

Lab4_report

PB21111686_赵卓

实验目的

- 掌握`Mem2Reg`的优化思路。
- 了解中间代码优化的原理。
- 提升代码实践能力。

实验步骤

- 步骤零：找到每个函数中，每个`bb`块的控制边界。

`bb`块的控制边界是指该`bb`块的后续`bb`块中，有多个前置`bb`块的`bb`块。因此对于此类`bb`块，控制流来自不同的`bb`块，因此需要在这些`bb`块的开头插入`phi`指令，实现变量的正确赋值。我们对每个函数遍历所有`bb`块，对于前置`bb`块数量大于1的`bb`块，将其加入每个前置`bb`块的控制边界即可。实现代码如下：

```
//步骤零：找到每个函数中，每个bb块的控制边界。
void Dominators::create_dominance_frontier(Function *f)
{
    for (auto &bb1 : f->get_basic_blocks())
    {
        auto bb = &bb1;
        auto &pre_bb_list = bb->get_pre_basic_blocks();
        if (pre_bb_list.size() > 1)
        {
            for (auto &pre_bb : pre_bb_list)
            {
                if (dom_frontier_.find(pre_bb) == dom_frontier_.end())
                {
                    std::set<BasicBlock *> pre_bb_set;
                    dom_frontier_.insert(std::make_pair(pre_bb, pre_bb_set));
                }
                dom_frontier_[pre_bb].insert(bb);
            }
        }
    }
}
```

- 步骤一：找到活跃在多个`bb`块的全局名字集合，以及它们所属的`bb`块。

“全局名字”的意思并非是指全局变量，而是在某个函数中出现在多个`bb`块的`alloca`指令。对于这些`alloca`值，在不同`bb`块中取不同的值，需要设置栈存放值，以及可能需要`phi`函数。我们对每个函数遍历所有的`bb`块，如果是`alloca`指令，则将其加入全局名字集合，并为改名字设置所属`bb`块集合，如果后续`bb`块中出现了该名字，则将该`bb`块加入其所属`bb`块集合。实现代码如下：

```

// 步骤一：找到活跃在多个 bb 的全局名字集合，以及它们所属的 bb 块
std::set<Value *> global_live_name;
std::map<Value *, std::set<BasicBlock *>> global_live_name_bb;
for (auto &bb1 : func_>get_basic_blocks())
{
    auto bb = &bb1;
    for (auto &instr1 : bb->get_instructions())
    {
        auto instr = &instr1;
        if (instr->is_store())
        {
            auto store_rval = instr->get_operand(1);
            if (!IS_GLOBAL_VARIABLE(store_rval) && !IS_GEP_INSTR(store_rval))
            {
                if (global_live_name.find(store_rval) == global_live_name.end())
                {
                    global_live_name.insert(store_rval);
                    std::set<BasicBlock *> store_rval_set = {};
                    global_live_name_bb.insert(std::make_pair(store_rval, store_rval_set));
                }
                if (global_live_name_bb.find(store_rval) != global_live_name_bb.end())
                {
                    if (global_live_name_bb[store_rval].find(bb) == global_live_name_bb[store_rval].end())
                    {
                        global_live_name_bb[store_rval].insert(bb);
                    }
                }
            }
        }
    }
}
}

```

- 步骤二：通过支配边界信息，在对应位置插入`phi`指令。
 对于每个函数，遍历活跃在多个`bb`块的全局名字集合的所属`bb`块，在其支配边界的开头插入对应全局名字的`phi`指令。同时注意，插入`phi`指令后，该支配边界也已成为的全局名字所属`bb`块，因此对其进行同样的操作。实现代码如下：

```
// 步骤二：通过支配边界信息，在对应位置插入phi指令
std::map<std::pair<BasicBlock *, Value *>, bool> bb_has_val_phi;
std::vector<BasicBlock *> work_list;
for (auto val : global_live_name)
{
    work_list.assign(global_live_name_bb[val].begin(), global_live_name_bb[val].end());
    int len = work_list.size();
    for (int i = 0; i < len; i++)
    {
        auto bb = work_list[i];
        if (dominators->get_dominance_frontier(bb).size() != 0)
        {
            for (auto bb_dominance_frontier : dominators->get_dominance_frontier(bb))
            {
                if (bb_has_val_phi.find(std::make_pair(bb_dominance_frontier, val)) == bb_has_val_phi.end())
                {
                    std::vector<Value *> vals;
                    std::vector<BasicBlock *> vals_bbs;
                    auto phi1 = PhiInst::create_phi(val->get_type()->get_pointer_element_type(), bb_dominance_frontier, vals, vals_bbs);
                    auto phi = static_cast<Instruction *>(phi1);
                    bb_dominance_frontier->add_instr_begin(phi);
                    bb_has_val_phi.insert(std::make_pair(std::make_pair(bb_dominance_frontier, val), true));
                    bool if_bb_dominance_frontier = true;
                    for (int j = 0; j < len; j++)
                    {
                        if (work_list[j] == bb_dominance_frontier)
                        {
                            if_bb_dominance_frontier = false;
                        }
                    }
                    if (if_bb_dominance_frontier)
                    {
                        work_list.push_back(bb_dominance_frontier);
                        len++;
                    }
                    phi_val_map.insert(std::make_pair(phi, val));
                }
            }
        }
    }
    work_list.clear();
}
}
```

- 步骤三：判断是否需要回溯。

由于对bb块递归进行重命名，可能出现某bb块中全局名字值是来自其他bb块，但该bb块还没有进行重命名或者其栈值已经被弹出。因此我们记录下每个bb块退出重命名时的栈情况，对于当前bb块的load指令，如果其对应栈为空，那么则遍历前置bb块的栈，如果找不到load指令对应的值，则需要回溯，先对其他bb块进行重命名。实现代码如下：

```
// 步骤三：判断是否需要回溯
for (auto &instr1 : bb->get_instructions())
{
    auto instr = &instr1;
    if (instr->is_load())
    {
        auto load_val = static_cast<LoadInst *>(instr)->get_operand(0);
        if (!IS_GLOBAL_VARIABLE(load_val) && !IS_GEP_INSTR(load_val))
        {
            if (val_stack[load_val].empty())
            {
                if (value_if_define.find(load_val) == value_if_define.end() && get_undefine(bb, load_val) == nullptr)
                {
                    return;
                }
            }
        }
        if (instr->is_store())
        {
            auto store_rval = static_cast<StoreInst *>(instr)->get_operand(1);
            if (!IS_GLOBAL_VARIABLE(store_rval) && !IS_GEP_INSTR(store_rval))
            {
                value_if_define.insert(store_rval);
            }
        }
    }
}
```

- 步骤四：用phi指令值替换进入栈图中对应的val栈。
将phi指令值push入对应val值的栈即可。实现代码如下：

```

// 步骤四：用phi指令值替换进入栈图中对应的val栈
for (auto &instr1 : bb->get_instructions())
{
    auto instr = &instr1;
    if (instr->is_phi())
    {
        if (phi_val_map.find(instr) != phi_val_map.end())
        {
            auto phi_val = phi_val_map[instr];
            val_stack[phi_val].push_back(instr);
        }
    }
}

```

- 步骤五：将load的值替换，若不存在则从前置bb块中寻找。
将所有load指令值替换成对应栈顶值，如果栈空，那么从前置bb块的栈中寻找。实现代码如下：

```

// 步骤五：将load的值替换，若不存在则从前置bb块中寻找
if (instr->is_load())
{
    auto load_val = static_cast<LoadInst *>(instr)->get_operand(0);
    if (!IS_GLOBAL_VARIABLE(load_val) && !IS_GEP_INSTR(load_val))
    {
        if (val_stack[load_val].empty())
        {
            instr->replace_all_use_with(get_indefine(bb, load_val));
        }
        else
        {
            if (val_stack.find(load_val) != val_stack.end())
            {
                instr->replace_all_use_with(val_stack[load_val].back());
            }
        }
    }
}

```

- 步骤六：将store值入栈，同时删除store指令。
将store指令值push入对应val值的栈即可，实现代码如下：

```

// 步骤六：将store值入栈，同时删除store指令
if (instr->is_store())
{
    auto store_lval = static_cast<StoreInst *>(instr)->get_operand(0);
    auto store_rval = static_cast<StoreInst *>(instr)->get_operand(1);
    if (!IS_GLOBAL_VARIABLE(store_rval) && !IS_GEP_INSTR(store_rval))
    {
        if (val_stack.find(store_rval) == val_stack.end())
        {
            std::vector<Value *> store_rval_value;
            val_stack.insert(std::make_pair(store_rval, store_rval_value));
        }
        val_stack[store_rval].push_back(store_lval);
        wait_delete.push_back(instr);
    }
}

```


- 步骤七：补全后续**bb**块的 ϕ 指令。

对当前**bb**块重命名结束之后，再遍历后续**bb**块，如果存在 ϕ 指令，说明后续**bb**块对应 ϕ 指令值的一个来源是当前**bb**块，在步骤二中已经设定。因此将对应 ϕ 指令值的栈顶值插入 ϕ 指令即可。实现代码如下：

```
// 步骤七：补全后续bb的phi指令
for (auto succ_bb : bb->get_succ_basic_blocks())
{
    for (auto &instr1 : succ_bb->get_instructions())
    {
        auto instr = &instr1;
        if (instr->is_phi())
        {
            auto phi_instr = static_cast<PhiInst *>(instr);
            if (phi_val_map.find(instr) != phi_val_map.end())
            {
                auto phi_val = phi_val_map[instr];
                if (val_stack.find(phi_val) != val_stack.end() && !val_stack[phi_val].empty())
                {
                    phi_instr->add_phi_pair_operand(val_stack[phi_val].back(), bb);
                }
            }
        }
    }
}
```

- 步骤八：对后续**bb**块递归进行重命名。

每次重命名完**bb**块后，记录其命名情况，已经命名或者没有命名。对于后续没有命名的**bb**块依次进行重命名。实现代码如下：

```
// 步骤八：对后续bb块递归进行重命名。
bb_if_rename.insert(std::make_pair(bb, true));

val_stack_record.insert(std::make_pair(bb, val_stack));

for (auto succ_bbs : bb->get_succ_basic_blocks())
{
    if (bb_if_rename.find(succ_bbs) == bb_if_rename.end())
    {
        rename(succ_bbs);
    }
}
```

- 步骤九：结束当前**bb**块重命名，弹出栈值。

结束当前**bb**块重命名后，由于要回到控制流到达该**bb**块之前的情况，因此遍历**bb**块的指令，将其对所有 ϕ 指令的栈的修改还原，弹出 ϕ 指令入的栈值。实现代码如下：

```
// 步骤九：结束当前bb块重命名，弹出栈值。
for (auto &instr1 : bb->get_instructions())
{
    auto instr = &instr1;
    if (instr->is_store())
    {
        auto store_rval = static_cast<StoreInst *>(instr)->get_operand(1);
        if (!IS_GLOBAL_VARIABLE(store_rval) && !IS_GEP_INSTR(store_rval))
        {
            if (val_stack.find(store_rval) != val_stack.end() && !val_stack[store_rval].empty())
                val_stack[store_rval].pop_back();
        }
    }
    else if (instr->is_phi())
    {
        if (phi_val_map.find(instr) != phi_val_map.end())
        {
            auto phi_val = phi_val_map[instr];
            if (val_stack.find(phi_val) != val_stack.end() && !val_stack[phi_val].empty())
                val_stack[phi_val].pop_back();
        }
    }
}
}
```

- 步骤十：删除store指令

删除当前bb块的store指令，Deadcode会将对应的load和alloca死代码删除。实现代码如下：

```
// 步骤十：删除store指令
for (auto &instr : wait_delete)
{
    bb->erase_instr(instr);
}
}
```

- 步骤十一：补全对phi指令的后端翻译

后端应该可以将phi指令翻译成对应汇编语言。若在遍历到phi指令时直接翻译，困难较大，因为寻找前置bb块难度较大，而且此时对应val值也可能改变。因此我们变换思路，在个bb块翻译结束后，对后继bb块遍历，如果存在phi指令，那么说明后续bb块的phi指令值来自当前bb块，因此将后续bb块所需val值赋给其phi指令值。实现代码如下：

//步骤十一：补全对phi指令的后端翻译

```
void CodeGen::gen_phi()
{
    auto bb = context.inst->get_parent();
    auto &bb_succ_list = bb->get_succ_basic_blocks();
    for (auto bb_next : bb_succ_list)
    {
        for (auto &instr1 : bb_next->get_instructions())
        {
            auto instr = &instr1;
            if (instr->is_phi())
            {
                int num = instr->get_num_operand();
                for (int i = 0; i + 1 < num; i = i + 2)
                {
                    if (instr->get_operand(i + 1) == bb)
                    {
                        if (instr->get_operand(i)->get_type()->is_float_type())
                        {
                            load_to_freg(instr->get_operand(i), FReg::ft(0));
                            store_from_freg(instr, FReg::ft(0));
                        }
                        else
                        {
                            load_to_greg(instr->get_operand(i), Reg::t(0));
                            store_from_greg(instr, Reg::t(0));
                        }
                    }
                }
            }
        }
    }
}
```

正确性验证

- 测试结果如下：

当前作业

Lab4

历史作业

第五次编译原理作业

第四次编译原理作业

Lab3-阶段一

Lab3-阶段二

第三次编译原理作业

Lab2-阶段一

Lab2-阶段三

Lab2-阶段二

第二次编译原理作业

Lab1

第一次编译原理作业

Lab0预热实验

Lab4

作业时间: 2023-11-27 14:47:00 至 2023-12-19 00:00:00

作业满分: 200.00, 共 2道 题

本次实验需要完成

1. 补全 `src/passes/Dominators.cpp` 文件, 使编译器能够进行正确的支配树分析
2. 补全 `src/passes/Mem2Reg.cpp` 文件, 使编译器能够正确执行 Mem2Reg
3. 将 phi 指令转化为 copy statement, 令后端可以正确处理 phi 指令

并在希冀平台提交实验仓库的 URL 与实验报告 (实现方法、正确性验证、性能验证等)

文件上传题

#	题目	点击题目标题, 进入答题	分值	提交状态
1.	2023-lab4-报告提交		100.00	未提交文件

通用评测题

#	题目	点击题目标题, 进入答题	分值	批阅信息
1.	2023-lab4-代码提交		100.00	下载源文件

得分: 70.00 最后一次提交时间: 2023-12-03 00:38:31

点击查看/隐藏评测结果

Accept

35/35个通过测试用例

状态: Accept

codegen-0-io.cminus	Accept	2
codegen-1-return.cminus	Accept	2
codegen-2-calculate.cminus	Accept	2
codegen-3-output.cminus	Accept	2
codegen-4-if.cminus	Accept	2
codegen-5-while.cminus	Accept	2
codegen-6-array.cminus	Accept	2
codegen-7-function.cminus	Accept	2
codegen-8-store.cminus	Accept	2
codegen-9-fibonacci.cminus	Accept	2
codegen-10-float.cminus	Accept	2
codegen-11-floatcall.cminus	Accept	2
codegen-12-global.cminus	Accept	2
codegen-13-complex.cminus	Accept	2
general-1-return.cminus	Accept	2
general-2-decl_int.cminus	Accept	2
general-3-decl_float.cminus	Accept	2
general-4-decl_int_array.cminus	Accept	2
general-5-decl_float_array.cminus	Accept	2
general-6-num_add_int.cminus	Accept	2
general-7-assign_int_var_local.cminus	Accept	2
general-8-assign_int_array_local.cminus	Accept	2
general-9-assign_cast.cminus	Accept	2
general-10-funcall.cminus	Accept	2
general-11-funcall_chain.cminus	Accept	2
general-12-funcall_recursion.cminus	Accept	2
general-13-if_stmt.cminus	Accept	2
general-14-while_stmt.cminus	Accept	2
general-15-if_while.cminus	Accept	2
general-16-if_chain.cminus	Accept	2
general-17-while_chain.cminus	Accept	2
general-18-global_var.cminus	Accept	2
general-19-global_local_var.cminus	Accept	2
general-20-gcd_array.cminus	Accept	2
general-21-comment.cminus	Accept	2

性能优化验证

- 优化结果如下:


```

● bingyu@bingyu:~/2023ustc-jianmu-compiler/tests/4-mem2reg$ ./test_perf.sh
[info] Start testing, using testcase dir: ./performance-cases
=====./performance-cases/const-prop.cminus=====
=====mem2reg off

real    0m14.977s
user    0m14.649s
sys     0m0.068s
=====mem2reg on

real    0m12.273s
user    0m12.099s
sys     0m0.016s
=====./performance-cases/loop.cminus=====
=====mem2reg off

real    0m8.683s
user    0m8.605s
sys     0m0.020s
=====mem2reg on

real    0m6.981s
user    0m6.952s
sys     0m0.008s
=====./performance-cases/transpose.cminus=====
=====mem2reg off

real    0m17.676s
user    0m17.417s
sys     0m0.032s
=====mem2reg on

real    0m12.580s
user    0m12.453s
sys     0m0.016s

```

可见`Mem2Reg`会进行有效的优化。

实验总结

- 本次实验在前置实验的基础上，进行来中间代码的优化，并采用`Mem2Reg`方法。由优化结果可以看出，`Mem2Reg`方法进行了较为有效的优化，但优化程度存在提升空间，思考如何进行进一步的优化是有意义的。