Nick Roper
Glen Madsen

Computer Organization Group Project

Instruction Pipeline and Cache (IPCL) Simulator

General Overview:

- The program starts by taking input in the main function, and once it is done, it calls the cache function iplc_sim_init.
- iplc_sim_init assigns input values to global variables, and, using these input variables, dynamically allocates the cache and puts nops into the pipeline so that it has defined values.
- Then, in the main function, iplc_parse_instruction is called for each line of the trace file, which corresponds to an instruction.
- For the input instruction, iplc_parse_instruction calls the cache function iplc_trap_address to determine whether or not the instruction was in the cache, because if it was a miss, then the pipeline must be pushed 9 times by calling iplc_sim_push_pipeline_stage because the cache miss delay is 10 cycles, and another cycle will be added when the pipeline is pushed again later, after the instruction has been found.
- Then, it parses the rest of the instruction and calls one of:
    - iplc_sim_process_pipeline_rtype
    - iplc_sim_process_pipeline_lw
    - iplc_sim_process_pipeline_sw
    - iplc_sim_process_pipeline_branch
    - iplc_sim_process_pipeline_jump
    - iplc_sim_process_pipeline_syscall
    - iplc_sim_process_pipeline_nop

  depending on what the instruction type was determined to be.
- After parsing through the entire input file, iplc_sim_finalize is called to finish processing instructions still left in the pipeline, then frees the memory allocated for the cache in iplc_sim_init before printing the summary statistics.

Cache Details:

When designing our cache, we thought it would be best if it was organized as an array, of arrays, of structs. In the iplc_sim_init function we modified the allocation part of it so that it would fit this design, where the outer array's size is determined by the cache bit, and the inner array size by the desired associativity. We also changed the implementation of the innermost struct so that it now only contains a valid bit integer and tag integer (making sure to set all the valid bits of the struct to 0). Next, when implementing the first cache function, iplc_sim_trap_address, we had to calculate the tag and index within the address. The tag we found by right shifting the address by the cache index bit plus the block offset bits which was calculated for us in iplc_sim_init. Then, to calculate the index we once again right shifted the address by the block offset bits and 'anding' it with 1 left shifted by the cache index bit minus one ((1 << cache index bit) – 1). After performing this bitmask, we implement the real meat of

the function, determining whether or not the given tag is in our cache. We do this by going through the inner array (of associative sets) and looking to see if any of the elements share a tag, or if the valid bits are zero. Here, we can determine whether or not it either missed, or hit, increment our variables, and call either the iplc_sim_LRU_update_on_hit or iplc_sim_LRU_replace_on_miss functions respectively.

iplc_sim_LRU_update_on_hit function: To update on hit means to essentially reorder or mark the associative sets such that we know the ordering of most, to least recently used. Because our struct has no placement array, history integer, or anything, we make sure to keep an ordering (of associative sets) when placing elements into the cache to begin with. So, when ordering the associative sets, we make sure the least recently used element is at the left most end (if the array was drawn out on paper horizontally), the most recently used element on the right end of all the sets with a valid bit of 1, and any sets with a valid bit of 0 following the most recent element. So, when updating on hit, we have to reorder the sets such that the tag we found is at the rightmost end (and followed by any remaining valid bit zeroes). To do this, we save the tag we need moved to the right in an integer named tag, and then enter a for loop. This for loop starts at the location of our element, and first checks if we are not the last element in the array, then it looks at the to the right of it (i+1), checking if it's valid bit is not 0. If it does happen to be zero, then the element is already on the rightmost edge and thus 'most recently used' so we do nothing. Otherwise, we look at the next element (tag, not the variable), and set our current element (tag, not the variable) equal to it. We then repeat this process till we hit the edge of the array or find a valid bit of zero as the next element. This essentially performs a shift of all elements to the right of our initial element to the left, so we can safely place our tag variable at the end and be sure it in the most recent position, completing the update process.

iplc_sim_LRU_replace_on_miss function: The third main cache function we had to implement was a simple replace cache function. So, based on the ordering described in the previous function, the first thing we did was go forwards through our array looking for valid bits of 0 (iterating i positively starting at 0). If we find a zero we place our element there and change the valid bit, but if not then we go to the next for loop. This for loop is very similar to the for loop in the previous function, but since we know there are no zeroes and our element is not present in the array, we start at beginning of our array. From here we set each element to the element ahead of it (i+1) until the end where we stop before going out of bounds, and set that last element to the tag that we missed finding earlier. By shifting all the elements to the left this gets rid of the oldest element at the leftmost end (since elements are always placed left to right and shifted left to make room for new ones or just reordered on a hit). This completes the replacing function while also making sure that we always replace the least recently used element.

With those three functions implemented, our cache is nearly complete. All we have left is to eventually free the allocated memory, which we do in the iplc_sim_finalize function. So in conclusion, we use an array of arrays of structs to represent our cache, and make sure the structs with tag and index variables that are always ordered from least to most recently used. This means we can always place them in the location specified by the index, get or check for the tag's we

need, and get rid of the least recently used element and update the most, thus fulfilling all the requirements for our cache.

Pipeline Details:

When processing an instruction, one of the iplc_sim_process_[instruction type] functions is called depending on what the instruction type is. Each one of these functions calls iplc_sim_push_pipeline_stage, so this will be explained first:

iplc_sim_push_pipeline_stage is split up into 7 different steps:

1. If the instruction in the writeback stage is not a nop, instruction_count is incremented, as the instruction will be removed from the pipeline in step 6.
2. If the instruction in the decode stage is a branching instruction, branch_count is incremented, as a branching instruction has been found, and then, if the previous instruction is not a nop, checks branch prediction:
    a. If predicting taken, checks the instruction in the fetch stage to see if its address is 4 after the branching instruction. If it is not, correct_branch_predictions is incremented, as the prediction was correct, and if the address is 4 after, then it increments pipeline_cycles, as the prediction was incorrect and the pipeline must stall to fetch the proper instruction.
    b. If predicting not taken, checks the instruction in the fetch stage to see if its address is 4 after the branching instruction. If it is, correct_branch_predictions is incremented, as the prediction was correct, and if the address is not 4 after, then it increments pipeline_cycles, as the prediction was incorrect and the pipeline must stall to fetch the proper instruction.
3. If the instruction in the memory stage is a load word instruction, checks to see if the instruction in the ALU stage is an r-type/branching instruction. If it is, and either reg1 or reg2 of the ALU instruction is the same as the dest_reg for the load word instruction, then we mark that a nop must be inserted, as there will be a stall because the r-type/branching instruction has to wait for the load word instruction to finish loading the data from memory before it is able to use it. iplc_sim_trap_address is then called on the data address, and if it is a hit, nothing need be done unless we found that we had to insert a nop, in which case pipeline_cycles is incremented to account for the incorrect branch prediction. If the address misses in the cache, then the pipeline cycles is incremented by one less than the cache miss delay (since it is automatically incremented in stage 5) to represent that the pipeline has to stall while it waits for the data to be found. In this case, the incorrect branch prediction does not matter, because enough time is spent stalling due to the data miss that by the time it is done stalling, the correct instruction has already been found, and so no extra nop has to be inserted.
4. If the instruction in the memory stage is a store word instruction, checks to see if the instruction in the ALU stage is an r-type/branching instruction. If it is, and either reg1 or reg2 of the ALU instruction is the same as the src_reg for the store word instruction, then a nop must be inserted, as there will be a stall because the r-type/branching instruction

has to wait for the store word instruction to finish getting and storing the data from memory before it is able to use it. iplc_sim_trap_address is then called on the data address, and if it is a hit, nothing need be done unless we found that we had to insert a nop, in which case pipeline_cycles is incremented to account for the incorrect branch prediction. If the address misses in the cache, then the pipeline cycles is incremented by one less than the cache miss delay (since it is automatically incremented in stage 6) to represent that the pipeline has to stall while it waits for the data to be found. In this case, the incorrect branch prediction does not matter, because enough time is spent stalling due to the data miss that by the time it is done stalling, the correct instruction has already been found, and so no extra nop has to be inserted.

5. pipeline_cycles in incremented to represent that the pipeline is being pushed through another cycle.
6. Each instruction is advanced to the next stage in the pipeline, so the instruction that was in the writeback stage is pushed out, as it has finished processing.
7. The instruction in the fetch stage is made to be a nop to make room for a new instruction and get rid of the instruction that is still in there, which is the same as the instruction that is now in the decode stage.

iplc_sim_process_pipeline_rtype calls iplc_sim_push_pipeline_stage, then stores in the fetch stage of the pipeline the instruction data for a parsed r-type instruction, including the instruction type, the instruction address, the name of the instruction, reg1, reg2_or_constant, and dest_reg.

iplc_sim_process_pipeline_lw calls iplc_sim_push_pipeline_stage, then stores in the fetch stage of the pipeline the instruction data for a parsed lw instruction, including the instruction type, the instruction address, dest_reg, base_reg, and the data address.

iplc_sim_process_pipeline_sw calls iplc_sim_push_pipeline_stage, then stores in the fetch stage of the pipeline the instruction data for a parsed sw instruction, including the instruction type, the instruction address, src_reg, base_reg, and the data address.

iplc_sim_process_pipeline_branch calls iplc_sim_push_pipeline_stage, then stores in the fetch stage of the pipeline the instruction data for a parsed branching instruction, including the instruction type, the instruction address, the name of the instruction, reg1, and reg2.

iplc_sim_process_pipeline_jump calls iplc_sim_push_pipeline_stage, then stores in the fetch stage of the pipeline the instruction data for a parsed jump instruction, including the instruction type, the instruction address, and the name of the instruction.

iplc_sim_process_pipeline_syscall calls iplc_sim_push_pipeline_stage, then stores in the fetch stage of the pipeline the instruction data for a parsed syscall instruction, including the instruction type and the instruction address.

iplc_sim_process_pipeline_nop calls iplc_sim_push_pipeline_stage, then stores in the fetch stage of the pipeline the instruction data for a parsed nop instruction, including the instruction type and the instruction address.

Nick Roper
Glen Madsen

Results:

Main Data Table

Main Data Table	Index	Block Size	Associativity	Branch Prediction (0: Not Taken, 1: Taken)	Block Offset Bit	Cache Size (Bits)	Cache Misses	Cache Hits	Cache Miss Rate	Correct Branch Predictions	CPI		Additional Static Info	Number Of
Cache 1	7	1	1	0	2	7168	1390	34473	0.038759	5730	1.400426		Cache Accesses:	35863
Cache 2	6	1	2	0	2	7296	502	35361	0.013998	5739	1.17587		Total Instructions:	34753
Cache 3	5	1	4	0	2	7424	363	35500	0.010122	5744	1.14111		Total Branch Instructions:	7044
Cache 4	6	2	1	0	3	5632	1301	34562	0.036277	5726	1.377464			
Cache 5	5	2	2	0	3	5696	357	35506	0.009955	5736	1.138751			
Cache 6	4	2	4	0	3	5760	200	35663	0.005577	5746	1.099416			
Cache 7	6	4	1	0	4	9664	770	35093	0.021471	5729	1.243288			
Cache 8	5	4	2	0	4	9728	144	35719	0.004015	5749	1.084022			
Cache 9	4	4	4	0	4	9792	78	35785	0.002175	5749	1.068052			
Cache 10	7	1	1	1	2	7168	1390	34473	0.038759	1279	1.528501			
Cache 11	6	1	2	1	2	7296	502	35361	0.013998	1280	1.304175			
Cache 12	5	1	4	1	2	7424	363	35500	0.010122	1284	1.269444			
Cache 13	6	2	1	1	3	5632	1301	34562	0.036277	1290	1.505107			
Cache 14	5	2	2	1	3	5696	357	35506	0.009955	1290	1.266682			
Cache 15	4	2	4	1	3	5760	200	35663	0.005577	1290	1.227635			
Cache 16	6	4	1	1	4	9664	770	35093	0.021471	1291	1.37099			
Cache 17	5	4	2	1	4	9728	144	35719	0.004015	1291	1.212298			
Cache 18	4	4	4	1	4	9792	78	35785	0.002175	1291	1.196328			

The Main Data Table has <u>all</u> the relevant information for different cache configurations alongside some static info for every cache on the right hand side. Each cache configuration is named and reused in the following tables. Cache's 10-18 are simply Cache's 1-9 in configuration except for the branch prediction.

Number of Missed Branches

Number of Missed Branches	Correct Branch Predictions: Not Taken	Correct Branch Predictions: Taken	Taken + Not Taken	Difference from 7044 Total
Cache 1	5730	1279	7009	35
Cache 2	5739	1280	7019	25
Cache 3	5744	1284	7028	16
Cache 4	5726	1290	7016	28
Cache 5	5736	1290	7026	18
Cache 6	5746	1290	7036	8
Cache 7	5729	1291	7020	24
Cache 8	5749	1291	7040	4
Cache 9	5749	1291	7040	4

There are cases in the pipeline where directly after a branch instruction, the cache misses. This results in the branch being recorded as neither a correct or incorrect branch prediction. So in the first row, Cache 1's correct predictions for Not Taken or Taken (Cache 9) are summed, and the difference from the total branches found. When this occurs everything is pushed out of the pipeline and cycle count is incremented by 9, because of this, no prediction needs to occur since the delay to get the instruction is long enough to calculate the actual branch. This is the reason why correct branch predictions from Taken + Not Taken != the number of branches total While this is not critical, it is an interesting quirk.

Cache Miss Rate

Cache Miss Rate	Index	Block Size	Associativity	Cache Miss Rate
Cache 1	7	1	1	0.038759
Cache 2	6	1	2	0.013998
Cache 3	5	1	4	0.010122
Cache 4	6	2	1	0.036277
Cache 5	5	2	2	0.009955
Cache 6	4	2	4	0.005577
Cache 7	6	4	1	0.021471
Cache 8	5	4	2	0.004015
Cache 9	4	4	4	0.002175
Min, Max				Cache 9, Cache 1

The Cache Miss Rate simply takes the different cache configurations, diplays the miss rate, and finds the min/max. Since the type of branch prediction does not affect the miss rate, Caches 10-18 are the same as 1-9 respectively. (As shown in Main Data Table)

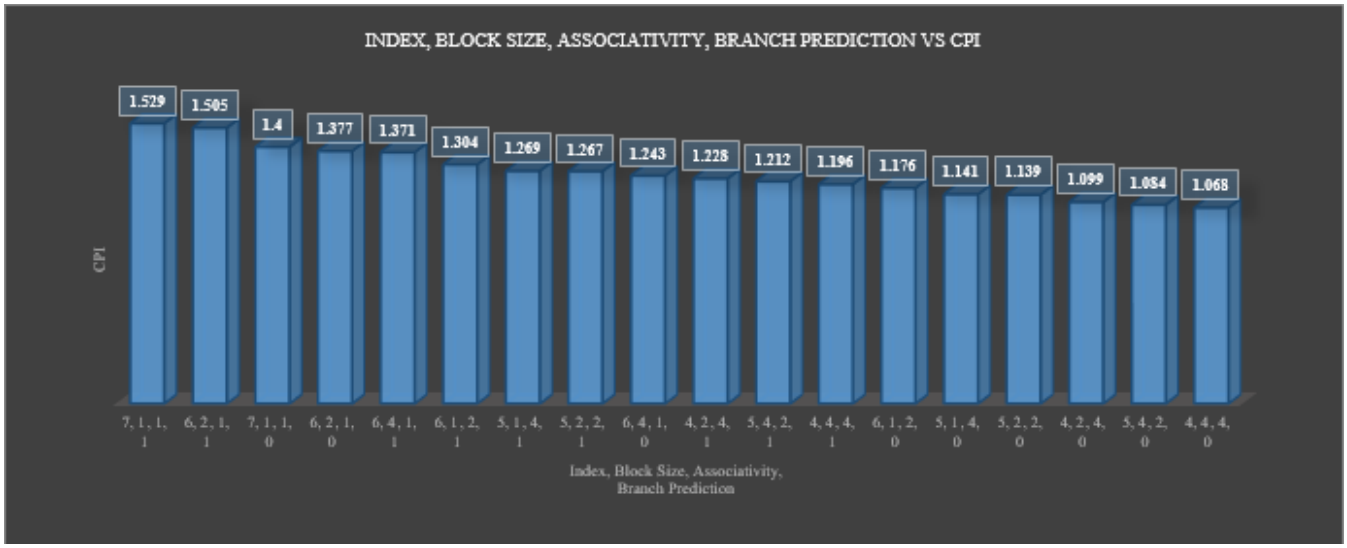
Nick Roper
Glen Madsen

Branch Prediction & CPI

Branch Prediction & CPI	Branch Prediction (0: Not Taken, 1: Taken)	CPI		Branch Prediction (0: Not Taken, 1: Taken)	CPI	Difference in CPI From Not Taken to Taken
Cache 1	0	1.400426	Cache 10	1	1.528501	-0.128075
Cache 2	0	1.17587	Cache 11	1	1.304175	-0.128305
Cache 3	0	1.14111	Cache 12	1	1.269444	-0.128334
Cache 4	0	1.377464	Cache 13	1	1.505107	-0.127643
Cache 5	0	1.138751	Cache 14	1	1.266682	-0.127931
Cache 6	0	1.099416	Cache 15	1	1.227635	-0.128219
Cache 7	0	1.243288	Cache 16	1	1.37099	-0.127702
Cache 8	0	1.084022	Cache 17	1	1.212298	-0.128276
Cache 9	0	1.068052	Cache 18	1	1.196328	-0.128276
Averages	0	1.1920443		1	1.320129	-0.12808456

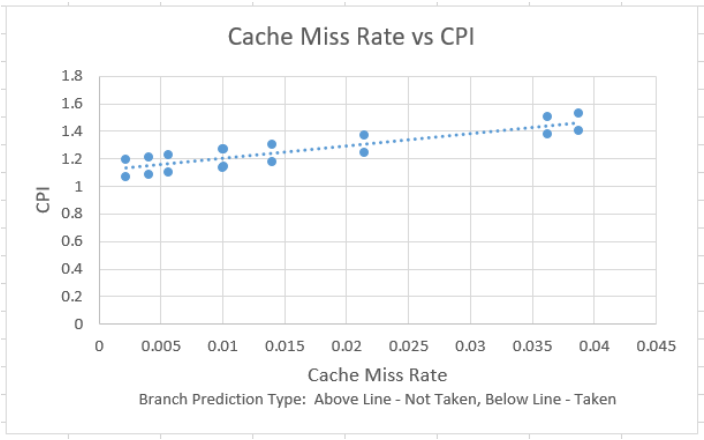
The branch prediction table simply records the CPI for every cache separated in the middle by type of branch prediction. It also shows the average for each type of branch prediction and calculates the difference between the two.

Cache Configuration vs CPI



This graph takes all the cache configurations and ranks them from greatest to least according to CPI (greatest = worst, lowest = best). The bottom numbers are the index, block size, associativity, and branch prediction type respectively, while the top number is the rounded CPI. So the best cache design and branch prediction type for CPI is on the right, and the worst is on the left

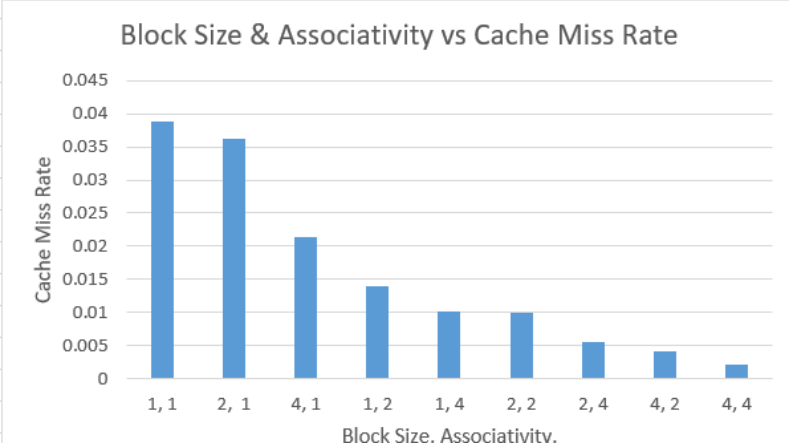
Cache Miss Rate vs CPI



This graph shows the correlation between a lower cache miss rate and a lower CPI, where the left end of the graph is the lowest CPI while the right is the highest. Also, as noted by the axis, the points below the line have a branch prediction type of Not Taken (0) while those above the line all have a type of Taken (1). So, in general, lower miss rate results in lower CPI and Not Taken prediction type results in a lower CPI as well.

Nick Roper
Glen Madsen

Block Size & Associativity vs Cache Miss Rate



The graph shows the trend of a decreasing Cache Miss Rate as both the Block Size and Associativity increase. The graph also illustrates how it in this case it is better to improve both together, not just one at the expense of the other. Overall the graph shows a size of 4 and associativity of 4 as having the lowest miss rate, while size of 1 and associativity of 1 having the worst.

Performance Evaluation:

Out of the statistics our simulator produces, one in particular is head and shoulders above the rest when it comes to measuring the performance of different cache and branch prediction configurations. This measurement being the CPI, or cycles per instruction. Since the CPI is effected by both the pipelines performance (See Branch Prediction & CPI) and the cache's performance (Cache Miss Rate vs CPI) it can give us an overall rating of cache performance. The same could be done with the number of cycles, but since the instruction set is the same for every configuration, it is easier to work with the equivalent (ratio-wise), and smaller, CPI numbers. Other valuable measurements include the cache miss rate and the correct branch predictions, but they each have their flaws. Namely, since the cache miss rate is independent of the branch prediction type (As shown in Main Data Table), it can't tell us about the overall performance of the simulator. In contrast, the number of correct branch predictions is influenced by the cache (See Number of Missed Branches) and does affect the overall CPI (See Branch Prediction & CPI), yet overall numbers are so similar it doesn't give a clear winner (See Number of Missed Branches for numbers). That leaves the CPI as a good measure of performance that takes into account both the cache and branch prediction (comparable to cycle count), but just what is the best cache configuration paired with a branch prediction type? Judging by the data illustrated in the Block Size & Associativity vs CPI graph, an increase in both block size and associativity at the expense of the index bit will decrease the cache miss rate. The best cache configuration in this regard is Cache 9, with an index of 4, block size of 4, and associativity of 4, it also happens to have the lowest recorded miss rate. With regards to the Branch Prediction Type, the Number of Missed Branches and Branch Prediction & CPI tables show clear results. For this instruction set, it is **Always** better predict that the branch is Not Taken. Based on the results of these two conclusions, the best cache should be the one with a configuration of index = 4, block size = 4, associativity = 4, and branch prediction type = 0 (Not Taken). Referring back to the Main Data Table, this conclusion is supported as Cache 9 has the lowest CPI of 1.068052. So, in conclusion, we used CPI to calculate overall performance due to it being equivalent to the cycle count (since the instruction set is constant), and the resulting cache/branch prediction

configuration that performed the best was Cache 9 in the Main Data Table, so it is the **best** configuration.

Group Participation:

Nick Roper:

- Helped Design the Cache and Designed/Implemented the Pipeline
- Wrote the General Overview and Pipeline Details sections of the Write-up
- Commented the majority of the program

Grade: 100% / 100%

Glen Madsen:

- Helped Design the Cache and Implemented it
- Analyzed the data and constructed graphical and tabular representations
- Wrote the Cache Details, Results, and Performance Evaluation parts of the Write-up

Grade: 100% / 100%