# 编译课设申优文章

16231092魏淑越

目录

[编译课设申优文章 1](#_Toc534294986)

[1优化思想概述 2](#_Toc534294987)

[2初始架构 2](#_Toc534294988)

[2.1中间变量的分配 2](#_Toc534294989)

[2.2寄存器的分配 3](#_Toc534294990)

[2.3中间变量不跨块约定 4](#_Toc534294991)

[3常量传播、合并和复制传播 4](#_Toc534294992)

[3.1基本块的划分 4](#_Toc534294993)

[3.2常量、复制传播 5](#_Toc534294994)

[3.3常量合并 6](#_Toc534294995)

[4内联函数 6](#_Toc534294996)

[4.1函数内联的条件 7](#_Toc534294997)

[4.2内联函数的处理 7](#_Toc534294998)

[5寄存器分配 9](#_Toc534294999)

[5.1局部变量寄存器分配$s和$a 9](#_Toc534295000)

[5.2临时寄存器的分配$t 9](#_Toc534295001)

[5.3现场的个性化保存 10](#_Toc534295002)

[6窥孔优化 10](#_Toc534295003)

[7优化总结 10](#_Toc534295004)

## 1优化思想概述

对于编译的优化，总的指导思想是(1)降低访存,将更多的内容分配到寄存器中(2)将代码中某些重复的量重复使用，减少运算。

针对降低访存，首先想到的是进行局部寄存器的分配和全局寄存器的分配，这样能够将中间变量和局部变量存储在寄存器中随时使用，降低了访存。同时观察到有很多功能性小函数，如计算两点间距离，计算平方和计算余数等函数，针对这些函数如果使用内联，可以省去很多保存现场所需要的时间。

针对重复内容的使用，则可以进行常量传播，对于定义的常量标识符和普通的常数，直接进行替换，可以在编译时计算出的值直接计算;对于中间变量,如果其是相同的赋值或者相同的常数，则可以直接用原来的常量。最后删除没有使用的死代码，可以减少一些运算量。

## 2初始架构

在最开始的设计编译时，为了保证工程规范性和准确性。产生了很多无用的操作，但是对于保证正确性却有非常大的好处，下面将说明为保证正确性和规范性而做的“无用功”。

### 2.1中间变量的分配

1. 对于所有变量的引用都会先分配一个中间变量保存其值，如使用x时#1 = x。
2. 对于所有常数的使用都会先分配一个中间变量保存其值，如使用5时#2 = 5。
3. 对于所有变量的赋值都会先分配一个中间变量保存其值，如 x = #2

这种架构屏蔽了立即数和翻译的目标代码，使得所有的操作只需要处理分配中间变量和对中间变量的操作即可,如 x = a + b，中间代码会翻译成：

|  |  |
| --- | --- |
| x = a + b | #1 = a  #2 = b  #3 = #1 + #2  x = #3 |

所以下一步进行优化的时将会缩减这些无用变量。

### 2.2寄存器的分配

在最初的非优化版本中，定位寄存器的作用是能够缓存内存中的值并且参与计算，但是寄存器的值对于一条中间代码有效，使用结束后必须立即回写内存。这样所有数据都存储在内存中，没有数据一致性的问题，比较容易维持正确性，而在以后寄存器中会保持一段时间的内存中变量的值，所以内存和寄存器数据一致性将会是一个重要问题。

下面说明对于所有中间代码如果将其翻译成目标代码，最少使用两个寄存器，假定我们使用的这两个寄存器为$t8, $t9.

四则运算:

|  |  |
| --- | --- |
| 中间代码  #3 = #2 + #1  (对于mips 指令,mul 伪指令可以和addu同样格式 mul $t8 $t8 $t9) | 目标代码:  lw $t8 #1 (暂时使用#3表示变量内存地址)  lw $t9 #2  addu $t8 $t8 $t9  sw $t8 #3 |

数组取值运算:

|  |  |
| --- | --- |
| 中间代码  #1 = A[#2] | 目标代码:  lw $t9 #2  sll $t9 $t9 2  lw $t8 A($t9)  sw $t8 #1 |

数组赋值运算：

|  |  |
| --- | --- |
| 中间代码  A[#2] = #1 | 目标代码:  lw $t8 #1  lw $t9 #2  sll $t9 $t9 2  sw $t8 A($t9) |

其他中间代码的翻译均不超过以上中间代码

所以在之后的优化中，采用保留$8 $t9为临时寄存器缓冲，当寄存器数目不够用的时候，采用$t8 $t9 以及访存指令进行运算。

### 2.3中间变量不跨块约定

对于中间变量，在理论课学习的时候我们知道其是不跨块的，因为中间变量的产生是缓存某些句子中的值，所以对于中间变量给其分配寄存器的时候，一旦这个中间变量没有在当前块内使用，那么这个中间变量的生命周期就结束了，可以将所占寄存器及时释放出来。但是由于C0文法特点，导致了switch(#1=<expr>)中#1变量是跨越块使用的，所以将该变量转为@1-@10(switch-case最多嵌套10层)的局部变量，填入到函数的符号表中，其使用时和函数内其他定义变量一样，至此所有中间变量均不会跨越基本块。

## 3常量传播、合并和复制传播

在进行常量传播和赋值传播之前，第一个问题是，传播的对象是谁以及传播到什么地方结束。

对于中间变量为了寄存器分配的高效，其生命周期不会超过一个基本块，所以中间变量的传播不会跨越基本块。对于常量类似，常量本身是先赋值给中间变量在使用，所以会将本块内所有该值尽可能替换为常量。对于局部变量，在之前的架构中同一局部变量的使用会多次分配中间变量，现在可以使用第一次分配的中间变量来代替。对于全局变量，由于在调用函数的时候全局变量的值可能会修改，这就意味着下一次使用全局变量时，必须重新从内存中获取数值，不能使用之前中间变量的值，所以在此全局变量我们不进行复制传播。

### 3.1基本块的划分

因为常量传播和复制传播的最大区域都是在某一基本块内，所以首先要进行基本块的划分。基本块按照课本理论的说明:

1. 函数的第一条语句属于基本块的入口
2. 能够跳转到的语句属于基本块入口
3. 跳转语句下一条语句输入基本块入口

(4)所有基本块的入口和入口之间形成左闭右开的区间是一个基本块：

基本块入口a,

基本块和流图不同，流图需要分析各个基本块之间的关系，而基本块只需要划出分界线即可。

以下是基本块划分的语句:

1. label 标签是跳转语句跳转到的，所以一个标签意味者新的基本块开始
2. return, goto, branch等跳转指令意味着一个基本块的结束，下一条语句是下一个块

在进入每个基本块之前都要初始化传播关系，这样才能够保证结果的正确。

### 3.2常量、复制传播

在常量传播时，按照之前架构的方法第一次一定会将常量赋值给某一中间变量，然后对中间变量进行使用。所以，只需要构建一个对应表数据结构，里面是每一个中间变量和其对应的常量或变量。每次使用某一中间变量时，首先查看其表中是否有对应的常数，如果有则进一步考察是否可以进行替换。

|  |  |  |  |
| --- | --- | --- | --- |
| 原中间代码：  #1 = 5  #2 = 3  #3 =#1 + #2 | 常量传播：  #1 = 5  #2 = 3  #3 =5 + 3 | 死代码删除：  #3 =5 + 3 | 常数合并：  # 3 = 8 |

表格 1常量传播示意

对于中间变量，局部变量其传播周期与常量稍有不同，因为变量有可能被赋值，在变量赋值之后，那么之前可以替代它的中间变量都不能继续替代他，所以需要在对应表中抹除对应关系，然后再增加新的对应关系。

|  |  |  |
| --- | --- | --- |
| x = #1 | | 对应x的中间变量修改对应为NULL  增加#1对应的量为x |
| # 2 = #1 | | 中间变量操作同上 |
| 中间代码：  x = #1  # 3 = x  #2 = #1  #4 = #3 + #2 | 变量传播：  x = #1  #3 = #1  #2 = #1  #4 = #1 + #1 | 删除死代码：  x = #1  #4 =#1 + #1 |

表格 2变量传播示意

### 3.3常量合并

经过常量传播之后某些值在编译阶段是可以计算出的，如表格1中的 #3 = 5 + 3

可以直接计算出 #3 = 5。所以下面针对一些常量替换后的结果进行下一步的优化。

·四则运算

**对于加减乘除四则运算在常数传播后可以执行以下优化：**

1. **如果两个操作数都是常数则可以计算结果并改为赋值语句**
2. **如果两个操作数第二个是常数则可以对应addi,subi等立即数指令**
3. **如果乘法和加法第一个操作数为常数则交换两个操作数使用(2)**

|  |  |  |
| --- | --- | --- |
| **(1)**  **#1 = 3 + 5 (op = add\_op)**  **修改为：**  **#1 = 5 (op == assign\_op)** | **(2)**  **#1 = #4 - 2 (op=sub\_op)**  **修改为:**  **#1 = # 4-2 (op =subi\_op)** | **(3)**  **#1 = 2+#3(op =add\_op)**  **修改为:**  **#1 = #3 +2 (op = addi\_op)** |

表格 3常量合并示意图

·跳转逻辑判断

**对于branch类中间代码逻辑判断可以直接计算出是跳转或是不跳转**

1. **如果两个操作数都是常数,可以直接逻辑判断改为goto或删除代码**
2. **如果两个操作数有一个是常数那么可以修改为branch-#-number类型(在mips指令中beq $t1 3 label是一条伪指令，其会翻译成两条指令，但是尽可能地利用$1这个mars自己使用的寄存器，会腾出空间给$t——让更多寄存器处于使用状态总是更优的)**

|  |  |  |  |
| --- | --- | --- | --- |
| **(1)**  **beq 1 3 label**  **修改为**  **NULL(删除该指令)** | **(1)**  **beq 1 1 label**  **修改为**  **goto label** | **(2)**  **beq 1 #1 label**  **修改为**  **beq #1 1 label** | **(2)**  **bgte 1 #1 label**  **修改为**  **blte #1 1 label** |

表格 4跳转指令常量合并

## 4内联函数

在上文中已经说明对于一些小函数如计算距离、计算平方根等来回的调用需要花费大量的时间用于保存现场，内联展开可以避免这些无用的保存。

### 4.1函数内联的条件

首先对被内联的函数提出一些要求:

1. 内联函数中不能够定义变量
2. 内联函数中只能有一个基本块
3. 内联函数参数不超过4个
4. 内联函数不能调用其他函数

下面说明这些要求的必要性。(1)如果内联函数中自己定义了局部变量，那么就意味着要么这个中间变量不能够分配寄存器，要么调用者的寄存器需要保存，两者抢占寄存器导致难以平衡，极有可能出现负优化的情况，所以规定，定义变量的函数不能进行内联展开。(2)内联函数中如果出现多个块最直接的问题是，原本的一个块会被切分成多个部分，这样可能会导致原函数块过于琐碎出现负优化，同时分支语句难以直接判断函数从何处返回，需要增加新的标签和跳转指令，过于繁琐，综上两点规定内联函数中只能够有一个基本块。(3)内联函数中的参数不能超过四个，由(3)可知如果内联的函数过于庞大，会导致抢占$t寄存器，形成额外的访存，所以为了限制内联函数块的大小，规定内联函数的参数个数不多于4个。(4)如果内联函数调用其他函数，那么将会面临如递归函数是否要自循环展开，展开多少层，并且需要分析函数相互调用关系，过于繁琐，所以只对不调用其他函数的叶子节点进行展开。

### 4.2内联函数的处理

**·参数传递**

对于内联参数传递不再需要通过压栈来解决，而是通过中间变量的赋值即可，遂于所有的参数,全部使用原本计算出待压栈的中间变量直接替换。这样不但节省了保存现场的时间，而且参数可以直接参与常量传播和复制传播的优化。

|  |  |
| --- | --- |
| 内联函数  int add(int x,int y){  return(x+y)  }  中间代码  Push #3  Push #4  call add()  #5 = #ret | 内联函数  #1 = x  #2 = y  #3 =#1 + #2  ret #3  内联后代码:  Inline:  #3 = #3 + #4  #inline\_ret = #3 + #4  #5 = #inline\_ret |

表格 5内联函数参数传递

**·中间变量**

**对于内联函数中出现的中间变量，需要将其统一归为调用者的中间变量，所以这个时候会出现内联函数的中间变量与调用者中间变量重名(如表格5中内联后 #3重名)的现象。为了解决这个问题，在每个函数中都为其专门开辟一部分空间供内联函数的中间变量使用，并且在符号表中增加一个指针，表示当前内联的中间变量已经用到多少，比如当前指针为80,那么内联函数中的某个中间变量#5内联之后就会成为#85,这样在调用者函数中所有的中间变量都不会重名。**

|  |  |
| --- | --- |
| **内联函数**  **int add(int x,int y){**  **return(x+y)**  **}**  **中间代码**  **Push #3**  **Push #4**  **call add()**  **#5 = #ret** | **内联函数**  **#1 = x**  **#2 = y**  **#3 =#1 + #2**  **ret #3**  **内联后代码:**  **Inline:**  **#81 = #3**  **#82 = #4**  **#83 = #81 + #82 (inline\_tpr =80)**  **#inline\_ret = #83**  **#5 = #inline\_ret** |

表格 6解决内联函数中间变量重名

**内联函数的优化效果在一些反复调用的小函数中是惊人的，如给出的竞速排名代码**

**在使用内联函数之后访存不到100万，整个的finalCycle从4000,0000级别降至了2000，000级别，效果非常明显。**

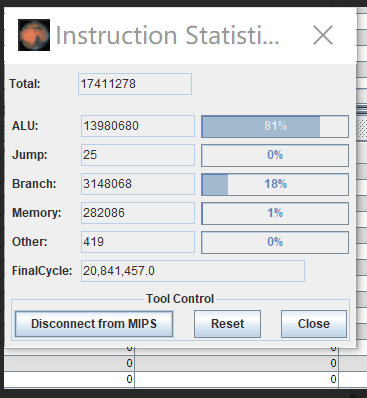


图 1内联优化后效果

## 5寄存器分配

### 5.1局部变量寄存器分配$s和$a

对于局部变量寄存器分配采取了简单的计数分配——优先分配循环中使用的局部变量，然后其他局部变量按照使用次数进行分配。由于一个函数通常不超过300行所以8个$s基本是足够使用的。对于参数，对前三个参数依次分配$a1,$a2,$a3($a0留给系统调用)。对于全局变量不分配临时寄存器。因为全局变量可能会在调用函数的过程中修改，这样就导致寄存器和内存中的数据不一致，引发错误。

对于参数其use 先于def 所以必须在进入函数伊始就从内存中lw到寄存器中值，然后再整个函数中全权代理该参数。对于局部变量，其第一次使用一定是对其赋值所以不需要从内存中加载该寄存器的值。同样，如果某个局部变量绑定了某个寄存器，那么到函数结束之前该变量的所有操作都直接使用寄存器与内存无关，其他未分配寄存器的少数变量则会通过缓冲寄存器直接访存写存将数据存储到内存中。

### ****5.2临时寄存器的分配$t****

对于中间变量，在上文中已经提到，其作用域不会超过一个基本块。所以很容易控制其生命周期。在使用一个中间变量时，如果当前寄存器池有空位，那么将某个寄存器分配给该变量。寄存器池有空位意味者，寄存器池有一个寄存器未被中间变量占有或者，占有该中间变量的寄存器在当前基本块中不再使用。如果寄存器池全部占满，则用保留的$8 $9完成数据缓冲，并且进行访存写存工作。同样的，对于中间变量的def 型语句(#3 = 1)则分配的寄存器不需要从内存中加载值，如果是use型语句(printf(#3))，则首先查看#3是否对应寄存器，如果没有对应，则需要先从内存中加载数据，然后使用。

### 5.3现场的个性化保存

在调用函数的时候，需要保存现场，但是如果所有寄存器全部保存，将会是很大的一笔开销。所以对于$ra $fp该类必须保存的寄存器予以保存。对于$t首先检查其对应的中间变量是否仍将使用，如果不在基本块中被使用，那么不必保存和加载该寄存器。这样在函数调用的时候能够节省大量访存指令。

## 6窥孔优化

窥孔优化只是进行了一些小细节的优化，但是有些效果依然十分明显。

· 跳转指令跳转到的地址恰好是下一条。则可以不必跳转。

这个在if-else语句中非常常见，因为如果else 后面跟的是空语句，那么直接运行就可以到else后面的部分不必进行跳转。

·翻译目标代码时div 指令不要采用三元组

div $t1 $t2 $t3 会产生判断除0的分支判断条件，每次都会判断，所以会增加许多跳转指令。而如果采用div $t2 $t3 mflo $t1,两条指令则不会进行分支判断，节省时间。

## 7优化总结

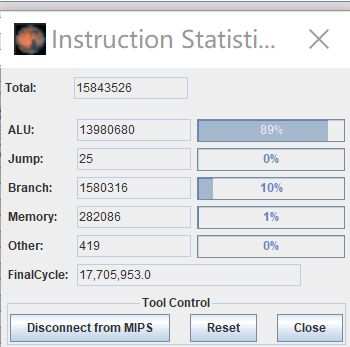


图 2最终优化效果

本着降低访存和利用重复内容的思想，进行了一系列的优化，在这个过程中能够感受到优化时有些臃肿复杂的方法有时效果却不是最好，有些不起眼的小细节却能优化很多，而且随着优化的不断进行，为了提升一小点通常要花费越来越多的精力。最终针对给出的优化测试样例，finalCycle优化到了1700,0000,memory 更是优化到了不到1%，优化效果非常明显。