优化前的观察

为了后续优化的顺利进行, 我们在优化前需要进行观察并发掘中间代码的性质. 以下给出一些基于本人实现生成的 llvm-ir 的性质, 并且为了利于优化多遍(pass)化的设计, 在优化过程中不会破坏这些性质.

性质的证明从简, 并且只证明初始状态下具备这种性质, 不证明进行优化不会破坏这些性质(这些性质在优化前后的一致性通常是显然的).

SSA形式

在基本块内, 每个变量只会在定义时被赋值一次, 这是一个很重要的性质, 在后续优化中会常常用到.

关于 store 和 load 操作的地址

`store` 和 `load` 操作的地址一定时下列三种之一.

1. `getelementptr` 指令
2. `alloca` 指令
3. 全局变量

其中 `getelementptr` 获取的一定时数组中的地址, `alloca` 指令一定是局部非数组变量, 直接用 `store` 和 `load` 操作全局变量一定是全局非数组变量, 所以不同类指向的地址一定不同.

更进一步, `alloca` 指令指向相同的地址(更严谨一点是相同对象)当且仅当是同一条 `alloca` 指令, 全局变量指向相同的地址当且仅当是同意相同变量, 这两点不言自明. 但是 `getelementptr` 指令即便不同, 也有可能指向相同的地址, 比如如下的例子:

```
void f(int a[], int b[][2]) {
    a[0] = 1;
    b[1][0] = 2;
}

int main() {
    int c[2][2];
    f(c[1], c);
}
```

函数 `f` 中的两个赋值语句会产生两个不同的 `getelementptr` 指令, 但是会指向同一片地址空间.

指令被使用的位置

除了 alloca 指令, 所有指令产生的临时寄存器中的值只会在本基本块中使用(函数调用不划分基本快).

本人没有进行消除全局公共子表达式的优化, 所以这条性质会一致保持.

中间代码优化（局部优化）

1. 消除公共子表达式

优化局部非数组变量的 load / store

观察

考察下面的代码

```
int x;
x = getint();
int a = x;
```

会生成如下中间代码:

```
    %Local_Var_0 = alloca i32 ; 为局部变量 x 分配空间
    %Local_Var_1 = call i32 @getint()
    store i32 %Local_Var_1, i32* %Local_Var_0 ; 保存 x
    %Local_Var_2 = alloca i32
    %Local_Var_3 = load i32, i32* %Local_Var_0 ;  取出 x
    store i32 %Local_Var_3, i32* %Local_Var_2
```

注意到 `%Local_Var_3 = load i32, i32* %Local_Var_0` 取用的数据的来源与 `store i32 %Local_Var_1, i32* %Local_Var_0` , 实际上有 `%Local_Var_3 = %Local_Var_1` .

故产生将指令改为"赋值"操作的想法, 注意llvm-ir并没有赋值指令，容易想到是利用加 0 实现赋值, 但其实对于优化的效果益处甚小. 这部分之后将采用更好的方法加以优化.

优化效果分析

1. `load` 的效率较低，如果改写成赋值可以直接加快效率.
2. 优化后指令形如 `x = y` , 可以采用复制传播的思想进行进一步优化.
3. 复制传播后可能产生相同指令, 利于后续进一步优化.

优化原理及方式

原理很简单, 对 `load` 操作的地址空间向前追溯上次对其的 `store` 操作.

根据之前对 `store` 和 `load` 操作的地址是否相同的讨论, 如果 `load` 操作的是 `alloca` 指令或者全局变量, 向前找到最近的相同的操作同意地址 `value` 的 `store` 即可.

但是如果 `load` 操作的是 `getelementptr` 指令, 我们最好的情况下只能判别两个 `getelementptr` 指令是否相同, 难以判别是否不同, 所以我们找到最近的操作 `getelementptr` 的 `store` , 如果其 `getelementptr` 指令和 `load` 相同, 我们可以利用赋值的方式优化, 否则不但由于 `getelementptr` 指令可能**不同**, 不能将 `store` 的存储的值赋值给当前 `load` , 并且由于 `getelementptr` 指令可能**相同**, 不能继续向前寻找对应 `store` . 所以只能放弃对当前 `load` 的优化.

前文提到 '采用更好的方法加以优化' , 是指直接去掉这条 `load` 指令, 将后续对其的使用全部直接改为 `store` 要存储的值. 这里的正确性依赖于 **除了 alloca 指令, 所有指令产生的临时寄存器中的值只会在本基本块中使用** 这项性质.

例子中的中间代码会优化成下面的样子:

```
    %Local_Var_0 = alloca i32 ; 为局部变量 x 分配空间
    %Local_Var_1 = call i32 @getint()
    store i32 %Local_Var_1, i32* %Local_Var_0 ; 保存 x
    %Local_Var_2 = alloca i32
    store i32 %Local_Var_1, i32* %Local_Var_2
```

合并相同指令

观察

考察如下代码：

```
int x, y;
x = getint();
y = getint();
int a = x + y, b = x + y;
```

经过前面的优化会生成如下中间代码

```
    ; 为 x, y 分配空间
    %Local_Var_0 = alloca i32
    %Local_Var_1 = alloca i32
    %Local_Var_2 = call i32 @getint()
    store i32 %Local_Var_2, i32* %Local_Var_0
    %Local_Var_3 = call i32 @getint()
    store i32 %Local_Var_3, i32* %Local_Var_1
    %Local_Var_4 = alloca i32
    %Local_Var_7 = add i32 %Local_Var_2, %Local_Var_3
    store i32 %Local_Var_7, i32* %Local_Var_4
    %Local_Var_8 = alloca i32
    %Local_Var_11 = add i32 %Local_Var_2, %Local_Var_3
    store i32 %Local_Var_11, i32* %Local_Var_8
```

观察到 `%Local_Var_7 = add i32 %Local_Var_2, %Local_Var_3` 和 `%Local_Var_11 = add i32 %Local_Var_2, %Local_Var_3` 指令式相同的, 考虑改为 `%Local_Var_11 = %Local_Var_7`

优化效果分析

1. 优化后指令形如 `x = y`, 结合复制传播的思想可以有效减少指令条数.

优化原理及方式

对除了 `alloca`, `ret`, `call` 等会(或可能会)改变内存数据或者运行流的指令, 找到块内第一个何其和其相同的的指令斤进行替换即可.

通过上面两个优化环节，**本质上完成了局部的消除公共子表达式的优化。**因为公共子表达式最终一定会产生相同的指令, 合并相同指令就可以实现消除公共子表达式.

核心代码:

```
    for (int i = 0; i < instruction.size(); ++i) {
        Instruction* inst = instruction[i];
        if (inst->getValueName().size() == 0) continue;
        if (!inst->getType().compare("alloca") ||
            !inst->getType().compare("call i32"))
            continue;
        if (inst->getType().compare("load")) {
            for (int j = i - 1; j >= 0; --j)
                if (inst->isSame(instruction[j])) {
                    inst->setSameInst(instruction[j]);
                    break;
                }
        } else {
            Value* loadPtr = inst->getOperand()[0];
            Instruction* loadPInst = dynamic_cast<Instruction*>(loadPtr);
            for (int j = i - 1; j >= 0; --j) {
                if (!instruction[j]->getType().compare("store")) {
```

```
                    Value* storePtr = instruction[j]->getOperand()[1];
                    if (storePtr->isSame(loadPtr)) {
                        inst->setSameInst(instruction[j]->getOperand()[0]);
                        break;
                    } else {
                        Instruction* storePInst =
                            dynamic_cast<Instruction*>(storePtr);
                        if (loadPInst == NULL && storePInst == NULL) break;
                        if (loadPInst == NULL || storePInst == NULL) continue;

                        if (!loadPInst->getType().compare(
                                storePInst->getType()))
                            break;
                        else
                            continue;
                    }
                } else if (inst->isSame(instruction[j])) {
                    inst->setSameInst(instruction[j]);
                    break;
                }
            }
        }
    }
```

2. 消除常量指令

在生成 llvm-ir 的过程中, 本人其实已经将常数表达式进行了优化(即根据运算符两边的子表达式是否均为常数表达式来决定当前表达式是否为常数表达式), 但是经过前面的优化, 可能产生新的常量指令, 如下面的例子

```
int x = 1;
a = x + 1;
```

优化前已无常量指令:

```
    %Local_Var_0 = alloca i32
    store i32 1, i32* %Local_Var_0
    %Local_Var_1 = alloca i32
    %Local_Var_2 = load i32, i32* %Local_Var_0
    %Local_Var_3 = add i32 %Local_Var_2, 1
    store i32 %Local_Var_3, i32* %Local_Var_1
```

但是优化后会出现常量指令:

```
    %Local_Var_0 = alloca i32
    store i32 1, i32* %Local_Var_0
    %Local_Var_1 = alloca i32
    %Local_Var_3 = add i32 1, 1
    store i32 %Local_Var_3, i32* %Local_Var_1
```

所以需要重新扫描所有指令并消除常量指令即可.

核心代码:

```
for (int i = 0; i < instruction.size(); ++i) {
        Instruction* inst = instruction[i];
        if (inst->getValueName().size() == 0) continue;
        if (!inst->getType().compare("alloca") ||
            !inst->getType().compare("call i32"))
            continue;
        vector<Value*> operand = inst->getOperand();
        int flg = 1;
        for (int i = 0; i < operand.size(); ++i)
            if (!operand[i]->IsNum()) {
                flg = 0;
                break;
            }
        if (!flg) continue;

        string type = inst->getType();
        Value* value;
        if (type.substr(0, 4).compare("icmp") == 0) {
            if (!type.compare("icmp ne")) {
                value = new Value(operand[0]->getNumVal() !=
                                  operand[1]->getNumVal());
            } else if (!type.compare("icmp eq")) {
                value = new Value(operand[0]->getNumVal() ==
                                  operand[1]->getNumVal());
            } else if (!type.compare("icmp slt")) {
                value = new Value(operand[0]->getNumVal() <
                                  operand[1]->getNumVal());
            } else if (!type.compare("icmp sgt")) {
                value = new Value(operand[0]->getNumVal() >
                                  operand[1]->getNumVal());
            } else if (!type.compare("icmp sle")) {
                value = new Value(operand[0]->getNumVal() <=
                                  operand[1]->getNumVal());
            } else if (!type.compare("icmp sge")) {
                value = new Value(operand[0]->getNumVal() >=
                                  operand[1]->getNumVal());
            }
            value->setIsI1();
        } else if (type.compare("zext") == 0) {
            value = new Value(operand[0]->getNumVal());
        } else if (type.compare("add") == 0 || type.compare("sub") == 0 ||
                   type.compare("mul") == 0 || type.compare("sdiv") == 0 ||
                   type.compare("srem") == 0) {
            if (!type.compare("add")) {
                value = new Value(operand[0]->getNumVal() +
                                  operand[1]->getNumVal());
            } else if (!type.compare("sub")) {
                value = new Value(operand[0]->getNumVal() -
                                  operand[1]->getNumVal());
            } else if (!type.compare("mul")) {
                value = new Value(operand[0]->getNumVal() *
                                  operand[1]->getNumVal());
            } else if (!type.compare("sidv")) {
                value = new Value(operand[0]->getNumVal() /
                                  operand[1]->getNumVal());
            } else if (!type.compare("srem")) {
                value = new Value(operand[0]->getNumVal() %
```

```
                                operand[1]->getNumVal());
            }
        } else {
            cerr << type << endl;
            assert(0);
        }
        inst->changeToConst(value);
    }
```

中间代码优化（全局优化）

1. 消除无用指令

观察

```
int x = 1;
a = x + 1;
```

优化后生成:

```
%Local_Var_0 = alloca i32
store i32 1, i32* %Local_Var_0
%Local_Var_1 = alloca i32
store i32 2, i32* %Local_Var_1
```

不难发现我们对 `x` 和 `a` 的计算其实并么有任何作用, 我们之后并没有用到它, 那么我们还保留他们做什么呢?

优化效果分析

1. 直接减少指令条数.
2. 简化结构, 十分利于产生新的优化空间, 在多遍化的设计下有更大的优化潜力.
3. 优化幅度取决于源代码中是否有大量未使用的表达式, 依赖与程序员水平.

优化原理即方式

如果一个指令不会改变内存数据或者运行流并且从来没有被**使用**过, 那么显然可以直接删除.

如果是 `store, alloca` 指令, 如果其操作的地址空间之后再也没有被访问, 同样可以直接删除.

相对的 `ret, call, br` 这些指令视为有用的, 并作为判断其他指令是否**使用**过的起点. 将这些指令加入队列中并进行 BFS : 每次把队头指令取出, 将指令的参数加入队列中, 反复操作直到队列为空.

核心代码:

```
queue<Instruction*> q;
    while (!q.empty()) q.pop();
    vector<Instruction*> uselessStore(0);

    for (BBlock* bBlock : this->bBlock) {
        bef += bBlock->instruction.size();
        for (Instruction* inst : bBlock->instruction) {
            inst->setUseless(1);
            string type = inst->getType();
```

```
            if (!type.compare("ret i32") || !type.compare("ret void") ||
                !type.compare("call i32") || !type.compare("call void") ||
                !type.compare("br")) {
                inst->setUseless(0);
                q.push(inst);
            } else if (!type.compare("store")) {
                Instruction* operand =
                    dynamic_cast<Instruction*>(inst->getOperand()[1]);
                if (operand == NULL ||
                    operand->getType().compare("alloca")) {  // 不为 alloca
                    inst->setUseless(0);
                    q.push(inst);
                } else {
                    uselessStore.push_back(inst);
                }
            }
        }
    }

    while (!q.empty()) {
        Instruction* inst = q.front();
        q.pop();
        vector<Value*> operand = inst->getOperand();
        for (int i = 0; i < operand.size(); ++i) {
            Instruction* nInst = dynamic_cast<Instruction*>(operand[i]);
            if (nInst == NULL) continue;
            if (!nInst->isUseless()) continue;
            nInst->setUseless(0), q.push(nInst);
        }
        if (inst->getType().compare("alloca")) continue;
        for (int i = 0; i < uselessStore.size(); ++i) {
            Instruction* store = uselessStore[i];
            if (store->getOperand()[1] == inst) {
                // cerr << inst->to_string() << endl;
                store->setUseless(0), q.push(store);
                swap(uselessStore[i], uselessStore[uselessStore.size() - 1]);
                uselessStore.pop_back();
                --i;
            }
        }
    }
```

2. 合并基本快

优化方式

合并基本快有两条策略:

1. 如果整个基本块只有一个无条件跳转指令(记为 `inst`), 那么将所有跳转到该基本块的指令都改为跳转到 `inst` 的目标基本块, 并删除当前基本快.
2. 如果基本快的结尾是无条件跳转, 且目标基本快只会从当前基本快进入, 可以合并两个基本块.

同时为了利于增大优化空间, 如果条件跳转指令的两个出口是同一个基本块, 那么可将其改为无条件跳转.

核心代码:

```
    for (int i = 0; i < bBlock.size(); ++i) {
        Instruction* inst = *bBlock[i]->instruction.rbegin();
        if (inst->getType().compare("br")) continue;
        vector<Value*> operand = inst->getOperand();
        if (operand.size() == 1) continue;
        if (operand[0]->IsNum()) {
            Value* label = operand[0]->getNumVal() ? operand[1] : operand[2];
            inst->clearUse();
            inst->appendOperand(label);
        } else if (operand[1] == operand[2]) {
            Value* label = operand[1];
            inst->clearUse();
            inst->appendOperand(label);
        }
    }

    for (int i = 1; i < bBlock.size(); ++i) {
        if (bBlock[i]->instruction.size() > 1) continue;
        if (bBlock[i]->instruction[0]->getType().compare("br")) continue;
        Instruction* inst = bBlock[i]->instruction[0];
        vector<Value*> operand = inst->getOperand();
        if (operand.size() > 1) continue;
        bBlock[i]->instruction.pop_back();
        Value* label = operand[0];

        for (int j = 0; j < bBlock.size(); ++j) {
            if (bBlock[j]->instruction.size() == 0) continue;
            Instruction* brInst = *bBlock[j]->instruction.rbegin();
            if (brInst->getType().compare("br")) continue;
            brInst->changeOperand(bBlock[i], label);
        }
    }

    vector<int> ideg(bBlock.size()), odeg(bBlock.size());
    ideg[0] = 1;  // start
    for (int i = 0; i < bBlock.size(); ++i) {
        if (bBlock[i]->instruction.size() == 0) continue;
        Instruction* brInst = *bBlock[i]->instruction.rbegin();
        if (brInst->getType().compare("br")) continue;
        vector<Value*> operand = brInst->getOperand();
        odeg[i] = operand.size() == 1 ? 1 : 2;
        for (int j = 0; j < bBlock.size(); ++j) {
            if (bBlock[j]->instruction.size() == 0) continue;
            if (odeg[i] == 1) {
                if (operand[0] == bBlock[j]) ++ideg[j];
            } else if (odeg[i] == 2) {
                if (operand[1] == bBlock[j]) ++ideg[j];
                if (operand[2] == bBlock[j]) ++ideg[j];
            }
        }
    }

    for (int i = 0; i < bBlock.size(); ++i) {
        if (bBlock[i]->instruction.size() == 0) continue;
        while (odeg[i] == 1) {
            Instruction* brInst = *bBlock[i]->instruction.rbegin();
            vector<Value*> operand = brInst->getOperand();
```

```
            int flg = 0;
            for (int j = 0; j < bBlock.size(); ++j) {
                if (bBlock[j]->instruction.size() == 0) continue;
                if (operand[0] != bBlock[j]) continue;
                if (ideg[j] == 1) {
                    bBlock[i]->instruction.pop_back();
                    for (Instruction* inst : bBlock[j]->instruction)
                        bBlock[i]->instruction.push_back(inst);
                    bBlock[j]->instruction.clear();
                    odeg[i] = odeg[j];
                    flg = 1;
                }
                break;
            }
            if (!flg) break;
        }
    }
```

3. 删除不可达基本快

核心代码:

```
    vector<int> vis(bBlock.size());
    queue<int> q;
    vis[0] = 1;
    while (!q.empty()) q.pop();
    q.push(0);
    while (!q.empty()) {
        int i = q.front();
        q.pop();
        Instruction* brInst = *bBlock[i]->instruction.rbegin();
        if (brInst->getType().compare("br")) continue;
        vector<Value*> operand = brInst->getOperand();

        for (int j = 0; j < bBlock.size(); ++j) {
            if (bBlock[j]->instruction.size() == 0 || vis[j]) continue;
            if (odeg[i] == 1) {
                if (operand[0] == bBlock[j]) vis[j] = 1, q.push(j);
            } else if (odeg[i] == 2) {
                if (operand[1] == bBlock[j] || operand[2] == bBlock[j])
                    vis[j] = 1, q.push(j);
            }
        }
    }
```

目标代码优化

消除基本快末尾多余跳转语句

由于 llvm-ir 规定每个基本快最后一条指令必须是跳转指令或者 `ret` 指令, 所以在生成 mips 的时候可能产生冗余的跳转指令, 去掉即可.

监控临时寄存器的活动周期

在消除公共子表达式的过程中, 编译器收获了不小的时间上的优化, 但无意间破坏了原本代码的一些优秀性质.

比如"除了 alloca 指令, 所有指令产生的临时寄存器中的值只会在本基本块中使用"这一性质, 其实优化前有对应的更强的形式:"除了 alloca 指令, 所有指令产生的临时寄存器中的值只会在本基本块中使用**至多一次**".

基于优化前的强性质, 分配寄存器将变得十分简单, 指令的值在使用一次后就不需要考虑再写回内存.

不过幸运的是, 强性质在不可避免地被破坏后, 我们仍然可以根据弱性质分配寄存器:, 只需要对于非 alloca 指令跟踪其在基本块中最后一次使用即可. 在最后一次使用后我们就可以丢弃指令的值了.