优化文档

编译实验中，生成到最终的 mips 汇编可以做很多优化，主要分成了中端优化和后端优化。以下是本人在这两部分优化过程中所遇到的问题和解决办法的汇总。

一、中端优化

中端优化指的就是机器无关优化，即对中间代码进行简化。

1.1 SSA

在理论课上我们学习了 SSA 形式的中间代码可以为后续优化带来很多便利，因此本人在中端做的第一个优化就是把从 AST 生成的 LLVM 通过 `Mem2Reg` 转化成 SSA 形式。

实现 `Mem2Reg` 需要有每个定义点所在块的支配集，支配边界的信息。因此准备工作包括：生成流程控制图(CFG)，生成（严格）支配集，生成支配树，生成支配边界集合。

生成流程控制图：由于每个基本块的最后一条指令都是跳转指令(`br`、`ret`)，因此根据基本块的最后一条指令就可以生成 CFG，但是实际上由 AST 生成 LLVM 可能会在基本块的最后产生很多跳转指令，比如下面的例子。因此 AST 转 LLVM 时可以对当前的基本块的最后一条指令做特判，如果已经是一条跳转指令，则不需要再生成任何代码。

```
int test() {
    printf("Hello World");
    return 1;
    return 2;
}
int main() {
    test();
    return 0;
}
```

生成严格支配集：这部分由于没有教程，需要自己设计，但因为笔者没有想到很好的办法，因此选择了最朴素的做法——根据定义暴力计算。X 块的支配 Y 块，充要条件是从入口块开始到达 Y 的任何一条路径都必然经过 X。因此求取任何基本块的支配集，可以采用深度优先搜索，当遍历到当前计算的基本块时就终止这条路的搜索。上述完成一次深度优先搜索，记录经过的基本块，从总的基本块减去被记录的块就是当前计算的基本块的支配集了。

生成支配树：根据支配集可以很方便地得到支配树。算法的大致思路就是：对所有基本块进行遍历处理，记为 X，遍历其支配的基本块 Y，如果 Y 的支配子树已经生成，则把其加入到 X 的子节点，并且把 Y 的支配集从 X 的支配集中剔除；如果 Y 的支配子树尚未生成，则递归处理基本块 Y。

生成支配边界：这部分完全按照教程给出的伪代码写就可以了。

`Mem2Reg` 优化的思路是：遍历每个基本块中的每个指令，找到定义点（`alloca` 指令以及对应的 `store` 指令），这些定义点的基本块的所有的支配边界都插入空 `phi` 指令。完成上述操作后，在支配树上做深度优先遍历，删除 `alloca` 指令，并为其建立堆栈，对于 `store` 指令，将存入的值push到栈顶，并删除，对于 `load` 指令，取出栈顶元素进行替换。然后查看当前基本块的后继块(CFG)，对所有 `phi` 指令，从对应的栈顶取出值填入。

1.2 死代码删除

`Mem2Reg`优化过程中删去了大部分的`alloca`、`store`和`load`指令，其实已经删去了一部分死代码，即无用的重复赋值语句。但是中间代码中还是会保留很多可能存在的死代码。

死代码可以粗暴地理解为对程序输出结果无关的代码，比如下面两个程序的结果是一致的，定义语句`int a = ...`就是死代码。

```
int main() {
    int a = 1 + 2 + 3 + 4 + 5;
    return 0;
}

int main() {
    return 0;
}
```

判断一条指令属不属于死代码，可以依据 def-use 链，即查看该指令后续是否被使用了，如果使用则不属于死代码。当然这样判断会错过许多可以优化的机会，比如`instr1 -> instr2 -> instr3`，指令 3 使用指令 2，指令 2 使用指令 1。如果指令 3 不被使用且对程序的输出没有任何作用，则仅会删除指令 3。然而实际上，上述指令均应识别为死代码。

因此死代码删除被调整为一个递归过程：首先确定一些对程序有用的指令：`call`、`br`、`ret`、`store`。对于每一条指令，追踪其 def-use 链，检查其user是否是上述指令的一种，如果是则保留该指令，否则进行删除。下面展示出伪代码：

```
private static boolean deleteDeadInstr(Instr instr) {
    ... // 如果是有用指令，返回 true

    boolean isDead = true;
    for (Value user : instr.getUserList()) {
        isDead &= deleteDeadInstrDFS((Instr) user);
    }
    if (isDead) {
        // 删除 instr
    }
    return isDead;
}
```

1.3 常量折叠

常量的计算可以在编译期间完成，对于计算型的程序，这可能带来很大的性能提升。

对于计算指令两个操作数都是常数，可以直接计算。计算可以迭代进行，比如`b=a+1，c=b+1`，如果b不被其他指令使用，则可以简化成`b=a+2`。

此外，还有一些简单的常量折叠也可以实现：

`a+0`->`a`，`a-0`->`a`，`a*0`->`0`，`a*a`->`a+a`，`a/a`->`1`，`a%a`->`0`。

在做常量折叠时需要注意维护指令间的 use-def 链。

常量折叠优化可以分到多个过程执行，比如 AST 中 addExp、mulExp 结点转 LLVM 时，就可以完成一部分常量折叠。在 `Mem2Reg` 后，中间代码会暴露出更多可以做常量折叠的机会，包括后文提到的 `GVN` 优化也会产生常量折叠的机会。

1.4 基本块合并

生成的 LLVM 中间代码会出现很多一个基本块中只有一条 `br` 指令，且目标块仅有次一个入口，这种情况下可以将两个基本块做合并，减少一条跳转指令。这看似不会有多少性能提升，但对于循环指令，优化收益还是可观的。

合并基本块同样需要追踪 def-use 链，合并时需要更改对应 `br` 指令的操作数，保证合并后的 use-def 链正确。

1.5 GVN 优化

`GVN` 优化个人认为就是提取公共子表达式，通过复用之前可达的前述指令，从而减少指令数量，实现优化。因为中间代码被转化成 SSA 形式，所以提取公共子表达式变得简单许多。

本人实现的 `GVN` 策略是：在支配树上做深度优先遍历，对于经过的每一个指令，用代表性的字符串进行标记，若后续指令的操作数 `operand` 的代表性字符串已经出现，则可以使用前面的指令进行替换。需要注意的是，遍历过程的回溯是需要恢复现场的，保证标记的指令都是可达的。比如下图block1 可以跳转到 block2 和 block3，因此 block1 的指令对于 2、3 都是可见的，但block2、block3不是父子结点关系，指令不互相可见，因此标记指令是严格依赖于支配树的关系。

```
block1
  |
  |--block2
  |--blcok3
```

"用代表性的字符串进行标记" 中的代表性字符串是不包含指令的寄存器信息的。比如 `%local_2 = icmp ne i32 %local_1, 0` 的代表性

字符串就是 `icmp ne i32 %local_1, 0`。

当然也不是所有指令都可以当作公共子表达式，比如 `load` 指令就不能被当作公共子表达式，举例说明：

```
store i32 1, i32* @global_0
%local_1 = load i32, i32* @global_0
call void @putint(i32 %local_1)
store i32 -1, i32* @global_0
%local_2 = load i32, i32* @global_0
call void @putint(i32 %local_2)
```

显然 `%local_2` 不能被替换成 `%local_1`。

二、后端优化

后端优化指生成汇编指令过程中或生成后进行的代码优化。

2.1 消除 phi

中端优化通过插入 `phi` 指令，得到 SSA 形式的 LLVM，但因为 `phi` 指令无法直接翻译成 mips，因此需要先消除 `phi` 指令。按照实验教程给的思路是：找到 `phi` 指令使用的基本块，在基本块的尾部插入 `move` 指令。但是对于有多个后继块的前驱块来说，直接插入 `move` 指令会导致程序语义错误，所以需要一个中间的基本块承接这些`move`指令。这部分并不困难，按照教程就可以完成。

消除`phi`会产生很多的 `move`指令，一方面原因是因为是`phi`可能关联多个基本块，每个基本块需要产生一条`move`指令，另一方面是多个`phi`指令是并行的，而一对一地生成`move`指令是会导致错误，因此需要引入临时寄存器来存储，比如下面的例子：

```
move $1 $2
move $3 $1
```

需要再引入一条新的`move`指令：

```
move $1_temp $1
move $1 $2
move $3 $1_temp
```

实际上，可以把每一条`move`指令当作是一个图上的结点，如果指令 x 使用的寄存器是指令 y 存入的寄存器，则说指令 x 是 指令 y 的前驱。如此可形成一张有向图，依靠拓扑排序重新排序`move`指令，比如上面的例子就可以重新排序得到：

```
move $3 $1
move $1 $2
```

当然，拓扑排序要求无环图，因此需要先检查是否有环，如果有，则必须引入新的`move`指令，破除环路。

笔者在消除 `phi`指令时虽然是放在生成 mips 前做的，但还是直接在中间代码层面进行消除。需要自定义新的指令`pcopy`仿照`move`指令，`palloca`定义额外`move`指令的中间变量，同时考虑到生成 mips 时需要先分配寄存器后使用，所以把 `phi`转移到其支配树的父节点上，保留下来，当作定义点，而实际不会生成任何汇编代码，仅用于分配寄存器。

2.2 寄存器分配

实验教程推荐使用的是图着色算法，但是流程比较复杂，因此我选择**线性寄存器分配**的算法，即计算每一条指令的活跃区间。当指令活跃时分配保留其寄存器，不活跃时释放寄存器，最终的效果还比较突出。

计算指令活跃区间的方法运用了实验课的数据流分析法，采用逆序的顺序依次计算每个指令的活跃信息。

```
in[s] = use[s] ∪ (out[s]-def[s])
out[B] = ∪in[P]
```

`in[s]`是指令 s 入口的活跃变量集合，`out[s]`是指令 s 出口的活跃变量集合，`use[s]`是指令使用的变量集，`def[s]`为指令 s 本身。

P 是 B 的后继块，所有基本块的出口活跃变量集是其所有后继块入口活跃变量的并集。

计算出每一个指令的活跃区间后，就可以在生成 MIPS 前先为每一个指令分配好寄存器或栈上空间，建立映射关系。之后携带这部分映射信息进行 MIPS 生成。

2.3 指令选择

指令选择对最终的汇编程序效率影响很大。因为 LLVM 翻译成 mips，几乎是一对一地翻译，所以指令选择的好坏从整个程序来看会带来很大的影响。

对于乘除法，可以用加法或位运算尽可能进行替换。

对于`gep`中间代码的翻译，需要计算数组的地址信息，因此有大量的计算操作，减少一条指令最终带来的收益都是可观的。如果是常量计算则直接在编译期间完成。在转成成地址时，需要乘上类型的字节大小，又因为所有类型的字节都是2的幂次，因此可以用移位指令(`sll`)替代。

对于`call`指令，会产生非常多的存取内存指令。笔者采用的策略是，首先需要将目标函数所用的寄存器存入到栈上，然后将参数传给`$ai`寄存器，其中前三个寄存器存放到`$a1`、`$a2`、`$a3`(`$a0`寄存器仅用于系统调用)，多余的参数则推入栈中，符合 MIPS 标准。在转函数调用指令时，本人遇到了一个容易被忽视的错误，就是函数的参数是所在函数的形参。比如下面的情况，如果直接将参数存入到指定的位置，就会变成`move $a1$ a2`、`move $a2 $a1`，这样参数传递就是错误的。一种可行粗暴的方法就是所有传参都从栈上取，因为寄存器已经被`sw`到栈上，所以取栈上数据可以保证正确，另外高效的方法是采用类似`phi`指令的并行化处理的思路，使用拓扑排序调整指令顺序，必要时引入中间变量(临时寄存器)破除环路。

```
1 void test(a, b) {
2     test(b, a);
3 }
```

其余的指令翻译可以一一对应，需要注意的是跳转指令`br`以及`zext`指令，其操作数必然有`icmp`指令，而`icmp`指令不便对应到 MIPS 中，因此所有的`icmp`都需要和`br`或`zext`联合，利用`slt`、`sltu`、`beq`、`bne`、`xor`等指令进行组合来翻译。

2.4 乘除法优化

乘除法优化我主要针对操作数是 2 的幂次的情况进行优化。

对于乘法，可以用移位指令`sll`代替。

对于除法或取模运算，可以用`sra`、`andi`指令代替。比如`a/64`，64的幂次是 6，因此可以代替成`a >> 5`。

教程中给出了除法优化更一般的算法，但是受限于时间紧迫，这部分只能暂时放弃。