软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2012,23(6):1413−1428 [doi: 10.3724/SP.J.1001.2012.04102]

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

全路径剖析方法*

王璐璐, 李必信+, 周晓宇

(东南大学 计算机科学与工程学院,江苏 南京 211189)

Profiling All Paths

WANG Lu-Lu, LI Bi-Xin+, ZHOU Xiao-Yu

(School of Computer Science and Engineering, Southeast University, Nanjing 211189, China)

+ Corresponding author: E-mail: bx.li@seu.edu.cn, http://cse.seu.edu.cn/people/bx.li/index.htm

Wang LL, Li BX, Zhou XY. Profiling all paths. *Journal of Software*, 2012,23(6):1413−1428. http://www.jos.org.cn/1000-9825/4102.htm

Abstract: Path profiling, an important technique in dynamic program analysis, which collects the execution times of different paths, has been widely used in a variety of areas. However, existing intra-procedural profiling techniques have inadequate abilities in loops, i.e., they can only either solve profiling acyclic paths, or limit loop iterations to a certain number first, and then profile paths under such a limitation. This paper presents a new profiling technique called PAP (profiling all paths), which can profile finite-length paths inside a procedural. PAP consists of two basic phases: one is the probe instrumentation which assigns a unique pathid for each path, and the other is backwalk which uses the pathids to determine the corresponding executed paths. Furthermore, breakpoints are introduced to store the probe value which may overflow during long executions, and the probe amount is reduced base on the integration of PAP with an existing profiling technique. Besides, this paper also discusses how to use PAP to profile executed sequences on the method level. As shown in the results of the case study and experiments, PAP is effective and efficient in profiling cyclic paths.

Key words: path profiling; dynamic analysis; cyclic paths; probe instrumentation; path backwalk

摘 要: 路径剖析是动态分析的一项重要技术,通过获取和分析程序中各条路径的执行次数,在编译优化、软件调试和测试等诸多方面发挥重要作用.针对现有技术剖析能力不足的情况(即只能或者剖析非循环路径,或者首先界定循环体执行次数的上限、然后对于执行循环体不多于该次数的路径进行剖析),对使用单个探针变量剖析过程内路径的方法进行了改进,提出了全路径剖析 PAP 方法,利用探针插装和回溯过程获取路径的执行次数,可以剖析过程内包含任意有限长度的路径;进一步地,针对 PAP 方法所需探针数目多于 EPP 方法的问题,通过对控制流图中包含的可规约无环子图实施 EPP 方法,可以减少 PAP 方法所需探针的数目.另外,作为 PAP 方法的一个典型应用,还讨论了如何通过在方法调用图中添加返回边,再利用 PAP 方法获取方法层次的执行序列的基本思想,满足了某些方法级动态影响分析技术的需要.实验和实例分析表明,PAP 在处理循环路径剖析的问题上是有效的,并有很好的效率.

关键词: 路径剖析;动态分析;循环路径;探针插装;路径回溯

中图法分类号: TP311 **文献标识码**: A

* 基金项目: 国家自然科学基金(60973149); 国家教育部博士点基金(20100092110022)
收稿时间: 2010-12-16; 定稿时间: 2011-08-24

路径剖析(path profiling)是一项重要的动态分析技术,用于获取程序一系列执行中各条路径的执行次数,在计算机架构、程序的编译、调试、测试和软件维护等多个方面有着重要的应用[1].路径剖析技术希望使用尽量少的消耗来获取程序各条路径的执行次数,一般可以分为3个步骤:(1) 探针(probe)变量的设定及其操作代码的插装;(2) 在程序执行的过程中计算探针变量值(简称探针值)并收集相关信息;(3) 根据路径编码(即路径末端的探针值)和相关信息获取各条路径及其执行次数.

为了获取路径剖析结果,有的研究尝试从边的剖析结果中推断路径剖析结果[2],由于边的剖析可以使用很低的耗费实现[3−8],使用这种方法进行路径剖析耗费低廉,但是部分路径(实验为 52%[9])的执行信息无法从边的剖析中获取.

Ball 等人针对不包含回边(backedge)的路径,给出了过程内单变量剖析方法 EPP(efficient path profiling),即整个程序只使用单个探针变量进行剖析,该变量的不同取值代表不同的路径[10,11].此后,文献[9,12−14]基于仅剖析部分非循环路径的需求改进了该方法的效率,但未能解决包含循环的路径剖析问题.

针对过程内包含循环的路径,Tallam 等人提出了对包含循环体两次以内执行的路径进行近似剖析的方法,实验中的误差在−4%~+8%之间[15];Roy 等人则在此基础上提出了一种新的方法(profiling k-iteration paths,简称 kIPP),能够准确地剖析含有循环体 k 次执行的路径.但是这种方法仅能剖析不超过 60 000 数目的静态路径,导致 k 的大小严重受限于程序规模[16].由于静态路径的数目随 k 呈指数增长,这种方法难以处理较为复杂的程序和执行循环体次数较多的路径.

控制流图(control flow graph,简称 CFG),本文讨论的 CFG 仅限于单入口单出口 CFG,路径仅限于从入口到出口的完整路径,其他形式的 CFG 可以通过添加统一的入口和出口节点转化为单入口单出口 CFG.CFG 是一个有向图 $G=(N,E,entry,exit)$,其中,N 是节点集、E 是边集、*entry* 和 *exit* 分别为程序的入口和出口节点.CFG 中不同的边的组合顺序产生了不同的路径.除了 CFG 的入口节点之外,每个节点都应该存在入边,而且有些节点存在多条入边(例如循环入口节点).在路径剖析中,当 CFG 中某个节点具有多个入边时,为这些边插装探针代码,使用乘法与加法相结合的方式保证不同入边对应于不同的探针值;在被剖析程序执行之后进行回溯,利用上述运算的逆运算区分同一节点的不同入边,从而获得执行路径.基于这种观察,本文提出了一种新的单变量全路径剖析方法 PAP(profiling of all paths).PAP 方法通过区分同一个节点的不同入边,利用不同入边区分不同路径,可以有效地剖析包含循环体任意多次执行的路径.但随着被剖析程序的执行,探针值会不断增大.为此,我们还设计了断点机制来解决可能的探针值溢出问题,使剖析过程不受路径长度的限制.

计算表明,在剖析不含有循环的路径时,PAP 方法使用的探针的数目略多于 EPP 方法使用的探针的数目,但不超过后者的两倍.针对此问题,本文通过对控制流图中的可规约无环子图使用 EPP 方法进行插装,将 PAP 与 EPP 相结合,减少了探针数目,提高了执行的效率.

在软件分析过程中,有时仅需要方法级的路径信息(如文献[17−21]中的方法),但是据我们所知,很少有文献专门讨论如何将路径剖析方法用来获取方法层次的执行序列.虽然目前的剖析方法(如 EPP)可以依据控制流图进行路径剖析以获取方法层次的执行序列,但按照这种技术路线,即便在控制流图中忽略与方法调用无关的节点,还是需要保留部分与方法调用相关的控制结构.本文通过在方法调用图(call graph)中添加调用返回边(return edge),并在扩展后的调用图基础上,利用 PAP 方法剖析循环的能力获取方法执行序列,可以忽略方法内部的所有结构信息,计算工作量相对较小.

1 PAP 方法

PAP 方法的的基本过程如图 1 所示:首先,在单变量剖析方法的基本过程之上改进了探针插装算法,并设计了回溯算法以通过探针值获取相应的路径;然后,在此基础上提出了断点机制以处理探针值可能溢出的情况,并在 CFG 的可规约无环子图中结合使用 EPP 方法进行剖析,改进了效率.接下来,首先阐述 PAP 方法的探针插装和路径回溯是如何实现的,然后依次讨论如何利用断点机制和可规约无环子图(reducible acyclic subgraph,简称 RAS)对 PAP 方法进行进一步的改进.

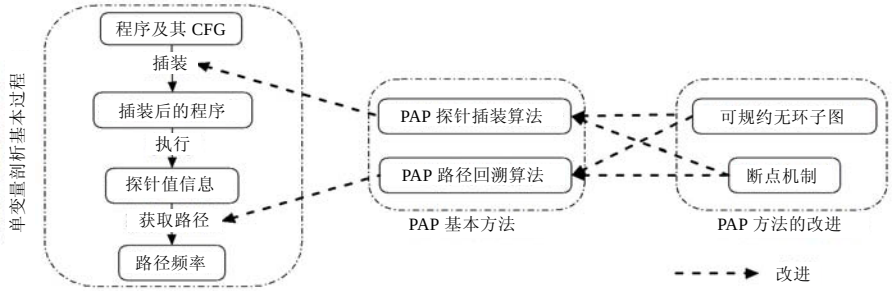


Fig.1 Framework of PAP

图 1 PAP 过程

1.1 探针插装与路径回溯

据资料显示,在不考虑循环时,过程内的路径数目一般较少.因此,现有的剖析方法通过枚举的方式很容易建立探针值与路径的一一对应关系.这样,在被剖析程序执行之后,根据探针值检索就能够获取相应的路径[11];在允许路径包含循环体有限次执行的情况下,现有的方法首先界定循环体的执行次数,确定满足此条件的静态路径(static path)的集合,最后在运行完毕后通过不同路径执行次数之间的关系进行推理,从而获得相关路径的执行次数[15,16].总之,由于现有剖析方法能够剖析的路径数目较少,所以在被剖析程序执行之前就可以先确定路径集合,并建立路径与探针值的对应关系.而在考虑循环的任意次执行时,路径的数目没有上限,很难利用现有的剖析方法建立执行路径和探针值之间的对应关系.因此,新的探针插装算法以及基于探针值进行回溯以获取路径的算法必不可少.

EPP 等剖析非循环路径的方法[9,11−14]普遍采用的路径编码方式是:对 CFG 中的每条边指定一个整数权值,当程序运行中执行了非零权值的边时,依据该权值探针变量做相应的增减;相应的,每条非循环路径的编码就是该路径中的所有边的权值之和.这种编码方式难以应用到循环路径的剖析中,因为可能存在两条循环路径由完全相同的边组成,仅对边上权值求和的计算方式无法将其区分开来,会导致剖析结果错误.而已有的循环路径剖析方法[15,16]采用在执行过程中记录回边,以较为复杂的推理方式区分路径编码,但是由于这种推理过程受限于静态路径数目,难以应用到回边执行次数较多的情形.

本文提出的 PAP 方法采用的路径编码方式与 EPP 等方法类似,但是在其基础上改进了编码的计算,不是单纯地使用加法,而是采用乘法与加法相结合的方式,能够保证所有的路径对应不同的编码.

在 PAP 方法中,我们用 CFG 中节点的序列来表示路径;而在 CFG 中,不同边的组合顺序导致了不同的路径.在剖析过程中,无论是区分同一个节点的不同入边还是区分其不同出边,都可以建立探针值与路径的对应关系.考虑到将来的回溯过程是由程序出口点开始的,我们通过区分同一个节点的不同入边来区分路径.

算法 1 显示了 PAP 插装算法的基本思想(addP 表示在 CFG 的边或节点上添加探针语句,下文中将“*r*:=*r***s*+*i*”简写为“*s*(*i*)”)[2]**.该算法找出 CFG 中全部具有多个入边的节点,按照算法 1 中(#)句的方式,可以使得不同的入边对应于不同的探针值,即执行完毕后不同的路径对应不同的编码.

算法 1. PAP 探针插装算法.

Input: CFG *f*;

addP(*f.entry*,“*r*:=0”); //添加探针初始化

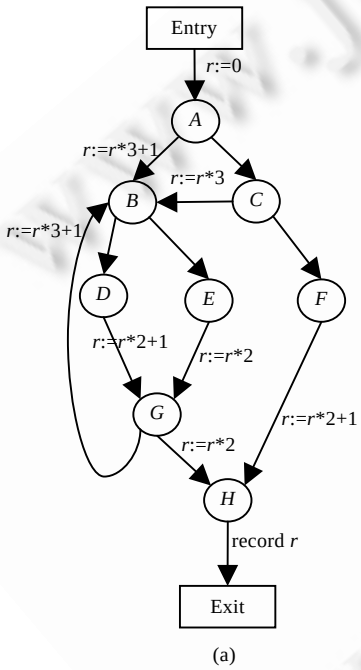
foreach *node n* in *f* **do**

** EPP 等方法收集探针值的语句为“count[r]++”,表示优先使用数组存储,效率较高;当路径编码超出数组长度限制时转而使用链表存储,效率较低.而在 PAP 中,本文统一将探针值的记录语句记为“record r”,表示类似的含义.

```
int s=n.fanIn();                    //节点 n 的入边数目
if s>1 then
    int i=0;                        //i 为计数器,对入边进行标识
    foreach n 的入边 e do
        addP(e,"r:=r*s+i");         //添加探针                    (#)
        i++;
    end
end
addP(e_exit,"record r");            //记录 r 的值
end
```

当程序执行完毕,根据探针值进行回溯,从 CFG 的出口节点开始迭代计算每一个节点的入边.回溯过程中,取余和取整运算是探针值计算过程中累加和累乘运算的逆运算.这样,在计算探针值和回溯过程中,就能够对节点的不同入边进行区分.而且无论入边是循环的回边还是普通边,也无论循环是自然循环还是非自然循环,区分方法是一致的.这样,PAP 方法就能够剖析各种控制流结构形成的所有路径.另外,由于探针值的计算和回溯过程中的互逆运算在探针值与路径之间建立了一一对应关系,所以在多次执行的同一条路径时,只需进行一次回溯.

图 2(a)给出了一个 CFG 的实例(为统一起见,为 CFG 添加了 *entry* 和 *exit* 节点,边上的标记为该边所对应的探针值计算操作).例如,节点 *B* 有 3 条入边 *AB*,*CB* 和 *GB*,其中,*GB* 是回边.回溯过程中,节点 *B* 处探针值的取余结果 0,1,2 分别对应入边 *CB*,*GB* 和 *AB*.图 2(b)给出了带有非自然循环的 CFG.图 2 右侧部分给出了部分路径对应的探针值.



路径	r
(a)	
ACBEGH	0
ACFG	1
ACBDGH	2
ABDGF	10
ABDGBDGH	66
…	
(b)	
ABDE	1
ABCDE	0
ABDCDE	6
ABCDCDE	2
ABDCDCDE	26
ABCDCDCDE	10
…	
(c)	
ABC	1
ABCBC	2
ABCBCBC	4

Fig.2 Examples of PAP algorithm

图 2 PAP 算法示例

1.2 断点机制:应对探针值溢出问题

现有的剖析方法仅处理路径数目和长度较小的情况,没有应对探针值溢出的机制.而 PAP 方法针对的路径

包括循环的任意次执行,随着程序的执行,探针值不断增大,可能在剖析的过程中出现探针值溢出的问题.在程序的执行过程中,一条路径中往往包含很多大量重复的路径片段,如果这些路径片段能够与部分探针值相对应,那么相同的探针值信息可以使用编码的方式压缩存储空间,减少耗费.因此,我们为 PAP 方法设计了断点机制,当溢出发生时,记下前一位置的探针值和执行节点,这样记录的信息称为断点,然后将探针值重置为前一条边上探针对应的余数,继续计算.关于在实际应用中断点的重复性以及能够节约的存储空间,将在后文的实验中进行检验.

在基于断点的回溯过程中,每个断点对应于一段路径,其回溯过程从该断点所存储的节点开始,至前一断点所存储的节点为止(如果没有前一断点,则回溯到 CFG 的入口节点).各段路径的回溯过程可以独立进行,便于采用并行的方法提高效率.将各个断点的回溯结果依次连接,便可得到完整的回溯结果.

如图 2(a)中的 CFG,假设一次执行过程中,从入口节点执行到 G 节点处(记 G 节点的该次执行为 G_x),探针值均未溢出;但是执行了 GB 边之后,在 B 节点处探针值溢出(记 B 节点的该次执行为 B_y,判断溢出的方法比较简单,不再赘述),那么将 G 节点与 G 处的探针值作为断点进行记录后,将 B 处的探针值重置为 1(即 GB 边对应的余数),继续执行并计算探针值.相应地,在使用断点进行分段回溯时,由该断点从 G 节点开始回溯,可以计算出 G_x 及其之前的路径,与其他断点回溯所得到的 B_y 及其之后的路径相衔接,即可得完整的执行路径.

特别地,在使用断点机制时,后向边(back edge)[22]对应的余数不能为 0,否则如图 2(c)所示,如果执行了 CB 边后发生了溢出,则 C 点及其探针值成为一个断点,CB 边的此次执行成为新的阶段的开始.该边上探针计算操作的余数部分为 0,因此将 CB 边上的 r 值重新设定为 0.由于 C 点只有一条入边,因此作为循环体的 BC 边上不存在探针操作,对探针值没有影响.这样,无论 CBC 这样的循环执行多少次,只执行“r*2”这个操作,r 的值总是为 0,无法反映循环的执行路径.注意到,一个节点的入边至少有一条不为后向边,插装探针时将余数为 0 的探针对应于一条非后向边即可.图 2(c)中将 AB 边和 CB 边上的探针交换后,就可以应用断点机制进行剖析.

引入断点机制对 PAP 剖析过程带来了一些影响:首先,路径不再由探针值标识,而是由一个断点序列标识,相同的断点内容及顺序代表相同的路径;其次,为了检测探针值溢出和保存断点所需信息,需要在插装时使用额外的变量进行记录,增加了插装和执行的耗费.但是,这样的影响并不会改变 PAP 方法的正确性和精确性,而且第 2 节的实验分析可以说明增加后的耗费依然在合理范围之内.

1.3 探针的数目

由于插装的探针需要随着被剖析程序一起执行,所以探针的数目不仅影响插装算法的效率,也影响插装后的执行效率.EPP 方法是一种不剖析循环的单变量剖析方法,我们计算和比较了 DAG(directed acyclic graph)中 PAP 和 EPP 需要的探针数目.

首先,在 CFG 中,定义 $N_{fan_out}(a)$为节点 a 的出边数目,$N_{fan_in}(a)$为 a 的入边数目.那么由 PAP 探针插装算法可知,对于多入边的节点的每条入边都需要一个探针,此外还需要两个探针分别用于探针变量的初始化和探针值的收集,故 PAP 插装的探针数目满足:

$$N_{PAP\_probe}=\sum_{a\in CFG,N_{fan\_in}(a)>1}N_{fan\_in}(a)+2.$$

通过对 Ball-Larus 方法分析可知,针对 DAG 的每个具有多出边的节点,该节点的一条出边不需要探针,其余出边均需要一个探针,考虑到探针变量的初始化和探针值的收集可能会增加探针的数目,故其插装的探针数目满足:

$$N_{EPP\_probe}\geqslant\sum_{a\in DAG,N_{fan\_in}(a)>1}(N_{fan\_in}(a)-1).$$

对于 DAG,记总节点数目为 N,单入边节点的数目为 N_{fain_in}=1,则 PAP 需要的探针数目满足

$$N_{PAP\_probe}=\sum_{a\in DAG,N_{fan\_in}(a)>1}N_{fan\_in}(a)+2=\sum_{a\in DAG}N_{fan\_in}(a)-N_{fan\_in=1}+2.$$

又由于 DAG 中所有出边之和与所有之和相等:

$$\sum_{a\in DAG} N_{fan\_in}(a) = \sum_{a\in DAG} N_{fan\_out}(a).$$

故

$$N_{PAP\_probe} = \sum_{a\in DAG} N_{fan\_out}(a) - N_{fan\_in=1} + 2 \quad (*)$$

而 DAG 至少有一个节点没有出边,故 EPP 需要的探针数目满足:

$$N_{EPP\_probe} \geqslant \sum_{a\in DAG, N_{fan\_in}(a)>1} (N_{fan\_in}(a)-1) \geqslant \sum_{a\in DAG} N_{fan\_out}(a) - N + 1.$$

DAG 中多入边节点的数目具有上限:

$$N_{fan\_in>1} \leqslant \sum_{a\in DAG} N_{fan\_out}(a) - N + 1.$$

又由于 DAG 至少有一个节点没有入边,使用上式可以推出:

$$N_{fan\_in=1} = N - N_{fan\_in>1} - N_{fan\_in=0} \geqslant N - N_{fan\_in>1} - 1 \geqslant 2N - \sum_{a\in DAG} N_{fan\_out}(a) - 2,$$

带入公式(*)可得:

$$N_{PAP\_probe} \leqslant 2\sum_{a\in DAG} N_{fan\_out}(a) - 2N + 4.$$

与 EPP 探针数目相比

$$N_{PAP\_probe} \leqslant 2N_{EPP\_probe} + 2.$$

即 PAP 方法使用的探针数目大约不超过 EPP 方法的两倍.

1.4 使用EPP改进

从上面的分析可知,针对 DAG,PAP 的探针数目略高于 EPP.为了减少探针数目,提高执行速度,我们在 CFG 的某些无环子图中应用 EPP 方法.也就是说,可以利用 EPP 方法来改进 PAP 方法.

直观地说,能够应用 EPP 的无环子图必须是 CFG 中相对独立的部分,与 CFG 中的其他部分的直接联系仅限于子图的出口和入口.这样,可以使用 EPP 方法枚举子图中的路径,并将 EPP 的探针值纳入 PAP 的探针计算之中.故此,我们给出可规约无环子图(简称无环子图)的概念.

无环子图是 CFG 中满足以下条件的子图:

(1) 子图具有单入口单出口,且入口节点可达子图所有节点,子图所有节点可达出口节点;

(2) 子图中的边均不是后向边;

(3) 子图入口节点是出口节点的前必经节点,子图出口节点是入口节点的后必经节点.

应用 EPP 方法的子图应该在所有极大无环子图中选择.易知,如果一个无环子图中多入边节点不少于 3 个,那么在子图中应用 EPP 方法就能够减少探针数目.

对于子图中的 n 条路径,EPP 方法得到的编码是从 0~n–1 的连续整数.从整个控制流图来看,等同于子图的出口节点有 n 条入边.故我们将可规约无环子图规约成一个节点,并使用 EPP 方法为其配置路径表和局部探针,将 PAP 与 EPP 相结合.见算法 2.

算法 2. 无环子图中应用 EPP.

Input: CFG *f*, Graph *b* //*b* 是一个可规约无环子图

b 中使用 EPP 方法,插装的探针为 *r*′,*b* 中路径数目为 *n*

addP(*b.exit*,"*r*=*r***n*+*r*′") //探针插装在 *b* 块的出口

图 3(a)给出了一个示例,节点 *G*,*B* 之间有一条回边,假设虚线框中为一个无环子图,那么将该子图归结为一个节点 *K*(统一起见,为 *K* 添加入口和出口),如图 3(b)所示.假设子图 *K* 的具体结构如图 3(c)所描述,其中有 9 条路径,EPP 方法得到的探针值为 0~8.通过分析可知,PAP 方法在 *K* 中插装需要 8 个探针,而使用 EPP 后缩减为 5 个,减少了 2 个(子图规约本身需要一个额外的探针).例如,某次执行的探针值为 1 060,通过回溯得到路径为 *AKKH*,且第 1 个 *K* 处 *r*′=1,第 2 个 *K* 处 *r*′=8,查局部路径表替换即得执行路径为 *ABDMIJGBEMJGH*.

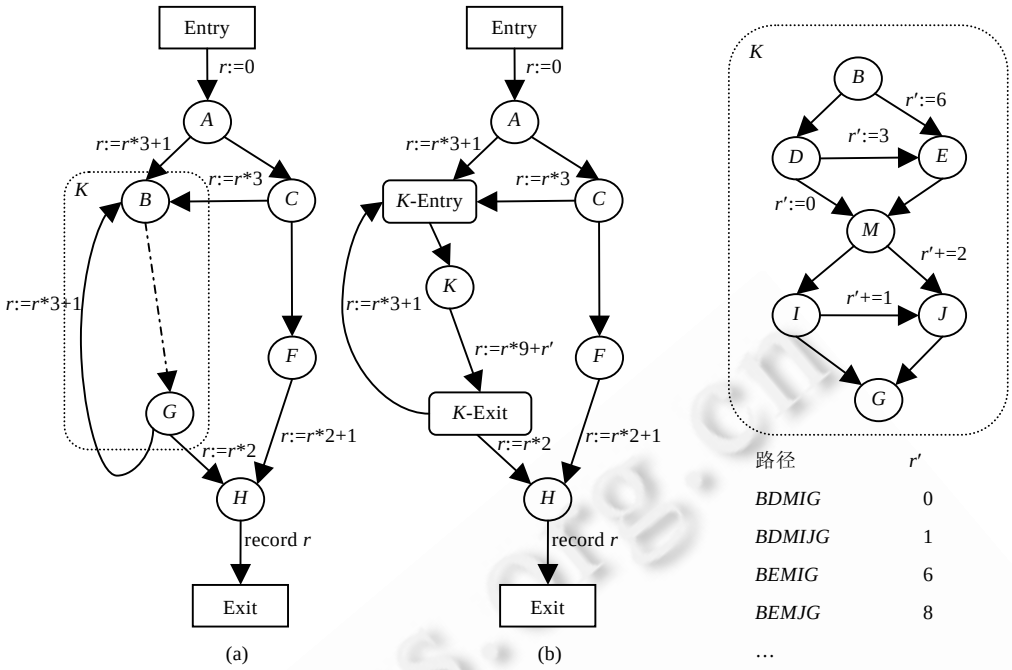


Fig.3 PAP integrated with EPP

图 3 结合 EPP 示例

2 实验分析

上节中,我们讨论了在 PAP 插装和回溯算法的基础上如何使用断点机制和在无环子图中应用 EPP 对 PAP 方法实施改进的两种方式.本节讨论如何通过实验使用随机生成的 CFG 对两种改进的有效性进行验证,并使用一个实际的程序 *lufact* 作为实例对 PAP 方法的过程进行说明,同时对其整体耗费进行分析.最后,我们使用 JGF.section1[23]中的 9 个基准测试程序对 PAP 和现有方法(主要是 EPP 和 kIPP)的剖析能力与耗费进行全面的比较.

2.1 实验1:断点的数目

用来检验断点的实际效果:首先,随机生成 4 个 CFG,分别为 CFG1~CFG4(节点数从 30~202 不等),然后为每个 CFG 随机生成长度不等的数条路径(路径包含的节点数从 500~5000 不等),相应执行的断点数目见图 4 所示.

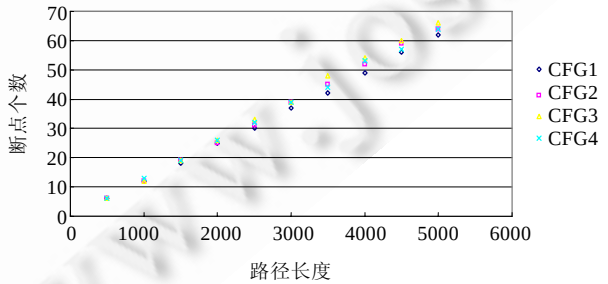


Fig.4 Results of breakpoints

图 4 断点数目的实验结果

可见,断点的数目基本随路径长度呈线性增长,而且远小于路径的长度,是一种高效的路径存储方式.断点

需要存储的内容包含一个整型数和一个节点名称,断点的数目随路径长度呈线性增长.只要存储空间足够,就能够对任意长度的路径进行剖析.这是以往的剖析方法所不能做到的.对于在实际程序中断点的存储效率,我们将在第 2.4 节作进一步的检验.

2.2 实验2:探针的数目

用来检验探针的数目.随机生成无环的控制流图,应用EPP方法,检测实施EPP之后的效果.实验结果如图5所示,表明在无环子图中结合EPP能够有效地减少探针数目(*sn*表示子图的节点数目,*cn*表示使用PAP方法插装的探针数目,*in*表示结合EPP方法减少的探针数目).

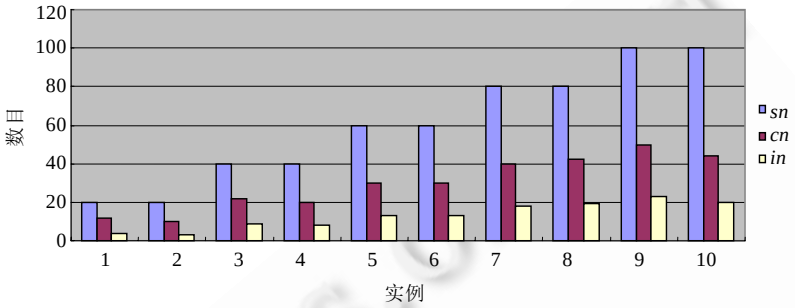


Fig.5 Reducing probe amounts with EPP

图 5 结合 EPP 减少探针数目

2.3 实验3:耗费的实例分析

用来进行整体耗费分析和比较.这里使用程序 *lufact* 来对 PAP 方法的整体耗费进行分析(文献[16]中使用了基准测试程序 *ludcmp* 对其方法与 EPP 方法的效率进行比较,*ludcmp* 使用 C 语言编写,实现 LU 分解的功能,本文使用的 *lufact* 是实现同样功能的 Java 基准测试程序,来自 JGF[23]).PAP 方法的耗费分为 3 个部分:插装、执行和回溯.

2.3.1 插装的耗费

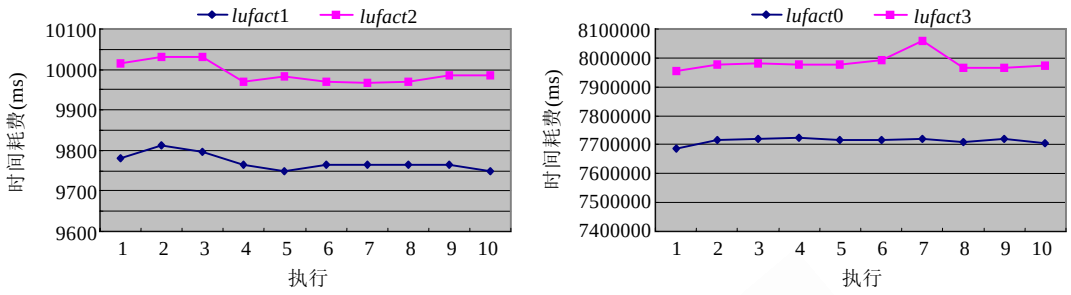
PAP 插装耗费分为两部分:首先,探针插装算法需要对 CFG 中每个节点的所有入边进行分析,即其时间耗费与 CFG 中的边数呈线性关系,这与 EPP 等方法是相同的(kIPP 方法的插装耗费与 CFG 的边数与 *k* 的乘积成正比);而后,由算法得到的结果将探针插装在程序代码中,其耗费与探针的数目呈正比.

lufact 的程序结构中共有 13 个节点和 23 条边,其中 4 条是回边.分析可知,EPP 方法需要插装 15 个探针,PAP 方法共需要 24 个探针.此实例中,PAP 插装算法的耗费与 EPP 相当,探针数目小于 EPP 的两倍,故总的插装耗费小于 EPP 的两倍.

2.3.2 执行的耗费

对于需要多次执行以获取路径频率的剖析过程,插装后的执行效率对剖析方法的整体耗费有着决定性的影响.PAP 方法的执行耗费包括执行探针语句的时间耗费和存储断点的空间耗费.

在时间耗费方面,由于 PAP 与 EPP 剖析的路径不同,我们使用插装后执行时间增加的百分比来进行衡量.为了进行比较,首先对程序做修改,记原始版本为 *lufact*0,对 *lufact*0 在回边处添加结束语句,记为 *lufact*1;对 *lufact*0 使用 EPP 方法插装,记为 *lufact*2;对 *lufact0* 使用 PAP 插装,记为 *lufact*3.输入相同参数时,*lufact*0 和 *lufact*3 的执行路径相同,*lufact*1 和 *lufact*2 的执行路径相同.使用相应的运行时间差值,即可作为 EPP 和 PAP 的执行时间耗费.独立随机生成 10 组输入参数,使用每组参数执行 4 个版本各 100 000 次,时间耗费如图 6 所示.



(a) EPP 时间耗费对比 (b) PAP 时间耗费对比

Fig.6 Time cost of EPP and PAP

图 6 时间耗费对比

计算结果显示,EPP 插装后程序的执行时间增加了 2.07%~2.41%,平均约为 2.24%;PAP 插装后执行时间增加了 3.20%~4.40%,平均约为 3.50%.在空间耗费方面,PAP 在每次执行中使用的断点数目均为 64.

2.3.3 回溯的耗费

EPP 等剖析方法普遍在执行前将待剖析路径一一枚举,并建立路径与探针值的对应表,这样在执行后根据探针值可以检索到相应的路径.由于其剖析的路径数目较少,采用这种方式可以高效地完成探针值到路径的转换.对于 PAP 方法,路径集合无法在执行前进行穷举,需要在执行后使用相应的断点进行回溯.由于每条路径对应的各个断点的回溯过程可以独立进行,故回溯的耗费与执行后得到的相异断点有关.

对于实例 *lufact*,每次执行产生的 64 个断点,相异的仅有 11 个.由于回溯过程中的运算是执行过程中探针值计算的逆运算,故每个断点的回溯耗费与相应路径片段上的探针执行耗费大致相同,那么本实例中,每条路径的回溯耗费约占其执行耗费的 17%.再考虑到多条路径的断点可能再次重复,回溯的耗费会更小.即使以 17%计算,PAP 方法的执行与回溯耗费之和约为 4.10%,低于 EPP 执行耗费的两倍.

2.3.4 耗费的比较

通过比较 PAP 的各个阶段可知:插装的耗费较小,而且与剖析过程中的执行次数无关.但是由于需要从程序获取其控制流图,并且要将探针无误地插装到程序中,整个过程难以完全自动化;执行和回溯易于自动实现,其耗费都与执行过程相关,二者相比,回溯的耗费小于执行耗费.

另外,通过此实例分析中与 EPP 相比,PAP 插装耗费不超过 EPP 的两倍,执行与回溯耗费之和也不超过 EPP 的执行耗费的两倍.而文献[15]中的实验表明,其文中方法的耗费平均为 EPP 的 4.2 倍.

文献[16]中方法的耗费和预先限定的循环次数有关,在 *ludcmp* 实例上的实验表明:限定执行循环体两次时,耗费为 EPP 的 1.21 倍;限定 3 次时,耗费为 EPP 的 4.47 倍.相比之下,PAP 可以高效地剖析包含循环任意次执行的路径.

2.4 实验4:耗费的实验分析

本节使用基准测试程序检验 PAP 方法的耗费和剖析能力.

2.4.1 基准测试程序

我们采用 JGF 的 section1 部分作为基准测试程序,因为其中的程序都是独立的单个过程,符合 PAP 等过程内剖析方法的应用对象.section1 部分共有 9 个程序,其 CFG 的信息见表 1.

Table 1 CFG information of programs in JGF.section1

表 1 JGF.section1 中各个程序的 CFG 信息

基准测试程序	CFG 节点数	CFG 边数	CFG 回边数	PAP 探针数	EPP 探针数
JGFArithBench	382	447	26	124	64
JGFAssignBench	301	352	20	100	50
JGFCastBench	186	207	8	40	20
JGFCreateBench	442	511	34	136	68
JGFExceptionBench	81	94	6	24	12
JGFLoopBench	42	52	3	18	9
JGFMathBench	966	1 117	60	300	150
JGFMethodBench	219	252	16	64	32
JGFSerialBench	162	182	10	36	19

2.4.2 时间耗费

对于每一个基准测试程序,我们参照第 2.3.2 节的实验步骤来统计执行时间.分别在基准测试程序的基础上使用 PAP 方法插装,在消除回边得到的无环程序上使用 EPP 方法插装,计算插装后的程序与插装前程序执行时间的比值.实验结果如图 7 所示.可以发现,在大部分程序中,PAP 方法插装后没有带来明显的执行时间增加,这与 EPP 的插装类似.但是在 *JGFAssignBench* 和 *JGFMathBench* 的结果中,PAP 插装后执行时间增加了约 70%~80%,这是由于这两个程序的循环体中由运算速度很快的语句组成(如赋值语句),而且这样的循环体执行次数很多,在其中插装的 PAP 语句对于整体程序的执行时间产生了较大的影响;在 EPP 插装后的程序并没有多次执行这样的循环体,所以对执行时间影响较小.

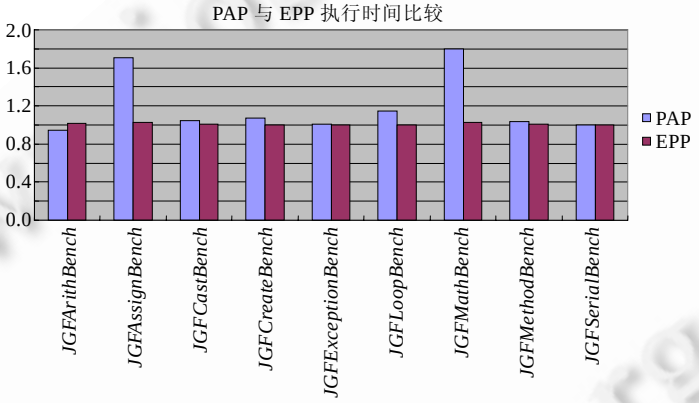


Fig.7 Profiling time cost on JGF.section1

图 7 JGF.section1 插装后的执行时间比较

2.4.3 空间耗费

在空间耗费方面,PAP 在执行中使用的断点数目见表 2.

Table 2 Space cost of profiling programs in JGF.section1

表 2 JGF.section1 中各个程序执行的空间耗费

基准测试程序	断点数目	相异断点数目	回边的执行次数
JGFArithBench	4 107 380	129	117 639 262
JGFAssignBench	3 300 010	109	102 299 566
JGFCastBench	1 320 004	43	40 919 764
JGFCreateBench	6 272 604	192	194 446 680
JGFExceptionBench	3 549	9	109 947
JGFLoopBench	1	1	0
JGFMathBench	3 923 245	258	121 618 336
JGFMethodBench	2 309 683	78	71 599 666
JGFSerialBench	988	34	30 136

从表 2 中可以看出,程序执行了很多的回边,虽然 PAP 使用的断点数目很大,但是其中断点大量重复,相异的断点很少.对于相异断点进行编码,就可以节约大量的存储空间.以 *JGFArithBench* 为例,129 个相异断点的等长编码为 8bit,存储 4 107 380 个断点总共需要约 4MB 的存储空间.考虑到各个断点出现频率的差异,使用不等长编码能够进一步缩小存储空间.故使用 PAP 方法进行剖析在存储空间方面是可行的.

2.4.4 剖析能力比较

在已有的循环路径剖析方法中,只有文 kIPP 能够精确地剖析执行循环体多次的所有路径.该方法需要首先确定循环体执行次数的上界(假设为 k 次),满足该条件的静态路径称为 k 次迭代路径(k-iteration paths),然后按照在 CFG 上展开循环体、将循环路径转化为非循环路径的原理,按照类似于 EPP 方法的步骤进行插装,保证所有的 k 次迭代路径对应于不同的编码.这种方法的优点在于,它对 k 次迭代路径的编码是连续的,能够在一定程度上节约存储空间;其缺点在于需要在执行前确定 k 的值,不能够剖析执行中超出 k 次迭代范围的路径.此外,该方法能够处理的路径数目非常有限.文献[16]中的实验表明,如果所有的 k 次迭代路径的数目超过 6 000,则剖析信息的存储必须用链表结构;如果路径数目超过 60 000,则无法剖析.

依前文所述,我们在实验中将 60 000 条静态路径作为 kIPP 能够剖析的上限,然后计算了在 k 的不同取值情况下 9 个基准程序中 k 次迭代路径的数目,以此得出 kIPP 能够进行剖析的最大 k 值,见表 3.由表 3 中的结果可以看出,在大部分程序中,kIPP 的剖析能力明显不足:9 个程序里,其无法剖析的有 4 个,仅能剖析无环路径的有一个;在剩余的 4 个程序中,能剖析的最大 k 值不超过 7.与表 2 所示的执行回边数目相比可知,kIPP 难以剖析含有多次循环的执行.

Table 3 *k*-iteration paths of programs in JGF.section1

表 3 JGF.section1 中各个程序的 k 次迭代路径数目

基准测试程序	k=1(即非循环)	k=2	k=3	kIPP 能够剖析的最大 k 值
JGFArithBench	478 515 625	23 720 703 125	661 580 078 125	—
JGFAssignBench	9 765 625	361 328 125	7 658 203 125	—
JGFCastBench	625	9 625	99 225	2
JGFCreateBench	45 349 632	1 511 654 400	28 749 147 264	—
JGFExceptionBench	27	162	738	6
JGFLoopBench	27	135	495	7
JGFMathBench	9.31×10^{20}	1.02×10^{23}	—	—
JGFMethodBench	6 561	94 041	831 789	1
JGFSerialBench	108	1 026	7 110	4

下面我们在类似的 k 值约束下检验 PAP 方法的断点数目,以比较二者的空间耗费.k 的取值范围为 1~20,在 k 的每一个取值下,找到使用断点最多的 k 次迭代路径,记录其所需的断点数目,结果如图 8 所示.

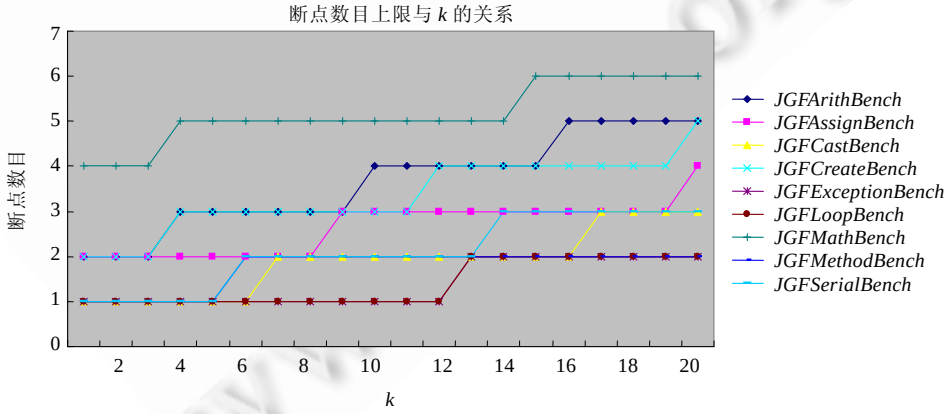


Fig.8 Most breakpoints needed by k-iteration paths

图 8 JGF.section1 断点数目上限与 k 次迭代路径的关系

从图 8 可以看出:首先,PAP 使用很小数量的断点(最多不超过 6 个),就能够精确地剖析 9 个程序的每条 20 次迭代路径;其次,断点数目上限随 k 的增长缓慢,相比之下,k 次迭代路径的数目随 k 的增大呈现爆炸式的增长,kIPP 受限于路径数目,难以运用于 k 值较大的情况;再次,虽然 *JGFMathBench* 的路径数目远远超出其他程序(如其非循环路径数目约为 *JGFArithBench* 非循环路径数目的 1.9×10^{12} 倍),但是断点数目上限却与其他程序相差不大(只比 *JGFArithBench* 多出 1~2 个),说明程序结构的复杂性对于 PAP 方法的空间耗费影响较小,因而对于更复杂的程序,PAP 方法也能够使用较低的空间耗费对 k 次迭代路径进行剖析.相比之下,kIPP 方法记录每一条路径需要一个长度为 k 的整型数组,依赖整个数组所存储的值来计算相应路径的编码,即其空间耗费与 k 成线性关系,要高于 PAP.

2.5 实验结论

由以上实验可以得出如下结论:

(1) PAP 的插装过程是高效的,时间耗费不超过 EPP 插装的两倍;

(2) PAP 的探针对被剖析程序的执行时间一般影响不大,除非该程序含有简单(执行一次迭代耗时很少)的循环体;而且该循环体的迭代次数很大,对于整个程序的执行时间有着决定性的影响;

(3) 即使对于比较复杂的程序中执行了大量回边的路径,PAP 的存储耗费依然是可行的,可以有效地应对已有剖析技术难以应用的程序结构复杂、循环次数过多的情况[5]***;

(4) 将 PAP 剖析的每条路径的空间耗费限制到很小的范围(数个断点)时,其剖析循环的能力也远远超出 kIPP 等方法;由于 PAP 只为剖析过程中执行的路径(而不是所有静态路径,类似于 kIPP 等方法)开辟存储空间,而据资料显示,静态路径中执行的路径仅占极小的部分,二者满足 100-0 法则[1],所以当每条路径的空间耗费很小时,整个剖析过程的空间耗费也非常有限;

(5) 使用断点机制能够有效地应对探针值溢出的问题,RAS 中结合 EPP 也能够有效地减少探针数目,这说明本文在第 1.2 节和第 1.4 节所提出两种改进方式是有效的.

2.6 进一步的探索:路径编码的紧凑性

EPP 方法给出的无环路径编码是紧凑的(编码是连续的整数);而 PAP 方法不能保证编码紧凑,会导致一些编码空间的浪费和存储耗费的增加.为了检验这种耗费,本节采用实验来检验其非紧凑性,定义如下:

$$\text{非紧凑性}(uncompactness)=\frac{k\text{次迭代路径的最大编码}}{k\text{次迭代路径总数}}.$$

针对 k 的取值从 2~6,我们统计了在 JGF 的 9 个程序上 PAP 编码的非紧凑性,结果见表 4.

Table 4 *Uncompactness* of programs in JGF.section1

表 4 JGF.section1 中各个程序的编码非紧凑性

基准测试程序	k=2	k=3	k=4	k=5	k=6
JGFArithBench	4.2×10^4	3.4×10^4	2.2×10^5	4.6×10^5	1.1×10^6
JGFAssignBench	2434.4	559.9	231.6	140.6	97.4
JGFCastBench	5.4	2.6	2.5	2.6	3.0
JGFCreateBench	1.5×10^6	6.5×10^5	7.3×10^5	9.9×10^5	1.5×10^6
JGFExceptionBench	1.8	1.2	1.4	1.5	1.7
JGFLoopBench	1.1	1.8	2.5	3.6	5.3
JGFMathBench	1.3×10^{12}	7.3×10^{11}	1.5×10^{11}	3.7×10^{10}	1.4×10^9
JGFMethodBench	101.9	36.0	22.6	15.4	11.6
JGFSerialBench	10.2	13.2	39.4	126.0	420.6

从结果可以看出,非紧凑性的值随着 k 值的增大(即路径变长)没有呈现统一的变化趋势.由于断点机制采用整型变量进行存储,而一个整型变量的取值范围约为 -2×10^9~2×10^9,而表中的非紧凑性最大值为 1.3×10^{12},故对

*** 不仅 kIPP 方法无法应用到多个基准测试程序上,而且 EPP 方法也不能正确地剖析 *JGFMathBench*,因为非循环路径数目过多,编码已经溢出.

于其中的一条路径来说,PAP 编码所需要的断点数目与紧凑编码(如果存在这样的有环路径编码的话)相比,多使用至多两个断点.

3 方法执行序列的获取

作为 PAP 方法的一个典型应用,本节讨论 PAP 方法在方法层次执行序列获取过程中是如何发挥作用的.很多动态影响分析(dynamic impact analysis)技术都需要方法层次的执行序列作为基础[17−21],针对这样的问题,最好能够抛开方法内部的细节,只进行方法级的剖析,但是目前还没有文献给出这样的剖析方法.文献[24]将 EPP 方法扩展到过程间调用的情况,但没有讨论如何忽略过程内部结构的问题.如果将这种方法应用于方法执行序列的获取,尽管可以合并与方法调用无关的节点进行化简,却依然不能脱离方法的内部结构;另一方面,由于现有方法处理循环的能力不足,难以处理以下情况:(1) 循环体内有方法调用;(2) 方法之间存在递归调用;(3) 某种方法中对同一个方法有多处调用.其中,情况(3)也可能在控制流图上形成循环.如图 9(a)所示,方法 *A* 中有两处调用了 *B*,那么 c_2-$Entry_B$-$Exit_B$-r_1 就构成了一个循环.虽然该静态路径不可执行,却依然需要剖析方法对其编码.为了处理多处调用引起的循环,Melski 等人在各个调用点生成被调用方法的一个拷贝,以使多个调用点互不影响,但是该方法耗费太高[15].

方法调用图表达了方法之间的调用关系,不包含方法的内部结构,非常简洁.我们只需为方法调用图扩充返回边,即可直接应用 PAP 方法进行剖析.图 9(b)中给出了一个扩充后的调用图,增加了返回边(虚线表示),并为 *Main*(·)方法添加了入口和出口.图中包含大量的循环,这些都可以由 PAP 方法有效处理.

与剖析 CFG 中的路径类似,PAP 探针插装算法可以直接在扩充后的方法调用图上进行,然后将探针插装到边所对应的调用点.如果在探针插装算法的结果中,一条调用边(返回边)上有探针,则将该边对应的调用者的调用点紧邻的前(后)作为探针的插装位置.对于前述 3 种引起循环的情况:(1) 因为 PAP 可以忽略方法内部结构,所以不需要考虑方法调用点所在的控制结构;(2) 递归调用形成的循环与 CFG 中的循环结构没有本质的区别,可以应用 PAP 剖析循环的能力;特别地,如图 9(c)所示,如果两个方法相互调用,那么在改进后的方法调用图中就会出现两条边源节点和目标节点相同的情况,这和 CFG 的形式有所不同.但是由于 PAP 方法对节点的不同入边进行区分,且插装探针的位置在调用者中,所以这并不会影响 PAP 的应用;(3) 因为 PAP 探针插装算法的结果仅与扩充后的方法调用图的边相关,而方法内对同一种方法的多处调用对应于同一条调用边和返回边,所以如果相应的边上有探针,那么在每个调用点插装相同的探针即可.

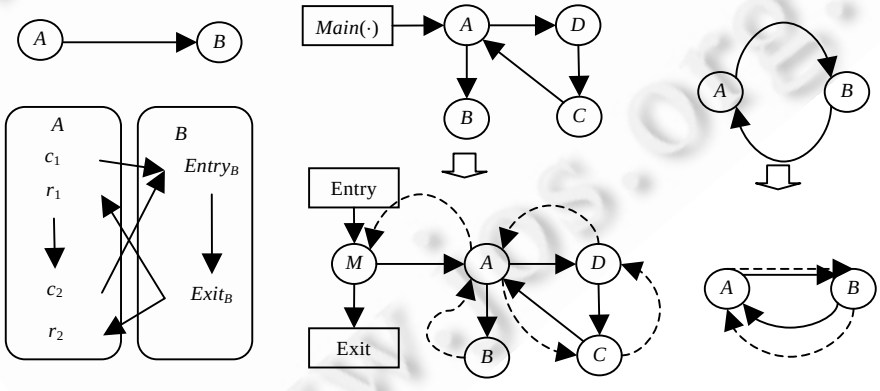


Fig.9 Examples of call graphs

图 9 方法调用示例

图 10 给出了使用 PAP 进行方法级剖析的示例,图的左边是一个带有递归调用的方法调用图,经过扩充后,增加了返回边和 *Main*(·)方法的入口和出口,并应用了 PAP 探针插装算法.图中还给出了应用该插装结果后,部

分方法序列和探针值的对应关系(执行序列中的“*r*”表示返回至调用者).

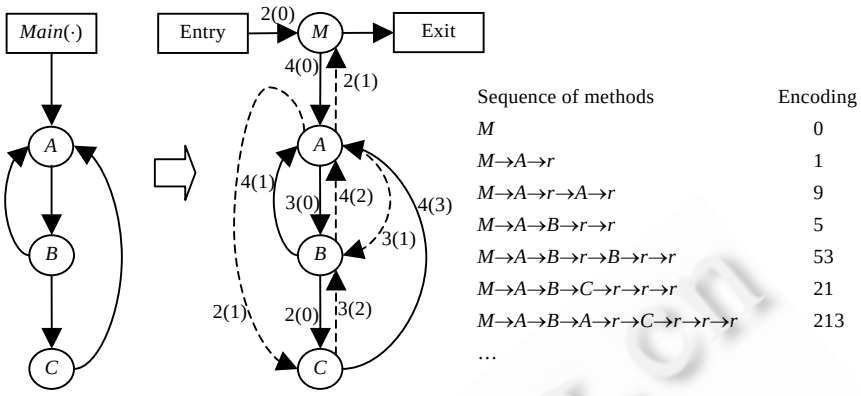


Fig.10 An example of profiling call graphs

图 10 方法调用图剖析示例

4 相关工作比较

针对非循环路径的主要剖析方法是:Apiwattanapong 等人提出了 SPP(selective path profiling)方法,通过仅对非循环路径的子集进行剖析来减少不必要的插装[12];Joshi 等人提出了 TPP(targeted path profiling)方法,通过处理明显路径(obvious path)和冷边(cold edge)降低剖析的耗费[13];Bond 等人提出了 PPP(practical path profiling)方法,通过在 TPP 方法的基础上做出改进减少插装耗费和提高剖析效率[9];Vaswani 等人则提出了 Preferential Path Profiling 方法,通过优化路径编码的方式提高存储效率,减小执行的时间耗费[14].

而在循环路径的剖析方面,Tallam 等人提出的方法是对 BL 路径(Ball-Larus paths)进行了扩充,以实现更多路径(执行循环体两次以内)的剖析,这种方法耗费较大而且是非精确的[15];Roy 等人改进了 Tallam 的方法,能够精确剖析包含循环体多次执行的路径,但是由于剖析过程严重受限于静态路径数目,难以应用于路径较多的情况(原文中实验将路径数目限制为 60 000 条,当循环 3 次时,部分实例已经超出限制,无法剖析)[16].

此外,Duesterwald 等人提出 NET(next executing tail)方案来推测热路径(执行次数多的路径),于在线环境下,比使用剖析方法具有更低的耗费[25];Vaswani 等人提出基于硬件支持的可编程路径剖析器,能够使用较低的耗费获取比较准确的热路径表[26];Yasue 等人提出 Structural Path Profiling 方法,按照循环结构将过程内控制流图转换为多个分层嵌套的图,对每个图独立剖析,具有较高的精确性(约 90%),并能够较好地支持 JIT(just in time)编译器[27].

PAP 方法可以处理由循环引起的任意长度和任意数目的路径,其中的回溯技术可以避免对未执行路径的处理,能够对执行的各条路径进行独立剖析.理论和实验结果表明,PAP 方法的插装过程是高效的、时间耗费是合理的;在空间耗费方面,因为 PAP 方法的插装能够保证每条静态路径对应唯一的编码,所以在剖析过程中只需对执行的路径进行存储,由于被执行路径仅占静态路径极小的部分,所以使用这种存储方式能够有效地突破静态路径数目限制、节约存储空间,提高实用性.此外,PAP 方法还可以忽略方法内部结构,直接获取方法层次的执行序列.表 5 从多个方面对 PAP 与相关的剖析方法进行了比较.

Table 5 Comparison of profiling techniques

表 5 PAP 与相关剖析方法比较

剖析方法	EPP	文献[9,12−14]	文献[15,16]	PAP
耗费比较	—	低于 EPP	数倍于 EPP	数倍于 EPP
剖析对象	BL 路径	BL 路径的子集	*k* 次迭代路径	任意有限长度的路径
精确程度	精确	精确	文献[15]略有误差,文献[16]精确	精确
扩展至过程间	已扩展	未扩展	文献[15]已扩展,文献[16]未扩展	已扩展

5 总结与展望

本文提出了全路径剖析方法 PAP,该方法通过区分同一个节点的不同入边来区分不同的执行路径,并结合断点机制可以有效剖析带任意次循环的路径.文中分析并比较了 PAP 与 EPP 方法的探针数目:在剖析不带循环的路径时,PAP 方法的探针数目略多于 EPP 方法.为此,文中将 EPP 与 PAP 相结合,通过在可规约无环子图中应用 EPP 方法,可以减少 PAP 方法在处理不带循环的路径时所需探针的数目;文中还通过随机生成的 3 个控制流图及多条路径对断点数目进行了实验,结果表明,使用少量的断点就可以对很长的路径进行剖析;通过随机生成的可规约无环子图对探针数目进行了实验,结果表明,结合 EPP 方法后能够有效地减少探针数目,提高了效率;通过基准测试程序 *lufact* 对 PAP 和 EPP 方法的耗费进行了实例分析,结果表明,PAP 方法的效率略低于 EPP,但是 PAP 可以精确剖析带循环的路径;通过基准测试程序 JGF.section1 中的 9 个过程对 PAP 和相关方法进行了全面的比较,结果表明,PAP 的时间耗费在大部分情况下仅略多于 EPP,空间耗费较为实用,且将空间耗费限定在较小的范围之内时,剖析能力依然强于现有方法.作为 PAP 方法的一个典型应用,文中还讨论了如何将 PAP 应用于方法执行序列的获取.

PAP 方法的缺点在于,虽然剖析循环的能力比现有方法更高,但是对于较长的路径需要较多的空间耗费;PAP 不能保证路径编码是连续的,存在某些编码不对应路径的情况,在一定程度上浪费了存储空间,但是无环子图中结合 EPP 的方法可以改善这一状况.另外,通过比较我们可以发现,虽然已有的各种剖析技术存在着诸多限制,但是同时也有各自的优点和适用的场景.如果能有适当的途径将多种剖析方法相结合,充分发挥其优点,就会获得更好的效果.

References:

[1] Ball T, Larus JR. Programs follow paths. Technical Report, MSR-TR-99-01, Washington: Microsoft Research, 1999.

[2] Ball T, Mataga P, Sagiv M. Edge profiling versus path profiling: The showdown. In: Proc. of the 25th ACM SIGPLANSIGACT Symp. on Principles of Programming Languages. New York: ACM Press, 1998. 134−148 [doi: 10.1145/268946.268958]

[3] Anderson JM, Berc LM, Dean J, Ghemawat S, Henzinger MR, Leung STA, Sites RL, Vandevoorde MT, Waldspurger CA, Weihl WE. Continuous profiling: Where have all the cycles gone? Journal of ACM, 1997,15(4):357−390. [doi: 10.1145/265924.265925]

[4] Zhang XL, Wang Z, Gloy N, Chen JB, Smith MD. System support for automatic profiling and optimization. Journal of ACM, 1997, 31(5):15−26. [doi: 10.1145/269005.266640]

[5] Conte TM, Menezes KN, Hirsch MA. Accurate and practical profile-driven compilation using the profile buffer. In: Proc. of the 29th Annual ACM/IEEE Int'l Symp. on Microarchitecture. Washington: IEEE Computer Society, 1996. 36−45. [doi: 10.1109/MICRO.1996.566448]

[6] Conte TM, Patel BA, Cox JS. Using branch handling hardware to support profile-driven optimization. In: Proc. of the 27th Annual Int'l Symp. on Microarchitecture. New York: ACM Press, 1994. 12−21. [doi: 10.1145/192724.192726]

[7] Dean J, Hicks JE, Waldspurger CA,Weihl WE, Chrysos G. ProfileMe: Hardware support for instruction-level profiling on outof-order processors. In: Proc. of the 30th Annual ACM/IEEE Int'l Symp. on Microarchitecture. Washington: IEEE Computer Society, 1997. 292−302. [doi: 10.1109/MICRO.1997.645821]

[8] Heil T, Smith JE. Relational profiling: enabling thread-level parallelism in virtual machines. In: Proc. of the 33rd Annual ACM/IEEE Int'l Symp. on Microarchitecture. New York: ACM Press, 2000. 281−290. [doi: 10.1145/360128.360156]

[9] Bond MD, McKinley KS. Practical path profiling for dynamic optimizers. In: Proc. of the Int'l Symp. on Code Generation and Optimization (CGO). Washington: IEEE Computer Society, 2005. 205−216. [doi: 10.1109/CGO.2005.28]

[10] Ball T, Larus JR. Optimally profiling and tracing programs. ACM Trans. on Programming Languages and Systems, 1994,16(4): 1319−1360. [doi: 10.1145/183432.183527]

[11] Ball T, Larus JR. Efficient path profiling. In: Proc. of the 29th Annual ACM/IEEE Int'l Symp. on Microarchitecture. Washington: IEEE Computer Society, 1996. 46−57.

[12] Apiwattanapong T, Harrold MJ. Selective path profiling. In: Proc. of the 2002 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering. New York: ACM Press, 2002. 35−42. [doi: 10.1145/634636.586104]

[13] Joshi R, Bond MD, Zilles CB. Targeted path profiling: Lower overhead path profiling for staged dynamic optimization systems. In: Proc. of the Int'l Symp. on Code Generation and Optimization (CGO). Washington: IEEE Computer Society, 2004. 239−250. [doi: 10.1109/CGO.2004.1281678]

[14] Vaswani K, Nori AV, Chilimbi TM. Preferential path profiling: Compactly numbering interesting paths. In: Proc. of the 34th Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL). New York: ACM Press, 2007. 351−362. [doi: 10.1145/1190215.1190268]

[15] Tallam S, Zhang X, Gupta R. Extending path profiling across loop backedges and procedure boundaries. In: Proc. of the Int'l Symp. on Code Generation and Optimization: Feedback-Directed and Runtime Optimization (CGO). Washington: IEEE Computer Society, 2004. 251−264. [doi: 10.1109/CGO.2004.1281679]

[16] Roy S, Srikant YN. Profiling *k*-iteration paths: A generalization of the Ball-Larus profiling algorithm. In: Proc. of the 2009 Int'l Symp. on Code Generation and Optimization (CGO). Washington: IEEE Computer Society, 2009. 70−80. [doi: 10.1109/CGO.2009.11]

[17] Breech B, Tegtmeyer M, Pollock L. A comparison of online and dynamic impact analysis algorithms. In: Proc. of the 9th European Conf. on Software Maintenance and Reengineering (CSMR 2005). Washington: IEEE Computer Society, 2005. 143−152. [doi: 10.1109/CSMR.2005.1]

[18] Orso A, Apiwattanapong T, Harrold MJ. Leveraging field data for impact analysis and regression testing. In: Proc. of the ACM SIGSOFT Symp. on Foundations of Software Engineering. New York: ACM Press, 2003. 128−137. [doi: 10.1145/949952.940089]

[19] Law J, Rothermel G. Incremental dynamic impact analysis for evolving software systems. In: Proc. of the Int'l Symp. on Software Reliability Engineering (ISSRE). Washington: IEEE Computer Society, 2003. 430−441. [doi: 10.1109/ISSRE.2003.1251064]

[20] Law J, Rothermel G. Whole program path-based dynamic impact analysis. In: Proc. of the 25th Int'l Conf. on Software Engineering. Washington: IEEE Computer Society, 2003. 308−318. [doi: 10.1109/ICSE.2003.1201210]

[21] Apiwattanapong T, Orso A, Harrold MJ. Efficient and precise dynamic impact analysis using execute-after sequences. In: Proc. of the 27th Int'l Conf. on Software Engineering. New York: ACM Press, 2005. 432−441. [doi: 10.1145/1062455.1062534]

[22] Zhao KJ, Shen ZY, Trans. Advanced Compiler Design and Implementation. Beijing: China Machine Press, 2005. 129 (in Chinese).

[23] JGF. http://www2.epcc.ed.ac.uk/computing/research_activities/java_grande/seq/download.html

[24] Melski D, Reps TW. Interprocedural path profiling. In: Proc. of the 8th Int'l Conf. on Compiler Construction, Held as Part of the European Joint Conf. on the Theory and Practice of Software (ETAPS'99). London: Springer-Verlag, 1999. 47−62.

[25] Duesterwald E, Bala V. Software profiling for hot path prediction: Less is more. ACM SIGPLAN Notices archive, 2000,35(11): 202−211. [doi: 10.1145/356989.357008]

[26] Vaswani K, Thazhuthaveetil MJ, Srikant YN. A programmable hardware path profiler. In: Proc. of the Int'l Symp. on Code Generation and Optimization. Washington: IEEE Computer Society, 2005. 217−228. [doi: 10.1109/CGO.2005.3]

[27] Yasue T, Suganuma T, Komatsu H, Nakatani T. An efficient online path profiling framework for Java just-in-time compilers. In: Proc. of the 12th Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT 2003). Washington: IEEE Computer Society, 2003. 148.

附中文参考文献:

[22] 赵克佳,沈志宇,译.高级编译器设计与实现.北京:机械工业出版社,2005.129.

王璐璐(1985－),男,江苏赣榆人,博士生,主要研究领域为动态程序分析.

周晓宇(1972－),男,博士,副教授,主要研究领域为软件分析与理解,逆向工程与再工程.

李必信(1969－),男,博士,教授,博士生导师,CCF 高级会员,主要研究领域为软件建模、分析、测试与验证,软件维护相关技术.