**申优文章**

目录

[一、寄存器分配策略的问题与解决 2](#_Toc534181281)

[1.1 寄存器分配策略综述 2](#_Toc534181282)

[1.1.1 分配与被分配的对象 2](#_Toc534181283)

[1.1.2 分配策略 2](#_Toc534181284)

[1.2临时寄存器分配的问题与解决 3](#_Toc534181285)

[1.3 全局寄存器分配的问题与解决 5](#_Toc534181286)

[二、保存现场的问题与解决 5](#_Toc534181287)

[2.1 保存现场策略综述 5](#_Toc534181288)

[2.2 t寄存器的保存现场 6](#_Toc534181289)

[2.3 s寄存器的保存现场 6](#_Toc534181290)

[三、参数传递的问题与解决 6](#_Toc534181291)

[3.1 常规参数传递 7](#_Toc534181292)

[3.2 针对特别函数的优化 7](#_Toc534181293)

[四、其他问题 7](#_Toc534181294)

[4.1 符号表的创新 7](#_Toc534181295)

[4.2 常量传播和复制传播 8](#_Toc534181296)

[4.3公共子表达式的消除 8](#_Toc534181297)

# 一、寄存器分配策略的问题与解决[[1]](#footnote-1)

## 1.1 寄存器分配策略综述

### 1.1.1 分配与被分配的对象

寄存器分配主要的分配对象为s寄存器和t寄存器。其中s寄存器分配给全局变量，这里的全局变量指的是跨越基本块的变量，而t寄存器分配给局部变量，这里的局部变量是指不跨越基本块的变量，包括用户声明的未跨越基本块的变量和中间变量（根据我所抽取到的文法的特点，中间变量一定不会跨块）。

而真正的全局变量，也就是在源代码中函数定义之前所声明的变量，并不分配寄存器，因为按照教材中的有关描述，为了保证线程安全，全局变量不应该分配寄存器，所以每次使用都要访存，虽然这样可能带来效率低下的问题，但是这也是合理的，因为现代程序设计理念中就要求尽可能少用全局变量，违反这些的规范的程序员写出的代码势必不会有很好的效率。

### 1.1.2 分配策略

基于上面的结论，我采用的寄存器分配方式为动态与静态相结合的方法。所谓动态分配，是指寄存器所对应的变量会随着程序的执行而改变，而静态分配则是指在一个函数运行的过程中，变量和寄存器的对应关系是固定不变的。

我采用的动态分配方式用于t寄存器的分配，静态分配方式用于s寄存器的分配。

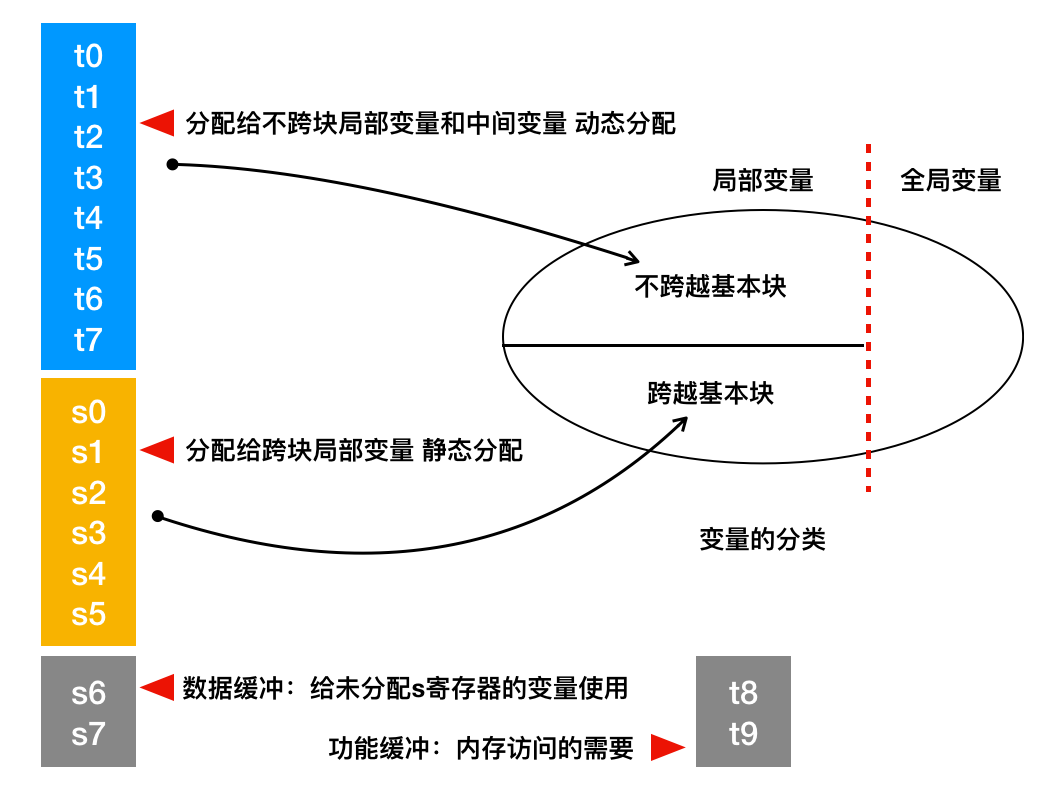


图 1 寄存器分配策略

教材中对于t寄存器的分配介绍并不充分，只是提到了寄存器池的概念，但是并没有给出当寄存器用满时的取舍问题，其核心在于应该尽可能舍弃最不经常使用的寄存器，这就启发我联想到操作系统中的页面分配算法，他们所解决的问题有着一定的相似性。最优算法是可以实现的但是过于复杂，所以我采用了最近最久未使用算法（LRU）进行t寄存器的分配。

我把t0～t7寄存器分配给局部变量，t8～t9留作特殊功能使用（比如向内存中写入常数必须借助寄存器）。这8个寄存器构成一个栈，每次使用都取从栈底取寄存器，每次访问寄存器都把它放到栈的最顶部，这样栈底一定是最近最久未使用的变量，寄存器不够用的时候也会优先淘汰。

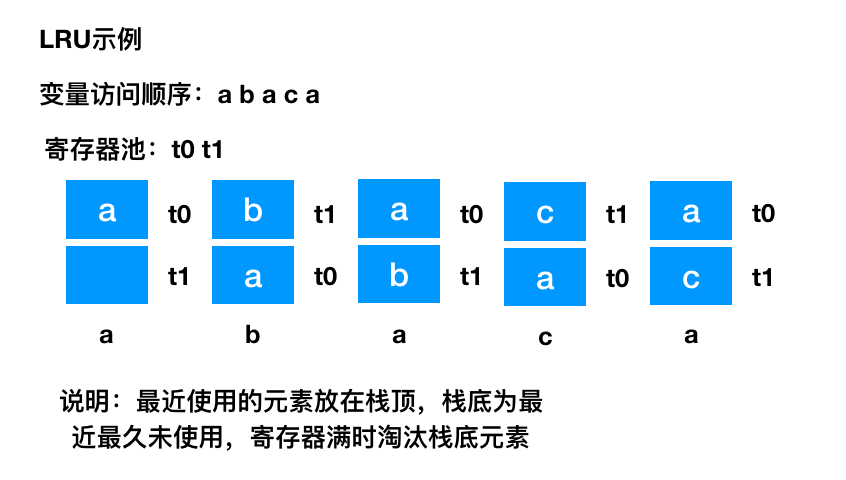


图 2 LRU示例

这种分配方式有很大的好处，和顺序静态分配相比，好处是很明显的，顺序分配并不能保证最频繁使用的变量获得寄存器，而循环则会导致劣势变得非常明显。

## 1.2临时寄存器分配的问题与解决

这样的分配方式会存在问题，考虑下面的情况。

程序运行到下面的中间代码时，因为是一个块的开始，所有的t寄存器都可用，假设在栈中是顺序排列的，那么会按照t7~t0的顺序分配。结合代码解释。

LABEL：

a = expr1 //a分配t7寄存器 因为是第一次使用所以不需要从内存中读取

b = expr1 //b分配t6寄存器 因为是第一次使用所以不需要从内存中读取

c = expr1 //c分配t5寄存器 因为是第一次使用所以不需要从内存中读取

d = expr1 //d分配t4寄存器 因为是第一次使用所以不需要从内存中读取

e = expr1 //e分配t3寄存器 因为是第一次使用所以不需要从内存中读取

f = expr1 //f分配t2寄存器 因为是第一次使用所以不需要从内存中读取

g = expr1 //g分配t1寄存器 因为是第一次使用所以不需要从内存中读取

h = expr1 //h分配t0寄存器 因为是第一次使用所以不需要从内存中读取

i = expr1 //i分配t7寄存器 使用前需要先把t7寄存器之前的值回写到a的内存空间

GOTO LABEL

这个基本块编译结束了，但是问题在于，变量a和i分配了同一个寄存器，而这段代码时循环执行的所以重新回到a = expr1这句话的时候，并没有任何把t7会写到i的内存之后把a加在到寄存器的操作，这会导致变量和寄存器之间的对应关系出现错误。这是寄存器别名问题。出现这个问题的原因是，编译的过程和代码的执行过程并不一致，所以编译时并不能发现可能存在的隐含问题。但是基本块已经保证了代码的执行顺在块内是一致的，为什么依然不能发现呢？原因在于从流图的角度看，这些变量会出现在这个块的in集合中，也就是说，并不能保证在进入这个块的时候此变量的状态的和第一次进入（这也是编译过程知道的状态）时是一样的。因此，严格来说，这类用户定义的变量时跨块变量。

针对上面的问题，我采用的解决方法是构建流图，进行分析。对于出现在in和out集合中的变量，都视为全局变量，为其分配s寄存器，采用冲突图着色（面临的问题会在后面详述）算法进行寄存器分配。in集合与out集合的定义与算法和教材上是一致的，唯一的区别在于in out def use四类集合都会去除掉用户定义的全局变量，因为它们不参与寄存器分配，写在集合中没有意义。

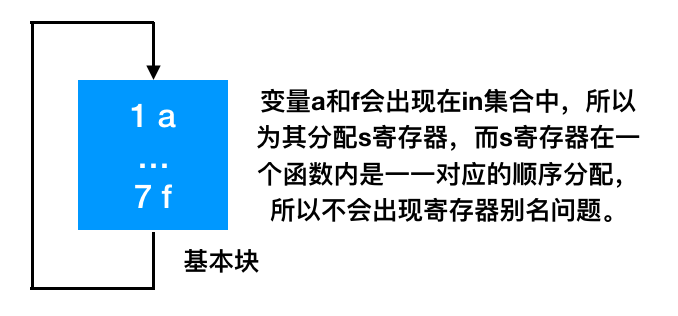


图 3 寄存器别名问题解决

至此，我的寄存器分配策略为：

不跨块的变量（用户定义的不跨块变量和中间变量）分配t寄存器

跨越基本块的变量分配s寄存器

跨越基本块的充分必要条件为：该变量出现在某个基本块的in集合中。

这样，我们就避免了寄存器别名问题，并且也能根据程序访问变量的特点来有效的分配临时寄存器了。

## 1.3 全局寄存器分配的问题与解决

为了让s寄存器能够有效的分配，决定采用冲突图着色的算法。

首先是冲突图的构建。按照教材的说法，两个变量冲突当且仅当它们在同一个基本块的入口处活跃。但是按照这样的说法，也就是可以把一个函数的每个基本块in集合的变量两两冲突，看上去就可以了。

但是考虑下面的情况。

x = a

y = b

显然x会出现在def集合里，所以不会出现在这个基本块的in集合中。但是可能出现在其他的基本块里，所以依然有可能分配s寄存器。而a和b出现在use集合里，所以一定会出现在这个基本块的in集合中分配s寄存器，如果仅仅要求in集合中的变量冲突，a和b是不会和x、y冲突的。这样的话，b可能和x分配了同一个s寄存器，这样就会出现错误。

这里之所以出问题，是因为变量可能出现在多个基本块，但并不总是出现在in集合里，所以，应该将in集合和def集合的元素也冲突，def和def集合的元素也要冲突。这样是合理的，因为如果这些变量出现在了后面的块里，它们还会被继续使用，分配同一个寄存器显然是不合理的。

这样，看上去我们的冲突图变的很强，好像不能起到高效分配寄存器的效果，但是因为不跨块的变量是不分配s寄存器的，所以其实最后我们用来着色的冲突图只是一个子图。

具体的图着色算法如下。

根据前面冲突图建立方式，使用图着色算法，进行寄存器的分配，充分利用寄存器减少访问内存次数。首先是全局（相对基本块而言）寄存器算法如下：

**Input**：冲突图，全局寄存器数量K

**Output**：分配结果

**Algorithm**：

**while** 图中的点超过1个：

**while** 冲突图中存在度小于K的结点：

移走这个点

在图中选取一个适当的结点

标为不分配全局寄存器结点，移走。

对最后一个点任意分配一个寄存器

按照移走的顺序的逆序添加结点和边，保持不同色。

# 二、保存现场的问题与解决

## 2.1 保存现场策略综述

函数调用时需要保存现场，最简单的策略是把31个（0号除外）寄存器全部保存，但是这样每次函数调用会出现大量的访存指令，严重降低效率，所以采用的办法是个性化保存现场。因为寄存器的分配情况在编译的时候是知道的（即使是t寄存器采用动态分配策略，由于基本块中的编译顺序和执行顺序一致，所以依然是知道寄存器的分配情况的），因此可以只把使用的寄存器进行保存。这样寄存器的保存就仅仅局限在了s寄存器和t寄存器了，a寄存器仅仅是函数参数传递的媒介并不需要保存。

## 2.2 t寄存器的保存现场

函数调用并不是一个基本块的划分依据，所以对于t寄存器仍然需要保存，如果能够对中间变量的生存期有着精准的认识，就可以及时的清理掉中间变量。这样一方面缓解t寄存器的压力，一方面也能在保存现场的时候少保存寄存器。

为此主要运用了中间变量的两处特征。

第一，如果赋值语句的左侧是一个用户定义的变量，那么这一定是一个语句的结束，中间变量可以全部去除。

第二，如果遇到一条函数调用指令，那么所有传参数使用的寄存器都可以删除。

如果这样的处理方法，增加了一个隐含的限制条件，也就是在做公共子表达式、复制传播的时候，不能跨越赋值语句和函数调用，否则可能会导致中间变量的生存期超过了预想的长度，不保存现场就会出现错误。

## 2.3 s寄存器的保存现场

s寄存器使用量是比较大的，因为通常一个函数都会有5个左右的跨块变量，他们都要占据s寄存器，为此，保存现场的时候还可以进行一些特判减少保存现场开销。

第一，如果被调用的函数是一个叶子函数，由于它不会再调用其他函数，那么可以很清楚的知道未来会被使用的s寄存器有哪些，这样只需要保存那些父过程和被调用的子过程都是用的寄存器就可以了。

第二，这其实是第一种情况的特殊情况，如果这个函数是一个叶子函数而且不跨块，那么可以直接不保存s寄存器，这其实是一种类似内联展开的做法。

总之，上面的做法都极大降低了保存和恢复现场产生的访存指令。经过这样的优化以后，我的t寄存器几乎不需要保存，而s寄存器保存的数量也会降低。

# 三、参数传递的问题与解决

## 3.1 常规参数传递

如果直接把a固定在参数上，那么会造成的问题是增加了保存现场的负担。因为函数再调用其他就可能出现覆盖。所以我采用的办法是仅仅把a寄存器看作是一种媒介。如果这个参数分配了s寄存器，那么就把参数放到a上，进入到这个函数的时候，会第一时间从a上转移到应该对应的s寄存器上。如果这个参数没有分配s寄存器，那么就把它写到内存中，使用时在load出来。

## 3.2 针对特别函数的优化

上面的这种方式使得我对于那些参数没有分配s寄存器的函数对a的使用不充分。而且对于叶子函数，使用a寄存器之后其实没有必要转移到s寄存器上。因此针对叶子函数，我会直接指定它的前三个参数和a1、a2、a3对应，注意不能把a0分配参数，因为某些函数调用会用到a0，可能造成意料之外的错误。这样，就能把寄存器使用的更加充分了。

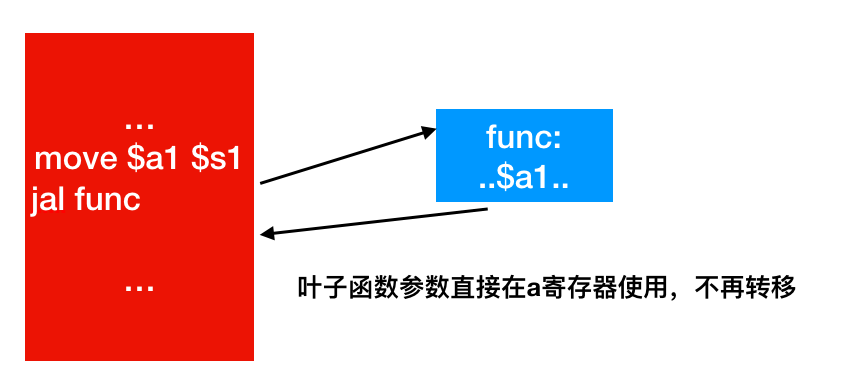


图 4 a寄存器使用充分

上面的这些保存现场和参数传递的优化目的都是降低在发生函数调用时的访存开销，这些优化的效果也是立竿见影的，按照发布的测试样例来看，我的访存数量从一开始的一千五百万量级下降到一百五十万量级，减少了90%。

# 四、其他问题

## 4.1 符号表的创新

符号表的查找是一个很频繁的操作，因此它的查询效率至关重要，栈式符号表是最简单的，所以这是基础，但是，考虑到很多时候并不是查找局部变量而是查找函数，或者全局变量，为了加速这部分查找，我为全局符号建立了一个map索引，因为c++的map时按照Key的字典序排列的，所以查找速度会降到O(logn)。

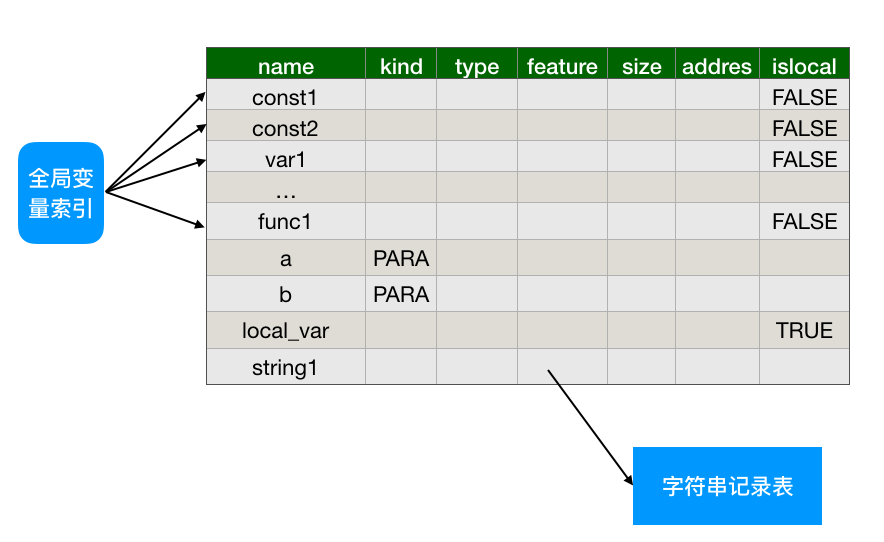


图 5 符号表示意

## 4.2 常量传播和复制传播

常量传播是一个很有效的优化手段，我也进行了传播，敞亮的传播不仅仅局限于替换，更重要的是它可能会让很多代数运算的两个操作数都变成常数，提前进行计算，进一步传播，这样会极大缩小运算量。

复制传播则有可能减少赋值语句，不过要注意，前面已经对中间变量的生存期有了严格的控制，所以如果赋值的来源是一个中间变量那么传播的范围也是有限的。

## 4.3公共子表达式的消除

公共子表达式的消除公认的方法是DAG图，但是这里面会有很多复杂的问题，而且最关键的是这种优化带来的好处其实并不太大，所以我采用了暴力扫描的方法进行解决。

其基本思想是，对于一条运算或者取内存元素的指令，我会检查后面是否有操作数和它一样并且没有被改变的指令，如果有，那么将其变成一条赋值语句，之后在使用复制传播进行优化。这样也会有不错的效果，实现起来也还比较简单。

1. 本部分的设计与实现和郑明悟（16231019）同学讨论完成。 [↑](#footnote-ref-1)