Author: Clifford Imhomoh

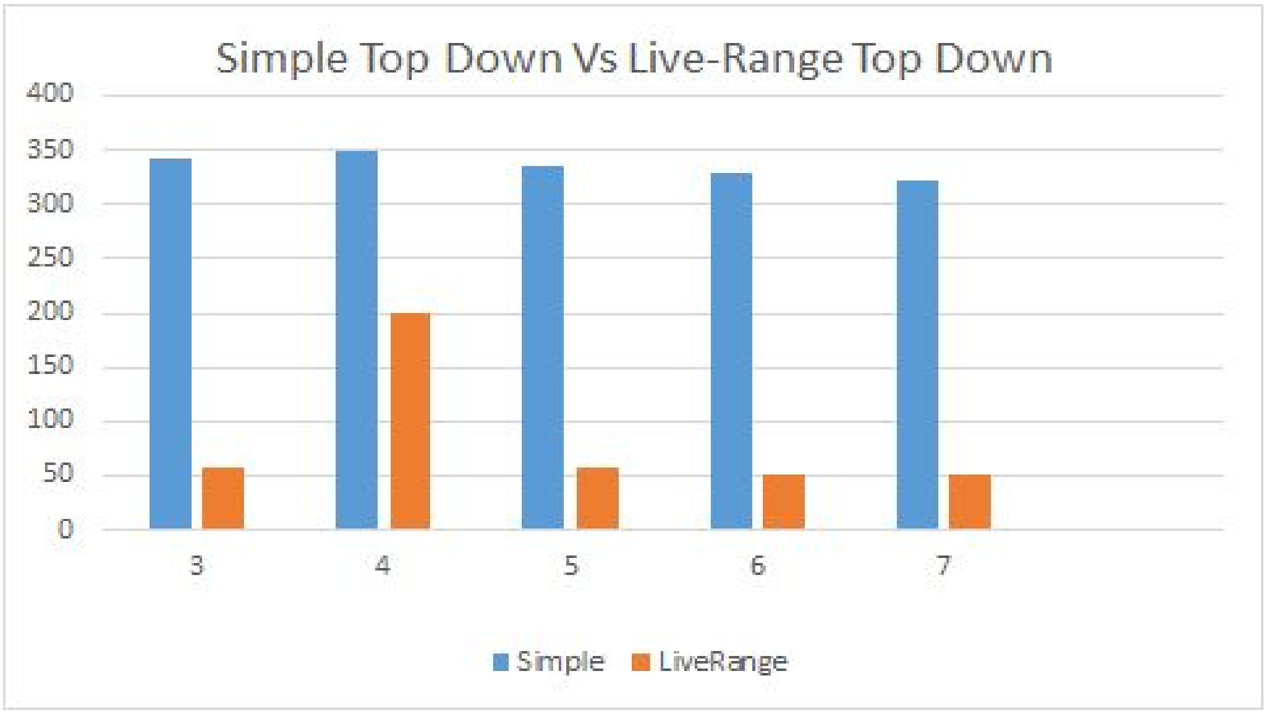
Abstract

The goal of this project was to implement the three different register allocation heuristics. This heuristics are the Simple Top Down Allocator, Top Down Allocator with Live Ranges, and The Bottom Up allocator. The Simple top down allocator takes in a set of K physical registers and maps the virtual registers with the most occurrences to those registers. The other virtual registers are spilled to memory. The live range top down allocator spills when we come across a max live that is greater the the total amount of physical registers available. It spills the registers with the least occurrences but when there is a tie, it picks the one with the longest live range. This ensures that register pressure is relieved across multiple max lives that the spilled register may be part of. The Bottom up allocator does not take live ranges or occurrences into consideration. It works similarly to caches given that it spills a register based on how far in the future it is going to be use (although caches base their knowledge on what is known from the past). Whenever a register that has been spilled into memory is encountered again, it is simply loaded back into a physical register from the same offset it was spilled to. Running multiple tests on these different heuristics beared some very interesting results. Some heuristics worked better than others on certain sizes of inputs while some kept a linear growth in their number number of cycles. The biggest deterministic factors in the cycles of these allocators are the total number of virtual registers and the total number of available physical registers we have for allocation. The main metric that was used to compare the different heuristics was the amount of cycles that each took after their respective allocation algorithm was applied. For the sake of simplicity, only one report block was used for testing.

Simple Top Down Vs Live Range Top Down

First we will compare the simple top down and the live range allocator. We will show this comparison by graphs that shows the total cycles they ran against the amount of physical registers that were available for allocation. This graph represent the performance of each heuristics on the input file report6.i . The x-axis represents the amount of allocable registers and the y-axis represents the amount of cycles.

Test on Report6.i with 5,6,7,8,9 Physical Registers, 2 feasible and 3,4,5,6,7 allocatable registers respectively

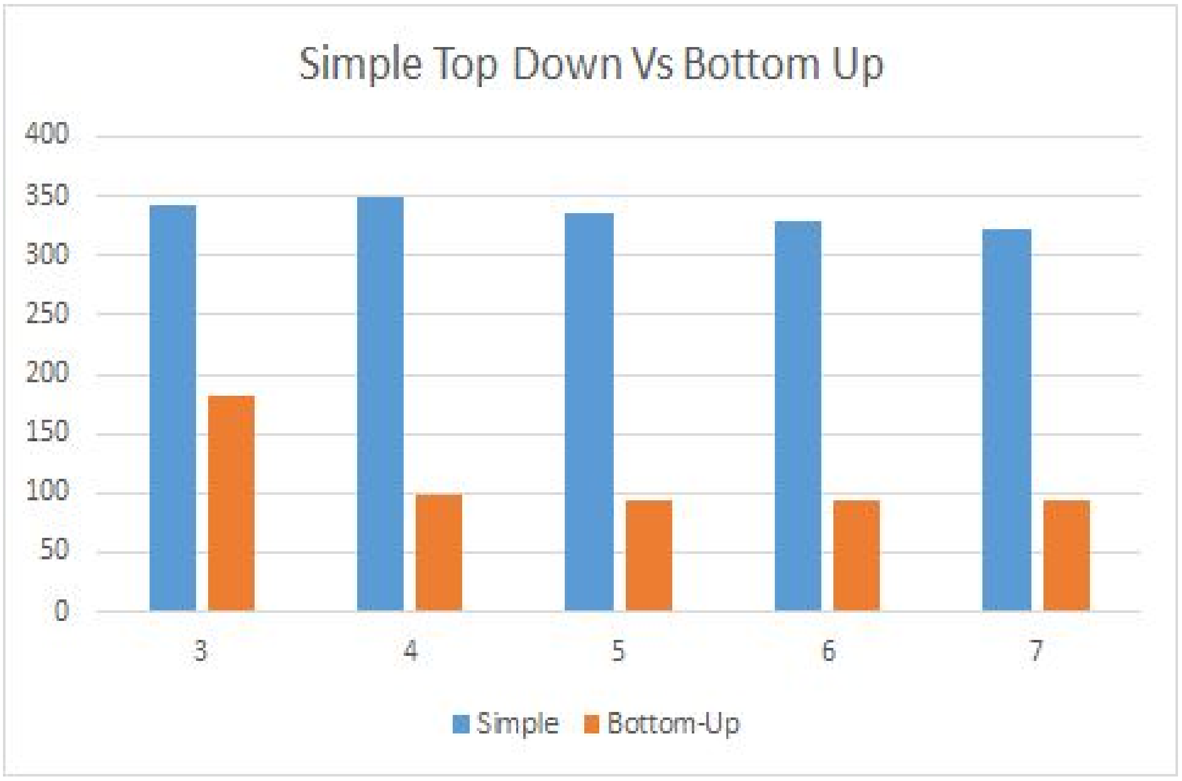


Graph Analysis: As evidently portrayed by the plot above, The top down allocator that considers liver ranges is more optimal than the simple top down allocator. There are two main reasons why this is the case. The first reason is because the live-range top down allocator spills only when MAXLIVE > number of allocable registers. With this in mind, this allocator is able to get rid of the unnecessary spilling present in the simple top down. The second reason is the part of the live-range allocator that break ties on what to spill by using live-ranges. This ensures that register pressure is relieved across other MAN LIVES in the code which in turn would reduce the amount of spilling. One important thing to note is the unexpected rise of cycles in the live-range top down allocator when allocable registers = 4. This behavior can be explained by the fact that some times, this heuristic may select a register with a very low occurrence but and a very small live-range. What this means is that more register pressure will be present across more live-ranges which in turn would lead to more spilling. The simple top down allocator maintains cycles over 300 for all the inputs although (as expected) it slowly declines in the number of cycles as the number of allocable registers grow larger.

Simple Top Down Vs. Bottom-up Allocator

The next two allocators to be compared are the simple top down allocator and the bottom-up allocator. Below is a graph showing their cycles of execution plotted against the amount of allocable registers. The x-axis represents the amount of allocable registers and the y-axis represents the amount of cycles.

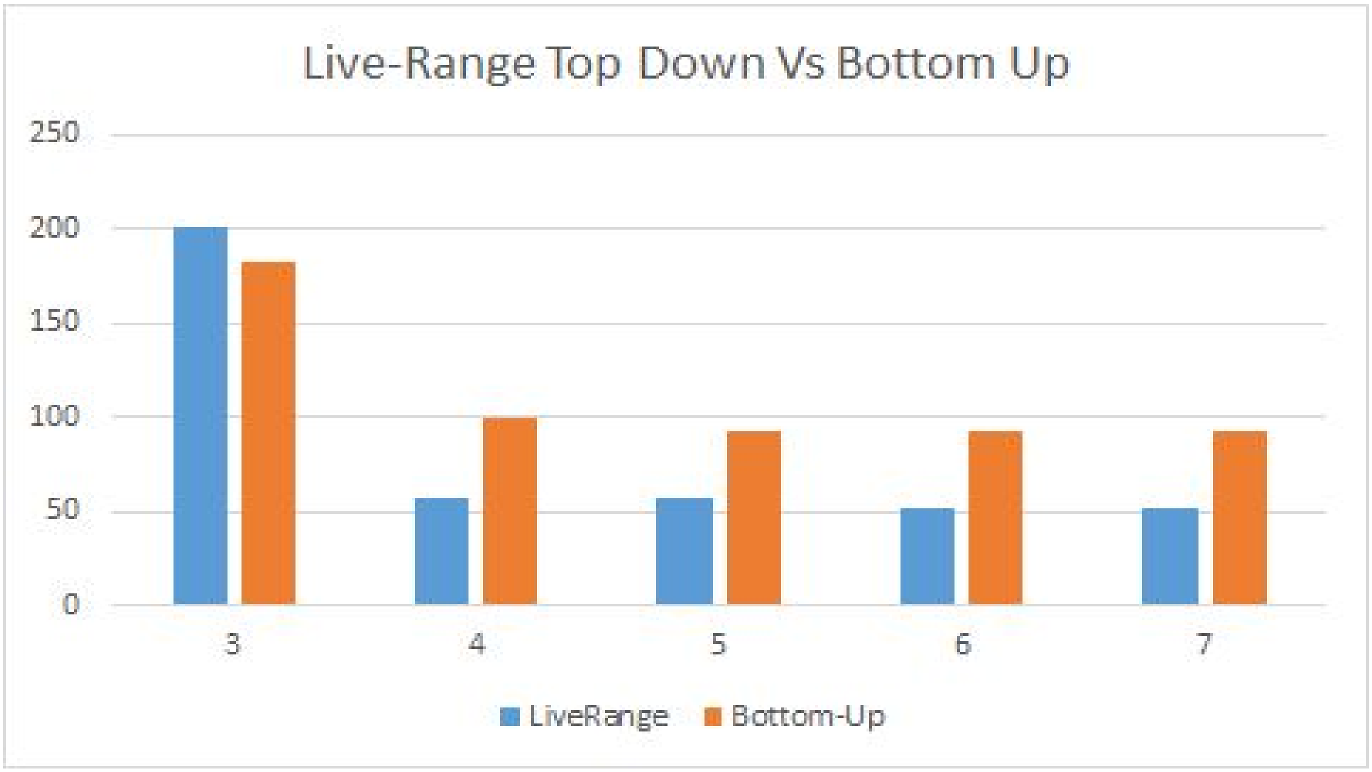
Test on Report6.i with 5,6,7,8,9 Physical Registers, 2 feasible and 3,4,5,6,7 allocatable registers respectively



Graph Analysis: Based on the graph. The bottom-up register allocator performs much faster that the simple top down register allocator. There are two main parts of the bottom up heuristic that gives it so much efficiency. The first thing is that the bottom up allocator only spills when it is out of physical registers. It does not consider MAX LIVE or live ranges in its algorithm. This means that it will not compute registers to spill before hand and it is greedy in the sense that it adapts to the code shape by only spilling registers that are the farthest use in the future. The second thing that makes this algorithm so efficient is its ability to re-use registers. When a virtual register currently living in a physical register has a next use value of infinity, the bottom up allocator gives back its physical register to the available pool of allocable registers. This reduces the amount of spill code generated because any new virtual registers can re-use that freed up physical register. The simple top down allocator maintains its previous poor performance as represented by this graph (It is just a terrible algorithm to put simply).

Bottom-Up Allocator vs Top down Live-Range Allocator

Now for the final comparison, we will compare the two of the more efficient register allocation algorithms. To restate, the live-range top down algorithm considers max lives and live-ranges while the bottom up algorithm allocates on the go and when it needs to spill, it spills the register that has the farthest use in the future.



Graph Analysis: The results above are very surprising and are open to a lot of analysis. As we can see, the live range top down allocator performs better than the bottom up allocator through most of the inputs. One important thing to note is that the bottom up allocator actually performs better than the live range allocator when the number of allocable registers are small. This is mainly because the bottom up allocator does not determine what registers to spill before hand. So on very low amount of allocable registers, it is able to optimally use the registers to reduce the amount of spill code that is produced. The main question that still lingers is why the live range allocator uses less cycles than the bottom up allocator over so many inputs. There are a few things that we can consider. One thing to consider is the ability of the top down allocator to reduce register pressure when enough registers are presented. Assume this code block for example:

```
loadI      1020  => r0
loadI     1        =>  r1  // a               r1
loadI     2        =>  r2  //b                r1 r2
subI      r2, 4    =>  r3   //c           r1 r2 r3
add       r1, r2   =>  r4  //d            r1      r3 r4
addI      r4, 1    =>  r5  //e            r1       r3  r4 r5
mult      r3, r5  =>  r6                  r1            r4 r5 r6
sub       r5, r6  =>  r7  //f             r1            r4 r5     r7
add       r7, r5  =>  r8                  r1                          r7 r8
add       r8, r7  =>  r9 // g             r1                                  r9
add       r9, r1  =>  r10    // h                                                  r10
loadI     1020  =>  r11                                                            r10 r11
Store      r10    =>  r11                                                                r11
outputAI r11
```

As we can see, the MAXLIVE in this code block is 4. Assuming we have only 3 allocable registers, is it possible to relinquish the register pressure throughout the code by spilling a single register? The answer is yes based on the live range top down allocator. By spilling only one register, more notably “r1” , we the register pressure is relieved throughout the code and we are able to allocate registers without having to use any spill code. Because “r1” has a very long live range, spilling it gives us this property of possibly reducing the amount of spill code .The resulting code after allocation looks like this:

Feasible reg = f1 f2   available = ra , rb, rc

ILOC CODE:

```
loadI         1024   =>    r0
loadI         1        =>     f1
storeAI       f1       =>    r0, 0   // spill r1 to relieve register pressure
loadI         2        =>    ra       //r2
subI          ra, 4     =>    rb      // r3
loadAI       r0, 0    =>     f1     // load r1
add         f1, ra    =>    ra // r4
addI        ra, 1    =>    rc // r5
mult        rb, rc   =>    rb // r6
sub         rc, rb   =>    rb // r7
add         ra, rc   =>   ra  // r8
add         ra, rb  =>   rc   //r9
loadAI      r0, 0   =>    f1   // load r1
add         rc, f1  =>    ra   // r10
loadI       1020  =>    rc  // r11
store       ra      =>     rc
outputAI  rc
```

As we can see above, nothing else had to be spilled after spilling “r1”. Although the bottom up allocator maintains a very good heuristic, it still lacks in its ability of relieving register pressure because it does not use the concept of max lives.