21231115-张家仪-优化文章

December 2023

1 整体设计

最顶层的 compiler 中对前中后端的协同运作进行了调度。前中后端分别保存在 frontend、llvm、backend 文件夹中, 中间代码优化器与 llvm 的分析器 IRbuilder 同放在 llvm 中, 目标代码优化器和基于虚拟寄存器的优化器也和 mips 分析器一起放在 backend 中。

IRbuilder 生成的中间代码经过 MidOptimizer 优化后送入 BackEnd 进行 Mips 指令的生成, 生成后的 Mips 代码依次通过 BackVirtualOptimaizer 和 BackOptimizer 进行优化后最终输出。

2 优化思路

在本次作业中，我主要进行了常量传播、公共子表达式提取、死代码删除和绝对跳转的优化等，我的大部分优化主要做在分配虚拟寄存器后，分配物理寄存器前，这也得益于我的虚拟寄存器分配是 SSA 的。

2.1 常量传播

在进行常量传播时，最开始只对声明为 const 类型的变量做了常量量播，后来发现这样传播后还有很多变量在运行时明显值已确定，且很易判断其值，但却没有被传播。因此改进了传播方法，在后端分配虚拟寄存器后，分配物理寄存器前增设一次常量传播优化。在保证虚拟寄存器为 SSA 的前提下，当某个寄存器被 Li 指令赋值时，即认为其为常量，并对其进行传播。

在此之后，发现最初的常量传播没有对跳转指令进行传播，即使条件跳转指令实际可以判断出来必然执行或必然不执行，仍然会将其当作条件跳

转指令并再次产生一个基本块，这样大大缩小了传播的范围，因此我对跳转指令也增设了常量传播，在跳转指令的结果确定时，将其替换为相应的绝对跳转指令。

2.2 绝对跳转优化

当跳转的目标地址及是否跳转结果是确定的时候，我即将该跳转语句删去，并将目标基本块中的语句克隆后加入原基本块中，并不断进行循环迭代，直到基本块都不再进行更新。特别的，每次克隆我都会在原虚拟寄存器后加入不同的后缀，以保证其 SSA 性不变。

```
@Override
public Object clone() throws CloneNotSupportedException {
    MipsInstruction m = (MipsInstruction) super.clone();
    m.changeName();
    return m;
}
```

之所以这样进行优化，是因为我在进行常量传播时是不能跨块的，因此进行这样块连接的绝对跳转优化不仅有益于减少无用的跳转语句和上下文保存，还能增大常量传播的范围。

2.3 局部 MemToReg

由于我的寄存器分配策略不支持跨基本块分配寄存器，因此我只在同基本块内，不包含函数调用且不存在对下标为变量的数组元素进行访问的指令序列进行了局部的 memToReg，通过将对同一地址的 lw 和 sw 转换为寄存器间的 move 指令减少执行周期数。

2.4 叶函数优化

当调用叶函数时，因为其不再调用其他函数，所以理论上 $ra 寄存器中的值是不会改变的，因此不需要对 $ra 的值进行存取，所以对其进行了优化。当某个函数不调用其它函数时，不将其 $ra 的值存入内存，以减少 mem 指令的周期数。

2.5 死代码删除

如果经过优化后某个寄存器中的值不会被其他指令所使用，那么产生其中值的指令也不再有存在的必要了，因此可将其从全部代码中删去，以减少无用的指令。在进行其他优化后可能会产生许多这样的指令，均在死代码删除优化中将其删去。

2.6 公共子表达式提取

当在同一个基本块内，同一个运算结果被多次执行并赋值给不同的虚拟寄存器时，我会使用其第一个被赋值的虚拟寄存器替换其余的所有相应虚拟寄存器，其剩余的求值指令便转换为了死代码，在死代码删除优化中删去。

```
public class MipsHashInst {
    private Class c;
    private MipsInstruction instruction;

     @Override
    public int hashCode() {
        int result = c != null ? c.hashCode() : 0;
        result = 31 * result + (instruction != null ? instruction.
            hashCode() : 0);
        return result;
    }
}
```

为了便于管理与统计，加快优化效率，我选择用 Hash 为每个指令根据其类型及使用的虚拟寄存器为其分配相应 Hash 值，若其 Hash 值一样则认为其为其含有公共子表达式。

2.7 乘除优化

通过查阅相关论文，我用 sra，srl 等指令替换了 div 指令，以减少指令条数。特别的，当乘法的立即数为 2 的指数时，我使用移位操作替换乘法运算，以减少指令的执行周期。

2.8 指令选择

同样是对立即数进行仅需要一条 addiu 的语句转换成 subiu 可能会扩展为两条，因此我将所有的 subiu 均替换成 addiu 以减少一些不必要的指令条数。同时在生成目标代码后，将所有的 la 伪指令转换成以 gp 为基的访存指令以减少指令条数。

3 遇到的困难和解决方法

3.1 SSA 维护问题

在进行绝对跳转的优化时，一开始在指令克隆后，未对虚拟寄存器命名添加后缀，导致虚拟寄存器失去了 SSA 性，在进行常量传播优化时，将错误的常量值赋值给了虚拟寄存器导致了错误，因此在克隆时为虚拟寄存器增加名字后缀使得其仍具有 SSA 性解决了这个问题。

3.2 数组访存问题

由于在访问数组时需要先将数组首地址赋值给寄存器，再用数组首地址加上偏移量才能得到该元素的地址，当偏移量为变量时，其目标访问元素实际是无法确定的，在进行局部 MemToReg 时，如果不对其进行特殊处理可能会导致无法正确对每个地址的 store/load 指令进行分类，因而使得 load 出的地址不是原始最新存入该地址的数据，进而造成错误。因此我对这种情况进行了特殊处理，每当遇到对未知地址的 store/load 指令时，便停止局部 MemToReg，将所有当前未存入内存的数据存入内存，再次进行 MemToReg。