COMPUTER ARCHITECTURE

Project #2

Studying effect of varying Cache Configurations on CPU Performance
&
Implementing Cost Function to deduce an Optimal Configuration

Akash Biswal (axb200166)
Mayank Kumar Singhal (mks200001)

Part1 : What is a Cache?

- In computing, a cache is a component that stores data so that future requests for that data can be served faster.
- Caches employ SRAM technology, which is faster, expensive but, has lower bit density than DRAM
- **Caches ($) - Terminology**
    - **Hit** : data appears in some block in the $
    - **Hit Rate**: the fraction of memory access found in the $
    - **Hit Time**: Time to find data in $
    - **Miss rate** : 1-Hit Rate
    - **Miss penalty** : Time to replace a block in the $ time to deliver the block to the processor
    - **Average Memory access time (AMAT)**
        - **AMAT**= Hit time + (Miss rate x Miss penalty)
- Miss rate affects the overall CPI(cycles per instruction) of the CPU, which means decrease in performance.

Part1: Our Setup

- We used *UTD server*, one *local Ubuntu-VM* and one *remote Ubuntu machine*(in India), with rdp access.
- UTD server comes with all the dependencies pre-installed (we copied Gem5 to run on a local directory)
- We installed dependencies and and build Gem5 on other two machines.
- As total configurations to simulate were 288, an automation script was written in python to automate the whole process.
- Automation script was divided in 4 parts with minor tweaks to simultaneously run on four different machines together.

Part1: Different cache configurations used

Benchmark	L1d	L1i	L2	L1a	L2a	CBS
458.sjeng	{64,128}	{64,128}	{256, 512, 1024}	{1, 2, 4}	{1, 2}	{32, 64}
470.lbm	{64,128}	{64,128}	{256, 512, 1024}	{1, 2, 4}	{1, 2}	{32, 64}

Part1: Automation

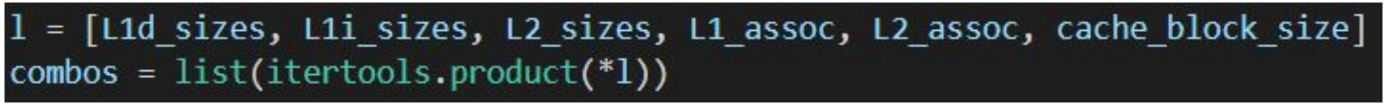
Automation algorithm:

- Libraries used: os, subprocess, itertools
- Rebuild the gem5 environment by running “scons build/X86/gem5.opt” without adding Branch Predictors
- Edit the “runGem5.sh” in the particular Benchmark’s folder by adding the desired output location and necessary values for all cache configurations
- Run “sh runGem5.sh” for each iteration using the subprocess library
- The used scripts have been attached

Part1: Automation

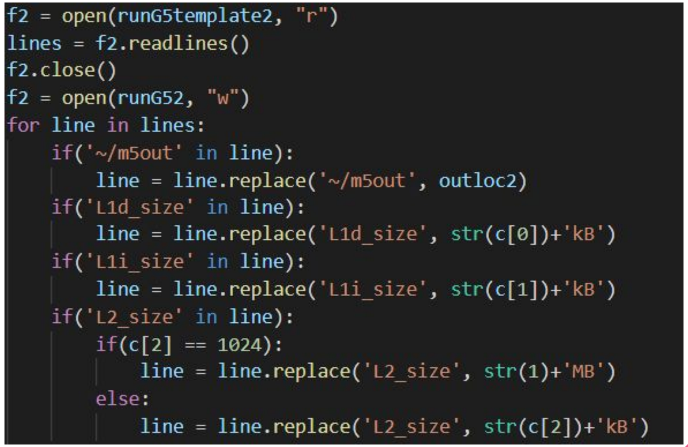
Code Snippets:

- generating all the possible cache configurations:



```
l = [L1d_sizes, L1i_sizes, L2_sizes, L1_assoc, L2_assoc, cache_block_size]
combos = list(itertools.product(*l))
```

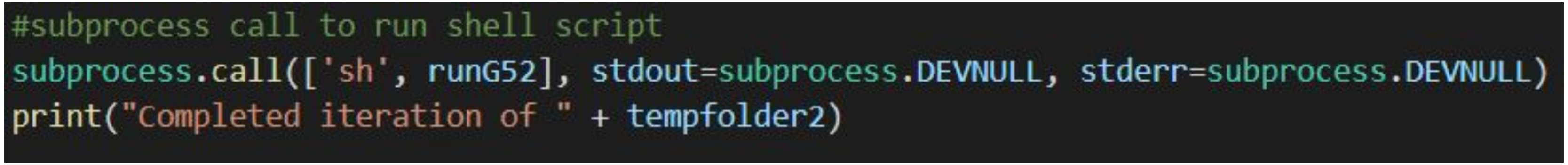
- Editing the runGem5.sh file with necessary parameters



```
f2 = open(runG5template2, "r")
lines = f2.readlines()
f2.close()
f2 = open(runG52, "w")
for line in lines:
    if('~/m5out' in line):
        line = line.replace('~/m5out', outloc2)
    if('L1d_size' in line):
        line = line.replace('L1d_size', str(c[0])+'kB')
    if('L1i_size' in line):
        line = line.replace('L1i_size', str(c[1])+'kB')
    if('L2_size' in line):
        if(c[2] == 1024):
            line = line.replace('L2_size', str(1)+'MB')
        else:
            line = line.replace('L2_size', str(c[2])+'kB')
```

Part1: Automation

- Running the edited shell script



```
#subprocess call to run shell script
subprocess.call(['sh', runG52], stdout=subprocess.DEVNULL, stderr=subprocess.DEVNULL)
print("Completed iteration of " + tempfolder2)
```

Part2: Finding CPI from equation

CPI Equation as per Deliverables:

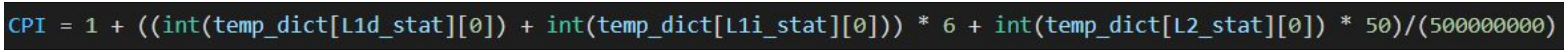
$$CPI = 1 + \frac{(IL1.miss\_num + DL1.miss\_num) \times 6 + L2.miss\_num \times 50}{Total\_Inst\_num}$$

- Wrote a python script *(cpi.py)* for
  - Created a dictionary with all stats.txt files generated.
  - Extracting miss numbers for L1i, L1d and L2 from all output *“stats.txt”* files.
  - Computed the CPI for each output using above formula.
  - Exporting all the values to *“All_Stats.xlsx”*.

Part2: Finding CPI from equation

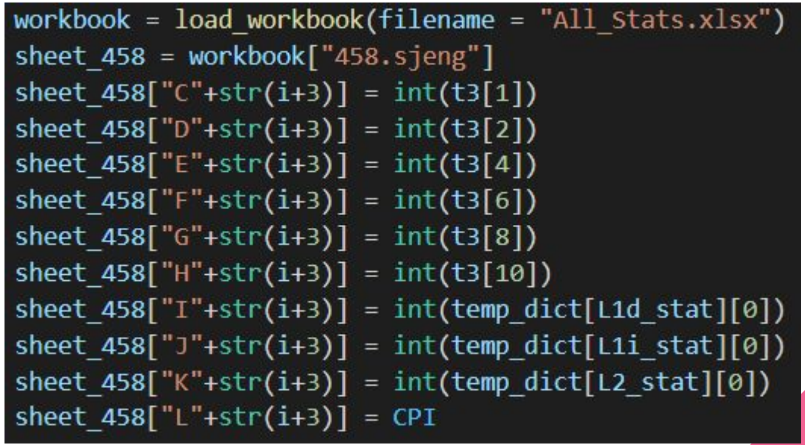
Code Snippets(cpi.py):

- Formula to calculate CPI:



```
CPI = 1 + ((int(temp_dict[L1d_stat][0]) + int(temp_dict[L1i_stat][0])) * 6 + int(temp_dict[L2_stat][0]) * 50)/(500000000)
```

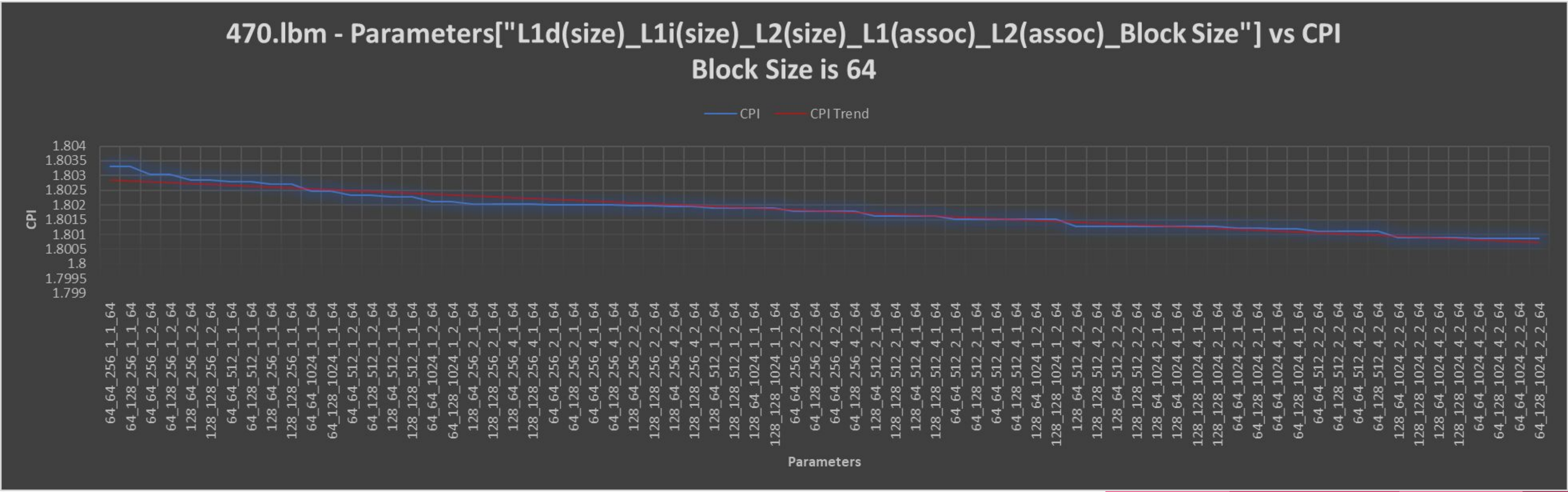
- Write the extracted data to excel sheets



```
workbook = load_workbook(filename = "All_Stats.xlsx")
sheet_458 = workbook["458.sjeng"]
sheet_458["C"+str(i+3)] = int(t3[1])
sheet_458["D"+str(i+3)] = int(t3[2])
sheet_458["E"+str(i+3)] = int(t3[4])
sheet_458["F"+str(i+3)] = int(t3[6])
sheet_458["G"+str(i+3)] = int(t3[8])
sheet_458["H"+str(i+3)] = int(t3[10])
sheet_458["I"+str(i+3)] = int(temp_dict[L1d_stat][0])
sheet_458["J"+str(i+3)] = int(temp_dict[L1i_stat][0])
sheet_458["K"+str(i+3)] = int(temp_dict[L2_stat][0])
sheet_458["L"+str(i+3)] = CPI
```

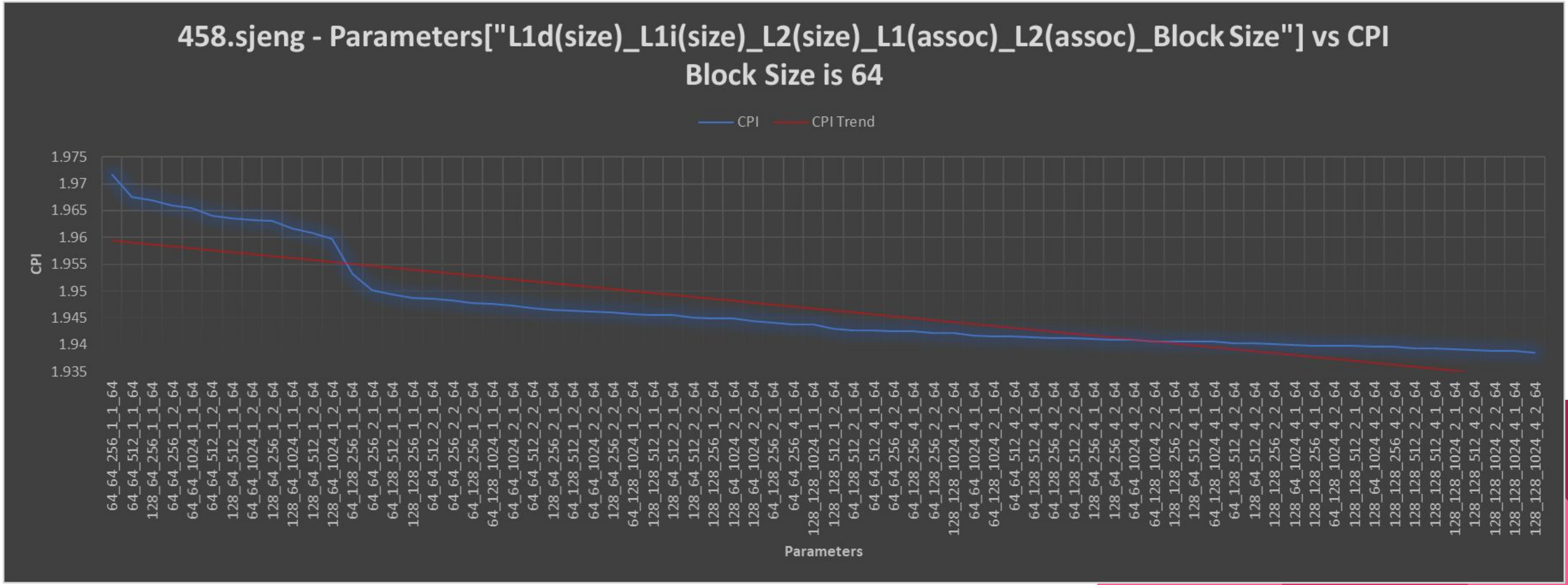
Part3: Optimal cache configuration to achieve lowest CPI

CPI vs cache configurations for 470.lbm:



Part3: Optimal cache configuration to achieve lowest CPI

CPI vs cache configurations for 458.sjeng:



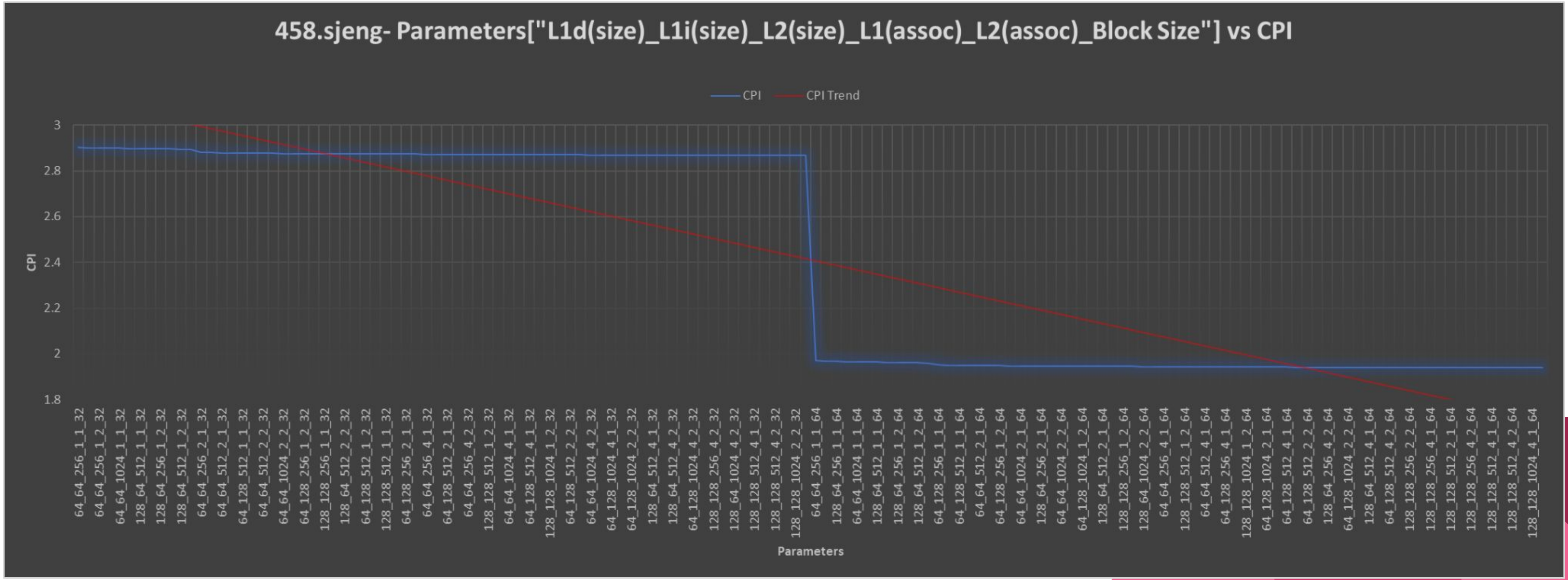
Part3: Optimal cache configuration to achieve lowest CPI

Observations:

- In the graphs seen for the both the benchmarks it is clear that CPI reduces i.e. there is an increase in performance with increase in L1 and L2 cache sizes as per the trend line
- On close observations it is seen that there are dips in the “CPI” when associativity is increased for L1, hinting that increased associativity means increased performance
- On increasing the L1 instruction cache size a significant change in CPI is seen for both benchmarks
- Differing L2 sizes does not change performance adversely

Part3: Optimal cache configuration to achieve lowest CPI

CPI vs cache configurations for 458.sjeng:(comparing block size of 32 and 64)



Part3: Optimal cache configuration to achieve lowest CPI

Final Inferences:

- When the cache block size was changed from 32 to 64 a massive change in CPI was noticed. The CPI reduced by roughly 1 clock cycle per instruction
- This was seen in the graph for 470.lbm as well
- **For 458.sjeng, it is clearly seen that when L1.i and L1.d are both 128kB and L2 ranges from 256kB to 1MB, irrespective of associativity, CPI tends to be low. Therefore these group of configurations are the most optimal.**
- **For 470.lbm CPI is lowest when L1.i and L1.d are either 64kB, 128kB or same, when L2 is 1MB and associativity is greater than 1(in all cases).**
- **This means overall, an optimal configuration would be L1.i, L1.d = 128kB and L2 = (512kB, 1MB), associativity at all levels should be higher than 1 and cache block size is 64.**

Part4: Cost Function

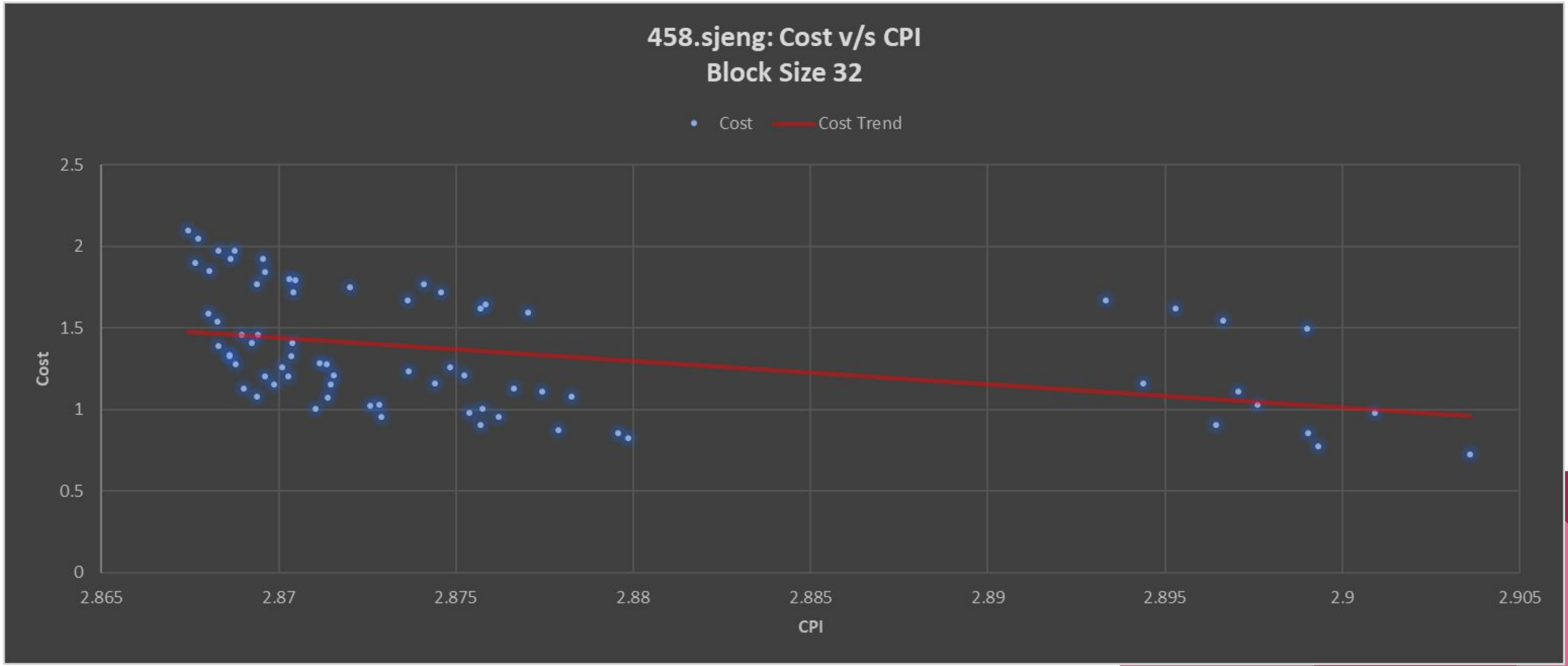
- According to the observation and our intuition, cost function should
  - Directly depend on,
    - L1 cache size (L1i + L1d)
    - L1 associativity
    - L2 cache size
    - L2 associativity
  - And Indirectly depend on,
    - Cache Block Size(CBS)
- Therefore, we defined a cost function as follows

$$\boxed{Cost = (L1.i + L1.d) \times 0.002 + L2 \times 0.001 + L1.a \times 0.1 + L2a \times 0.05 + \frac{2}{CBS}}$$

- L1 is more costly than L2 cache.
- Constants were chosen on the basis of priority to fit graphical scale

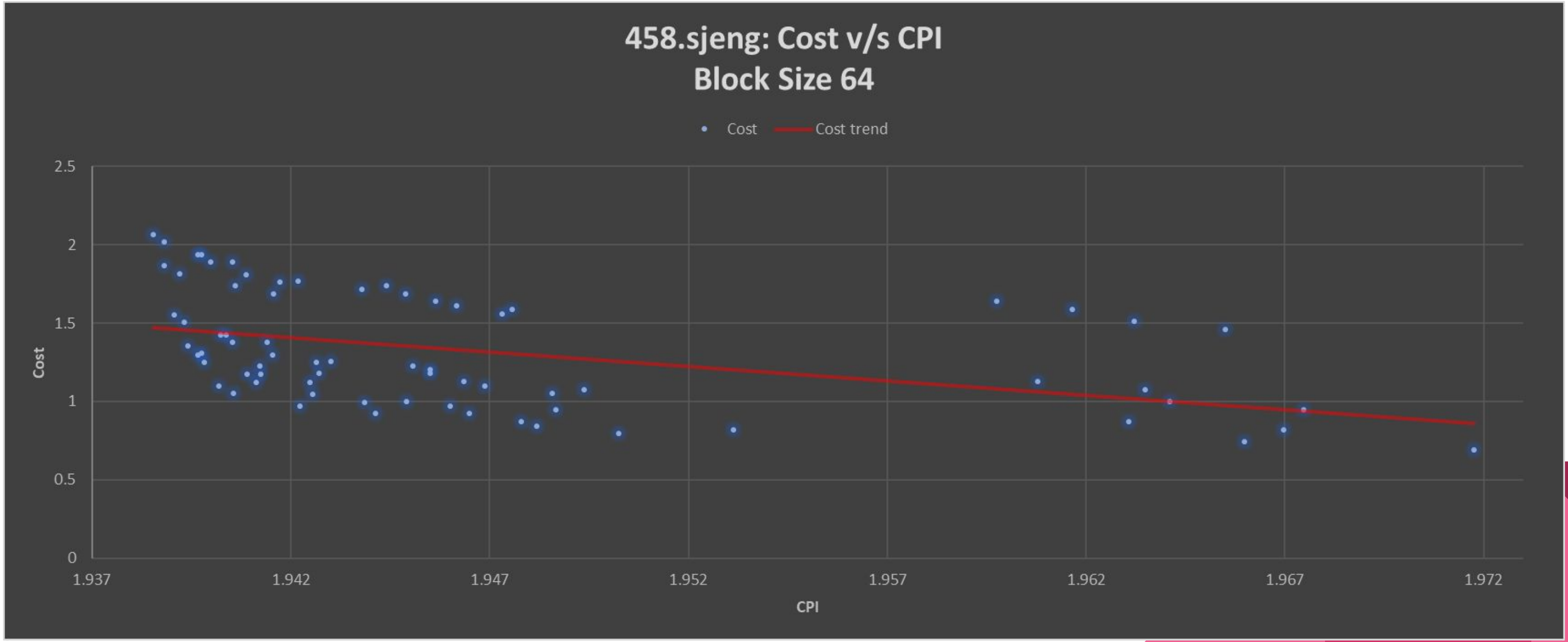
Part5: Optimizing cache for performance v/s cost

Cost v/s CPI for 458.sjeng (block size 32):



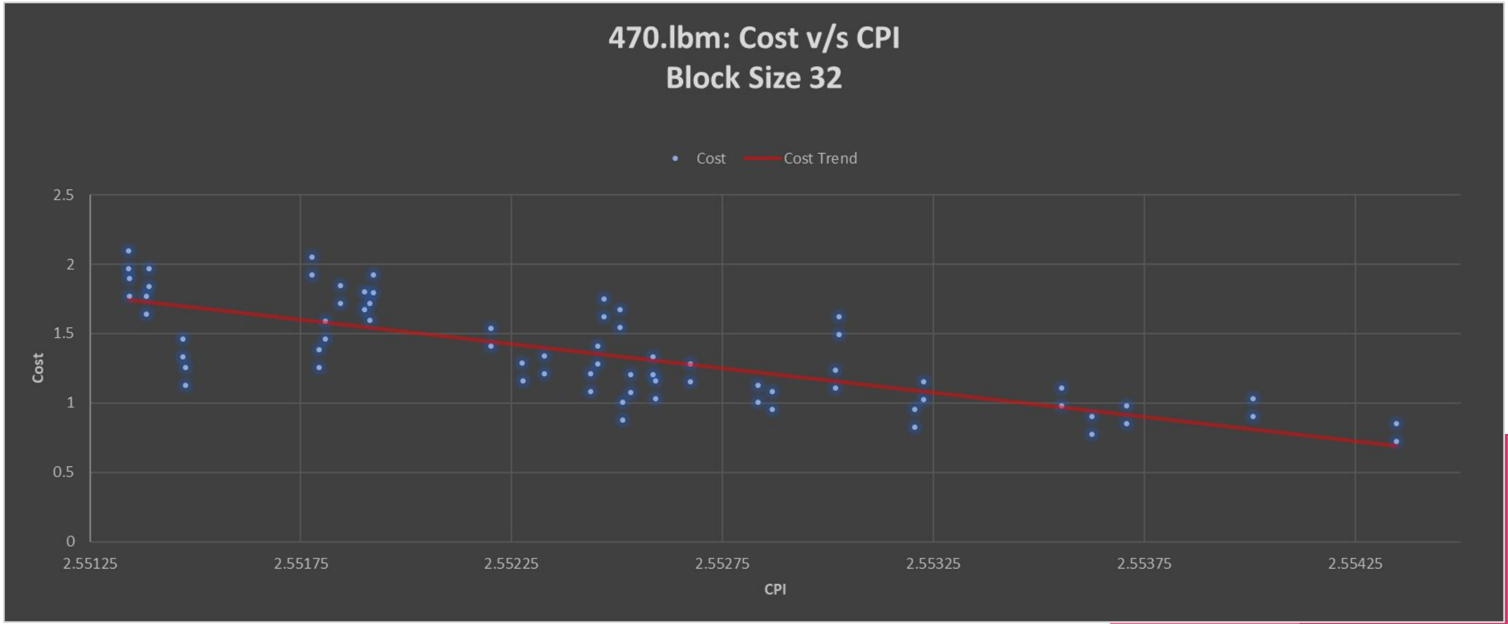
Part5: Optimizing cache for performance v/s cost

Cost v/s CPI for 458.sjeng (block size 64):



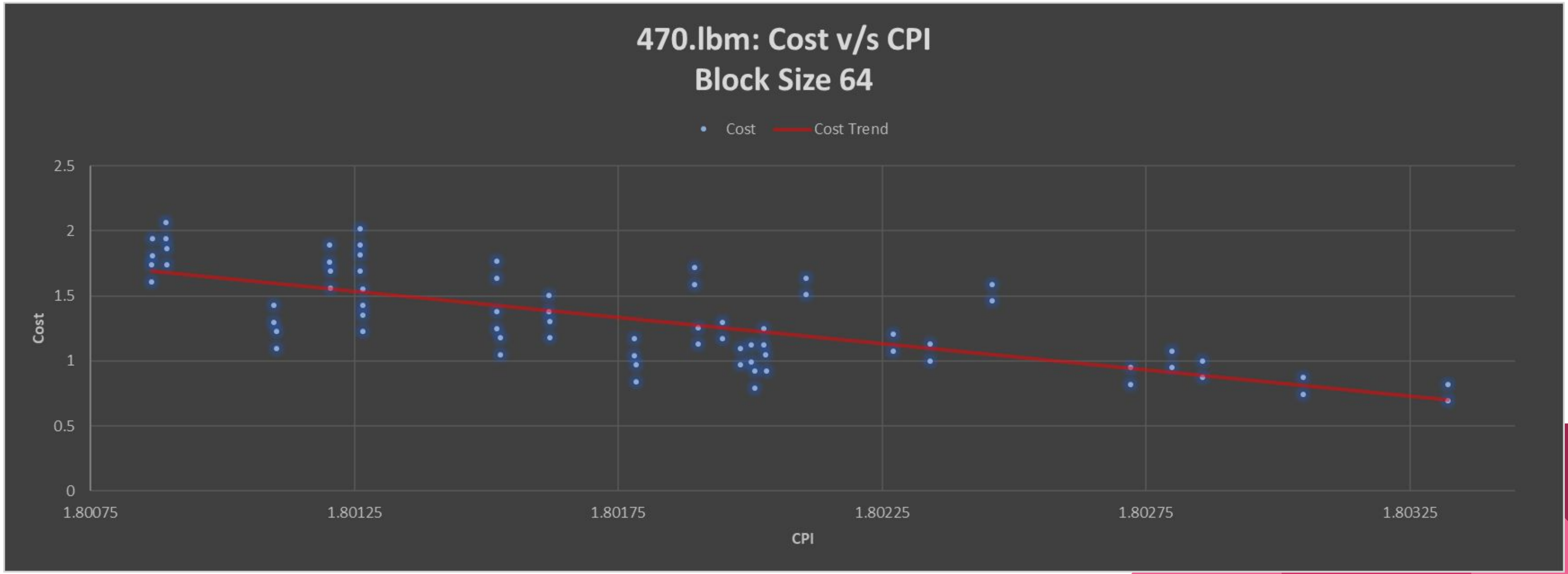
Part5: Optimizing cache for performance v/s cost

Cost v/s CPI for 470.lbm (block size 32):



Part5: Optimizing cache for performance v/s cost

Cost v/s CPI for 470.lbm (block size 64):



Part5: Optimizing cache for performance v/s cost

- The main observation is lower CPI means higher Cost, this is true in the real world as well as better performing CPUs come with a cost
- We can clearly infer from the graphs above that, at the same cost range we can see lower CPI values in configuration with cache block size as 64 Therefore, this eliminates using cache block size of 32, as it is not cost optimal as well as performance optimal.
- If observed closely, for both benchmarks, the cost trendline is placed slightly lower in case of cache block size 64 than it is for cache block size of 32. This means that at a lower baseline cost we get a higher performing CPU when cache block size is 64

Part5: Optimizing cache for performance v/s cost

Final Conclusions:

- From the graphs, we see that for 458.sjeng the cost optimal cache configuration would be in the CPI range of (1.94,1.95) and its Cost range should be (0.5,1) units.
- In 470.lbm the cost optimal cache configuration ranges are CPI=(1.8,1.802) and Cost range=(0.5,1)