Name: Bruno J. Lucarelli
NetID: bjl145

Report- Register Allocator Project

Discussion- Algorithms

Please note: Through the last week of the project's open period, I had a death of a family member (uncle) whom I was close with, and spent a good amount of time with his family and/or assisting with the wake/funeral preparations. I will also be at the funeral tomorrow (Monday 3/5). I am submitting this project and report in a fully-finished state, tested etc. If I had more time, I would have been able to make more improvements/improve things I may not have caught.

While my algorithms for each of the allocations follow the specified formats closely, my implementation makes use of some integral data structures. I'll specify them below for future reference:

regNodes: these (pointers to) structs store the information about a virtual register in the input ILOC program. Members include such information as the virtual ID, the physical register ID (if applicable), the memory offset at which the virtual register is stored, the status of the register (spilled, physical, active and an input/output), and the next regNode in the linked list of regNodes. I also tacked on additional functionality as the project progressed, such as the first/last occurrence of the virtual register (by instruction number) and the next instruction from the current line which the virtual register shows up in. regNodes handle almost all of the heavy-duty logic, and the fact that regNodes are actually pointers to regNodeStructs means that we can also manipulate pointer variables without affecting the structs themselves.
intNodes: these are the very simple linked list nodes that store an integer and a reference to the next node. They are surprisingly versatile in this project, however, because they can represent a variety of different concepts. I use them to represent the list of a virtual register's occurrences (instructions on which the register is present), and a list of virtual registers that are alive at a given line.

With this out of the way, I can discuss each of the algorithms in more detail. While I get into the nitty-gritty of each of the algorithms, please note that *the algorithms are designed to follow the given allocation styles virtually identically*. I did not home-cook any special heuristics, and I did not optimize! Any tie-breaking that isn't specified in class or in the textbook (and thus explicitly included in the algorithm) is totally arbitrary.

Top-Down Allocation (Simple/Textbook)
I chose to do this algorithm first because it makes use of the minimum amount of information relative to the others, particularly in terms of regNode modifications. The line of logic goes as follows:

I. First, generate the regNodes for the virtual registers that show up in the program. We'll call the linked list of the program's regNodes "Head". The algorithm for this is described below, but it's actually uniform for all three allocation algorithms.
   a. Go through the file line-by-line and check if the line is a comment or blank. If it's not, identify the type of the operation by scanning the characters up until the next space and compare it to any of the operation types we scan for ("add", "loadI", etc).
   b. For a non-blank line, we can tell the form of the ILOC code by the type of operation. We're bound to find 2 to 3 integers on the line (for example, "r3 and r4", or "1024 and r0"). We ignore constants and identify registers depending on the form of the code. We add a regNode to the list given the virtual ID of the register, if it's not already in Head's list.
   c. Anytime a regNode is already in the list, we update its list of occurrences by adding an intNode to its list of occurrences (firstOcc) matching the current instruction number.

II. With the information about the regNodes and list of occurrences generated, we now proceed to actual allocation. For this top-down algorithm, we allocate physical registers to virtual ones by descending order of occurrences.
   a. We count the number of virtual registers in Head (ignoring r0). Then we compare it to the number of physical registers given by the user.
      i. If we have at least as many physical registers as (non-r0) virtual registers, we go through Head's list and provide a physical register starting at r1 to each of the virtual registers (again, ignoring r0).
      ii. If we have fewer physical registers than virtual ones, we set aside r1 and r2 as the feasible registers. Then we start allocating from r3 (if applicable).
   b. If, as above, we had fewer physical registers than virtual ones, we begin top-down allocation.
   c. We call a sorting function that sorts an array of pointers to regNodes by descending order of occurrences. Then we go through that array and start allocating physical registers to each one that we come across, until we've run out of physical registers.
   d. After that, we go through Head and provide decreasing negative offsets to each of the non-physical-status registers remaining. We mark them as "MEM" indicating that they are to be spilled.
   e. With that, the registers have physical and spilled statuses with corresponding physical register ID's or negative offsets.

III. With the registers allocated, we again proceed to go through the file line-by-line and look for valid lines. This part will be identical to the output for the lecture-style top-down allocation, meaning that I'll detail it only here.
   a. At each line, we similarly check the operation type, and determine which virtual registers/constants are present.
   b. For each input register that is spilled (status MEM), we fetch it into a physical register. This means that we set its physical ID to the first available feasible

register (either r1 or r2) and perform the loadAI output into that feasible register. If an input isn't spilled, we don't do anything in this step.

c. If the output's virtual register (if any) is spilled as well, we'll have r1 as the output register.
d. We perform the output for that line, converting virtual register references to physical register references (either feasible or allocated registers). If virtual r5 is spilled to offset -8 and virtual r6 is allocated in physical r3, and if virtual r7 is spilled to offset -12, the operations are (for add r5, r6 => r7):
    i. loadAI r0, -8 => r1
    ii. add r1, r3 => r1
    iii. storeAI r1 => r0, -12
e. As implied above, we spill a register that was assigned as being spilled to its respective offset.
f. Doing the above for each line, we form the ILOC code using top-down generated information. If we have more than enough physical registers, then the ILOC code is identical line-to-line besides the labels of the registers.

Top-Down Allocation (Lecture):

This is actually quite similar to the above, but with the addition of an intNode list called liveList. liveList keeps track of the virtual registers live at each line, and is updated at each line. It's used to track MAXLIVE and determine when, during our allocation pass, we spill registers.

I. Generate the list of regNodes, Head, as before.
II. Perform the top-down allocation, but line-by-line this time for MAXLIVE considerations rather than by just the number of occurrences in general. Note that we just assign virtual registers to physical ones starting at r1 if we do have more than enough physical registers.
    a. From the first instruction (ignoring r0), keep track of the list of live registers at each line in liveList.
    b. Create a sorted list of regNodes similar to the first algorithm, but we use live range as a tie-breaker. Live range is measured as the difference between the instruction number of the last instruction and first one in which the virtual register appears. Unlike the previous one, we sort in ascending order of number of occurrences, and a greater live range puts the regNode earlier in the list. Also, any regNode that doesn't *have* another instruction is that of a dead virtual register, prioritizing its placement earlier in the list. This list is "first to spill" rather than "first to allocate".
    c. When we get to a line where MAXLIVE > number of allocatable registers:
        i. Determine how many registers must be spilled on this line by iterating through liveList and subtracting the number of physical registers.
        ii. Go through the sorted register list and spill the appropriate number of those registers that are also in liveList, until we don't need to spill anymore. Spilling includes providing an offset to the given register, etc.
    d. After populating liveList and spilling, we perform cleaning and more allocation.

i. Remove any of the references in liveList whose corresponding regNodes are spilled.
ii. Remove any of the references in liveList who are at or past their last physical instruction.
iii. Using a list of physical registers and their statuses (USED or FREE) called physStatuses, free any of the physical registers that were assigned to "dead" registers (physStatuses[physId – 2] = FREE).
iv. Go through liveList and for each of the registers that hasn't been spilled or allocated yet, assign a physical register ID from physStatuses and mark the physStatuses reference as USED. Make corresponding regNodes have PHYS status and the given physical ID.

III. Perform output at each line identical to how we did in the first top-down allocator, as we now know which regNodes are spilled and their offsets, and which are allocated in which physical register.

Bottom-Up Allocation:

This method is radically different to the previous ones in terms of how the regNodes are levied. However, it still makes use of the same data structures, and even reuses liveList.

I. Generate Head from the file as before.
II. Perform allocation AND output at each line dynamically as follows:
   a. Obtain up to three virtual register ID's, the operation type, and up to one constant for the operation.
   b. Update the nextInstr member of each regNode in Head. Given the current instruction number, we update the number of the next instruction. We include/update this member dynamically because the allocation/spilling section sorts regNode references by descending order of next instruction, in order to spill registers sequentially along the array.
   c. Update liveList for the current line, and also assign a status to each of the "active" registers on the line. We update the status of each active regNode for the line (partaking in the operation) such that we know that it's not only active, but whether it's in a physical register already or whether it needs one (spilled or brand-new).
   d. Spill, fetch, and assign physical registers all in one step and in that order. Then perform the actual operation for the line after those two. Do note that this spill -> fetch -> assign cycle seems counter-intuitive, but it follows the general order of ILOC code.
      i. First, attempt to assign however many physical registers as possible to the active INPUT registers that need them. We look through the physStatuses list and look for any indexes with a "FREE" status, updating the regNode(s) for the active regNode(s) and the physStatus index.

ii. Then, if we still have more registers we need to spill for both the INPUTS and OUTPUTS, we make a sorted array of regNode references in descending order of nextInstr. Thus, we spill whichever non-active regNode references have physical registers, and go for the ones with the latest next instruction.
    1. If we're spilling a register that hasn't been spilled yet, we give it a negative offset and then perform the output for a spill operation (storeAI). This is the "Spill" step.
iii. For the above steps, each time we allocate a physical register to a register that was previously spilled, we perform the output for a fetch operation (loadAI). This is the "Fetch" step.
iv. Then we perform similar logic for the OUTPUT register, etc etc.

e. With the regNodes on this line primed accordingly, we perform output for the line given the operation type, possible register(s) 1-3, and possible constant. If reg1 == 5, reg2 == 6, reg3 == -1, constant == -1, and op == LOAD, then we would do:
    i. load r[physical_id of r5] => r[physical_id of r6]

Tables- Results (Number of Cycles)

Simple top-down allocation

	# cycles (default)	**# cycles (5 registers)**	**# cycles (10 registers)**	**# cycles (20 registers)**
Report Test #				
1	**65**	**497**	**457**	**324**
2	**62**	**382**	**326**	**221**
3	**52**	**318**	**263**	**131**
4	**46**	**252**	**217**	**143**
5	**50**	**364**	**297**	**168**
6	**52**	**357**	**322**	**253**

MAXLIVE/Live-Range top-down allocation

	# cycles (default)	**# cycles (5 registers)**	**# cycles (10 registers)**	**# cycles (20 registers)**
Report Test #				
1	**65**	**425**	**266**	**65**
2	**62**	**289**	**62**	**62**
3	**52**	**228**	**115**	**52**
4	**46**	**237**	**46**	**46**
5	**50**	**308**	**199**	**61**
6	**52**	**217**	**52**	**52**

Bottom-up allocation

	# cycles (default)	# cycles (5 registers)	# cycles (10 registers)	# cycles (20 registers)
Report Test #				
1	**65**	**250**	**153**	**101**
2	**62**	**87**	**82**	**72**
3	**52**	**125**	**86**	**58**
4	**46**	**81**	**66**	**56**
5	**50**	**208**	**147**	**75**
6	**52**	**101**	**85**	**75**

Tables- Execution Times

Simple top-down allocation

	milliseconds to execute (5 registers)	milliseconds to execute (10 registers)	milliseconds to execute (20 registers)
Report Test #			
1	**1.222**	**0.521**	**0.582**
2	**0.830**	**0.470**	**0.477**
3	**0.721**	**0.508**	**0.586**
4	**0.967**	**0.528**	**0.546**
5	**0.849**	**0.572**	**0.661**
6	**0.752**	**0.513**	**0.752**

MAXLIVE/Live-Range Dependent allocation

	milliseconds to execute (5 registers)	milliseconds to execute (10 registers)	milliseconds to execute (20 registers)
Report Test #			
1	**1.632**	**1.527**	**1.047**
2	**0.969**	**0.855**	**0.937**
3	**0.861**	**0.812**	**0.785**
4	**0.923**	**0.826**	**0.769**
5	**1.106**	**1.076**	**0.900**
6	**1.052**	**0.812**	**0.913**

Bottom-up allocation

	milliseconds to execute (5 registers)	**milliseconds to execute (10 registers)**	**milliseconds to execute (20 registers)**
Report Test #			
1	**1.684**	**1.883**	**2.489**
2	**1.157**	**1.182**	**1.413**
3	**1.037**	**1.196**	**1.394**
4	**0.945**	**1.040**	**1.223**
5	**1.203**	**1.349**	**1.536**
6	**1.210**	**1.339**	**1.616**

Discussion- Results and Conclusions

Looking at the provided tables, I actually yielded some pretty fascinating observations about the allocators. I sum up specific observations below, and discuss them in greater detail underneath each.

1) *The simpler the allocator algorithm*, the smaller the execution time.
    a. It's plainly-obvious that the simple top-down allocator used up the least time for all of the reports, while the bottom-up allocator tended to take more time than the others. The MAXLIVE top-down allocation was in the middle of the two. This is reasonable because...
        i. The simple top-down allocator only performs two passes through the file (one to generate regNodes, one to output operations) with relatively-few operations after the first pass (one dynamic allocation of an array and comparisons between its values), and very little logic in the second pass.
        ii. The lecture top-down allocator actually performs *three* passes through the file (one in between to update a list of live registers, spill registers, and perform I/O [for fetch/spill operations] while iterating through the output).
        iii. The bottom-up allocator performs two passes through the file (one to generate regNodes, one to perform all of the chunky logic for each line). Noticeably, a far greater amount of logical complexity occurs for each line of the file, whether or not values are actually spilled, just because so much information needs to be checked ad-hoc before making decisions about spilling/fetching/allocating registers. The unpredictable nature of bottom-up allocation means far more information is necessary to make any given decision, which shows in the runtime.
2) *The more registers provided*, the smaller the execution time for top-down allocation.
    a. Paradoxically, the more registers provided, the *greater* the execution time for bottom-up! Looking through my bottom-up algorithm, I think this has to do with

the fact that with more registers provided, come more "live" registers at each line that are never spilled or removed from the liveList and are thus checked/analyzed/iterated through. More information is necessary at each line given the unpredictability of bottom-up allocation.

3) *The more registers provided to any allocation algorithm*, the fewer number of cycles are used in the resulting ILOC for all three allocation algorithms.
    a. This makes sense- after all, more registers = fewer loadAI/storeAI operations = fewer expensive and unnecessary operations.
4) Simple top-down allocation has a steady decline as more physical registers are provided, but its yielded ILOC takes far more cycles than the results of the other two algorithms.
    a. Once a virtual register is dead in this algorithm, the physical register is also dead. So for every virtual register that dies, we lose another physical register.
5) MAXLIVE top-down allocation has a fascinating pattern that is between bottom-up and simple top-down: its resulting ILOC code will steadily decline from a significantly lower number of cycles than its simple top-down counterparts. However, once a "critical number" of physical registers is provided, it will instantly drop down to the identical number of cycles to register-register ILOC. It's like a steady slope followed by a straight drop down to a flat point.
    a. This is because once we have enough physical registers that MAXLIVE doesn't exceed that, we don't ever need to perform spill operations. So it pans out to be identical to register-register code besides where we start allocating registers.
6) Bottom-up allocated ILOC code tends to not only have far fewer cycles than the other two given far fewer registers, but it also slopes and drops similarly with a critical mass of physical registers.

My final thoughts on these observations are, as Professor Kremer says, "There is no free lunch." All three of the allocation algorithms work wonderfully in regards to correctness. However, there are trade-offs in terms of their performance: smaller execution time happens to yield a greater number of cycles in the resulting ILOC code, and greater execution time happens to yield a smaller number of cycles.

If I was looking for an algorithm that would be able to perform allocation on a massive number of register-register ILOC code, and if the execution time was more important than the efficiency of the code, I would choose simple top-down allocation. If I was looking for an algorithm that would be as efficient as possible given fewer and fewer registers, I would look at the importance of execution time. If execution time is somewhat important, I would choose MAXLIVE/Lecture top-down allocation. If execution time was less important than the number of cycles in the yielded code, I would quickly go to bottom-up allocation.

Overall, I think of them as follows: simple top-down is great for cutting down execution time, lecture top-down is a great middle-of-the-road approach for execution time and number of cycles, and bottom-up is fantastic if I'm dealing with few registers and I'm in need of a great amount of efficiency while execution time isn't a problem.

Hope you found this informative, if not a bit dry. :-)