高效而精确的锁别名分析方法①

陈　露, 顾乃杰, 黄　理, 杜云开

(中国科学技术大学 计算机科学与技术学院, 合肥 230027)
(中国科学技术大学 安徽省计算与通信软件重点实验室, 合肥 230027)
(中国科学技术大学 先进技术研究院, 合肥 230027)

摘　要: 锁别名分析能够得到锁指针变量的指向信息, 有效的锁别名分析可以更好地辅助数据竞争分析和死锁分析. 现有锁别名分析往往采用保守的方式处理, 进而影响分析结果的准确性. 针对这一问题, 提出了一种锁别名分析方法, 该方法首先使用 GCC 插件获取 SSA 形式的中间代码, 然后对中间代码进行预处理以获得与锁、函数指针操作相关的语句, 最后对预处理后的程序使用本文提出的 FP_LOCK 算法进行准确的流敏感、上下文敏感分析. 实验结果表明该方法能精确地确定锁别名, 并且经过预处理后的 FP_LOCK 算法对分析大程序平均有 9.95 倍的加速比.

关键词: 锁别名分析; SSA; 中间代码; FP_LOCK; 流敏感; 上下文敏感

Efficient and Accurate Lock Alias Analysis Method

CHEN Lu, GU Nai-Jie, HUANG Li, DU Yun-Kai

(School of Computer Science and Technology, University of Science and Technology of China, Hefei 230027, China)
(Anhui Province Key Laboratory of Computing and Communication Software, Hefei 230027, China)
(Institute of Advanced Technology, University of Science and Technology of China, Hefei 230027, China)

Abstract: Lock alias analysis can obtain the precise lock pointer information. Efficient lock alias analysis can make a good contribution to data races analysis and deadlock analysis. However, existing lock alias analyses often use a conservative approach, which impacts the accuracy of analysis. This paper proposes a lock alias analysis method. This method firstly gains SSA-form intermediate code by a GCC plugin. Then it preprocesses the intermediate code to obtain the statements related to function pointer and lock operation. Based on this we present the flow-sensitive and context-sensitive function pointer and lock analysis algorithm (referred as FP_LOCK algorithm). Experimental results show that this method has well precision and advances 9.95 times on average compared with the original algorithm.

Key words: lock alias analysis; SSA; intermediate code; FP_LOCK; flow-sensitive; context-sensitive

1　引言

随着多线程程序的广泛应用, 多线程程序中对锁的不恰当使用容易造成程序错误, 如数据竞争、程序死锁等. 因此很多现有工具使用静态分析技术来提前发现多线程程序中的错误. 典型工具有 Lockset[1]、RacerX[2]和 Gadara[3,4]等, Lockset、RaceX 确定一个线程对共享变量进行访问时维护的锁集合; Gadara 利用 Petri 网对多线程程序进行建模并进行相关的死锁检测和避免, 建模时需要确定每个加锁或者解锁操作的锁. 然而这些工具大部分使用变量或者参数的类型来确定锁的语义, 这可能会导致分析时引入不必要的误判. 例如, 假设一个函数内进行了两次加锁操作, 并且锁变量是同一种类型但是处在不同的内存地址, 如果保守的用类型信息来确定这两个锁, 那么分析时误以为对同一个锁进行操作, 这违背了程序的语义. 因此需要利用程序静态分析技术来确定锁变量的值.

① 基金项目:安徽省自然科学基金(1408085MKL06);高等学校学科创新引智计划(B07033)
收稿时间:2015-05-07;收到修改稿时间:2015-06-08

© 中国科学院软件研究所 http://www.c-s-a.org.cn

由于锁变量在程序中是以指针的形式存在, 因此如果要静态确定锁变量的值, 需要进行指针别名分析. 按照分析的精度来区分, 现有的指针别名分析主要有流不敏感和流敏感、上下文不敏感和上下文敏感分析、域敏感和域不敏感分析. 如果利用流不敏感、上下文不敏感分析算法[5], 则分析的准确度低; 如果利用比较精确的上下文敏感、流敏感分析[6,7], 则分析时间复杂度高. 文献[8]针对 LLVM 平台提出一种上下文敏感、流敏感的锁别名分析算法, 分析时需要先把源程序在 LLVM 平台上进行编译, 然而现有多线程程序大部分使用 Makefile 文件和 GCC 编译器编译目标程序, 并且LLVM存在与GCC产生冲突的编译选项, 因此该算法需要建立在源程序正确通过 LLVM 编译的前提下.

针对上述问题, 本文在 GCC 平台上, 实现了一个针对 C 多线程程序的锁别名分析方法, 该方法包括三个部分: 首先利用 GCC 插件提取源程序的中间代码; 然后对中间代码进行预处理, 得到程序中与锁、函数指针信息相关的操作语句; 最后使用本文提出的 FP_LOCK 算法对预处理后的中间代码进行上下文敏感、流敏感分析. 与已有的研究相比, 本文能够精确地在 GCC 平台上进行锁别名分析, 并且可扩展性好, 分析大型程序的时间复杂度低.

本文剩余部分组织如下: 第 1 节介绍背景, 第 2 节描述多线程程序的预处理过程, 第 3 节介绍具体的上下文敏感、流敏感分析过程, 第 4 节通过实验证明 FP_LOCK 算法的高效, 第 5 节结束语.

2 背景

2.1 控制流图和 SSA 形式中间代码

控制流图(CFG)是由基本块和边构成, 基本块的内容表示程序中的顺序执行语句, 边表示基本块之间的控制流向. CFG 中包括特殊的入口基本块和出口基本块. CFG 是编译器进行数据流分析的基础.

SSA[9]即静态单一赋值, 表示赋值时每个变量只有唯一定义. 并且 SSA 通过重新分配变量名来支持每个变量需要扩展出多个不同实例的情况. 当在控制流路径的汇聚点处需要汇聚同一变量的多个实例时, SSA 在该汇聚点处插入 PHI 函数, PHI 函数的输入参数是进入汇聚点的不同实例.

2.2 GCC 插件

GCC 插件[10]是 GCC-4.5.0 版本后可用来改变程序行为的动态链接库. 它提供可注册的事件列表和事件对应的回调函数接口. 插件提供的事件贯穿于整个 GCC 编译的过程, 包括词法分析、语法分析和中间代码优化等阶段.

本文利用 GCC 提供的插件功能提取 GCC 生成的类型、初始化语句和 CFG 信息, 并且 CFG 中的语句是以 SSA 形式存在的, 然后将提取的信息以格式化的形式打印到文件中, 以便为后继分析作准备.

GCC 插件根据 GCC 提供的事件功能, 注册用于获得类型和 CFG 的回调函数, 具体流程包括两个部分: (1)当 GCC 内部解析完类型定义和初始化信息后, 调用获取文件中类型定义和初始化信息的回调函数, 类型信息包括结构体、函数指针等, 初始化信息包括程序中对全局变量进行初始化的语句; (2)在 GCC 内部生成了 SSA 形式的中间代码后, 调用生成函数信息的回调函数, 函数信息包括返回值、函数名、参数列表、局部变量列表、基本块中 SSA 形式的中间代码等信息.

2.3 别名图

别名分析之前需要确定指针指向信息的表示方法. 别名图[11]是用来表示指针别名关系的一种表示方式. 别名图是由结点集合和边集合构成, 其中结点表示内存位置, 边表示结点之间的指向关系. C 程序中使用的内存可以抽象地分为两类: 为变量分配的内存和堆分配的内存.

为变量分配的内存与变量的类型有关, 包括标量类型、数组类型和结构体类型. (1)标量类型: 直接以变量的名字表示为变量分配的内存; (2)数组类型: 使用数组的头元素表示这个数组元素; (3)结构体类型: 使用不同的结点表示结构体中不同的成员域. 另外, 针对递归数据结构如单链表、双链表、树、图等, 本文识别这样的数据结构, 并把这样的结构看做一个整体并用一个唯一的结点表示.

对于堆分配的内存, 本文以调用堆分配内存的调用点来表示堆分配内存的名字[12].

2.4 别名图的生成过程

别名分析时需要获得所有影响分析结果的指针赋值语句. 根据 GCC 生成 SSA 形式的中间代码特点, 本文将要分析的指针赋值操作语句分为以下 6 种:

(1)取地址操作, 如 p=&A 形式;

(2)简单赋值操作, 如 p = q 形式;

© 中国科学院软件研究所 http://www.c-s-a.org.cn

(3)写操作，如*p = q 形式；

(4)读操作，如 p = *q 形式；

(5)域写操作，如 p->f = q 形式；

(6)域读操作，如 p = q->f 形式.

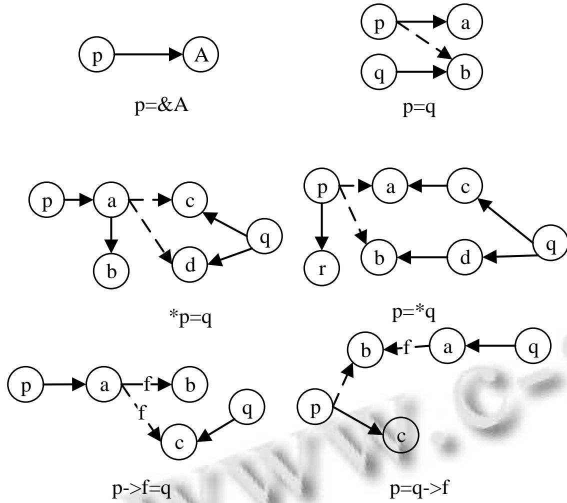


图 1　别名图的建立过程

图 1 表示了指针赋值操作对应别名图的建立过程. 图 1 中的实边表示在语句执行前的状态，虚边表示语句执行后新增加的边. 如果按照流不敏感分析，那么赋值后左值指向了包括实边和虚边指向的对象；如果按照流敏感分析，那么赋值后左值指向了虚边指向的对象. 通过别名图可以直观地得到了指针之间的别名关系，并且还可以获得指针最后所指向的对象.

3　程序预处理

要进行锁别名分析，可以直接使用现有的流敏感、上下文敏感(FSCS)算法进行精确分析. 然而现有的 FSCS 算法是针对程序中的所有语句进行分析的，包括与锁别名无关的语句. 如果能够去除这些对锁别名无关的语句，并且保留与锁别名有关的语句，则可以加快FSCS算法的分析速度. 另外，由于函数指针能够影响函数调用关系，进而影响 FSFC 算法分析的准确度，因此还需要在程序中保留与函数指针有关的操作语句. 为了便于下文说明，用 FP_LOCK 表示锁、函数指针.

本文首先对每个函数进行流不敏感、域不敏感分析，按照图 1 处理指针赋值语句并建立别名图. 如果别名图建立过程中出现环，那么将别名图规约成一个由强连通分量组成的有向无环图(SCC-DAG)[13]. 最终得到粗粒度的别名图. 然后根据这个别名图中结点之间的指向关系，找到与 FP_LOCK 有关的操作语句. 以图 2 为例：

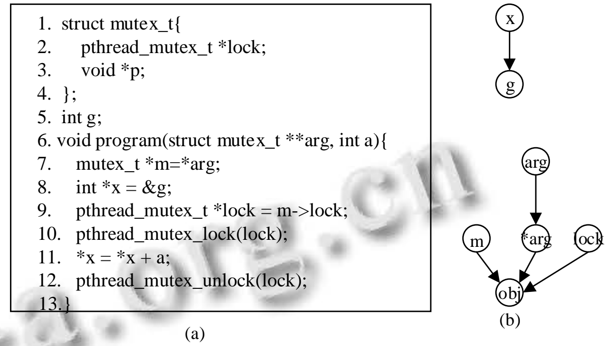


图 2　预处理分析的示例

根据流不敏感分析得到如图 2(b)的别名图. 进一步观察图 2(b)，图 2(b)的别名图分为两组，这两组之间不存在指向关系，并且每组中的结点之间存在层次的指向关系，其中上一层与下一层之间的关系表示上一层的指针变量指向下一层的对象，同一层之间的关系表示指向同一个下一层的对象. 假设要得到与锁指针变量 lock 有关的操作语句，首先找到 lock 指针变量所在的分组，然后从该组别名图中找到所有影响 lock 变量指向集的语句. 寻找的过程如下：与 lock 变量同一层的别名图中，寻找所有赋值给 lock 变量的赋值语句，例如图 2(a)第 9 行中 lock=m->lock，并且任何改变 m 或者*m 指向关系的赋值语句都会改变 lock 的指向集，因此需要考虑任何对*m 或者 m 进行赋值的操作语句，例如图 2(a)中第 7 行. 最后只有第 7、9 行影响 lock 的指向关系，而其他赋值语句对 lock 的指向关系不影响.

上述分析过程对应算法 1，给定一个粗粒度的别名图和关心的指针变量 x，在分析时需要维护一个集合 V，初始时 V 只有一个元素 x，然后每次循环时取 V 中的元素 p，并在分析程序中寻找所有改变 p 指向关系的赋值语句，寻找的规则为：

(1)由于 p 的值可以通过被 q 赋值来改变，如 p=q 的形式，因此把 q 加入集合 V 中，并标记赋值语句 p=q.

(2)存在指针 q，q 在别名图中指向 p，并且存在*q=r 形式的赋值语句，那么把*q、q 和 r 都加入到集合 V 中，并标记赋值语句*q=r.

利用算法 1 对每个函数进行分析，最后得到函数

© 中国科学院软件研究所 http://www.c-s-a.org.cn

内带有标记的赋值语句. 不带标记的赋值语句表示接下来不对该语句进行分析.

算法 1. 标记与 FP_LOCK 有关的赋值语句
输入: 粗粒度的别名图和关心的指针变量 x
输出: 标记所有影响 x 的赋值语句

```
1.   flag=true,V={x};
2.   while(flag){
3.     flag = false;
4.     p = 取 V 中的元素;
5.     if (存在 p=q 形式的赋值语句, 并且 q 是一个
          指针变量, q 不属于 V){
6.        把 q 加入集合 V 中,并标记语句 p=q.
7.        flag = true;
8.     }
9.    if(存在指针变量 q, q 在别名图中指向 p, 并且
        存在*q=r 形式的赋值语句){
10.       把*q、q 和 r 都加入到集合 V;
11.       flag=true;
12.   }
```

4 FP_LOCK算法

通过第 2 节的预处理后, 此时要分析的程序规模比较小, 这样可以对 FP_LOCK 信息进行精确地上下文敏感、流敏感分析, 称该过程的分析算法为 FP_LOCK 算法.

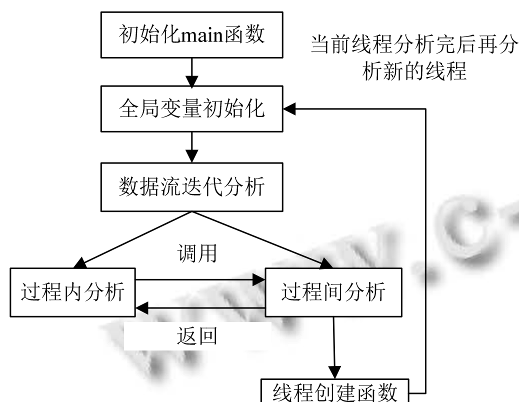


图 3 上下文敏感、流敏感分析框架

FP_LOCK 算法处理多线程程序过程包括两个部分:

(1)首先自顶向下地从每个线程的程序入口函数开始分析, 包括全局变量初始化、过程内分析和过程间分析. 分析时先从 main 函数开始, 并且在分析时动态地维护一个调用栈, 在进入函数内时压栈, 当过程内分析完后出栈, 调用栈可以用来判断函数是否存在递归分析的情况; 同时在每个线程内维护所有对全局变量进行赋值操作的语句, 形成一个线程摘要, 以便为接下进行跨线程间分析做准备. 当遇到创建线程的函数时记录线程的入口函数信息, 并在当前线程分析完后, 继续对未分析的线程进行分析. 整个分析流程如图 3 所示.

(2)当分析完所有线程后进行跨线程间分析, 主要对全局变量的赋值信息进行跨线程间传递.

4.1 全局变量初始化

全局变量初始化阶段记录程序(线程)开始分析时所有对全局变量进行初始化的语句, 包括对函数指针和锁变量的初始化语句. 分析时首先找到所有对全局变量进行初始化操作的语句, 然后将初始化语句的处理转换为对应赋值语句的处理, 关于赋值语句的别名图建立过程可以通过接下来的过程内分析获得. 这样最终得到全局变量的初始别名图.

4.2 过程内分析

过程内分析是一个迭代的执行过程, 它依次遍历 CFG 中的每条语句, 计算每条语句对指针指向信息的影响, 直到整个过程内的别名信息达到稳定为止. 算法 2 用来描述整个过程内分析的过程.

算法 2. 迭代的过程内分析
输入: 输入参数的指向信息
输出: 函数返回时参数和全局变量的指向信息

```
1.   while(flag){
2.     flag = false;
3.     for each bb in CFG{
4.        for stmt in bb{
5.          if stmt is ASSIGN:赋值处理;
6.          if stmt is CALL:过程间分析;
7.          if stmt is PHI: 对输入信息进行合并.
8.        }
9.        if(存在对别名图的更新操作)
10.         flag = true;
11.    }
12.  }
```

算法 2 主要分析三种类型的语句: 赋值语句、函数调用语句和 PHI 函数. 遇到函数调用语句则进入过程间分析, 因此本节主要讨论赋值语句和 PHI 函数.

赋值语句是过程内分析最核心的部分, 它包括左

© 中国科学院软件研究所 http://www.c-s-a.org.cn

值和右值，左值表示变量或者对象的内存地址，右值表示求解表达式得到的值. 赋值语句按照指针赋值和非指针赋值分开处理:

1)非指针赋值: 主要考虑结构体变量赋值，本文将结构体变量之间的赋值转换为对结构体内部成员变量之间的赋值，如果某个成员变量是结构体，那么继续展开直到左值表示一个标量对象.

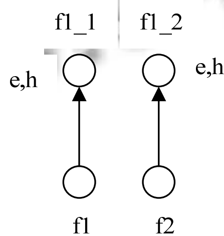
2)指针赋值: 指针赋值语句可以统一按照图 1 的方式处理. 由于本文考虑流敏感的方式处理，因此在处理左值表达式时需要根据左值指向的对象数目分别处理，如果左值指向一个对象结点，那么进行强更新，如果指向了多个对象结点，那么对每个指向的对象分别进行弱更新操作.

对于 PHI 函数，则将输入参数中所有指针类型参数的指向信息进行合并，并向下传递.

4.3 过程间分析

当遇到函数调用语句时进入过程间分析. 对于同一个被调用函数 callee，由于 callee 可以存在不同的调用点，并且在每个调用点处指针类型的实参的指向信息可能不同. 为了方便将不同调用点的指向信息传递到被调用函数，本文引入扩展变量[7]抽象地表示 callee 中指针类型的全局变量和形参的指向信息，扩展变量包括两种情况: (1)对于一个多级指针 p，引入扩展变量 p_i, i 表示一个 p 的解引用次数，例如一个指针变量 int **p, *p 用变量 p_1 表示，**p 用变量 p_2 表示，并且 p 指向了 p_1, p_1 指向了 p_2. (2)对于一个结构体变量 s，结构体有 n 个成员 f1,…,fn，则引入扩展变量 s.f1,...,s.fn 共 n 个变量.

```
1.  foo(){                  bar(f1, f2){
2.     if (cond){              f1();
3.        fp1 = &e;            f2();
4.        fp2 = &h;         }
5.     }else{               e(){
6.        fp2 = &e;            …
7.        fp1 = &h;         }
8.     }                    h(){
9.     bar(fp1, fp2);          …
10. }                       }
```



(a)　　(b)

图 4　扩展变量实例化的示例

引入扩展变量后，当分析不同调用点并且这些调用点调用了同一个被调用函数 callee 时，函数 callee 在这些不同调用点中的扩展变量是不变的，变化的是扩展变量所代表的值，即每个调用点处实参和全局变量的值，称这个过程为扩展变量的实例化. 如图 4(b)所示，函数 bar 的参数为两个函数指针 f1 和 f2，相应的扩展变量为 f1_1 和 f2_2，在图 4(a)的第 9 行调用点处调用 bar 时两个实参都指向函数 e 和 h，这相当于扩展变量 f1_1 和 f2_2 实例化后也都等于 e 或者 h，因此实例化后 f1 和 f2 之间互为别名.

由于一个函数内的局部变量、非指针类型的形参对调用点处的上下文无影响，因此本文将被调用函数中扩展变量之间的别名关系作为上下文，这样可以避免传统使用调用串表示上下文时带来指数级的数量. 当在函数内 caller 内遇到一个函数调用语句 s，设 s 对应的被调用函数为 callee，首先对扩展变量进行实例化得到 callee 中扩展变量在函数 caller 中的值，然后计算扩展变量之间的别名模式，得到上下文 ctx1，最后按照 callee 的分析情况分为两种情况来考虑:

1)当 callee 未分析时则进入 callee 函数内进行过程内分析，分析完后得到扩展变量和返回值的指向信息和它们之间的别名关系，并保存这些信息作为下次匹配上下文的依据.

2)当 callee 已分析时，那么将 callee 中的上下文 ctx1 与 callee 对应已存在的上下文进行匹配. 如果 ctx1 与某个上下文 ctx2 匹配，则直接复用 ctx2 对应扩展变量和返回值的指向信息. 如果找不到这样的上下文与 ctx1 匹配，那么按照(1)的方式重新进入 callee 内分析.

为了加快分析的速度，可以进一步对扩展变量进行优化. 由于本文只关注锁、函数指针的信息，因此在进行扩展变量实例化时可以忽略与锁、函数指针无关的扩展变量，这样上下文中得到更少的别名信息，有利于加快上下文匹配的速度.

另外，过程间分析需要考虑以下特殊情况:

1)函数指针: 当遇到调用点处的函数名是一个函数指针时，则需要首先确定这个函数指针指向的值. 由于本文模拟程序的执行过程，因此在某个调用点处可以根据别名图中指针的指向信息确定函数指针的可能值. 如果函数指针指向的函数有多个，那么依次对每个函数进行过程间分析，然后在分析完后对这些函数返回的结果进行合并，作为这个调用点的返回信息.

2)递归函数: 递归函数可以根据调用栈是否存在环来判断. 当判断出递归函数形成的环后，然后对这个环中所有的函数进行迭代处理. 为了加快分析的速

© 中国科学院软件研究所 http://www.c-s-a.org.cn

度同时尽可能确保分析的精度, 本文限制环迭代的次数最多分析 5 次.

3)线程操作函数: 由于 C 多线程程序主要利用 Pthread 线程库实现并发操作, Pthread 线程库提供 pthread_create 函数来产生新的线程. 因此当遇到 pthread_create 函数时, 需要记录该函数产生线程的入口函数和相关的参数指向信息, 并在本线程分析完后继续对未分析的线程进行分析.

4.4 全局变量的跨线程间传递

由于存在全局变量在不同线程之间赋值的情况, 因此需要考虑一个线程内对全局变量进行的赋值操作可能影响其他线程的分析结果. 本文根据多线程程序执行的不确定性特点, 采取保守的方式处理这种跨线程之间全局信息的传递, 即在已获得每个线程摘要的基础上, 依次对每个线程摘要进行传递, 传递的方法为: 如果线程 T1 有对全局变量进行赋值, 线程 T2 使用了这个全局变量, 那么从 T1 的线程摘要中获得关于这个全局变量赋值语句的右值表达式, 然后在 T2 中使用这个右值继续进行分析. 上述过程不断迭代, 直到最终建立的别名图收敛. 采用这种方式处理跨线程之间全局信息的传递加快了分析的速度, 并且能够尽可能地将所有影响全局变量可能值的情况都考虑到. 虽然这样做影响了分析的精度, 但是针对锁别名分析, 如果存在多个线程对共享的锁指针同时进行修改时, 这种编码不符合多线程程序中锁的安全编码规范[4].

上述分析的最终目的是确定锁别名的信息, 因此当遇到 Pthread 线程库提供的加锁和解锁函数时, 可以根据当前建立的别名图信息, 确定这些函数操作的锁指针指向的锁对象, 以及确定指向锁对象的不同锁指针之间的别名关系.

5 实验与分析

本文选取了常用的开源软件作为 Benchmark 来验证 FP_LOCK 算法的高效性和准确性, 具体见表 1. 表 1 中的程序使用广泛, 并且在已有文献[3,4]中作为测试用例, 其中锁数目表示通过人工查阅源代码后得到的.

表 1 测试用例

程序	代码行数	锁数目
Apache-2.2.8	228317	18
OpenLDAP-2.2.20	252930	41
FastDFS-4.06	82648	18
Pfscan-1.0	752	4
Aget-0.4	835	1

实验环境为: CPU 为 Intel(R) Xeon(TM)@2.10GHz, 内存 4GB, 操作系统: Redhat 4.8.2, 内核版本为 3.11.10, GCC 版本为 4.7.1.

为了说明 FP_LOCK 算法分析结果的精度和时间上的高效, 本文指定如下的实验方案: (1)利用 FP_LOCK 算法对比 GCC 采用流不敏感、上下文不敏感分析算法(FICI)的分析结果, 比较这两种算法产生的不同别名对, 并且以表 1 中锁的数目作为正确性的参照标准. (2)对比 FP_LOCK 算法与没有经过预处理的流敏感、上下文敏感分析算法的运行时间, 以表明经过预处理后的 FP_LOCK 算法的高效性. 实验结果如表 2 和表 3 所示.

表 2 FICI 算法与 FP_LOCK 算法的分析结果

程序	中间代码行数		锁别名数	
	未预处理	预处理	FICI	FP_LOCK
Apache	92925	4652	4	18
OpenLDAP	98021	5651	9	41
FastDFS	35494	2337	13	18
Pfscan	732	30	4	4
Aget	571	104	1	1

表 2 中别名数表示分析结果中互相别名的锁的个数, 别名数可以体现本文分析的精确度. 由表 2 可得 GCC 利用 FICI 算法时保守地估计锁变量之间可能互为别名, 因此产生的锁变量别名数大大减少, 虽然针对 Pfscan 和 Aget 程序, FICI 和 FP_LOCK 分析的结果一致, 但通过源码可知这是因为 Aget 和 Pfscan 操作的锁直接是一个全局变量, 对于规模较大的其他程序, 当内部操作的是一个局部锁时, FICI 算法就不能精确识别不同的锁. 而 FP_LOCK 算法针对不同的上下文识别不同的锁, 并且锁的别名数与表 1 中程序使用的锁数目一致, 因此 FP_LOCK 算法的分析是比较准确的. 另外, 表 2 还可得经过预处理后的中间代码规模比未预处理要减少很多, 这大大可以提升 FP_LOCK 算法的分析效率, 表 3 的实验结果说明了这点.

表 3 预处理后的实验效果(单位: 秒)

程序	分析时间		加速比
	未预处理	预处理	
Apache	147.78	8.12	18.2
OpenLDAP	288.96	13.63	21.2
FastDFS	37.24	5.32	7.0

© 中国科学院软件研究所 http://www.c-s-a.org.cn

Pfscan	0.12	0.08	1.5
Aget	0.13	0.07	1.86

表 3 比较了经过预处理后的 FP_LOCK 算法与未预处理的算法. 由表 3 的加速比可得经过预处理后的 FP_LOCK 算法比未预处理算法有较高的时间效率, 并且随着程序规模的增加, FP_LOCK 算法的优势明显. 整体而言, FP_LOCK 算法有着 9.95 倍的平均加速比, 适合用来静态分析大型多线程程序.

6 总结

本文针对 C 多线程程序提出了一种锁别名分析方法, 该方法首先利用 GCC 编译器提供的插件功能获得 SSA 形式的中间代码, 然后对中间代码进行预处理, 以获得与函数指针、锁指针操作有关的语句, 最后利用更精确的 FP_LOCK 算法得到程序中锁指针的别名关系. 实验结果表明本文提出的锁别名分析方法适合处理规模较大的多线程程序. 由于本文主要针对 C 程序, 而现有针对 C++的多线程程序也使用比较广泛, 因此下一步可以针对 C++程序的语法特点, 通过 GCC 编译器生成虚表和类型信息, 并根据类型信息来解析继承关系, 从而可以在过程内分析时根据继承信息和虚表来进行精确的函数查找, 包括对重载的分析.

参考文献

1 Stefan S, Michael B, Greg N. Eraser: a dynamic data race detector for multithreaded programs. ACM Trans. on Compute Systems, 1997, 15(4): 391–411.

2 Dawson EK. A RaceX: Effective, static detection of race conditions and deadlocks. Proc. of the 9th ACM Symposium on Operating Systems. 2003. 237–252.

3 Wang Y, Kelly T, Kudlur M. Gadara: Dynamic deadlock avoidance for multithreaded programs. Proc. of the 8th USENIX Conference on Operating Systems Design and Implementation. 2008. 281–294.

4 Cho HK, Wang Y, Liao HW. Practical Lock/Unlock Pairing for Concurrent Programs. IEEE International Symposium on Code Generation and Optimization, 2013:1–12.

5 Andersen LO. Program Analysis and Specialization for the C Programming Language[Thesis]. Copenhagen, Denmark: University of Copenhagen, 1994.

6 Nagaraj V, Govindarajan R. Parallel flow-sensitive pointer analysis by graph-rewriting. Proc. of the 22th International Conference on Parallel Architectures and Compilation Techniques. 2013. 19–28.

7 董玉坤,金大海,宫云战,等.基于区域内存模型的 C 程序静态分析.软件学报,2014,25(2):357–372.

8 章迪.基于指针别名分析的高效锁集分析工具[硕士学位论文].上海:上海交通大学,2013.

9 Cytron R. Efficiently Computing static single assignment form and the control dependence graph. ACM Trans. on Programming Languages and Systems, 1991, 13(4): 451–490.

10 hubicka. GCC Wiki. https://gcc.gnu.org/wiki. [2015–03-10].

11 Sheng T, Chen W, Zheng W. A context-sensitive pointer analysis phase in Open64 compiler. The 2nd Annual Workshop on Open64 in Conjunction with IEEE International Symposium on Code Generation and Optimization. 2009.

12 Altucher TRZ, Landi W. An extended form of must alias analysis for dynamic allocation. Conference Record of the 22nd ACM SIGACT Symposium on Principles of Programming Languages. 1995. 74–84.

13 于洪涛.精确高效的C语言指针分析技术研究[学位论文].北京:中国科学院计算技术研究所,2010.

© 中国科学院软件研究所 http://www.c-s-a.org.cn