**编译申优文章**

16231019 郑明悟

这次编译大作业的完成过程中，遇到了很多困难的和问题。但是只要认真思索，大胆实践，还是有办法的。

编译课设主要分为词法分析、语法分析、生成中间代码、活跃变量分析、生成目标代码和优化等。代码行数最长的就是语法分析、活跃变量分析、生成目标代码和优化。我将分节讨论我做的工作及其与众不同之处。

### ****语法分析****

**语法分析的主要问题是识别语法成分、和生成中间代码。**

**识别语法成分**

**在识别语法成分时，会遇到FIRST集合相交的问题，最典型的就是变量和函数的声明：**

**＜变量定义＞ ::= ＜类型标识符＞(＜标识符＞|＜标识符＞'['＜无符号整数＞']'){,(＜标识符＞|＜标识符＞'['＜无符号整数＞']') }**

**＜有返回值函数定义＞ ::= ＜声明头部＞'('＜参数表＞')' '{'＜复合语句＞'}'**

**＜声明头部＞ ::= int＜标识符＞|char＜标识符＞**

**在语法分析程序读到类型标识符和标识符后，无法判断现在识别的是全局变量还是有返回值函数。我没有改写文法来避免冲突。因为可以看出，函数和变量都是要填符号表的内容，已经读到的类型和标识符不必丢弃。判断下一个符号是否是左括号就可以分辨具体的标识符类型。**

**具体实现上，我采用一个循环识别多个变量，当遇到左括号时循环退出，进入函数分析循环。**

**至于标识符的冲突可以使用语义分析解决，查询符号表可以判断当前处于何种语法成分。**

**生成中间代码**

**生成中间代码时，我进行了常量替换，同时采用属性翻译文法来实现对字符的处理和表达式计算。**

**常量替换方面，我直接在符号表存储了常量的值，遇到常量标识符直接替换。**

**在表达式计算时，我对表达式、因子和项设置了综合属性bool isch（是否字符单字）和string val（值）。isch通过指针引用实现传递，值通过返回值传递。**

**＜表达式＞↑isch↑val::= ［＋｜－］＜项＞↑isch1↑val1{＜加法运算符＞＜项＞↑ischn↑valn}**

**＜项＞↑isch↑val::= ＜因子＞↑isch1↑val1{＜乘法运算符＞＜因子＞↑ischn↑valn}**

**因子直接根据内容计算是否单字和常量值即可；对于项，当isch1为真且没有跟乘法运算符时isch也为真，当va1…valn均为常量时val为计算结果，否则为中间变量名。同理，在表达式中，当isch1为真，没有正负标号以及没有加法运算符时isch为真。**

**isch在判断条件、赋值是否合法以及打印字符时都有重要作用。**

### ****活跃变量分析****

**流图和着色图生成主要部分想必做优化的人都做了，就不在额外赘述。这里我想提一提我的基本块生成逻辑。**

**一般来讲，基本块是从第一条扫描，确认入口语句后逐块划分完毕，随后遍历每一块，识别跳转并链接基本块。算法复杂度，因为识别跳转时需要从头遍历基本块表。**

**我自己设计了一个且能删除不可达代码的基本块生成逻辑。首先，在语法分析中，我使用数组保存了每个标签对应的中间代码下标。其次，可以明确我们的文法基本块出口至多只有两条分支，故可以递归实现。算法流程如下：**

算法：Recursively generating basic blocks and flow graphs.

函数名：gen\_block

**Input:** 当前块的起始中间代码序号pos、前一个块preblk（调用的父块的块号）；

**Output:** 新建的块的块号（当前块的相关信息存到流图类中）；

1: **If** pos已经有对应的块 **Then**

直接将preblk加入该块前驱节点

**Return**该块块号

2: **If** preblk==NONBLK（宏 -1） **Then**

说明是入口块，置当前块next1为1，next2 为NONBLK，调用gen\_block(pos+1,0)

**Return** 0

3: **If** pos==-1 **Then**

说明到达出口块，置next1和next2为NONBLK，新建并返回块号

4: **For each** 下一个入口语句前的中间代码

计算 def和use，并设置在该块中。

5: 设出口语句是m，当前块号b，初始化next1和next2为NONBLK

6: **If** m是BZ语句 **or** m是BNZ语句 **Then**

置next1为gen\_block(m+1，b)

置next2为gen\_block(label对应的坐标，b)

7: **Else if** m是GOTO语句 **Then**

置next1为gen\_block(label对应的坐标，b)

8: **Else if** m是EXIT语句 **or** m是RET语句 **Then**

置next1为gen\_block(-1，b)

9: **Else** 置next1为gen\_block(m+1，b)

10: **Return** b

**算法递归遍历，每条中间代码访问一遍可以生成流图，复杂度。起始输入形式为gen\_block(函数入口，NONBLK)。**

**另外，该算法只会将程序可达的部分划入基本块，例如有代码：**

**if(condition){**

**…**

**return(…);**

**}**

**else {**

**…**

**return(…);**

**}**

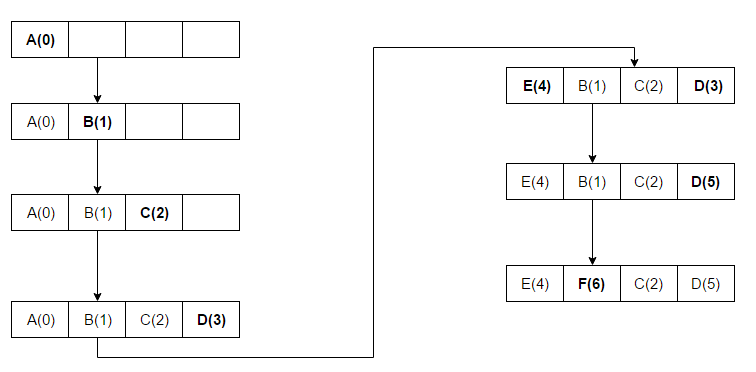
**…(\*)**

**则星号及以后的代码均不会被执行，这包括显示的函数代码和隐式的中间代码（如函数默认的返回语句）。在递归生成基本块的算法中，这些内容将被去掉，因为只会分析程序流可达的内容。**

### ****生成目标代码****

**我将主要讨论临时寄存器的分配。在一个基本块内，不跨块的局部变量和中间变量（以下简称变量）可以分配临时寄存器。一种策略是静态分配：在目标代码翻译前完成9个临时寄存器和变量的映射，未映射的变量使用访存实现赋值和使用。另一种策略是动态分配，即一个块内根据变量出现的顺序动态决定临时寄存器的分配。如果临时寄存器满了，则选择合适的变量踢出分配表（此时要存入内存），并为新的变量分配临时寄存器（以下简称寄存器）。**

**这个思想很类似操作系统课上提到的页面置换算法：在操作系统中，在虚拟内存申请一个页面时，如果物理内存已满，需要从中置换走一页。不同的是，我们的变量相当于虚拟内存（没有实体），mips内存相当于硬盘，寄存器相当于物理内存。**



**这里综合各种情况，我选择了LRU算法。当一个变量被访问时，把其提到队列的尾部。在发生“缺页”错误时，总是向内存回写队列头的变量，并把新变量放在队列的尾部。**

**没有选择OPT算法的理由是，代码书写的逻辑有很强的局部性，一个长期不使用的不跨块变量很大可能是不会再使用的变量（已经完成使命），OPT和LRU算法相差无几。这在实际的观察中也有所体现。另外还有一个十分隐蔽的理由，将在后文提到。**

**举一个例子来说明动态分配寄存器的好处，考虑以下块内中间代码：**

**ADD a b #1**

**BECOME #1 c**

**…**

**…**

**…**

**BECOME #1 d**

**#1若仅仅在表达式c=a+b中被计算，如果为其静态分配寄存器$tx，则接下来的代码中其他变量无法使用$tx寄存器，效率大大浪费。若采用动态分配策略，后面的变量可以申请到#1所占用的寄存器。即使后面重新使用了#1，也可以重新为其分配寄存器，并将之前保存的值从内存中加载。当然，在变量不多的情况下，#1也可以一直持有自己的寄存器，而无需从内存加载。**

**然而，动态分配看似美好，实则暗藏杀机。书上始终没有明确谈论临时寄存器的分配问题，在实际实现中，我们发现该方法有很多很深刻的问题，解决这些问题需要缜密严谨的思维和大胆创新的想法，接下来我将讲述这些问题和采用的解决策略。**

**问题1 编译程序编写时获得的寄存器号的有效期**

**程序编写时，理论上动态寄存器的分配应当是一个封装的逻辑块，该块接受变量名，返回寄存器号。这其中调用者无需关心这个寄存器号是如何获取的。**

|  |  |  |
| --- | --- | --- |
| **变量名→** | **LRU临时寄存器分配器** | **→t寄存器号** |

**例如，在翻译中间代码ADD #1 #2 #3时，就会向分配器请求得到三个变量的寄存器号，以便翻译成形如add $t3, $t1, $t2的形式。**

**问题在于，我们必须依次获得全部的寄存器号后才能生成目标代码。假设第一个变量调用apply\_reg获取寄存器t1，以此类推，第n个变量也调用apply\_reg获取寄存器tn。每次调用，临时寄存器分配器都会根据分配表决定输入变量对应的是哪个寄存器，这期间如果寄存器池满，就有可能会踢出一些变量并将其写到内存。如果这个变量恰好是之前的变量，比如第一个变量，那么t1存的值将不是第一个变量。**

**在生成代码时，最终的目标代码一定会使用获得的寄存器号来生成，而这时t1的值已经不是第一个变量的值了，这条代码就是错误的。**

**这个问题在FIFO（先进先出）等算法中十分明显，假设某个变量恰好处于要被踢出的状态，在一条目标代码的生成过程中，后面的变量就会挤掉这个变量，而此时编译程序仍持有的是这个变量原来对应的寄存器号**

**幸运的是，在认真思考后。我们发现在LRU算法上这个问题恰好不会发生，可以证明如下：**

**首先，一条mips指令最多只涉及到三个寄存器，也就是说分配器只会涉及到三个寄存器的调度。在LRU算法中，访问的变量无论是下面任意一种情况：**

1. **还未分配寄存器，需要申请寄存器**
2. **已经分配寄存器，直接获取对应寄存器**
3. **已经分配寄存器，由于被踢出过值在内存里，需要重新访存并申请**

**在完成一次调度后一定会在队列尾端，则三次调度中，最差情况下分配表如下：**

**第一次调度（a被压入尾端，x0踢出）**

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | **x0** | **x1** | **x2** | **x3** | **…** | **…** | **…** | **…** | **a** |

**第二次调度（b被压入尾端，x1踢出）**

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **x0** | **x1** | **x2** | **x3** | **…** | **…** | **…** | **…** | **a** | **b** |

**第三次调度（c被压入尾端，x2踢出）**

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **x0** | **x1** | **x2** | **x3** | **…** | **…** | **…** | **…** | **a** | **b** | **c** |

**由于队列长度为9（共9个临时寄存器），a、b、c一定不会在三次调度中被踢出。如果一条代码涉及的变量数大于可用寄存器个数（这里3<9，其他架构和体系未必能保证），这种问题仍然会出现。**

**问题2 程序数据流非线性向下时产生的别名问题**

**考虑如下代码中的变量x：**

**SET *$label***

**BECOME x y**

**…(大量中间变量，无x)**

**BECOME x m**

**BECOME z x**

**GOTO *$label***

**假设x分配了临时寄存器$t1，则第一条语句生成类似y=$t1的目标代码。在经过一段代码后，x被踢出。第二条语句时，由于x不在表里，会为x生成一条lw指令，把踢出时写入内存的值加载到新分配的临时寄存器上，比如$t2。目标代码示例如下：**

***$label:***

**move $y, $t1 *#$y表示y对应寄存器，这里不重要***

**…**

**sw $t1 addrx *#x被踢出，产生回写，注意这里之后$t1对应了其他变量***

**…**

**lw $t2 addrx**

**move $m $t2**

**move $t2 $z**

**j *$label***

**在跳转前，x的值被映射到了$t2上，然而跳转后的第一条move指令里，仍然使用的是$t1。**

**问题的核心在于，x在这个基本块内使用先于定义，如果为其分配临时寄存器，使用的临时寄存器并不能保证获得正确的值。要想解决这个问题，出路之一是为其分配全局寄存器，使这个变量对应的寄存器是唯一的。**

**首先需要明确，什么样的变量会产生这类问题呢？不难看出，如果一个变量在基本块内先定义后使用（例如中间变量就一定是这样的变量），那么每次使用一定是在赋过值之后，且二者之间没有数据流的分支和切换，动态寄存器分配策略这种情况下就是正确的。问题只出在哪些在基本块入口处仍然活跃的变量上，这些变量必须有指定的寄存器映射。**

**因此，我们分配全局寄存器的对象为：所有出现在in集合里的变量。这和书上的想法似乎稍有不同，书上认为出现在多个in集合里的变量才需要分配全局寄存器。但结合LRU可以证明，在使用着色图算法的情况下，对所有in集合变量分配全局寄存器不比对跨越多个in集合变量分配寄存器效率更低。理论证明这里不再赘述，感兴趣或者有疑问的老师和助教可以联系我，我将给出具体证明。**

**问题3 通过in集合计算冲突变量时的不一致问题**

**在活跃变量分析一章中，书上认为，两个活跃范围重合的变量冲突。并举了一个冲突的例子，例子中两个变量所在的基本块集合（这里所在基本块指在in集合里）重叠，则冲突。但请看下面的例子：**

**function(int a, int b){**

**int x, y;**

**x = a;**

**y = b;**

**if(…){**

**…**

**}**

**…**

**}**

**例子中a、b是参数，x、y是局部变量。我们只考虑x和b，在程序的后续部分，假设x和b在不同的基本块被使用，则根据定义in=use∪(out-def)，他们一定会出现在不同块的in集合里。考察第一块，由于x在def而不在use，x不会在第一块的in集合，而由于b在use，b会出现在第一块in集合。**

**这也就是说x和b会分配全局寄存器，同时按照书上的定义不会冲突。假设他们被分配到寄存器$s0，在第一句话a的值会被写入$s0，此时$s0不存有b的值，第二句话的赋值自然也就错误了：**

**move $s0 $a**

**move $y $s0**

**这个例子说明，至少在活跃变量分析中，书上对于冲突的描述是错误的。我结合各个教材，还有定义-使用链算法，自己定义了冲突，正确的定义可以参考设计文档。**

### ****目标代码优化****

**我把部分优化放在了目标代码上，这时因为有许多优化需求是在目标代码翻译过程中产生的，和中间代码无关，且目标代码优化涵盖了中间代码可以做的优化。**

**复制传播**

**主要是为了解决在保证少部分情况下的正确性时，牺牲了大部分情况下的效率问题。最典型的例子就是参数传递，设计文档中提到过，**在参数赋值时，如果出现子函数的参数是父函数的传入参数，有可能出现形如

move $a1 xxx

move $a2 $a1

的代码，父函数的第一个参数传给子函数第二个参数，这时在传$a1时$a1被修改，传$a2时将得不到正确的值。解决方法可以是将每个参数都使用中间变量（临时寄存器）缓存，然而这样会有大量冗余代码：

move $t1 xxx

move $t2 $a1

move $a1 $t1

move $a2 $t2

**很多情况下，父函数参数不会直接传给子函数，然而为了正确性我们也不能冒险认为不会遇到这种情况。复制传播可以保证在保证正确性的情况下，尽可能减少无用代码。传播算法如下;**

算法：Copy propagations.

**Input:** 目标代码序列M

**Output:** 无；

1: **For each** M的指令m

2: **If** m是move指令 **and** m的目标寄存器des是临时寄存器**Then**

设m移动的源寄存器为src

3: **For each** m到下一条跳转语句和标签中的指令n

4: **If** n中有对des的使用 **Then**

使用src替换des

5: **If** n中有对des的定义 **Then**

**Break**;

6: **If** n中有对src的定义 **and** 在n后有对des的使用 **Then**

置move语句m为不可删除;

**Break**;

7: **If** m没有被置为不可删除 **Then**

删除m

**算法第6个标号就是为了防止出现类似参数传值时的错误：在move的目标寄存器被使用前，如果move的源寄存器被修改了，说明move指令是有意义的，不可删除。则运行算法后，前文的传参结果如下：**

move $t2 $a1

move $a1 xxx

move $a2 $t2

**可以看出，无意义的赋值语句被删除了，而有意义的赋值语句得到了保留。**

**函数调用优化**

**函数调用时的压栈出栈是大部分访存的来源，如果能够尽可能只压需要压的寄存器，那么传参过程会精简很多。首要的比较简单的想法当然是只压已分配的寄存器。除此之外，我还采用了三大方法删除无用的压栈出栈指令：**

1. **识别目标函数**

**在mips代码中，如果遇到函数调用，我会检测即将跳转的目标函数内有哪些寄存器会被修改，对于不会在目标函数内被修改的寄存器，我将删除对应的lw、sw指令。**

1. **识别当前块内的临时寄存器**

**对于临时寄存器，我会检测在这个基本块结束前，该值是否会被使用，如果该临时变量不会再被使用，也就没有必要为其保存。**

1. **识别特殊性质的全局和参数寄存器**

**这个较为特殊。如果跨块全局变量（不是代码意义全局变量）既没有在这个块被使用，也没有被后续的块使用，则也不压栈。这里没有被后续的块使用，我的判断方法为该基本块的out集合是否有该变量，如果没有，就不进行lw和sw。**



**可以看出，进行优化前访存是优化后的17倍。这从目标代码也可以看出来，很多函数调用基本只压$ra寄存器了。**