Lab 2 Questionnaire

Comp 412, Fall 2018

Name:	**Luoqi (Rocky) Wu**
Net ID:	**lw31**
Date Submitted:	**10/14/2018**

Implementation Discussion

1. **Provide a high-level description of the register allocation algorithm that you implemented. Highlight any differences between your allocator and the method discussed in class.**

   Do not use pseudo code. Do not include descriptions of your scanning or parsing algorithms.

   The driver 412alloc.py *first scans and parses* the input file and provides the ready-to-go IR for the Renamer.

   Next, the Renamer does a bottom-up pass through the IR to *rename the source register names* into more informative virtual register names. It also keeps track of MAXLIVE, the maximum number of live values per line throughout the block.

   Then, driver 412alloc.py checks whether MAXLIVE>k to decide whether to use AllocatorWithSpill or AllocatorWithoutSpill (we will mainly discuss with spill below).

   Afterwards, the allocator loops over each of the n instructions of the IR, first *allocating PRs* for OP1 and OP2, then frees OP1 and OP2's PRs if this is their last, and then allocating a PR for OP3.

   The allocator calls get_a_pr to *allocate a PR* for an OPn.

   get_a_pr first checks the available_prs stack and pops an available PR if possible. If the stack is empty, it will spills and return an in-use PRx. The PRx will be restored later on when its needed again.

   After the allocation process is done, 412alloc.py calls Allocator's print method and *prints out the lines in the IR*, often into an ILOC file.

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

   My key design decisions include:

   1. Letting the driver 412alloc decide whether to allocate with spill or without spill by comparing the MAXLIVE returned by the Renamer;
   2. Adding an extra column to the IR lists that indicates whether to print this line or not, which enables delaying printing loadI until needed;

3. Delegating the task of finding, spilling a PR to getaPR;
4. Deciding to directly insert spill and restore lines into the IR and then print out the IR line by line after done with the main allocation loop;
5. Breaking ties among furthest next use PRs by simply picking the one in the smallest index.

These are the five important design decisions I made for my allocator.

3. When your allocator must spill, how does it choose the value to spill? What optimizations, if any, did you implement?

When my allocator must spill, i.e. available_pr stack is empty, it checks PRNU and chooses the value with the furthest next use to spill. It also makes sure that when allocating to OP2, it does not allocate the same PR as OP1 to OP2, unless OP1VR = OP2VR.
I tried three optimizations to break tie: randomly picking one of the tied PRs to spill; always picking the first PR; always prioritize a PR whose value is defined in a loadI. In practice it is interesting to find out that always picking the first PR actually yields the fewest number of cycles.

4. State the asymptotic complexity and expected case complexity of your register allocator.

[Asymptotic Complexity]

O(nk), where n=number of instructions in the input ILOC block, and k=number of physical registers allowed.

The reason is in allocate, it loops over each input instruction, then within each iteration, the worst case is when it needs to spill and needs to loop over the PRNU list of length k to find the best PR to spill. Therefore the asymptotic complexity is O(n•k).

[Expected Case Complexity]

Worst Case: the worst case is when k<MAXLIVE, so we need to reserve a spill register, as well as having to constantly spill and restore. In this case, the complexity is O(nk).

Average Case: the average expected case complexity is still O(nk) since we loop over n instructions and on average a constant t% of these iterations we need to spill and visit PRNU. So complexity is O(t%n•k) = O(n•k)

Best Case: the best case for my allocator is when k ≥MAXLIVE. In this case, no spilling is needed and the complexity is O(n). If we want to be even more aggressive, we can find the best case to be when the input ILOC consists entirely of unused loadI instructions and a constant number of store and outputs. In that case, the complexity is a constant O(1).

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

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

Quantitative Results

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

Table 1: Total Cycles Required for Lab 2 Report Blocks & Submitted Block *										
Input Block	Allocator	Units	Available Registers						Original ILOC Code	
			k=3	k=4	k=5	k=6	k=8	k=10		
report1.i										
	lab2_ref	cycles	215	161	125	95	65	54	54	
	412alloc	cycles	230	190	143	110	70	54		
	Difference	percent	-7.0%	-18.0%	-14.4%	-15.8%	-7.7%	0.0%	100.0%	-10.5%
report2.i										
	lab2_ref	cycles	171	117	105	99	92	86	56	
	412alloc	cycles	173	154	121	115	103	91		
	Difference	percent	-1.2%	-31.6%	-15.2%	-16.2%	-12.0%	-5.8%	100.0%	-13.7%
report3.i										
	lab2_ref	cycles	409	351	312	266	217	179	82	
	412alloc	cycles	411	364	317	262	219	179		
	Difference	percent	-0.5%	-3.7%	-1.6%	1.5%	-0.9%	0.0%	100.0%	-0.9%
report4.i										
	lab2_ref	cycles	169	162	156	150	138	124	68	
	412alloc	cycles	171	164	158	152	140	127		
	Difference	percent	-1.2%	-1.2%	-1.3%	-1.3%	-1.4%	-2.4%	100.0%	-1.5%
report5.i										
	lab2_ref	cycles	84	61	48	36	30	30	30	
	412alloc	cycles	79	60	42	36	30	30		
	Difference	percent	6.0%	1.6%	12.5%	0.0%	0.0%	0.0%	100.0%	3.3%
report6.i										
	lab2_ref	cycles	276	243	196	185	166	154	105	
	412alloc	cycles	339	268	256	185	170	163		
	Difference	percent	-22.8%	-10.3%	-30.6%	0.0%	-2.4%	-5.8%	100.0%	-12.0%
report7.i										
	lab2_ref	cycles	115	90	73	68	60	46	44	
	412alloc	cycles	116	96	93	89	61	50		
	Difference	percent	-0.9%	-6.7%	-27.4%	-30.9%	-1.7%	-8.7%	100.0%	-12.7%
submitted.i										
	lab2_ref	cycles	79	60	42	36	30	30	30	
	412alloc	cycles	79	60	42	36	30	30		
	Difference	percent	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	100.0%	0.0%
* The simulator runs used to generate the results shown in this table ran without errors and produced correct output.										-6.0%

Figure 1: Table 1: Total Cycles Comparison

Table 2: Allocator Timing Results (k = 15)		
Input (lines)	Allocation time (seconds)	
	lab2_ref	412alloc
1000	0.003395	0.095513
2000	0.004204	0.172790
4000	0.005614	0.334796
8000	0.009171	0.691276
16000	0.016065	1.505889
32000	0.029343	3.631425
64000	0.056002	9.913889
128000	0.111982	30.053411

Figure 2: Table 2: Allocator Timing Results (k=15)

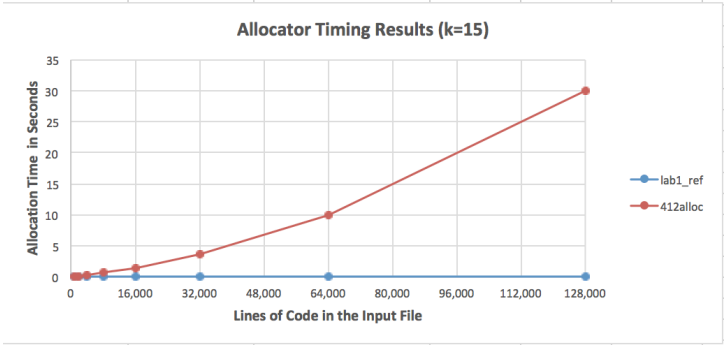

Figure 3: Allocator Timing Graph

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

      Correctness: I used AllocAndRunAll scripts to check correctness and simulator cycles for my allocator. My allocator is correct across all the ILOC input blocks I tested, including **report1.i – report7.i** and **block1.i – block12.i**.

      Effectiveness: My allocator comes close to the reference allocator in terms of effectiveness. Across the 7 report blocks, the average cycle difference percentage between reference and mine ranges from **-13.7% to +3.3%**, with a total average of **-6.8%**.

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

      My allocator did particularly well in report3 (-0.9%), report4 (-1.5%), and report 5 (+3.3%).

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

      My allocator did poorly in report2 (-13.7%), report6 (-12.0%), and report7 (-12.7%).

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

      I worked on (1) rematerializable values, (2) allocate without a reserved register when MAXLIVE ≤ k, and (3) some cases of clean values.

      (1) There are two improvements regarding rematerializable values. 1. Since remat values come from a loadI, when we spill it, we don't need to add any additional code, and when we restore it, we simply use one line of loadI to restore. 2. A further noticeable improvement is that we can actually defer the printing of loadI operations until that value is actually needed. This means we essentially free up a PR in between those lines and hugely improve the allocator's effectiveness.

      (2) I also calculated the maximum number of live values in the ILOC block, called MAXLIVE, in parser. The benefit is the allocator only needs to reserve a spill register in the case where MAXLIVE > k. In practice, however, only a handful cases actually fulfill this condition.

      (3) I also kept track of some guaranteed clean values, namely those that come from a restore line. These values are guaranteed to be clean since the proceeding loadI loads an address in the reserved memory address set. This saw some minor improvements in terms of effectiveness.

The average cycle difference percentage improved as I made more changes: Basic implementation (-57.6%) → Implement naive rematerializable cases (-21.3%) → Allocate without spill if permitted (-20.3%)→ Defer loadI instructions until needed (-7.9%) → Some clean values from restore (-6.8%).

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

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

      My allocator is written in Python and compared to the reference implementation its efficiency is comparable (40%) until input n = 64k and 128k (4.78 sec and 9.66 sec vs. 9.91 sec and 30.05 sec).

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

      1. Use of dictionary: I use some dictionaries to keep track of, for example, OPCODE types. It likely could be faster if I used an array.

      2. String comparison: my internal bookkeeping lists such as PRNU, PR2VR, and VR2PR are initialized as a list of "invalid" strings. I also check OPCODE of each line by string comparison.

      3. Class construction: to make the code more modular, I created different Python classes such as Scanner, Parser, IR, Renamer, Allocator, etc. Although this makes the code look cleaner, it certainly slows down the code.

      4. Inserting list into IR: I directly insert the spill and restore line into the IR, which is a Python list. Since list insertion takes O(n), and in some cases the allocator spills a lot, the speed gets compromised.

      5. Not printing on the go: my allocator loops over the entire ILOC file first while doing allocation, then loop over the IR again to print out every line. If the allocator prints on the go while it is allocating, the run time will be reduced in practice.

      The five reasons are the main contributing factors for the difference in efficiency.

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

It is **somewhat consistent** with the complexity O(nk) up until n = 64k. Since in our timing test we fix k=15, we should expect to see a linear relationship between allocation time and line of code in the input file, resulting in a straight line.

In practice my allocation time vs. line of code is more or less quadratic than linear. I suspect the reason is attributed to the reasons mentioned above in point b. In particular, inserting spill and restore line directly into the IR goes under the radar, not taken into account by idealized complexity analysis. Whereas a pointer chasing IR structure or print-as-you-go method would both result in a constant insert time, inserting directly into the IR, a Python list, would incur almost on average an $O(n)$ cost each time we need to spill and restore, therefore in the worst cases, the actual time complexity is approaching $O(n \cdot (n + k))$, when n>>k, the actual complexity approaches $O(n^2)$.

3. **In your testing, did you identify specific ILOC blocks that demonstrated particular strengths of your allocator? If so, list the blocks and what features they demonstrated.**

   Three ILOC blocks demonstrated particular strengths of my allocator, namely report3 (-0.9%), report4 (-1.5%), and report 5 (+3.3%).

   report3.i and report5.i shows the strength of **keeping track of clean values** since they both constantly use and define values, resulting in a lot of spilling and restoring. These restored values are clean values and thus saves a lot of precious cycles.

   report4.i shows the benefits of **delaying printing loadI until needed**. It defines R1 to R20 upfront using loadI, then uses them in the reverse order, R20-R1. Therefore simply moving the loadI definitions until later when they are actually needed frees up many registers and avoids a lot of unnecessary spilling and restoring.

Experience

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

   Slightly, I considered switching to printing the ILOC on the go instead of printing it whole after done allocating but ultimately opted against so. For the most part, I just got the algorithm to work then added a number of improvements.

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

   Yes, I would. The algorithm I used is a combination of the one from lecture slides and that from the book. It is very intuitive and once I got it to work, I just needed to add things to improve its efficiency. It is friendly to change.

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

   I chose Python as my implementation language and it helped me significantly with writing, improving, debugging, and ultimately completing the project on time.

4. **What advice would you give future COMP 412 students embarking on Lab 1?**

   Focus on making a prototype allocator that works first! This is in my opinion the most important step in this lab.
   Also, do not rely on the algorithms provided by the tutorial and book too much since they are often incomplete and not inspiring enough for you to come up with efficiency improvement designs. Instead, talk to your peers, TAs, and professors to find inspirations and ideas.