编译优化文章

黄雨石

20376156

以下基本按照我所做优化的顺序来进行讲述，同时由于遇到的难点以及困难大部分其实并不是在优化本身而是在于要修改之前的架构，`de`由于变动之前代码而产生的`Bug`，因此只有部分优化会介绍难点与解决方案

同时优化架构的初步设计是对每个优化尽量做成单独的模块，并为其设置易于管理的优化开关，并且要保持对每一个优化版本的兼容性，优先保证编译程序的正确性。初步优化思路是先对程序进行数据流析，然后基于数据流分析进行优化，优化主要分为中间代码层面和目标代码层面，以及穿插在各个部分的小 tricks.

当然本人除介绍优化外其实还做了一些没有什么用的优化，比如将`main`中的数组变量放在全局段，但是其实根本没有效果，这种比较野且无用的优化在此不介绍了

运算强度削弱

乘除

- 首先对于乘法，可以将赋值操作转换为移位操作，例如将`mul $s1, $s2, num`可以翻译为：

```
sll $s1, $s2, x0
sll $a1, $s2, x1
addu $s1, $a1, $s1
sll $a1, $s2, x3
addu $s1, $a1, $s1
# ...
nge $s1, $s1 # if num < 0
```

注意$|num| = 2^{x_i} + 2^{x_{i-1}} + \ldots + 2^{x_0}$，其中$x_i > x_{i-1} > \ldots > x_0 \geq 0$，该拆分可以提前计算出来，同时需要特判$num = 0$的情况，我们可以计算出这种转换所用的指令的`cycle`数，如果`cycle`数大于等于4，那么应该直接翻译为一条乘法指令，不然就是负优化

- 对于除法，可以参考下面的算法：

procedure CHOOSE_MULTIPLIER(**uword** d, **int** $prec$);
Cmt. d – Constant divisor to invert. $1 \leq d < 2^N$.
Cmt. $prec$ – Number of bits of precision needed, $1 \leq prec \leq N$.
Cmt. Finds m, sh_{post}, ℓ such that:
Cmt. $2^{\ell-1} < d \leq 2^{\ell}$.
Cmt. $0 \leq sh_{\text{post}} \leq \ell$. If $sh_{\text{post}} > 0$, then $N + sh_{\text{post}} \leq \ell + prec$.
Cmt. $2^{N+sh_{\text{post}}} < m * d \leq 2^{N+sh_{\text{post}}} * (1 + 2^{-prec})$.
Cmt. Corollary. If $d \leq 2^{prec}$, then $m < 2^{N+sh_{\text{post}}} * (1 + 2^{1-\ell})/d \leq 2^{N+sh_{\text{post}}-\ell+1}$.
Cmt. Hence m fits in $\max(prec, N-\ell)+1$ bits (unsigned).
Cmt.
int $\ell = \lceil \log_2 d \rceil$, $sh_{\text{post}} = \ell$;
udword $m_{\text{low}} = \lfloor 2^{N+\ell}/d \rfloor$, $m_{\text{high}} = \lfloor (2^{N+\ell} + 2^{N+\ell-prec})/d \rfloor$;
Cmt. To avoid numerator overflow, compute m_{low} as $2^N + (m_{\text{low}} - 2^N)$.
Cmt. Likewise for m_{high}. Compare m' in Figure 4.1.
Invariant. $m_{\text{low}} = \lfloor 2^{N+sh_{\text{post}}}/d \rfloor < m_{\text{high}} = \lfloor 2^{N+sh_{\text{post}}} * (1 + 2^{-prec})/d \rfloor$.
while $\lfloor m_{\text{low}}/2 \rfloor < \lfloor m_{\text{high}}/2 \rfloor$ **and** $sh_{\text{post}} > 0$ **do**
    $m_{\text{low}} = \lfloor m_{\text{low}}/2 \rfloor$; $m_{\text{high}} = \lfloor m_{\text{high}}/2 \rfloor$; $sh_{\text{post}} = sh_{\text{post}} - 1$;
end while; /* Reduce to lowest terms. */
return (m_{high}, sh_{post}, ℓ); /* Three outputs. */
end CHOOSE_MULTIPLIER;

Inputs: **sword** d and n, with d constant and $d \neq 0$.
udword m;
int ℓ, sh_{post};
$(m, sh_{\text{post}}, \ell) = \text{CHOOSE\_MULTIPLIER}(|d|, N-1)$;
if $|d| = 1$ **then**
  Issue $q = d$;
else if $|d| = 2^{\ell}$ **then**
  Issue $q = \text{SRA}(n + \text{SRL}(\text{SRA}(n, \ell-1), N-\ell), \ell)$;
else if $m < 2^{N-1}$ **then**
  Issue $q = \text{SRA}(\text{MULSH}(m, n), sh_{\text{post}}) - \text{XSIGN}(n)$;
else
  Issue $q = \text{SRA}(n + \text{MULSH}(m - 2^N, n), sh_{\text{post}})$
    $- \text{XSIGN}(n)$;
  Cmt. Caution — $m - 2^N$ is negative.
end if

if $d < 0$ **then**
  Issue $q = -q$;
end if

当然如果对于立即数属于$\{1, -1\}$的情况特判会更快，上述算法的主要思想是将$\frac{n}{d}$转化为$\frac{n \times m}{2^{N+l}}$，即乘法加移位运算，当然还需要考虑$m$的溢出、$n$的正负性的问题

在进行该优化的时候，如果是运用`java`的同学需要注意了，对于算法中的`CHOOSE_MULTIPLIER`：

```java
private static long[] chooseMultiplier(int d, int p) {
    long l = (long) Math.ceil((Math.log(d) / Math.log(2)));
    long sh = l;
    long low = (long) Math.floor(Math.pow(2, N + l) / d);
    long high = (long) Math.floor((Math.pow(2, N + l) + Math.pow(2, N + l - p)) / d);
    while ((low >> 1) < (high >> 1) && sh > 0) {
        low >>= 1;
        high >>= 1;
        sh -= 1;
    }
    return new long[]{high, sh, l};
}
```

对于设置`low`和`long`的初值，譬如`long low = (long) Math.floor(Math.pow(2, N + l) / d);`如果直接进行移位运算，即写成`long low = (1 << (N + l)) / d`，会导致运算溢出，这个问题十分隐蔽

地址计算

对于访存一个数组的元素，比如`a[k]`，我们可以用移位运算代替乘法去计算偏移量

常量合并

- 对于类型为`const`或者位于全局的变量的初始值都可以在中间代码生成的过程中计算出来，于是可以写一个计算类，通过`Java`的异常机制，计算不出来那么就生成正常的中间代码
- 对于`stmt`中的一些计算语句，或者比较语句我们也可以将其算出来，但是如果遇到左值，可以用初值代替其中为`const`的变量，而非`const`变量则不行，防止变量除了初始化之外还在其它地方被赋值

对于如下代码：

```
const int a[10] = {1,2,3,4,5,6,7,8,9,10};

int main() {
    int b = a[0] + 5;
    printf("%d", a[7] + 10);
    printf("%d", b);
    return 0;
}
```

可以翻译成如下中间代码：

```
// ...
======FUNC: main(0,1)======
DEF_VAL 6 b(1,1)
PRINT_INT 18
PRINT_INT b(1,1)
RETURN 0
======FUNC_END: main(0,1)======
```

`do-while`

对于`Sys`语言中的`while(cond) {...}`可以翻译成`if (cond) {do{...} while (cond)}`，即如下：

```
# while
begin:
beqz cond end
...
j begin
end:

# do - while
beqz cond end
begin:
...
bnez cond begin
end:
```

对于一开始不满足情况那么二者均为跳转一次，但是如果多次进入循环体的内部，易得上面的`while`会执行$2n-1$次跳转，而下面的`do-while`会执行n次跳转，且不会有其他副作用

但是注意的是由于要翻译两次`Cond`，一次是满足要跳转`begin`，一次是不满足要跳转到`end`，这里有个问题就是如果你是解析两次`Cond`，那么如果遇到错误，你会报两次错，所以建议用一个行号为键值的`HashMap`来存储错误，这样就不会报两次同一行的错误了

数据流分析

划分基本块与流图的建立

正如在`mips`代码生成那里所说的一样，没有划分基本块，函数执行顺序信息是得不到的，只能通过一些及其不优雅的方法，例如生成空跳转来解决进入一个基本快寄存器状态不一致的问题，所以要先划分基本块并建立流图，注意这是函数内部划分，具体方法如下：

- 跳转型中间代码`JUMP`、`EQZ_JUMP`、`NEZ_JUMP`、`RETURN`的下一句作为入口点，同时除`RETURN`外，其它跳转要跳到的`LABEL`也作为入口点
- 所有入口点的前一句作为出口点，将所有代码按照入口点与出口点从小到大逐一匹配就划分完成基本块
- 将末尾为无条件跳转的基本块与其跳转到的基本块相连，注意`RETURN`算无条件跳转，且直接跳转到B_{exit}；其余则将自己与自己按代码顺序的下一个基本块相连，同时如果最后一条是条件跳转，那么也将它与要跳转的的基本块相连

注意在此过程进行前其实可以删除一些不必要的跳转，比如刚好要跳到的就是下一句话，或者是无条件跳转下方接一条跳转；还可以优化部分跳转，比如跳转到的`LABEL`下方恰好是一条无条件跳转，那么可以将自己跳转的`LABEL`直接变为找到的无条件跳转的`LABEL`，注意这个过程可以用递归循环找，但是要特判跳转成环的情况

在刚化完流图时，我们其实可以从所有基本块的入口开始`dfs`，遍历流图，然后将不会经过的基本块删除，这样在后续的到达定义分析，活跃变量分析得到的结果会更加准确

同时由于划分了基本块，我们可以改掉之前为使寄存器保持状态一直而生成空跳，并且在后续窥空也无法删掉它们的做法，新的做法如下：

- 在进入一个基本块前清空寄存器映射关系
- 在出基本块时，且最后一条语句不是跳转或者`RETURN`时执行完它立刻回写，如果是跳转则在跳转前回写，`RETURN`不需要回写

到达定义分析

注意对于数组由于较为复杂因此是不参与的，按照书本的定义$out[B]=in[B]\cup(gen[B]-kill[B])$，$in[B]=\cup_{B的前驱}out[B_{pre}]$从$B_{exit}$开始反复迭代即可，但是要注意:

- 对于一个基本块自身而言它的$kill[B]=kill[d_1]\cup kill[d_2]\cup...\cup kill[d_n]$，对于每一个$kill[d_i]$，它等于考虑的范围是这个函数的所有基本块中除了它自己之外所有中间代码
- 对于$gen[B]=gen[d_n]\cup(gen[d_{n-1}]-kill[d_n])\cup...\cup(gen[d_1]-kill[d_2]-...-kill[d_n])$，即我们可以从后往前倒着考虑每一条中间代码，如果发现该条中间代码所生成的信息已经在$gen[B]$里面了就不用添加了

活跃变量分析

基本上与到达定义分析相同，但是是从后往前分析，同时还有注意：

- 在计算$use[B]$的时候，数组元素不参与考虑，但是数组的下标如果是一个普通变量的话要算进来，同时函数调用的返回值也不必考虑
- $def[B]$和$use[B]$需要先统计每一个变量在基本块中第一次被定义和使用的时候，如果第一次使用的位置小于等于第一次被定义的位置，那么加入$use[B]$，否则加入$def[B]$

在这里需要说明的是活跃变量分析我们分析的是变量，不需要记录代码行号以及所在代码，而到达定义分析分析的是一条一条的中间代码，我由于一开始并没有注意到这个区别，全部都是按照变量所在代码来分析的，于是产生的活跃变量分析结果十分庞大，且不够精准

死代码删除

该优化以活跃变量分析的结果为基础：

- 首先维护一个队列q将基本块的out_{active}中的所有变量加入其中，然后倒着遍历该基本块的所有代码
- 对于当前遍历到的代码，如果它的def不为空，且为于q中或者它的def为空（def为空一定为跳转或者`printf`这种会产生副作用的代码），那么将它的use加入队列，同时将该def移除出队列；如果它的def不为空且不在队列中，那么则可以删掉改代码

注意对于含有**对全局变量赋值意义的代码，我们不能删除**，同时也要按照def不为空时将它的use加入队列处理；对于`getint`，即使他的def不为空且不在队列中，我们也不可以删掉，只能将它的结果由一个变量改为`(EMPTY)`，这样也省略了一条赋值语句

由于删除了很多死代码，于是也产生了很多地方可以窥孔，以至于会导致划分的基本块不再准确于是又可以重新划分基本块再优化一遍，直到优化结果不变为止

常量传播与复写传播

策略是常量可以跨基本块传播，变量只能块内传播，因为跨基本块前一定会回写内存，解除寄存器映射，如果跨基本块传播变量，那么极有可能需要读取内存，因此很有可能导致负优化

需要注意全局变量不能传播与被传播，因为全局变量的值可能在中间某次函数调用的时候发生改变，而我们的数据流分析并不会考虑函数调用因此需保守处理，同时数组同理也不处理，具体算法如下：

- 首先维护一个产生了def信息的代码的队列q，该队列中代码之间的顺序与它们在基本块中的顺序一致；
- 对于当前遍历到的代码，取出它的use信息，然后倒着遍历q，一旦发现q中有def的变量为use中的变量，那么停止遍历，
    - 同时检测该def的代码是否是常量赋值或者将变量定义为常量的代码，如果是那么传播常量；
    - 如果为变量赋值或者将变量定义为变量，那么接着从q的当前位置正向遍历，检查该变量赋值或者变量定义的代码右侧所使用的变量是否在之后被重新定义，如果没有，那么将改变量复写传播给use
- 如果倒着遍历完了整个队列q都没有可以进行的传播，且use变量没有出现在q中代码所产生的def信息中，那么可以检测该基本块的in_{arrive}中的变量，如果use信息中的变量只在其中出现过一次且满足上述常量传播的条件，那么可以进行跨基本块的常量传播

同样做了此优化会有很多窥空的空间，以至于会导致划分的基本块不再准确于是又可以重新划分基本块再优化一遍，直到优化结果不变为止

这个优化需要注意不要将同一个变量传播到它所在的下一个位置，否则会检测到出现传播，于是会不断重新划分基本块进行优化，程序无法停止运行

寄存器分配

图着色分配

对于局部变量和临时变量采取图着色分配策略

- 首先先计算每一条中间代码的out_{active}，然后通过分析该中间代码的操作数、结果、out_{active}之间是否有冲突关系建立冲突图

- 按照书上此步操作应该导出染色节点的序列，但是这步书上并未给出一个较好的算法，这里推荐使用MCS算法，求出该图的完美消除序列，对于实现了`SSA`的同学，这个算法是一个最优解（`SSA`代码导出的冲突图是一个弦图），但是即使没有实现`SSA`，该算法也十分有效，具体可以参考https://oi-wiki.org/graph/chord/
- 当求出消除序列就可以按照理论课上的算法对原图进行染色了，对所有节点染完色后，由于先前求出的序列已经是一个"完美消除序列"，因此在此直接对每种颜色所覆盖的节点数量进行排序，先为覆盖节点数量最多的颜色建立一个寄存器映射、接着为覆盖节点第二多的节点……

按图着色分配寄存器最大的好处不仅仅是寄存器利用更加充分了，而且每次跨越基本块的时候不需要回写变量了，注意如果一开始没有按照图着色进行分配，这里很可能要`de`很久`Bug`，因为之前所有的寄存器分配都是一跨基本块就会清空，现在却不用了，图着色分配到寄存器的变量只有两个地方需要读写内存：

- 首先是函数调用，刚进入被调用函数的时候，需要将参数中分配到图着色寄存器的变量全部加载到寄存器中，因为图着色算法无法知道寄存器中是否已经加载了变量，而函数调用相当于对参数的一次赋值
- 进入被调用函数前需要将所有分配到图着色寄存器同时在接下来接着活跃的变量存到内存中，函数调用结束就恢复，首先一定要恢复，因为用图着色法分配寄存器无法得知当前变量是否真的已经加载到在寄存器中（没有办法用到的时候从内存中加载，默认已经存在），只可以得知当前寄存器中应该是什么变量，同时对于在将变量保存到内存的时候，如果当前寄存器在整个函数中是被分配给一个实参和一个普通变量，且当前寄存器建立的映射是普通变量的，那么需要将映射变为实参，然后保存，比如以下情况：

```
void fun() {
    // ...
}

int add(int a) {
    int i = 1;
    if (i < 2) {
        a = a + 1;
    } else {
        int b;
        b = b + 1;
        printf("%d\n", b);
    }
    fun();
    return a;
}
```

`a`、`b`并不冲突，印次`a`、`b`可以分配到同一个寄存器中，按照顺序翻译那么最后运行到`fun()`压栈时该寄存器中保留的映射会是`b`，故会压`b`到栈中，而`a`就被丢掉了，故此时需要将映射变为实参，然后保存，注意只有参数比较特殊，并且对于参数需要将一个函数的所有参数全部建立冲突，不可以分配同一个寄存器，因为进入函数后需要将所有分配了寄存器的参数加载到寄存器中

这里其实有一个优化可以做，但是时间原因来不及了，因为将所有参数放在栈上传递，故会多出两倍的访存开销，如果将被调用函数分配到寄存器的参数都使用寄存器传递，那么可以快不少，但是这是一个比较复杂的问题，首先我们假设一种简单且普遍的情况，函数调用的实参此时全部在寄存器中，相当于是要将此时寄存器中的变量转移到另一个寄存器中，又由于形参都在不同的寄存器中，故将实参所在寄存器与要移到的寄存器连上一条有向边就会发现这是一个基环外向树森林，对于环，我们需要将其拆去一条边，改边中变量的移动需要借助外来的辅助寄存器，先将变量存入辅助寄存器中，然后对于已经拆掉一条边的基环外向树森林进行拓扑排序，按照排好的拓扑逆序移动寄存器中变量到后一个寄存器中，最后将存入辅助寄存器中的变量亦如对应的寄存器中，于是省去了参数的压栈和弹栈

当然以上只是最普适的情况，考虑到全局变量以及常数都没有分配图着色寄存器，因此还需分别的情况讨论

$CLOCK$分配

对于在图着色中未分配到寄存器的变量可以使用操作系统中学过的页面分配的改良过的$CLOCK$算法：

- 维护一个指针用于遍历当前寄存器池中的寄存器
- 当要分配寄存器时，如果当前指向的寄存器为空，那么分配，同时指针指向相邻的下一个寄存器，如果不为空指针指向下一个寄存器重复之前的操作，如果一直没有空寄存器那么就直到指针回到要为变量分配寄存器时的初始位置，然后此时再循环一轮，在此轮过程中查看寄存器的脏位是否为`true`，不为`true`就分配，反之接着直到右回到要为变量分配寄存器时的初始位置，同时将该寄存器中的变量回写内存寄存器的脏位置为`false`，指针移动

注意将一个变量写回内存时脏位置为`false`，`ASSIGN`、`ALU`类指令的目的寄存器的脏位置为`true`

同时存在图着色分配的寄存器和$CLOCK$分配的寄存器，两种运行方式是完全不一致的：

- 图着色：只需读写两次内存——跨函数时压栈和弹栈，其它时候直接建立映射后无需加载变量，当作变量已经在寄存器中直接用
- $CLOCK$：跨基本、函数调用时快时要回写，同时解除映射，建立映射后也要加载变量

引用计数

对于**未分配到图着色寄存器**的临时变量，可以直接扫描中间代码中临时变量出现的次数，统计一遍，然后在翻译为`mips`
的过程中记录已经使用的次数，如果已经使用的次数与出现次数相同那么在翻译当前中间代码的过程中不需要将其回写内存，同时翻译完该条中间代码就可以将该变量与寄存器的映射关系清空

注意分配到图着色的临时变量不行，因为图着色已经是最优解了，这样做可能会产生副作用，而同时只有临时变量可以这样处理，因为临时变量不会跨基本块，这样做的目的主要是保证在我的$CLOCK$算法中该解除映射的寄存器将会在第一轮被选中，而不会被跳过

窥孔优化

对于中间代码可以在删除完四代码以及做完常量传播的时候做，比如：

- 对于`ADD`的加0，`SUB`的减0，`MUL`的乘1，`DIV`的除1，`MOD`的摸1
- 对于跳转到的刚好是下一条句子，那么可以删除；其次：

```
ASSIGN 8 a
NEZ_JUMP a LABEL
```

可以优化为：

```
JUMP LABEL
```

还有：

```
EQZ_JUMP a LABEL1
JUMP LABEL2
LABEL1:
```

那么可以优化为：

```
NEZ_JUMP a LABEL2
LABEL1:
```

上述情况其实很多时候都会出现，如果做了短路求值，以及常量合并的情况下

- 对于进行完常量传播的代码，有的可以直接算出结果，那么也可以用一条`ASSIGN`去替代之前的计算指令，有可能在之后的再次常量传播以及死代码删除的过程中就可以被优化掉

对于目标代码可以将：

- `move`到同一个寄存器的指令优化掉
- 连续的`sw`和`lw`如果是对于同一个寄存器和地址那么可以优化掉，比如：

```
sw $v0, 28($sp)
lw $v0, 28($sp)

# 或者
lw $v0, 28($sp)
sw $v0, 28($sp)
```

可以只保留第一条代码

- 如果检测到是个叶子函数，那么进入函数与离开函数时对于`$ra`寄存器的保存与恢复都可以删掉，同时即使不是叶子函数，通过`dfs`当前基本块前驱也可以判断是否有过别的函数调用，如果没有就不需要`load`一遍`$ra`
- 对于跳转指令和置位指令，有的在翻译过程中无法优化，需要窥空解决，一定要多分析这种情况，我就是靠着这种窥空一个点从20多直接前10了，比如：

```
li $a0, 10
sle $a0, $v1, $a0
bnez $a0 LABEL
```

可以通过窥空被优化为：

```
li $a0, 11
slt $a0, $v1, $a0
bnez $a0, LABEL
```

注意`sle`是一条伪指令，其实会翻译成3条指令，而`slt`和`sgt`不是，接着还可以优化为：

```
slti $a0, $v1, 11
bnez $a0, LABEL
```

由于`slti`对于16位立即数还不会被翻译为多条指令，同时一般测试也不会用太大的数去为难大家，于是又少了一条指令，但是注意不需要再优化了比如用`blt`这种，你会发现拓展后是一样的，于是原本的$1+3(\text{拓展后})+1=5$条就变为2条指令了，类似的还有很多注意就是多用`bnez`、`beqz`、`bne`、`beq`、`blt`、`bgt`、`slt`、`slti`、`sgt`等等，至于具体怎么窥空需要按照自己的翻译结果分析情况

无用函数调用删除

对于一些没有实际效益的函数调用我们可以直接去除关于它的调用，那么什么叫没有实际效益呢：

- 被调用函数没有返回值或者说返回值没有被用到

- 被调用函数没有对全局变量进行赋值修改
- 被调用函数没有 `IO` 语句，没有调用其它函数
- 调用函数传递的实参中没有数组地址

以上检查可以在语义分析生成中间代码的时候做，如果满足上述条件，那么此次函数调用可以直接在中间代码中删除

指令选择

我们可以注意到如下事实：

- `div` 的三操作数指令会多出判断除数是否为 `0` 的分支，可替换成 `div + mflo`
- `mul` 比 `mult + mflo` 要少一个指令
- 用 `addi` 来替代 `subi`，`subu`，`addu` 直接对立即数进行使用。在立即数少于等于 `16` 位下，即在 `[-32768, 32767]` 下效果显著。
- 跳转以及置位指令要用 `slti`、`slt`、`sgt`、`bne`、`bnez`、`beq`、`beqz`，而其它比如 `blt` 等等都会在 `Mars`
  被翻译成三条或者更多的指令

循环不变式外提

由于没有实现 `SSA`，因此该优化比较难做，但是速度的提升是显著的，在完成数据流分析后首先需要找到属于循环的基本块，可以在中间代码直接标记，也可以由以下算法得到，注意在此 `while` 循环均已经化为 `do-while` 循环：

- 计算到达每个节点的必经节点集合D：

  假定有x个节点：

  - 对每一个节点B_i，初始化$D(B_i)=\{B_1,B_2,\dots,B_x\}$，除$D(B_1)=\{B_1\}$
  - 每一步计算中，若当前节点B_i的父节点集合为$\{B_r,B_s,B_t\dots\}$，那么$D(B_i)=[\cap D(B_{fa})]\cup\{B_i\}$
  - 如果在一次迭代中，x个节点中没有节点的必进节点集发生变化，就退出
- 查找回边：
  - 如果一个程序流图中存在有像边$<B_i,B_j>$并且$B_j\in D(B_i)$，那么我们就说$<B_i,B_j>$为一条回边
- 查找循环：
  - 找出回边$<B_i,B_j>$
  - 则B_i,B_j必定属于回边$<B_i,B_j>$组成的循环L中，即$B_i\in L$且$B_j\in L$
  - 若$i\neq j$且i的父节点不在L中将它加入L中，对于求出的父节点，令其为i，重复执行这步操作，直至不再有新节点加入为止

我们可以记循环的入口节点为B_{entry}，即上述算法中的B_j，对于入口节点相同的循环，我们也可以将他们直接合并（可能是在做别的优化时导致），同时我们定义循环的出口节点集为$Exit$，即$B_t\in Exit$那么存在B_t的某个后继节点不在L中，接下来要判断那些是循环不变代码，然后外提，判断循环不变代码的算法如下：

- 一次查看L中各个基本块的代码，如果他们的use集合的变量为常数或者它们的in_{arrive}中它们自己的定义点都来自于L外，那么将这句代码标记

  > 注意这里要求每句代码的到达定义分析，可以在完成数据流分析的基础上再在基本块内逐句遍历求解，复杂度为线性，无需对整个中间代码多次迭代

- 接下来依次查看L中是否还有没有被标记的代码，如果它的use集合中变量满足为常数，或者它们的in_{arrive}中定义点在L以外，或者只有一个且该点代码已经被标记，则将此代码标记，这步操作需要不断迭代，直到不再有代码需要标记为止

但是上述算法所求到的并不是最终的循环不变式集合，因为此时只考虑了中间代码的使用变量，并未考虑它的定义变量，如果该中间代码在循环内不一定被到达，那么提出循环很有可能就错了，所以需要加强约束，对于所有标记的代码s，如果它是循环不变式，那么它需要同时满足以下条件：

- s所在节点是循环L的所有出口节点的必经节点，或者s在离开循环后都不再活跃

  > 即对于所有出口节点的后继节点中不在L中的节点，它们的所有in_{active}中都不包含s代码所定义（赋值）的变量

- s在循环L的**其它基本块**中没有定值语句
- 循环L中所有对于s定义（赋值）的变量的引用，只有s**所在的基本块**中对于它的定义可以到达

  > 注意此处约束均是保证s所在基本块即可，并不需要约束到s，网上参考资料均要求约束到s（可能是本身说得不太清楚），实际上在一个基本块内代码顺序执行，所以没有必要约束到s

最后就是按照所标记循环不变式的顺序，将符合要求的循环不变式s提到循环外，注意如果s的use集合中的变量是在L中定义（赋值）的（一定是循环不变式），**那么只有当这些定义（赋值）语句都提到循环外面，s才能提到循环外**

> 对于s的use集合中的变量是在L中定义（赋值）的，可以求s的in_{arrive}，然后查看其中对于use中变量的在L中定义（赋值）语句是否已经提出

至于具体如何提出，可以是新建一个基本块节点，将所有提出代码加入其中，然后将它指向循环入口，将之前指向循环入口的所有基本块指向该新建基本块，当然正常情况下L中的入口节点只有一个，且它的前驱中不属于L的节点只有一个，直接将提出代码放到入口的此前驱节点最后即可

总结

- 重视细节，不要忽略任何翻译的细节，很多时候可能在翻译到`mips`的过程中没有特别注意可以少一两句冗余指令，但是其实极少成多，可能会对于整个竞速排名有非常的影响；同时细节也包括了窥孔，这本来就是一个细节优化，这个优化可能很多人都不在意，但是其实这个优化非常有用，多分析一下自己的汇编又或者竞速中每次提交的`cycle`数的变化，以此为凭据来进行窥空绝对会有意想不到的优化结果，这绝对比做一个大的优化划算很多，比如我自己就可以说就是因为这个地方做的不错，进前十的点中有3个均是靠这种细节优化
- 一定要重视寄存器分配，有点特别慢那么大概率是寄存器分配的问题，只要寄存器分配的好，不用做太多优化也可以非常快
- 建议在翻译前就先做数据流分析，这样就不会在后续过程中对以往的架构进行大的调整，同时翻译为目标代码时建议按照基本快来翻译，不然代码会很乱
- 不要像我一样在直接在建立符号表生成中间代码的时候就分配好了栈空间，因为这样栈空间就定死了，虽然在翻译为目标代码的时候只用看符号表就可以知道在内存中的位置，但是这限制了你没有办法去做函数内联，而函数内联这个优化非常重要，如果做这个优化应该不止一个点会提升
- 对于数组元素，不要把他当作一个变量，建议是地址当作一个变量，然后数组元素当作是对于地址的一个运算操作所取得的一个结果，即在翻译为中间代码有单独的地址变量，而不是出现数组的某一个元素（不应该存在），只有这样在之后的优化过程中才可以将优化的范围扩大，使数组也可以参与进来，而不是进行保守处理，这一点可能不太容易讲清楚，需要自己体会，比如如下两个中间代码：

  ```
  ASSIGN 1 a[8]
  -------------------
  ADD a 32 T
  SAVE 1 T
  ```

  肯定是下面这种更好

- 如果开学就确定自己有时间的可以做`SSA`，不想设计中间代码，也可以直接做`llvm`（`llvm`本生就是一种`SSA`的中间代码），也有软院之前的参考教程，最后再生成`mips`，毕竟`SSA`形式的中间代码对应的优化很多，且网上大多都很详细，而普通的中间代码要想优化得很快得靠自己且在优化时更花时间，注意`SSA`要在前期生成中间代码的时候去做，后期基本没法改架构了

最后推荐一下做优化的顺序：

- 在生成中间代码的过程中就进行常量合并、`do-while`等等
- 生成完中间代码后先进行数据流分析，删除死代码、常量传播等等，这两个应该都会做吧
- 如果确定做图着色建议先做，避免之后变换寄存器分配需要对于架构大改
- 翻译`mips`过程中进行指令的合并和选择、运算削弱、乘除优化等等
- 最后就可以去做窥空、循环不变式外提、内联等优化

如果实在没时间可以不做循环不变式外提、内联、循环展开，这些都做完肯定就前十了，当然其实没做完也前十了