Bottom-Up Local Register Allocation

August Herron

December 2, 2024

Problem Statement

Given a branch-free block of ILOC assembly code, and an integer $k \geq 3$, produce an equivalent block of ILOC code that uses at most k registers.

Methods

To solve the problem of local register allocation I used modified versions of the **Compute Last Use** and **Local Register Allocation** algorithms described on the Allocator Component project webpage. In the process of implementing these algorithms I made several modifications, including the addition of helpful tables and annotations, and the implementation of several different optimizations and a heuristic.

Tables Used

The tables used that are not included in the pseudo-code of the algorithms are:

- **PRtoVR:** Maps physical registers to virtual registers. Used so that you don't need to iterate over virtual registers to find the physical registers they are assigned to.
- **PRtoNU:** Maps physical registers to their corresponding virtual registers next use. Used to find the next uses and farthest next use quickly.
- **VRtoML:** Maps clean virtual registers to their locations in memory. Used to quickly check if a `VR` is a clean value and what it's rematerializable memory address is.
- **VRtoRM:** Maps virtual registers to their rematerializable value. Used to quickly check if a `VR` is rematerializable and what it's associated value is.
- **isVRSpilled:** Indicates whether a virtual register is currently spilled and needs to be restored at it's next use.

Optimizations Implemented

I implemented three optimizations for reducing the total number of cycles my output code uses. These optimizations take advantage of the fact that some values do not need to be stored to memory when they are spilled. A regular spill and restore takes 6 cycles, but my algorithm reduces it in the following cases:

- **Rematerializable values:** These values do not need to be stored to memory on a spill, since it is cheaper to just restore them with a single `loadI`, which only takes one cycle. To do this I used the mapping `VRtoRM` described above. Using this table I could determine if a `VR` is rematerializable and, if so, bypass my spill code. Then, when it needed to be restored I can use the value stored in the table to insert the corresponding `loadI` instruction. This replaces 6 cycles of work with 1 cycle.
- **Re-spilled values:** These are values that have previously been spilled to memory and so do not need to be stored again to memory. To do this I used the mapping `VRtoML` described above, putting the address of a value into the table when it is first spilled. Using this table I could determine whether a `VR` was stored in memory (due to a previous spill) and so bypass my spill code. Then, on a restore I could use the value stored in the table as the address for my `loadI`/`load` pair. This replaces 6 cycles of work with 3.
- **Clean values:** These are values created by a `load` instruction from user memory. Since the value resides in memory already, it does not need to be spilled and can be restored from its location in memory, as long as it's clean. We assume it is clean until there is a `store` instruction before it's next use. To do this I added an additional annotation to my IR, `NS`, which stores the index of the next store instruction. I computed this in my **Compute Last Use** algorithm by keeping track of the last `store` seen. Once I had this annotation, I was able to use the mapping `VRtoML` to store the memory locations of values created by a `load` instruction, and quickly check if a value is clean and what it's address is. Then, when processing use operands, I check if the corresponding `VR` has a `store` before it's next use, and if so set it's memory location in `VRtoML` to invalid. This replaces 6 cycles of work with 3.

Register Spilling Heuristic

After implementing the optimizations for rematerializable, re-spilled, and clean values, spills are no longer equal in cost. When spills were equal cost it was clear that choosing to spill the register with the farthest next use provided the best results, however now it may sometimes be better to spill a closer, but cheaper, value. Determining the choice of which register to spill is NP-Complete, and so we must rely on heuristics to get us good results, even though they may not always be optimal. The heuristic I chose to implement for choosing which register to spill is as follows:

Spilling Heuristic

1. **Identify candidate registers:**
   - Iterate over physical registers, identifying the ones with the furthest next use for **dirty values**, **clean values** (if one exists), and **rematerializable values** (if one exists).
   - Call these registers `maxNU`, `maxClean`, and `maxRemat`.
2. **Determine spilling order:**
   - If `maxRemat` is valid and its next use is at least **20%** of the way to `maxNU`'s next use, spill `maxRemat`.
   - Else, if `maxClean` is valid and its next use is at least **50%** of the way to `maxNU`'s next use, spill `maxClean`.
   - Otherwise, spill `maxNU`.

The reasoning behind this heuristic is that while it may sometimes be better to spill closer, but cheaper, values, it is usually not worth it if the maximum dirty value is significantly further away. This heuristic ensures that rematerializable and clean values are within some reasonable distance from the maximum dirty value before they are chosen to be spilt. Since rematerializable values are the cheapest, they have a greater chance to beat spilling the maximum dirty value, and so they can be closer. Clean values are in between the cost of rematerializable and dirty, and so can be closer than dirty but not as close as rematerializable.

Asymptotic Complexity

In both my **Compute Last Use** and **Local Register Allocation** I loop over instructions in my intermediate representation. These instructions are linear in the number of lines of the input block of code. I also allocate tables and initialize them by looping over them. The size of the tables are determined by either k, the largest source register, or the largest virtual register. Since these are at most linear in the size of the input block, the allocations and loops are linear. In each of these loops I only ever do a constant amount of work, with conditionals, table accesses, creating structs, and looping over physical registers (k is constant in the size of the block). Thus, the asymptotic complexity of my register allocation is $O(n)$, where n is the number of lines in the input block of code.

Results

Cycle Counts

As shown in Table 1, my allocator beats the reference allocator 13 times, loses to it 8 times, and ties 14 times. My allocator thus has a win-loss ratio of 1.625, beating the reference allocator on average for the report block. However, this analysis does not take into account the magnitude of the difference of the number of cycles, just the sign of the difference (win/loss). The actual magnitude of the difference is usually pretty small, except for maybe `report06.i` with $k = 4$. With a sample size this small it is hard to make any sure conclusions about which allocator produces more efficient code on average.

Report	Uncompiled	3 Regs	4 Regs	8 Regs	12 Regs	16 Regs
report01.i	53	223/223	169/169	73/73	59/59	54/55
report02.i	56	175/175	120/120	89/104	70/80	56/56
report03.i	82	389/385	348/324	212/202	144/140	106/106
report04.i	68	169/169	164/161	144/141	111/117	85/91
report05.i	30	93/92	63/77	34/43	30/30	30/30
report06.i	105	282/282	242/278	175/174	147/148	123/124
report07.i	44	123/123	95/98	59/72	45/50	44/44

Table 1: Comparison of cycles used between my allocator and the reference allocator, where the left number is my allocator's result and the right is the reference. Green cells indicate cases where my allocator beats the reference allocator, red where it loses.

Running Time

As shown in Figure 1, both my allocator and the reference allocator run linearly in the size of the input block and are consistent with the asymptotic time complexity argued in the Methods section. Technically, my allocator ran faster than the reference, but the small sample size and small difference make it difficult to say which allocator is for sure faster. One thing I could improve to make my allocator faster is making my linked-list data structure a doubly-linked list. With a doubly-linked list I would be able to eliminate an extra pass over the instructions I do for creating a reversed list, reducing my constant factor by one. This, however, would have no effect on the asymptotic complexity of the algorithm.

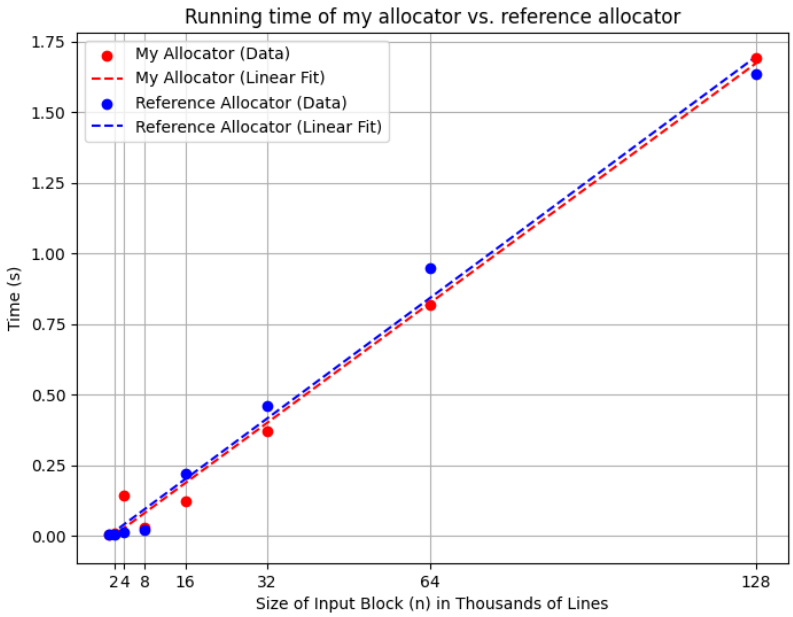

Figure 1: Plot of the running time of my allocator vs. the reference allocator on the timing blocks with $k = 15$. The time was measured by averaging 5 runs of each allocator on each input block. Linear regressions have respective R^2 values 0.991 and 0.990.

Failed Blocks

My allocator intentionally fails on blocks containing a use without a definition. Instead of initializing all uses without definitions with zero, like the simulator does, I throw an error, preventing the program from being ran. This is because a use without a definition results in undefined behavior and should never be used intentionally. These are the only blocks my allocator fails on.

Experience

If I had to start over I would of created a more robust list data structure, with double-links and a tail pointer. This would of made appending to the list constant time and allow a reversed traversal of the list easily. I would advise other students attempting this component to make sure their scanner/parser is linear first, before working on the allocator. I got stuck for a while trying to figure out why my allocator was quadratic, and it turns out it was actually my parser that was quadratic because it was using a linear list operation.