Alec Bernardi and Allison Ariemma

Instruction Pipeline and Cache Simulator

Distribution of Work:

For this project, Allison completed the cache simulator and this write up. Alec completed the pipeline simulator and ensured that the output matched exactly for the test case provided.

Allison Ariemma: 100%

Alec Bernardi: 100%

Overview of Implementation:

The structure of the cache simulator is essentially an array of cache lines, each stored as a struct. Each cache line consists of a valid bit and an array to store the correct number of tags, with the maximum number being equal to the associativity. The LRU is implemented by storing the tags in order from most recently used to least recently used. That way when a tag needs to be replaced, the last element is kicked off, the rest of the tags are shifted down one index, and the new tag is put in index 0 of the array.

The pipeline simulator contains the 5 stages: fetch, decode, ALU, mem, and write back. Essentially upon every instruction that is read in, the pipeline checks for LW delays in the ALU stage, SW memory accesses and data misses, as well as the branch prediction when necessary. If the prediction was incorrect, we must add a nop. The pipeline simulator keeps track of the pipeline cycles, so that we can find the CPI in the end with the instruction count.

Tabulation of Results for 18 Different Combinations of Inputs

Block size	Associativity	Max Index	Taken?	Cache Miss Rate	CPI	Correct number of branch predictions (out of 7044)
1	1	7	0	0.038627	1.405359	5730
1	2	6	0	0.013878	1.186419	5739
1	4	5	0	0.010025	1.152652	5744
2	1	6	0	0.036162	1.381997	5726
2	2	5	0	0.009868	1.149133	5736
2	4	4	0	0.005521	1.110924	5746
4	1	6	0	0.021334	1.251535	5729
4	2	5	0	0.003979	1.09435	5749
4	4	4	0	0.002153	1.079451	5749
1	1	7	1	0.038627	1.533886	1279
1	2	6	1	0.013878	1.315875	1280
1	4	5	1	0.010025	1.282285	1284
2	1	6	1	0.036162	1.510061	1290
2	2	5	1	0.009868	1.278231	1290
2	4	4	1	0.005521	1.240485	1290
4	1	6	1	0.021334	1.380083	1291
4	2	5	1	0.003979	1.223849	1291
4	4	4	1	0.002153	1.209097	1291

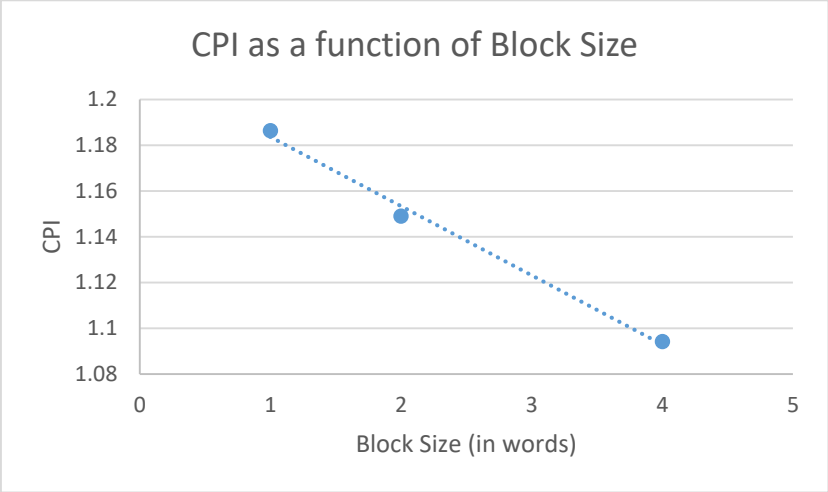
The maximum index values were calculated using the following formula with the restriction that the maximum CacheSize is 10,240 bits

$$\text{CacheSize} = \text{Associativity} * (2^{\text{IndexBits}} * (32 * \text{BlockSize} + 33 - \text{IndexBits} - \text{BlockOffSetBits}))$$

From the table above it is clear that the best configuration occurs with a block size of 4 words and a 4-way set associative cache, as well as predicting the branch as not taken. This is true because it has the lowest cache miss rate of .002155, the lowest CPI of 1.079451, and the highest number of correct branch predictions which is 5749 out of the 7044 total. Clearly, the branch prediction does not affect the cache miss rate, so thus the prediction is irrelevant for this variable. Having a larger block size is beneficial because of spatial locality. Additionally, having a greater associativity is important because this way if elements map to the same index in the cache, more of them can be stored and if we have too many conflicts, we just remove the least

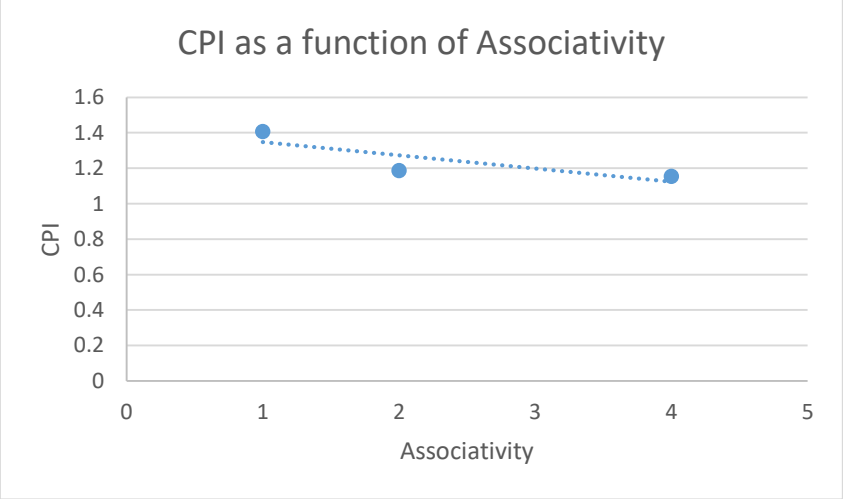
recently used. Thus increased associativity decreases the miss rate; however, at a certain point, perhaps around 8-way, it no longer provides the same amount of return. Additionally, it is clear from the table that all of the tests that the branch was not taken perform better and predict more correctly then their counter parts.

Graph Showing Block Size Effect on CPI (Associativity held constant at 2 and prediction held constant as not taken):



This graph confirms that as block size increased, CPI decreased for this assignment.

Graph Showing Associativity's Effect on CPI (Block size held constant at 2 and prediction held constant as not taken):



This graph confirms that as associativity increased, CPI decreased for this assignment.