编译原理 PA4 实验报告

郑逢时 2016012177

2019年12月28日

概要
完成过程
- 更新框架
- 阅读文档及熟悉框架
- 求解活跃变量数据流
- 逐语句消除死代码

遇到困难及解决方案
- 几乎所有测试点都优化出错
- 合并PA4进入完整框架时的问题

性能测试结果
- 测试结果
- 结论

1. 概要

本次实验为编译优化之死代码消除。我通过学习编译原理Lecture12一讲的编译优化相关知识，认真阅读框架文档和已有的其他优化实现，实现了rust版本的死代码消除。并比较开启/关闭死代码消除优化和其他几个优化所得到的tac程序性能，验证和讨论优化的有效性。

2. 完成过程

特别感谢李晨昊助教提供的文档和框架原有的三种优化实现，它们对我完成本次PA帮助巨大。

2.1. 更新框架

rust框架近期有几次更新，我开动PA4前先pull了整个框架的最新版本，但由于与PA1-3冲突太多，我先暂时新开了一个PA4分支来pull框架。由于本次PA与新特性无关，所以我打算直接在这个独立的PA4分支上完成本次PA，暂时不支持PA1-3的新特性。

2.2. 阅读文档及熟悉框架

本次PA需要对TAC程序做数据流分析，需要的数据结构支持和算法支持较多，实现离不开框架提供的Flow、BB 等数据结构和算法模版。因此我花了半天左右时间认真阅读了文档和框架已有的一些实现。

2.3. 求解活跃变量数据流

根据课上所学知识及实验文档，我们要先以基本块为单位，求解活跃变量数据流方程：

$$LiveIn(B) = use_B \cup (LiveOut(B) - def_B)$$
$$LiveOut(B) = \cup_{S \in sub[B]} LiveIn(S)$$

按照文档建议，use_B 和 def_B 在求解方程前算出来，而且 def_B 就用所有定值变量的并集就可以了，这个算法很容易想出来，只需O(n)时间，n是基本块B的指令数：从前向后扫描所有语句，每扫一句就对语句中的每个引用的变量，判断 def_B 中有没有它，如果没有就加入 use_B，然后再对把语句的写入变量（如果有的话）加入 def_B 集合。

然后试图使用框架提供的数据流分析模板算法 `Flow::<flow::Or>` 求解数据流方程（`flow::Or` 是因为活跃变量数据流方程的运算符是并集）。由于模板是默认前向流的，所以在用给求解器传参的时候要注意反过来，in接liveout，out接livein，传入有向图的时候也要注意用基本块的 next 边，而不是 previous。livein的初始值定为 use_B，而 liveout 初始值为空集。

2.4. 逐语句消除死代码

求解了每个基本块的出入活跃变量后，就可以确定每条语句的活跃变量了，为了高效，我的实现是从后往前扫描，滚动计算，并逐语句消除死代码。

活跃变量集合传递函数：

$$LiveOut(S_{i-1}) = (LiveOut(S_i) - def_{S_i}) \cup use_{S_i}$$

然后判断：

- 如果当前指令形如 `a = Call XXX`，并且 $a \notin LiveOut(S)$，说明a不活跃，应消除，同时确保副作用还在，函数调用应保留，即优化为 `Call XXX`；
- 如果当前指令形如 `a = b, a = (b+c), a = *(b+1), a = -b` 等，并且 $a \notin LiveOut(S)$，说明a不活跃，而且没副作用，可以删除这条指令。（框架的实现保证一边遍历一边删除是安全的）

注意由于我们的语义约定，应该在删除死代码后再更新 liveout ，确保不会误删。

3. 遇到困难及解决方案

3.1. 几乎所有测试点都优化出错

原因是忘记了基本块的最后一句指令（Jif/Jump/Ret之一），它在框架里比较特殊，并没有存储在链表里，而是在一个单独的域。

在计算初始 def_B, use_B 时，应该在扫描完后对这个域做计算，而在滚动删除死代码的时候，则应在扫描前处理这个域。

如此修改后能通过所有测例。

3.2. 合并PA4进入完整框架时的问题

在了解到PA评分是以仓库里master版本为准的，我最后将PA4的成果合并进PA3提交时的master框架，处理各种冲突后，测试了一通 PA3 的测例（用带编译优化的模式测试），但遇到诸多莫名其妙的bug。

仔细分析，发现大部分是PA3的实现细节尚有小问题，主要是生成函数entry和匿名函数时，忘记指定或错误指定了 param_num 和 reg_num，导致代码优化阶段出现数组越界的运行错误。魔改一通修复后，能通过除了 lambda-2 之外的所有测例。

又魔改了一通，还是无法解决问题，怀疑不是我实现的死代码消除带来的问题，屏蔽后依旧出错。关掉剩下三个优化后，问题消失！说明rust原框架的优化函数不能兼容新特性。这个issue如下：

如果前面定义了某数组x，在某lambda表达式中修改这个数组，在调用完lambda后再次引用x，则这个x会被“公共变量提取”优化给优化成调用lambda前的旧值。

```
var x = new int[10];
Print(x[1],"\n");  // 引用 x[1]，应该输出 0
var addx = fun() { // 捕获数组 x
  x[1] = 1;
};
addx();
Print(x[1],"\n");  // 再次引用 x[1]，应该输出 1，但实际输出 0
```

折磨了好长时间后，最后发现是因为rust框架在Call函数时有一个 CallHint 的参数，可以指明调用函数时会不会对数组或对象造成“副作用”，如果指定有副作用，则优化会保守一点，会认为函数可能修改调用前后的某个变量，从而防止优化出错。否则会优化得比较激进，忽略掉副作用导致出错。

修改了 CallHint 后，bug解决。

4. 性能测试结果

4.1. 测试结果

主要对S4的 mandelbrot, rbtree, sort 三个测例做性能测试，分别用*不开优化*、*只开原有三种优化*（公共表达式提取、复写传播、常量传播）、*只开死代码消除优化*、和*开启所有优化*的四种编译器编译并运行，输出结果均正确，测得TacVM执行的指令条数统计如下：（这个测试是在我合并PA4到master之前做的，没有加入新特性，性能更好，所以可能跟master最新版本的测试结果有出入，特此说明）

测例 \ 指令条数	1.不开优化	2.只开原有三种优化	3.只开死代码消除优化	4.开启所有优化
mandelbrot	3664076	4605304	3664076	**3088148**
rbtree	2299376	2583042	2257973	**2030896**
sort	489113	608105	489113	**413865**

这个结果还是比较有意思的，甚至有些出乎意料。

- 首先比较第一列和第二列，我们竟然得出，单独开启公共表达式提取、复写传播、常量传播优化，不仅没有产生优化，甚至恶化了性能，猜测可能是优化过程中新引入了更多寄存器（例如公共表达式提取等），或者更多语句（tac程序确实比以前更长）导致的。
- 比较第一列和第三列，可以得出，只开死代码消除能带来优化，但十分不显著（只对rbtree有少量优化），事实上从tac程序也可以看出，这几个测例本身源码几乎没有出现过冗余代码，几乎无需消除。
- 真正有意义的是第一列和第四列的比较，很明显，四种优化全开，显著提升了性能，而且说明我的死代码消除对于代码优化是至关重要的一步。这是容易理解的，因为前三种优化的过程中提供了很多死代码优化的空间，充分利用效果斐然。比如一个简单例子：

  ```
  a = 1
  b = a
  c = b
  Param c
  Call _PrintInt
  ```

  经过常量传播、复写传播优化后：

  ```
  a = 1
  b = 1
  c = 1
  Param c
  Call _PrintInt
  ```

  显然，这时候定值完后的c是活跃变量，而a、b在定值后变成不活跃的了，经过死代码优化得到的代码大大简化：

  ```
  c = 1
  Param c
  Call _PrintInt
  ```

4.2. 结论

经过以上数据与分析，可以得出，我实现的死代码消除算法是有效的，单独使用时，能得到性能不差于优化前的代码。同时我们也看到，死代码消除最好与其他几种优化方案共同使用，相互配合，才能发挥出优化算法的最大能力，而且四种优化算法在其中都扮演了举足轻重的角色，都是有效的。