PA6: Register Allocation

Introduction

LLVM implements register allocation as a `MachineFunctionPass` – this is like a regular function pass, but during it both instruction selection and scheduling has completed.
LLVM also provides a `RegAllocBase` class that custom register allocation passes can inherit from. Subclasses can determine the order virtual register intervals are assigned to physical registers ***and*** specify how virtual registers should be split or spilled if no physical register is free.

uses on customizing the order register intervals are given physical register
rrent implementation in `parse/RegAlloc.cpp` is copied from LLVM's `RegAllocBasic`
erates valid allocations with no splitting. However, this basic allocator simply
descending order based on their spill cost, i.e. virtual registers with the highest
d first.

ou will implement part of a Chaitin-Briggs graph coloring allocator. Specifically,
e that generates the interference graph from intervals and then simplifies the
ring the simplification step, any virtual registers that are trivially colorable (with
emoved from the graph and pushed onto a stack. If no trivially colorable registers
ual register with the lowest spill cost is removed from the graph and pushed onto

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 help understand what's happening in the register allocator, there's quite a bit of debug output. To ask USCC to generate assembly code (and run your register allocator), you need to pass in the `-s` flag. For example, these arguments in Visual Studio Code will allocate registers and generate assembly for `quicksort.usc`:

```
"args": ["-s", "quicksort.usc"]
```

Notably, you will see output but this is just the basic register allocation order.

However, this only tests that you are generating *some* assembly and there is *some* debug output. To confirm the assembly isn't garbage and the code works, you need to run the `testAsm.py` test suite:

```
python3 testAsm.py
```

This test case will generate assembly for your machine using the register allocator and then compile this assembly into a native binary executable. Then, it will run the native binary and confirm that its output matches the expected program output.

Implementation

In this assignment, you only need to make changes to `RegAlloc.cpp`. When the `allocatePhysRegs` function is called from `runOnMachineFunction`, LLVM will call the overridden `enqueueImpl` function once for every live interval. Once all the intervals are enqueued, they will then be dequeued one by one and `selectOrSplit` is invoked on each interval.

In order to change the order the registers are allocated, you will need to create the interference graph and simplify it prior to the call to `allocatePhysRegs`. Then, since the queue is a priority queue, you can use this ordering information to control the order in which virtual registers are dequeued.

You must implement the `initGraph` function to create the interference graph and the `simplifyGraph` function that simplifies the graph according to Chaitin-Briggs. These functions are already called prior to the `allocatePhysRegs` call, so you just need to implement them.

> **NOTE**
> 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 fine.

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 `std::priority_queue`, which can be done by implementing a custom comparator function object (functor) that uses your ordering and changing the declaration of `Queue` to use your comparator instead of `CompSpillWeight`.

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:

```
for (unsigned i = 0, e = MRI->getNumVirtRegs(); i != e; ++i) {
    // reg ID
    unsigned Reg = Register::index2VirtReg(i);

    // if is not a DEBUG register
    if (MRI->reg_nodbg_empty(Reg))
      continue;

    // get the respective LiveInterval
    LiveInterval *VirtReg = &LIS->getInterval(Reg);

    // Do something with the interval
    // ...
}
```

- To check if two `LiveInterval` variables interfere with each other, use the `overlaps` member function
- For the debug output you need to add, you should just use `std::cout`, but make sure you flush the stream after each line of debug output (remember `std::end` also flushes the stream)
- 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.

Additionally, you should confirm that you see appropriate debug output when you run the `-s quicksort.usc` test. However, here are a couple of examples of what the output for the `partition` function in `quicksort.usc` should roughly look like:

▶ M1 Pro (arm64 Mac) Output
▶ GitHub Actions (x86_64 Linux) Output

Because the assembly instructions will vary depending on the host architecture, your output may not match exactly as what is shown here. But the main thing is that registers should get removed both for being trivially colorable and for being a spill candidate, and that the registers are allocated in the reverse order they were simplified from the graph.

Testing on GitHub

Once you pass all the tests locally, you should push your code to GitHub and manually run the PA6 unit tests. If they don't pass or your code doesn't compile, you should make the necessary fixes and try again.

For this assignment, there are 20 test cases. Each test case failure is -5 points.

We will also look at your output of `quicksort.usc` register allocation. The physical registers should be assigned in the reverse order in which they are removed from the graph. For example, if `%vreg5` is removed last, then it should be the first which is assigned to a physical register.

- If you do the assignment order incorrectly, the deduction is -15 points
- If you didn't implement the graph coloring allocator at all, you will get a 0

Submitting Your Assignment

Once you have a GitHub Actions run you are happy with, to submit the assignment you need to submit this form on Gradescope. You will have to provide the *full URL* for the actions run you want us to grade, and also tell us if you are using any of your four slip days.

> **DANGER**
> Make sure you submit this form, as otherwise we will not know that you have a submission you want us to grade. We only grade based on your GitHub Actions run and code on GitHub, and we will not download everyone's repo. If your code does not compile or you do not submit the form, you will get a 0 on the assignment.

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.