Learning portfolio: Graph coloring and register allocation

Lily Lin

While implementing a compiler for CPSC 411 last year we had to implement a register allocator, which is a real world usage of graph coloring that I encountered last year. It has some interesting computational properties and it's use of graph coloring is pretty interesting so I thought I could write about it for my learning portfolio. It turns out that for register allocation of real programs, the problem is so large that even a greedy approximation is too slow and an even faster algorithm is required!

What is register allocation?

Register allocation is a key pass in a compiler and represents one of the major shifts in abstraction that a compiler targetting assembly takes. A compiler lowers a high level language to a lower level language, one can represent a compiler as a series of passes that take a feature/abstraction that exists in the higher level language and represent it equivalently in terms of the lower level language.

Register allocation is the pass that removes the abstraction of physical memory. For all but the lowest level programming languages, you don't want to have to consider where your variables are physically stored because it's extremely difficult to keep track of, therefore almost all languages will handle allocation of these variables to physical registers/memory locations in a register allocation pass.

What is the register allocation problem and why is it important?

So as mentioned in the last paragraph, the register allocation pass just has to assign each variable to physical memory, easy right? We obviously can't have two variables living in the same location so we can just give every single variable it's own address in memory. 64 bit architectures have massive address spaces so we actually can do this, and that will in fact give you a register allocator. So what's the problem?

Not all storage is created equal, some are fast and some are slow. Typically the idea of CPU design is to have very fast storage very close to the CPU, and as you physically get farther and farther away from the CPU storage gets slower but much much larger. What we could do is allocate everything to faraway RAM memory where we have basically "infinite" space, but our resulting programs will be **slow**. **Extremely slow**. Orders of magnitude slower.

So why not just put everything into the faster memory? Well for that we have very limited space, with the fastest memory being registers, which we only have around 15 of at a time. Do we only get 15 variables stored in the fast registers then?

No! The key point behind register allocation is that variables can live in the same register, *as long as they aren't being used at the same time.* So the register allocation problem then becomes like an exam scheduling problem, where two students (variables) can't have two exams for classes they're in (usage overlap) scheduled into the same timeslot (register).

We've seen in class that this exam scheduling problem becomes a graph coloring problem, which we can then solve using a greedy algorithm, but it turns out that register allocation is actually different. In math 441, we've discussed many NP-complete problems, problems which take exponential time to solve optimally and quickly become unfeasible to solve in a reasonable amount of time. Usually this leads us to approximate them using things like linear programming or greedy algorithms for combinatorial problems. Register allocation actually goes even further, it is in fact *impossible* to write a program that can produce optimal register allocations for all programs. The register allocation problem is in fact "unsolvable"

Why?

Why is register allocation hard?

Most people have heard of the halting problem, which is a problem proven by Alan Turing in 1986 to be impossible to write a program to solve. Impossible. Not just slow, but impossible to write if you want it to actually be correct and stop on all inputs.

A lesser known fact is that the halting problem is in fact a special case of Rice's theorem, which states that all non-trivial semantic properties of a program are undecidable, meaning you cannot write a program (Turing machine) that can produce the right result and always halt. Rice's theorem is a massive buzzkill in general because it means that anything cool our compilers want to do is actually *impossible* to do. Sucks.

So what should our compilers do then? We have to approximate our register allocation problem by another problem that hopefully produces similar results, and we can do that by approximating the step that is impossible under Rice's theorem, which is knowing which variables have overlapping usage.

The actual algorithms employed by compilers to approximate usage conflicts are varied and complicated but the basic idea is 1. Figure out how long a variable is "alive" for, which is essentially the lines of code between when it is first used and when it is last used 2. Produce one of those liveliness ranges for all variables 3. If two variables have overlapping liveliness ranges, then they are in conflict and cannot be allocated together

So this then results in a set of nodes and a set of conflict edges, which we can solve using graph coloring, with the caveat that we have slightly different colors. We have a set number of colors for registers, and then an "infinte" number of colors for locations that get allocated off into RAM (which is called *spilling*).

What algorithms do real compilers use?

Our first instict is probably to use a greedy algorithm like largest first, but it turns out that this is actually too slow! Our register allocator will need to assign many many locations so even the sort is too costly for the compiler. So what do we do?

In many other real world compilers use a different heuristic for choosing which nodes to color first, one that only takes linear time. The idea is based on the fact that if you have 32 colors and a node that has less than 32 neighbors, then as long as you can color everything else, you must be able to color that node. Now this is always true, but it fails to hold if we remove more than one node, for example . However this still gives a good heuristic on which nodes should be able to be colored last, so instead of doing an expensive sort what we can do is just repeatedly remove nodes that have fewer than k edges and color those last

This gives us the algorithm 1. Remove a node that has fewer than k edges 2. Check if removing that node made another node now have fewer than k edges, if so then go back to 1 3. Otherwise all nodes have k or more edges, so remove a random one and to go 2 4. Once all nodes have been removed, color them in the reverse order that they were removed in

In CPSC 411 the book describes the above algorithm, but since it's just a compiler for a course we were let off easy and told to just implement the simpler greedy smallest last coloring algorithm.

This paper customizes the `gcc` allocator to instead use an ILP and an optimizer to solve the graph coloring problem to see the differences in compiler runtime and produced binary runtime, and surprisingly it found that the coloring heuristic is in fact 99.9% able to color as well as the ILP version despite being much faster to run. In fact if the ILP is unable to consider additional properties of the register allocation problem (ie: more information from other analysis passes), the resulting ILP allocation is actually worse! Even with the addtional aspects added the optimal coloring doesn't always beat the heuristic based coloring, for example in cases where the heuristic is able to correctly assign a register to an "ideal register" at the cost of causing many spills, knowing that this "ideal register" would be so good for performance that the extra spills don't matter. The ILP is unable to incorporate this and thus loses in some programs!

The key observation that this paper makes is that this heuristic and it's implementations in modern compilers have the benefit of being customizable to the register allocation problem and the specific details of it, while the ILP version is "just" a graph coloring with some of the extra details tacked on, so the benefit of a slightly more optimal coloring is lost in the inability to consider real world details. It's interesting to see how what we learn in class might seem like it would be "optimal", but once it reaches the real world and its actual applications of these problems we might see that these algorithms can be beat by much simpler, purpose built algorithms like this register allocation heuristic.

So how does this algorithm compare to other strategies?

If I have time I'll try to find out, but that'll be for another learning portfolio entry!