软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software, 2022,33(8):2964–2979 [doi: 10.13328/j.cnki.jos.006608]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

基于抽象解释的函数内联过程间分析优化方法*

陈涛清[1], 范广生[1], 尹帮虎[2], 陈立前[1], 王 戟[1,3]

[1](国防科技大学 计算机学院, 湖南 长沙 410073)

[2](国防科技大学 系统工程学院, 湖南 长沙 410073)

[3](高性能计算国家重点实验室(国防科技大学), 湖南 长沙 410073)

通信作者: 尹帮虎, E-mail: bhyin@nudt.edu.cn; 王戟, E-mail: wj@nudt.edu.cn

摘 要: 分析实际程序时往往需要分析程序中函数的调用, 一般使用过程间分析来实现全程序分析. 函数内联是一种最为精确、易于实现的过程间分析方法. 通过函数内联, 可以使得已有过程内分析方法和工具支持包含函数调用的程序的分析. 但是函数内联后代码的规模急剧增加, 同时将产生大量中间变量, 增加程序分析的变量维度, 导致程序分析过程时空开销大大增加. 考虑基于抽象解释框架下函数内联过程间分析的一些不足, 并提出了相应的优化方法. 基于抽象解释的程序分析关注自动推导程序变量之间的不变式约束关系, 因此程序变量构成的程序环境大小(即各程序点处须考虑的相关变量集合)对分析的时空开销具有重要影响. 为了减少函数内联后程序分析的开销, 提出了面向内联函数块的程序环境降维优化方法. 该方法针对内联函数后的程序代码, 分析确定不同程序点处需维护的程序环境(即相关变量集合), 而不是所有程序点共享同一全局程序环境, 从而实现程序状态的降维. 详细描述了基于该方法所实现的工具 DRIP (dimension reduction for analyzing function inlined program)的架构、模块及算法细节. 并在 WCET Benchmarks 测试集开展了分析实验. 实验结果表明: DRIP 在变量消除上取得的效果良好, 甚至在某些测试集上能减少一半以上的变量, 并在一定程度上降低了分析过程的时空开销.

关键词: 过程间分析; 抽象解释; 函数内联; 变量消除; 降维

中图法分类号: TP311

中文引用格式: 陈涛清, 范广生, 尹帮虎, 陈立前, 王戟. 基于抽象解释的函数内联过程间分析优化方法. 软件学报, 2022, 33(8): 2964–2979. http://www.jos.org.cn/1000-9825/6608.htm

英文引用格式: Chen TQ, Fan GS, Yin BH, Chen LQ, Wang J. Optimization Technique for Interprocedual Analysis Using Function Inlining in Abstract Interpretation. Ruan Jian Xue Bao/Journal of Software, 2022, 33(8): 2964–2979 (in Chinese). http://www.jos.org.cn/1000-9825/6608.htm

Optimization Technique for Interprocedual Analysis Using Function Inlining in Abstract Interpretation

CHEN Tao-Qing[1], FAN Guang-Sheng[1], YIN Bang-Hu[2], CHEN Li-Qian[1], WANG Ji[1,3]

[1](College of Computer Science and Technology, National University of Defense Technology, Changsha 410073, China)

[2](College of Systems Engineering, National University of Defense Technology, Changsha 410073, China)

[3](State Key Laboratory for High Performance Computing (National University of Defense Technology), Changsha 410073, China)

Abstract: When analyzing real-world programs, it is often necessary to analyze the function calls in the program, and interprocedural analysis is generally used to achieve full program analysis. Function inlining is one of the most accurate and easily achievable methods for interprocedural analysis. Function inlining allows existing intraprocedural analysis methods and tools to support the analysis of programs that contain function calls. However, the size of the code increases dramatically after function inlining, and a large number of

* 基金项目: 国家自然科学基金(62032024, 61872445, 62102432); 湖南省自然科学基金(2021JJ40697)
本文由“形式化方法与应用”专题特约编辑陈立前副教授、孙猛教授推荐.
收稿时间: 2021-09-06; 修改时间: 2021-10-14; 采用时间: 2022-01-10; jos 在线出版时间: 2022-01-28

intermediate variables will be generated, which increases the variable dimension of program analysis and causes the process of program analysis to consume a lot of memory and time. This study considers some shortcomings of interprocedural analysis of function inlining based on the abstract interpretation framework and proposes a corresponding optimization method. Program analysis based on abstract interpretation is concerned with automatically deriving invariant constraint relations among program variables, so the size of the program environment constituted by program variables (i.e., the set of relevant variables to be considered at each program point) has an important impact on the time and space overhead of the analysis. In order to reduce the overhead of program analysis after function inlining, this study proposes a program environment reduction optimization method for inline function blocks. The method analyzes the program code after inlining functions to determine the program environment (i.e., set of related variables) to be maintained at different program points, instead of sharing the same global program environment at all program points, so as to realize the dimensionality reduction of the program state. The architecture, modules, and algorithm details of the tool DRIP (dimension reduction for analyzing function inlined program) implemented based on this method are described in detail. The results show that DRIP achieves sound results in variable elimination, even reducing more than half of the variables in some test suites, and the time and space spent is also reduced to a certain extent.

Key words: interprocedural analysis; abstract interpretation; function inlining; variable elimination; dimension reduction

发现软件的缺陷与漏洞, 保障软件的可信性, 是研究人员持续关注的课题[1]. 程序作为软件的载体, 对其进行分析、确认或发现其特性变得尤为重要[2]. 程序分析的基础理论——抽象解释[3]给出了程序语义的可靠近似理论, 为静态程序分析提供了一个自动推导程序性质的通用框架[4]. 从抽象解释的角度出发, 程序分析即通过一定的求解策略去计算由抽象语义近似的具体语义, 在每个程序点得到刻画程序性质的不变式. 基于抽象解释的程序分析关注自动推导程序变量之间的不变式约束关系, 因此, 程序变量构成的程序环境大小(即各程序点处须考虑的相关变量集合)对分析的时空开销具有重要影响.

在分析实际程序的过程中, 程序中不可避免地会出现函数调用, 甚至会有同一个函数多次调用或者一个函数嵌套调用多层函数. 更进一步, 实际应用中, 待分析的程序往往具有很大的规模. 为了得到较为精确的分析结果, 需要引入过程间分析. 过程间分析是程序分析中的一种关键技术, 它对整个程序进行处理, 将信息从调用者传到被调用者, 或者是反向传递, 是一种跨过程边界进行信息跟踪的方法[5]. 函数内联是过程间分析中一种最为精确、且易于实现的方法, 它把函数调用替换为函数代码拷贝, 以区分不同的函数调用点和返回点, 从而提高分析精度, 适用于需要精确结果的程序分析场景[6]. 通过函数内联, 可以使得已有过程内分析方法和工具支持包含函数调用的程序的分析. 然而, 函数内联后代码的规模会急剧增加, 并且会产生大量中间变量, 增加了程序分析的变量维度, 导致程序分析过程的时空开销大大增加.

在基于抽象解释的理论框架下, 使用函数内联的过程间方法对程序进行分析, 可以对函数调用进行精确的处理. 然而, 函数内联作为一种简单和完整的函数体拷贝替换的方法, 函数内联后代码的规模急剧增加, 同时将产生大量中间变量, 增加了程序分析的变量维度, 导致程序分析过程有较大时空开销. 在基于抽象解释的分析过程中, 这些中间变量(包含代替形参的临时变量与被内联函数体中的局部变量)与原程序中的变量一起存储在全局程序环境(environment)中, 且生存期贯穿于整个分析过程, 加大了分析过程的开销, 尤其是空间开销. 同时, 环境作为不变式的变量维度信息, 其中的临时变量与被内联函数体中的局部变量体现在每个程序点的不变式上, 使得各程序点的不变式可能包含很多逻辑上不相关变量之间的约束, 从而包含较多冗余信息, 影响了分析效率.

针对抽象解释理论框架下使用函数内联的过程间分析方法带来的问题, 本文提出一种面向内联函数块的程序环境降维优化方法. 该方法针对内联函数后的程序代码, 分析确定不同程序点处需维护的程序环境(相关变量集合), 而不是所有程序点共享同一全局程序环境, 从而实现程序状态的降维. 本文基于该方法实现了工具 DRIP (dimension reduction for analyzing function inlined program), 并在 WCET Benchmark 测试集开展了分析实验.

本文的主要贡献包括以下两点.

(1) 提出了一种面向内联函数块的程序环境降维优化方法;

(2) 基于该方法实现了工具 DRIP, 实验表明: DRIP 在临时变量与被内联函数体中的局部变量的消除上取得的效果良好, 且在线性抽象域中进行分析时, 一定程度上降低了分析过程的时空开销.

本文第 1 节简要介绍抽象解释理论框架下进行函数内联过程间分析的相关背景. 第 2 节举例阐述研究动机, 并简要介绍降维优化方法的实施步骤. 第 3 节给出了基于该降维优化方法的工具框架和方法细节. 第 4 节介绍工具的实验、结果及分析. 第 5 节简述相关工作. 第 6 节总结全文并进行未来展望.

1 相关背景

1.1 抽象解释

抽象解释理论由 Cousot 等人于 1977 年提出, 是一种对程序语义进行可靠近似的通用理论, 从理论上保证了所构建的程序分析的终止性和可靠性[3]. 基于抽象解释的程序分析, 本质上是通过对程序语义进行不同程度的抽象, 以在分析精度和计算效率之间取得权衡. 这种由某种语义抽象及其上的操作所构成的数学结构称为抽象域, 抽象解释采用 Galois 连接来刻画具体域与抽象域之间的关系(图 1 解释了其内涵, 其中, C 表示具体域, A 表示抽象域, α表示抽象化函数, γ表示具体化函数. 利用α和γ函数在具体域与抽象域之间建立映射关系, 从而刻画两者的关系), 通过在抽象域上计算程序的抽象不动点来表达程序的抽象语义. 在基于抽象解释的程序分析中, 程序状态集合通过抽象域中的域元素来近似, 而程序语义动作(如赋值、条件测试等)通过抽象域中的域操作(如迁移函数)来可靠建模. 此外, 为了加速抽象域上不动点迭代的收敛速度并保证不动点迭代的终止性, 抽象解释框架提供了加宽算子(widening). 抽象解释提供了严格的理论来保证基于上近似抽象的推理的可靠性, 能够保证抽象域上迭代求得的抽象不动点是程序的最小不动点.

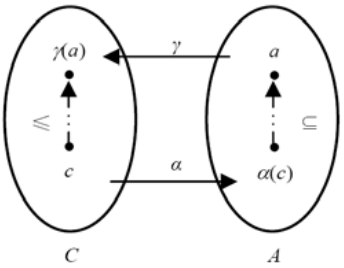

图 1 Galois 连接

数值抽象域是抽象解释理论中非常重要的概念. 数值抽象域中, 域元素描述的是数值变量的可能取值. 对于包含 n 个变量的程序, 记这 n 个变量构成的集合为 $Vars$. 实际上, 每个程序环境ρ: $Vars\to R$ 都可以看作是 n 维空间 R^n 上的一个点 $p=(\rho(v_1),\ldots,\rho(v_n))$. 直观上讲, 每个程序变量 v_i 对应 n 维空间 R^n 中的一维; 点 p 在该维对应坐标轴上的值, 即为变量 v_i 在程序环境ρ中的值. 程序环境的集合 $Env\equiv\wp(Vars\to R)$对应于空间 R^n 的子集, 即该空间上一些点的集合. 在程序分析过程中需要维护程序环境, 而程序环境取决于所关心的变量的集合 $Vars$.

不同的数值抽象域从不同表达能力上来表示、从不同计算效率上来操作这些点集. 目前, 存在着许多表达能力、计算效率各有特色的数值抽象域, 包括区间域[7]、八边形域[8]和多面体域[9]等经典抽象域. 需要注意到: 区间域只用维护每个变量的数值信息, 而八边形域和多面体域需要维护每个变量间的关系. 区间域的时空复杂度均为 $O(n)$; 八边形域的时间复杂度为 $O(n^3)$, 空间复杂度为 $O(n^2)$; 而多面体域的时空复杂度均为指数级. 可见, 变量维数会极大影响抽象域分析效率. 这些抽象域面向各种不同的数值性质, 并在分析精度和计算代价间取得某种权衡. 此外, 还出现了若干开源的抽象域库, 如 APRON[10], ELINA[11], PPL[12]等. 这些抽象域库提供了一组必需的公共接口, 从而避免了实现不同的抽象域所带来的复杂性问题, 使用者只需要在使用时指定相应的抽象域库, 再调用相应的实现函数, 便能实现对应的功能, 这令开发相关工具的效率大大提升.

与此同时, 基于抽象解释的程序分析工具也不断地出现在开源社区和商业公司, 如 Astree[13], Frama-C value Analysis[14], Sparrow[15]和 Interproc[16]等. 基于抽象解释的程序分析方法是对大规模软、硬件系统进行自动化分析与验证的有效途径之一, 已被广泛地应用于大型软、硬件系统的验证研究中, 其中的典型代表是 Astree. Astree 可以检测 C 程序编写的运算密集型的安全攸关嵌入式实时软件系统中运行时错误(包括算术溢

出、除零错、数组越界等)[13]. Astree 已成功地对空客 A340(约 13.2 万行 C 代码), A380(约 35 万行 C 代码)等系列飞行控制软件进行分析, 并实现了“零误报”[17]; 其扩展版本 AstreeA 支持多线程 C 程序中运行时错误、数据竞争、死锁等的检测, 并成功应用于 ARINC 653 航空电子应用软件(约 220 万行代码)的分析[18].

1.2 过程间分析

过程间分析(全局分析)对包含多个函数和函数调用的整个程序进行分析, 是在不改变程序代码和过程间调用关系的前提下对过程间信息流动的分析[19]. 过程间分析是程序进行精确分析的必要条件, 当前的方法主要是在效率上和精度上进行权衡. 从数据流分析的精确性来看, 过程间分析可以分为上下文敏感[20−23]和上下文不敏感[24]两种方法. 内联是上下文敏感分析方法的特例, 是最精确的函数间分析方法.

内联方法分为语法内联和语义内联. 其中, 语法内联将被调函数直接展开插入调用点并参与分析, 再对完整程序进行过程内分析[6]. 类似地, Emami 等人[19]提出的跨过程指针分析方法把调用函数代入到主调函数中, 使得上下文关系在主调函数中得以直接体现. 语义内联是指在分析代码时动态地维护一个调用栈信息, 以区分函数在不同的上下文中的调用. 语义内联与语法内联相比, 主要优点在于可以方便地对调用栈进行抽象. 由于语法内联是直接对函数代码进行了内联扩展, 其在实际分析过程中并不区分函数调用与普通语句, 而语义内联使用了额外的变量去模拟调用过程中函数调用栈的具体情况, 因此, 语义内联比语法内联更容易实现.

内联方法被广泛应用于许多程序分析工具(Astree[17], CBMC[25], ESBMC[26], SMACK[27], CPACheck[28]和 Frama-C[14,29])的过程间分析处理中, 因此, 针对内联方法的优化可以极大提升现有程序分析工具的性能, 具有极大的现实价值.

2 概 述

2.1 研究动机

为了对程序进行精确分析, 在抽象解释理论框架下, 我们使用函数内联的过程间分析方法进行过程调用的信息流动分析. 以选取文献[30]中的janne_complex.c程序作为例子进行说明, 为了更好地描述, 我们对其进行了部分的改动. 表 1 展示了其程序代码, 包括主函数 *main* 以及被调函数 *complex* 和 *foo*, 其中, *complex* 是一个内层循环上限依赖于外层循环的双重循环, *foo* 是两个数之差的绝对值. 对该程序使用函数内联的过程间分析方法, 并将函数内联之后的程序保存为 janne_complex_inline.c(由于篇幅原因, 此处没有展示其代码).通过对比内联前后的代码, 即 janne_complex.c 与 janne_complex_inline.c, 可以发现函数内联是在原程序的主函数 *main* 中, 对其中出现的函数调用(表 1 第 39 行 *complex* 和第 40 行 *foo*)使用被调函数的函数体(表 1 第 3–23 行与第 27 –33 行)进行代码替换. 这种替换并不是直接拷贝被调函数的代码, 而是重新创建中间变量代替形参和函数体中的局部变量, 用中间变量构成的语句进行替换, 这使得程序的规模与变量数显著增加(代码行数和变量增加的原因, 除了使用函数内联的方法之外, 还有 C 程序处理前端对原程序进行中间表示转换, 将在第 3.1 节前端模块说明).

我们使用实际代码行数(去除空行和注释行)来刻画程序规模, 实际代码行数与变量数在程序内联前后的对比见表 2. 其中, 使用工具[31]对 janne_complex.c 与 janne_complex_inline.c 的行数进行统计, 原程序与函数内联后程序的行数相差两倍; 变量数来源于程序分析过程中对程序环境的记录, 两者相差一倍. 表 1 中给出的例子是一个简单的程序, 当待分析的程序规模达到百行或千行级别时, 可以预测, 其函数内联后全局程序环境中的变量数必然是成倍的增长, 这将会给分析过程带来很大的时空消耗. 特别地, 临时变量与被内联函数体中的局部变量在函数内联创建后就存在于全局程序环境中, 并一直持续存在直到分析结束. 然而, 程序环境中的临时变量与被内联函数体中的局部变量对最终的分析结果没有影响, 不仅占用了一定的内存, 还使得最终结果冗余. 为此, 提出了一种面向内联函数块的程序环境降维优化方法来消除程序环境中的中间变量.

表 1 janne_complex.c 程序代码

```
1  int complex(int a,int b)
2  {
3     int i, j;
4     i=j=0;
5     while (a<30)
6        {
7           ++i;
8           while (b<a)
9           {
10             ++j;
11             if (b>5)
12                b=b*3;
13             else
14                b=b+2;
15             if (b≥10 && b≤12)
16                a=a+10;
17             else
18                a=a+1;
19       }
20          a=a+2;
21          b=b−10;
22       }
23    return 1;
24 }
25 int foo(int x,int y)
26 {
27    int m, t;
28    m=xy;
29    if (m>0)
30       t=m;
31    else
32       t=−m;
33    return t;
34 }
35
36 int main(·)
37 {
38    int a=1, b=1, answer=0;
39    answer=complex(a,b);
40    foo(a,b);
41    return answer;
42 }
```

表 2 janne_complex.c 程序内联前后对比

程序	实际代码行数	变量数
janne_complex.c	41	9
janne_complex_inline.c	124	18

2.2 方法概述

代码块(block)是 C 程序中的基本结构, 它通常处于一对花括号之间, 常见的代码块有条件分支语句(包括 if-else 和 switch-case 等情形)、循环语句(包含 while 和 for 等情况)和赋值语句等. 为了对函数内联的方法产生的临时变量与被内联函数体中的局部变量进行消除, 基于代码块, 我们定义了一种名为内联函数块的优化场景, 其指代主函数中被调函数在函数内联进行代码替换后的代码块, 它可能是上述几种类型的代码块中一种或多种的组合. 表 3 第 1–17 行给出了表 1 中 *foo* 函数的内联函数块. 结合 *foo* 函数的定义可知: 内联函数块包含了由中间变量构成的函数参数传递赋值语句(第 3–6 行)、函数体语句(第 7–15 行)和返回语句(第 16 行), 其包含赋值语句和条件分支语句等多个代码块.

以表 3 所示内联函数块为例, 我们给出面向内联函数块的程序环境降维优化方法的概述, 其主要分为待消除变量的选取、类型转换和增加删除这 3 个步骤.

(1) 待消除变量的选取

程序环境降维优化是对环境中的变量进行删除, 为了使变量消除的同时不影响分析过程的正常运行, 我们需要合理地选取待消除的变量以防止误删. 同时, 在不误删的前提下, 我们也要尽可能多地选取变量, 减少内存消耗. 内联函数块是使用函数内联方法进行代码替换得到, 该过程创建中间变量作为被调函数形参和函数体局部变量的替代, 其中, 我们将形参替代引入的变量视作临时变量, 与函数体内原有的局部变量进行区分. 对于一个内联函数块而言, 这些中间变量的作用范围仅限于该代码块中, 对其的删除不影响内联函数块以外的分析过程. 于是, 我们选取一个内联函数块中的中间变量集合作为待消除的变量. 以表 3 中的内联函数块为例, 根据上述可知: 待消除的变量为变量集合{$\_\_x_{10}$,$\_\_y_{11}$,$\_\_m_{12}$,$\_\_t_{13}$}, 其中, 变量$__x_{10}$和$__y_{11}$是内联代码替换后引入的临时变量, 而变量$__m_{12}$和$__t_{13}$是对原程序中变量 m, t 的替换, 即内联代码替换过程包含了对原程序中局部变量的替换, 这两个变量即为被内联函数体中的局部变量. 于是, 待消除的变量集合包含了临时变量与被内联函数体中的局部变量, 对此变量集合进行消除, 就能够实现既定的消除临时变量与被内联函数体中的局部变量的目标.

(2) 待消除变量的类型转换

程序环境是抽象域库中的结构, 我们选用的抽象域库 APRON 仅支持整型和浮点型(int/real)这两种类型的变量. 然而, 选取的待消除变量的类型来源于程序中的定义, 其不局限于整型和浮点型, 如数组、结构体等. 故对于待消除变量需要进行类型转化, 方能在抽象域库的环境中进行增删操作. 变量类型转化的处理细节, 我们会在第 3.2 节变量类型转换中详细阐述. 对于上述步骤(1)中的变量集合, 其类型均为 C 语言中的 int 类型. int 类型是 APRON 库所支持的, 故只需要将其加入环境中相应的变量数组即可.

(3) 待消除变量的增加删除

在程序环境降维优化中, 进行变量增加是为了应对同一函数多次调用, 即某个内联函数块多次出现的情况, 再次分析该内联函数块时, 需事先将已经删除的相关变量集合再次添加到环境中, 这是满足分析正常运行的必要条件; 进行变量删除的目的是降低环境维度和减少内存消耗. 无论是变量的增加或是删除, 其保障分析过程正常运行的关键是选取合适的变量增加或删除时机. 对于一个内联函数块的待消除变量集合而言, 其增加的时机应为该函数块的入口语句前, 以表 3 中的内联函数块为例, 其变量增加应在表 3 中第 4 行以前; 其删除的时机应为该块的出口语句后, 以表 3 中的内联函数块为例, 其变量删除应在表 3 中第 16 行以后. 增删时机的语句位置对应于原程序中的程序点, 由于代码替换, 内联函数块的程序点相较于原程序发生变化, 因此需要对其出入口程序点位置进行匹配, 其相关规则和实现细节我们会在第 3.2 节内联函数块出入口位置匹配部分详细展开. 同时, 变量的增加删除操作是在不动点迭代过程中进行, 其相关细节我们会在第 3.2 节变量增加删除部分说明.

表 3 内联函数块

1	{
2	…
3	{
4	$\_\_x_{10}=a$;
5	$\_\_y_{11}=b$;
6	}
7	{
8	$\_\_m_{12}=\_\_x_{10}-\_\_y_{11}$;
9	**if** ($\_\_m_{12}>0$) {
10	$\_\_t_{13}=\_\_m_{12}$;
11	} **else** {
12	$\_\_t_{13}=-\_\_m_{12}$;
13	}
14	goto *Lret_foo*;
15	}
16	*Lret_foo*: /*CIL Label*/;
17	}

3 面向函数内联的程序分析降维优化方法及实现

我们基于抽象解释框架下函数内联过程间分析的一些不足, 提出了面向内联函数块的程序环境降维优化方法. 该方法基于抽象解释框架, 并根据其内容对框架下的各个模块进行修改, 将方法集成到框架中, 从而实现分析环境中变量的降维优化工具 DRIP. 下面的第 3.1 节我们回顾基于抽象解释的静态分析工具架构及其功能模块; 第 3.2 节中, 我们在架构下对面向内联函数块的程序环境降维优化方法的各步骤的内容进行详细阐述, 并说明每个步骤相应集成的功能模块.

3.1 基于抽象解释的静态分析框架

基于抽象解释的静态分析工具一般包括前端处理、抽象域库和不动点求解器等模块. 图 2 给出了基于抽象解释的静态分析工具的框架. 分析一个程序的处理流程为: 对于待分析的原程序(如 C 程序), 使用前端模块对其进行解析(该过程包含使用内联方法对函数调用进行代码替换)得到中间表示, 再根据中间表示构建控制流图(CFG, 即语义方程); 随后, 在抽象域中, 控制流图的边信息会转化为域表示和域操作; 接着, 求解器对转化后边信息进行不动点迭代求解, 最终获得程序的数值不变式.

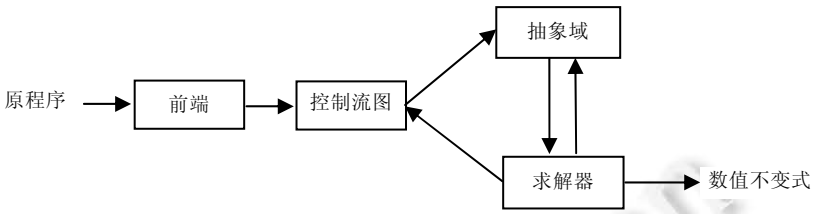

图 2 基于抽象解释的静态分析工具框架

我们基于抽象解释框架实现了静态分析工具 DRIP. 下面对工具的各个功能模块进行介绍.

(1) 前端模块

前端模块用于编译并解析输入的原程序, 得到原程序的中间表示. 工具 DRIP 使用 CIL (C intermediate language)[32]作为前端, 它是面向 C 程序的编译工具, 其实现了一种源到源的转换, 用于获取 C 程序的中间表示. CIL 支持对 C 程序文件的函数调用, 使用函数内联的过程间分析方法生成其中间表示. 前面我们提到, 内联后的程序代码规模增加显著. 原因除了使用函数内联的方法进行代码替换外, 还有 CIL 会对内联后程序的简化. 该过程使用一定的规则去简化程序中复杂的结构, 比如使用静态单赋值(SSA)形式简化赋值语句, 这也使得代码行数增加, 如表 1 展示的 janne_complex.c 程序内联得到的 janne_complex_inline.c 程序代码行数成倍地增加, 主要原因在于 CIL 前端内联的同时简化了原程序.

(2) 控制流图模块

控制流图以图的形式表示一个过程内所有基本块执行的可能流向, 它与前端处理得到的中间表示文件等价, 是不动点迭代求解的基础. 控制流图中的每个节点表示程序点, 每条边表示程序点间的迁移操作(如赋值、条件判断等). 程序点信息由(*line*,*sid*)二元组唯一确定, 其中, *line* 表示该点在原文件的行号, *sid* 表示语句的标号(CIL 简化后原程序的一条语句会拆成多条语句, 需要语句标号唯一标识程序点).

(3) 抽象域模块

抽象域是抽象解释框架下的核心要素, 通过选择特定的语义表示(域表示)和操作算法(域操作)来发现所关注的性质. 在程序分析中, 程序状态集合通过抽象域中的域元素来近似, 而程序语义动作通过抽象域中的域操作来可靠建模. 工具 DRIP 使用 APRON[10]库, 它是一个数值抽象域库, 为不同的抽象域提供了通用的函数接口, 便于开发者与接口进行直接交互, 省去了查看底层实现细节的步骤, 使得抽象域可重用性大大增强. 前文提到, APRON 中一个重要的结构是抽象环境, 在调用 APRON 库进行程序分析时, 环境就已经创建, 其包含了分析过程所需的所有变量的信息, 并在分析过程中一直持续存在. 在分析过程中, 环境中的变量随着控制流图边信息(语句抽象语义)而进行值更新, 直到达到不动点(环境中变量取值不改变). 降维优化就是对环境中的变量进行合理地增删, 减小变量的同时, 不影响分析过程的进行.

(4) 求解器模块

求解器模块用于对控制流图的边信息(语句)进行迭代求解不动点, 以获取程序的数值不变式. 工具 DRIP 使用不动点迭代库 Fixpoint[33], 其接口由抽象域参数化, 在抽象域上执行不动点计算; 其迭代策略采用了基于 worklist 的前向分析算法[34]. Fixpoint 的迭代求解流程为: 先将主函数入口处的抽象状态(环境各变量的值范围)初始化为 Top(抽象域完全格上最大值), 同时将其他程序点处的抽象环境初始化为 Bottom(抽象域完全格上最小值); 接着, 根据控制流图的迁移顺序, 对每条有向边调用抽象域上的域操作来进行迭代计算, 获取结果并更新到抽象状态中.

3.2 面向内联函数块的程序环境降维优化方法

对于没有函数调用的程序, 程序分析过程中, 我们一般维护一个所有程序点共享的全局程序环境, 该环境中包含所有的程序变量. 对包含函数调用的程序, 如果采用函数内联将其转换为不含函数调用的程序后, 代码的规模急剧增加, 同时将产生大量全局型中间变量, 增加了程序分析的变量维度, 导致程序分析过程有

大量的开销. 因此, 针对函数内联后的代码开展程序分析时, 降低分析的变量维度对于降低分析开销至关重要.对于一个内联函数块, 本文提出的程序环境降维优化方法主要包括待消除变量的选取、内联函数块出入口位置匹配、变量类型转换和变量增加删除这 4 个步骤, 下面对各步骤进行详细介绍, 并阐明如何将其对应集成到程序分析工具的各功能模块.

(1) 待消除变量的选取

函数内联时, 会引入临时变量来实现参数和返回值的传递. 同时, 被调用的函数内部也可能包含多个局部变量. 根据 CIL 中函数内联方法的特点, 对于一个被调函数, 其创建的用于替换代码的中间变量只用于该函数, 即一个被调函数对应一个中间变量集合. 当一个函数多次调用时, 代码替换中该函数的中间变量集合会复用. 因此, 一个被调函数的内联代码块有以下特点.

- 拥有一定数量的临时变量和函数内局部变量;
- 一个代码块对应一个临时变量和函数内局部变量集合, 函数多次调用时替换代码复用该集合;
- 对于一个内联函数块而言, 临时变量和函数内局部变量的作用范围仅限于该代码块中.

以上特点体现出内联函数块中的临时变量和函数内局部变量具有局部作用的性质, 其作为一个集合又具有整体性. 在该代码块之外的程序点对变量集合的消除可以保证分析过程正常运行, 于是, 我们选取函数内联产生的临时变量和函数内局部变量集合作为待消除的变量.

相应地, 在前端模块, 我们进行如下扩展.

- 针对内联函数块待消除变量的选取. 对每个被调函数, 我们记录内联做代码替换时创建的所有临时变量、函数内局部变量及其类型的集合, 并加入到相应的哈希表中;
- 为进行内联函数块出入口位置匹配. 对每个被调函数, 我们记录该函数在原程序中的行号, 并记录该函数调用的类型(有无参数、有无返回值和是否一行调用多个函数), 并加入到相应的哈希表中.

以表 3 中的 *foo* 内联函数块为例, 我们记录内联过程创建的中间变量作为待消除的变量集合, 相应的变量集合为{$\_\_x_{10}$,$\_\_y_{11}$,$\_\_m_{12}$,$\_\_t_{13}$}, 对应的类型集合为{int,int,int,int}, 并将其加入到哈希表中. 同时, 我们记录 *foo* 内联函数块的原行号与调用类型二元组(40,[3,4]), 其中, 40 为 *foo* 函数在原程序主函数 *main* 中的行号, 调用类型 3 和 4 代表函数有参数而不接收返回值, 并将其加入到哈希表中.

(2) 内联函数块出入口位置匹配

程序环境降维优化要求我们选取合适从程序环境中增加删除变量的时机. 对于从程序环境中删除变量而言, 需选取合适的时机以防止过早删除变量; 同时, 在分析过程中, 满足非过早删除的前提下, 尽可能地选取靠前的时机, 减少无关变量在环境中存在的持续时间, 降低存储开销. 相反地, 对于添加变量而言, 选取合适的时机防止增加变量太迟影响分析过程, 在此基础上, 再尽可能地选取靠后的时机.

增删变量时机需要选择合适的语句位置, 这对应于原程序中的程序点. 对于内联函数块而言, 其出入口程序点位置因代码替换发生改变, 为了把握好消除时机, 需要匹配定位内联后的出入口在控制流图中的位置. 由控制流图模块的介绍, 函数块出入口位置, 即程序点, 由(*line*,*sid*)二元组唯一标识. 以此为基础, 我们结合已经记录的原程序中被调函数的行号和调用类型来进行内联函数块出入口位置的匹配. 下面介绍匹配过程的基本步骤: 1) 创建两个空的哈希表 *blockEnterPos* 和 *blockExitPos*, 分别表示块入口位置和块出口位置, 表的键为函数名, 值为(*line*,*sid*)二元组; 2) 根据上述步骤(1)中记录的被调函数的行号和类型, 对于入口和出口的情况, 采用不同的规则去处理, 匹配到相应的位置标识二元组; 3) 将匹配到的二元组及其对应的函数名加入到创建的哈希表中, 最终完成匹配过程. 下面对步骤 2)中使用的规则进行说明, 其中, 规则 1~规则 3 是入口位置匹配规则, 规则 4~规则 6 是出口位置匹配规则.

- 规则 1: 函数调用无参数. 以被调函数第 1 条语句的入口位置作为内联函数块的入口位置, 并记录下对应的 sid;
- 规则 2: 函数调用时, 同一行有其他函数调用. 在控制流图中, 迭代程序点查找被调函数第一条语句的前一句的入口位置, 若其行号与记录的行号相等, 将该位置作为内联函数块的入口位置, 并记录下

对应的 sid;

- 规则 3: 函数调用有参数. 迭代控制流图中的程序点, 当节点行号等于记录的行号时, 若其 sid 小于当前的 sid, 更新 sid, 即迭代控制流图去查找记录的行号对应的最小 sid, 以记录的行号和更新的最小 sid 作为入口位置;
- 规则 4: 函数调用不接收返回值. 在控制流图中, 迭代程序点查找被调函数最后一条语句的后一句的出口位置, 若其行号为–1, 将该位置作为内联函数块的出口位置, 并记录下对应的 sid;
- 规则 5: 函数调用时, 同一行有其他函数调用. 在控制流图中, 迭代程序点查找被调函数最后一条语句的后一句的出口位置, 若其行号与记录的行号相等, 将该位置作为内联函数块的出口位置, 并记录下对应的 sid;
- 规则 6: 函数调用接收返回值. 迭代控制流图中的程序点, 当节点行号等于记录的行号时, 若其 sid 大于当前的 sid, 更新 sid, 即迭代控制流图去查找记录的行号对应的最大 sid, 以记录的行号和更新的最大 sid 作为出口位置.

相应地, 在控制流图模块, 我们进行如下扩展.

- 在控制流图构建完成之后, 根据上述匹配过程对内联函数块进行出入口位置匹配, 并加入到相应的哈希表中.

以表 3 中的 *foo* 内联函数块为例, 已经记录其调用类型为有参数而不接收返回值. 对于入口, 匹配规则 3, 迭代控制流图, 获取行号为 40 的程序点对应的最小 sid, 得到块入口位置为(40,99), 并加入到 blockEnterPos 哈希表; 对于出口, 匹配规则 4, *foo* 函数最后一条语句行号为 33, 迭代控制流图, 获取行号为 33 的程序点的后继节点, 行号为–1 的后继节点(–1,109)即为块出口位置, 并加入到 blockExitPos 哈希表.

(3) 变量类型转换

对于待消除的变量而言, 其类型是多样的, 而程序环境降维优化过程需要与数值抽象域库 APRON 进行交互, 于是, 需要将待删除的变量转换为 APRON 的变量类型(仅支持整型和浮点型). 对于类型转换, 下面介绍其执行过程的基本步骤.

1) 对每个变量集合, 去除其中未在程序分析过程中使用的变量;
2) 创建一个空的哈希表 *blockEnv*, 表的键为函数名, 表的值为环境(即 *Apron.Environment*, 由整型和浮点型两个变量数组构成), 以环境作为类型转换后结果的存储位置;
3) 对于一个变量集合, 先创建整型和浮点型两个数组, 再从中间表示文件中获取其变量信息列表; 接着, 根据变量信息对不同类型的变量使用不同的规则进行处理, 使其转化为整型或浮点型的变量(即 *Apron.Var*)并加入到相应的数组中. 如此重复, 直到处理完变量集合. 之后, 根据两个数组创建抽象环境;
4) 将函数名与对应的抽象环境加入到 *blockEnv*, 最终完成变量类型的转换.

不同类型的变量, 其转化规则各异, 下面对步骤 3)中提到的处理规则进行介绍.

- 对于整型和浮点型变量不做处理, 加入到对应的数组中;
- 对于指针类型变量, 考虑采用指向分析的方法(指向分析的方法可以描述指针的基地址和偏移, 从而防止出现诸如别名而误删的情形)做处理, 即使用一个整型变量刻画指针相对于基地址的偏移, 于是创建一个名为 *offset_x*(*x* 为指针变量名)的整型变量, 加入到对应的数组中;
- 对于结构体和联合体类型变量, 根据字段进行展开, 即把其中的每个字段视作一个变量, 根据字段的类型做处理, 并加到相应的数组中; 若遇到字段为结构体、联合体和数组等复杂数据结构, 需要递归处理, 我们设置递归层数小于 3;
- 对于数组类型变量, 有两种处理方式: 一是将整个数组聚合处理, 即将整个数组视为一个变量, 再根据数组内存储的数据类型做区分, 并加到相应的数组中; 二是将整个数组离散处理, 即把每个下标都视作一个变量, 根据数组中存储的类型做区分, 并加到相应的数组中; 若数组类型为复杂数据结构,

需要递归处理, 我们设置递归层数小于 3.

相应地, 在抽象域模块, 我们进行如下扩展.

- 对各内联函数块, 将前端模块记录的变量集合根据上述变量类型转换的步骤进行处理, 先转换为 APRON 中的环境类型, 再加入到相应的哈希表中.

以表 3 中的 *foo* 内联函数块为例, 已经记录其变量名及类型, 由于变量类型均为整型, 直接将其加入到对应的整型数组中, 为后续的环境构造做准备.

(4) 变量增加删除

对内联函数块而言, 环境中变量的增删, 除了找到块出入口的位置外, 还要在求解器中寻找合适的时机进行操作. 由于不动点迭代模块 Fixpoint 库中使用 *apply* 函数来计算有向边上所标注语句的迁移影响, 一个有向边由两侧顶点相连而成, 各顶点由(*line*,*sid*)二元组唯一确定, 这与匹配的内联函数块出入口位置契合, 于是考虑在 *apply* 中匹配顶点来进行环境中变量的增删. 变量的增删遵循一定的规则, 下面对增删规则进行介绍.

- 函数 *apply* 当前执行的程序点与内联函数块入口匹配, 判断内联函数块对应的中间变量集合是否在环境中: 若是, 则不操作; 否则, 将该变量集合加入到环境中;
- 函数 *apply* 当前执行的程序点与内联函数块出口匹配, 判断内联函数块对应的中间变量集合是否在环境中: 若不是, 则不操作; 否则, 将该变量集合从环境中删除.

相应地, 在求解器模块, 我们进行如下扩展:

- 对各内联函数块, 由出入口位置匹配结果在 *apply* 函数中判断当前程序点是否为内联函数块的出入口, 并根据上述增删规则对环境中的变量进行相应地增删.

以表 3 中的 *foo* 内联函数块为例, 已经记录其出入口位置, 当 *apply* 匹配到入口位置(40,99), 待消除变量的集合在环境中, 故不进行操作; 当 *apply* 匹配到出口位置(–1,109), 待消除变量的集合在环境中, 故将变量集合从环境中删除.

结合上述步骤(1)–(4)部分的内容, 我们总结面向内联函数块的程序环境降维优化方法的具体步骤, 并在算法 1 给出.

算法 1. 面向内联函数块的程序环境降维优化方法.

输入: C 语言程序;

输出: 降维优化前后环境中的变量数.

```
1  blockEnterPos={·};
2  blockExitPos={·};
3  blockEnv={·};  //创建 3 个空 map
4  while preProcess(IR) do  //根据中间表示做预处理并分析
5     tmpBlockVar.insert(funName,(varNameList,varTypList));
6     tmpBlockInfo.insert(funName,(line,callType));
7  for i=1~tmpBlockInfo.size(·)
8     enterLine,enterSid=enterRulePro(line,callType);  //参数取自 tmpBlockInfo, 下同
9     exitLine,exitSid=exitRulePro(line,callType);
10    blockEnterPos.insert(funName,(enterLine,enterSid));
11    blockExitPos.insert(funName,(exitLine,exitSid));
12 for i=1~tmpBlockVar.size(·)
13    intArray,realArray=convertTyp(varNameList,varTypList);  //参数取自 tmpBlockVar
14    tmpEnv=makeEnv(intArray,realArray);
15    blockEnv.insert(funName,tmpEnv);
16 beforeOpti=Apron.Environment.size(·);  //环境的大小即为变量个数
```

```
17  for i=1~transfer.size(·)
18    if enterMatch(enterLine,enterSid)
19      tmpEnv=find(blockEnterPos,name);
20        if varNotInEnv(tmpEnv[0])   //取出临时环境 tmpEnv 中的变量, 判断是否在环境中
21          addEnv(tmpEnv);
22    if exitMatch(exitLine,exitSid)
23      tmpEnv=find(blockExitPos,name);
24        if varInEnv(tmpEnv[0])
25          removeEnv(tmpEnv);
26  afterOpti=Apron.Environment.size(·);
27  return (beforeOpti,afterOpti)
```

算法的输入为 C 原程序, 其包含的文件数为一个或多个; 输出为降维优化前后环境中的变量数. 由于只删除了中间变量, 对最终的数值不变式不产生影响, 故不对优化前后的不变式进行比较, 不输出不变式. 其大致思路是:

- 根据各内联函数块, 去记录对应的待增删变量集合以及相应的出入口位置(增删时机);
- 随后, 在迁移边的迭代过程进行出入口位置的匹配: 若入口匹配成功且环境中不存在相应变量集合, 执行增加变量操作; 若出口匹配成功且环境中存在相应变量集合,执行删除变量操作.

值得注意的是: 由于待消除的变量来源于内联时创建的中间变量, 同时我们已经获取了内联函数块的出入口位置, 故在迭代求解过程中的合适的删除时机, 只要环境中存在待消除的变量, 就能通过我们的方法进行变量消除. 算法的具体解释如下.

- 第 1–3 行: 创建 3 个空 *map*, 具体的键值对细节在第 3.2 节内联函数块出入口位置匹配和变量类型转换部分中进行了介绍;
- 第 4–6 行: 预处理阶段. 第 4 行 IR 为中间表示文件, 第 5 行、第 6 行进行第 3.2 节变量选取部分中描述的记录操作, 为后续过程做准备;
- 第 7–11 行: 内联函数块出入口位置的匹配. 使用一定的规则在控制流图中做匹配, 细节在第 3.2 节内联函数块出入口位置匹配部分进行了介绍, 参数取自 *tmpBlockInfo*, 结果存入对应的 *map* 中;
- 第 12–15 行: 变量类型转换. 转换规则的细节在第 3.2 节变量类型转换部分中进行了介绍, 参数取自 *tmpBlockVar*, 结果存入对应的 *map* 中;
- 第 16 行: 记录降维优化前的变量数, 即环境的大小;
- 第 17–25 行: 增删变量过程, 在迁移边的迭代求解中进行. 第 18–21 行进行变量增操作: 第 18 行表示入口匹配判断, 第 20 行表示环境中不存在相应变量集合判断, 若两者均为真, 在环境中执行变量集合增加操作. 第 22–25 行: 第 22 行表示出口匹配判断, 第 24 行表示环境中存在相应变量集合判断, 若两者均为真, 在环境中执行变量集合删除操作. 出入口匹配的细节在第 3.2 节变量增加删除部分中进行了介绍;
- 第 26 行: 记录降维优化后的变量数, 即环境的大小;
- 第 27 行: 返回降维优化前后环境中的变量数.

4 实验及分析

本节通过在经典测试集上开展实验对原型工具 DRIP 的优化效果进行评价, 实验将围绕以下两个研究问题展开: (1) DRIP 的变量降维优化效果是否显著; (2) DRIP 降维优化能否有效降低抽象解释分析的时间和内存开销. 实验中 DRIP 配置情况: CIL 和 Fixpoint 库都使用默认配置, 抽象域选用 APRON 中实现的多面体抽象域 Polka. 本文采用的实验平台: Ubuntu14.04 操作系统, 8 GB RAM, Intel i5 CPU (1.6GHz, 四核)处理器. 实验所

用的测试用例来自 WCET Benchmarks[30], 该测试集广泛用于验证和比较不同类型的最坏运行时间(WCET)分析工具和方法. 选择该测试集的原因分为两点: 一是测试集中的单个测试用例独立成一个 C 文件, 且其来源广泛, 涵盖多种复杂数据类型和控制结构, 可以充分检验工具的处理能力; 二是测试集中大部分测试用例都包含函数调用, 这是内联函数块的基础, 便于验证降维优化方法的效果. WCET Benchmarks 包含 35 个测试用例, 本文选取其中 13 个进行实验, 其余测试用例未选取原因主要包括: 包含递归函数调用(递归函数的定义会使得内联方法在代码替换过程陷入死循环, 即内联的方法无法处理递归函数, 故我们不考虑包含递归函数的情况)、无函数调用等.

4.1 降维优化效果分析

表 4 给出了 DRIP 降维优化效果评价的实验结果, 第 2 列代码行数表示原程序大小; 第 3 列表示优化前的变量数, 即经过前端模块处理构建控制流图后环境中的变量数; 第 4 列表示优化后的变量数, 选取分析结束时主函数出口处程序点的环境中的变量数; 第 5 列表示分析结束时消除的临时变量数; 第 6 列表示分析结束时消除的局部变量数; 第 7 列减少率的计算方式为(优化前变量数–优化后变量数)/优化前变量数; 第 8 列表示优化前后的精度对比(=表示精度一致).

表 4 DRIP 降维优化效果

程序名	代码行数	变量数(前)	变量数(后)	消除临时变量数	消除局部变量数	减少率(%)	优化前后精度对比
bs.c	66	15	10	1	4	33.3	=
bsort100.c	69	18	10	2	6	44.4	=
cover.c	231	17	8	3	6	52.9	=
duff.c	33	32	9	5	18	71.9	=
edn.c	198	75	13	31	31	82.7	=
fdct.c	147	31	9	2	20	71.0	=
fibcall.c	20	13	7	1	5	46.2	=
janne_complex.c	26	12	9	2	1	25.0	=
jfdctint.c	166	29	10	0	19	65.5	=
ns.c	415	14	9	1	4	35.7	=
qsort-exam.c	69	21	9	1	11	57.1	=
select.c	68	19	8	2	9	57.9	=
ud.c	61	24	16	2	6	33.3	=

从表 4 结果可以看出: 对于这些测试用例, DRIP 降维优化能够在未使用优化的基础上最多减少了 82.7%的变量数, 最少也减少了 25.0%的变量, 平均减少了 52.1%. 其中, edn 等 7 个程序都减少了 50%以上. 通过对原程序进行人工走查发现, 这些程序中全局变量和主函数中局部变量较少, 被调函数中局部变量较多, 且有多个函数调用. 对于此类程序, DRIP 降维优化效果显著; 反之, 如 janne_complex.c 等程序, 它们包含的全局变量和主函数中的局部变量明显多于被调函数中局部变量, 且被调函数通常只有一次调用, DRIP 对于这类程序有降维优化效果但不够显著. 同时, 按照变量类型来看, 函数局部变量的消除效果优于临时变量的消除效果, 这是由于被调函数的局部变量数量往往大于形参数量. 本文的工作并未涉及程序分析的结果, 即程序不变式, 但我们对比了优化前后的不变式, 结果显示, 中间变量的消除不影响不变式精度.

4.2 降维优化提升抽象解释分析时空效率分析

(1) 运行时间分析

表 5 给出了 DRIP 提升抽象解释分析时间效率的实验结果, 第 2 列和第 3 列分别表示不使用和使用本文优化方法时抽象解释分析产生程序不变式的时间开销; 第 4 列减少率是指优化后时间相对于优化前的时间降低比例, 即(不使用优化时间开销–使用优化时间开销)/不使用优化时间开销; 第 5 列表示使用优化方法本身的时间开销; 第 6 列比率是指优化阶段的时间开销相对于使用优化方法进行程序分析的时间开销的百分比, 即优化阶段耗时/使用优化总时间. 需要说明的是: 对于每个测试用例, DRIP 工具分析时除了是否使用本文优化, 第 2 列和第 3 列结果对应的实验配置完全一样, 而本文优化时减少的变量都是内联引入的中间变量, 不影响抽象解释分析产生的不变式精度, 故接下来的实验中不再对比优化前后的分析精度.

从表 5 可以看出: 运行时间减少率最高可达 59.6%(cover.c), 7/13 的测试用例分析运行时间减少率高于

25%. 通过对这些程序进行深入分析, 发现运行效率提升的主要原因是这些程序中全局变量个数少, 变量个数对运行效率影响大, 删去中间变量后, 使得不动点迭代过程中的每个操作时间开销低(因为不动点迭代过程中使用的抽象域操作的复杂度与变量数直接相关), 从而提升了不动点迭代效率. 但在这些测试用例中, 仍然有 3 个程序(bsort100.c, jfdctint.c 和 qsort-exam.c)的分析耗时不增反降. 通过分析原程序发现: 上述程序变量类型复杂, 在使用优化方法对其进行类型转换时耗费大量时间, 使得整个分析过程的时间增加, 降维优化方法带来的不动点迭代效率提升比不上变量类型转换带来的效率降低, 故整体的分析时间反而增加. 同时, 除少数几个测试用例外, 优化方法本身的耗时相对于使用优化方法进行程序分析的耗时比率均小于 10%, 说明优化方法本身耗时所占比重低.

表 5 DRIP 运行时间

程序名	不使用优化总时间(s)	使用优化总时间(s)	减少率(%)	优化阶段耗时(s)	优化阶段耗时比率(%)
bs.c	0.3	0.18	40.0	0.02	11.1
bsort100.c	0.524	0.548	–4.6	0.069	12.5
cover.c	6.024	2.432	59.6	0.097	4.0
duff.c	3.468	1.728	50.2	0.027	1.6
edn.c	29.176	21.148	27.5	0.383	1.8
fdct.c	4.008	2.648	33.9	0.065	2.4
fibcall.c	0.244	0.204	16.4	0.019	9.1
janne_complex.c	0.368	0.264	28.3	0.019	7.1
jfdctint.c	4.08	4.496	–10.2	0.128	2.8
ns.c	0.64	0.544	15.0	0.021	3.8
qsort-exam.c	1.764	1.776	-0.7	0.058	3.3
select.c	1.144	0.772	32.5	0.034	4.4
ud.c	1.988	1.66	16.5	0.048	2.9

(2) 内存消耗分析

表 6 给出了 DRIP 提升抽象解释分析空间效率的实验结果, 第 2 列和第 3 列分别表示不使用和使用本文优化方法时抽象解释分析产生程序不变式的空间开销, 即内存使用量; 第 4 列减少率是指优化后内存相对于优化前的内存使用降低比例, 即(不使用优化内存开销–使用优化内存开销)/不使用优化内存开销.

表 6 DRIP 内存消耗

程序名	不使用优化内存(KB)	使用优化内存(KB)	减少率(%)
bs.c	2 808	2 664	5.1
bsort100.c	8 952	6 104	31.8
cover.c	53 720	34 180	36.4
duff.c	142 128	96 672	32.0
edn.c	398 768	268 328	32.7
fdct.c	91 288	88 380	3.2
fibcall.c	3 752	3 580	4.6
janne_complex.c	4 636	4 500	2.9
jfdctint.c	99 356	91 204	8.2
ns.c	8 076	7 932	1.8
qsort-exam.c	68 908	67 072	2.7
select.c	49 440	45 668	7.6
ud.c	52 820	52 688	0.2

表 6 实验结果表明, 使用优化的内存开销均小于不使用优化的内存开销. 这说明降维优化方法能够有效降低抽象解释分析所需的内存消耗. 具体来看, 对于 cover.c, bsort100.c, duff.c, edn.c 这 4 个程序, 其内存开销减少率均在 30%以上. 通过对原程序人工走查发现, 这些测试用例存在多个被调函数, 故降维优化对内存消耗效果明显. 而 ud.c 等 9 个程序, 其减少率均在 10%以下. 查看原程序发现, 这些测试用例仅有一个被调函数, 且被调函数是在主函数出口附近被调用, 故变量降维优化对于降低内存消耗带来的增益不大(这是因为降维优化方法只能减少被调函数调用结束之后和主函数结束之前的内存消耗).

5 相关工作

5.1 基于内联的过程间分析

当前, 关于内联方法的研究主要针对递归程序的分析处理. 在不考虑效率的情形下, 内联可以解决除递归调用外的过程调用分析. 对于递归调用, 内联方法若不加限制, 则分析会进入死循环. 一种做法是将其展开至一定的层数并忽略后续的调用, 该方法已在工具 CBMC[25], ESBMC[26]和 SMACK[27]中实现. 由于不能保证后续未展开的递归中是否有错误, 该方法是不可靠的. 同时, 存在无界的框架如工具 CPAChecker[28], 它基于不同域进行函数内联的分析[35]. 本文主要针对提升内联方法在引入大量中间变量情况下的分析效率.

5.2 抽象解释框架下过程间分析优化

在程序的全局抽象解释分析过程中, 如何减少抽象环境中冗余变量的维数、提升程序分析的效率, 一直是研究的热点. Astree[17]使用基于语义内联的方法, 提升过程间分析的效率, 但是不能处理递归程序和 goto 语句. 除此之外, 基于变量可达性的局部性原理的研究[36−39]实现了对部分程序块的冗余计算的消除. 在这基础上, Oh 等人采取更为优化的基于变量访问的局部性原理技术[40,41], 进一步降低当前抽象环境中所涉及的变量维数, 并根据数据流依赖关系, 分析时只关注相关语句的迁移函数, 从而降低抽象状态的存储开销和传播开销. 值得一提的是: Oh 等人还基于该思想进一步提出了一种通用的全局稀疏分析框架, 在不损失分析精度的前提下, 能够显著降低时空开销, 并在商业化静态分析工具 Sparrow[15]上进行了应用, 取得了显著的可扩展性提升效果, 将分析代码的规模提升到了百万行[42]. 需要注意到: 该稀疏分析框架基于控制流与数据流的依赖关系分析, 使用更为通用的方法, 可以减少冗余的函数调用分析, 但预分析的代价和对分析器的修改工作量都比较大; 而本文针对函数内联的方法, 致力于动态地消除函数内联所带来的冗余中间变量对程序分析的影响, 没有预分析阶段, 实现代价也比较低.

6 总结与展望

本文针对函数内联过程间分析方法存在的引入中间变量多、空间开销大、分析效率低的问题, 在基于抽象解释的程序分析场景下, 提出了一种面向内联函数块的程序环境降维优化方法. 该方法针对内联函数后的程序代码, 分析确定不同程序点处需维护的程序环境(相关变量集合), 而不是所有程序点共享同一全局程序环境, 从而实现程序状态的降维. 本文阐述了该方法的原理与算法以及基于该方法实现的工具 DRIP. 最后, 在 WCET Benchmarks 开展了实验. 实验结果表明: 本文方法在中间变量消除上取得的效果良好, 甚至在某些测试集上能减少一半以上的变量, 并在一定程度上降低了分析过程的时空开销.

本文方法和工具仍然存在一些不足: 目前只以整个函数体为单元对分析的程序环境进行降维, 下一步将研究面向降维的程序块划分以及以程序块为单元的程序环境降维方法. 同时, 对一些函数指针等函数调用类型, 工具不能很好地支持. 在未来的工作中, 我们还将继续完善场景和算法, 并改进工具的实现, 进一步提高工具的效率.

References:

[1] Chen HW, Wang J, Dong W. High confidence software engineering technologies. Chinese Journal of Electronics, 2003, S1: 1933−1938 (in Chinese with English abstract).

[2] Zhang J, Zhang C, Xuan JF, *et al*. Recent progress in program analysis. Ruan Jian Xue Bao/Journal of Software, 2019, 30(1): 80−109 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5651.htm [doi: 10.13328/j.cnki.jos.005651]

[3] Cousot P, Cousot R. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proc. of the 4th ACM SIGACT-SIGPLAN Symp. on Principles of Programming Languages. 1977. 238−252.

[4] Cousot P, Cousot R. Systematic design of program analysis frameworks. In: Proc. of the 6th ACM SIGACT-SIGPLAN Symp. on Principles of Programming Languages. 1979. 269−282.

[5] Allen FE. Interprocedural analysis and the information derived by it. In: Proc. of the IBM Germany Scientific Symp. Series. Berlin, Heidelberg: Springer, 1974. 291−321.

[6] Waddell O, Dybvig RK. Fast and effective procedure inlining. In: Proc. of the Int'l Static Analysis Symp. Berlin, Heidelberg: Springer, 1997. 35−52.

[7] Cousot P, Cousot R. Static determination of dynamic properties of programs. In: Proc. of the 2nd Int'l Symp. on Programming. Paris: Dunod, 1976. 106−130.

[8] Miné A. The octagon abstract domain. Higher-order and Symbolic Computation, 2006, 19(1): 31−100.

[9] Cousot P, Halbwachs N. Automatic discovery of linear restraints among variables of a program. In: Proc. of the 5th ACM SIGACT-SIGPLAN Symp. on Principles of Programming Languages. 1978. 84−96.

[10] Jeannet B, Miné A. Apron: A library of numerical abstract domains for static analysis. In: Proc. of the Int'l Conf. on Computer Aided Verification. Berlin, Heidelberg: Springer, 2009. 661−667.

[11] Singh G, Püschel M, Vechev M. A practical construction for decomposing numerical abstract domains. In: Proc. of the ACM on Programming Languages. 2017. 1−28.

[12] Bagnara R, Hill PM, Zaffanella E. The Parma Polyhedra Library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. Science of Computer Programming, 2008, 72(1-2): 3−21.

[13] Cousot P, Cousot R, Feret J, *et al*. The ASTRÉE analyzer. In: Proc. of the European Symp. on Programming. Berlin, Heidelberg: Springer, 2005. 21−30.

[14] Kirchner F, Kosmatov N, Prevosto V, Signoles J, Yakobowski B. Frama-C: A software analysis perspective. Formal Aspects of Computing, 2015, 27(3): 573−609.

[15] Sparrow static analyzer. http://ropas.snu.ac.kr/sparrow/

[16] Interproc. http://pop-art.inrialpes.fr/interproc/interprocweb.cgi

[17] Blanchet B, Cousot P, Cousot R, *et al*. A static analyzer for large safety-critical software. In: Proc. of the ACM SIGPLAN 2003 Conf. on Programming Language Design and Implementation. 2003. 196−207.

[18] Miné A, Delmas D. Towards an industrial use of sound static analysis for the verification of concurrent embedded avionics software. In: Proc. of the 2015 Int'l Conf. on Embedded Software (EMSOFT). IEEE, 2015. 65−74.

[19] Emami M, Ghiya R, Hendren LJ. Context-sensitive interprocedural points-to analysis in the presence of function pointers. ACM SIGPLAN Notices, 1994, 29(6): 242−256.

[20] Shivers O. Control-flow analysis of higher-order languages [Ph.D. Thesis]. Pittsburgh: Carnegie Mellon University, 1991.

[21] Lhoták O, Hendren L. Relations as an abstraction for BDD-based program analysis. ACM Trans. on Programming Languages and Systems (TOPLAS), 2008, 30(4): 1−63.

[22] Sridharan M, Fink SJ. The complexity of Andersen's analysis in practice. In: Proc. of the Int'l Static Analysis Symp. Berlin, Heidelberg: Springer, 2009. 205−221.

[23] Yorsh G, Yahav E, Chandra S. Generating precise and concise procedure summaries. In: Proc. of the 35th Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages. 2008. 221−234.

[24] Ruf E. Context-insensitive alias analysis reconsidered. ACM SIGPLAN Notices, 1995, 30(6): 13−22.

[25] Clarke EM, Kröning D, Lerda F. A tool for checking ANSI-C programs. In: Proc. of the Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems. Berlin, Heidelberg: Springer, 2004. 168−176.

[26] Gadelha MR, Monteiro FR, Morse J, Cordeiro LC, Fischer B, Nicole DA. ESBMC 5.0: An industrial-strength C model checker. In: Proc. of the 33rd ACM/IEEE Int'l Conf. on Automated Software Engineering. 2018. 888−891.

[27] Rakamarić Z, Emmi M. SMACK: Decoupling source language details from verifier implementations. In: Proc. of the Int'l Conf. on Computer Aided Verification. Berlin, Heidelberg: Springer, 2014. 106−113.

[28] Beyer D, Keremoglu ME. CPAchecker: A tool for configurable software verification. In: Proc. of the Int'l Conf. on Computer Aided Verification. Berlin, Heidelberg: Springer, 2011. 184−190.

[29] Cuoq P, Kirchner F, Kosmatov N, *et al*. In: Proc. of the Int'l Conf. on Software Engineering and Formal Methods. Berlin, Heidelberg: Springer, 2012. 233−247.

[30] Gustafsson J, Betts A, Ermedahl A, *et al*. The Mälardalen WCET benchmarks: Past, present and future. In: Proc. of the 10th Int'l Workshop on Worst-Case Execution Time Analysis (WCET 2010). 2010. 137−147.

[31] Count lines of code (cloc). https://github.com/AlDanial/cloc

[32] CIL: Infrastructure for C. https://people.eecs.berkeley.edu/~necula/cil/

[33] Fixpoint. http://pop-art.inrialpes.fr/people/bjeannet/bjeannet-forge/fixpoint/index.html

[34] Xu Z, Kremenek T, Zhang J. A memory model for static analysis of C programs. In: Proc. of the Int'l Symp. on Leveraging Applications of Formal Methods, Verification and Validation. Berlin, Heidelberg: Springer, 2010. 535−548.

[35] Müller P, Vojnar T. CPAlien: Shape analyzer for CPAchecker. In: Proc. of the Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems. Berlin, Heidelberg: Springer, 2014. 395−397.

[36] Chen LL, Harrison III WL. An efficient approach to computing fixpoints for complex program analysis. In: Proc. of the 8th Int'l Conf. on Supercomputing. 1994. 98−106.

[37] Gotsman A, Berdine J, Cook B. Interprocedural shape analysis with separated heap abstractions. In: Proc. of the Int'l Static Analysis Symp. Berlin, Heidelberg: Springer, 2006. 240−260.

[38] Marron M, Hermenegildo M, Kapur D, *et al*. Efficient context-sensitive shape analysis with graph based heap models. In: Proc. of the Int'l Conf. on Compiler Construction. Berlin, Heidelberg: Springer, 2008. 245−259.

[39] Might M, Shivers O. Improving flow analyses via ΓCFA: Abstract garbage collection and counting. In: Proc. of the 11th ACM SIGPLAN Int'l Conf. on Functional Programming. 2006. 13−25.

[40] Oh H, Brutschy L, Yi K. Access analysis-based tight localization of abstract memories. In: Proc. of the Int'l Workshop on Verification, Model Checking, and Abstract Interpretation. Berlin, Heidelberg: Springer, 2011. 356−370.

[41] Oh H, Yi K. Access-based localization with bypassing. In: Proc. of the Asian Symp. on Programming Languages and Systems. Berlin, Heidelberg: Springer, 2011. 50−65.

[42] Oh H, Heo K, Lee W, *et al*. Design and implementation of sparse global analyses for C-like languages. In: Proc. of the 33rd ACM SIGPLAN Conf. on Programming Language Design and Implementation. 2012. 229−238.

附中文参考文献:

[1] 陈火旺, 王戟, 董威. 高可信软件工程技术. 电子学报, 2003, S1: 1933−1938.

[2] 张健, 张超, 玄跻峰, 等. 程序分析研究进展. 软件学报, 2019, 30(1): 80−109. http://www.jos.org.cn/1000-9825/5651.htm [doi: 10.13328/j.cnki.jos.005651]

陈涛清(1997－), 男, 硕士生, 主要研究领域为程序分析.

范广生(1997－), 男, 博士生, 主要研究领域为程序分析与验证, 抽象解释.

尹帮虎(1989－), 男, 博士, 讲师, 主要研究领域为程序分析与验证, 系统建模与仿真.

陈立前(1982－), 男, 博士, 副教授, CCF 高级会员, 主要研究领域为程序分析与验证, 抽象解释.

王戟(1969－), 男, 博士, 教授, 博士生导师, CCF 杰出会员, 主要研究领域为软件方法学, 软件分析与验证, 并行与分布计算.