优化文档

author：21373405 周靖宇

代码生成

在代码生成中，我做了一些显而易见的简单优化，

1. 在对`llvm`进行计算的时候，如果计算的两个`value`均为常数，直接返回一个新的常数为他们的计算结果。如果计算两旁的值为`const`直接将其值取出来即可。
2. 在进行判断时，我也对比较计算进行了类似的操作，可以简化比较结果的获得。

中端优化

中端优化我做了`Mem2Reg`，`GVN`，死代码删除，函数内联。

构建`CFG`

`CFG`的重要性不必多言，他站在了全局的视角去对编译器的资源进行了分配。下面是我对其的主要实现。

删除不可达块

不可达块的做法非常简单，不断进行`DFS`算法，将所有可到达块保存起来，其余的块即为不可到达块。

确定支配块

确定支配块的思路很简单，就是*拦路虎*思想，对于一个块，我从入口处进行`DFS`算法进行不断推进，如果遇到该块，我马上进行截断，然后继续不断进行，此时不可到达的块即为该块的支配块。

确定支配边界和直接支配关系

参考了教程的做法，不过多赘述。

`Mem2Reg`

据助教学长所言，这个优化可以说是所有中端优化之母，有了他就可以生成`SSA`式的中间代码，在做各种优化的时候都可以事半功倍。在做了这个优化之后我也发现了其特别之处：对于整数型变量而言，他取消了`alloc`和`store`，采用了并行赋值的操作，降低了io的次数，将代码转化为了严格的SSA形式，对之后的优化有着奠基的作用。

实现方法与困难

实现方法就是对着教程的实现方法学习（

教程的解读对于简单的SSA实现已经是足够，完全足以实现一个简单的`Mem2Reg`

`phi`的插入

在`Mem2Reg`中的第一个困难就是无法理解教程中的算法，尤其是对`phi`指令的插入，为此我也专门去找了助教，助教告诉了我一定的实现方法，但是我完全无法理解，主要问题还是对支配的关系理解不够深刻，在答疑结束之后，我细细对我之前的一些测试样例进行画图，构建出了支配关系图，理解了插入`phi`函数的意义。

> 节点n的支配边界时CFG中刚刚号不被n支配的节点的集合，形式化的定义是：

> $$DF(x) = \{x | n支配x的前驱节点，n不严格支配x\}$$
>
> 就是说，在此之间n对节点都是路径支配的，但是在这个集合中出现了其他的可能性，这些其实很好想的一点就是这些可能性并不是凭空产生的，所以说一个节点x，他时一个支配边界，那他必然是不止一个节点的支配边界，对于在这些块中定义的变量，如果这个节点x也对这个节点进行了定义，那么这个节点必然也是`join node`

`phi`指令应该针对的是某个变量有可能存在不同的到达定义的基本块，因为在支配链中，任何的到达定义都是可以预见的，只有在支配边界的地方，由于对该块不再支配，此时可能有另一个支配链的末尾指向该基本块，此时就需要判断是从哪个支配链的到达定义，因此教程中的算法就显得相当自然，就应该插入到某个支配链末尾的后继基本块处，这是因为此时我们不再能确定他的到达定义。

变量重命名

变量重命名也是`Mem2Reg`中一个比较有趣的点，其实这个也是相当自然的，我的操作是对每一个Value进行重命名的时候对他开一个栈，对其的支配树进行前序遍历，在遇到新的到达定义的时候就需要把他压倒栈中，当离开这个基本块的时候就把压入这个栈的到达定义取出来，这样就可以简单的完成重命名的操作，而支配树加前序遍历的操作在之后的`GVN`中也是相当常见而且自然。

后续

在完成了上面的几步之后，我意外的非常轻松的实现了`SSA`，几乎没有出现`bug`，非常轻松的完成了这个优化。

感想

对于`Mem2Reg`，我的最大的感觉就是图论，`Mem2Reg`是一个与图论相当强相关的优化，在进行了`Mem2Reg`之后，我对编译优化的整体架构发生了巨大的改变，我意识到我之前眼中的优化只不过是着眼于小局部处的小修小补，站在更高的角度才能更好的完成对编译的优化。

死代码删除

函数删除

这个非常简单，只需要删掉没有被调用的函数就ok

代码删除

版本1

死代码删除是完全基于上面的`Mem2Reg`优化做的，我的本意是对所有必须存在的语句比如`call, syscall, br, store, alloca`做`def-use`闭包，将完全无关的指令删除，这样的思路是相当自然的，但是其实也是相当低效的，举个简单的例子

```
void f(int a[]) {
    a[1] + a[2];
    a[3] + a[4];
}
```

这样的语句是完全没有意义的语句，但是我并没有对其进行删除，导致我在某个自己手捏的样例进行测试的时候发现效率相当一般，这个时候cjh同学给我提供了另一种解决思路：

版本2

从上面可以知道，我是在寻找有用的指令，然后保留下来，但是其实寻找无用的指令可能是更加高效更加果断的死代码删除方式，思路就是

```
    private void deleteDeadCode() {
        for (Function function : module.getFunctionList()) {
            HashSet<Instr> deadInstrSet = new HashSet<>();
            HashSet<Instr> records = new HashSet<>();
            for (BasicBlock block : function.getBlockArrayList()) {
                for (Instr instr : block.getInstrs()) {
                    if (instr.canBeDeleted(deadInstrSet, records)) {
                        deadInstrSet.add(instr);
                    }
                }
            }
            for (BasicBlock block : function.getBlockArrayList()) {
                ArrayList<Instr> instrs = block.getInstrs();
                instrs.removeIf(instr -> deadInstrSet.contains(instr));
            }
        }
    }


    public boolean canBeDeleted(HashSet<Instr> deadInstrSet, HashSet<Instr>
records) {
        if (this instanceof BranchInstr || this instanceof ReturnInstr || this
instanceof CallInstr || this instanceof StoreInstr)
            return false;
        if (deadInstrSet.contains(this)) return true;
        if (records.contains(this)) return false;
        records.add(this);
        for (Value value : usedByList) {
            if (value instanceof Instr instr && !instr.canBeDeleted(deadInstrSet,
records)) return false;
        }
        deadInstrSet.add(this);
        return true;
    }
```

就是去查找指令是不是被不可删除的指令使用或者本身是不可删除的指令，因此可以大大加强对死代码的审核程度，于是就完成了代码部分的死代码删除。

感想

死代码删除是每一个学习编译原理的学生都能想到的优化，但是具体的实现却互相之间不尽相同，我在这里也遇到了小部分困难，删除的效率不高，但是在同学们的帮助下，也成功提高了删除的效率。

函数内联

说来有趣，函数内联是所有中端优化中最简单明了的，即使是完全没有接触过编译原理的大一学生也完全可以理解和讲出函数内联的思路：

把函数插入到基本块中不就行了？！

说起来是容易，做起来恶心。

思路确实是非常简单的，但是由于我在`llvm`中坚持了一切皆`Value`的设计方法，我的每个`Value`之间实际上是高度耦合的，好处就是在生成MIPS并不需要根据其名字来寻找其所在的寄存器，坏处就是拷贝起来非常复杂。

函数代码拷贝

为了更好的映射，我对每个Value包括Block构建了映射表，根据其本身的值寻找得到其相对的映射，其中所有的使用关系也被替换成映射，此时可以继续保持丝丝相扣的关系。对于参数，我将形参的映射镜像设定为实参，就可以很好的把实参插入到内联的函数中。

插入新的`phi`

因为内联之后不存在`ret`语句，如果函数的类型是`int`，那我们需要把其结果映射到一个`phi`函数中，此时就需要对每个`ret`进行记录，将其存入到一个`phi`中。

基本块的分割

基本块的分割是函数内联的核心之一，因为函数往往是存在多个块，因此必须将之前的块分割开，为`b1`和`b2`，这时有一个细节需要注意，因为`phi`指令判断的块是根据其最后一个指令，而对于`branch`是跳入到块的第一个指令，因此对于原本的块`b0`，在`branch`语句中被替换为`b1`，在`phi`语句中被替换为`b2`。而对于函数内联产生的块而言，将其插入到`b1`和`b2`之间就Ok，其`phi`则插入到`b2`的开头处就ok

感想

函数内联是所有中端优化中思维难度最低的优化，但是可能是我花时间最久的优化，前前后后写了两三天，最终还好也是写出来了，但是刚开始他的效果却是一个问号，因为我到底有多少寄存器够他用？如果我把函数都内联进来了，我的寄存器怎么办？这也坚定了我写图着色优化的信心，因为真的是很有必要，如果内联函数块一直占着寄存器不放的话，我的寄存器资源会被大大浪费，寄存器分配和活跃变量分析也就配套而生。

重新构建`CFG`

由于函数内联增加了许多多余的块，我重新构建了`CFG`，为了方便之后的使用。方法和上面的相同

`GVN`

很遗憾，这个优化我并没有完全写完，我只写了一个简易的版本，由于我没有时间去写`GCM`，因此我的优化并不是非常激进的优化，思路的话非常简单，我在生成每个`Value`的除了固定的整数，都生成了一个固定的`Hash`，我只需要将算式中的进行hash标号就ok，按照字典顺序进行标号也可以保证对于`a+b`和`b+a`这种类型算式的相同，同时在进行计算的时候我也进行了简单的计算指令的优化，实在是不值一提，就不在赘述。

具体的操作

在我看来，`GVN`是除了`Mem2Reg`之外和支配关系最大的中端优化，因为在一个支配链或者支配树中，父节点是子节点的必经之路。因此在父节点的标号在子节点也是可以确定可以使用的，但是对于兄弟子树就无法进行使用了，因此我和之前的变量重命名使用了类似的方法。前序遍历支配树，在出节点的时候消去在本结点产生的所有表达式即可。

感想

非常好写而且符合直觉的优化，是在中端优化中算是非常简单的优化。但是非常有效。

后端优化

消去`phi`

基本块整合

图着色寄存器分配

因为时间关系，我没有实现教程中的图着色，在机缘巧合下，我看到了https://github.com/gyp2847399255/SysY-compiler/blob/master/%E7%94%B3%E4%BC%98%E6%96%87%E6%A1%A3.md这位学长的申优文档，了解到了一个不需要真正建立冲突图的寄存器算法，了解了基本想法之后，我采用了这个算法。

基本步骤就是对每个块进行活跃变量分析，然后对块进行具体的分析，具体步骤如下：

0. 首先建立一个寄存器的集合，表达空闲的寄存器
1. 首先存储块中的每个`Value`的最后一次使用
2. 对指令进行遍历，如果当前指令为某个之前的`Value`的最后一次使用，而且该基本块的$outset$中不含有该`value`，将其所占有的寄存器释放。
3. 如果指令是定义型指令，对该指令的结果进行一个寄存器分配，如果没有足够的寄存器，则进行权衡，进行spill操作，spill操作的准则是：1. 在in集合的寄存器不能释放 2. 计算每个`Value`的使用次数，选出不在in中最小的一个进行释放
4. 处理完基本块之后，我们选择遍历支配树，对其子节点进行分配，首先查看当前的子节点的in集合，我们可以将in集合之外的那些属于out集合的`Value`进行释放，之后再进行恢复即可。
5. 离开该基本块之后，释放所有寄存器

想了很久我对于`spill`的情况到底该怎么办，还是没有一个很好的想法，没有办法了，如果想得到一个好的`spill`效果，循环分析什么的必然少不了，而我已经没有时间了，只能去选择消去使用次数最少的`value`。。。

困难

寄存器分配的困难主要在于无法找到对应的寄存器，这往往会造成各种无法预测的错误，为了防止这种错误，我将分配释放的寄存器信息打印了出来进行具体的查看去寻找bug

一些补充

这种寄存器算法想要释放好的效果最好在拆除`phi`之前，否则在`move`指令中可能会造成活跃变量分析的bug

感想

提升效果最明显的优化，但是其实是与之前的优化相辅相成，很可惜没有尝试真正的图着色。