Hila Ben Abraham
Adina Raluca Stoica
Zheng Qin

SPECULATIVE SCHEDULING IN OUT-OF-ORDER PROCESSORS

1. INTRODUCTION

For this final project, we explored scheduling heuristics for selecting which instructions from the issue queue to execute next in an out-of-order processor. When discussing Dynamic Scheduling and the Issue Queue in class, we decided that the easiest way to schedule the next instruction to be executed is by age, the oldest ready instruction in the queue going first. For this project, we explored other possible scheduling heuristics and evaluated whether and how they improved the overall performance of the processor by increasing the number of instructions that could be dispatched in each cycle on an n-wide processor. The goal of this project was to find a heuristic that improves processor performance.

2. SCHEDULING HEURISTICS DESCRIPTION

We have explored various criteria for which instructions from the issue queue to schedule next. The baseline heuristic we used is the **AGE** heuristic, in which the oldest ready instructions are the ones that get scheduled next. In other words, this is a first-in-first-out framework.

The other baseline heuristic we looked at was the **LATENCY** heuristic, in which ready instructions that would take longer to execute are the ones that get scheduled to issue next. This turned out to be a very good heuristic for most of the benchmarks we used.

We examined criteria both independently and incrementally by creating a heuristic function that assigns points to various criteria and adds them up to compute a score, where the ready instruction that has the highest score is dispatched next.

We assigned a **sequence number** (snr) to each instruction in the ROB, such that the sequence number of each instruction corresponds to its position in the ROB (divided by the width of the machine).

We combined this sequence number with the LATENCY heuristic to create a score for a new **LATENCY_SNR** heuristic, where each instruction is assigned a score based on the sum of its sequence number and its execution time. When this didn't perform as well as we expected, and after receiving the presentation feedback which pointed out somewhat unexpected (but explainable) behavior in our preliminary experiments, we decided to increase the weight of the latency in the sum, and ran the code on 3 of the traces using weights of 2 and 3 respectively. We call these extra heuristics **LATENCY_SNR2** and **LATENCY_SNR3**, respectively.

For the first (and original) experiment, we computed all of our additional heuristics by adding up the sequence number to the various numbers of points that we decided to assign for each heuristic, but since the results were different from what we expected (we're explaining why in the results section), we decided to add two extra experiments to our project, somewhat at the last moment. The second experiment calculates the scores disregarding the sequence number, while the third sets the base score to the latency and computes the other heuristics on top of that base score.

The first two experimental calculation heuristics are explained together, while the third is explained separately.

The **BASE_SCORE** heuristic assigns scores to instructions based on instruction types. The **BASE_SCORE_SNR** heuristic assigns scores to instructions based on the sum of the sequence number and the number of points assigned by BASE_SCORE. For both experiments, branches are assigned a higher number of points than loads. We also assign points for multiplies and divides, but since these operations are rare on the benchmarks (see **Figure 1**) we cannot assess how important these point assignments are. All other operations are not assigned any points.

The (improved) **SCORE** heuristic assigns a relatively high number of points for mispredicted branches on top of the number of points already assigned by BASIC_SCORE. The (improved) **SCORE_SNR** assigns scores based on the sum of the sequence number and the number of points assigned by SCORE. The way we assign points for mispredicted branches assumes that we have access to an extra predictor that can predict whether a branch prediction we make will be correct, and that this branch-prediction correctness predictor is 100% accurate.

The **LOADS_FIRST** heuristic assigns a special number of points for priority loads (higher than the number of points for mispredicted branches) than the SCORE heuristic. All other instructions are assigned like in SCORE. Again, **LOADS_FIRST_SNR** sums this score with the sequence number to get its score. These new heuristics then calculate the score the same way as SCORE, using this new number of points for priority loads. Note that the heuristic name is somewhat of a misnomer, as there is no guarantee that loads will go first: they are just further prioritized compared to the SCORE heuristic.

The **LOAD_MISSED** heuristic assigns the original SCORE number of points for loads, as well as a special score for the loads that we assume will miss in the cache; **LOAD_MISSED_SNR** sums this number with the sequence number. These heuristics assume, by locality, that if a load missed in the cache, then the immediate next load will also miss, so they assign a special number of points for loads when the previous load was a missed load. Following the class project presentation, we tested the accuracy for this way of predicting cache misses, and the predictor turns out to be 74% accurate on average.

The **LOADS_FIRST_MISSED** heuristic computes its finals score by summing up the previous two heuristics (special score for loads plus special score for predicted misses). The **LOADS_FIRST_MISSED_SNR** heuristic is the sum of this the sequence number.

The third type of experiment we ran assumes that **BASE_SCORE_LAT** is the same as the LATENCY heuristic. We calculate the **SCORE_LAT** heuristic by adding the same number of points as in the previous heuristics for branches we predict (with 100% accuracy) as being mispredicted. We calculate the **LOADS_FIRST_LAT** heuristic by adding the same special number of points for loads as in the previous experiment to the loads' latency. We calculate the **LOAD_MISSED_LAT** score by adding the special number of points for missed loads to the latency of the loads that we predict will miss. To calculate the **LOADS_FIRST_MISSED_LAT** score we combine the previous two heuristics, that is we add points for loads and further add points for loads that we predict would miss.

3. SIMULATION DETAILS

Our simulation assumes a 4-wide processor for most tests, though we do conduct two extra experiments on a 2-wide and 8-wide machine respectively for one heuristic for some of the traces that produce the best results. This means we can run at most 4 instructions at a time, in general. For the implementation, we modified the Java-based cycle-level UNCAS simulator, which combines components of previous homework assignments. We built upon the Homework 5 code and implemented an Issue Queue with the scheduling heuristics that we wanted to explore.

Since we built upon the homework 5 Java simulator, the same restrictions apply as in the assignment: the model of the processor core is approximate in many ways, failing to capture many second-order effects, but it does capture the most important first-order effects of instruction execution. We modeled a reorder buffer (ROB) of finite size (of 256), but made sure that there are enough physical registers, instruction queue entries, load queue entries and store queue entries. The main difference from the homework assignment is that we also model the following (more realistic) execution unit restrictions: (1) no more than one load, (2) no more than one multiply, (3) no more than one divide, and (4) no more than one branch.

The latencies of the instructions we used are also different: we split out micro-ops into ALU operations, multiplies, divides, loads, stores, conditional branches, unconditional branches and other instructions. All instructions take one cycle, except for multiplies, which take 3, loads, which take 4 and divides, which take 20 cycles. The pipeline is simplified in that we modeled a relatively short pipeline.

To predict the branches, we use the gshare branch predictor, which was included in the UNCAS code, with 2^{16} counters, 2^{16} history bits for a total size of 16 KB. The branch prediction penalty (the time it takes to flush the pipeline and start again) is 5 cycles. We didn't change the size of the predictor since we get prediction rates of over 80% with this size. Our score calculation code assumes that we have a perfect misprediction predictor, so that every time we make a prediction we know with 100% accuracy whether or not that prediction will be accurate or not.

We are using an L1 cache as per the UNCAS2 simulator that we were given. We chose 2-way associative cache, with 4KB capacity and a block size of 64B. The cache miss penalty (which affects the execution latency of a load) is 20 cycles. We also conducted experiments with higher and lower cache miss penalties for sensitivity tests.

4. ALGORITHMIC IMPLEMENTATION

Our simulator is heavily based on UNCAS simulator that was provided to us. The simulator framework consists of calling the commit(), issue() and fetch_and_rename() functions. Our implementation includes adding 2 new issue functions, issue_Latency and issue_Score; we renamed the original issue function to issue_Age. In addition to the new issue functions, we've added additional helper functions (such as a function for sorting a copy of the ROB according to a score and functions to calculate the scores to assign the instructions) as well as execution unit restrictions into the original isUopReady function. We ended up making changes to / adding the following files: PerfSim, Heuristic and HeuristicsRunner.

Before getting into details about our new functions, we would like to point out a few important details. The first is the way new instructions are added to and removed from the ROB. Instructions are added to the end of the ROB during the fetch and rename stage, and removed from the head of the ROB during the commit stage. Due to the fact that instructions must be committed in-order, and in order to keep commit simple and to focus on the issue stage, we maintained this queue behavior. All of our issue manipulations work on copied instances of the ROB that were sorted according to the heuristic we explored. Second, as we learned in class, the issue stage includes select and wakeup operations. We find it important to mention that our project focused on the select part of the issue function, and maintained the original mechanism of wakeup (updating the scoreboard after issuing n instructions in an n-wide processor).

To be more specific about our implementation, we'll now explain our code changes in more detail. We started by merging the cache structures from UNCAS2 into the simulator. The original UNCAS simulator already included a branch predictor that set mispredictedBranch_f to true for each mispredicted branch. We decided to use this structure and assume that our processor has a branch prediction correctness predictor.

During the fetch and rename stage, we call a new function named calculateInitialScore that initiates the instruction score according to the explored heuristic. We used another function named calculateBasicScore to initiate the baseline score that applied to all heuristics used in issue_Score(), unless we are using the base score = latency heuristic, in which case we use the calculateBasicScoreSpecial function. To be specific, calculateInitialScore sets the initial score according to the instruction type and whether or not a branch is mispredicted.

The baseline scores we used for each instruction type are: 2 points for loads, 1 point for multiplies, 3 points to divides and branches. In addition, we also added 10 points to every branch that we knew would be mispredicted before execution. We also used 20 points for loads in the LOADS_FIRST* heuristic and 20 points for missed loads for the LOADS_FIRST_MISSED* heuristic. We decided on these numbers to be the baseline score numbers according to a set of experiments and in relation to the instruction latencies. However, we had the limitation of a very long simulation time that forced us to quickly decide on basic score numbers and to continue with all other heuristic experiments.

As described before, we added 2 new issue functions: issue_Latency and issue_Score. We changed the original issue() function to select which function to use based on the heuristic type (acceptable heuristic types are defined in Heuristic.java).

In order to maintain the way the original issue_Age function works, we created a sort function, mysort, that sorts an array in descending order according to its second column. Thus, if we copied the ROB entry fields that we want to sort (latency and score respectively) to the second column of an array, finalArr, that has the original ordering of the ROB entries on the first column (sequence of consecutive numbers from 0 to number of entries -1) and we used mysort to sort it, the first column will be reordered according to the sorted second column. After the columns are sorted thus, instead of issuing instructions in the order they are in the ROB, we issue them in the order of finalArr[order][0].

We used the issue_Latency function to run the latency heuristic. We then created a finalArr that has the second column equal to the latency of the corresponding ROB instruction and used

mysort to sort it. In case of more than one instruction have the same latency, the sorting set the older instruction to be first. Right after the sorting, we went over the ROB in the correct order and issued the first n ready instructions (in n wide processor). Instruction readiness is determined by the isUopReady function. We used the framework of the original isUopReady function and added the execution unit restriction to this function. After an instruction is found to be ready to issue, we update the corresponded entry in the original ROB with the issue cycle, done cycle and issued flag. In case the instruction is a load-miss in cache, we add the cache-miss penalty to the done cycle. At this point we also add the mispredicted branch penalty to the fetchReady variable if the instruction is a mispredicted branch.

The issue_Score function works similarly to the issue_latency function. The only difference is that the ordering in this function is based on the instruction score instead of the instruction latency. As in the issue_Latency function, in case of more than one instruction has the same score, we chose the older one to be first.

We tested all the previously described heuristics. To set which of the base, *_SNR or *_LAT heuristics we're using, we added a heurType variable. When it's set to 1, we run the base experiments (just adding points), when it's set to 2, we run the *_SNR experiments (adding points to the sequence number), and when it's set to 3, we run the *_LAT experiments (adding points to the latency). We compute the sequence number by running through the ROB and incrementing the sequence number. We also added the latency multiplier for the LATENCY_SNR experiment, and this value can be set to any number (we used 1,2 and 3).

Our code is executed using the HeuristicsRunner file. This file runs through all the traces and simulates each experiment, unless set otherwise. It either runs the base experiment, the *_SNR experiment or the *_LAT experiment, by setting heurType to 1, 2 or 3 respectively. You can also increase the multiplier for the latency for the LATENCY_SNR heuristic.

5. SIMULATION SANITY CHECK

In order to verify our work, we performed sanity tests to validate the correctness of our new issue functions. We used the debugger to follow the fetch and issue cycles for a few of the instructions. We compared the observed numbers in each cycle with the numbers we expect to find in the explored heuristic. We also verified that not taken branches created a delay in the fetch cycle for the following instructions.

The main difference between the original age heuristic and the latency and score heuristics is the ROB sorting function. Because our whole idea was based on the correctness of the sort, we built a separate java module and verified that the sort function is working correctly on different input arrays.

Another important check was to make sure that the instruction commit is performed in-order. We used the simulator printout to validate that by observing the commit cycle.

6. PRELIMINARY DATA ANALYSIS

We decided to use all (8) of the large (50M or 100M) traces we were provided with in order to test our heuristics. These traces have the instruction type distributions as in **Figure 1**. It is interesting to note that multiplies and divides are very rare in the traces (appear almost exclusively in the art-100M trace), so very few traces will show any change by prioritizing those types of instructions. Therefore, we cannot evaluate whether assigning points for multiplies or divides makes any difference.

It is also important to see that loads are the most frequently occurring operation of the ones we decided to look at. Because of the execution unit restrictions that we impose, issuing loads fast will help reduce the chances that loads will compete for the same port in further cycles. For the same reason, issuing conditional branches fast is also a good idea.

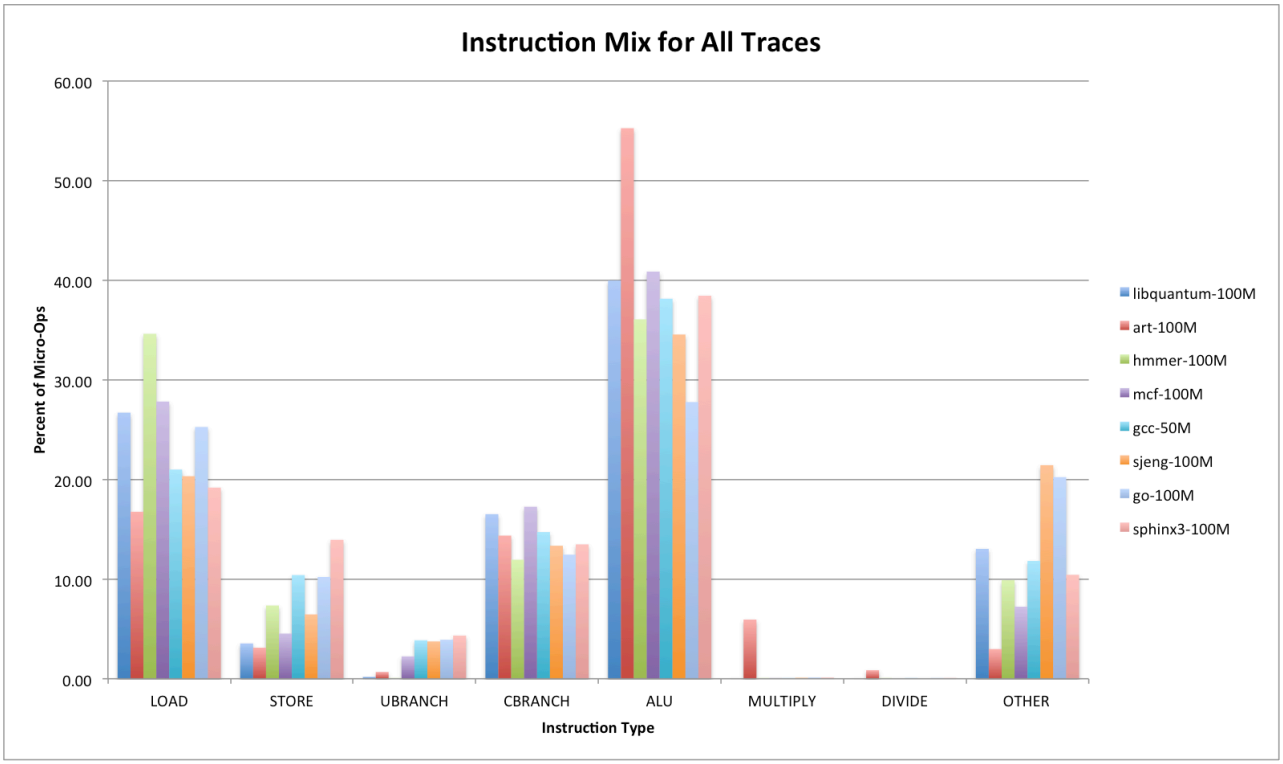

Figure 1: The Instruction Distribution by Instruction Type

7. EXPERIMENTAL RESULTS AND ANALYSIS

We used the simulator to measure the effects of the different scheduling heuristics described in section 2 on the program. On our benchmarks, the preliminary results from running on a 4-wide machine show that scheduling the instructions that would take longer to complete first works better than scheduling the oldest instruction first.

AGE vs. LATENCY vs. LATENCY_SNR (with weights 1, 2 and 3)

For this experiment, we compare the age heuristic to the latency heuristic as well as to the latency heuristic combined with the sequence number, with weights 1, 2 and 3 for the latency. These latter two experiments were only run subsequently on 3 of the traces, so the graph shows only these 3 traces.

As can be seen in the graph in **Figure 2**, adding the sequence number to the latency of the instruction only reduces its IPC. The graph shows the relative IPCs for the instructions with the AGE being the baseline (relative IPC = 1) for 3 trace files, as well as the average of each heuristic over these traces. While LATENCY does take into account the fact that between two ready instructions with the same latency the one that is older goes first, the mechanics for the LATENCY_SNR heuristic is different (regardless of the latency component weight). The fact that the number of cycles an instruction has been in the ROB for can be orders of magnitude greater than the maximum latency for any of the new instructions added means that adding the sequence number and the latency gives a lot higher preference to older instructions than to long instruction.

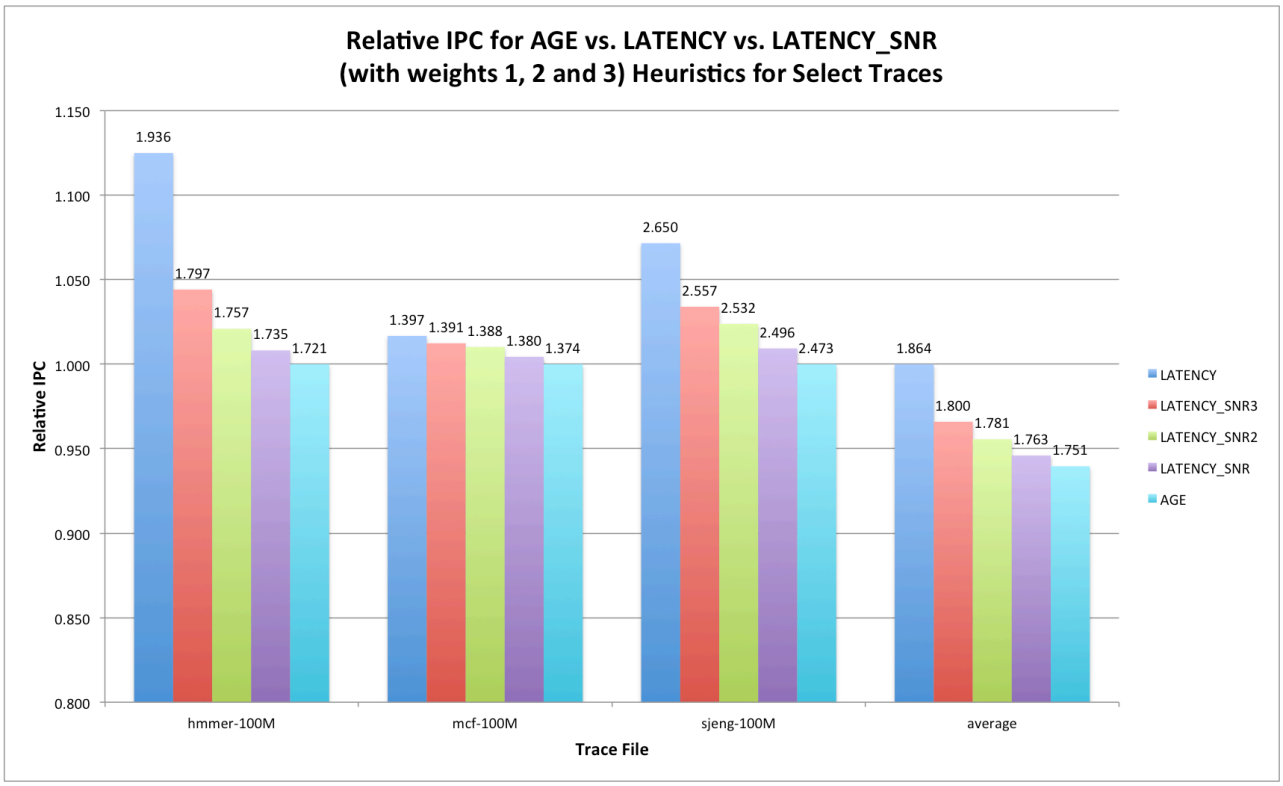

Figure 2: The relative IPC for AGE, LATENCY and different combinations of latency and sequence number. The absolute IPC for each heuristic is above the bars.

This is fundamentally different from choosing the oldest instruction with the same score to issue first as in the LATENCY heuristic (see **Table 1** for an example). As expected, increasing the weight of the latency in the calculation get us closer to the LATENCY heuristic.

Instruction	Sequence Number	Ready?	Latency	LATENCY		LATENCY_SNR	
				Score	Issues?	Score	Issues?
sub	7	**Yes**	1	1	**Yes**	8	**Yes**
add	6	**Yes**	1	1	No	7	**Yes**
:	:	:	:	:	:	:	:
:	:	No	:	:	No	:	No
:	:	:	:	:	:	:	:
ld	1	**Yes**	4	4	**Yes**	5	No

Table 1: Example for difference between LATENCY and LATENCY_SNR issue orders. Using the LATENCY heuristic (processor width = 2), we issue the div and the sub first. Using the LATENCY_SNR heuristic, we issue the add and the sub first because they've been in the ROB longer.

Heuristics Comparison

TRACE # / Heuristic	1	2	3	4	5	6	7	8	Mean
LATENCY	1.006	1.000	1.125	1.017	1.067	1.072	1.054	1.065	1.053
BASE_SCORE_SNR	1.001	1.000	1.007	1.001	1.008	1.004	1.011	1.007	1.005
BASE_SCORE	1.006	1.000	1.125	1.016	1.068	1.072	1.054	1.065	1.053
BASE_SCORE_LAT	1.006	1.000	1.125	1.017	1.067	1.072	1.054	1.065	1.053
SCORE_SNR	1.001	1.019	1.007	1.002	1.021	1.015	1.015	1.011	1.011
SCORE	1.006	1.019	1.126	1.017	1.082	1.084	1.059	1.068	1.059
SCORE_LAT	1.006	1.019	1.126	1.017	1.083	1.085	1.060	1.069	1.059
LOADS_FIRST_SNR	1.006	1.019	1.078	1.015	1.070	1.063	1.053	1.056	1.046
LOADS_FIRST	1.006	1.019	1.126	1.017	1.082	1.084	1.059	1.068	1.059
LOADS_FIRST_LAT	0.993	0.970	1.126	1.019	1.090	1.091	1.061	1.079	1.057
LOAD_MISSED_SNR	0.999	1.019	1.016	1.015	1.025	1.019	1.021	1.016	1.017
LOAD_MISSED	1.006	1.019	1.127	1.025	1.083	1.085	1.061	1.068	1.061
LOAD_MISSED_LAT	1.006	1.019	1.127	1.026	1.084	1.086	1.062	1.069	1.062
LOADS_FIRST_MISSED_SNR	0.614	0.577	0.872	0.864	0.735	0.675	0.850	0.721	0.757
LOADS_FIRST_MISSED	1.006	1.019	1.127	1.025	1.083	1.085	1.061	1.068	1.061
LOADS_FIRST_MISSED_LAT	0.989	0.970	1.128	1.028	1.091	1.092	1.063	1.079	1.059

Table 2: Relative IPC (with AGE=1) comparison of different heuristics and computations. The best performing heuristic is highlighted in green (darker shade of green if it beats LATENCY). Since the basic and *_LAT heuristics perform on average the same, but, since *_LAT falls under the AGE heuristic for some traces, we decided to report best results using the basis score.

Our original heuristics are the *_SNR heuristics, since we thought that issuing very old instructions first would be a good feature. However, when this didn't produce the expected results, we decided to run the base heuristics (just the scores, without adding the sequence number) as well as the *_LAT heuristics, where the original base score is the instruction's latitude. This section compares the results obtained by running all of these heuristics. In particular, we are interested in comparing how well they do with respect to the LATENCY and the AGE heuristic. **Table 2** shows the relative IPCs vs. the AGE of each heuristic for all traces, as well as the average relative IPC. Since, on average, both *_LAT and base perform the same, but *_LAT sometimes falls under the AGE heuristic for some traces, we decided to report final results for the basis.

Best Results

The best results we were able to obtain are shown in **Figure 3**. The figure shows the relative IPC with respect to the IPC of the age heuristic, as well as the LATENCY, BASE_SCORE, SCORE, LOADS_FIRST, LOADS_MISSED and LOADS_FIRST_MISSED heuristics.

TRACE # / Heuristic	1	2	3	4	5	6	7	8	Mean	Speedup
AGE	3.255	3.371	1.721	1.374	2.275	2.473	1.409	2.569	2.075	1.000
LATENCY	3.275	3.371	1.936	1.397	2.427	2.650	1.485	2.737	2.184	1.053
BASE_SCORE	3.275	3.371	1.936	1.396	2.429	2.652	1.485	2.735	2.184	1.053
SCORE	3.275	3.436	1.937	1.397	2.462	2.681	1.492	2.744	2.196	1.059
LOADS_FIRST	3.275	3.436	1.937	1.397	2.462	2.681	1.492	2.744	2.196	1.059
LOAD_MISSED	3.275	3.436	1.94	1.409	2.464	2.682	1.495	2.744	2.202	**1.061**
LOADS_FIRST_MISSED	3.275	3.436	1.94	1.409	2.464	2.682	1.495	2.744	2.202	**1.061**

Table 3: The best performing heuristics and their absolute IPCs. The mean absolute IPC is shown in the 10th column and the average speedup is shown in the last. LOAD_MISSED and LOADS_FIRST_MISSED perform marginally better than other heuristics, but 6.1% better than AGE.

Table 3 shows the absolute IPCs as well as the average absolute IPC and the speedup over the AGE heuristic. The biggest average speedup we obtain is by running either the LOADS_MISSED or the LOADS_FIRST_MISSED heuristics, and this speedup is of 6.1% versus a speedup of only 5.3% for the LATENCY heuristic. For the traces that we have, running either of these heuristics gives the same IPC, however the fact that using the other scores *_SNR and *_LAT don't produce identical results, we still believe that these two heuristics can produce different results on a different trace, which is why we are still showing them as different heuristics.

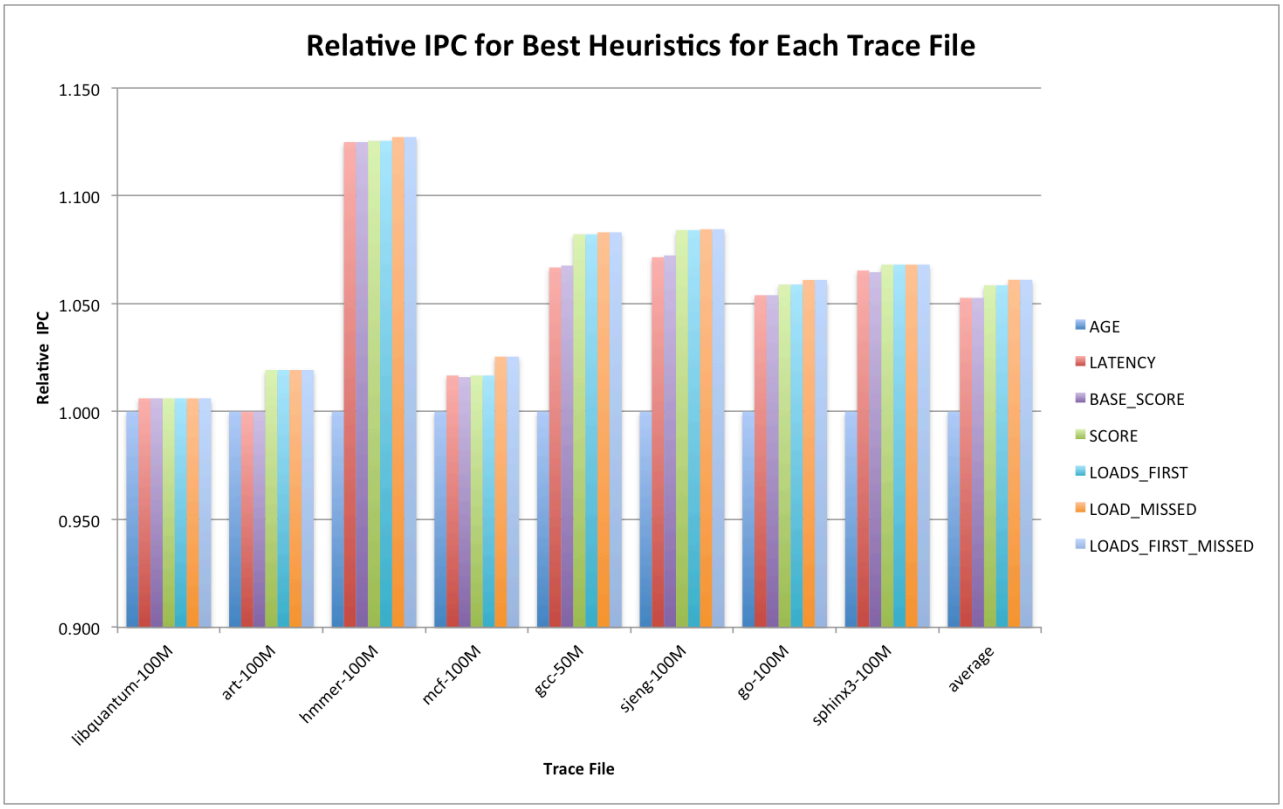

Figure 3: The Best Results. Best performing heuristics were selected and their relative IPC to the AGE heuristic is shown. LOAD_MISSED and LOADS_FIRST_MISSED perform consistently better than other heuristics, if only marginally.

8. SENSITIVITY STUDY

We conducted sensitivity studies to test how well our algorithm performs under different settings from the experimental ones. We performed experiments by changing the width of the processor and changing the branch misprediction penalty. We are going to show these experiments on 3 of the traces.

Our idea doesn't require many resources (see the actual implementation section), so although the speedup is small, there isn't a lot else to do with the resources we require.

Processor Widths

The LOADS_FIRST experiment was conducted using a processor of different widths. As expected, the IPC increases as the processor width increases. However, increasing the width from 4 to 8 has less of an effect in speedup as increasing the processor width from 2 to 4. The results are shown in **Figure 4**.

Branch Misprediction Penalties

Because we selected a score for branches that is higher than the score for other operations with respect to the latencies, and because we added an extra 10 points to that for mispredicted branches, we wanted to test how well we would do with different misprediction penalties, expecting that smaller penalties should give better results. We ran the same heuristic, LOADS_FIRST, and we tested with branch misprediction penalties of 2, 5 and 10. The results are in **Figure 5**.

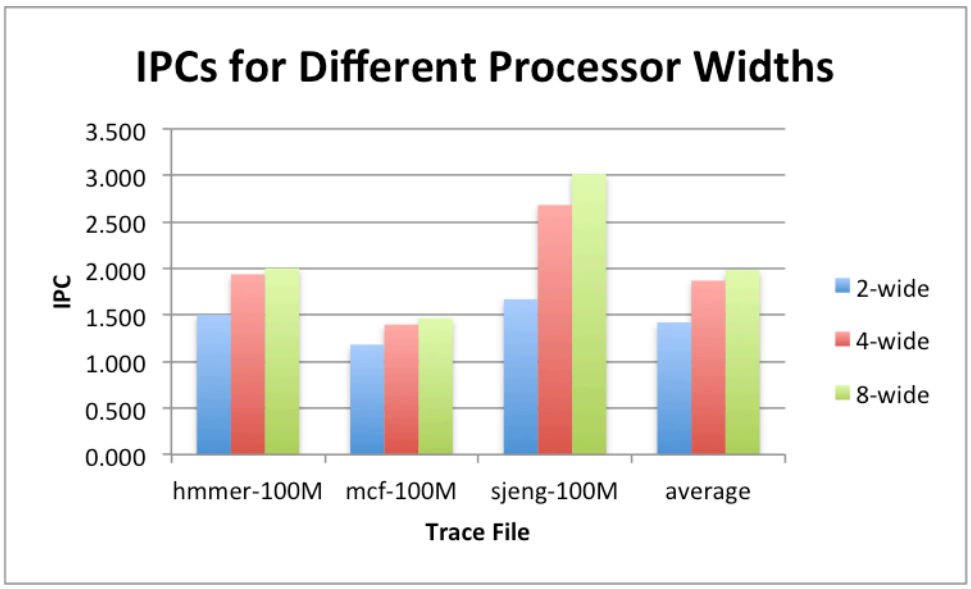

Figure 4: Effects of modifying the processor width on IPC for heuristic LOADS_FIRST

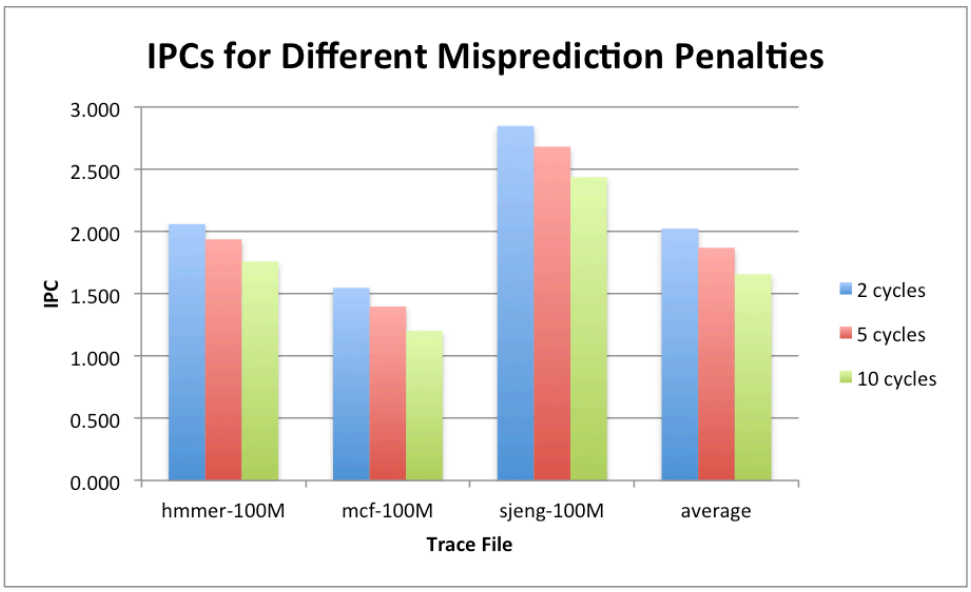

Figure 5: Effects of modifying the branch misprediction penalty on IPC for heuristic LOADS_FIRST

ACTUAL IMPLEMENTATION

As we learned in class, the processor uses the issue queue to decide whether the instruction is ready to be issued. This decision is performed by comparing the ready bits of the input and output registers using an AND gate. Since our knowledge in hardware implementation is pretty limited, we can't discuss this implementation it in detail. However, we'll try to explore whether there is any new infrastructure that should be added to implement our idea, and what are the costs and power requirements for our implementation.

Due to the fact that we perform the select part in the issue using a heuristic other than age, we assume that an additional layer of logic is required to save the instruction score (latency can also be saved in this new logic structure). In order to select the ready and higher scored instruction, the hardware needs to perform a comparison between all the ROB entries.

As we learned in class, as opposed to the sequential simulator, the hardware performs comparisons in parallel once in every cycle. This parallel comparison will probably require additional lines connected between the new layer of logic and the AND gate. It might be that such a comparison propagation will take longer than the original clock cycle, and therefore it might increase the clock cycle, and by doing so, decrease the clock rate. In addition, this new layer of logic will probably increase the processor power consumption.

Our solution uses a perfect branch predictor correctness prediction, so the hardware to implement such a prediction also requires an additional logic structure. Since this additional structure doesn't have extraordinary benefits, it could be bypassed completely and the BASE_SCORE heuristic can be used as the SCORE.

Our solution doesn't use the cache to predict cache misses, and the processor will probably use cache anyway, so we don't think that additional logic structure is specially required in that respect. However, we do need a small structure to count whether a previous load was a hit or a miss.

We don't think that the additional structures needed for our solution demand a significant increase in area, but the solution will probably cause an increase in power consumption and might reduce the clock rate. Generally speaking, the successful heuristics showed an increase of approximately than 6% in IPC, so we think that, as long as it's not trumped by a high increase in the clock rate, the cost may be worth the changes.

9. CONCLUSIONS

There are a few questions we wanted to answer in this project. The main goal was to find a heuristic that would do better than the default AGE heuristic for scheduling instructions in an out of order processor. Although our best heuristics, LOADS_MISSED and LOADS_FIRST_MISSED obtain 6.1% speedup versus the AGE heuristic, the very simple LATENCY heuristic performs almost as well, getting a 5.3% improvement on average. Is this 0.7% improvement enough to say our idea is better? Maybe, or maybe not, especially since there are tweaks that we didn't have time to perform in assigning the scores to the instructions. It also depends on whether or not we decide to add the prediction correctness predictor, since without it we will probably do marginally worse than we currently do, so the gap will become smaller than 0.7%. This score can be better improved

by having a better load-miss predictor (though that may add overhead) or by assigning better original scores to the instructions. We believe that this 6.1% improvement is something to consider, as long as the decrease in clock speed doesn't trump it, especially as it comes at a relatively small cost.

The second, side question we intended to answer relates to having a branch predictor correctness predictor. In our simulator, we had a 100% accurate predictor at no cost, which is not an accurate assumption to make. The performance increase gained by adding this predictor is reflected in the relative speedup between the BASE_SCORE and the SCORE heuristics, which is a 0.5% speedup on average. We would posit that this is not enough to warrant the addition of this structure.

10. RELATED WORKS

While we didn't find articles that are similar to what we've done, such as implementing a SCORE heuristic, we found that other instruction issue procedures have been implemented and resulted in improvement on processor's performance. Shlomo Weiss and James E. Smith use the CRAY-1 scalar architecture issue logic, which forces instructions to issue strictly in program order similar to our AGE heuristic, as a baseline algorithm for comparison in their article, *Instruction Issue Logic For Pipelines Supercomputers*. They compared three issue algorithms: Tomasulo's algorithm, Thornton's "scoreboard" algorithm and an issue method using a Direct Tag Search (DTS), and computed their effects on IPC. They get speedups of 1.58, 1.28 and 1.38 respectively. Although we do not get as much of a speedup as these algorithms, their implementation concerns more of a hardware change than ours.

Sendag, R.; Yi, J.J.; Chuang, Peng-fei address the concept of a branch misprediction predictor in the article *Branch Misprediction Prediction: Complementary Branch Predictors*. Their misprediction predictor, coupled with a conventional predictor, achieves the same prediction accuracy as a conventional branch predictor that is 4 to 16 times larger, but without significantly increasing the overall complexity. We assume we have a similar structure for our branch prediction correctness predictor.

11. CREDITS

Hila Ben Abraham

- Wrote most of the code: execution unit restrictions and heuristics
- PowerPoint presentation slides
- Performed the sanity check
- Final write-up: sections 4 and 5

Adina Raluca Stoica

- Made some revisions to the program code: new heuristics, heuristics flags
- Basic outline for the PowerPoint presentation
- Ran most of the experiment code (on the CEC server / lab server)
- Result graphs
- Final write-up: sections 1-3, 8-10, second half of section 11
- *Submitted the SVN repository*

Zheng Qin

- Wrote the Latency heuristic and ran a ROB size experiment which we used to decide the best ROB size for our experiments
- Preliminary result graphs
- Final write-up: first half of section 11

The main idea for the project came from class, when we were discussing out of order processors. The experiment design and analysis of results are our work. We would like to thank Anne for suggesting that we also add a cache to make our experiments more significant.