CE 6304: Computer Architecture

Group No:17

Project #1
Optimization of Cache

Submitted by

Kalyan Sunder Manivannan - kxm162730
(Contribution:50%)

Jayaramaraja Balaramaraja - jxb162030
(Contribution:50%)

Contents

List of Figures

Introduction:

This project focusses on the Optimization of the Cache hierarchy for the given X86 architecture CPU on the gem5 simulator. The Problem Statement of the Project is given below.

Problem Statement:

The goal of this project is to Optimize the cache hierarchy on X86 architecture for timing based CPU type on gem5 simulator. Various parameters such as L1cache data size, L1cache instruction size, L2 cache size, associativity and block size are varied to get the minimum CPI for 6 individual benchmarks, namely 401.bzip2, 429.mcf, 456.hmmer, 458.sjeng, 470.lbm, scimark. At the final stage of the project, a cost function is defined and an optimum configuration is obtained by plotting the CPI against the cost function for each cache configuration.

The project is simplified with step by step procedure.

PART 1: SET-UP & GENERATE VALID CACHE CONFIGURATIONS

Here, we generate the valid cache configurations using the python script we developed which loops over various parameters such as **L1cache data size, L1cache instruction size, L2 cache size, L1 & L2 associativity and block size** without worrying about performance of each configuration with 500000000 instructions.

This also involves the initial set up that is needed to run the benchmarks for various parameters. This includes downloading gem5 with corresponding benchmarks and installing them.
In addition to the given benchmarks in the problem statement, an extra benchmark - **'scimark',** which is a floating point benchmark is also analyzed to get better insight of CPU performance.

CONFIGURATION:

- CPU Models: TimingSimpleCPU (timing)
- Cache levels: Two levels
- Separate L1 data and L1 instruction caches with same associativity, unified L2 cache.
- Total Combinations:

With ($ sizes in kB, line size in B)
L1d cache : 1,2,4,8,16,32,64,128
L1i cache : 1,2,4,8,16,32,64,128
L1 Associativity: 1,2,4,8,16
L2 size : 1,2,4,8,16,32,64,128,256,512,1024
L2 Associativity: 1,2,4,8,16
Line size : 16,32,64,128,256,512

Generating Valid Combinations:

Albeit a lot of combinations are possible from the above set of cache configuration, only a few are valid. The total configurations are filtered out with the following conditions: -

- L2 size should be greater than or equal to the total L1size
- Number of Sets (l1i, l1d & l2) should be of the order of 2^n

These conditions generate the valid combinations for the cache configuration

Script in python to generate valid combinations for each benchmark:

```
(********Total Cache configuration combinations initialized*******)
l1size    = [1,2,4,8,16,32,64,128]
l1assoc   = [1,2,4,8,16]
l2size     = [1,2,4,8,16,32,64,128,256,512,1024]
l2assoc    = [1,2,4,8,16]
linesize   = [16,32,64,128,256,512]
#Filter loop starts
for linesize_var in linesize:
   for l1isize_var in l1size:
      for l1dsize_var in l1size:
         for l1assoc_var in l1assoc:
            for l2size_var in l2size:
               for l2assoc_var in l2assoc:
                  if( (l2size_var >= l1isize_var) & (l2size_var >= l1dsize_var) & (l2size_var >=
(l1isize_var+l1dsize_var)) ):
                     l1i_num_lines = int(l1isize_var*1024)/int(linesize_var)
                     l1d_num_lines = int(l1dsize_var*1024)/int(linesize_var)
                     l2_num_lines  = int(l2size_var*1024)/int(linesize_var)
                     l1i_num_sets = int(l1i_num_lines)/int(l1assoc_var)
                     l1d_num_sets = int(l1d_num_lines)/int(l1assoc_var)
                     l2_num_sets  = int(l2_num_lines)/int(l2assoc_var)
                     if ( (l1i_num_lines >= l1assoc_var) & (l1d_num_lines >= l1assoc_var) & (l2_num_lines >=
l2assoc_var) ):
                        if(  powTwoBit((int(l1isize_var)*1024)/(int(linesize_var)*int(l1assoc_var))) ):
                           if(  powTwoBit((int(l1dsize_var)*1024)/(int(linesize_var)*int(l1assoc_var)))  ):
                              if(  powTwoBit((int(l2size_var)*1024)/(int(linesize_var)*int(l2assoc_var))) ):
                                 if( ( l1isize_var + l1dsize_var ) <= 256 ):
                                    (  ********  Valid combinations generated  ********  )
```

About **45549 valid combinations** are generated from the above python script.

Configuration Analyzed under each Benchmark:

Due to time constraint, only limited benchmarks are analyzed. Considering the worst case and best possible configuration, A list of few combinations have been submitted for each benchmark.

Benchmark	Cache Configurations Analyzed
Bzip2	:647
Mcf	:250
Lbm	:314
Hmmer	:328
Sjeng	:224
Scimark	:337
Total	:2100 valid combinations

PART 2: DEFINING COST FUNCTION:

In order to obtain the best possible cache configuration, it is better to analyze the Cost of each configuration simultaneously with the CPI.

Cost becomes an important factor while optimizing a cache hierarchy. With an increase in cost, a better CPI can be achieved. Hence, a trade-off between CPI and the cost needs to be achieved in order to fine tune a cache hierarchy. The cost function defined in this section includes parameters that constitute to an increase in the cost. The parameters that have been considered to develop the cost function are :

- L1 & L2 Cache size
- Associativity (# of comparators)
- Miscellaneous Cost (Extra bits stored in cache & mux size based on # of sets)

Dependence on Cache size:

Cache size is considered to be the most important factor in predicting the cost of memory hierarchy. And even a small increment in the cache size increases the overall cost by a great price Now, L1 cache is costlier than compared to L2 cache, because it is faster as compared to L2 cache as it runs at the same clock as of the CPU. Here L1 cache is divided into L1 Data cache and L1 Instruction cache and unified L2 cache.

We have taken an assumption for L1 and L2 cache cost (in units)

L1cache cost factor = 8 units (arbitrary value)

L2cache cost factor = 1 units (8x less than L1 – arbitrarily assumed)

Dependence on Associativity:

We assume that the associativity of the CPU configuration increases the number of comparators that are used for each set in the n-way set associative cache. This in turn raises the cost by a small factor. Cost factor due to Associativity is directly proportional to usage of n-way associative sets with increase in the number of comparators per set.

Miscellaneous Cost:

Despite the cache cost there are other miscellaneous cost associated to support each configuration. This miscellaneous cost is proportional to cost of extra bits stored in cache (32-tag-offset+1valid bits) & considering extra hardware (bigger mux for the n-way set associative cache). Miscellaneous cost is set by scaling the main cache cost by arbitrary values of 250x and 500x.

Script to calculate the Cost function:

```
**********************************************************
*Initial calculations to compute number of lines, number of sets for l1i, l1d & l2 caches.
**********************************************************
l1i_cost_factor = 8.0
l1d_cost_factor = 8.0
l2_cost_factor  = l1i_cost_factor/8.0
l1i_num_lines = (l1isize*1024)/linesize
l1d_num_lines = (l1dsize*1024)/linesize
l2_num_lines  = (l2size*1024)/linesize
l1i_num_sets = l1i_num_lines/l1assoc
l1d_num_sets = l1d_num_lines/l1assoc
l2_num_sets  = l2_num_lines/l2assoc
**********************************************************
*Main cache cost: directly proportional to size and associativity (# of Comparators for set)
**********************************************************
l1i_cost = (l1i_cost_factor*l1isize) + ((l1i_cost_factor/4)*l1assoc)
l1d_cost = (l1d_cost_factor*l1dsize) + ((l1d_cost_factor/4)*l1assoc)
l2_cost  = (l2_cost_factor*l2size) + ((l2_cost_factor/4)*l2assoc)
**********************************************************
*Miscellaneous cost manipulation
**********************************************************
l1i_misc = (l1i_cost_factor*((32+1-(math.log(linesize,2))-
(math.log(l1i_num_sets,2)))*(1/(1024*8))*(l1i_num_lines) ) ) + ( ( (l1i_cost_factor/250) *
l1i_num_sets) if l1assoc!=1 else 0 )
l1d_misc = (l1d_cost_factor*((32+1-(math.log(linesize,2))-
(math.log(l1d_num_sets,2)))*(1/(1024*8))*(l1d_num_lines))) + ( ( (l1d_cost_factor/250) *
l1d_num_sets) if l1assoc!=1 else 0 )
l2_misc  = (l2_cost_factor*((32+1-(math.log(linesize,2))-
(math.log(l2_num_sets,2)))*(1/(1024*8))*(l2_num_lines))) + ( ( (l2_cost_factor/500) * l2_num_sets) if
l2assoc!=1 else 0 )
```

****
*Total Cost
****
misc_cost = l1i_misc + l1d_misc + l2_misc
cache_cost = l1i_cost + l1d_cost + l2_cost
total_cost = cache_cost + misc_cost

The above script provides the total cost for each configurations.

WEIGHTED PRIORITY:

For cache configurations with higher cost CPI value will be lower and vice versa. So in order to select a best cache configuration, a weighted Priority has been added and computed. Hence by adding Priority to obtain minimum CPI, it will be possible to obtain a good cache configuration.

The Total Cost and the CPI values are normalized to an equal range by **using the normalizing function** so that it becomes effective to give weightage to the CPI and total cost. The normalized CPI and cost factor is given weightage according to the priority.

Weighted Cost: 80% CPI dependence +20% cost dependence
cpi_vs_cost = (cpi*0.8) + (cache_cost*0.2)

We believe that performance of the CPU should be given more priority over the cost as it will give a better experience to the user who is running different applications. Thus, we have given a weightage of 80% to the CPI and only 20% to the total cost.

PART 3: ANALYZING OPTIMUM CONFIGURATION FOR EACH BENCHMARK

In this part of the project, different types of Benchmark files are analyzed by evaluating the cycles per instructions (CPI) for all the Six individual benchmarks. **The goal behind this part of the project is to analyze which minimum possible configuration sufficient to run the below six benchmarks (Not worrying about the effective overall performance of the configuration).** The best possible configuration for each benchmark is analyzed below.

SPEC CPU2006 Benchmark

bzip2 (Compression):

401.bzip2 is based on Julian Seward's bzip2 version 1.0.3. This is an integer benchmark. All compression and decompression happens entirely in memory. This is to help isolate the work done to only the CPU and memory subsystem.

401.bzip2's reference workload has six components: two small JPEG images, a program binary, some program source code in a tar file, an HTML file, and a "combined" file, which is representative of an archive that contains both highly compressible and not very compressible files.

Each input set is compressed and decompressed at three different blocking factors ("compression levels"), with the end result of the process being compared to the original data after each decompression step.

From the stats.txt created from each configuration, number of Instruction misses given configuration is obtained. The CPI is calculated from the following formula.

$$CPI = 1 + \frac{\left((L1d_{miss_{inst.}} + L1i_{miss_{inst}})x4\right) + (L2_{miss_{inst}}x80)}{Instruction\ count}$$

The graph is plotted between the CPI and Total Cost for analysis

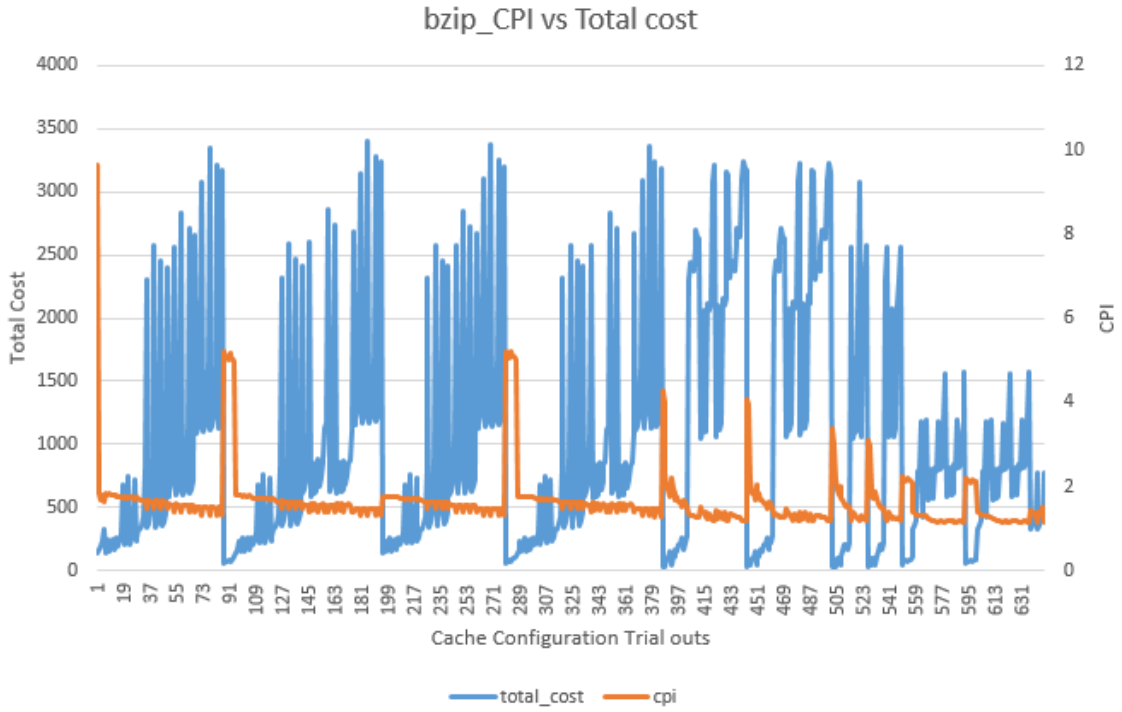


Figure 1: Graph between CPI and the Total Cost for bzip2

From the graph, it is clear that a lot of combination have a decent configuration with low CPI value. Our analysis concentrates more on obtaining a better performance (low CPI) with medium cost.
The CPI and Total cost is normalized , then weighted. Hence adding Weighted Priority to the CPI(80%) and Cost(20%) we get

Weighted Cost: 80% CPI dependence +20% cost dependence

Redrawing the graph by **sorting the CPI value** optimum cache configuration for the bzip2 Benchmark:

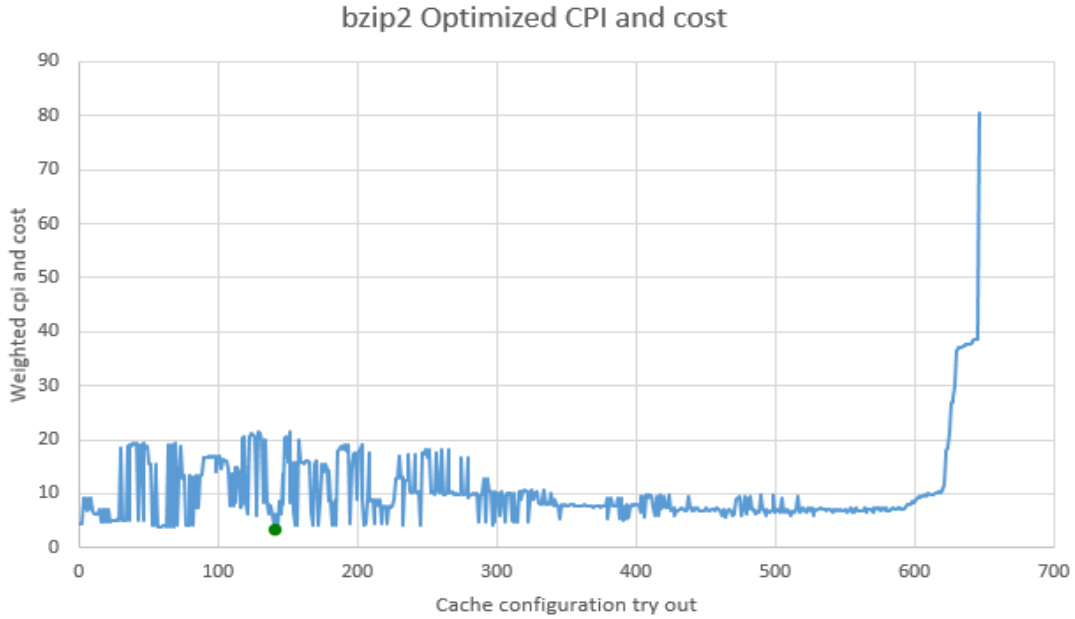


Figure 2: Optimized graph for CPI and cost for bzip2

The green mark denotes the lowest weighted value for this Benchmark.
From the valid set of cache configurations executed, the below configuration was found to produce best result

L1dsize	L1isze	L1_assoc	L2_size	L2_assoc	Line_size	CPI	Total_Cost
16kB	16kB	2	128kB	8	512	1.30646002	395.088(units)

ANALYSIS:

In order to analyze the reason why this configuration got the best result, the nature of computation of bzip2 Benchmark have to be studied. Bzip2 is a Compression Benchmark so the L1d and L1i has to be sufficiently large to keep in the image files in the L1 cache (16kBi and 16kBd). Also, the presence of L2 cache helps store extra information for the benchmark to run smoothly, thus reducing the miss rate (reducing accessed to main memory). The line size is very large which helps in spatial locality, caching higher fraction of program in the cache, also reducing the miss rate.

mcf (Combinatorial optimization)

This falls under the general category of Combinatorial optimization / Single-depot vehicle scheduling
429.mcf is a benchmark which is derived from MCF, a program used for single-depot vehicle scheduling in public mass transportation. The benchmark version uses almost exclusively integer arithmetic.
From the stats.txt created from each configuration, number of Instruction misses given configuration is obtained. The CPI is calculated from the following formula.

$$CPI = 1 + \frac{((L1d_{miss_{inst.}} + L1i_{miss_{inst}})x4) + (L2_{miss_{inst}}x80)}{Instruction\ count}$$

The graph is plotted between the CPI and Total Cost for analysis

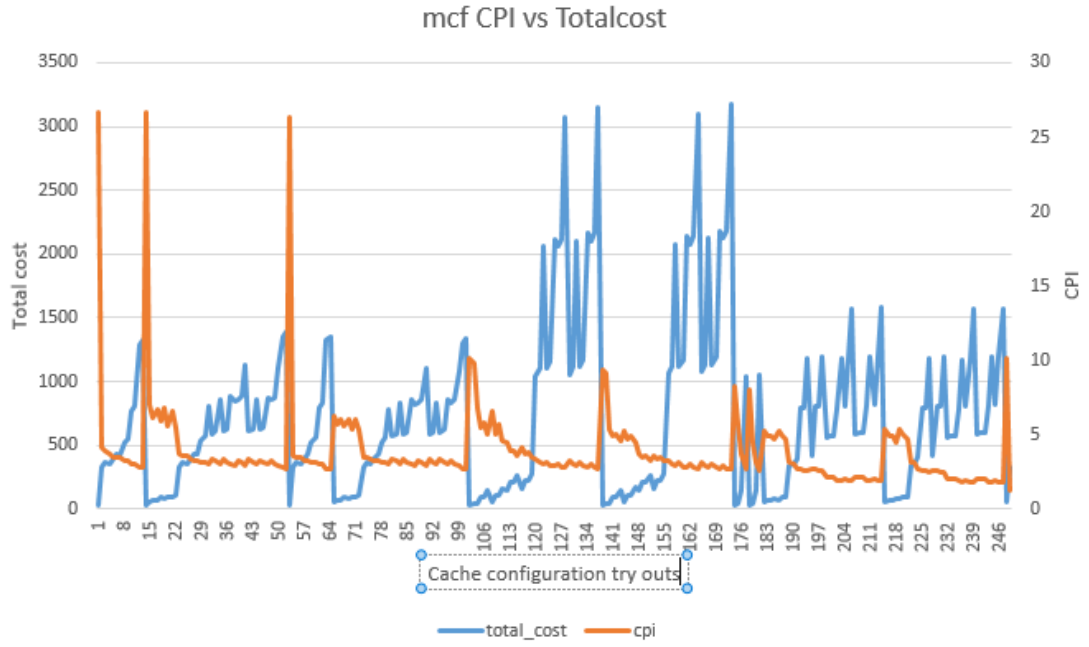


Figure 3: Graph between CPI and the Total Cost for mcf

Redrawing the graph by **sorting the CPI value** optimum cache configuration for the mcf Benchmark:

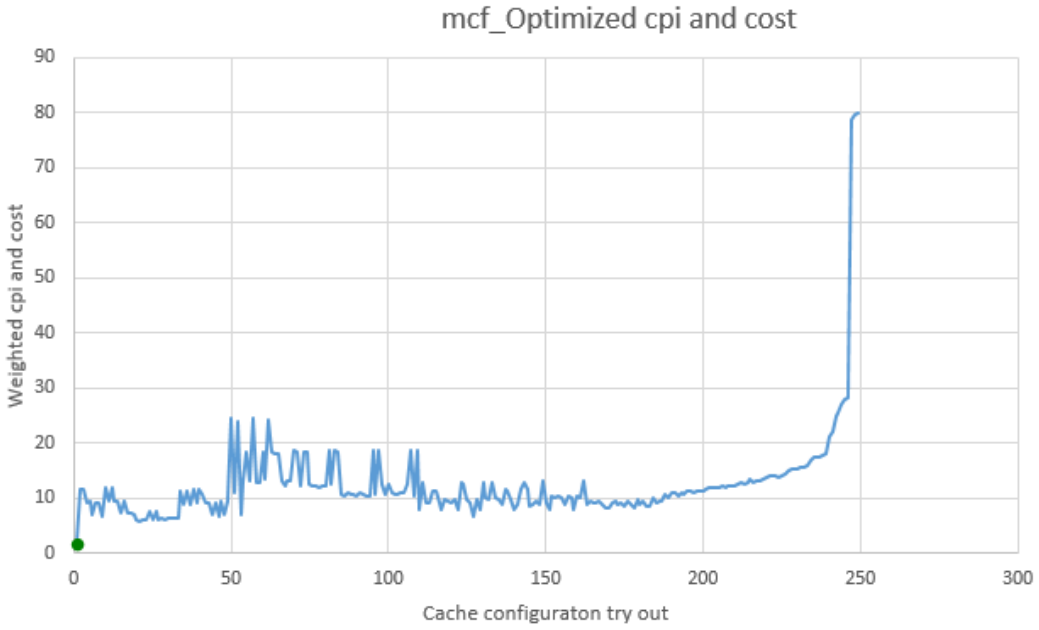


Figure 4: Optimized graph for CPI and cost for mcf

The green mark denotes the lowest weighted value for this Benchmark.
From the valid set of cache configurations executed, the below configuration was found to produce best result

L1dsize	L1isze	L1_assoc	L2_size	L2_assoc	Line_size	CPI	Total_Cost
16kB	16kB	2	64kB	2	512	1.22515376	329.652(units)

ANALYSIS:

mcf requires more memory to analyze huge data. Such large data can be handled only if the L1dcache is sufficiently large. Minimum of L1dsize(16kB) and L2isize(16kB) is required to store the data and program of Single-depot vehicle scheduling model with high cache hit rate. L2 cache(64kB) reduces the memory cycle latency thus increasing the performance.

hmmer (Database Search)

This benchmark uses profile Hidden Markov Models (profile HMMs) or statistical models of multiple sequence alignments, which are used in computational biology to search for patterns in DNA sequences. From the stats.txt created from each configuration, number of Instruction misses given configuration is obtained. The CPI is calculated and the graph is plotted between the CPI and Total Cost for analysis

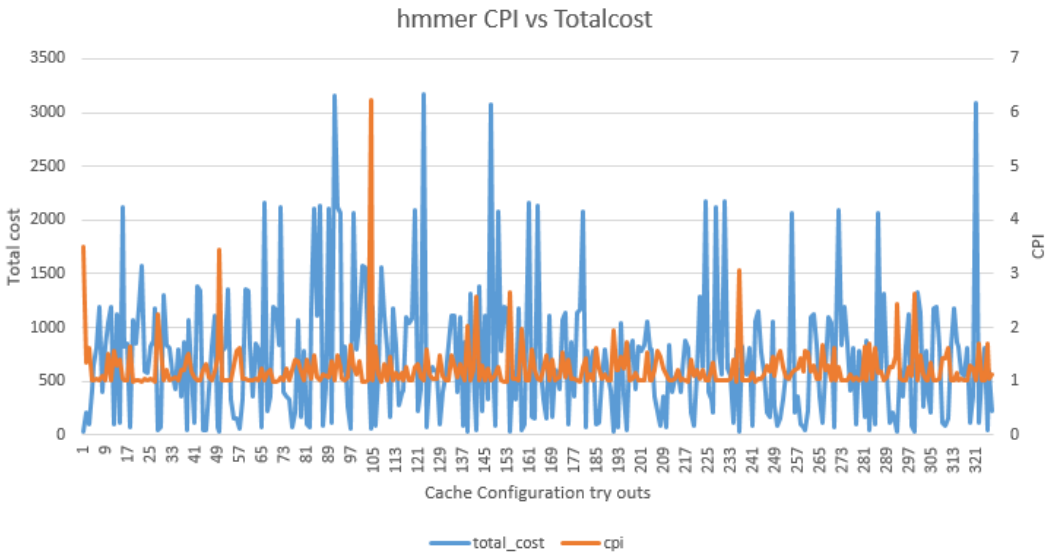


Figure 5: Graph between CPI and the Total Cost for hmmer

Redrawing the graph by **sorting the CPI value** optimum cache configuration for the hmmer Benchmark:

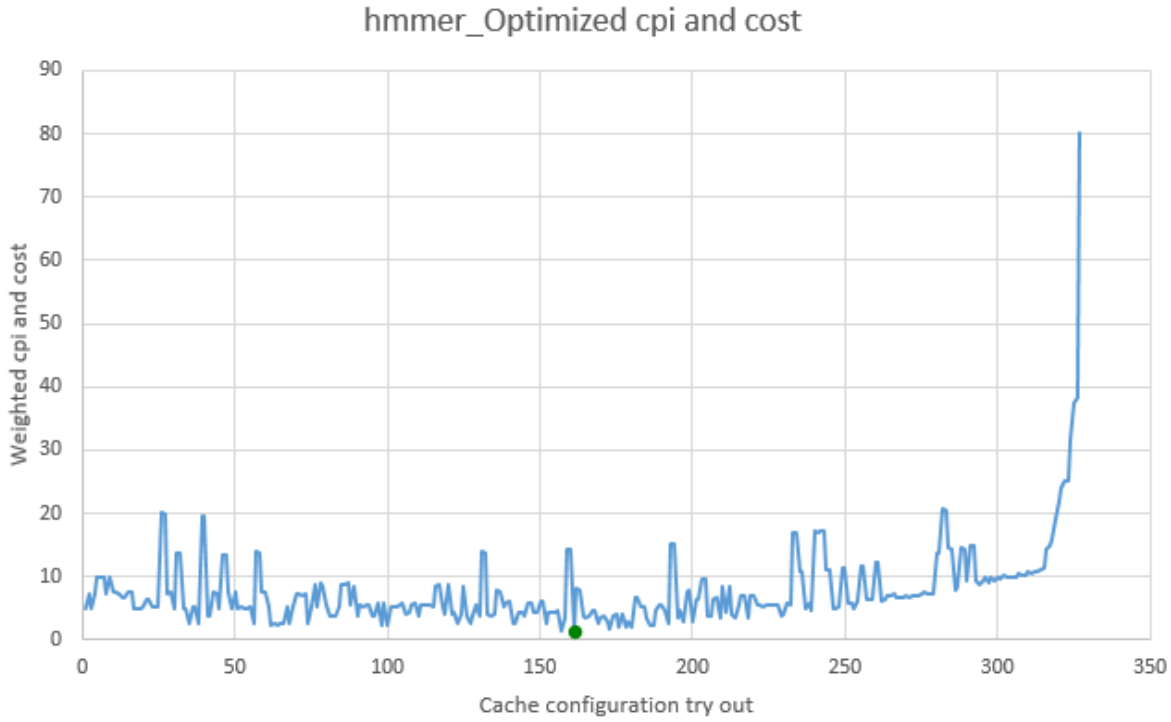


Figure 6: Optimized graph for CPI and cost for mcf

The green mark denotes the lowest weighted value for this Benchmark.

From the valid set of cache configurations executed, the below configuration was found to produce best result

L1dsize	L1isze	L1_assoc	L2_size	L2_assoc	Line_size	CPI	Total_Cost
4kB	4kB	8	8kB	8	512	1.22515376	329.652(units)

ANALYSIS:

From the analysis of the above graph, the combinations with equal L1 and L2 size is found with minimum CPI. Line size of 512B helps much of the program to reside in single cache line, thus reducing cache misses by spatial locality.

sjeng (Artificial Intelligence)

sjeng is based on Sjeng 11.2, which is a program that plays chess and several chess variants, such as drop-chess (similar to Shogi), and 'losing' chess. It attempts to find the best move via a combination of alpha-beta or priority proof number tree searches, advanced move ordering, positional evaluation and heuristic forward pruning. Practically, it will explore the tree of variations resulting from a given position to a given base depth, extending interesting variations but discarding doubtful or irrelevant ones. From this tree the optimal line of play for both players ("principle variation") is determined, as well as a score reflecting the balance of power between the two.

458.sjeng's input consists of a text file containing alternations of

1. a chess position in the standard Forsyth-Edwards Notation (FEN)
2. the depth to which this position should be analyzed, in half-moves (ply depth)

The SPEC reference input consists of 9 positions belonging to various phases of the game.

From the stats.txt created from each configuration, number of Instruction misses given configuration is obtained. The CPI is calculated and the graph is plotted between the CPI and Total Cost for analysis

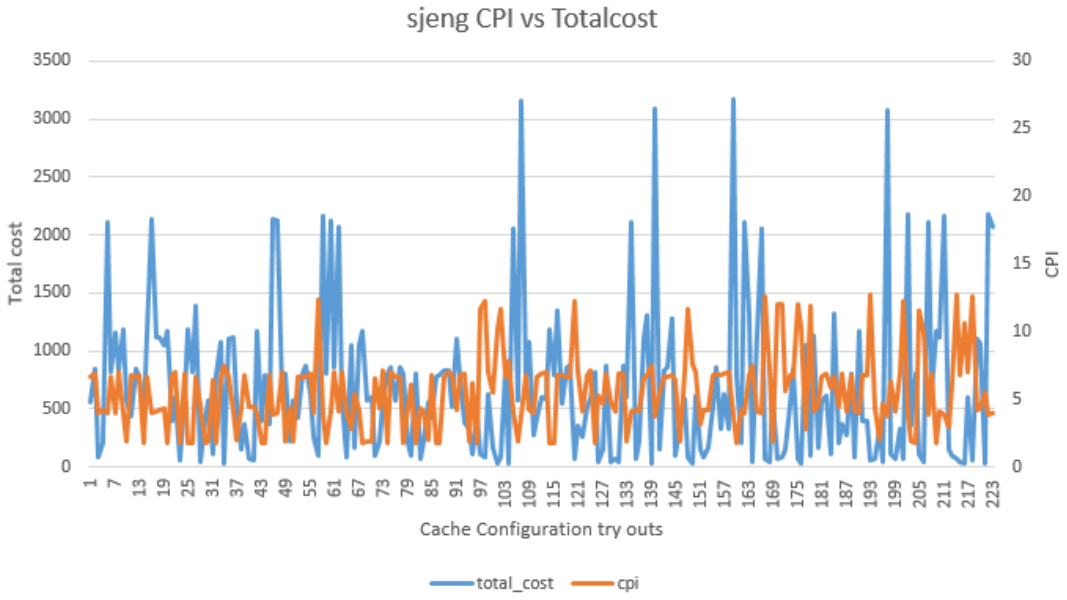


Figure 7: Graph between CPI and the Total Cost for sjeng

Redrawing the graph by **sorting the CPI value** optimum cache configuration for the sjeng Benchmark:

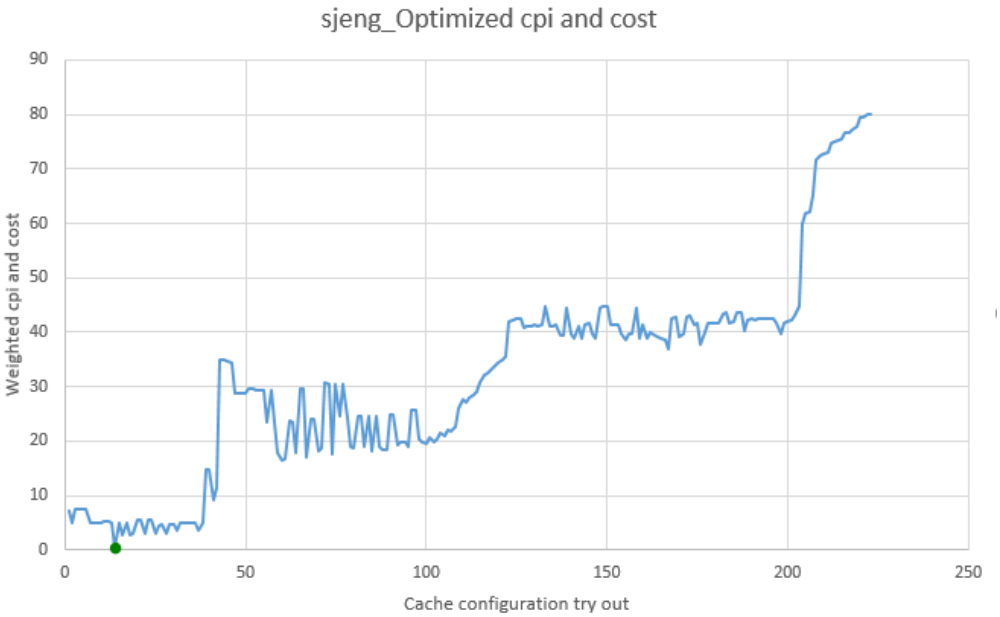


Figure 8: Optimized graph for CPI and cost for sjeng

The green mark denotes the lowest weighted value for this Benchmark.

From the valid set of cache configurations executed, the below configuration was found to produce best result

L1dsize	L1isze	L1_assoc	L2_size	L2_assoc	Line_size	CPI	Total_Cost
1kB	1kB	8	2kB	16	128	1.76748	54.066(units)

ANALYSIS:

sjeng Benchmark consists of Artificial Intelligence algorithm to play the chess game. The input of the Benchmark has only a single text file hence L1 cache size of 1kB is sufficient to load the input text file. Rather than memory usage, this Benchmark is more concentric towards the computation hence it can have low cache size to effectively run the benchmark. Line size of 128B is enough for this program to reside nearby in cache lines, thus reducing cache misses.

FLOATING POINT BENCHMARKS

lbm (Computation Analysis)

This belong to the general category of Computational Fluid Dynmaics, Lattice Boltzmann Method. This program implements the so-called "Lattice Boltzmann Method" (LBM) to simulate incompressible fluids in 3D. For benchmarking purposes, where the SPEC tools are used to validate the solution, the computed results are only stored.

In the Lattice Boltzmann Method, a steady state solution is achieved by running a sufficient number of model time steps. For the reference workload, 3000 time steps are computed. For the test and training workloads, a far smaller number of time steps are computed.

The geometry used in the training workload is different from the geometry used in the reference benchmark workload. Also, the reference workload uses a shear flow boundary condition, whereas the training workload does not. Nevertheless, the computational steps stressed by the training workload are the same as those stressed in the reference run.

From the stats.txt created from each configuration, number of Instruction misses given configuration is obtained. The CPI is calculated and the graph is plotted between the CPI and Total Cost for analysis

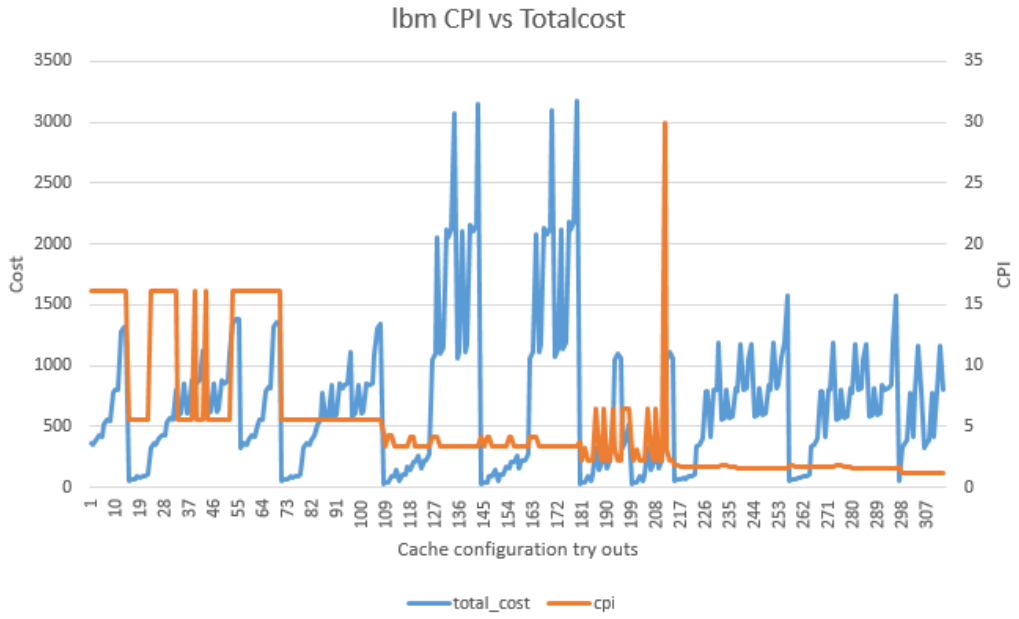


Figure 9: Graph between CPI and the Total Cost for lbm

Redrawing the graph by **sorting the CPI value** optimum cache configuration for the lbm Benchmark:

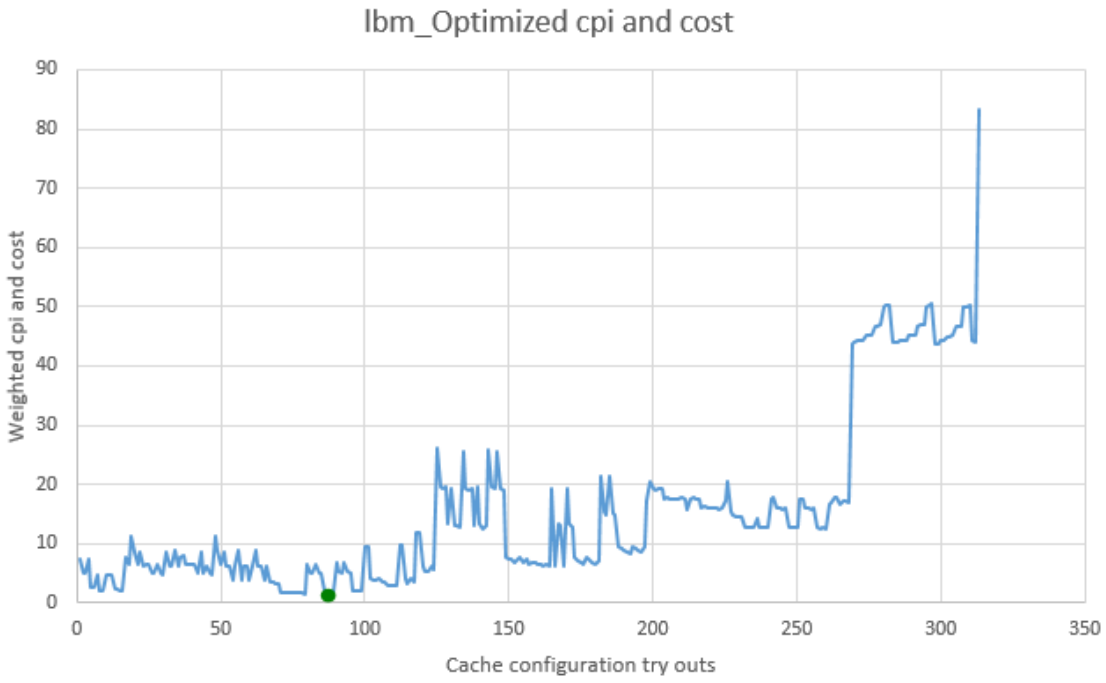


Figure 10: Optimized graph for CPI and cost for lbm

The green mark denotes the lowest weighted value for this Benchmark.

From the valid set of cache configurations executed, the below configuration was found to produce best result

L1dsize	L1isze	L1_assoc	L2_size	L2_assoc	Line_size	CPI	Total_Cost
1kB	1kB	8	2kB	16	128	1.6625312	54.066(units)

ANALYSIS:

lbm Benchmark is a floating point benchmark to simulate incompressible fluids in 3D. It consists of training workload to each step. The input data to the Benchmark involves only values of shear representation of fluid in single data file hence it does not require more data cache memory to load the input values. This Benchmark aims to train the workload which implies it has a large computation steps (3000 steps) large line size effectively increase the hit rate of the cache for all steps manipulated.

Scimark

In addition to the general Benchmarks given as per specification, we have included extra Benchmark in order to get clean insight of the variation in the cache configuration. This is a floating point benchmark which uses performs FFT, dense Lu matrix factorization. This test runs the Java version of SciMark 2.0, which is a benchmark for scientific and numerical computing developed by programmers at the National

Institute of Standards and Technology. This benchmark is made up of Fast Fourier Transform, Jacobi Successive Over-relaxation, Monte Carlo, Sparse Matrix Multiply, and dense LU matrix factorization benchmarks.

From the stats.txt created from each configuration, number of Instruction misses given configuration is obtained. The CPI is calculated and the graph is plotted between the CPI and Total Cost for analysis

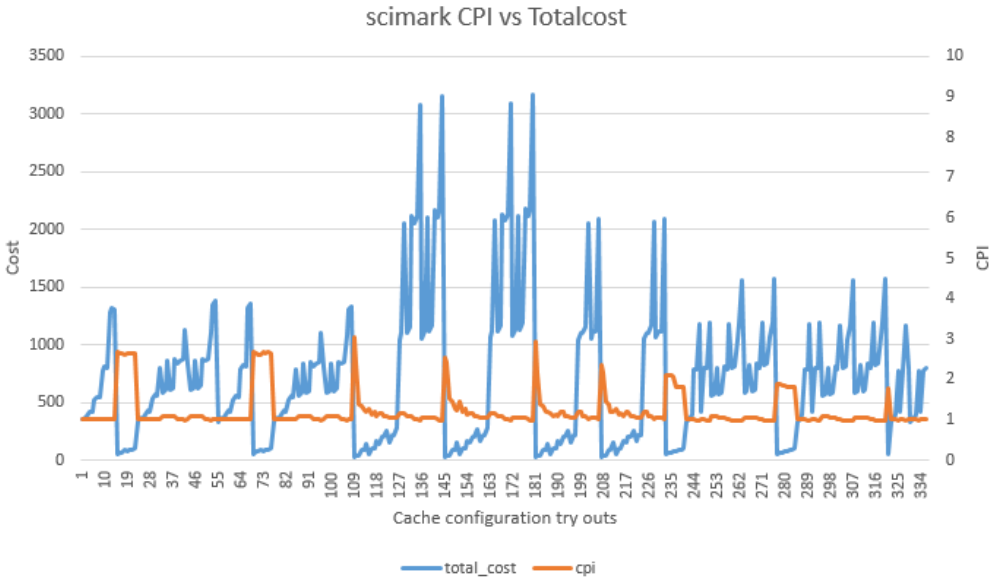


Figure 11: Graph between CPI and the Total Cost for scimark

Redrawing the graph by **sorting the CPI value** optimum cache configuration for the scimark Benchmark:

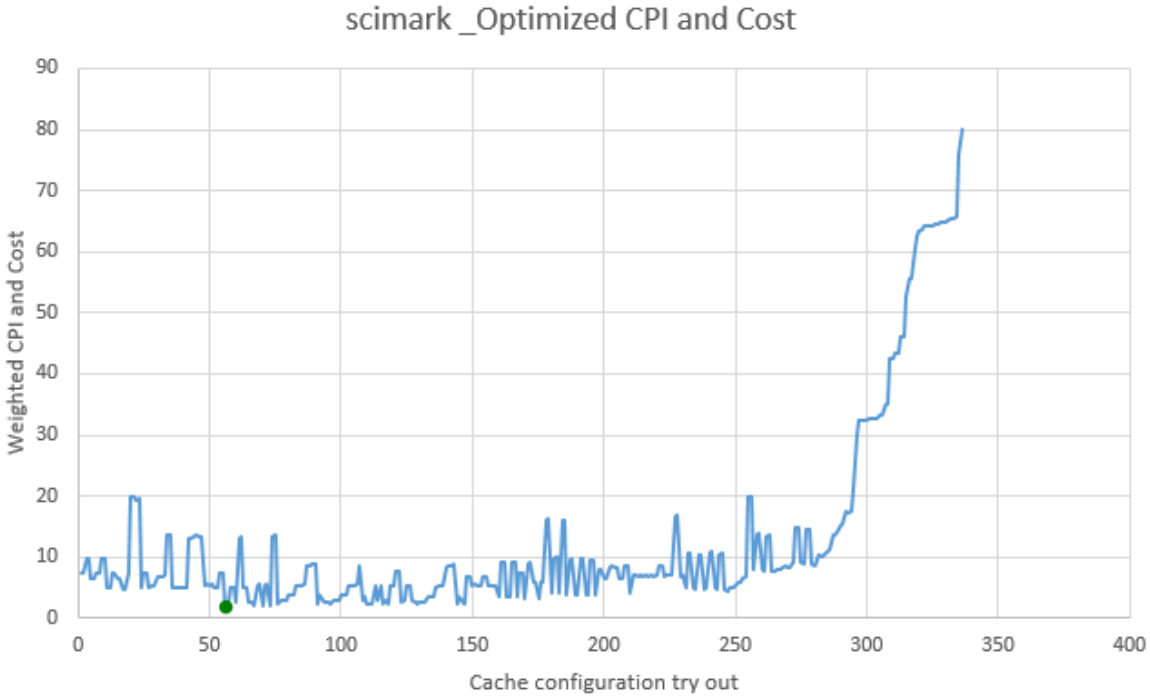


Figure 12: Optimized graph for CPI and cost for scimark

The green mark denotes the lowest weighted value for this Benchmark.

From the valid set of cache configurations executed, the below configuration was found to produce best result

L1dsize	L1isze	L1_assoc	L2_size	L2_assoc	Line_size	CPI	Total_Cost
16kB	16kB	2	64kB	8	512	1.00482526	331.056(units)

ANALYSIS:

scimark Benchmark is a floating point benchmark to simulate Fast Fourier Transform. Scimark mostly runs matrix computations and complex transform functions with bigger input data file. This makes L1dcache and L1icache of minimum size to be 16kB to load a large block of data. To make up memory latency and reuse the computed matrix Transforms it requires more L2size(64kB). Also, larger line size of 512B helps retrieve the program faster from cache.

Plotting Cache params Vs CPI Vs COST Trends

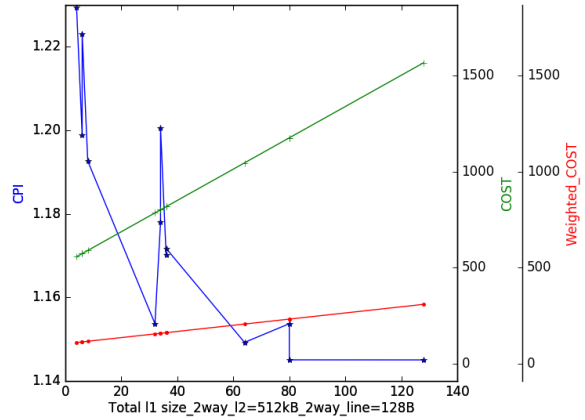
In this step, the main aim is to analyze the various cache parameters and its cost and CPI trend changes from the data obtained from running different combinations of each benchmark. Graphs are plotted against each parameter of the Cache Configuration with others parameters as constant to analyze the change in the CPI and cost-factor. These graphs are plotted using another python script that analyzes all the data from benchmark runs and extracting/filtering sub-groups which have one parameter varying and other parameters constant, which helps in plotting meaningful graphs.

Bzip2 plots:

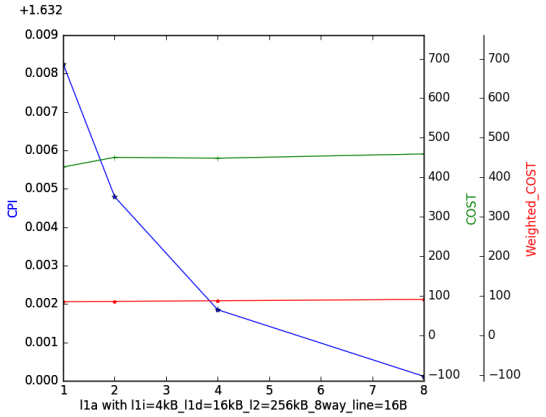
Keeping other parameters as constant, the cost and CPI trend for each and every valid combination are plotted. Analyzing CPI and cost seeks to increase accordingly satisfying the conditions to be true.

Varying Total **L1 size**

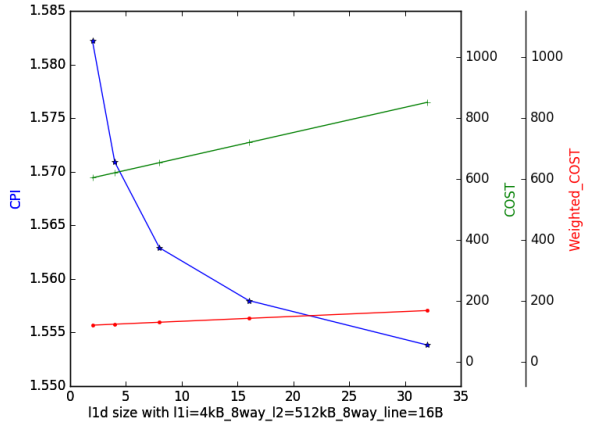
Varying **L1 associativity**



Varying **L1d size**



Varying **L1i size**



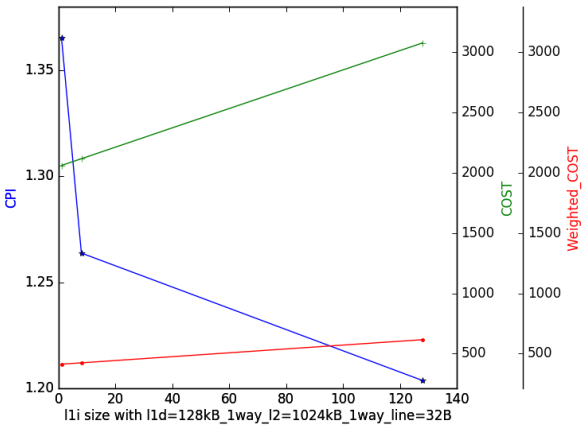
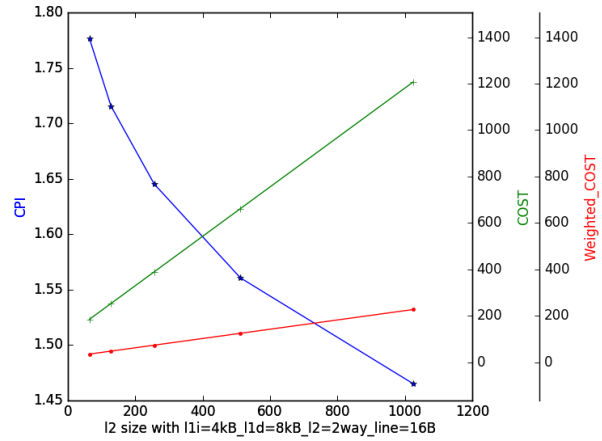
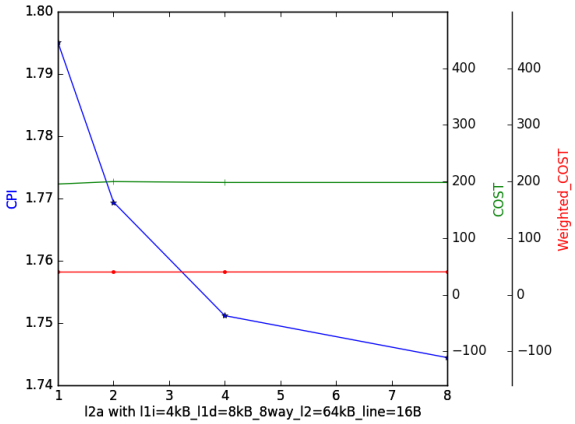


Figure 13: bzip2: cache params vs CPI vs COST plots

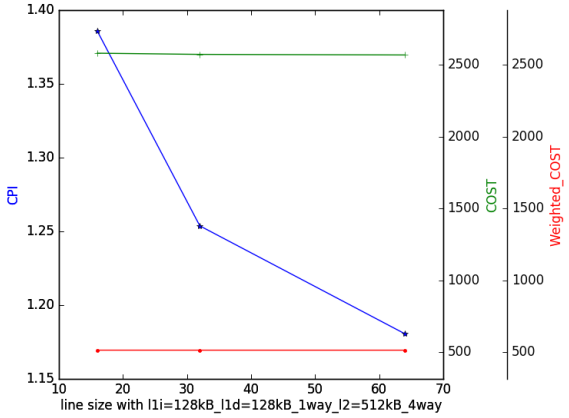
Varying **L2 size**



Varying **L2 associativity**

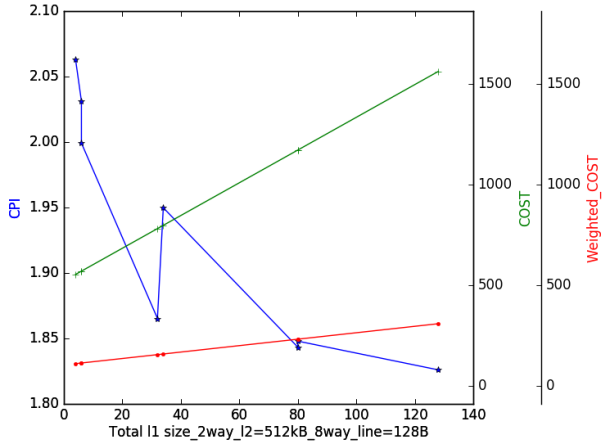


Varying **Line size**

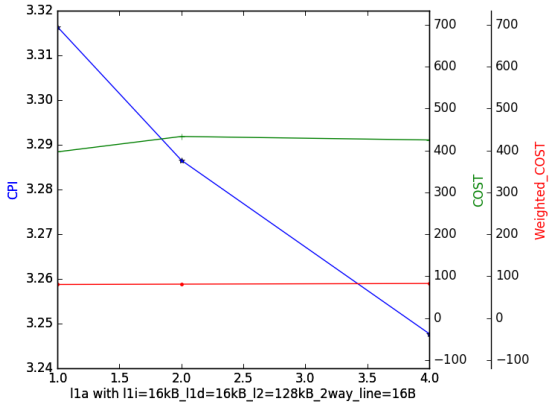


Mcf:

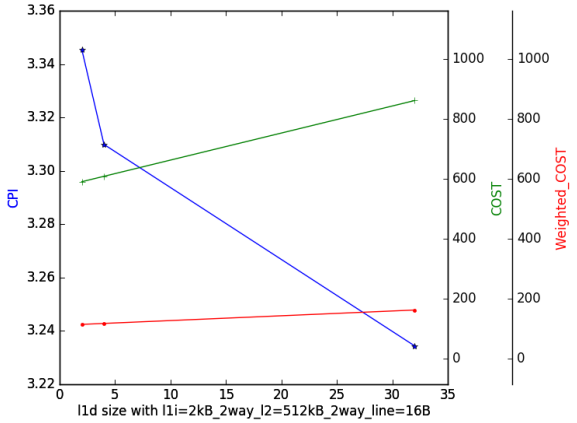
Varying total **L1 size**



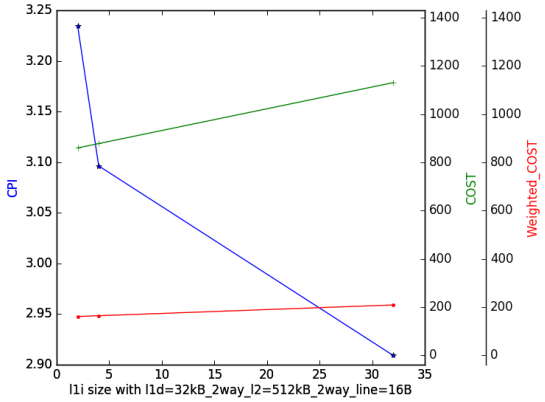
Varying **L1 associativity**



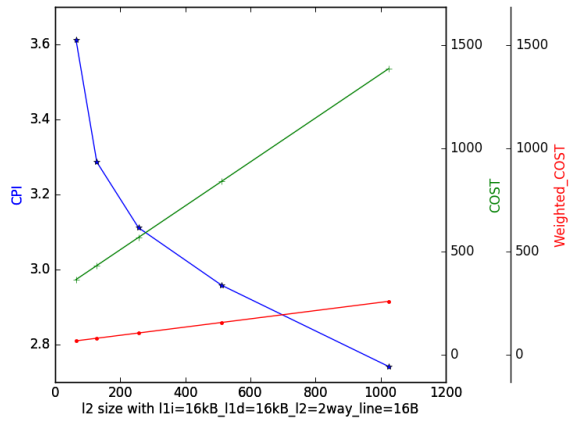
Varying **L1d size**



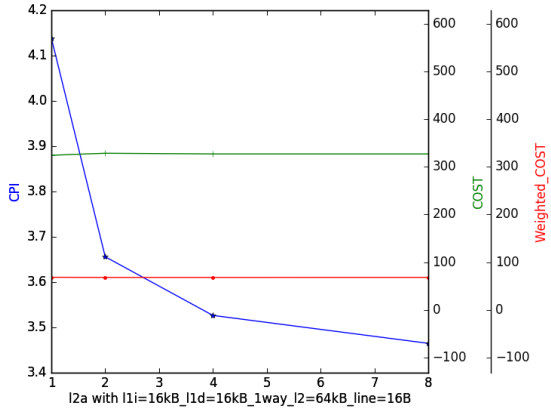
Varying **L1i size**



Varying **L2 size**



Varying **L2 associativity**



Varying **Line size**

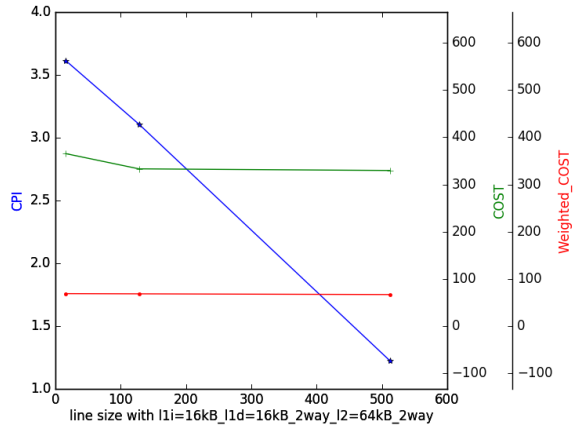
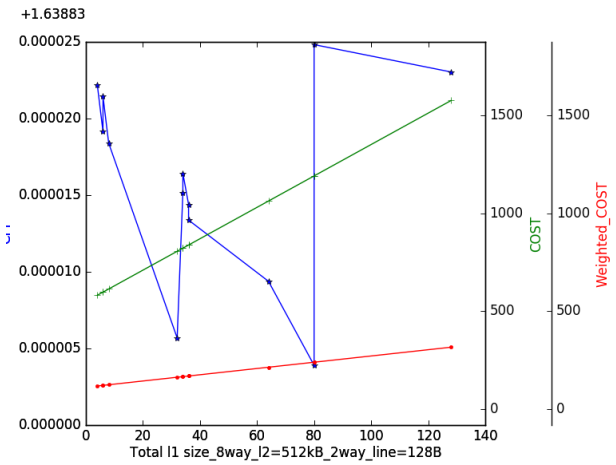


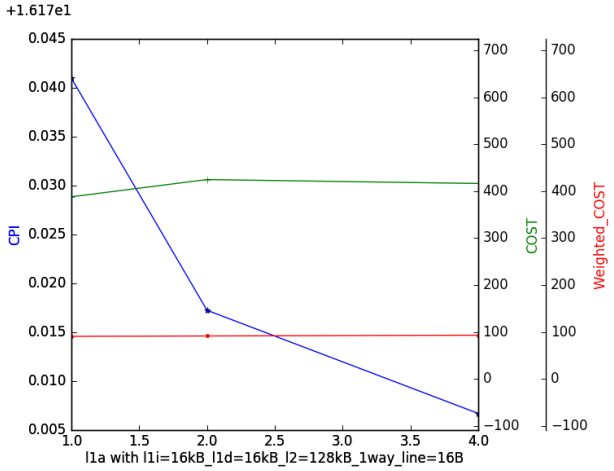
Figure 14: mcf: cache params vs CPI vs COST plots

Lbm:

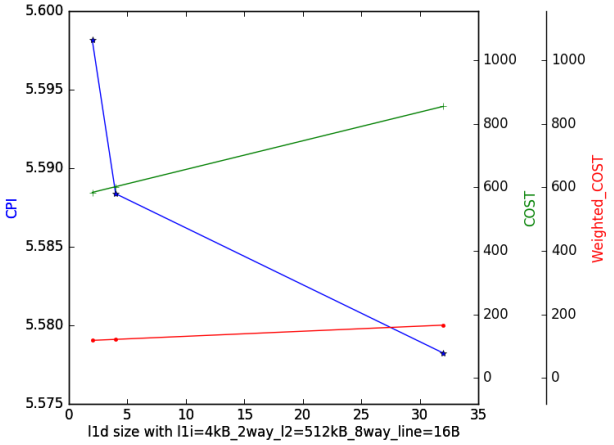
Varying total **L1 size**



Varying **L1 associativity**



Varying **L1d size**



Varying **L1i size**

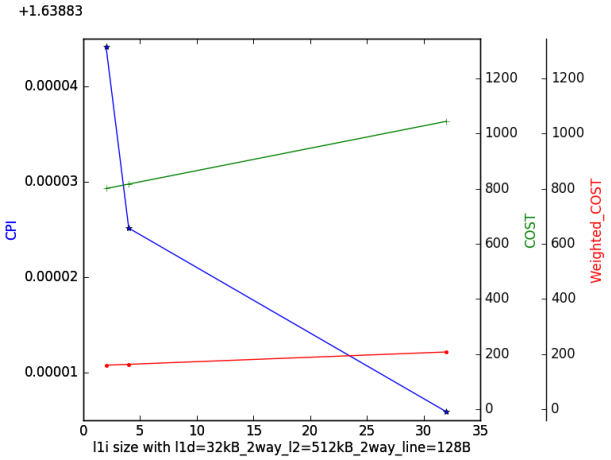
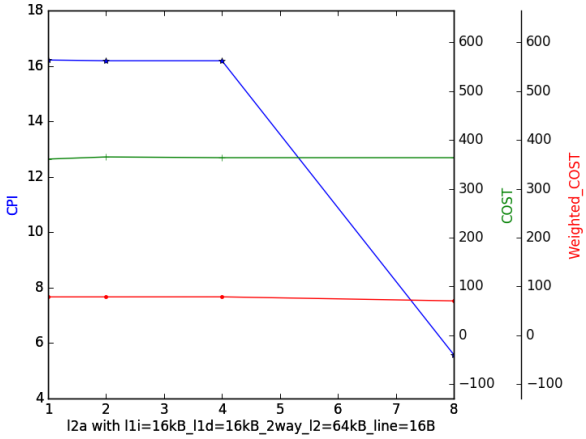
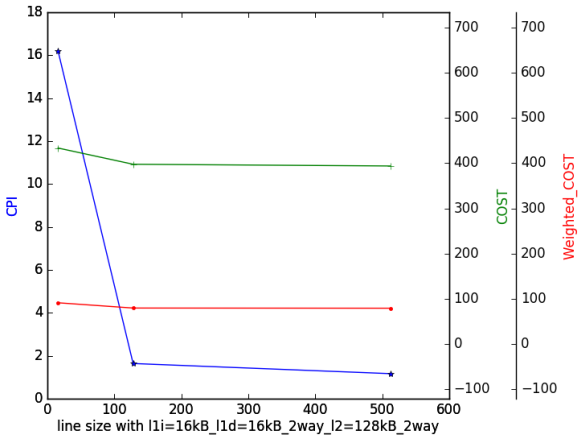


Figure 15: lbm: cache params vs CPI vs COST plots

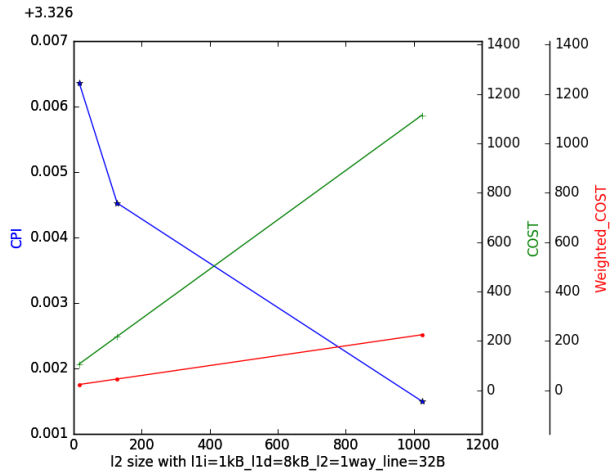
Varying **L2 associativity**



Varying **line size**

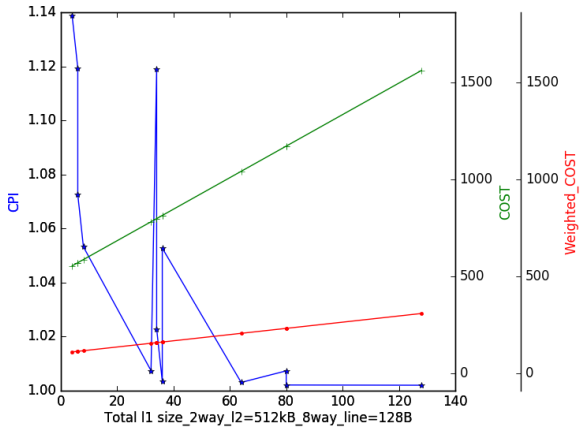


Varying **L2 size**

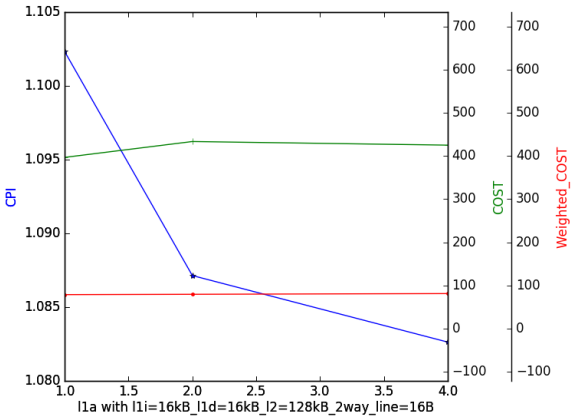


Hmmer:

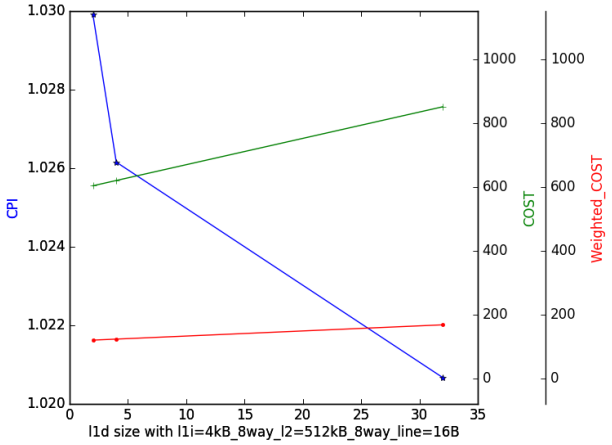
Varying total **L1 size**



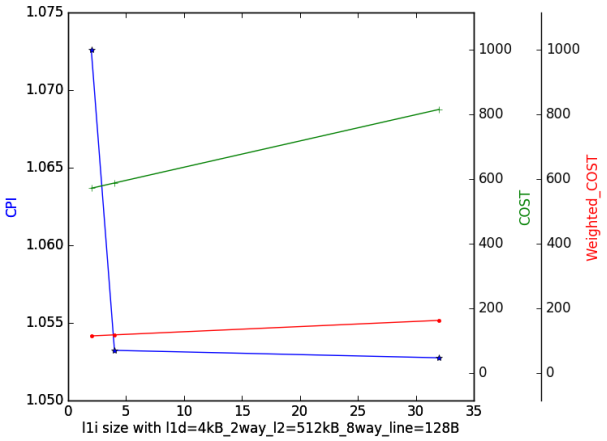
Varying **L1 associativity**



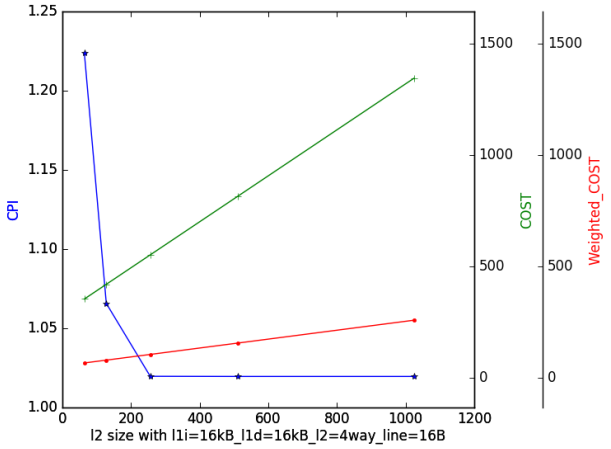
Varying **L1 dsize**



Varying **L1i size**



Varying **L2 size**



Varying **L2 associativity**

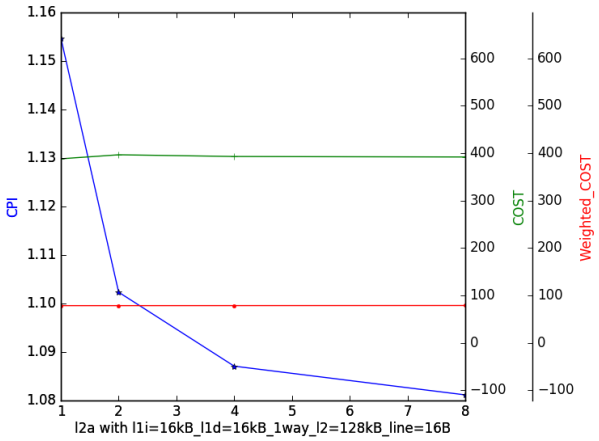
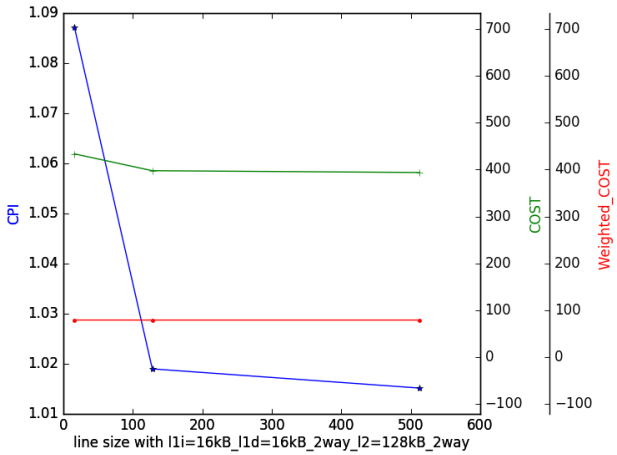


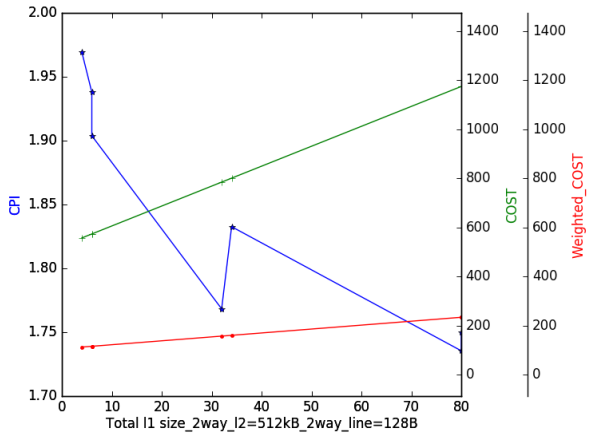
Figure 16: hmmer: cache params vs CPI vs COST plots

Varying **Line size**

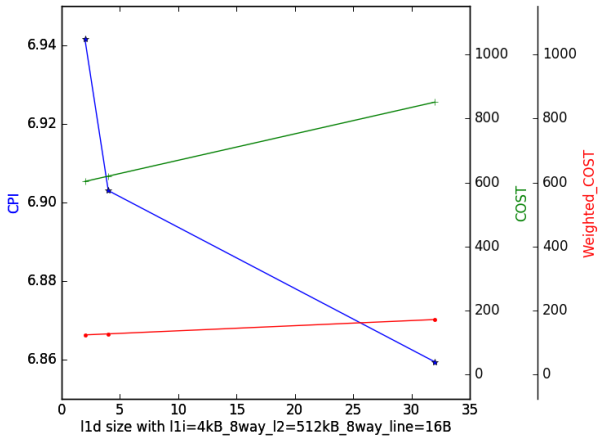


Sjeng:

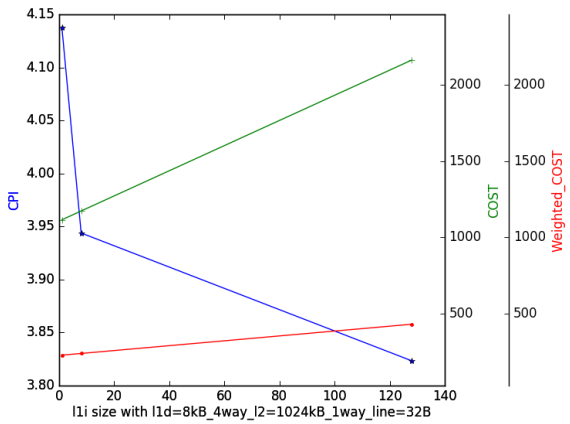
Varying total **L1 size**



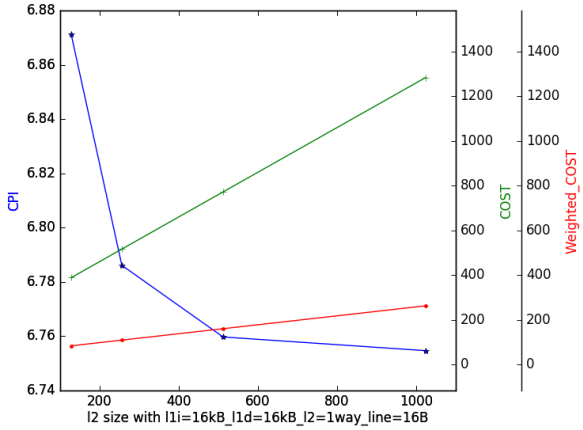
Varying **L1d size**



Varying **L1i size**



Varying **L2 size**



Varying **L2 associativity**

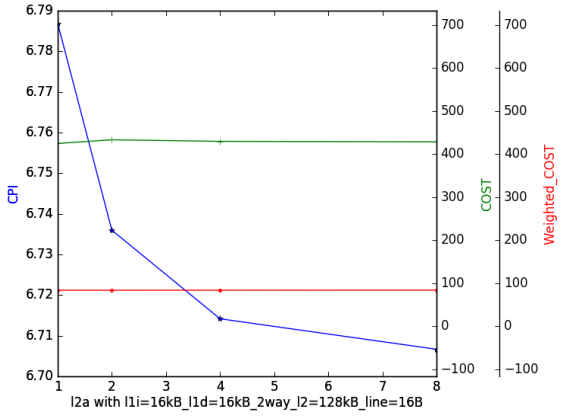
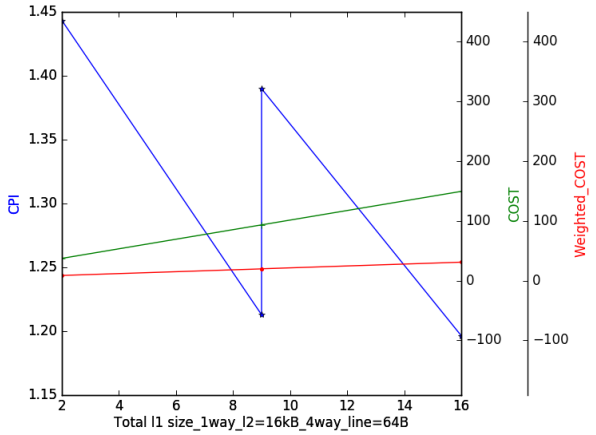


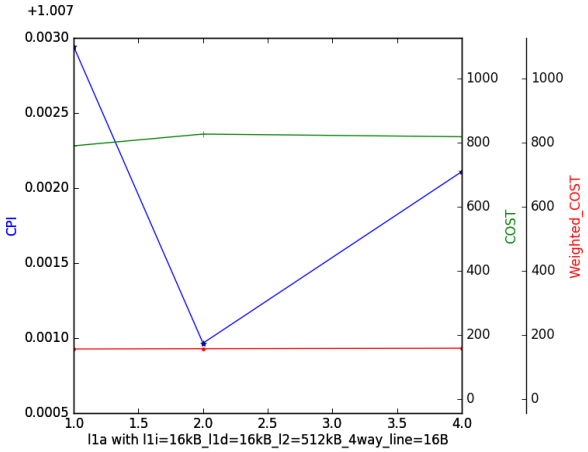
Figure 17: sjeng: cache params vs CPI vs COST plots

Scimark:

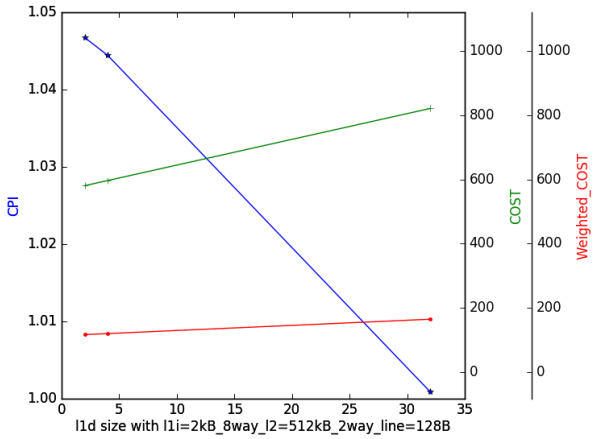
Varying total **L1size**



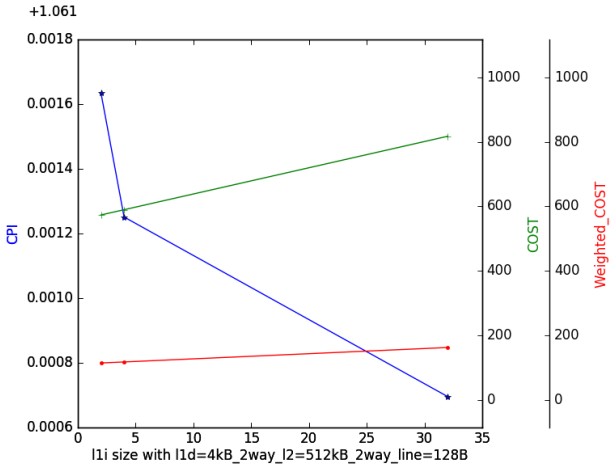
Varying **L1 associativity**



Varying **L1dsize**



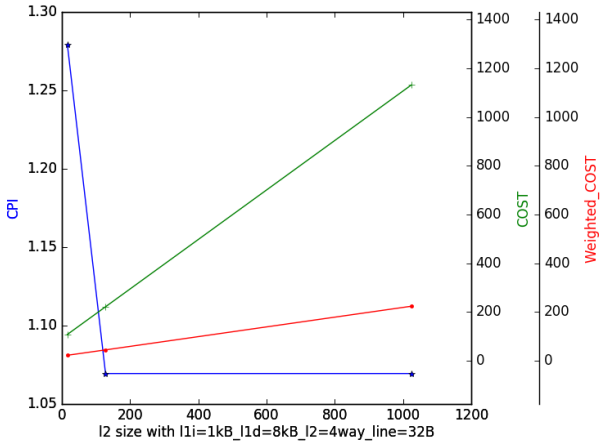
Varying **L1isize**

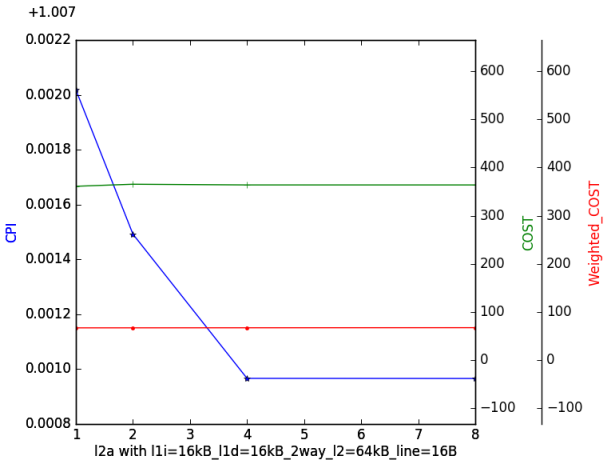


Varying **L2size**

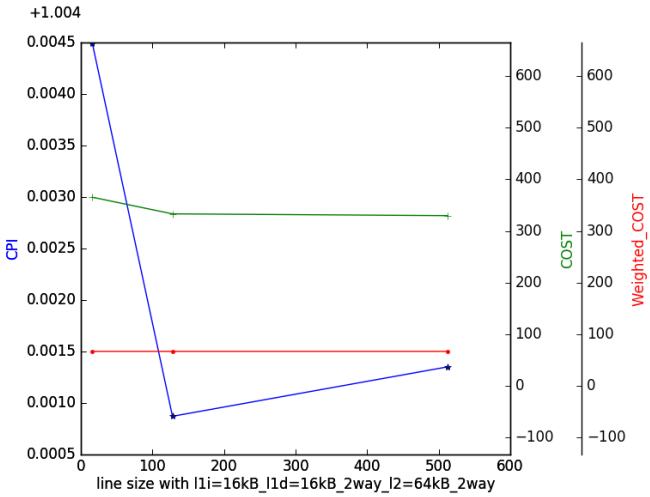
Varying **L2 associativity**

Figure 18: scimark: cache params vs CPI vs COST plots





Varying **Line size**



PART 4: Optimize caches for performance/cost

Until this part the various analysis has been done to estimate the performance of cache hierarchy for respective Benchmarks. Now considering the overall performance of the System and cost factor, a configuration with good CPI and decent cost is evaluated.

By using the weighted priority (Evaluation function) the normalized cost factor is obtained. The below mentioned cost factor is found to have good performance.

L1dsize	L1isze	L1_assoc	L2_size	L2_assoc	Line_size	Total_Cost
64kB	64kB	8	128kB	8	128	54.066(units)

ANALYSIS:

By having a larger L1 d cache size and L1i cache size the performance of the system can be increased thus reducing the cache miss rate. L2 unified cache is 128kB improves the holds both data and instruction. Though the cost of the configuration seems to be pretty much on the higher side it guarantees maximum hit. This Cache hierarchy can handle even more complicated Benchmarks which includes high cache usage, with a high cache hit rate. Thus this forms a robust cache hierarchy with the providing the best performance. Line size is taken to be 128kB which loads the program in the nearby memory by Spatial locality, it attains maximum hit rate.

This cache hierarchy was found to have following CPI values for various Benchmarks.

Benchmark	CPI
Bzip2	1.290242872
mcf	2.544986168
hmmer	1.003956032
lbm	1.7154342
sjeng	1.639343736
scimark	1.0001082

The position of the cache configuration for each Benchmark is shown below:
The Black dot denotes the position of the cache configuration in the try outs.

Bzip2:

The sample command line for the benchmark is given below.

command line:

/home/jayaram/comp_arch/gem5-stable/build/X86/gem5.opt -d /home/jayaram/comp_arch/Prj1/Project1_SPEC/401.bzip2/m5out/bzip2_64kBiCache_64kBdCache_8way_128kBL2_8way_128cache /home/jayaram/comp_arch/gem5-stable/configs/example/se.py -c /home/jayaram/comp_arch/Prj1/Project1_SPEC/401.bzip2/src/benchmark -o '/home/jayaram/comp_arch/Prj1/Project1_SPEC/401.bzip2/data/input.program 10' -I 500000000 --cpu-type=timing --caches --l2cache --l1d_size=64kB --l1i_size=64kB --l2_size=128kB --l1d_assoc=8 --l1i_assoc=8 --l2_assoc=8 --cacheline_size=128

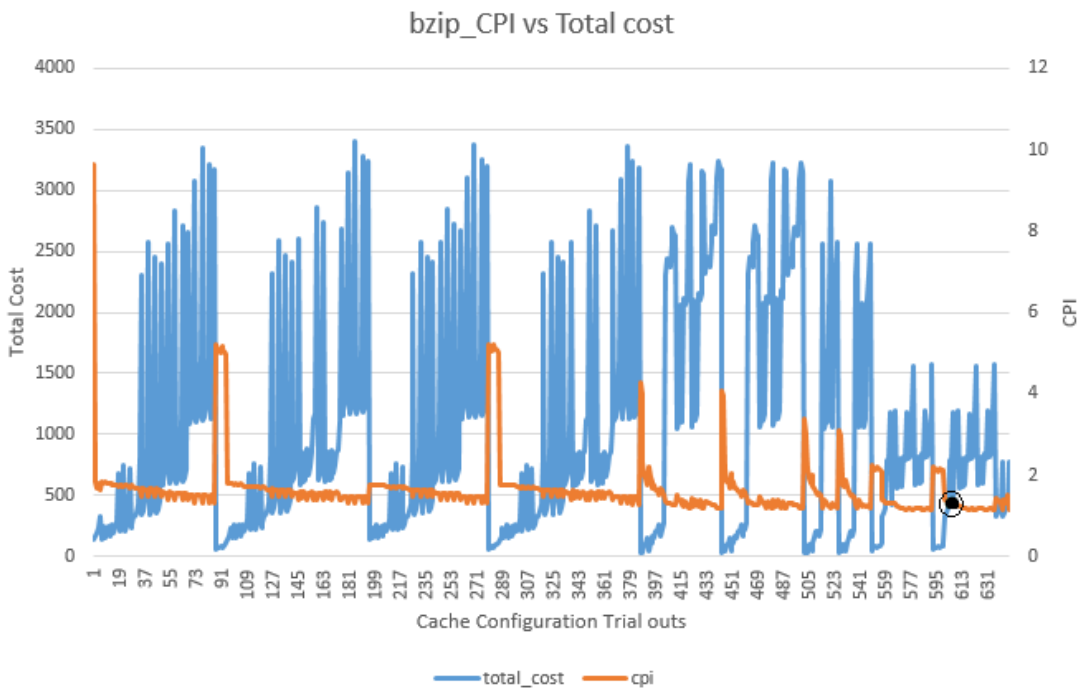


Figure 19: bzip2: cache conf tryouts vs CPI vs COST

The Black dot denotes the position of the cache configuration in the try outs.

Mcf:

The sample command line for the benchmark is given below.

command line:

/home/jayaram/comp_arch/gem5-stable/build/X86/gem5.opt -d /home/jayaram/comp_arch/Prj1/Project1_SPEC/429.mcf/m5out/mcf_64kBiCache_64kBdCache_8way_128kBL2_8way_128cache /home/jayaram/comp_arch/gem5-stable/configs/example/se.py -c /home/jayaram/comp_arch/Prj1/Project1_SPEC/429.mcf/src/benchmark -o /home/jayaram/comp_arch/Prj1/Project1_SPEC/429.mcf/data/inp.in -I 500000000 --cpu-type=timing --caches --l2cache --l1d_size=64kB --l1i_size=64kB --l2_size=128kB --l1d_assoc=8 --l1i_assoc=8 --l2_assoc=8 --cacheline_size=128

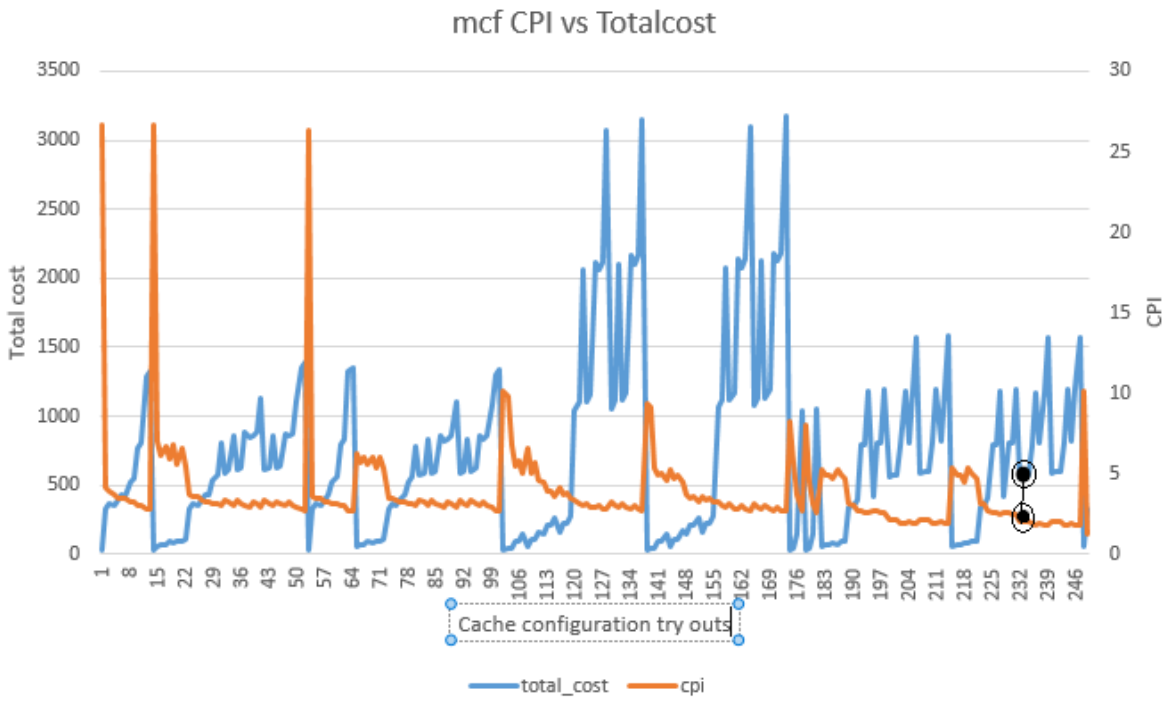


Figure 20: mcf: cache conf tryouts vs CPI vs COST

The Black dot denotes the position of the cache configuration in the try outs.

Hmmer:

The sample command line for the benchmark is given below.

command line:

/home/jayaram/comp_arch/gem5-stable/build/X86/gem5.opt -d /home/jayaram/comp_arch/Prj1/Project1_SPEC/456.hmmer/m5out/hmmer_64kBiCache_64kBdCache_8way_128kBL2_8way_128cache /home/jayaram/comp_arch/gem5-stable/configs/example/se.py -c /home/jayaram/comp_arch/Prj1/Project1_SPEC/456.hmmer/src/benchmark -o '--fixed 0 --mean 325 --num 45000 --sd 200 --seed 0 /home/jayaram/comp_arch/Prj1/Project1_SPEC/456.hmmer/data/bombesin.hmm' -I 500000000 --cpu-type=timing --caches --l2cache --l1d_size=64kB --l1i_size=64kB --l2_size=128kB --l1d_assoc=8 --l1i_assoc=8 --l2_assoc=8 --cacheline_size=128

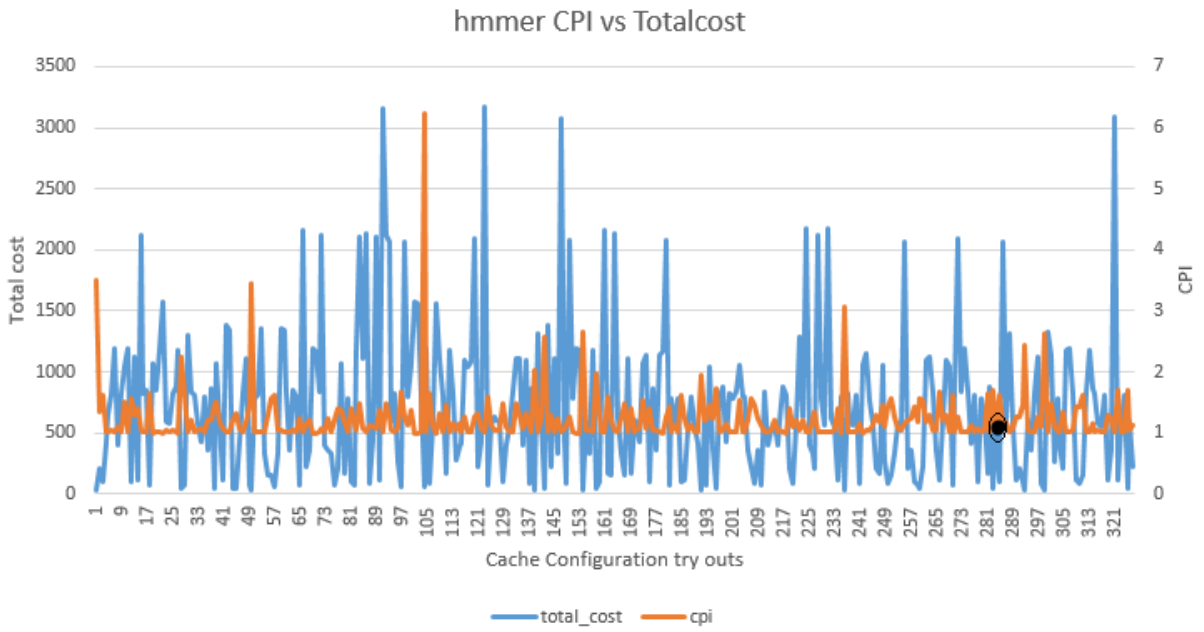


Figure 21: hmmer: cache conf tryouts vs CPI vs COST

The Black dot denotes the position of the cache configuration in the try outs.

Sjeng:

The sample command line for the benchmark is given below.

command line:

/home/kalyan/comp_arch/gem5-stable/build/X86/gem5.opt -d /home/ kalyan /comp_arch/Prj1/Project1_SPEC/458.sjeng/m5out/sjeng_64kBiCache_64kBdCache_8way_128kBL2_8way_128cache /home/ kalyan /comp_arch/gem5-stable/configs/example/se.py -c /home/ kalyan /comp_arch/Prj1/Project1_SPEC/458.sjeng/src/benchmark -o /home/ kalyan /comp_arch/Prj1/Project1_SPEC/458.sjeng/data/test.txt -I 500000000 --cpu-type=timing --caches --l2cache --l1d_size=64kB --l1i_size=64kB --l2_size=128kB --l1d_assoc=8 --l1i_assoc=8 --l2_assoc=8 --cacheline_size=128

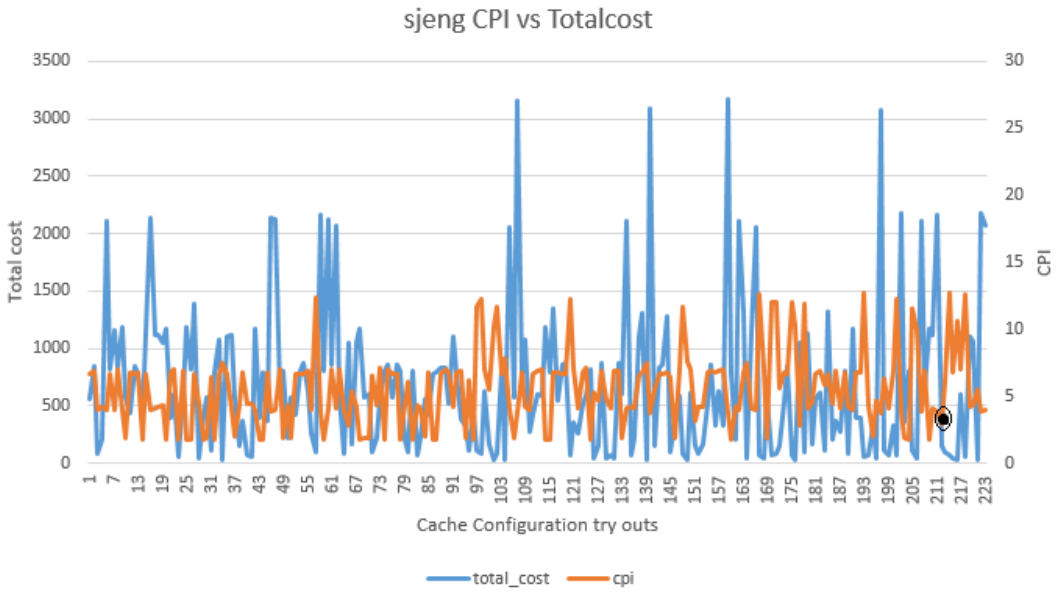


Figure 22: sjeng: cache conf tryouts vs CPI vs COST

The Black dot denotes the position of the cache configuration in the try outs.

Lbm:

The sample command line for the benchmark is given below.

command line:

/home/kalyan/comp_arch/gem5-stable/build/X86/gem5.opt -d /home/jayaram/comp_arch/Prj1/Project1_SPEC/470.lbm/m5out/lbm_64kBiCache_64kBdCache_8way_128kBL2_8way_128cache /home/ kalyan /comp_arch/gem5-stable/configs/example/se.py -c /home/ kalyan /comp_arch/Prj1/Project1_SPEC/470.lbm/src/benchmark -o '20 /home/ kalyan /comp_arch/Prj1/Project1_SPEC/470.lbm/data/reference.dat 0 1 /home/ kalyan /comp_arch/Prj1/Project1_SPEC/470.lbm/data/100_100_130_cf_a.of' -I 500000000 --cpu-type=timing --caches --l2cache --l1d_size=64kB --l1i_size=64kB --l2_size=128kB --l1d_assoc=8 --l1i_assoc=8 --l2_assoc=8 --cacheline_size=128

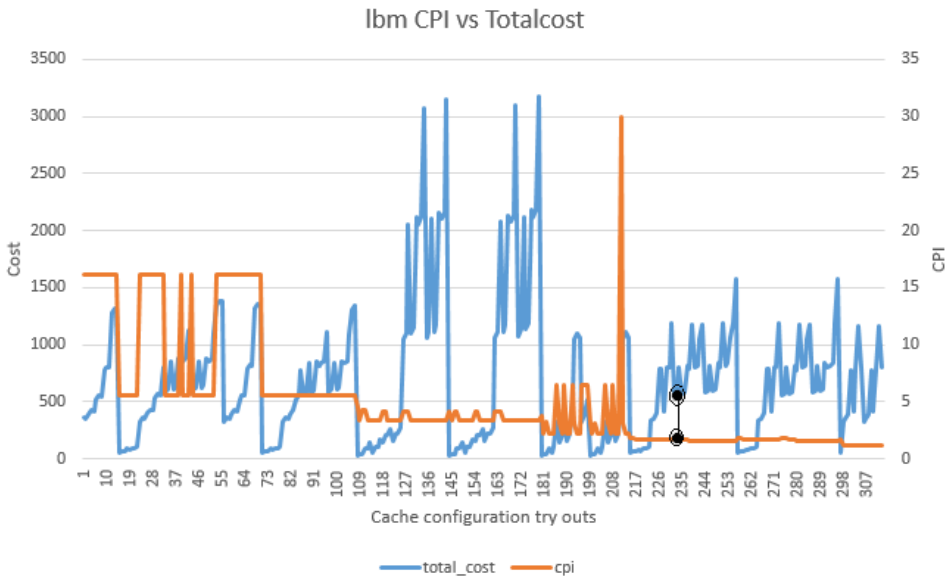


Figure 23: lbm: cache conf tryouts vs CPI vs COST

Scimark:

The sample command line for the benchmark is given below.

command line:

/home/kalyan/comp_arch/gem5-stable/build/X86/gem5.opt -d /home/ kalyan /comp_arch/Prj1/Project1_SPEC/scimark/m5out/scimark_64kBiCache_64kBdCache_8way_128kBL2_8way_128cache /home/ kalyan /comp_arch/gem5-stable/configs/example/se.py -c /home/kalyan/comp_arch/Prj1/Project1_SPEC/scimark/src/benchmark -o " -I 500000000 --cpu-type=timing --caches --l2cache --l1d_size=64kB --l1i_size=64kB --l2_size=128kB --l1d_assoc=8 –l1i_assoc=8 --l2_assoc=8 --cacheline_size=128

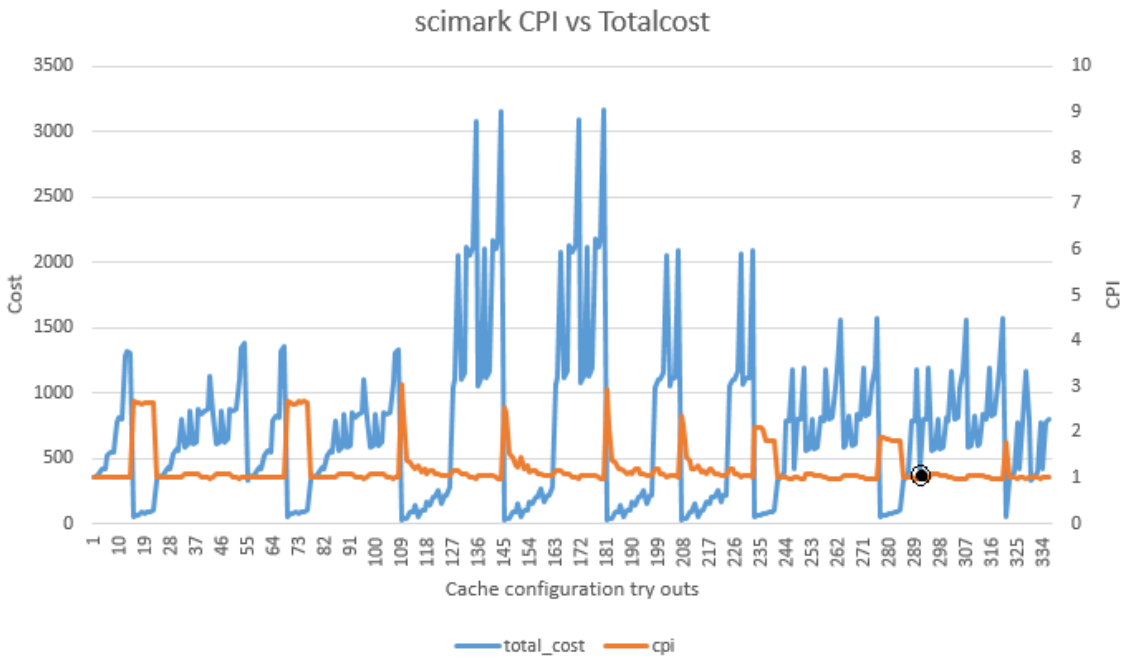


Figure 24: scimark: cacehe conf tryouts vs CPI vs COST

The Black dot denotes the position of the cache configuration in the try outs.

CONCLUSION:

Thus various cache hierarchy for X86 architecture for timing based CPU type is analyzed in the gem5 simulator. Various parameters such as L1cache data size, L1cache instruction size, L2 cache size, associativity and block size are varied to get the minimum CPI for 6 individual benchmarks, namely 401.bzip2, 429.mcf, 456.hmmer, 458.sjeng, 470.lbm, scimark. At the final stage of the project, a cost function is defined and an optimum configuration is obtained by plotting the CPI against the cost function for each cache configuration.

References:

1) *Computer Architecture A Quantitative Approach 5th Edition by John L. Hennessy and David A. Patterson*
2) *www.stackoverflow.com – python script building and reference*
3) *http://gem5.org/PARSEC_benchmarks*
4) *http://www.m5sim.org/SPEC_CPU2006_benchmarks*
5) *Computer Organization and Architecture: Designing for Performance (Hardcover)*
6) *by William Stallings*
7) *https://markgottscho.wordpress.com/2014/09/20/tutorial-easily-running-spec-cpu2006-benchmarks-in-the-gem5-simulator/*
8) *https://www.spec.org/cpu2006/Docs*
9) *http://math.nist.gov/scimark2/*