Decaf PA5 实验报告

计 76　张翔　2017011568

2019 年 12 月 24 日

1　实验简述

本阶段实验将生成的 TAC 码分配实际的寄存器并映射为 MIPS 汇编指令，从而使 Decaf 可以在 SPIM 模拟器/实体环境上运行。

2　主要工作

此阶段主要按论文 [1] 实现了基于图染色的全局寄存器分配算法，其中 spill 无法染色的寄存器没有实现（在现有 Java 框架上实现过为繁琐），算法的其余部分均有完整实现。

2.1　算法流程

1. 对已经进行过指令选择处理的函数进行基本块划分，计算出每个基本块以及每条指令的 liveIn, liveOut 集合

2. 逆向遍历每个基本块中的语句，构建寄存器干涉图。如果遇到 move 指令，将其加入 move 工作列表。此时已经有部分虚拟寄存器为了满足调用约定等条件，已经被设定为实际寄存器，只需要将剩下的虚拟寄存器加入初始列表

3. 根据初始列表中每个虚拟寄存器的情况，若是高度数节点 (degree $\geq K$)，加入 spill 工作列表；如果 move 相关，加入 freeze 工作列表，否则加入 simplify 工作列表。

4. 开始迭代算法，依照 simplify, move, freeze, spill 的优先级对上述四个工作列表进行处理，高优先级的列表为空后才去处理低优先级的列表。Simplify 操作是从干涉图中移除节点；Move 操作调用 Coalesce 函数尝试合并一些 move 语句用到的寄存器，合并成功的加入 coalescedNodes 列表中；Freeze 操作处理低度数的 move 相关节点；Spill 操作则是选择节点加入 spillNodes 列表。四个列表皆空后，迭代算法结束。

5. 如果 spilledNodes 为空，说明可以顺利染色，此时从 selectStack 中依次取出虚拟寄存器，将其分配实际寄存器；对于 coalescedNodes，它们是被合并的寄存器，而合并操作中记录了它们的 alias，可以沿着 alias 链查找到末端（类似于 PA4 优化中的 Copy Propagation），并将其置为相应的寄存器

6. 如果 spilledNodes 不为空，加入相应的 Load/Store 指令，以将这些节点 spill 到栈上，然后转到 1，重新开始迭代（这一步没有实现）

2.2 干涉图的构建

相应的伪代码在论文 [1] 中已经有非常详细的叙述，这里大致分析一下。

首先，干涉图的节点是虚拟寄存器，在 Java 框架中就是 `Temp` 类，实现时干涉图使用边表的形式存储，即 `HashMap<Temp, HashSet<Temp>>`。对于单条语句，我们用逆向数据流方程计算出的 liveOut 集合，含义为该语句出口之后使用到（活跃）的虚拟寄存器集合，显然这些虚拟寄存器不能和当前语句 def 集合中的寄存器共用同一个物理寄存器，对此需要在干涉图中将它们连边。因此整个函数构建的干涉图的边集可以表示为

$$E = \bigcup_{\forall B \in \mathrm{CFG(func)}} \left\{ (v_i, v_j) | \forall I \in B,\ \forall v_i \in def(I),\ \forall v_j \in liveOut(I),\ v_i \neq v_j \right\} \tag{1}$$

由此可以表示节点集合

$$V = \{v | (v, \cdot) \text{ or } (\cdot, v) \in E\} \ \cup \ \text{Precolored} \tag{2}$$

注意到预染色的寄存器也作为干涉图的节点，论文中认为它们可能和其他很多节点冲突（至少预染色节点之间一定互相冲突），为了算法效率，它们的度数定义为 $N+K$（$N = |V|$，K 为可供分配的物理寄存器数量）或 ∞，且不会显式构造它们的边表。

2.3 对于框架的一些修改

2.3.1 可分配寄存器的修改

原来框架中 `MipsSubroutineEmitter` 等实现过于丑陋，对此我进行了适当修改。

对于函数调用语句 `jr, jalr`，将其等效为写入所有 CallerSaved 寄存器（25 个，比 Rust 框架中的定义少一个 `$ra`，比 Java 框架原有实现多了 `$v0, $a0-$a3`），读取参数寄存器 (`$a0-$a3`) 的指令（对于 `jalr` 还要包括它本身用到的一个寄存器）。实现时可以修改对应指令的构造函数，或者重写 `getRead, getWritten` 方法，从而假装这些指令要读取/写入对应的寄存器。之后在图染色算法中将可用寄存器设为 25 个 CallerSaved+ 原有的 CalleeSaved，染色结束后，如果没有 spill，无需使用原框架的 `HoleInstr` 强行在函数调用前后进行 CallerSaved 寄存器的保存，因为经过这种修改之后算法会选择 CalleeSaved 寄存器来保存可能被函数调用修改的寄存器。由于 MIPS 的 CalleeSaved 寄存器非常多，在很多情况下（至少对于所有样例而言）是无需 spill 的，这样进一步减少了 Load/Store 的次数。

对于 `$ra` 寄存器，与 Rust 框架的处理方式不同，这里仍然把它视作保留寄存器。如果某个函数内存在函数调用语句，就在函数入口点处保存 `$ra`，出口点处恢复即可，这个功能在 Java 框架中已有实现。

2.3.2 参数传递的修改

根据 MIPS 调用约定，前 4 个不超过 32 位的参数由 `$a0-$a3` 传递（需要保留栈帧中的位置），后面的参数通过栈传递，而原来框架的处理是在函数体开始前将 `$a0-$a3` 写入栈中相应位置，又把通过栈传递的参数读取出来并写入到栈中另一块保存当前函数临时变量的区域，这种暴力的方法在图染色寄存器分配时是完全不必要的。

我的实现是在函数开始前加上 `load _T4, 0($sp)` 这样的指令，用于从栈中取出通过栈传递的参数，此时 offset 不能确定，先置为 0。当函数分配寄存器完毕后，变量在栈中偏移量已经确定，此时采用代码回填的技术将正确的偏移量写入就可以工作了。对于寄存

器传递的参数，它们将 _T0 这样的虚拟寄存器预染色了，从而直接使用即可。一个样例是 S4 目录下的 many_args.decaf，其中有两个接受 8 个参数的函数，包含了新语言特性（如 Lambda 函数对象）作为函数参数。

3 效果比较

这里以我自己写的 many_args.decaf 为例，部分源码如下

```
static void many_args_multi_type(int a, int b, int c, string d, bool e,
                                 string f, bool(int) g, int h)
{
    Print("Sum of a,b,c is: ");
    Print(a+b+c, "\n");
    Print(d, "\n");
    Print(e, "\n");
    Print(f, "\n");
    Print(g(h), "\n");
}
```

暴力寄存器分配的结果如下:

```
_L_Main_many_args_multi_type:  # function FUNCTION<Main.many_args_multi_type>
# start of prologue
addiu   $sp, $sp, -80  # push stack frame
sw      $ra, 68($sp)  # save the return address
# end of prologue

# start of body
sw      $a0, 0($sp)  # save arg 0
sw      $a1, 4($sp)  # save arg 1
sw      $a2, 8($sp)  # save arg 2
sw      $a3, 12($sp)  # save arg 3
lw      $v0, 96($sp)  # load arg 4
sw      $v0, 16($sp)  # save arg 4
lw      $v0, 100($sp)  # load arg 5
sw      $v0, 20($sp)  # save arg 5
lw      $v0, 104($sp)  # load arg 6
sw      $v0, 24($sp)  # save arg 6
lw      $v0, 108($sp)  # load arg 7
sw      $v0, 28($sp)  # save arg 7
la      $v1, _S2
move    $a0, $v1
li      $v0, 4
syscall
```

```
    lw      $v1, 0($sp)
    lw      $t0, 4($sp)
    add     $t1, $v1, $t0
    lw      $v1, 8($sp)
    add     $t0, $t1, $v1
    move    $a0, $t0
    li      $v0, 1
    syscall
    ...
    j       _L_Main_many_args_multi_type_exit
    # end of body

_L_Main_many_args_multi_type_exit:
    # start of epilogue
    lw      $ra, 68($sp)  # restore the return address
    addiu   $sp, $sp, 80  # pop stack frame
    # end of epilogue

    jr      $ra  # return
```

中间省略的部分是一堆用于打印的 `syscall`。可以看出，暴力寄存器分配算法反复使用几个 caller-saved 寄存器，几乎每次使用之前都要从栈上读取相应寄存器的值，代码丑陋，效率低下。

```
    _L_Main_many_args_multi_type:  # function FUNCTION<Main.many_args_multi_type>
    # start of prologue
    addiu   $sp, $sp, -72  # push stack frame
    sw      $ra, 68($sp)  # save the return address
    sw      $s0, 32($sp)  # save value of $S0
    sw      $s1, 36($sp)  # save value of $S1
    sw      $s2, 40($sp)  # save value of $S2
    sw      $s3, 44($sp)  # save value of $S3
    # end of prologue

    # start of body
    move    $t0, $a0
    lw      $v1, 88($sp)
    lw      $s2, 92($sp)
    lw      $s3, 96($sp)
    lw      $s1, 100($sp)
    la      $a0, _S2
    li      $v0, 4
    syscall
    add     $v0, $t0, $a1
```

```
    add     $a0, $v0, $a2
    li      $v0, 1
    syscall
    la      $s0, _S3
    move    $a0, $s0
    li      $v0, 4
    syscall
    ...
    j       _L_Main_many_args_multi_type_exit
    # end of body

_L_Main_many_args_multi_type_exit:
    # start of epilogue
    lw      $s0, 32($sp)  # restore value of $S0
    lw      $s1, 36($sp)  # restore value of $S1
    lw      $s2, 40($sp)  # restore value of $S2
    lw      $s3, 44($sp)  # restore value of $S3
    lw      $ra, 68($sp)  # restore the return address
    addiu   $sp, $sp, 72  # pop stack frame
    # end of epilogue
    jr      $ra  # return
```

中间同样省略了一堆 `syscall`。可以看出，函数入口处先从栈中读取了后 4 个参数，后续连续的 `syscall` 操作之间基本上只涉及寄存器之间的操作，而没有额外引入 Load/Store 指令。该算法得到结果的最明显特征是对 callee-saved 寄存器的充分使用，虽然增加了函数入口、出口处保存、恢复 callee-saved 寄存器的开销，但函数运行期间的大部分操作仅在寄存器内进行，总体上能够获得较大的效率提升。从这个角度来说，图染色寄存器分配算法得到的结果是更优的。

另外，值得一提的是，依照该论文实现的寄存器分配算法还有一些类似于常量传播/复写传播的功能，比如暴力寄存器分配得到下面的代码

```
    # start of body
    li      $v1, 4
    move    $a0, $v1
    li      $v0, 9
    syscall
    move    $v1, $v0
    la      $t0, _V_Main
    sw      $t0, 0($v1)
    move    $v0, $v1
```

而图染色算法得到的结果为

```
    # start of body
    li      $a0, 4
```

```
li      $v0, 9
syscall
la      $v1, _V_Main
sw      $v1, 0($v0)
j       _L_Main_new_exit
```

很明显，多余的 `move` 消失了，而 `la` 指令的立即数也直接载入了最终的目的寄存器。

参考文献

[1] L. George and A. W. Appel, "Iterated register coalescing," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 18, no. 3, pp. 300–324, 1996.