Decaf PA4 实验报告

计 76　张翔　2017011568

2019 年 12 月 19 日

1　实验简述

本阶段实验将 PA3 生成 TAC 中间代码进行基于数据流分析的优化，从而获得一定的性能提升。

2　主要工作

- 修改了框架中 Control Flow Graph 的实现，构建时自动移除不可达的基本块
- 实现了基于数据流分析的公共表达式提取、常量传播（包含算术恒等式优化）、复写传播以及死代码消除
- 实现了一些简单的窥孔优化，如移除返回值未被使用的 Alloc 和类构造函数等

3　死代码移除

死代码移除指移除执行后不会产生实际作用的代码，这里使用活跃变量分析的方法来实现。活跃变量的计算依赖于后向流，对于基本块来说，有如下数据流方程

$$\begin{aligned} LiveIn(B) &= LiveUse(B) \cup (LiveOut(B) - Def(B)) \\ LiveOut(B) &= \bigcup_{S \in Succ(B)} LiveIn(S) \end{aligned} \tag{1}$$

这个数据流方程的解法使用课上介绍的迭代算法即可，解之前需要遍历所有基本块以得到 Def 以及 $LiveUse$ 集合。这里的 $LiveUse$ 集合是 B 中定值前被引用的变量集合，而 Def 集合是基本块 B 中被定值的所有变量的集合，显然它是课上介绍的 Def 集合的超集，其多出的元素是 B 中被引用后又被定值的变量，显然它们也在 $LiveUse$ 中，因此不影响数据流方程的计算。对于每个基本块，正序遍历所有的指令，将定值变量加入 Def，被引用变量若不在 Def 中，加入 $LiveUse$ 即可。框架中的 `LivenessAnalyzer` 实现了后向流方程的求解，可以直接调用。

解出数据流方程后，利用基本块的 $LiveOut$ 集合可以计算出每个指令语句的 $LiveOut$ 集合，这里的计算需要从基本块中反向遍历指令，将当前 $LiveOut$ 附加到当前指令上，从中移除在当前指令定值的变量，并加入当前指令引用的变量，然后传递给前一条指令即可。

之后可以进行死代码的消除。从比较粗略的角度来说，Tac 中“没有既没有副作用，同时也没有赋值目标的语句”，因为 Tac 中可以没有赋值目标的指令只有分支跳转、函数调用、传参，通常分支和函数调用均被认为有副作用。对于有赋值目标、非函数调用的指令，

如果赋值目标不在当前指令的 $LiveOut$ 中，则可以直接移除；如果是函数调用，则应将返回值赋值部分优化掉，如 `_T1 = call _T2` 变为 `call _T2`。

值得一提的是，虽然函数调用会改变栈帧等，具有副作用，但从程序的 context 来说，某些函数调用前后并没有产生实际影响，如返回值未被使用的类构造函数、Alloc 函数实际上是没有作用的，但仅依赖 Tac 源码是无法获取这些信息的。因此实现时在 `TacInstr` 中加入了一个 `hint` 成员变量，用于指示该 Tac 指令的类型，如 `CONSTRUCTOR` 或是 `ALLOC`，这些信息在 PA3 的 Tac 代码生成时可以获取到。基于数据流分析的死代码移除结束后，可能会残留这类函数调用，之后进行一遍窥孔优化就可以将它们删除。另外，注意到 Tac 语言的虚拟寄存器与实际汇编的区别，可以发现 Tac 中 Callee 使用虚拟寄存器并不会影响 Caller 的同名寄存器，而汇编中 Caller 调用 Callee 前显然需要保存一些 caller-saved 寄存器从而保证调用约定的实现，因此一个没有 `store` 指令，也不调用 `print, alloc` 等，而仅仅在虚拟寄存器上进行各种操作的函数，如果其返回值不被使用，它也是没有实际副作用的，这种函数调用不会对 context 产生影响，在更为 aggressive 的优化中同样可以被移除（这里没有实现）。

另外，PA3 范式化的处理会生成一些不可达代码，如

```
    if (_T2 == 0) branch _L9
    return _T0
    branch _L10
_L9:
    return _T1
_L10:
    ...
```

显然 `branch _L10` 这里产生了一个不可达的基本块。可以通过对 CFG 从入口节点进行一遍 BFS，剩下不可达的节点全部移除即可。因为被删除的代码在控制流中本身是不可达的，这个优化只有减小生成代码体积的作用。

3.1 遇到的问题

基于活跃变量分析的死代码移除本身是非常容易实现的，但如果仅实现这个功能，在 PA4 官方测例上测试，所有样例的执行指令条数在优化前后保持不变。经过简单分析可知，除非在 decaf 中直接写一些在常用编译器中均会产生 warning 的 unused 变量，否则死代码移除优化是不生效的，因为只要保证定义的变量有被使用即可不被优化掉。显然 PA4 的实际情况与实验指导书中的描述是不一致的，不实现“公共表达式提取、常量传播、复写传播”无法进一步优化，因此后面又实现了这三种优化方法。

4 公共表达式提取

与死代码移除不同，该优化方法依赖于正向数据流，方程如下

$$\begin{aligned} Out(B) &= Gen(B) \cup (In(B) - Kill(B)) \\ In(B) &= \bigcap_{S \in Pred(B)} Out(S) \end{aligned} \tag{2}$$

这里使用集合交运算是因为某个表达式需要在基本块所有前驱均存在时才可能成为公共表达式。Gen 是基本块中产生的表达式集合，$Kill$ 是基本块中杀死的表达式集合，该数据流方程和前面的反向流求法几乎一致，只需要作替换 $LiveIn \leftrightarrow Out$，$LiveOut \leftrightarrow In$，$LiveUse \leftrightarrow Gen$，$Def \leftrightarrow Kill$，并把并集运算改为交集，执行迭代算法即可。

实现时，Tac 指令 `LoadVtbl, LoadStrConst, Binary, Unary` 可以安全地作为公共表达式的候选，而 `Load` 指令在不同时刻从内存同一区域加载的内容可能是不同的（当对应内存是虚表或字符串常量时对应指令的右值才可以作为公共表达式）。因此利用上述提到的新增 `hint` 成员变量，在生成 Tac 代码时如果 `Load` 指令是从只读的虚表/字符串区域加载，则打上相应标记，表示其右值作为公共表达式是安全的。之后可以遍历每个基本块内的指令序列，从而获取基本块的 Gen 和 $Kill$ 集合。为了方便，实现时 $Kill$ 集合中记录的是被重新定值的虚拟寄存器，Gen 中记录的是右值表达式，如果表达式引用了 $Kill$ 中的寄存器，它将被杀死。遍历某指令时，将它写入的寄存器放入 $Kill$ 中，并杀死 Gen 中引用它的表达式，如果该语句有安全的右值表达式，就把它放入 Gen 中。这里注意下述情况需要特殊处理：

```
x = x + y
x = -x
```

这里的 `x + y` 与`-x` 均不能放入 Gen，因为出现了自我修改的情况。

之后求解数据流方程，得到基本块的 Out 和 In 集合。注意此处 In 集合是前驱块 Out 的交集，需要合理设置初始值才能在迭代后得到正确的结果。对于入口块，它没有前驱，In 集合保持为空；对于其他基本块，In 设置为全集 U，实际程序实现取 $U = \bigcup_{\forall B} Gen(B)$ 是比较合理的，可以保证不会遗漏可能的公共表达式。然后可以通过正向遍历基本块内指令语句得到每条指令的 In 集合，它表示到该指令时所有可用的公共表达式。

之后可以进行公共表达式的消除。按顺序遍历基本块中的指令，若其右值表达式在指令的 In 集合中，就从该指令前一条开始对 CFG 进行逆向 DFS，找到距离最近的含有该右值表达式的指令，然后将它们合并。一个典型的操作方法如下

```
_L1:
    _T1 = _Ta + _Tb
    ...
    branch _Lk
_L2:
    _T2 = _Ta + _Tb
    ...
    branch _Lk
_L3:
    _T3 = _Ta + _Tb
    ...
    branch _Lk
...
_Lk:
    _Tk = _Ta + _Tb
```

这里基本块 `_Lk` 有一系列前驱节点 `_L1, _L2, ...`（不一定是直接前驱，`branch _Lk` 仅代表后继节点为 `_Lk` 的含义），且前驱节点中 `_Ta + _Tb` 之后的指令中没有杀死该表达

式的定值语句（DFS 时距离最近所保证的），则可以引入一个未被使用的寄存器 _Lm，原来的代码替换为

```
_L1:
    _Tm = _Ta + _Tb
    _T1 = _Tm
    ...
    branch _Lk
_L2:
    _Tm = _Ta + _Tb
    _T2 = _Tm
    ...
    branch _Lk
_L3:
    _Tm = _Ta + _Tb
    _T3 = _Tm
    ...
    branch _Lk
...
_Lk:
    _Tk = _Tm
```

前驱块中对应语句的目标寄存器修改为 _Tm，并增加一条复写语句，而 _Lk 中原来的表达式计算可以被替换为复写语句。可以看到，公共表达式消除引入了额外的复写语句，反而使得指令数量增加，需要配合复写传递进一步优化。

5 复写传递

与公共表达式提取类似，也是基于正向数据流，且数据流方程完全一致，只不过相应集合中的元素发生变化了，这里变为复写对 $< a, b >$，表示 $a = b$ 即 a 可以被 b 复写。通过遍历基本块内指令，可以计算基本块的 Gen 和 $Kill$ 集合，其中 $Kill$ 集合仍然加入每条指令定值的寄存器（若存在），且 $\forall < a, b > \in Gen, a == t \vee b == t$，若 t 刚被放入 $Kill$ 集合，则从 Gen 中移除 $< a, b >$，如果当前指令是复写语句，就向 Gen 集合加入新的复写对。解出数据流方程之后正向遍历一次所有指令得到每条指令的 In 集合，它表示该指令处可用的复写对集合，之后就可以把每条指令引用的寄存器尽可能地换成最远可达的寄存器，这里的“最远”是指将 In 集合中每个元素视作图的一条有向边，图顶点是寄存器，如果存在可用复写，源寄存器将处于图的一条链上，取该链可达的最远顶点作为最终寄存器，并将源寄存器替换即可。操作完成后可能会出现一些不活跃的寄存器，可以被后续死代码移除部分优化。

6 常量传播

为了简化，只实现了整数的常量传播。很显然，常量传播也需要借助正向数据流，相应集合实际上是寄存器值列表，每个寄存器可以取三种值 CONST，UNDEF，NAC，分别代表常

量，未定义，变量，且为常量时还会保存相应的值。另外，与前面的正向数据流方程不同，这里的并集、差集、交集等运算均与传统的集合运算不同，定义如下：

$$A \cap B \coloneqq \{A_i \wedge B_i | i = 0, , 1 \ldots, len(A) - 1\} \tag{3}$$

且

$$A_i \wedge B_i \coloneqq \begin{cases} \text{Const}, val(A_i \wedge B_i) = val(A_i) & A_i = B_i = \text{Const}, val(A_i) = val(B_i) \\ B_i & A_i = \text{Undef} \\ A_i & B_i = \text{Undef} \\ \text{Nac} & \text{Otherwise} \end{cases} \tag{4}$$

A 与 B 都是寄存器值列表，A_i 和 B_i 代表同一寄存器在两个列表中的不同取值，上述的合并操作和是符合直观感觉的，比如其中一个寄存器值未定义时，合并结果依赖于另一个寄存器的值。

另外，迭代算法也需要加以改动，每轮迭代时，In 集合按照上述定义的交集运算计算即可，但这里基本块级别的 Gen 和 $Kill$ 并不好定义，难以直接通过集合运算得到 Out，不过可以正向遍历基本块内的指令，从而将 In 集合转换为 Out 集合。主要有以下几种情况

1. 对于形如 $z = x \oplus y$ 这样的 `Binary` 指令，若两操作数为常量，直接计算得到结果，置寄存器 z 的值为常量；若其中一个操作数为变量，则 z 的值为变量，否则 z 未定义。对于 `Unary` 同理
2. 对于 `LoadImm`，直接将目的寄存器置为常量
3. 对于 `Assign`，目的寄存器的值与源寄存器一致
4. 对于 `Load, Call`，目的寄存器为 `Nac`，表示该值是变量
5. 对于 `LoadStr, LoadVtbl`，因为没有实现它们的常量推导，置目标寄存器为 `Undef`

其中置为常量的操作类似于 Gen，置为 `Undef` 类似于 $Kill$，显然这里的操作是比前面几种优化方法要复杂的。迭代算法结束后，得到每个基本块的 In 集合，可以用类似的方法得到每个指令的 In 集合，从而可以根据寄存器值的情况对指令进行修改，如 `Binary` 指令的两个操作数均为常量时，可以替换为 `LoadImm` 对应表达式的值，如果一个操作数为常量，可以实现算术恒等式优化，如 $x + 0, x * 1$ 这类操作均可以替换为 `Assign`，而 $x \,\&\&\, 0, x \,||\, 1, x * 0$ 这种有确定值的则可以变为 `LoadImm`；对于 `CondBranch`，如果条件为常量，可以将其转换为普通 `Branch`，且可能产生一些不可达块，从而可以进一步优化。

经过常量传播后，一些寄存器可能变得不活跃，可以通过死代码消除进行优化。可以看到，复写传递、常量传播、公共表达式提取均可能产生死代码，因此实现时先做这三种优化，之后做一次死代码消除，最后做一次窥孔优化，由此作为一轮 pass，总体上进行三轮，完成最终的优化。

7 优化结果

框架在 PA3 基础上进行开发，因此支持 Decaf 的新特性，运行后程序将输出优化前后的 tac 文件和 info 文件，后者包含了优化前后模拟器中执行的指令数量。，测例来自于官方

PA4 的 8 个测例（5 个 basic+3 个较大规模测试）、PA3 的 33 个测例（包含了 Decaf 新特性）、5 个自己编写的 Decaf 代码（用于测试常量传播、复写传播、公共表达式提取），运行时开启了上述的所有优化，结果如下

7.1 官方测例

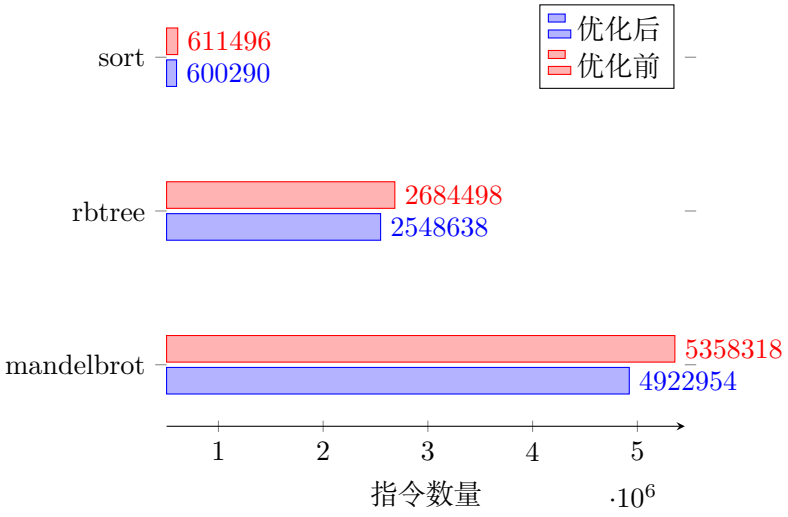

图 1: 官方样例测试结果 (较大样例)

样例名称	优化前指令数量	优化后指令数量
basic-basic	41	32
basic-fibonacci	7014	6747
basic-math	209	185
basic-queue	5827	5465
basic-stack	867	843

表 1: 其他 5 个官方样例测试结果

可以看出，

7.2 来自 PA3 的测例

表 2: PA3 样例测试结果 (部分)

样例名称	优化前指令数量	优化后指令数量
abstract-1	464	436
boolarray	3166	2853
call-1	58	54
call-2	118	110
call-3	92	91

lambda-1	272	247
lambda-2	279	235
lambda-3	436	408
math	209	185
matrix	35238	34637
method-name-1	354	349
test_divisionbyzero1	12	7
var-ok-1	181	134
var-ok-2	106	97

7.3 构造的特殊测例

7.3.1 common_expr.decaf

这个测例使用多个重复字串，多次调用成员函数，用于展示公共表达式提取能实现的一些功能，如常量字符串合并

```
string x = "dup_string";
Print(x);
string y = "dup_string";
Print(y);
string z = "dup_string";
Print(z);
var a = new A();
a.member_func();
a.member_func();
a.member_func();
```

最后生成的汇编代码一部分为

```
_T24 = "dup_string"
parm _T24
call _PrintString
parm _T24
call _PrintString
parm _T24
call _PrintString
```

可以看出"dup_string" 只被保留了一份。但 member_func 调用三次产生的三个函数对象却无法优化掉，因为该对象在堆区分配，而优化算法并不关注这些内容。

7.3.2 common_expr_2.decaf

这个样例用于测试公共表达式提取算法的正确性，和上个例子相比，Print(z) 前加了以下内容

```
if(get_choice()) {
```

```
 z = "another string";
}
```

生成的 Tac 为

```
_T31 = "dup_string"
parm _T31
call _PrintString
parm _T31
call _PrintString
_T4 = _T31
... // some codes to assign value to _T11
if (_T11 == 0) branch _L1
_T12 = "another string"
_T4 = _T12
_L1:
parm _T4
call _PrintString
```

可以看到，分支语句的加入阻断了 `z` 的公共表达式合并，并且由于 `_T4` 的两个复写语句不一致，`_L1` 块中的 `parm _T4` 语句没有得到复写传播，说明算法正确。

7.3.3 const_prop.decaf

这个样例用于测试常量传播的正确性

```
int x = 2;
int y = 3;
int z = x + y; // 5
int a = x * z; // 10
int b = a / 2; // 5
int c = b + b + b; // 15
Print(c,"\n");
c = 999;
Print(c,"\n");
```

由于常量传播、死代码移除均开启，优化后可以直接得到 `c` 的最终值。生成的 Tac 代码部分如下

```
_T14 = 15
parm _T14
call _PrintInt
```

可以看出，程序正确地完成了编译期计算。

7.4 const_prop_arith.decaf

这个样例用于测试对于算术恒等式的处理，如 `x + 0`, `x * 1`, `x * 0`, `x && 0`, `x || 1`, `0 / x` 等

7.5 copy_prop.decaf

这个样例用于测试复写传播，主要代码如下

```
int x = (fun() => 2)();
int y = (fun() => 3)();
int z = x + y * x + y;
int a = z;
int b = a;
int c = a + b;
Print(c,"\n");
```

注意如果前两行直接写成赋值语句将无法测试复写传播，因为可能直接被常量传播优化掉，写成 Lambda 表达式将规避这种优化（算法没有考虑函数返回值为常量的情况）。生成的 Tac 代码如下

```
_T6 = call _T5  // x
...
_T13 = call _T12 // y
_T15 = (_T13 * _T6)
_T16 = (_T6 + _T15)
_T17 = (_T16 + _T13)
_T21 = (_T17 + _T17)
parm _T21
call _PrintInt
```

可以看到，后续引用 `x` 和 `y` 的地方已经直接从对应寄存器取值，而复写传播后多余的语句也被死代码消除优化掉了。