Local Register Allocation

Brian Uribe

February 18, 2021

1 Introduction

Register allocation is one of the most important steps in compiler design. Since accessing data through memory is a costly operation; appropriate register utilization is vital for the generation of good code. We want to map an unbounded number of variables into a limited number of physical registers, and we want to do so efficiently. Unfortunately, this problem is known to be NP-complete. This prevents us from generating perfectly optimal allocations without hurting the performance of our compiler. We will look into four heuristics and algorithms that we can use to approximate the solution.

2 Simple Top Down Allocator

2.1 Description

The simple top-down allocator works under a straightforward heuristic. The most frequently used virtual registers must reside in physical registers. If this is not possible then the virtual register must be spilled. The algorithm can be summarized in the following steps:

1. Count the total number of occurrences for each virtual register (excluding any reserved registers)
2. Assign $k - F$ virtual registers to physical registers, where F is the minimum number of registers to execute any instruction with both operands in memory.

3. Assign each one of the remaining virtual registers to a unique offset in the stack.

4. Rewrite the code to reflect those choices.

During the rewriting process one small optimization we can apply is to keep track of the variables each feasible register is holding to avoid loading a value from memory that is already in a feasible(reserved) register. This prevents instructions such as $add \;\; r_x, \;\; r_x \rightarrow r_y$ from loading r_x twice.

2.2 Benchmarks

File name	**Total cycles using**		
	5 registers	10 registers	15 registers
block1.i	518	461	410
block2.i	396	337	302
block3.i	168	123	83
block4.i	163	118	79
block5.i	462	411	376
block6.i	313	278	226

Table 1: Total cycles after allocation

As we can appreciate from the table 1 there's a relatively high number of cycles on each allocation. This is due to the hidden implications of our heuristic. Although we want to keep the most used values in registers, they might not need to be *live* at the same time. A variable that was assigned to a physical register in the first half of the program might not be present in the second half. As a consequence, this physical register is holding a value that might not be used anymore throughout the remaining execution of the program. Granted this results in further spilling and a lower quality of the compiled code.

3 Live Ranges & Top Down Allocator

In the previous section we saw how the simple top down allocator produced code with a high amount of spill code. If we want to improve its allocation we would need to take into account more factors to help us decide which values must be spilled.

3.1 Description

This version of the Top down allocator uses the concept of **max live** and **live ranges** in addition to the usage frequency. A live range can be defined as the interval between a variable's declaration and its last usage. The maximum number of overlapping live ranges tells us the maximum number of registers we need to execute a given code. This number is also known as **max live**.

Now that we know how many registers are needed we can improve our heuristics by only spilling the necessary amount. If $maxlive < k$ we have enough registers to execute this code. Using the live ranges we can determine the availability of each virtual register. This allows us to re-allocate a physical register when the value it is holding is not needed anymore. Since not all live ranges would overlap, there will always be the necessary amount of registers to execute the code without spilling.

If $maxlive > k$ we need to spill t virtual registers such that $k - t = maxlive$. Similarly, to the simple top down allocator we will spill the virtual register that is *less* frequent, however, if two virtual register have the same frequency we would spill the one that has the *longest* live range. Finally, we can assume that we have enough registers to perform allocation, if that is not the case we would insert spills using a register from the feasible set.

3.2 Benchmarks

File name	Total cycles using		
	5 registers	10 registers	15 registers
block1.i	169	96	96
block2.i	273	96	81
block3.i	96	47	47
block4.i	96	36	36
block5.i	378	75	75
block6.i	208	54	54

Table 2: Total cycles using improved heuristic

Loading registers on-demand and minimizing spill variables generated much better results than what we saw previously; however, we assumed that by spilling t virtual registers, we would not need to make any further spills. When in fact, it is possible that even after spilling there is a $maxlive > k$ condition. Our heuristic tries to make what appears to be the best solution, but it does not consider which virtual registers are overlapping. Another inconvenient is the need to reserve F registers to our feasible set. This leaves the allocator with fewer registers available, which results in further spilling, even though no spills would have been required if we had chosen to use all registers instead.

4 Bottom Up Allocator

4.1 Description

In our previous approaches, we used high-level information to make our decisions. The bottom-up allocator uses low-level information such as register demand to drive its decision-making process. This allocator satisfies registers supply on-demand. If there is not enough physical registers available then; it spills the register that would be used the farthest into the future. This heuristic effectively minimizes the cost of each spill by making sure that every virtual register that is currently needed the most is in a physical register.

The allocator (*see algo. 1*) makes use of three procedures *ensure*, *allocate*, and *free*. The algorithm iterates through every instruction and *ensures* that

every virtual register being read is in a physical register. Once both operands are in a register, it then;*frees* any one of them that is holding an unnecessary variable. The allocator can, later on, reuse one of those registers to hold the result of the instruction. If at any point it runs out of registers. It selects the physical register with the highest next usage and spills the virtual register currently associated with it.

Algorithm 1: Bottom Up Register Allocator

```
Class Register:
    physical_name;
    virtual_name;
    next = ∞;
Function ensure(vr):
    if vr is in a physical location and is not spilled then
        return location[vr];
    if vr has been spilled then
        reg = allocate(vr);
        insert load instruction into reg;
        delete vr from spilled set;
        return reg;
    return allocate(vr);
Function allocate(vr):
    if there are available registers then
        reg = available.pop();
    else
        reg = max(register.next);
        insert spill code;
    location[vr] = reg.physical_name;
    reg.vr_name = vr;
    reg.next = −∞;
    return reg;
foreach instruction i do
    foreach vr that is being read from do
        reg = ensure(vr);
        rewrite i using reg.physical_name;
    foreach vr we previously allocated do
        reg = location[vr];
        if there are not more usages of vr then
            free(reg);
        else
            reg.next = next_use[vr];
    destination = allocate(vr);
    rewrite i using destination.physical_name;
     destination.next = next_use[vr];
```

4.2 Benchmarks

File name	Total cycles using		
	5 registers	10 registers	15 registers
block1.i	96	96	96
block2.i	104	81	81
block3.i	59	47	47
block4.i	52	36	36
block5.i	75	75	75
block6.i	54	54	54

Table 3: Total cycles after allocation

The performance of the bottom up allocator appears to be just slightly better than the top down allocator with live ranges when there are 10 - 15 registers available. However it is much better when there are only 5 allocatable registers. The reason is that the top down allocator reserved F registers to perform spilling, and it only used $k - F$ allocatable registers. This might not punish the allocation when there is a considerable amount of registers, however, as shown by the results it hurts the quality of the generated code when there is not enough registers. All things consider, the bottom up allocator is very close to optimal, while also being quite fast. To be optimal, the allocator must consider the cost of spilling a virtual register that needs to be stored and one that does not.

5 Linear Scan Allocator

5.1 Description

The linear scan allocator is a global register allocator first described in 1999 by M. Poletto & V. Sarkar[1]. Unlike the allocators we have seen so far, the linear scan allocator does not look at the instructions. Rather, it makes choices based on live intervals. it assigns a physical register to each live interval while making sure that intervals that have already expired are discarded. It only generates spills when the number of overlapping intervals is greater than the number of available registers. On each iteration the algorithm maintains a list of overlapping *active* live intervals sorted by the end

point. Every time it finds a new interval it removes any expired intervals from *active* and adds the register associated with that interval to the list of available registers. The *active* list is kept sorted to simplify the task of determining which live intervals overlap with the new one. If the length of *active* list is k it means that all registers are currently overlapping, and a spill is necessary if we want to include the new interval. There are several spilling heuristics we can use. The one we chose is spilling the last live interval from *active*. In other words, we spill the interval that would not be used in the nearest future. However, there are also cases in which the live interval we are trying to allocate ends much further in future than any live interval in the *active* list. In this particular case we can directly spill the new interval.

5.2 Benchmark

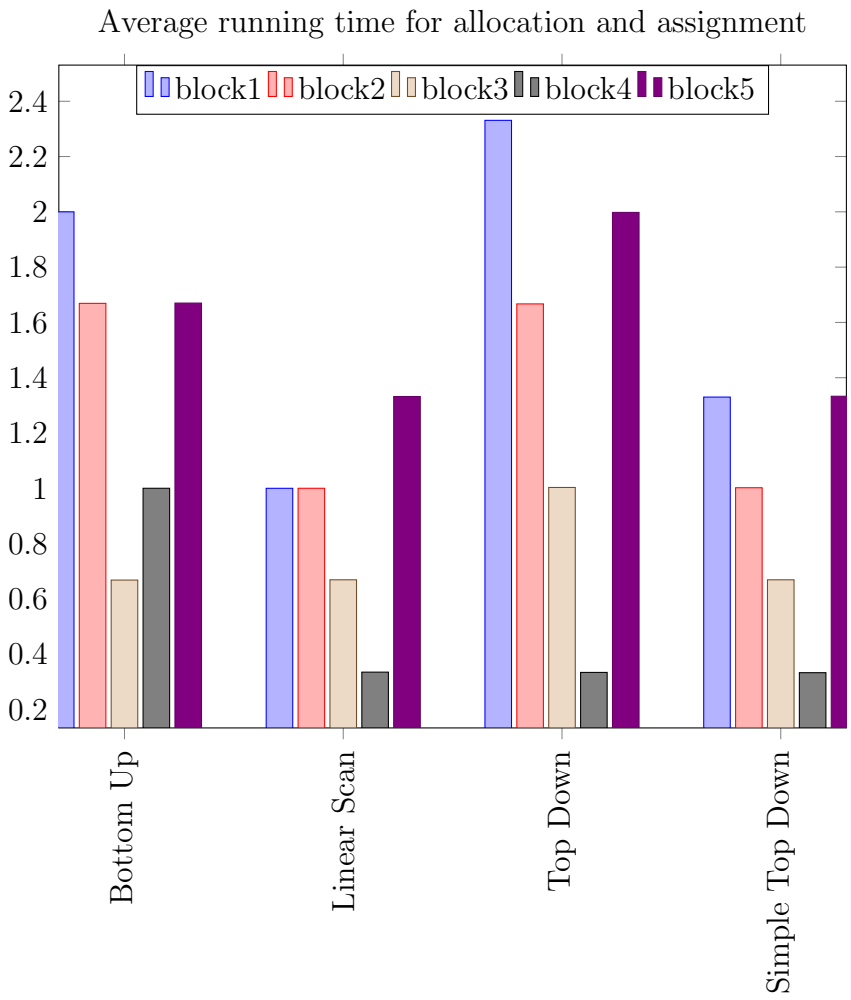
Since Linear Scan is a global register allocator, we made some simplifications to use it in a local context. By keeping track of the internal state of the algorithm, it is possible to generate better assignments. However, we opted for a different assignment strategy. During execution, we keep track of the mapping between physical registers and virtual registers; we also keep track of the memory stack with the corresponding offsets and virtual registers. To make spilling possible, we reserve F physical registers and feed the remaining ones to the allocator. We then iterate through each instruction and if we find a spilled virtual register as a destination we insert a *store* instruction; otherwise, we insert a *load*. If the virtual register is not in a memory location we can directly use its mapping instead.

File name	**Total cycles using**		
	5 registers	10 registers	15 registers
block1.i	146	96	96
block2.i	191	96	81
block3.i	83	47	47
block4.i	80	36	36
block5.i	232	75	75
block6.i	110	54	54

Table 4: Total cycles after allocation

Surprisingly, even with this simple assignment strategy linear scan, matches the bottom-up allocator's performance for 10 and 15 registers. As expected, the performance is not as good using only five registers. This, of course, is because we reserved registers for our feasible set, and we did not reuse any of them when rewriting the instructions. We always loaded/stored without considering whether or not the value is already in a feasible register. Linear scan, however, has some disadvantages, one of which is that a spilled virtual register may remain spilled throughout the rest of the program's execution. Unlike in the bottom-up allocator where spilled variables do have the opportunity to reside in physical registers.

6 Final Thoughts & Analysis



All tests where done using Windows running Python 3.8 on a AMD Ryzen 3200g.

Overall, The bottom-up allocator generated the best register allocation across any number of registers. Due to its innate ability to efficiently reuse and allocate registers exactly when they are needed. Linear Scan, as shown in the graph, is the fastest among them. Even though we used a very simple rewriting strategy, it was able to generate allocations that matched those of the bottom-up allocator. With some tweaks and optimizations to our rewriting strategy, it can potencially produce equal/better allocations than

the bottom up allocator. The top down allocator also generates good allocations, however, it is more computationally demanding. Computing the max live involves sorting the instructions, which bottlenecks the performance of the allocator. It can be improved by using linear time sorting algorithms such as Radix Sort. Finally, The simple top-down allocator although it is fast, and easy to implement; the quality of the allocation is not good, which results in too many insertions of spill code.

References

[1] Massimiliano Poletto & Vivek Sarkar. Linear scan register allocation. *ACM Transactions on Programming Languages and Systems*, 21(5):895, 913, 1999.