USCC
Programming Assignment 6 – Register Allocation

By: Sanjay Madhav
University of Southern California

Introduction

LLVM implements register allocation as a `MachineFunctionPass` – this is like a regular function pass, but runs after both instruction selection and scheduling has completed.

LLVM also provides a `RegAllocBase` class that custom register allocation passes can inherit from. A custom register allocation pass using this interface can determine the order virtual register intervals are assigned to physical registers, as well as specifying how virtual registers should be split or spilled if no physical register is free.

This assignment focuses on customizing the order register intervals are given physical register assignments. The current implementation in RegAlloc.cpp is copied from LLVM's `RegAllocBasic` allocator, which generates valid allocations with no splitting. The order registers are allocated is based on their spill cost; virtual registers with the highest spill cost are allocated first.

Instead of this basic approach, you will implement part of a Chaitin-Briggs graph coloring allocator. Specifically, you will write the code that generates the interference graph from intervals and then simplifies the graph. Recall that during the simplification step, any virtual registers that are trivially colorable (with <k neighbors) are removed from the graph and pushed onto a stack. If no trivially colorable registers remain, then the virtual register with the lowest spill cost is removed from the graph and pushed onto the stack.

Once all the virtual registers are removed from the graph, you will tell LLVM to attempt to assign registers in the reverse order the registers were removed. The code for attempting physical register assignments is given, as it comes directly from the `RegAllocBasic` implementation.

To test that your allocations are functional, you will need to request that USCC generates the assembly code. To do this, pass the `-s` flag to USCC. For example, the following will generate the `quicksort.s` assembly file:

```
$ ../bin/uscc -s quicksort.usc
```

Then, in order to assemble the .s file into a binary executable, you can use:

```
$ gcc quicksort.s
```

This will create an `a.out` binary, which you will then be able to execute on your system. The code that USCC uses to generate assembly is taken directly from the source code for the llc tool.

You can also specify the number of colors via the `--num-colors` command line option, which sets the `NUM_COLORS` global variable (the default is 4):

```
$ ../bin/uscc -s --num-colors 6 quicksort.usc
```

Implementation

All code for this assignment should be written in RegAlloc.cpp. When the `allocatePhysRegs` function is called inside `runOnMachineFunction`, LLVM will call the overridden `enqueue` functions once for every live interval. Once all the intervals are enqueued, they will then be dequeued one by one and `selectOrSplit` is invoked on every interval. This means you need to create the interference graph prior to the call to `allocatePhysRegs`. This way, the registers can be enqueued/dequeued using the ordering determined by the graph simplification step.

You will need to implement the `initGraph` function to create the interference graph and the `simplifyGraph` function that simplifies the graph according to Chaitin-Briggs. Note that these functions are already called prior to the `allocatePhysRegs` call. Any function-specific data (such as member data you may add) should be cleared in `releaseMemory`.

As a simplification, you may assume that any two overlapping intervals interfere. Normally, it would be necessary to check whether the intervals both require the same type of physical register (for example, a floating-point register as opposed to an integral one). However, since all types in USC are integral, this simplification is workable.

To verify that these functions are working correctly, you should implement some debug output. The final version of your code should include debug output demonstrating the order virtual registers are removed from the graph (and whether it was because it was trivially colorable or spilled). Sample output is shown at the end of these instructions.

Once these two functions are implemented, the last step is to change the behavior of the `std::priority_queue`, which can be done by implementing a custom comparator function object (functor) that uses your ordering.

Tips

- Before you start programming, consider how you want to represent your interference graph vertices, and how you will associate a `LiveInterval` with a vertex in the graph
- To loop over all the live intervals, you can use the following code pattern (from here):

```
for (unsigned i = 0, e = MRI->getNumVirtRegs(); i != e; ++i) {
  // reg ID
  unsigned Reg = TargetRegisterInfo::index2VirtReg(i);
  // if is not a DEBUG register
  if (MRI->reg_nodbg_empty(Reg))
    continue;
  // get the respective LiveInterval
  LiveInterval *VirtReg = &LIS->getInterval(Reg);
}
```

- To check if two `LiveInterval` variables interfere with each other, use the `overlaps` member function
- To output debug information for a `LiveInterval` (including the register name and range), use the `dump` member function
- Your code should use the `NUM_COLORS` member variable, as it can be changed via command line option

Testing Your Implementation

Once you've implemented all the functionality as outlined, you will want to make sure that the testAsm.py test suite still works correctly. To execute this suite, run the following command from the tests directory:

```
$ python testAsm.py
```

Additionally, when you run USCC with the `-s` flag, you should get debug output that demonstrates the approach is working correctly. For example, the partition function in `quicksort.usc` should give output along the lines of:

```
$ ../bin/uscc -s quicksort.usc
********** USCC REGISTER ALLOCATION **********
********** Function: partition
NUM_COLORS=4
Spill candidate (neighbors=5, weight=0.003125): %vreg9 [64r,144r:0)  0@64r
Removal 0: %vreg9 [64r,144r:0)  0@64r
Spill candidate (neighbors=4, weight=0.00446429): %vreg14 [432r,480r:0)  0@432r
Removal 1: %vreg14 [432r,480r:0)  0@432r
Found neighbors=3 for %vreg15 [448r,464r:0)  0@448r
Removal 2: %vreg15 [448r,464r:0)  0@448r
Spill candidate (neighbors=8, weight=0.0167572): %vreg8 [16r,480r:0)[528B,768B:0)  0@16r
Removal 3: %vreg8 [16r,480r:0)[528B,768B:0)  0@16r
Found neighbors=3 for %vreg12 [128r,144r:0)  0@128r
Removal 4: %vreg12 [128r,144r:0)  0@128r
Spill candidate (neighbors=6, weight=0.0338809): %vreg0 [80r,416B:0)[528B,768B:0)  0@80r
Removal 5: %vreg0 [80r,416B:0)[528B,768B:0)  0@80r
Found neighbors=3 for %vreg16 [320r,336r:0)  0@320r
Removal 6: %vreg16 [320r,336r:0)  0@320r
Spill candidate (neighbors=4, weight=0.0548041): %vreg6 [48r,480r:0)[528B,768B:0)  0@48r
Removal 7: %vreg6 [48r,480r:0)[528B,768B:0)  0@48r
Found neighbors=3 for %vreg21 [192r,208B:0)[208B,416B:2)[528B,704r:2)[704r,768B:1)  0@192r
1@704r 2@208B-phi
Removal 8: %vreg21 [192r,208B:0)[208B,416B:2)[528B,704r:2)[704r,768B:1)  0@192r 1@704r 2@208B-
phi
Found neighbors=2 for %vreg19 [560r,576r:0)  0@560r
Removal 9: %vreg19 [560r,576r:0)  0@560r
Found neighbors=1 for %vreg1 [32r,208B:1)[208B,496r:0)[528B,624r:0)[624r,656B:3)[656B,768B:2)
0@208B-phi 1@32r 2@656B-phi 3@624r
Removal 10: %vreg1 [32r,208B:1)[208B,496r:0)[528B,624r:0)[624r,656B:3)[656B,768B:2)  0@208B-
phi 1@32r 2@656B-phi 3@624r
Found neighbors=0 for %vreg18 [544r,592r:0)  0@544r
Removal 11: %vreg18 [544r,592r:0)  0@544r
Assigning to physical register: %vreg18 [544r,592r:0)  0@544r
Assigning to physical register: %vreg1
[32r,208B:1)[208B,496r:0)[528B,624r:0)[624r,656B:3)[656B,768B:2)  0@208B-phi 1@32r 2@656B-phi
3@624r
Assigning to physical register: %vreg19 [560r,576r:0)  0@560r
Assigning to physical register: %vreg21 [192r,208B:0)[208B,416B:2)[528B,704r:2)[704r,768B:1)
0@192r 1@704r 2@208B-phi
Assigning to physical register: %vreg6 [48r,480r:0)[528B,768B:0)  0@48r
Assigning to physical register: %vreg16 [320r,336r:0)  0@320r
Assigning to physical register: %vreg0 [80r,416B:0)[528B,768B:0)  0@80r
Assigning to physical register: %vreg12 [128r,144r:0)  0@128r
Assigning to physical register: %vreg8 [16r,480r:0)[528B,768B:0)  0@16r
Assigning to physical register: %vreg15 [448r,464r:0)  0@448r
Assigning to physical register: %vreg14 [432r,480r:0)  0@432r
Assigning to physical register: %vreg9 [64r,144r:0)  0@64r
```

Note that the virtual registers are assigned in the reserve order that they were removed from the graph, which is what should happen. The virtual registers that the program gets may vary slightly depending on the platform. Also note that in this example, despite having several spill candidates identified during the simplification process, all virtual registers were successfully assigned to physical registers (which can happen).

Conclusion

This assignment should help give some insight into how a Chaitin-Briggs graph coloring allocator works. It's interesting to note that the LLVM register allocation system is designed more so to provide mechanisms for interesting heuristics to decide on spilling or splitting virtual registers, as opposed to attempting to find more optimal colorings. This is predicated on the idea that in larger software programs, there will always be uncolorable interference graphs, so it is better for the compiler to focus on improving its behavior when this occurs.