编译原理 PA4 实验报告

2017011620 计 73 李家昊

2020 年 1 月 5 日

1 工作内容

本次实验基于 Java 框架，实现了死代码消除算法，为了证明其正确实现，这里还实现了复写传播和常量传播算法，并在公开测例上取得了不错的优化结果。

1.1 死代码消除

1.1.1 实现流程

中间代码优化是以函数为单位的，对于一个 TAC 程序的每一个函数，调用框架提供的`CFGBuilder` 建立控制流图，首先消除不可达代码块，这里采用 BFS 从 0 节点开始进行搜索，将不可达的顶点从 CFG 中删除。

为了描述一个数据流，我们只需要确定它的传递函数以及交汇函数。死代码消除是一种后向数据流，在一个程序单元 B 中，它的传递函数定义如下：

$$f(x_B) = use_B \cup (x_B - def_B) \tag{1}$$

其中 use_B 表示 B 中被定值之前要引用的变量集合，def_B 表示 B 中所有定值的变量集合。对任意两个程序单元 B_1, B_2，交汇函数定义如下：

$$merge(x_{B_1}, x_{B_2}) = x_{B_1} \cup x_{B_2} \tag{2}$$

其中的并集操作表示：只要变量 v 在程序单元 B 的任意一个后继 B_s 的入口处是活跃的，那么它在 B 的出口处也是活跃的。只要定义了传递函数和交汇函数，就可以使用通用的数据流求解算法确定数据流了。

在具体实现中，框架中的 `LivenessAnalyzer` 已经实现了活跃变量分析算法，包括基本块间的数据流求解以及每一条语句执行后的活跃变量分析。在此基础上，需要略微修改每条语句的分析逻辑，具体来说，从后向前遍历基本块中的每一条指令，根据传递函数维护一个活跃变量集合，若当前指令为 `DirectCall`，`IndirectCall` 其中之一，并且调用的函数返回类型为 `void` 时，则将目标寄存

器去掉；否则，若当前指令有目标寄存器，且其目标寄存器不在活跃变量集合内，则直接删除这条指令。这样就完成了死代码消除，且上述操作均不会带来副作用。

1.1.2 遇到的困难

在删除调用语句的目标寄存器时，发现 TAC 模拟器总是在运行时崩溃，一度以为是我写的代码有问题，后来发现是 Java 模拟器的 bug/feature，它不支持没有目标寄存器的函数调用，后来通过修改模拟器解决了这个问题。

1.2 复写传播

1.2.1 实现流程

复写传播是一个前向数据流，定义它的传递函数如下：

$$f(x_B) = gen_B \cup (x_B - kill_B) \tag{3}$$

其中 gen_B 表示在 B 中定值且能到达 B 出口的复写表达式集合，$kill_B$ 表示 B 中定值的寄存器集合。定义交汇函数如下：

$$merge(x_{B_1}, x_{B_2}) = x_{B_1} \cap x_{B_2} \tag{4}$$

其中的交集操作表示：只有所有 B 的前驱 B_p 的出口处都存在变量 v 的相同的复写表达式，那么在 B 的入口处才存在这个复写表达式。

在具体实现中，x 和 gen_B 均为复写表达式集合`HashMap<Temp, Temp>`，$kill_B$ 为寄存器集合`Set<Temp>`，定义 $x_B - kill_B = \{(k,v)|(k,v) \in x_B \wedge k \notin kill_B\}$。首先实现讲义上的前向数据流求解框架，得到基本块之间的数据流，然后在每个基本块内，从前向后遍历每一条指令，根据传递函数维护一个可用复写表达式集合，对于指令中的每个源寄存器，在集合中递归的查找其可用的复写表达式，并进行替换。

1.3 常量传播

1.3.1 实现流程

常量传播也是一个前向数据流，定义它的传递函数如下：

$$f(x_B) = gen_B \cup (x_B - kill_B) \tag{5}$$

定义交汇函数如下：

$$merge(x_{B_1}, x_{B_2}) = \{(k, v_1 \odot v_2) | k \in (x_{B_1} \cup x_{B_2}).keys, v_1 = x_{B_1}[k], v_2 = x_{B_2}[k]\} \tag{6}$$

其中 $\odot$ 运算符用于计算两个常量表达式的交汇类型，定义如下：

$$x \odot y = \begin{cases} x, & (x, y \in \text{CONST} \wedge x = y) \vee (x \in \text{CONST} \wedge y \in \text{UNDEF}) \\ y, & x \in \text{UNDEF} \wedge y \in \text{CONST} \\ \text{UNDEF}, & x, y \in \text{UNDEF} \\ \text{NAC}, & otherwise \end{cases} \tag{7}$$

在具体实现中，首先定义常量右值类 `ConstRhs`，包括 `CONST`, `NAC`, `UNDEF` 三种类型，x 和 gen_B 为常量表达式集合`HashMap<Temp, ConstRhs>`，$kill_B$ 为寄存器集合`Set<Temp>`，定义 $x_B - kill_B = \{(k, v) | (k, v) \in x_B \wedge k \notin kill_B\}$。与复写传播类似，首先根据传递函数和交汇函数求出控制流，然后从前往后遍历每一条 TAC 指令，根据传递函数维护一个可用常量表达式集合，对于不同的指令采取不同的处理逻辑，例如，在 `Binary` 指令中，如果两个操作数都是常量，则进行编译期计算，使得目标寄存器也是常量；在 `CondBranch` 指令中，如果寄存器的值是常量，则可以将其优化为无条件跳转或无条件不跳转，等等。

1.3.2 遇到的困难

Java 框架中需要对不同类型的表达式分别处理，如果直接用 `instanceof` 判断的话代码会非常 dirty，后来发现 `TacInstr` 提供了 `Visitor` 接口，就改为访问者模式了。

2 性能测试结果

这里对每一种优化算法进行详细的对比，给出每个公开测例的 TAC 执行指令的数量，以及 TAC 代码文件的行数。为了最大化优化效果，这里在测试每一组优化方法时，迭代运行优化算法 3 次后，取最终的优化结果。

不同优化算法下的 TAC 执行条数如 Table 1所示。可以看出，仅仅使用死代码消除时，并不能对公开测例有任何优化，原因是公开测例的代码写的非常规范，没有任何冗余代码，比如定义后未使用的变量。为了真正实现代码优化，还需要实现额外的优化算法，比如复写传播和常量传播，将它们配合死代码消除算法使用，能获得不错的优化效果。当三种优化方法共同使用时，在 `mandelbrot`, `rbtree`, `sort` 三个测例上的执行行数分别降低了 4.84%, 3.40%, 0.45%。

Test Case	Original	AL	CP+AL	CT+AL	CP+CT+AL
basic-basic	41	41	**37**	38	**37**
basic-fibonacci	3426	3426	3426	**3425**	**3425**
basic-math	139	139	139	**136**	**136**
basic-queue	2536	2536	2476	2532	**2472**
basic-stack	733	733	729	729	**725**
mandelbrot	3893085	3893085	3815344	3782271	**3704531**
rbtree	2439827	2439827	2356807	2439816	**2356796**
sort	560987	560987	559497	559978	**558488**

表 1: 不同优化方法下的 TAC 执行行数，其中 AL 表示死代码消除，CP 表示复写传播，CT 表示常量传播。

不同优化算法下的 TAC 文件行数如 Table 2所示。可以看出，仅使用死代码消除时，由于支持了不可达代码块消除，因此在代码文件行数上会有所减小。三种优化方法都降低了代码文件大小，而三种方法的配合使用更是达到了最好的优化效果

Test Case	Original	AL	CP+AL	CT+AL	CP+CT+AL
basic-basic	54	54	**50**	51	**50**
basic-fibonacci	74	74	74	**73**	**73**
basic-math	146	137	137	**134**	**134**
basic-queue	249	248	245	244	**241**
basic-stack	193	193	191	185	**183**
mandelbrot	540	540	538	486	**485**
rbtree	606	603	590	600	**587**
sort	684	684	682	663	**661**

表 2: 不同优化方法下的 TAC 文件行数，其中 AL 表示死代码消除，CP 表示复写传播，CT 表示常量传播。