编译原理 PA5 实验报告

郑逢时 2016012177

2020年01月06日

1. 概要

本次实验的主要内容是实现一种基于图着色的寄存器分配算法。具体在rust框架中，我阅读了George的论文 *Iterated Register Coalescing* ，然后参考论文附录里的伪代码实现图分配算法。然后在若干benchmark上做比较实验，证实算法实现的正确性，并尝试性地探讨我们现有框架的不足与可能的未来工作。

2. 实现过程

总体而言本次PA任务量比上次大，我大概用了两天完成。算法本身比较复杂，相比往届还增加了"干涉图结点合并"这一内容，但好在George这篇论文内容详实清晰，而且附有详细到数据结构层面的伪代码，在优雅而完善的rust框架的支持下，实现算法的过程简直可以用"按部就班"来形容。从理论到实践，我的收获还是蛮大的。（~~基本就是逐条翻译伪代码~~）

2.1. 总体流程

George提出的迭代算法的原理详见论文，其novelty大概是通过多轮迭代检测的方式，来改善前人有关图寄存器分配算法的工作里对冗余move指令的消除能力。总体流程图如下所示：

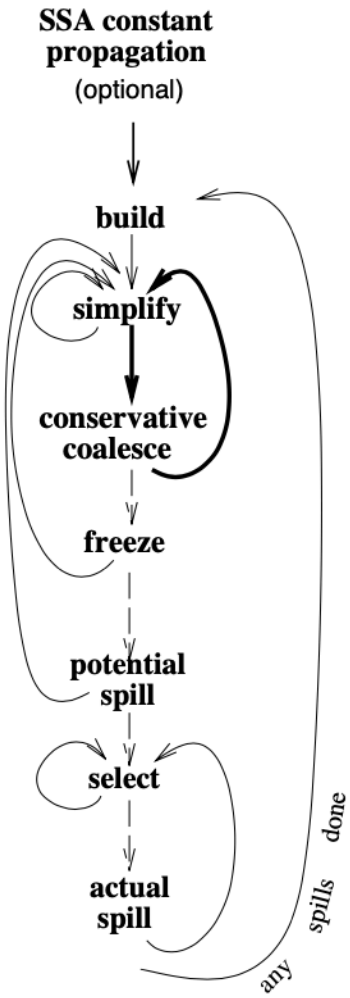


Fig. 5. Iterated algorithm.

我的实现是完全按照论文提出的伪代码框架进行的。算法主体是一个检测有无spill产生的循环，内层循环检测有没有 *简化一个图结点*、*合并一个move边*、*冻结一个move边*、和*添加一个潜在的spill结点* 这四种可能的更新，如果有则调用相应的更新过程，否则退出检测，执行染色、重写程序添加spill的语句、继续迭代分配spill产生的伪寄存器直到不再产生spill。（染色和重写程序这部分框架已实现）

```
let nodes = loop {
  let (initial, nodes) = ctx.initial();
  a.initial = initial;
  a.nodes = nodes;
  ctx.build(&mut a); // build interfere graph
  a.mk_work_list();
  loop {
    match () {
      _ if !a.simplify_work_list.is_empty() => a.simplify(), // remove one node
      _ if !a.work_list_moves.is_empty() => a.coalesce(), // coalesce one pair
      _ if !a.freeze_work_list.is_empty() => a.freeze(), // freeze one pair
      _ if !a.spill_work_list.is_empty() => a.select_spill(), // select a node
to *potentially* spill
```

```
      _ => break,
    }
  } // next iteration
  a.assign_color();
  if !a.spilled_nodes.is_empty() {
    a.rewrite_program(ctx);
  } else { break a.nodes; }
}; // any spill done: next iteration
```

2.2. 遇到的困难

主要的困难来自rust编译器的borrow checker的限制，导致很多论文伪代码中的集合运算不能直接翻译成rust程序。例如在 Combine 函数里，需要求 $nodeMoves[u] \cup nodeMoves[v]$ ，但在rust程序里，前者和后者分别是 `self.nodes[u as usize].move_list, self.nodes[v as usize].move_list` 这两个向量的借用，直接调用append方法会导致self的可变借用同时出现两次的错误。

解决方法是稍微迂回一下，多拷贝一份，“一个一个拿”：（尽管牺牲了一点点性能）

```
let mut nv_moves = std::mem::replace(&mut self.nodes[v as usize].move_list,
Vec::new());
self.nodes[u as usize].move_list.append(&mut nv_moves); // moves(u) = moves(u)
U moves(v)
```

另一个问题是论文的precolored列表和框架的表示方案不一致，框架中我们直接在图结点的字段里添加color，其类型可以是precolored、virtual、allocated三种枚举之一。最初我没注意到框架已有的color字段导致precolored列表始终为空，修改为框架中的表示方案后解决。

2.3. 简化实现

在实现完全，能通过所有测例后发现论文伪代码的几个字段其实不是必须的，它们对程序的正确性并没有影响，只是为了维持论文中提到的几个数量关系而引入，它们是：`colored_nodes, coalesced_moves, constrained_moves, frozen_moves`。将其注释掉后，结果不受影响，性能有所提高。

3. PA问答

1. 如何确定干涉图的节点？连边的条件是什么？

关于干涉图的构建，课程讲义和PA文档都讲得很清楚了。寄存器分配开始时，干涉图的结点分为预着色的和未着色的两种。

预着色的结点由汇编规范给出，例如函数的调用者、被调用者分别保存 `t0-t9` 、`s0-s7` 等，又例如 `jal` 指令的目的寄存器是 `ra`， `jr` 的源寄存器一般是 `ra` 等等。这些寄存器构成干涉图上预着好色的结点，其颜色就是与之对应的寄存器号。

未着色的结点由伪寄存器给出，伪寄存器是从三地址码翻译成汇编码时默认的占位符。

干涉图上u和v结点连边的条件是，伪寄存器u的某个定值点后变量v是活跃的，或者v是预着色的寄存器。

2. 结合实际的程序(decaf 或 TAC 程序)，比较你实现的算法与原来的贪心算法的寄存器分配结果。只从这个例子来看，两种算法哪个效果更好?

请见“分析与结论”。

4. 分析与结论

4.1. benchmark比较

为了检验算法实现的正确性，我在几个典型的benchmark上分别用蛮力算法（框架原有的Brute实现）与图分配算法编译生成汇编码，在保证汇编码在spim上运行结果一致的前提下，对比汇编码的长度和内容。

下表比较三个benchmark上用两种算法编译生成的汇编码长度：

benchmark | 代码行数	brute alloc	graph alloc
Mandelbrot	1982	**745**
Rbtree	3313	**1067**
Sort	2550	**977**

可见，图分配算法生成的汇编码的长度均远远少于蛮力算法的结果，这初步说明了图分配算法的正确性和有效性。

4.2. 具体样例分析

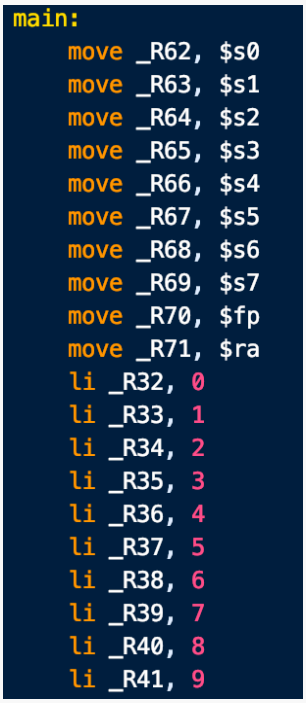
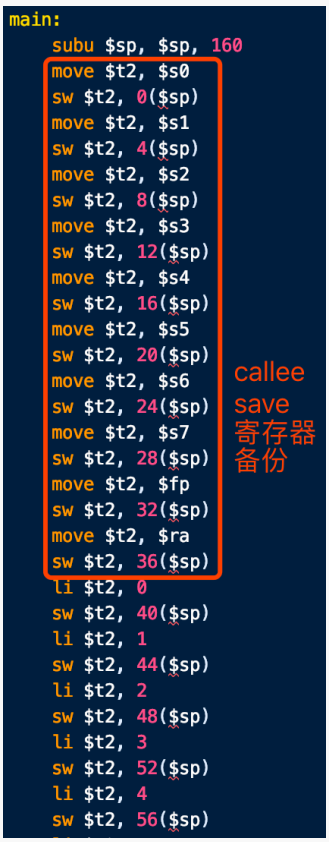
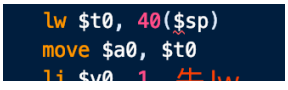
当然，只看代码行数是远远不够的，于是我构造了这样一个简单测例，试图从**汇编码的语义**上证实算法实现的正确性。测例关键代码如下：

```
static void main() {
  int r0 = 0;
  int r1 = 1;
  int r2 = 2;
  int r3 = 3;
  int r4 = 4;
```

```
  int r5 = 5;
  int r6 = 6;
  int r7 = 7;
  int r8 = 8;
  int r9 = 9;
  Print("r0 = ", r0, "\n");
  Print("r1 = ", r1, "\n");
  Print("r2 = ", r2, "\n");
  Print("r3 = ", r3, "\n");
  Print("r4 = ", r4, "\n");
  Print("r5 = ", r5, "\n");
  Print("r6 = ", r6, "\n");
  Print("r7 = ", r7, "\n");
  Print("r8 = ", r8, "\n");
  Print("r9 = ", r9, "\n");
}
```

可见，这个函数在生成汇编码后，分配物理寄存器前大概应该包含10个伪寄存器（decaf代码中的`r0-r9`），而且易知这10个伪寄存器应该都是活跃的。

为了排除常量传播优化等干扰，以更加清晰地显示出图分配算法本身的能力，我**暂时将PA4阶段的全部优化关闭**。用两种算法分别分配寄存器，所得结果如下：（代码较长，只显示节选）

	Before Alloc	Brute Alloc	Graph Alloc
行数	143	224	102
代码开头			
			

<table>
<tr>
<td>代码中部</td>
<td>

```
move $a0, _R42
li $v0, 4
syscall
move $a0, _R32
li $v0, 1
syscall
la _R43, _STRING2
move $a0, _R43
li $v0, 4
syscall
la _R44, _STRING3
move $a0, _R44
li $v0, 4
syscall
```

</td>
<td>

```
syscall
la $t2, _STRING2
sw $t2, 84($sp)
lw $t0, 84($sp)
move $a0, $t0
li $v0, 4
syscall
la $t2, _STRING3
sw $t2, 88($sp)
lw $t0, 88($sp)
move $a0, $t0
li $v0, 4
syscall
lw $t0, 44($sp)
move $a0, $t0
li $v0, 1
syscall
```

先lw再输出

</td>
<td>

```
li $v0, 4
syscall
move $a0, $t4
li $v0, 1
syscall
la $a0, _STRING2
li $v0, 4
syscall
la $a0, _STRING3
li $v0, 4
syscall
move $a0, $t8
li $v0, 1
syscall
```

直接输出

</td>
</tr>
<tr>
<td>代码结尾</td>
<td>

```
main_Ret:
move $s0, _R62
move $s1, _R63
move $s2, _R64
move $s3, _R65
move $s4, _R66
move $s5, _R67
move $s6, _R68
move $s7, _R69
move $fp, _R70
move $ra, _R71
jr $ra
```

</td>
<td>

```
main_Ret:
lw $t0, 0($sp)
move $s0, $t0
lw $t0, 4($sp)
move $s1, $t0
lw $t0, 8($sp)
move $s2, $t0
lw $t0, 12($sp)
move $s3, $t0
lw $t0, 16($sp)
move $s4, $t0
lw $t0, 20($sp)
move $s5, $t0
lw $t0, 24($sp)
move $s6, $t0
lw $t0, 28($sp)
move $s7, $t0
lw $t0, 32($sp)
move $fp, $t0
lw $t0, 36($sp)
move $ra, $t0
addu $sp, $sp, 160
jr $ra
```

还原备份到内存的寄存器

</td>
<td>

```
main_Ret:
jr $ra
```

</td>
</tr>
</table>

可见，这个函数的调用过程是很典型的。从before alloc的代码可见，函数先保存 callee save 寄存器，然后给`r0-r9`赋值，然后调用syscall打印字符串及赋好值的伪寄存器。

在brute alloc算法生成的结果中，prologue中的每句callee save被翻译成了两句指令，epilogue也同样如此，body部分的每个伪寄存器的访问都被翻译为了写存和读存，不仅体量大，而且几乎没有利用寄存器资源，运行开销也很大。

而在我实现的graph alloc算法生成的结果中，函数的三个部分都得到了很好的简化。充分利用 `t0 – t9` 等物理寄存器来存取伪寄存器，避免了spill到内存的开销。而且更值得一提的是，算法成功检测到了**冗余的move指令**，合并了prologue和epilogue处对 callee save 寄存器（`s0–s9`等）的备份与还原，使得生成汇编码的首尾都出奇的简短，总指令行数甚至比分配前的程序还少。当然这种现象有赖于rust框架的一个小细节，就是会在分配完寄存器后对汇编码进行简单的窥孔优化，优化包括了对形如 `move $s0, $s0` 这种复写move的消除。因此我们只要能正确实现对冗余move的两端以相同颜色染色，就能起到“合并”伪寄存器的效果。

这个例子和以上表格数据表明，我们的工作基本能达到George这篇论文提到的干涉图节点合并效果，并具备比较符合预期的高效寄存器分配能力。

5. 讨论与未来工作

上一节探讨了算法实现的正确性和高效性，然而我经过仔细的分析，发现图分配算法在我们的框架中的表现并不是十全十美的。具体来说，我发现的局限性有二：

5.1. 局部的“全局寄存器分配”

虽说基于多个基本块的寄存器分配算得上是一种“全局”的寄存器分配算法，但我们的框架毕竟还是只能以一个函数为单位来做到“全局”，即每个函数视为独立的寄存器分配问题，所以这种全局其实还是比较“局部的”。当涉及到一个函数调用另一个函数的时候，按照MIPS函数调用规则的约定，caller必须保存所有caller save的寄存器，即 t0-t9，被调用者callee需保存所有callee save寄存器，即 s0-s7 等。

如此一来，如果某个函数a内部调用了函数b，那么在调用语句处，框架会假定所有caller save的寄存器会被被调用者定值，直接导致调用语句后面的所有活跃变量会与 t0-t9 这么多个callee save寄存器**相干**，导致调用完后的语句难以分配到这9个寄存器，除非进行spill。但事实上不是所有callee都会更改 t0-t9 的值，所以caller那里有些定值的假定是不必要的。

又比如，某个函数a在prologue阶段，按照调用约定，一定会将 s0-s7、ra 等callee save的寄存器搬移到别的伪寄存器里（实际就是栈上内存），如果body里试图分配callee save寄存器，它们就不得不在一开始spill到栈上。所以我们的算法在很多样例里都表现出“刻意”避免分配callee save寄存器的现象。但事实上不是所有的caller都会用到所有callee save的寄存器，因此可能有些spill是不必要的。

综合以上考虑，如果一个函数内部有call的话，它实际可以自由分配的寄存器，既没有 t0-t9，也没有 s0-s7 等，实际上是所剩无几的，自然不能很充分地发挥图分配算法的性能。

由此设想，如果每个函数在生成汇编码的时候，都把自己更改了的（包括自己调用过的函数更改了的）caller save寄存器标记在头部，**作为一个冗余信息**，这样caller在call这个函数的时候，就能准确知道哪些寄存器是会在调用过程里被定值的，那么只对这些寄存器进行预着色，而不必对所有caller save都着色，这样一来可供调用者自由分配的寄存器就比以前多了不少。对 callee save 寄存器，情况稍微麻烦一些，但也应该可以类似考虑。希望未来的框架能考虑这样一个特性，目测难度也不会太大。

5.2. 汇编阶段的代码优化不足

如题。这是一个课上学习过，实践中我也经常遇到的问题。汇编码生成阶段也可能生成大量的冗余代码，例如复写 `move $t0, $v0; sw $t0, 0($sp)` ，条件判断 `slt $t0, $a0, $a1; bnz $t0, target` 等等。所以汇编码生成阶段的编译优化也是很必要的，希望未来的工作可以做的更充分。

事实上rust框架是实现了一个简单的窥孔优化的，这个前文也提到，主要是针对 `move $t0, $t0` 这样的无效复写，和一些无作用的算数指令。复写消除对我们的寄存器分配算法具有一定的意义，例如 prologue 和 epilogue 阶段备份到虚拟寄存器的那些 caller save 寄存器，如果body里没有分配到那些预着色的寄存器，它们就很可能会被迭代算法标记进 workListMoves 里，最终合并为同一个结点，即产生出 `move $s0, $s0` 这种形状的复写语句。所幸框架的窥孔优化刚好可以对付这种move，于是合并过后的代码得到了简化。

由此，我尝试在窥孔优化进行后，再重新执行一遍染色，企图让算法尽可能的利用被窥孔优化“解放”的那些寄存器，最后再进行一遍窥孔优化。试验了一些测例后，发现生成的汇编码的确能减少，但是效果并不显著（约1%），综合了编译时间的考虑，我最终放弃了这种做法。

也许未来框架有了更强大的PA5阶段优化，那样的话迭代图分配算法应该会有更多可以施展的空间。