Project 1 Report

Joseph A. Boyle

March 4, 2018

Abstract

In this report, we explore different types of local register allocators and their effectiveness. Specifically, we test two versions of the Top-Down Allocator, each with their own selection heuristic, and a Bottom-Up Allocator. The results of these allocation techniques are evaluated on the number of cycles and instructions needed for execution, through a variety of samples of ILOC code.

1 Introduction

We are presented with several snippets of ILOC code which utilize an unbounded number of registers, and utilize one of three allocation techniques to limit the number of registers used to a set number, k, while still maintaining code semantics.

1.1 Bottom-Up Allocation

In the Bottom-Up allocation scheme, we maintain a list of k registers, and note which of those k are currently free. When a register is needed by an instruction, we can either use one of those free registers, or if none exist, spill one of the in-use registers into memory. In-use registers that are good candidates for spilling are those which aren't going to be used for the longest amount of time in the future, therein allowing other values to take its place, reducing the number of loads for closer instructions. Not depending on a feasible set, F, but rather using the given register for loading/storing, all k physical registers may be used for allocation.

1.2 Top-Down Allocation

Top-Down allocation performs a scan of the local block, sorting each virtual register by its number of uses. Using some selection heuristic, we decide which of the virtual registers are best to keep in physical registers and which should be loaded in as needed, generally sorted by usage. These type of allocators require some feasible set, F, of registers for loading/storing values, and as such we are only allowed to use $(k-F)$ registers for allocation, reserving the other F registers for loading/storing values from memory as needed. In ILOC, the feasible set contains 2 registers (that is, $F = 2$), but we will use F throughout this discussion to ensure generality.

There are two variants of Top-Down allocation tested in this report. One approach, demonstrated in *Engineering: A Compiler*, computes the number of times each virtual register occurs, and uses the $(k - F)$ registers which occur most as the physical registers, and spill the other virtual registers. Another approach, which we utilized in class, computes the MAXLIVE (the number of virtual registers live at a given instruction) for each instruction, and then visits each instruction such that MAXLIVE $\geq (k - F)$, in textual order, and spills one virtual register contributing to the MAXLIVE that is used in the current instruction. The MAXLIVE is then recalculated, and this cycle continues until all instructions have a MAXLIVE $< (k - F)$.

1.3 Language and Design Choices

We used C to implement these allocators, given its speed and fairly easy to use syntax. Reading instructions from the source file is done via a quick interpreter which doesn't maintain the state of what each instruction expects. That is, we interpret an OP-code, and then read is as many arguments as are supplied, interpreting if they're inputs, outputs, immediate values, registers, etc, as they're loaded. It is entirely possible to feed invalid instructions (e.g: *loadAI r0, r1 => r2, r3*). Since it was said that there are only legal instructions fed in via input files, this was deemed okay, but this could present some problems in producing semantically correct output code.

C also has the advantage of pointers. The program represents each instruction as its own struct, which in turn has structs representing its arguments. When it comes time to actually assign registers, a combination of a Register struct and some nifty use of arrays makes the sorting, storage, and allocation of arrays quite painless.

2 Results

Implementation wise, the hardest to implement was the Top-Down allocator used in class. This was largely due to not including MAXLIVE calculations in the code from the beginning, and having to restructure the way instructions are interpreted. Testing of all of the files was done via a series of C and bash scripts. One bash script was used to automate the register-allocated code (plus calculate the time to do the allocation), while another was used to handle moving result files around. These two scripts were run through a C program that verified the results of the various file allocators (with differing register counts) with that of their original file, and generated CSVs that could be used to create charts, tables, etc.

All of our tests were performed locally and on the ilab machine, *cpp.cs.rutgers.edu*, using the Linux program *time* and recording the **real** time measurement.

Block Name	Number Cycles	MAXLIVE	Average MAXLIVE	Number of Registers
block1.i	96	5	3.3	50
block2.i	81	12	7.9	44
block3.i	47	6	3.3	19
block4.i	36	6	3.4	19
block5.i	75	5	3.9	49
block6.i	54	4	2.9	28
report1.i	65	14	9.5	57
report2.i	62	5	3.6	37
report3.i	52	13	5.3	27
report4.i	46	5	3.8	32
report5.i	50	19	9.0	41
report6.i	52	5	4.1	43

Figure 1: The number of cycles, registers, and MAXLIVE for each of the blocks used in testing with an unbounded number of virtual registers.

2.1 Bottom Up

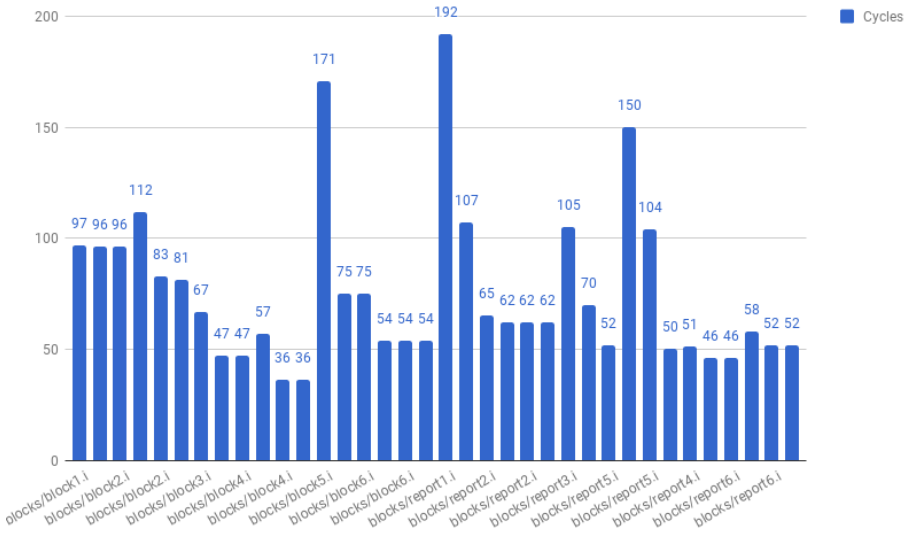

Figure 2: In order, the number of cycles for the bottom-up allocator for 5, 10, and 20 registers for each of block1, block2, . . . , report 6. i.e: column 4 is 5 registers for block 2.

Observing the number of cycles required as the number of registers increase is unsurprising at first glance – less registers are required to be spilled as k increases. Since Bottom-Up allocators require no Feasible set F, all k physical registers are able to be used throughout the allocated program. In blocks where MAXLIVE is $< k$, then, we find that the performance of the allocator is equivalent to the performance of the block without allocation, in terms of cycles. The upside of using the allocator, then, is the reduction in the number of registers: in *block1*, for example, we can do with $k = 6$ registers and achieve the same number of cycles as the program written using $k = 50$ registers.

When $k = 5$, we experience the highest number of cycles compared to when $k = 10$ or $k = 20$, as previously discussed. The extent to which this difference occurs, though, can be dramatically altered by how often MAXLIVE $> k$. In *block2*, *report1*, and *report5*, then, it makes sense that we see a dramatically higher number of cycles for $k = 5$ than when $k = 10$, since the number of instructions with MAXLIVE > 5 is larger than the number of instructions with MAXLIVE > 10. The number of instructions in which MAXLIVE exceeds k is the biggest factor in how many extra registers need to be spilled. It makes sense, then, that in *block1* we see such a small increase in the number of cycles required when $k = 5$ compared to when $k \geq 6$ – only one instruction has a larger MAXLIVE than 5, and so we only need to spill registers in that instruction.

This is a major upside when writing a compiler, so long as we assume that the values aren't intertwined too much. If users of the language are conscious of this and use variables that are not distributed throughout a block, the code produced can be very efficient even on machines with a low overall register count. This means that for Bottom-Up register allocators, it seems to be best to have many variables that are used quickly. Having variables that can be used once to perform a calculation and then used in the calculation following allows for an overall low MAXLIVE, but highly efficient register allocation scheme without much spilling, even when allowing hundreds of virtual registers.

Another upside of the Bottom-Up allocator is that it runs linearly through the instruction block. That is, given n instructions, it executes in $\Theta(n)$ time, as the allocator runs through each instruction, grabs a free register if available, or generates a spill instruction otherwise. The execution times in Figure 3, then, are all within negligibly close range to one-another. Running these tests several times yields execution times that show no significant statistical differences, since each testing block has roughly the same number of

instructions.

File	5 Registers	10 Registers	20 Registers
report1.i	0.007	0.006	0.008
report2.i	0.007	0.008	0.006
report3.i	0.01	0.007	0.005
report5.i	0.006	0.006	0.011
report4.i	0.007	0.005	0.005
report6.i	0.007	0.008	0.006

Figure 3: The execution times, in seconds, of the bottom up allocator for various k values.