Instruction Scheduler - CS415 - Compilers Project 1

Daniel Pattathil - dp865 - 170004119

March 10, 2019

Contents

Background on Instruction Scheduling

The main focus of the project was to provide a program that would allow for the rescheduling of ILOC instructions in order to lower overall cycle counts in execution. The designed program would be able to take in a serious of ILOC instructions using a particular set of operations {*add, sub, mult, div, load, loadI, loadAI, loadAO, store, storeAI, storeAO, and outputAI*}. These programs would have specific instructions using registers and operations leading to particular outputs that would be verified by the use of the ILOC Simulator *sim*. The scheduler.py program uses Python3 to implement three different heuristics for the scheduling itself and outputs the edited list of ILOC instructions for comparison and testing. The ideal scenario is having a set of heuristics that maintain the integrity of the initial ILOC code and hopefully manages to cut down on the amount of cycles required the perform the basic block operations.

1 Understanding the Implementation

The main task in implementing the entire program was to determine the dependencies between lines of ILOC code and be able to represent them in a form similar to having the dependence graphs/trees referenced in class. In order to approach this problem, the first goal was to determine an algorithm that would perform the same analysis of true and anti dependency relationships while iterating through the code.

1.1 Helper Methods and Classes

In order to tell which registers or values could be related, the helper methods leftparams() and rightparams() were used to set up a USE/READ and DEF/WRITE set for each instruction. For the large majority of the instructions, the registers and values to the left of the => symbol would tend to be part of the USE/WRITE set of the instruction and vice versa for the right side and DEF/WRITE set. There were a few specific cases that needed to be handled, such as when there would be multiple register values that could be on the USE/left side such as in any arithmetic operation. In order to group these more easily, there were sets for arithmetic operations, loads, and stores. Then, instructions were parsed and stored into objects from a defined "instr" class which held operation, field, and cycle counts for each line. Before going into the main loop, there were dictionaries that were declared to keep track of leaf nodes, parents for a particular node, and children for a particular node.

1.2 Dependence Generation in the Main Loop

After having all of these helper classes and methods, the actual processing of each ILOC file would happen in a large for-loop. In order to set up a dependency, the program would add an instruction j that is a child of i to the children set using $children[i].add(j)$ and then update the parents set with $parents[j].add(i)$. The beginning would involve setting up dependencies between the initial instruction that loads a memory value into r0 and any memory instructions that used r0. Then, there would be additional dependencies added so that the integrity

of the output would be maintained. In addition to this, it was ensured that loads would not cross stores and similarly for stores to cross outputs, because the program was not meant to actually calculate certain memory accesses.

NOTE: This was not the case for loadAI and outputAI which specifically gave tuples/paired values so that they could be properly compared.

Ex): leftparams(loadAI_inst) == (r0, 8).

After this, there was the internal loop that would check for true dependencies based on a particular type of instruction. For this loop, the current instruction i would check for similarity between items in its DEF/WRITE set and the sequential instruction j's USE/READ set, for all $n == length(inst) > j > i$. In the case that there was an intersection, the dependency would be created with the parents and children dict's. Then, there was a similar internal loop that checked for anti dependencies by comparing the same values, just in the case that $0 <= j < i$. For both of these loops, there were break conditions to cover the issues of incorrect dependencies that could arise from defining a register or memory location multiple times. After all of these loops, if the $parents[i]$ was equivalent to an empty set, we would know that the node i could be considered a leaf node and would be added to the set with $leaves.add(i)$.

1.3 Using the Dependency Graph to Schedule

Now that we have the leaves, children, and parents sets, we are able to implement the algorithm that orders the different instructions based on our dependencies and selected heuristic. The algorithm that we are following is fairly simple and will stop when there are no longer any unscheduled nodes. We currently have a set of leaves that came from our initial dependence generation and we know that they have no parents, so they are free to be queued for our final order. At this point, we reference our heuristic to know which node as referenced by the $chooseNode()$ method. Then, with this node, we add it as the instruction for the current cycle. Finally, we check whether nodes in the active list are completed, if so, we can remove them and release their children so that they can possibly be added to the leaves queue. There are occasionally cycles of "no op" that still will check if anything can be removed. Finally, the final order gets replaced with the respective instructions from the text input before all of the lines get written to an output file.

2 Heuristics Breakdown

2.1 Longest Latency-Weighted Path to Root $(-a)$

The entire basis of this scheduling heuristic is based on the calculated Weighted Path for each instruction node. These values are stored in a dict called h which is a set of paired values that updates as the loop calculates the Longest Path for each node to its root. The first part of this calculation is to find the root or roots of the graph. The root refers to any instruction that has no children when accessing the $children[i]$ dict and all the nodes that fall under this category are added to a queue that will be used to update path values. The queue functions similarly to BFS and as a result will cover all of the nodes. By having a queue with all the possible roots, we make sure to definitely hit all of the parents and

children in the tree. This implementation makes the most logical sense out of all mentioned heuristics, because it first calculates all the longest length paths. This is objectively useful, because it is a definite feature that gives an absolute answer as to which nodes should be treated first. It also shows in the data as the consistently lowest cycle count, which suggests that it is truly most optimal in our benchmarks.

2.2 Highest Latency Instructions First $(-b)$

The difference in this heuristic between $-a$ is that the node chosen follows the cycle count. In order to decide the correct node, the $chooseNode()$ implementation for $-b$ cycles through the leaves set and checks for the instruction with the largest cycle count value. In this implementation, when there are ties, it is not dealt with in an intelligent way, the chosen node comes from the first node with the highest count when iterating through the set. This should also be a fairly strong heuristic, because it is designed in a way that long latency instructions are put into the active state first. This would hopefully avoid cases where high-latency instructions slow down the overall run-time of the program. However, as seen in a few of the benchmarks, namely bench09 and bench10, this ends up being slower than lowest latency. This is generally going to be caused by the scheduler putting lower priority instructions (according to our longest weighted path values) into the active set before the higher priority instructions with lower cycle counts.

2.3 Lowest Latency Instructions First $(-c)$

The difference in this heuristic between $-a$ is that the node chosen follows the cycle count and that it is the min val compared to $-b$. In order to decide the correct node, the $chooseNode()$ implementation for $-b$ cycles through the leaves set and checks for the instruction with the smallest cycle count value. In this implementation, when there are ties, it is not dealt with in an intelligent way, the chosen node comes from the first node with the lowest count when iterating through the set. This tends to occasionally be a fairly strong heuristic and tends to not lag too far behind the other cycle counts in most of the benchmarks. I felt that in these cases where the instruction counts are not terribly high, it could end up being that finishing instructions quickly could lead to faster overall processing of the code. In Figure 1, it does seem to depict this heuristic as generally slower, but it not by a huge margin and occasionally it performs better than the inverse being $-b$.

3 Results

3.1 Graphs

For the particular data that was collected, here are the following graphs to respectively show Total Cycles per Benchmark and Cycles Saved per Benchmark:

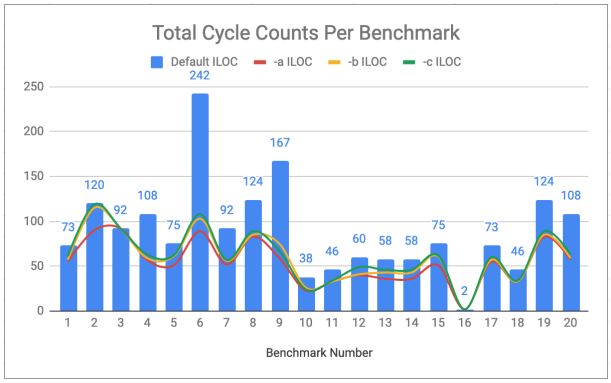

Figure 1: Cycle Counts Per Benchmark

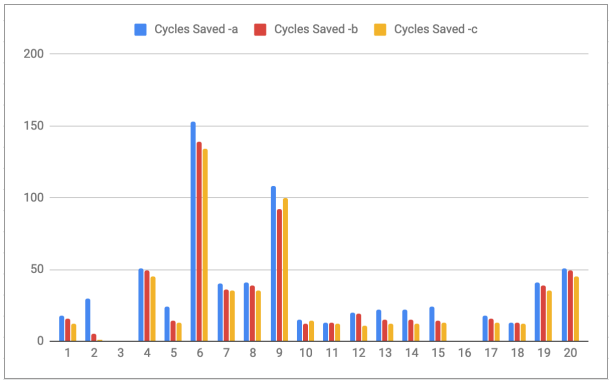

Figure 2: Cycles Saved Per Benchmark

3.2 Discussion - Advantages and Disadvantages

Now, we can analyze the actual graphs and data that was attributed to the testing of benchmarks from this project. The main analysis that helped in checking that the heuristic discussion from Section 2 was indeed correct. It ended up being that $-a$ was consistently the best performer. As discussed in class, we can find the maximum cycle count stemming from the dependence graph. That number itself is the minimum amount of cycles required for the entire execution according to the graph. It was good to see that this was true in the actual results with a good amount of clarity. There is however the cost of actually calculating the values for each node and storing those values to use for the scheduling. This is rather different from the heuristics used in $-b$ and $-c$ where we do not actually need to process all of the nodes and have the deeper optimization. When we use these basic latency measures, it ends up being much less information to store and less pre-processing before running the scheduling algorithm itself. When looking at Figure 2 and considering the amount of cycles saved, it becomes rather clear that the amount of cycles saved only is very significantly different in a few bench marks. Bench02 is the only case of a big disparity in cycles saved, and in the rets of the test cases, the difference tended to be less than 10 cycles. I believe in a general case, $-c$ is

a similar enough heuristic to $-b$ to put it up for consideration. However, the longer a program gets, the performance differences are likely to become more disparate and it would be a larger question of $-a$ compared to $-b$.

3.3 Collected Data

Cycle Counts:	Default ILOC	-a ILOC	-b ILOC	-c ILOC	Saved -a	Saved -b	Saved -c
bench01	73	55	57	61	18	16	12
bench02	120	90	115	119	30	5	1
bench03	92	92	92	92	0	0	0
bench04	108	57	59	63	51	49	45
bench05	75	51	61	62	24	14	13
bench06	242	89	103	108	153	139	134
bench07	92	52	56	57	40	36	35
bench08	124	83	85	89	41	39	35
bench09	167	59	75	67	108	92	100
bench10	38	23	26	24	15	12	14
bench11	46	33	33	34	13	13	12
bench12	60	40	41	49	20	19	11
bench13	58	36	43	46	22	15	12
bench14	58	36	43	46	22	15	12
bench15	75	51	61	62	24	14	13
bench16	2	2	2	2	0	0	0
bench17	73	55	57	60	18	16	13
bench18	46	33	33	34	13	13	12
bench19	124	83	85	89	41	39	35
bench20	108	57	59	63	51	49	45

Figure 3: All Recorded Data

4 Additional Details

The project report and analysis were seriously assisted with the use of project packaged bash scripts that were able to create the scheduled *.iloc files and then compare them easily. This was how the correct output was verified for all of the benchmarks. In addition to this, there was a bit of *.txt parsing from the reports that came out of the comparisons, using this, the data available in the table was collected and able to be analyzed.