Lab 3 Questionnaire

Comp 412, Fall 2019

Name:	**Dejan Grubisic**
Net ID:	**dx4**
Date Submitted:	**Dec. 5**

Implementation Discussion

1. Briefly describe the key design decisions you made in building your scheduler. Do **not** include a detailed description of the data structures, classes, fields, and methods that you implemented.

   In the core of building scheduler in this project is dependence graph data structure, that contains all information about each instruction and their dependence to previous instructions. Direct (reg. define ->reg. use) dependence are easy to construct directly from address of used registers, but the main difference in scheduling make IO dependences (load, store, output).

   For this it is necessary to propagate constants from loadI instruction and to 'emulate memory' with hash map. If there are loads from addresses unknown to hash map, we have to assume that this value can be anything. The main data structures for making IO dependencies is **listIO** that includes <line_num, opcode, mem_loc>. This list will contain all load, store, output instructions in executing order of ILOC code, with joint information about addresses they use. For each of instructions in the list the scheduler iterates backward until it finds another instruction with location possibly same as it's location.

2. How did you implement the ready and active queues (or lists, or sets, or …) ?

   The Ready queue is implemented as priority queue (C++ set) with elements pair <priority, ins_id> sorted in descending order by priority. In this way the scheduler doesn't have to go to far when choosing next instruction.

   The Active queue is implemented as multi-queue (C++ multiset), that contains pair<finish_cycle, ins_id> as members, so that finish_cycle stands for cycle in which some dependence towards the instruction with ins_id will become finished. As every instruction knows how many input edges it has, it will just increment counter for every dependence from active queue and put it into Ready queue when that count is the same as number of input edges.

3. What priority function or functions does your scheduler use to determine which operation to schedule next? Does it use any tie-breakers? How does

your scheduler ensure that the restricted operations—that is, load, store, and multiply—are scheduled legally? [1]

In this project scheduler implements several heuristics on choosing the next instructions. It prioritizes load over store for the first functional unit, because the it tries to avoid *store before load/output* dependency that has latency of 5 cycles, while if store goes after all *load/store/output* instruction latency is just 1 cycle. Nevertheless, load and store has priority over all other operations for the first functional unit.

For the second functional unit *mult* has the greatest priority over all valid operations for that unit.

For the output operation this scheduler puts the lowest priority, simply because it doesn't open opportunity to make ready some other instruction.

4. Does your scheduler perform any pruning on the dependence graph? If so, explain how you determine that an edge is not needed and any analysis that your scheduler performs to support that decision.

Yes, once the dependence graph is constructed with all dependencies, the scheduler is trying to prune IO edges come from loading value from some unknown location. For this, the scheduler puts some initial value like 11111 for the unknown value, makes propagation of that value all other values are known and writes that into *load* and *store* nodes that use that as address. In the second step, the scheduler just check all load/store operation with previously unknown addresses and if there is IO edge between that and some other instruction with different address, it delete that edge, and iterates the **listIO** mentioned before to check if there is some dependency with some previous instruction.

5. State the asymptotic complexity and expected case complexity of your register scheduler.

The overall complexity should be linear in terms of instruction counts. Creating dependence graph, computing priority, proving different unknown addresses and scheduling take linear time each, so the total complexity stays also linear .

[1] That is, no cycle can contain two or more operations drawn from the set { **load**, **store** } or two multiplies.

Effectiveness *refers to the speed at which the allocated code runs.*

Efficiency *refers to the speed at which your scheduler runs.*

Quantitative Results

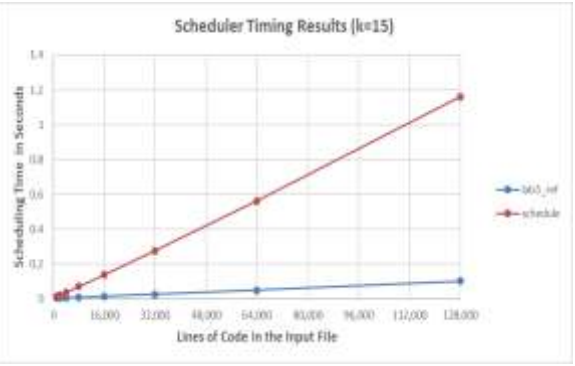
Insert Table 1 and Table 2 (described in the lab handout)

Table 1: Total Cycles Required for Lab 3 Report Blocks & Submitted Block *

Input Block	schedule (cycles)	lab3_ref (cycles)	Difference (percent)	Input Block	schedule (cycles)	lab3_ref (cycles)	Difference (percent)
report1.i	25	25	0%	report13.i	25	25	0%
report2.i	23	23	0%	report14.i	19	19	0%
report3.i	26	26	0%	report15.i	25	23	9%
report4.i	33	33	0%	report16.i	44	57	-23%
report5.i	25	25	0%	report17.i	22	21	5%
report6.i	43	43	0%	report18.i	18	22	-18%
report7.i	32	36	-11%	report19.i	38	42	-10%
report8.1	15	15	0%	report20.i	22	22	0%
report9.i	26	26	0%	report21.i	69	72	-4%
report10.i	41	40	3%	report22.i	50	52	-4%
report11.i	30	30	0%	report23.i	67	69	-3%
report12.i	21	21	0%	dx4.i	45	60	-25%

Table 2: Scheduler Timing Results

Input (lines)	Scheduling Time (seconds)	
	lab3_ref	schedule
1000	0.003401	0.011499
2000	0.003632	0.019476
4000	0.005167	0.036017
8000	0.008042	0.069876
16000	0.014152	0.137803
32000	0.02542	0.276947
64000	0.047866	0.56158
128000	0.101635	1.161942



1. With reference to Table 1, which shows the effectiveness of your scheduler:
   a. Discuss the results. In particular, how well did your scheduler perform versus the reference implementation in terms of correctness and effectiveness? Justify your answer quantitatively.

      The implemented scheduler gave all correct answers to gives slightly faster code than reference implementation.

   b. Were there blocks where you felt your scheduler did particularly well?

      The scheduler performs particularly well in the situations with unknown memory location in advance.

   c. Were there blocks where you felt your scheduler did particularly poorly?

      I have not find such block.

d. What changes did you make to your scheduler to improve its effectiveness?

Constant propagation for determining IO dependences, several heuristics in choosing between ready instruction and proving different address for reading from unknown location.

2. With reference to Table 2 and your graph:
   a. How well did your scheduler perform versus the reference implementation in terms of efficiency? Justify your answer quantitatively.

      If your scheduler is written in C, C++, Java, Haskell, OCaml, Python, Ruby, or R, refer to the scheduler timing results shown in § A-4 of the lab handout when discussing the impact of your programming language choice on the efficiency of your scheduler.

      The scheduler is written in C++ and perform few times worse than reference implantation. This is probably because of making optimizations separately instead.

   b. If your scheduler is less efficient than the reference implementation, discuss the design decisions that you made when implementing your scheduler that are most likely to account for the difference in efficiency.

      In the first implementation the edges in the dependence graph were implemented with std::list, and during their update while adding IO edges the scheduler iterated through all list, that make impossible scheduling of examples from Timer benchmark. In the latest implementation that has been changed to hashmap and the scheduling become possible in less than a second for 100K instructions.

      To improve implementation further, it will be beneficial to schedule instructions on the block level (for example 50 instructions by 50 instructions). That would make better locality and memory allocation.

   c. Are the timing results for your scheduler consistent with the complexity analysis that you gave earlier? If not, explain why they are different. Justify your answer.

Yes, from the Timing results it could be clearly seen that scheduler works in linear time.

Experience

1. Did your implementation experience change your plans or your algorithms?

This implementation come through several stages, with each new optimization and improving in data structures.

2. Based on your experience, would you use the same algorithm if you had to start from scratch? If not, what would you do?

This algorithm gave good results, but it would be wise to discuss about other algorithms with better locality like peephole.

3. How did your choice of implementation language affect your ability to complete the project on time?

C++ is a good choice because it provides all needed data structures and tools for development.

4. What advice would you give future COMP 412 students embarking on Lab 3

Start early, because once you start working, you will always want to implement some new optimization, for which you will not have time if you start later.