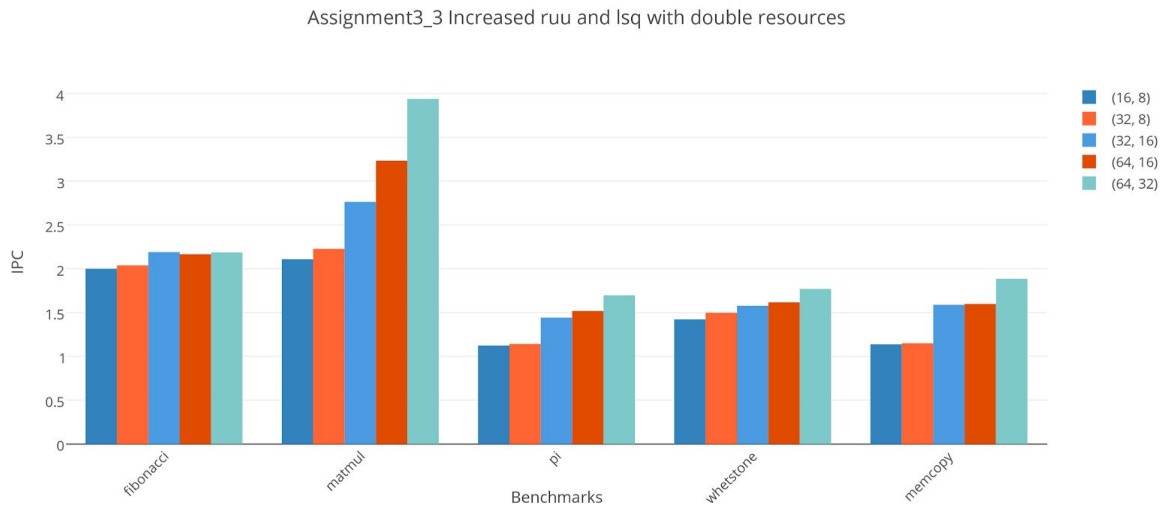
a. Default amount of execution resources

The effect of increasing the register update unit and load store queue for various value of (ruu, lsq). Below graph is for the default amount of resources.



b. Double amount of execution resources

The effect of increasing the register update unit and load store queue for various value of (ruu, lsq). Below graph is for the double amount of resources.



4. Description

A. **Branch Prediction Algorithm:** The branch penalty can be hazardous for the pipeline. If the code has a lot of branches with no prediction then it will lower IPC of the code. There are two types of branch prediction algorithms viz static and dynamic.The static branch prediction algorithms uses the static information or use test data to predict the branch. It can be as simple as always predicting to be taken or not taken. The performance of such algorithm is low compared to dynamic algorithm which can be observed in Task 1. The performance of nottaken branch prediction is low in fibonacci, pi, whetstone and memcopy. Since, the size of for loop (50 to be exact) in the case of matmul, nottaken has substantial performance this is because most of the time is spent in the for loop without taking branches. The usage of dynamic branch prediction algorithm has tremendously improved the IPC of the benchmarks. The performance of the algorithms in the benchmarks/code depends on the effectiveness of the algorithms and how many branches do the benchmarks/code have. If the benchmarks/code have a lot of branches then it can substantially improve the performance.

B. **Increased Width with inorder and outorder execution:** The inorder execution doesn’t have much benefit with the increment of fetch, decode, issue, commit and resources width. Instructions are executed in the compiler generated order. So, if one stalls all stalls. Instructions are statically scheduled. However, for the outorder execution instructions can be executed in any order which are dynamically executed. This is the reason why IPC of out of order execution is substantially higher than the IPC of in order execution. Comparing the IPC of benchmarks in Task2 for inorder and outorder, we can see that IPC of benchmarks for outorder is much higher than inorder. Similarly, there are two type of benchmarks: memory intensive and computation intensive. The computation intensive can leverage the larger issue and number of resources. However, it doesn’t make much difference for the algorithm which are memory intensive. The benchmarks like fibonacci, matmul which has a lot of multiplications and additions have enjoyed the increment in issue width the most. The most of the multiplications and additions in the above algorithms are independent and can be executed in parallel as well. However, the case of pi is totally different where each instruction is dependent in the value calculated in the previous calculations. Because of this despite having a lot of computations (multiply and divide), pi still have low IPC improvements comparatively. The memcopy is memory intensive benchmarks so it also has lower IPC improvements compared to other benchmarks. IPC improvements depends on the computations present in benchmark/code. If there is not much computation present increasing the width doesn't improve the IPC much.

C. **Increasing size of register update unit (ruu) and load store queue (lsq):** The register update unit handles register synchronization and communication. RUU unifies reorder buffers and reservation stations which is done by using circular queue and by allocating at dispatch and deallocating at commit. Similarly, load store queue handles memory synchronization and communication. It contains all load and stores in program order. Both ruu and lsq supports out of order issue. The doubling the amount of resources has slightly improved the performance of the benchmarks with slightly improved IPC. The performance is predominant in matmul whereas the improvement in rest of the benchmarks is little to negligible. Multiply and addition operation in multiplication may have exploited the use of ruu and lsq to improve the performance.

5. Cache Design

Assumptions:

1. 18 initial memory cycle is penalty for initial cache miss. It is not the total number of cycle to access memory but is the penalty for missing the cache.

Total number of cycle to access memory = number of cycle to access cache + miss penalty

Thus, two formula for AMAT can be derived based on whether we are considering penalty for cache miss or total number of cycle to access memory

AMAT = hit time + miss time

F1: **AMAT = number of cycle to access cache + miss rate x miss penalty**

F2: **AMAT = hit rate x number of cycle to access cache + miss rate x number of cycle to access memory**

F1 is the formula discussed in the class.

2. Latency for L2 cache is mentioned 6. The access time for L1 cache is 1.

Total access time of L2 cache = 6 + 1 (1 being the access time for L1 cache)

Since, there is no explicit mention of consideration for this latency, we will be considering 6 as total latency for calculating AMAT.

3. For two level cache design,

AMAT = L1 hit time + L1 miss rate x L1 miss penalty

L1 miss penalty = L2 hit time + L2 miss rate x L2 miss penalty

However, since we are considering only L2 cache for AMAT calculations, we have

AMAT = L2 hit time + L2 miss rate x L2 miss penalty

L2 hit time = 6 cycle

Following is the data for the default configuration of the benchmarks

```
ul2.accesses              869853 # total number of accesses
ul2.hits                721136 # total number of hits
ul2.misses               148717 # total number of misses
ul2.miss_rate             0.1710 # miss rate (i.e., misses/ref)
```

We are required to do the calculation for L2 cache only. We can consider the miss rate of L2 cache.

For the default of 18 cycle:
F1: AMAT for L2 cache = hit time + miss rate x miss penalty
= 6 + 0.1710 x 18
= 9.078
If 18 is total memory access time, it should be calculated as below:
F2: AMAT for L2 cache = hit time + miss rate x miss penalty
= 6 x (1-0.1710) + 0.1710 x 18
= 8.052 cycles

For the initial memory access cycle of 200:
F1: AMAT for L2 cache = hit time + miss rate x miss penalty
= 6 + 0.1710 x 200
= 40.2
If 200 is total memory access time, it should be calculated as below:
F2: AMAT for L2 cache = hit time + miss rate x miss penalty
= 6 x (1-0.1710) + 0.1710 x 200
= 39.174 cycles

Cache Design to mitigate the latency of 200 cycle memory access:
All the codes/benchmarks need to access memory so all of them can benefit and get improvement in IPC with proper memory hierarchy design. The consequence of proper design of cache for the high memory intensity benchmarks like memcopy is very high. The increment in the initial memory access time from 18 cycle to 200 cycle increases the AMAT from 9.078 to 40.2. This creates a substantial reduction in the IPC. IPC for 18 cycle (Keeping other parameters as stated in the problem) is 1.5894. However, for the same config for 200 cycle IPC reduces to 0.5430. The reduction is solely because of increment in the memory latency to 200. However, IPC can be improved by increasing the number of sets, block size and/or associativity. In accordance with the problem set, there is no penalty for increasing the size of block. Thus, increasing the block size increases the IPC with the peak reaching at 8192 with the IPC of 1.9649. The increment of the block size after 8192 results in decreasing IPC. After observing several simulations, increasing the number of sets in the cache did not help much to mitigate the extra latency we are adding. Thus, IPC did not improve rather was decreased with increment of number of sets. Following are the result from the simulation:

```
ul2:1024:64:4:l   latency =  6  : IPC =  0.5430
ul2:2048:64:4:l   latency =  7  : IPC =  0.5579
ul2:4096:64:4:l   latency =  8  : IPC =  0.5542
```

The increment in the associativity did help in increment of IPC up to a certain level despite the fact we increased the latency. This can be attribute to the functioning of the set-associative cache. In set-associative cache, data might be in one of the different associativity rather than the exactly mapped position like in directly mapped cache. This prevent scenarios where two or more memory collide in the same exact position causing cache hit and miss repeatedly.

```
ul2:1024:64:4:l   latency =  6  : IPC =  0.5430
ul2:1024:64:8:l   latency =  7  : IPC =  0.7243
ul2:1024:64:16:l   latency =  8  : IPC =  0.8463
```

```
ul2:1024:64:32:l   latency =  9  : IPC =  0.8634
ul2:1024:64:64:l   latency =  10  : IPC =  0.8315
ul2:1024:128:4:l   latency =  6  : IPC =  0.8056
ul2:1024:128:8:l   latency =  7  : IPC =  1.0224
ul2:1024:128:16:l   latency =  8  : IPC =  1.1675
ul2:1024:128:32:l   latency =  9  : IPC =  1.1080
```

The best performance however was observed with the increment of block size. Since, there is no penalty in improvement of block size, IPC improved until 8192.

```
ul2:1024:64:4:l   latency =  6  : IPC =  0.5430
ul2:1024:128:4:l   latency =  6  : IPC =  0.8056
ul2:1024:256:4:l   latency =  6  : IPC =  1.0408
…………………………………………………….
ul2:1024:4096:4:l   latency =  6  : IPC =  1.9520
ul2:1024:4096:8:l   latency =  7  : IPC =  1.7857
ul2:1024:8192:4:l   latency =  6  : IPC =  1.9649
ul2:1024:8192:8:l   latency =  7  : IPC =  1.7965
ul2:1024:16384:4:l   latency =  6  : IPC =  1.8630
```

Thus, it is observed that maximum IPC was observed for block size of 8192.
IPC = 1.9649. However, block size of 8192 is too big which will increase the cost of cache too much. So, for our design we will like to consider the cache configuration of **ul2:1024:1024:4:l** which will give the IPC of 1.7455.

So, the performance of L2 cache for the above configuration of **ul2:1024:1024:4:l** cache is:

```
ul2.accesses              869861 # total number of accesses
ul2.hits               864023 # total number of hits
ul2.misses                 5838 # total number of misses
ul2.miss_rate             0.0067 # miss rate (i.e., misses/ref)
```

Comparing the miss rate with the default configuration, it has decreased from 0.1710 to 0.0067. Because of this reduction in the miss rate, it is less penalized for the higher memory latency and hence its performance has improved.

AMAT = hit time + miss penalty x miss rate
= 6 + 0.0067 x 200
= 7.34

Thus, AMAT has decreased from 40.2 for ul2:1024:64:4:l to 7.34 for ul2:1024:1024:4:l. This configuration can mitigate the problem of increased memory latency.