编译原理PA5

程序实现

本阶段的主要任务是构造相干图，采用启发式算法进行染色，然后将各个寄存器分配给变量。我实现的是实验要求的无spill版本。

框架中通过内存传递参数的问题

框架中设计的`MipsSubroutineEmitter`令人困惑，它没有正确地处理参数在内存中的情况，可能是因为助教没有测试过不少于五个参数的函数。原框架把待传递的参数放在了调用者函数的栈帧中，而被调用者仍然按照参数在被调用者的栈帧去寻找参数。此外，在传参语句被遍历到的时候，被调用函数的栈帧大小有可能是无法确定的（至少当前函数的栈帧大小还不能确定，因为寄存器分配还没有完成，这样就无法完成递归调用），因此无法直接把参数放到栈帧的正确位置中。我不打算实现拉链回填，因此我采取的办法是，把被传递的参数放在下一个栈帧的顶部，也就是`sp + (i - numArg)*4`的固定位置上，然后由被调用者根据自身的栈帧大小再把参数拷贝到正确位置上。这部分拷贝代码要添加在`MipsSubroutineEmitter`的`emitEnd`函数中。

以上是实现的一些需要注意的地方，下面的内容就相对显而易见。

相干图的构造和染色

我构建了一个相干图类`InterfereGraph`，节点用`Set<Integer>`储存，边用`Map<Integer, Set<Integer>>`储存，每个`Integer`代表一个节点，它的具体数值代表变量的编号（TAC）或者负的颜色编号（MIPS寄存器，变符号以作区分）。这里的颜色编号我们规定采用对应寄存器在`emitter.allocatableRegs`数组中的下标。

根据课堂讲义，当在变量A的定值语句的$\mathrm{liveOut}$集合中出现了变量B（$A \neq B$），那么相干图中A和B两点间有一条边，代表这两个点不能放进同一个寄存器。本实验中，假设寄存器足够。此外，函数入口基本块的liveIn也需要两两连边，原因将在本节最后叙述。

调用`InterfereGraph::colorWith`染色完毕并返回染色结果（节点编号到颜色编号的映射），再遇到TAC变量就可以直接查表得到它所应处于的寄存器了。

需要注意如下情况，这些问题是在解决扩展框架：

- 在被处理的语句序列中，存在TAC的变量，也存在MIPS的寄存器，而后者的颜色已经预先确定，因此只需要对其他节点进行染色，染色过程中注意不要与后者颜色冲突；
- TAC中没有在被调用函数中显式地为参数定值，因此需要在开始处理各条语句之前专门进行处理。这里我一开始时仿照`BruteRegAlloc`，当寄存器被使用时才向寄存器中加载储存在内存中的参数值，直到后来才发现这样处理是错误的，需要在函数入口出就为它们分配好寄存器。此外，这些参数之间也不能占用相同的寄存器，因此在相干图中这些节点也需要连边。而这一处理也并不复杂，只需要对liveIn中各个节点两两连边即可。

调用者保存寄存器

最后，考虑`HoleInstr.CallerSave`和`HoleInstr.CallerRestore`两条伪指令。对于前者需要手动把Caller-Saved的参数保存到栈里。要被保存的寄存器要满足：(1) 符合Caller Saved约定；(2) 当前所储存的变量在该语句的$\mathrm{liveOut}$中。把保存这一步的所有寄存器保存一下，当遇到`CallerRestore`的时候，把这些寄存器再从栈中恢复出来就可以了。

运行测试

最终所有的程序都可以被Decaf正常编译，大部分程序可以直接用Spim运行，唯独`mandelbrot.decaf`需要在命令行中指定`-ldata`参数以放宽内存限制，我放宽为2000000后可以正常运行出结果。我和两个同学交流了一下，发现与是否采用了完整框架有关，我用的是本学期迭代开发过的版本，就无法在模拟器的默认参数下正常运行了。

算法比较

观察`BruteRegAlloc`，我认为它难以"聪明地"管理寄存器中存储的变量，会引入一些不必要的内存存取，因此不仅需要更多指令，而且会因为更多的访存而变慢。因此，对于原贪心算法与新实现的相干图染色算法，我打算从指令条数和运行时间两个方面进行比较。比较结果如下：

程序名	指令条数（贪心）	指令条数（染色）	执行时间（贪心）	执行时间（染色）
`sort.decaf`	1877	1537	0.688s	0.469s
`mandelbrot.decaf`	1440	1238	6.444s	5.795s
`rbtree.decaf`	2065	1808	2.403s	1.605s

注：指令条数采用生成的`.s`文件行数来估计。

结果是贪心算法~~不负众望地~~需要更多的指令和更长的运行时间，验证了相干图染色算法的优越性。