第六部分

毋哲

代码优化

任务：

代码优化模块通常在代码生成模块之前被调用，在不改变程序原有语义的前提下，为代码生成模块提供优化后的中间代码。代码优化可以分为与机器无关的代码优化（优化中间代码）和与机器有关的代码优化（优化目标代码）。其最终目的都是为了提高目标程序的运行效率，削减目标代码的存储空间。

我使用了的优化：

与机器无关的优化：局部公共子表达式删除。

与机器有关的优化：全局寄存器分配，临时寄存器池分配，窥孔优化。

代码优化的顺序：

基本块的划分

局部公共子表达式删除（对中间代码的优化，需要用到基本块信息）

全局寄存器优化（对目标代码的优化，需要用到中间代码和基本块信息）

临时寄存器池分配

窥孔优化

记录基本块的数据结构

我在考虑这个数据结构的时候，完全是为了更方便后面 DAG 图优化和全局寄存器分配优化而考虑了，因此不管在时间效率还是空间效率上，都不能算优秀。

既然是为了方便后面程序的编写，那这个数据结构应该如何来设计呢？我先自己思考了一下 DAG 图优化和活跃变量分析的过程，发现其中一个特点就是对于中间代码都是逐句来进行分析的，既然是逐句来进行分析，那么我就需要一个数据结构使得在逐句分析的时候既方便又快捷。由于我没有考虑效率，因此我使用的是一种类似于 hash 表一样的表来保存基本块的信息，只不过我的没有 hash 函数，或者 hash 函数就是 f(x)=x。

```
struct BlockModule{
    int  Before[100];//Before[0]存储的是这个块有几个前驱，Before[i]表示第 i 个前驱的块号。
    int  Follow[3];//Follow[0]存储的是这个块有几个后继，Follow[i]表示第 i 个后继的块号。
}Block[10000]; //Block[i]表示第 i 个块的信息
int MiddleBlock[CENTENCELENGTH];//MiddleBlock[i]表示第 i 巨中间代码属于哪个块
int IfEntry[CENTENCELENGTH];//第 i 条语句是不是入口语句，是的话为 1，不是的话为 0
```

这样做的好处是，生成基本块的时候可以逐句判断，在后面优化的时候用到基本块的信息可以很快的获取到。

比如现在处理第 i 句中间代码，判断它在哪个块，就是 MiddleBlock[i]，判断它所在块的

前驱，就是 Block[MiddleBlock[i]].Before,后继就是 Block[MiddleBlock[i]].Follow。

划分基本块的流程

```
//BlockIndex 是当前处理的块号，红色字为一些需要注意的地方
for (i=0;i<MiddleNum;i++)//逐句处理每一条中间代码
    {
        if (该句是 BEGIN 且下一句不是 END)
        {
            生成新块
            设置该句的下一句为块入口（）
        }
        else if (该句是 LABEL)
        {
            if (该句的上一句不是 JMP 且不是 JWNT 且不是 RET)
                生成新块
            else
                上一句已经生成了新块，因此直接赋值 BlockIndex 即可
            寻找 i 之前的，标签为此句标签的中间代码句，只找 i 之前，是因为 i 之后的还没有分配块。如果找到，则那句所在的块是该句 LABEL 所在块的前驱，则修改相应两块的 Block.Before, Block.Follow
            如果前一条语句不是跳转语句，则前一句所在的块是这个块的前驱，修改相应的 Block.Before,Block.Follow.
        }
        else if (该句是 JMP)
        {
            找该句之前的，标签与此句标签相同的语句 。如果找到，则那句所在的块是该句 JMP 所在块的后继，修改相应的块信息。
            如果下一句不是 END，则生成新块给下一句。
        }
        else if (该句是 JWNT)
        {
            找该句之前的，标签与此句标签相同的语句 。如果找到，则那句所在的块是该句 JMP 所在块的后继，修改相应的块信息。
            如果下一句不是 END，则生成新块给下一句，由于这是条件跳转语句，因此这句所在的块是下一句所在块的前驱，更新相应的块信息。
            }
        }
        else if (该句是 RET)
        {
            该块的后继是 EXIT
            如果下一句不是 END，则生成新块给下一句。
        }
        else if (如果该句不是 END)
        {
```

```
            MiddleBlock[i]=BlockIndex;
        }
        else if (该句是 END 且上一句不是 JMP 或者 RET)
        {
            将上一句所在的块的后继设为 EXIT
        }
    }
```

删除局部公共子表达式优化，何时应导出 DAG 图

在学习编译技术课中，我们可能对导出 DAG 图算法已经比较了解，而且也觉得不是很难，但是在实际进行编译程序编写的过程中，我们会发现实际情况并不是如此，如果是加减乘除等运算语句，还比较好处理，可是还有很多语句会让我们感到束手无策，比如说如果遇见一个 scanf 语句或者 printf 语句，我们应该怎样将其加入到 DAG 图中呢？

其实这种语句是无法完全按照生成 DAG 图的算法使其加入到 DAG 图中的，因为我们可以发现，由 DAG 图导出的语句的计算顺序，可能和原来的顺序完全不同，而类似 scanf 和 printf，call 这样的语句，它的有时却是有执行次序要求的。因此完全按照 DAG 图的算法将类似这样的语句导入 DAG 图中会产生错误的中间代码。而且我们无法知道 call 语句是否会改变已经加入到 DAG 图中的全局变量的值。

这里我给出两种解决办法，一种那就是遇到有类似 scanf 这样的语句就先将之前的 DAG 图导出，然后从这里之后重新建立 DAG 图进行优化，还有一种方法是王昊同学研究出来的，我们在后面会给出。

首先看第一种方法，我使用的可以加入 DAG 图的中间代码有：“+”“-”“*”“/”“=”“OPP”还有四种强制类型转换。其他的操作语句都是不能导入 DAG 图的。遇到上述十种语句，则建立 DAG 图，并以此将这些语句加入到 DAG 图中。而遇到其他的语句，则要先将此句之前的 DAG 图转化成中间代码输出，然后原样输出当前语句即可。我的这种方法可以保证生成的中间代码的正确性，但是显然无法保证其最优性，优化的效果比较有限，只有对上述十种操作可以进行优化。但是鉴于上述十种操作在程序中所占有的比率是很大的，因此优化的效果也还是不错的。

第二种方法是王昊同学的方法，他在我的基础上，又增加了很多其他的操作可以加入到 DAG 图中。以下是引用其论文中的论述。

> 针对 C0 扩充文法的特点，可以将一个基本块内的语句分为两类，一类为表达式语句（比如，＋，－，×，/），一类为非表达式语句（具体为 scanf，printf，赋值语句，参数）。需要注意的是，call 语句既会在表达式中出现，也会在非表达中出现，必须区分开来（在四元式中，表达式里的 call，其第四元为临时变量；而非表达式里的 call，其第四元为空）。
>
> 在 DAG 图建立的时候，对于非表达式语句来说，需要保证此类语句的先后执行顺序不变。具体的做法是在建立 DAG 图的时候，使前一句成为后一句的左操作数。对于表达式中的 call 语句来说，情况就更加复杂。因为在函数调用的过程中，由于缺乏跨函数数据流分析的支持，所以不知道全局变量的值在函数调用的过程中，是否发生了改变。因此需要采取保守做法，就是在处理到 call 语句时将全局变量的信息从节点表中删除，并且 call 语句也需要保证其相对先后执行顺序不变，类似处理非表达式语句，使其左操作数为前一个表达式语句。
>
> 最后在重新导出代码的时候，可以选择另一种导出方式。所给出的启发式是尽可能地先导出左子节点，而在本文的编译器中是尽可能先地导出右子节点，这样选择的原因有下：
>
> 1. 测试了几个关于全局变量值在函数调用后改变的样例，使用左子节点优先算法，所得的结果与关闭优化后出现了不一致；

2. 在 x86 体系结构里，对＋，－，×，/而言，被操作数（即左操作数）往往是先移入目标寄存器，然后再与操作数（即右操作数）进行运算操作，所以右子节点优先导出的话，就可以比较少的对操作数进行寄存器分配，这样效率会提升一些。

但是这两个原因更多的是经验与感觉的总结，由于所学知识有限及时间所限，无法更进一步地从理论上去探究其是对是错。

DAG 图的生成和导出

我认为 DAG 图的生成和导出还是比较容易的，只需要按部就班，一步一步按照实验书上的步骤来写可以了，唯一需要注意的一点就是临时变量的问题。

我们在生成中间代码的时候曾经生成了很多临时变量，比如一句 D=A+B+C，我们可能把它拆成 t1=A+B，D=t1+C,其中就用到了一个临时变量。在生成 DAG 图的时候，很有可能一个节点会对应很多的临时变量，这时候我们应该怎么处理呢？只需在这些临时变量中找出一个来代表这个节点，在以后的导出过程中，凡是使用到这个节点的语句，都只用这一个临时变量即可，其他的临时变量都可以删除不用。这样既节省了在代码生成阶段为临时变量分配的空间，也减少了优化有生成的中间代码的复杂度。

下面给出我使用的 DAG 图的数据结构仅供参考。

```
struct DAGform{
    int LeftNote;//左子节点的编号
    int RightNote;//右子节点的编号
    char op[STRINGLENGTH];//操作符
    int ifleaf;//是否是叶子节点
    int FatherNote[100];//父节点的编号
}DAG[100];//DAG 图
```

全局寄存器分配优化，活跃变量分析中，IN 集合与 OUT 集合里哪些变量应该被判断为是冲突的

活跃变量分析最终的结果是一张冲突图，而构建冲突图的原则是活跃范围重合的变量之间是重合的，基于这一原则，我们来考虑 IN 和 OUT 集合里那些变量应该被判断为是冲突的。

首先 IN[i]集合中的变量应该是互相冲突的，显然它们的活跃范围有重合，都一同进入了块 i。

其次，OUT[I]集合中的变量应该是互相冲突的，显然它们的活跃范围有重合，都一同走出了块 i。

这样判断之后，我们的冲突图就可以完全建好了吗？请看下面的例子。

假设有一个块，块中只有两条语句：

```
A=1;
printf(“%d”,C);
```

这时这个块的 IN 集合为{C},OUT 集合为{A}。那 A,C 这两个变量有冲突吗？假设没有冲突，它俩被分配给同一个全局寄存器 X，则在执行 A=1 之前，X 中存的是 C 的值，在执行 A=1 之后，X 中存的是 A 的值，此时，在执行 printf 语句的时候，输出的并不是 C 的值而是 A 的值。显然，这两个变量的活跃范围也是有重合的，它们不能被分配给同一个全局寄存器。

显然，这种情况也不可能总是出现，比如说如下情形：

```
printf(“%d”,C);
A=1;
```

A 和 C 的活跃范围就不重复，因此可以分配给同一个全局寄存器。

因此，在我们不深入研究每一个块中具体结构的时候，我们应该保守地将 IN[i]中的变量与 OUT[i]中的变量都互相冲突，才能保证生成的代码的正确性。

假设 IN[i]与 OUT[i]的交集为 U，其实我们只需要将 IN[i]-U 中的变量与 OUT[i]-U 中的变量互相冲突即可。

全局寄存器分配的程序实现

活跃变量分析：很简单，只需要按照算法一步一步来即可。使用的四个集合我使用数组来实现。

全局寄存器的分配：冲突图我使用邻接矩阵来表示，使用时时间效率较高，但是空间效率较低，但是完全可以接受。全局寄存器分配只是针对局部变量的，因此，全局变量不会分配给全局寄存器。这里需要注意的是，如果你给一个函数的形参分配了全局寄存器，则在该函数最一开始，需要调用该函数时，将传进来的形参值放入全局寄存器，或者形参也不参与全局寄存器的分配。具体的程序实现也不难，只需要按照算法步骤来即可。

使用临时寄存器池的算法

进入基本块时，清空临时寄存器池

为当前中间代码生成目标代码时，无论临时变量还是局部变量（抑或全局变量和静态变量），如需使用临时寄存器，都可以向临时寄存器池申请

临时寄存器池接到申请后，

– 如寄存器池中有空闲寄存器，则可将该寄存器标识为被该申请变量占用，并返回该空闲寄存器

– 如寄存器池中没有空闲寄存器，则将在即将生成的代码中不会被使用的寄存器写回相应的内存空间，标识该寄存器被新的变量占用，返回该寄存器

在基本块结尾，或者函数调用发生前，将寄存器池中所有被占用的临时寄存器写回相应的内存空间，清空临时寄存器池

使用了上述临时寄存器的分配方法，并不能保证最优的临时寄存器的分配，但是它已经是一个比较快且比较优的分配方法了，考虑到编译器的效率，我们可以采取这种方式来分配临时寄存器。

使用寄存器后的代码生成

如果在之前代码生成阶段，你已经为代码优化有了足够的考虑，那么此时将寄存器的优化加入进去会变得非常简单，而且得心应手。

在代码生成阶段，我曾经用如下程序举例，当时就应该为代码优化阶段做考虑。

```
int AddAsm()//+语句四元式对应的汇编代码生成
{
    GetStr(Middle[AsmIndex].op2,s1);//获得第 2 元所对应的汇编字符串，可能是寄存器，可能是全局变量，可能是立即数，可能是地址
    GetStr(Middle[AsmIndex].op3,s2); //获得第 3 元所对应的汇编字符串，可能是寄存器，可能是全局变量，可能是立即数，可能是地址
    addr=GetStr(Middle[AsmIndex].op4,s3);//获得第 4 元所对应的地址,-2 表示占用了全局寄存器，这里不需要考虑立即数的情况。
    fprintf(outAsm,"\t\tmov\t\teax , %s\n",s1);
    fprintf(outAsm,"\t\tadd\t\teax , %s\n",s2);
```

```
    if (!IfOptimize)//如果没有优化的情况下，直接 eax 中的值放回 s3 表示的地址中。
        fprintf(outAsm,"\t\tmov\t\t%s , eax\n",s3);
    else
        //在这里加临时寄存器分配的优化，开优化的情况下，eax 的值可能放回到全局寄存器中或临时寄存器中
    {}
    return 1;
}

*****************************************************************************

int GetStr(char op[],char s[])//获得 op 参与汇编运算的字符串，取值可为寄存器，全局变量，地址，立即数
{
    addr=-1;
    RegNum=GotWholeReg(op);//查看是否占用全局寄存器，如果占用的话返回寄存器编号
    if (RegNum>=0)
    {
        addr=-2;
    }
    else
    {
        RegNum=OccupyReg(op);//查看是否占用临时寄存器，如果占用的话返回寄存器编号
        if (RegNum>=0)//已占用寄存器
        {
        }
        else
        {
            addr=FindAddress(op);//寻找变量对应的地址
            if (addr>-1)
            {//说明是局部变量
                //返回地址字符串：“[ebp-X]”
            }
            else
            {
                if (('a'<=op[0] && 'z'>=op[0])||
                    ('A'<=op[0] && 'Z'>=op[0])||
                    (op[0]=='_'))
                {//说明是全局变量
                    //返回变量名
                }
                else
                {//说明是立即数
```

```
                    //返回立即数的字符串。
                }
            }
        }
    }
    return addr;
}
```

上述程序的红色部分即为当时为代码优化而做的考虑，如果去掉红色部分，程序照样可以运行。

下面，我们就来看看，我们是如何非常方便的将寄存器优化加入到上述代码中。

```
int AddAsm()//+语句四元式对应的汇编代码生成
{
    GetStr(Middle[AsmIndex].op2,s1);//获得第 2 元所对应的汇编字符串，可能是寄存器，可能是全局变量，可能是立即数，可能是地址
    GetStr(Middle[AsmIndex].op3,s2); //获得第 3 元所对应的汇编字符串，可能是寄存器，可能是全局变量，可能是立即数，可能是地址
    addr=GetStr(Middle[AsmIndex].op4,s3);//获得第 4 元所对应的地址,-2 表示占用了全局寄存器，这里不需要考虑立即数的情况。
    fprintf(outAsm,"\t\tmov\t\teax , %s\n",s1);
    fprintf(outAsm,"\t\tadd\t\teax , %s\n",s2);
    if (!IfOptimize)//如果没有优化的情况下，直接 eax 中的值放回 s3 表示的地址中。
        fprintf(outAsm,"\t\tmov\t\t%s , eax\n",s3);
    else
        //在这里加临时寄存器分配的优化，开优化的情况下，eax 的值可能放回到全局寄存器中或临时寄存器中
    {
        //全局寄存器优化
        if (addr==-2)
        {
            fprintf(outAsm,"\t\tmov\t\t%s , eax\n",s3);
        }
        else
        {
            //临时寄存器分配优化
            AssignTempReg(s3,addr);
        }
    }
    return 1;
}

*****************************************************************************

int GetStr(char op[],char s[])//获得 op 参与汇编运算的字符串，取值可为寄存器，全局变量，
```

```
地址，立即数
{
    addr=-1;
    RegNum=GotWholeReg(op);//查看是否占用全局寄存器，如果占用的话返回寄存器编号
    if (RegNum>=0)
    {
        MakeReg(RegNum,s);//将相应的寄存器的字符串名字放入 s 中
        Occupy(RegNum,op);//表示变量 op 占用了编号为 RegNum 的寄存器
        addr=-2;
    }
    else
    {
        RegNum=OccupyReg(op);//查看是否占用临时寄存器，如果占用的话返回寄存器编
号
        if (RegNum>=0)//已占用寄存器
        {
            MakeReg(RegNum,s); //将相应的寄存器的字符串名字放入 s 中
        }
        else
        {
            addr=FindAddress(op);//寻找变量对应的地址
            if (addr>-1)
            {//说明是局部变量
                //返回地址字符串：“[ebp-X]”
            }
            else
            {
                if (('a'<=op[0] && 'z'>=op[0])||
                    ('A'<=op[0] && 'Z'>=op[0])||
                    (op[0]=='_'))
                {//说明是全局变量
                    //返回变量名
                }
                else
                {//说明是立即数
                    //返回立即数的字符串。
                }
            }
        }
    }
    return addr;
}
```

上表格中，蓝色部分为加入优化后的代码。很短。这样，我们很容易的就将两种寄存器

分配加入到了代码生成中，而且测试容易，可靠性相对较高，也不会打乱、损害原来程序的正确性。