一种基于高阶函数摘要的依赖簇检测方法

杨嘉毅,张迎周*,李俊锋,马 锐,汪全盛,薛渝川
(南京邮电大学计算机学院、软件学院、网络空间安全学院,江苏南京 210023)

摘 要: 依赖簇是相互依赖的程序组件的最大集合,依赖簇中任意一点产生变动都会引起其他组件的连锁反应.在实际生产环境中,依赖簇检测对于软件理解、测试、维护具有非常重要的意义.传统的依赖簇检测方法基于系统依赖图(System Dependence Graph,SDG)实现过程间依赖关系的计算.但是SDG的构建过程比较复杂,时间空间的占用比较大.为提高依赖簇检测的效率并减少空间占用,本文提出了一种有效的轻量级依赖簇检测方法.该方法通过构建每个过程高阶函数形式的函数摘要,将形参和全局变量的数据依赖作为摘要参数,并用函数摘要的参数初始化过程内依赖信息.通过在调用点处对高阶函数形式的摘要进行实例化,即可将调用过程的依赖关系通过摘要参数进行传递,从而获取过程间的依赖信息.为了进一步提升检测效率,我们还提出了基于自适应计算的依赖簇检测优化策略,该策略可以减少因函数的递归调用产生的相关冗余计算.本文选取了不同规模不同领域的工程项目和基准测试集进行相关对比实验,结果表明:基于高阶函数的依赖簇检测方法相比系统依赖图的检测方法,能够提升2.689倍的分析效率并减少35.7%的空间占用;基于自适应计算的依赖簇检测优化策略在高阶函数方法的基础上能够减少56.7%的冗余计算,提升23.9%的分析效率.

关键词: 依赖簇;高阶函数;自适应计算;系统依赖图;形式概念分析
基金项目: 国家自然科学基金(No.62272214)
中图分类号: TP311 **文献标识码:** A **文章编号:** 0372-2112(2024)04-1337-12
电子学报URL:http://www.ejournal.org.cn **DOI:**10.12263/DZXB.20230862

A Dependence Clusters Detection Method Based on Higher-Order Function Summary

YANG Jia-yi, ZHANG Ying-zhou*, LI Jun-feng, MA Rui, WANG Quan-sheng, XUE Yu-chuan
(*School of Computer Science, Nanjing University of Posts and Telecommunications, Nanjing, Jiangsu* 210023, *China*)

Abstract: Dependence clusters are the largest sets of interdependent program components, where any change in one point of the cluster can trigger a chain reaction in other components. In practical production environments, detecting dependence clusters is of great importance for software understanding, testing, and maintenance. Traditional dependence cluster detection methods rely on the system dependence graph (SDG) to calculate dependencies, However, constructing an SDG is a complex process that incurs significant time and space costs when analyzing large programs. In order to improve the efficiency of cluster detection, this paper proposes a light-weight and efficient dependence cluster detection method based on higher-order functions, which can avoid constructing SDG in the analysis. This method constructs a function summary in the form of a higher-order function for each procedure, where the data dependencies of formal parameters and global variables are used to initialize the dependence inside the procedure. The dependence cluster information between procedures can be obtained by instantiating the function summary and passing the dependence through summary parameters at the call site, which avoids the construction of an SDG. To further improve the analysis efficiency of the higher-order function-based detection method, we propose an optimization strategy based on adaptive computing, which significantly reduces the redundant calculations caused by the mutual recursive calls between functions. In the end, we select benchmark test sets with different scales and fields and conduct relevant experiments, which demonstrate that the proposed program dependence cluster detection method based on higher-order functions can improve analysis efficiency by 268.9% and reduce space usage by 35.7% compared with the detection method based on SDG. The optimization strategy based on adaptive computing can reduce redundant calculations by

收稿日期:2023-09-13;修回日期:2023-12-08;责任编辑:覃怀银
*通讯作者:张迎周

56.7% and improve analysis efficiency by 23.9% compared with the method based on higher-order functions.

Key words: dependence cluster; higher-order function; adaptive computing; system dependence graph; formal concept analysis

Foundation Item(s): National Natural Science Foundation of China (No.62272214)

1 引言

随着软件系统的规模普遍增加,软件系统中的代码/项目之间的依赖关系也在不断复杂化,其中能够在程序中引起“涟漪效应”的相互依赖语句的集合引起了国内外学者的关注. Binkley 和 Harman 等[1]将程序中相互依赖的语句组成的集合定义为依赖簇(dependence cluster),并提出了基于程序切片的依赖簇定义. 他们通过大量的实例研究,发现依赖簇广泛存在于各种开源软件和商业软件中[2]. 依赖簇的检测与收集在软件工程中具有广阔的前景,可以有效提高代码的维护效率.

在随后的研究中,Binkley 等[2]通过单调切片尺寸图(Monotonic Slice-size Graph,MSG)方法来检测依赖簇,证实了依赖簇的广泛存在,并且证明基于 MSG 的近似方法在大型程序中可以实现超过 99% 的精度,但该分析仍存在误报的可能性. Islam 和 Krinke 在 Binkley 等[3]在先前的研究基础上,基于前向切片,进一步提出了相干依赖簇(coherent dependence cluster)的概念,有助于理解大型程序逻辑上和功能上的组成. 程克[4]针对 MSG 在依赖簇检测准确度方面的不足,提出了应用形式概念分析(Formal Concept Analysis,FCA)进行依赖簇检测的方法. 它的基本思想是将源代码中获取的依赖关系整理为形式背景(formal context),并根据概念格(concept lattice)构造算法生成概念. 这一方法尽管相比 MSG 方法在精度上有所提升,但是文中仍然需要通过构建系统依赖图(System Dependence Graph,SDG)获取依赖关系.

在基于 SDG 的分析算法当中,SDG 的生成占据了应用 SDG 的分析算法中相当一部分开销,而在生成 SDG 的过程当中,根据过程间依赖关系(过程间的数据依赖与控制依赖),SDG 算法往往会产生额外的边与节点,这些过程间边与节点会随着调用关系的增多而增加. 此外,为了实现上下文敏感,基于 SDG 的算法通常还需要对 SDG 进行额外的处理或是附加信息(如括号匹配法),在对存在复杂调用情况的待测程序的分析当中这将引入较高的额外开销.

为此,本文提出一种有效的轻量级依赖簇检测方法. 该方法通过构建每个过程高阶函数形式的函数摘要来处理过程间的依赖信息. 本文也将借助于形式概念分析方法进行依赖簇检测,但不再需要构建复杂的系统依赖图,而是通过高阶函数形式的函数摘要进行依赖关系的收集,从而减少了时间空间占用. 此外,为了进一步提升检测效率,我们将自适应计算(adaptive computing)[5]应用于含有递归调用的依赖簇检测中,进一步提高过程间分析的效率. 自适应计算可以根据输入数据和运行环境动态地调整计算过程和结果[6],在回归测试,编译器优化,数据流分析,程序切片等领域得到了广泛应用[7].

2 基于高阶函数的依赖簇检测方法

2.1 基本思想和整体框架

本文将选择 LLVM IR 作为分析的目标语言,通过构建高阶函数式的函数摘要来进行依赖簇的检测. 将形参和全局变量的数据依赖作为摘要参数,初始化过程内的依赖信息. 利用这些过程内高阶函数形式的摘要,在调用点处对摘要的参数进行相应代入,即可获取过程间因参数传递而产生的依赖信息. 最后,将计算得到依赖信息进行汇总,整理为形式背景三元组,通过概念格构造算法生成概念格,进而获取相互依赖的程序指令集合,从而检测出程序中出现的依赖簇. 高阶函数形式的函数摘要可以通过式(1)构建.

$$\lambda P_{\text{set}} \cdot (R_1, R_2, \cdots, R_n) \tag{1}$$

定义 $P_{\text{set}}=(f,g)$ 作为 Lambda 表达式的参数. f、g 分别是过程内形参和全局变量的数据依赖. $R_1, R_2, \cdots, R_n$ 是一系列含有参数 f、g 的依赖关系集合. 这样,一个过程的依赖分析结果以匿名函数的形式进行保存,并作为其他函数分析过程的参数,与传统的非函数式的摘要不同[8],我们称之高阶函数形式的摘要. 本方法整体框架如图 1 所示.

2.2 过程内依赖分析的流程及算法

过程内分析的基本思想是定义两种映射表形式的数据结构,即变量依赖表 S 和指令依赖表 L. 其中 S 用于存储参数、全局变量、局部变量的依赖关系,L 用于存储指令的依赖关系. 通过数据流分析遍历每条指令,计算每条指令对应的依赖关系,并根据计算结果相应更新 S、L.

在通过程序的控制流图(Control Flow Graph,CFG)进行分析时采用 in 和 out 函数来描述每个基本块(即 CFG 中的节点)的状态,函数如式(2)所示.

$$\begin{aligned} \text{in}(B) &= \bigcup_{B' \in \text{pred}(B)} \text{out}(B') \\ \text{out}(B) &= \text{gen}(B) \cup (\text{in}(B) - \text{kill}(B)) \end{aligned} \tag{2}$$

其中,pred(B)表示基本块 B 的前驱基本块的集合;in(B)表示当前流入基本块 B 的数据流值;out(B)表示当前基本块 B 流出的数据流值;gen(B)表示当前基本块 B 中所有计算所产生的数据的集合;kill(B)表示当前基

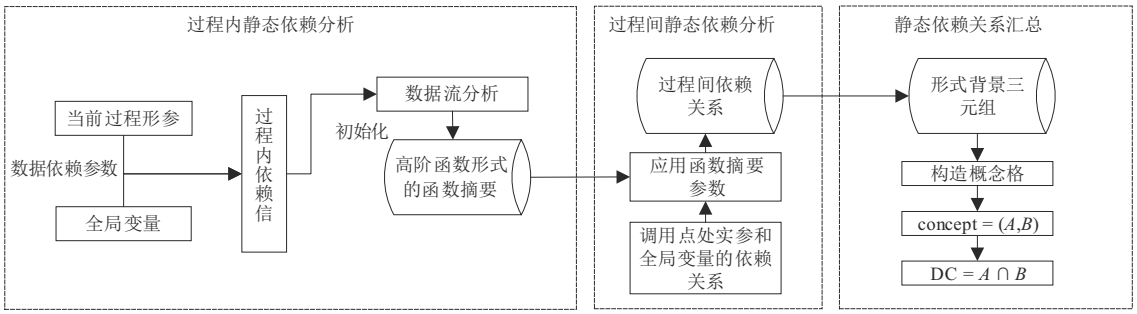

图 1　基于高阶函数的依赖簇检测方法整体框架

本块B中被重新赋值的数据的集合.

对每个过程,使用弱拓扑序(Weak Topological Ordering,WTO)作为数据流迭代策略[9]. 该策略的基本思想是将CFG按照强连通分量(Strong Connected Component,SCC)和顶点(Vertex)进行划分,并给出CFG的一个总体拓扑序,数据流迭代时便按照这一顺序对每个基本块内的指令进行依赖关系计算. 其中,Vertex表示单一基本块,SCC则是由基本块构成的强连通分量,WTO迭代方法下的SCC是一种递归定义的数据结构,即SCC内部可能继续包含Vertex或SCC,其结构为<头部基本块B,SCC/Vertex>,而Vertex是只包含单一基本块.

过程内依赖分析算法分为依赖信息初始化,依赖关系计算和高阶函数形式摘要生成三个部分. 在进行过程内分析之前,需要根据函数调用图的逆拓扑排序自底向上进行子过程的分析.

(1)依赖信息初始化

设当前待分析的过程为P,在分析开始时定义参数P_{set},包含过程P形参的数据依赖f,全局变量的数据依赖g. 即$P_{\text{set}}=(f,g)$并且f、g都是集合形式,如式(3)所示:

$$f=\{f_1,f_2,\cdots,f_n\},g=\{g_1,g_2,\cdots,g_m\} \tag{3}$$

在定义了P_{set}之后,初始化指令依赖表L和存储基本块出口数据流值的变量out为空.

其次,对于变量依赖表S,针对P内的形参和全局变量,分别将其变量依赖表初始化为f_n或g_m. 将f_n赋值给P的第n个形参,g_m赋值给第m个全局变量. 与此同时,针对局部变量,将变量依赖表初始化为空. $S(x)$的计算公式如式(4)所示.

$$S(x)\equiv\begin{cases}f_n, & x\in \text{FRML}(P),x\text{为第}n\text{个形参}\\ g_m, & x\in \text{GLOB},x\text{为第}m\text{个全局变量}\\ \varnothing, & \text{局部变量}\end{cases} \tag{4}$$

其中,FRML(P)是过程P形参的集合,GLOB是所有过程的全局变量集合.

(2)依赖关系计算

在完成依赖信息的初始化步骤之后,需要通过弱拓扑排序获取包含SCC/Vertex子结构的队列W. 数据流分析便按照W中的出列顺序依次进行每个基本块的遍历,进行依赖关系的计算. 分为以下几个步骤:

步骤1:取W中的第一个子结构c,获取子结构c的入口基本块B.

步骤2:根据式(2)计算当前基本块B入口处的数据流值(L,S),遍历B中的每一条指令,根据当前访问的指令类型及计算指令依赖,更新指令依赖表L和变量依赖表S.

指令依赖$L(i)$的计算方法如式(5)所示,包含了指令的数据依赖和控制依赖. 其中REF为指令i的使用集. CD(i)是对i具有控制关系的指令集合.

$$L(i)=\bigcup_{x\in \text{REF}(i)}S(x)\bigcup_{i'\in \text{REF}(i)}L(i')\cup\bigcup_{j\in \text{CD}(i)}L(j)\cup L(i) \tag{5}$$

针对赋值指令或内存修改指令i,更新该指令i中所定义变量x的变量依赖表$S(x)$为$L(i)$,如果当前指令为phi指令或select指令,则与之前的$S(x)$合并,即式(6):

$$S(x)=\begin{cases}L(i), & x\in \text{DEF}(i),x\text{是 single-valued}\\ S(x)\cup L(i), & x\in \text{DEF}(i),x\text{是 multi-valued}\\ \varnothing, & \text{其他变量}\end{cases} \tag{6}$$

步骤3:当前基本块B的终止指令分析完毕后按照式(7)更新变量out.

$$\text{out}(B)=(S,L) \tag{7}$$

如果步骤1中的子结构c是SCC结构,由于SCC内部可能包含更小单元的SCC和Vertex,因此需要递归进行内部Vertex和SCC的数据流分析,流入基本块B的数据流值保持不变. 针对Vertex和SCC的依赖分析算法可见算法1.

(3)高阶函数形式摘要生成

对于过程P的高阶函数形式摘要T_p如式(8)所示,其参数为过程内依赖信息初始化时定义的参数集合P_{set},返回值为结束指令(ret,unreachable)所在基本块流出的变量依赖表S和指令依赖表L.

算法1 针对Vertex, SCC的静态依赖分析算法

输入:Vertex或SCC类型的变量c,变量依赖集S,指令依赖集L,指令分析方法AnalysisInst,存放出口数据流值的变量out

输出:更新后的存放出口数据流值的变量out

```
1:FUNCTION  AnalysisComponent(c,out,S,L,AnalysisInst)
2:   IF c是一个Vertex实例 THEN
3:    B ← c //将c带入到基本块B
4:    out ← AnalysisVertex(B,out,S,L,AnalysisInst)
5:   ELSE
6:    (B,xs) ← c //xs指代该SCC中的第二个变量
7:    REPEAT
8:     s0 ← ⋃ out(B′)   (B′∈pred(B))
9:     out← AnalysisVertex(B,out,S,L,AnalysisInst)
10:    out← AnalysisComponent(xs,out,S,L,AnalysisInst)
11:    s1 ← ⋃ out(B′)   (B′∈pred(B))
12:   UNTIL s0 = s1//对SCC的分析达到不动点
13:  END IF
14:RETURN out
15:END FUNCTION
16:
17:FUNCTION  AnalysisVertex(B,out,S,L,AnalysisInst)
18:  IF B不是G的入口基本块 THEN
19:    (S,L) ← ⋃ out(B′)   (B′∈pred(B))
20:  END IF
21:  FOR 基本块B中的各个节点i DO
22:    (S,L) ← AnalysisInst(i,S,L)
23:  END FOR
24:  out(B) = (S,L)
25:RETURN out
26:END FUNCTION
```

$$(S,L)=\bigcup_{B'\in \mathrm{exit}(G)}\mathrm{out}(B')$$
$$T_P=\lambda P_{\mathrm{set}}\cdot(S,L) \tag{8}$$

2.3 过程间依赖分析的流程及算法

当被调用函数的依赖关系计算完毕,并生成高阶函数形式的函数摘要后,就可以在调用该函数的指令处,实例化函数摘要并将函数摘要的参数进行代入,从而获取过程间的依赖信息. 过程间依赖分析共有三个部分.

(1)调用点处依赖关系的收集

设当前分析过程为过程P,在函数调用指令处,收集调用点处的依赖信息到元组$P_{\mathrm{set,callsite}}$中. 如式(9)所示,调用点处的依赖信息包含函数调用指令中的实参的依赖信息,以及过程P中全局变量的依赖信息.

$$P_{\mathrm{set,callsite}}=(f_P,g_P) \tag{9}$$

(2)函数摘要的实例化

设当前调用的函数为Q,根据式(10)获取函数Q的高级函数形式的函数摘要T_Q.

$$T_Q=\lambda P_{\mathrm{set}}\cdot(S_Q,L_Q) \tag{10}$$

在完成了调用点处依赖关系的收集之后,就可以根据将式(9)定义的$P_{\mathrm{set,callsite}}$代入到$T_Q$中,获取过程$Q$的变量依赖表$S_Q$和指令依赖表$L_Q$.

(3)计算过程间依赖关系

在获取了过程Q的变量依赖表S_Q和指令依赖表L_q之后,需要计算过程间的依赖信息. 这个过程包含了三个部分. 首先是计算过程Q对调用点实参的依赖产生的影响,其次是将过程Q内指令依赖表和当前过程的指令依赖表进行合并. 最后利用Q中返回指令的依赖信息更新当前函数调用指令所定义变量的依赖.

首先,根据Q的变量依赖表判断Q中形参或全局变量的调用类型.

变量调用类型的判断方法如式(11)所示. 如果S_Q中的形参或全局变量依赖集合不等于依赖信息初始化时使用的摘要参数,则说明该变量是按结果调用类型(OUT(Q)),否则为按值调用类型(IN(Q)),即

$$\begin{aligned}
&\mathrm{OUT}_F(Q)=\{x\mid S_Q(x)\neq f_x,x\in \mathrm{FRML}(Q)\}\\
&\mathrm{OUT}_G(Q)=\{x\mid S_Q(x)\neq g_x,x\in \mathrm{GLOB}\}\\
&\mathrm{IN}_F(Q)=\{x\mid S_Q(x)=f_x,x\in \mathrm{FRML}(Q)\}\\
&\mathrm{IN}_G(Q)=\{x\mid S_Q(x)=g_x,x\in \mathrm{GLOB}\}\\
&\mathrm{OUT}(Q)=\mathrm{OUT}_F(Q)\cup\mathrm{OUT}_G(Q)\\
&\mathrm{IN}(Q)=\mathrm{IN}_F(Q)\cup\mathrm{IN}_G(Q)
\end{aligned} \tag{11}$$

其次,如式(12)所示,对于Q中按结果调用的形参类型或全局变量x,将其变量依赖$S_Q(x)$与调用点处的控制依赖进行合并,更新到对应实参(式中,actual(x)表示Q的形参x在调用过程P中对应的实参)的变量依赖表中. 与此同时,如式(13)所示,将L_Q中的每个元素也和调用点处的控制依赖进行合并.

$$S_P(\mathrm{actual}(x))=S_Q(x)\cup\bigcup_{j\in \mathrm{CD}(i)}L_P(j) \tag{12}$$

$$L_P(i')=L_P(i')\cup\bigcup_{j\in \mathrm{CD}(i)}L_P(j) \tag{13}$$

最后,利用返回指令对应的依赖信息更新当前调用指令定义变量DEF(i)的依赖,并将经过式(13)更新后的L_Q与调用过程当前的指令依赖表L_P进行合并,从而获取过程间的指令依赖信息.

在精度方面,本文有如下的两个定理可保证本文提出的基于高阶函数的依赖簇检测方法(dependence Cluster detection Algorithm based on Higher-Order function,HOCA)中涉及的依赖计算过程与基于SDG进行过

程内和过程间依赖关系计算的准确性基本相同.

定理1　完备性　对于系统依赖图中的每个节点n(不考虑函数入口节点),都可以在HOCA方法中找到与之对应的依赖关系.

定理2　可靠性　HOCA的过程内和过程间依赖计算方法与系统依赖图中的依赖计算方法等价.

2.4　依赖关系汇总和依赖簇检测

当程序内所有过程的依赖分析结束后,通过将得到的依赖关系进行汇总,整理为形式背景并构造形式概念格,就可以检测程序中出现的依赖簇并判断依赖簇类型.

程克应用形式概念分析进行依赖簇检测的基本思路如下:通过SDG图计算每个节点的后向切片,可以得到语句和它具有的属性(依赖关系),以此构成形式背景三元组$K=(G,M,I)$,其中G,M代表语句集合,I表示语句间是否依赖.再通过概念格生成算法得到概念格.概念格中每一对概念concept=(A,B),表示具有相同依赖B的语句集合A.

依赖簇DC最终可用A与B的交集表示,这是因为A与B的交集中的节点都拥有影响集B,被影响集B中所有节点影响,而且还相互存在于彼此的后向切片中.

程克在依赖簇检测中沿用了Binkley在判断依赖簇类型时设定的10%阈值[10],将依赖簇分为3类:小型SS(m<10%且g<10%),中型SL(g<10%且m>10%),大型LL(g>10%且m>10%).依赖簇类型反映了程序中存在相互依赖关系语句的规模.

本文方法检测依赖簇及其类型的方法主要有以下几个步骤:

步骤1:将程序中指令的集合INST作为形式背景中的对象集和属性集.通过指令依赖表L获取两个指令间是否存在依赖关系,以此构造形式概念格.

步骤2:针对步骤1构造的概念格中的每一对概念concept=(A,B),计算A和B的交集,从而检测出给定输入条件下的依赖簇.

步骤3:计算每一对概念concept=(A,B)的属性包含度g、对象包含度m如式(14)和式(15)所示:

对象包含度:

$$g=\frac{|A|}{|\mathrm{INST}|} \tag{14}$$

属性包含度:

$$m=\frac{|B|}{|\mathrm{INST}|} \tag{15}$$

整体包含度c为对象包含度和属性包含度的乘积.

步骤4:针对每一对概念,在计算了属性包含度g,对象包含度m和整体包含度c后,按照前文所述阈值10%判断依赖簇的类型.

3　针对特殊结构的依赖簇检测方法

3.1　针对结构体、数组、指针以及内存管理函数的依赖簇检测方法

对含有结构体,数组的程序进行依赖关系计算,传统方法需要对程序依赖图(Program Dependence Graph,PDG)进行扩展[11]以支持对数组和结构体访问相关的域敏感依赖性分析.本文通过高阶函数形式的函数摘要保存过程内结构体,数组的域相关依赖信息,并通过参数在过程间进行传递,从而对含有结构体,数组,指针的程序进行过程间的依赖簇检测.

借助LLVM的访问路径[12]分析,可以获取分析过程中与指针、结构体、数组有关的访问路径,从而精确地进行依赖关系的计算.访问路径信息由一个基本对象(base object)和一组访问路径(access path)组成.基本对象可以是指针、结构体、数组等,而访问路径是由一系列操作符组成的路径,例如指针解引用、数组访问,结构体域访问等.访问路径内容如式(16)所示.

$$\begin{aligned}\mathrm{accesspath}(e)=\mathrm{baseobject}(e)&\oplus\mathrm{op}_1(\mathrm{operand}_1)\\&\oplus\mathrm{op}_2(\mathrm{operand}_2)\oplus\cdots\oplus\mathrm{op}_n(\mathrm{operand}_n)\end{aligned} \tag{16}$$

在式(16)中,baseobject(e)表示获取元素e的基本对象;$\oplus$表示连接操作,用于将基本对象和每个操作符连接起来;op_i是第i步的操作符,$\mathrm{operand}_i$是第i步操作符的操作数.

对于依赖簇分析,访问路径可以进行辅助分析,对涉及对含有结构体,指针,数组结构的程序进行依赖关系计算,访问路径分析是递归的,是因为在程序执行的过程中,指令之间存在着一定的依赖关系,某个变量的值可能会在多个指令之间传递和修改,因此需要对整个程序的指令序列进行递归分析,才能完整地描述变量在程序中的路径和最终的值.对于LLVM IR的相关内存操作指令,具体处理如下:

(1)对于赋值或内存修改指令,根据式(17)更新其访问路径为对应赋值或修改的内存地址addr的访问路径accesspath(addr).

$$\frac{i=\mathrm{store}(\mathrm{sv},\mathrm{addr})}{\mathrm{accesspath}(i)=\mathrm{accesspath}(\mathrm{addr})} \tag{17}$$

(2)对于getelementptr指令,它用于在内存中访问一个结构体(Struct)或数组(Array)类型的元素.这个指令可以根据一个基础指针(base)和一系列偏移量($\mathrm{index}_1,\mathrm{index}_2,\cdots,\mathrm{index}_n$)来计算出最终要访问的元素的地址.根据偏移量更新基础指针访问路径.如式(18)所示,其中,accesspath(i)表示最终要访问的元素的访

问路径，size_of(element_type_j)表示第j个元素类型的大小，即在内存中占用的字节数.

$$\frac{i=\text{getelementptr}(\text{base},\text{index}_1,\text{index}_2,\cdots,\text{index}_n)}{\text{accesspath}(i)=\text{accesspath}(\text{base})+\sum_{j=1}^{n}\text{index}_j\cdot\text{size\_of}(\text{element\_type\_j})} \tag{18}$$

(3)对于load指令，需要递归分析load指令中加载的内存地址lv，得到内存地址的访问路径accesspath(lv). 此处采用式(19)更新load指令对应的指令依赖.

$$\frac{i=\text{load}(\text{lv})}{\text{accesspath}(i)=\text{accesspath}(\text{lv})} \tag{19}$$

(4)对于bitcast指令，根据式(20)更新bitcast指令对应的访问路径为其转换值(cv)的访问路径.

$$\frac{i=\text{bitcast}(\text{cv})}{\text{accesspath}(i)=\text{accesspath}(\text{cv})} \tag{20}$$

对于依赖簇检测来说，在没有引入访问路径分析前，结构体、数组在变量依赖表中是当作整体进行计算的，在引入访问路径分析后，对于赋值或内存修改指令，依赖信息就可以根据访问路径更新到相应的内存区域，因此变量依赖表的更新方式如式(21)所示：

$$S(x)=\begin{cases}L(i), & x\in\text{accessPath}(i),x\text{是 single-valued}\\ S(x)\cup L(i), & x\in\text{accessPath}(i),x\text{是 multi-valued}\\ \varnothing, & \text{其他变量}\end{cases} \tag{21}$$

指令依赖的更新方式相应修改为式(22)：

$$L(i)=\bigcup_{x\in\text{accessPath}(\text{REF}(i))}S(x)\cup\bigcup_{i'\in\text{REF}(i)}L(i')\cup\bigcup_{j\in\text{CD}(i)}L(j)\cup L(i) \tag{22}$$

其中，accesspath(REF(i))的作用是根据指令使用对象的访问路径，在变量依赖表S中获取对应的依赖.

由于结构体、数组在变量依赖表S中不再作为整体出现而是深入到内部区域，所以在使用函数摘要的数据依赖进行初始化的步骤需要相应做出调整. 需要根据传入的包含不同内存区域的数据依赖信息进行变量依赖表的初始. 变量依赖表的初始化方法如式(23)所示：

$$S(\text{field}(x))\equiv\begin{cases}\text{FIND}(\text{field}(x),f_n), & x\text{为第}n\text{个形参}\\ \text{FIND}(\text{field}(x),g_m), & x\text{为第}m\text{个全局变量}\\ \varnothing, & \text{其他变量}\end{cases} \tag{23}$$

其中，FIND(field(x),f_n)的含义为：设x为第n个形参，则在对应函数摘要参数f_n中查找x的访问区域field(x)对应的依赖；FIND(field(x),g_n)的含义为：设x为第n个全局变量，则在对应函数摘要参数g_n中查找x的访问区域field(x)对应的依赖. 利用上述查找的结果初始化过程内的变量依赖表.

对于指针，可以借助BasicAA(基本别名分析)进一步确定指针对应的内存位置，从而在变量依赖表中查询对应的依赖值. BasicAA是LLVM框架中的一个指针别名分析器. 它是一种较为粗略的指针别名分析技术，用于判断指针类型的内存操作之间是否可能存在别名关系，以及这些操作是否对同一内存位置进行操作. 它是在LLVM IR级别执行的，因此不需要进行高级别的代码分析，能够快速分析代码.

对于在分析过程中遇到的动态申请数据(如C语言malloc构造出的堆数据)，由于程序中通常采用基础库函数实现相应的功能，故本文方法将其当做一般的库函数调用，不对其进行过程间分析，只考虑其返回值.

3.2 针对异常处理结构的依赖簇检测方法

在依赖簇检测中，为了对含有异常处理结构进行过程间分析，传统方法需要对系统依赖图进行扩展[13]. 本文则通过高阶函数形式的函数摘要保存过程内的相关异常信息，使异常相关的信息像参数一样在过程间进行传递，从而进行过程间依赖簇检测. 在LLVM中，invoke用于调用一个可能会抛出异常的函数. 如果函数执行过程中出现异常，invoke指令将会跳转到一个异常处理器指定的基本块，而不是正常的返回地址. 除了invoke指令，LLVM还提供了一些与异常处理相关的指令和函数，例如landingpad指令用于声明异常处理基本块，cleanuppad指令用于声明清理代码基本块.

为了在过程间调用引发异常的情况下正确地进行依赖簇检测，需要对正常退出和异常退出的情形进行区分. 在正常退出时，函数摘要的返回值形式与式(1)所述一致. 然而对于含有异常抛出情况的函数，需要根据式(24)将异常对象的类型信息及其过程内依赖作为摘要返回值的一部分.

$$\begin{aligned}&\text{normal}=\lambda P_{\text{set}}\cdot(R_1,R_2,\cdots,R_{\text{n}})\\ &\text{except}=\lambda P_{\text{set}}\cdot(R_{\text{a}},R_{\text{b}},\cdots,R_{\text{c}},S_{\_\text{cxa\_throw}}(\text{eobj}),t)\end{aligned} \tag{24}$$

其中，normal和except对应了正常退出和异常退出两种情形的以Lambda表达式形式的摘要信息；R是一系列含有与参数P_{set}有关的依赖集合；eboj和t分别是异常对象及其类型信息.

构建了含有异常抛出情形的函数摘要后，就可以在调用点处实例化normal，更新invoke指令对应的正常退出基本块bnormal的迭代状态. 与此同时，实例化except并和调用点处的依赖关系进行合并，更新invoke指令对应的异常处理基本块bexcept的迭代状态. 这样异常对象及其类型信息就可以被传递到调用过程的异常处理结构下进行后续分析.

4　基于自适应计算的依赖簇检测优化策略

4.1　优化策略基本思想与步骤

在程序中不存在递归调用时,可以按照函数调用关系的逆拓扑排序依次分析各个过程.但是当递归调用存在时,就需要以递归调用的函数形成的SCC为单位[14],计算整体函数摘要.具体地,对每个SCC,内部需要进行迭代数据流分析,达到不动点后,再按照逆拓扑顺序进行下一个SCC的分析.

基于自适应计算的依赖簇检测算法如算法2所示,对每个SCC进行初次分析时,需要对内部每个函数都进行完整的分析,得到各个函数的摘要信息并汇总,作为该SCC的一个总体摘要,以此判断对该SCC整体的分析是否达到不动点(将流入SCC的摘要信息与初次分析的结果进行比较),若未达到不动点,则应用自适应计算中的改变(ChangeMod)和传播(Propagate)过程,得到更新后的摘要信息,直至SCC的摘要信息保持不变,过程间分析达到不动点.

对每个函数的过程内分析,在调用处需要采取计算边(ReadMod)的方式来实例化被调用函数的摘要信息,从而将由函数调用关系产生的摘要数据依赖关系正确反映到自适应计算的自适应计算图上.

算法2　基于自适应计算的依赖簇检测

```
输入:待分析的强连通分量集Set_scc
输出:依赖关系集T
1:FUNCTION  dependencyClusterDetection(Set_scc)
2:  S ← getSCCByITO(Set_scc) //以逆拓扑排序获取SCC
3:    WHILE S非空 DO
4:    FOR p 为S中的函数 DO
5:      NewMod(p)
6:    END FOR
7:    SUM_S ← {p | p为S中的函数}
8:    WHILE SUM_S 未到达不动 DO
9:      FOR p 为S中的函数 DO
10:       ChangeMod(p,InterDependenceAnalysis(p))
11:       Propagate(p)
12:     END FOR
13:     SUM_S ← {p | p为S中的函数}
14:   END WHILE
15:   S ← getSCCByITO(Set_scc)
16:   END WHILE
17:  T ← {t | t为Set_scc中所涉及过程的摘要}
18:  RETURN T
19:END FUNCTION
```

4.2　自适应计算与并行并发策略的结合

为了进一步提高依赖簇检测的效率,本节将自适应计算与一种基于信息流的并行并发策略[15]进行结合.从而在利用自适应计算图计算减少冗余计算的同时,将大规模程序分析任务有效地分而治之,提高分析效率.

基于信息流的并行并发策略有以下相关原语,fork为每个计算创建一个线程,每个fork创建的线程默认是并发的,信息使用交互变量IVar储存,IVar支持put和get操作,分别表示写入和读,get代表进行线程阻塞(等待),put代表将运行结果存在IVar中.

在依赖簇检测过程中,可以通过给每个调用图(call graph)中的SCC分配一个线程,使各SCC内部在各自的线程内进行基于自适应计算图的计算,随后通过IVar进行基于函数摘要的信息传递.分析结果后的摘要信息写入IVar中,在数据流迭代遇到call/invoke指令时进行相应的代入.

默认各个SCC间在各自的线程中进行并行分析,每个SCC在函数调用点使用get读取外部SCC的摘要,同时SCC内部是基于自适应计算的迭代数据流分析后的结果(函数摘要),保存在IVar中,供其他SCC调用.

当某个分析线程中出现需要外部SCC信息的情况时,该线程会阻塞自身直到能够从目标SCC的IVar中通过get获取到摘要信息.由于对IVar的put和get操作均为原子的,故可以保证各个线程之间的同步.

5　原型系统与实验分析

为了验证基于高阶函数和自适应计算的依赖簇检测方法的有效性,本章选取了不同规模不同领域的工程项目和基准测试集,并且给出实验所需评价指标,介绍本章所需实验环境和具体的实验方法.本节共提出2个研究性问题,通过具体的实验结果及其分析,以充分说明本文方法在时间空间上的优势.

5.1　实验数据集

本文实验环境如表1所示,所选取的测试集如表2所示,所选取的测试样例中均含有递归调用的情况,这不仅能够对比高阶函数方法相对于SDG的时间空间优势,并且能够更加有效地反映自适应计算能否对于递归调用情况做出优化.

表1　实验环境

配置名	选项
OS	Ubuntu 20.04 LTS
CPU	Intel(R)Xeon(R)CPUE5-2630v2@2.60 GHz
Memory	128 GB
Language	Haskell(GHC8.6.5)
LLVM	11.0

5.2　实验方法

对于依赖簇检测涉及的开源项目,使用wllvm进行基于llvm的完整编译.

实验使用Haskell自带的运行时系统选项RTS(RunTime System options),可获取对程序进行分析的所

表2 测试用例基本属性和依赖簇信息

Name	Inst	Pass	Proc	g	m	c	Type
openssl/libcryptoshlibasn1_parse	535	6 948	4	0.106 5	0.166 7	0.017 8	LL
openssl/libcryptoshlibtasn_prn	543	2 121	16	0.097 6	0.158 7	0.015 5	SL
ctags/libctags_a-json	632	4 022	6	0.231 0	0.244 0	0.056 4	LL
openssl/libcryptoshlibtasn_enc	685	2 569	8	0.258 8	0.387 5	0.100 3	LL
openssl/libcryptolibbn_mul	706	1911	8	0.147 0	0.394 3	0.057 9	LL
ctags/libctags_a-html	964	5 771	6	0.026 0	0.082 3	0.002 1	SS
ctags/libctags_a-protobuf	1 224	13 405	7	0.146 1	0.141 2	0.020 6	LL
ctags/libctags_a-vhdl	1 412	5 231	10	0.167 8	0.210 0	0.035 3	LL
openssl/opensslbinpkcs12	1 426	2 424	9	0.008 4	0.095 8	0.000 8	SS
gsl/francis	1 465	24 477	7	0.139 9	0.166 2	0.023 3	LL
hello/quotearg	1 483	8 409	33	0.078 8	0.226 0	0.017 8	SL
gnugo/filllib	1 491	5 680	3	0.015 5	0.045 1	0.000 7	SS
openssl/libcryptolibtasn_dec	1 602	6 395	10	0.259 7	0.331 4	0.086 1	LL
coreutils/lbracket	1 746	4 373	11	0.265 0	0.294 9	0.078 2	LL
ctags/libctags_a-ldscript	1 886	5 902	11	0.015 7	0.047 9	0.000 8	SS
findutils/libgnulib_aquotearg	2036	20 649	34	0.072 6	0.206 5	0.015 0	SL
diffutils/analyze	2 287	17 115	2	0.044 0	0.103 5	0.004 6	SL
byacc/lalr	2 359	10 826	3	0.035 7	0.233 4	0.008 3	SL
openssl/libcryptoshlibbn_exp	2 434	13 486	9	0.033 3	0.125 4	0.004 2	SL
ctags/libctags_a-ttcn	2 621	14 451	8	0.217 5	0.216 7	0.047 1	LL
tar/create	3 520	13 540	30	0.050 5	0.080 9	0.004 1	SS
coreutils/echo	3 656	28 454	78	0.065 1	0.137 7	0.009 0	SL
coreutils/pwd	3 810	28 126	81	0.069 5	0.142 9	0.009 9	SL
gsl/hyperg1F1	4 121	8 058	14	0.038 1	0.165 8	0.006 3	SL
coreutils/cat	4 252	31 897	83	0.059 2	0.139 7	0.008 3	SL
gnuit/git	4 461	38 076	27	0.024 6	0.055 4	0.001 4	SS

需时间和内存占用情况. 使用 Hackage 开源的 llvm-analysis 进行 LLVM IR 的解析,利用 Adaptive 库实现自适应计算,FGL(Functional vbb Graph Library)库构建系统依赖图[16],利用 Python concepts 库进行形式概念格构造. 开启 Haskell 内置的多线程选项-N4,使分析可以在并行环境下执行,利用4个物理核心.

5.3 实验结果与分析

为了评估基于高阶函数的依赖簇静态检测方法(dependence Cluster detection Static Algorithm based on Higher-order function,HSCA)与基于 SDG 的依赖簇检测方法(dependence Cluster detection Algorithm based on System dependence Graph,SGCA)之间的差异性以及验证应用了自适应计算的依赖簇检测方法(dependence Cluster detection Static Algorithm based on higher-order function with ADaptive computing,ADSCA)的有效性,本文研究了以下两个问题:

(1)RQ1:HSCA 是否可以获得比 SGCA 更好的效率和空间占用?

(2)RQ2:ADSCA 相比 HSCA 是否减少分析过程中的冗余计算,进一步提高迭代分析效率?

RQ1:HSCA 是否可以获得比 SGCA 更好的效率和空间占用?

为了研究这个问题,本文在所选依赖簇实验数据集上进行了5轮实验,比较了 HSCA,SGCA 的时间空间性能,得到的测试用例相关属性和依赖簇信息见表2. 时间空间性能对比如表3示.

表2中,Inst 表示指令静态数量,能够反映该测试程序的整体规模;Pass 表示实际数据流分析遍历的指令数量;Proc 表示过程数量;g、m、c 表示形式概念方法下得到的整体包含度最大的概念对应的相关属性;Type 该依赖簇类型. 表3中,HSCA-T 和 SGCA-T 是时间指标,分别代表了使用两种不同的算法进行依赖簇检测的时间;HSCA-S 和 SGCA-S 是空间指标,代表了两种方法在检测过程中所使用的内存. 最后两行分别是每个指标的平均值 Average 和相对于 SGCA 在时间和空间占用上减少或增加的百分比 Ratio.

表3中的实验结果表明,与 SGCA 相比,HSCA 在分析效率方面平均可以提高2.689倍,在空间占用方面平

表3　HSCA,SGCA的时间空间性能对比

Name	HSCA-T/s	SGCA-T/s	HSCA-S/MB	SGCA-S/MB
openssl/libcryptoshlibasn1_parse	10.7	13.4	34.2	118.2
openssl/libcryptoshlibtasn_prn	4.5	5.5	6.9	37.3
ctags/libctags_a-json	4.1	4.9	22.3	52.1
openssl/libcryptoshlibtasn_enc	10.9	45.9	13.4	60.2
openssl/libcryptolibbn_mul	7.1	6.1	11.5	26.1
ctags/libctags_a-html	4.9	13.5	40.2	47.0
ctags/libctags_a-protobuf	25.1	58.7	143.3	453.7
ctags/libctags_a-vhdl	15.7	19.5	47.5	110.1
openssl/opensslbinpkcs12	6.6	13.3	39.6	69.4
gsl/francis	15.7	37.9	67.6	78.3
hello/quotearg	29.7	107.1	128.6	285.3
gnugo/filllib	4.9	31.1	142.8	154.1
openssl/libcryptolibtasn_dec	34.9	62.4	48.7	278.9
coreutils/lbracket	15.1	110.3	38.3	124.0
ctags/libctags_a-ldscript	4.6	7.7	44.8	39.1
findutils/libgnulib_aquotearg	96.7	479.5	244.8	256.8
diffutils/analyze	76.7	502.5	138.4	277.8
byacc/lalr	10.0	99.7	187.2	202.0
openssl/libcryptoshlibbn_exp	10.3	57.4	47.8	97.2
ctags/libctags_a-ttcn	54.6	169.0	785.6	895.7
tar/create	114.3	300.8	309.4	998.3
coreutils/echo	164.2	556.7	339.9	423.7
coreutils/pwd	166.2	326.3	343.5	469.0
gsl/hyperg_1F1	28.7	156.7	222.2	253.1
coreutils/cat	197.5	495.3	537.0	1 080.0
gnuit/git	163.0	496.3	2 577.0	2 597.5
coreutils/seq	177.4	560.0	529.1	558.8
gnubg/makebearoff	62.9	192.4	479.4	935.6
ctags/readtags-es	63.4	508.3	172.7	1 551.2
gnugo/montecarlo	194.2	550.9	977.9	1 154.9
pkg-config/libglib_2_0_la-gmain	34.7	559.1	440.9	950.7
bc/number	91.6	462.2	1 221.2	1 513.7
Average	59.4	219.1	324.5	504.7
Ratio	−72.9%	#	−35.7%	#

均可以减少35.7%. 可以发现对于一些小型的程序(如libctags_a-ldscript,openssl/libcryptolibbn_mul),HSCA的时间和空间表现与SGCA相近.

但是随着程序规模的扩大以及过程数量的增加,HSCA相对于SGCA在分析效率方面有了明显的提升,由于HSCA可以直接通过应用函数摘要的参数获取过程间依赖信息,避免添加了额外的参数出/入节点和边,并且无需进行两阶段的图可达性算法,因此时间性能表现的提升较为明显.

空间表现方面,对于libglib_2_0_lagmain等过程数量较大(过程数量>100)的程序,HSCA的空间表现显著优于平均水平. 而对于部分过程数较少的大型项目(如git只有27个过程),HSCA对其检测的性能提升(主要是空间性能)较为有限,这是因为SDG算法中有相当一部分的构图开销用于了过程间边与节点的生成,基于高阶函数的分析能够优化这一部分的开销. 可以认为,SDG的过程间静态分析构图过程复杂的缺陷显著影响了这些过程数量较大的空间性能表现,所以HSCA对于存在大量过程以及调用关系复杂的项目的检测拥有较高的性能优势.

由实验结果可以得出,本文提出的HSCA在分析代码静态规模较大、过程较多以及程序实际遍历的指令

数量较大的程序时具有显著的优势,在分析指令数量、调用过程数量以及实际遍历指令数较小的程序时,HSCA与SGCA的时间空间表现相当.

RQ2:ADSCA相比HSCA是否减少分析过程中的冗余计算,进一步提高迭代分析效率?

为了研究这个问题,本文在RQ1的实验基础上应用自适应计算策略,比较了ADSCA和HSCA时间空间性能,实验结果如表4所示.

表4 ADSCA,SGCA的时间空间性能对比

Name	SCC	Reduce	Full	Max	ADSCA-T/s	HSCA-T/s	ADSCA-S/MB	HSCA-S/MB
openssl/libcryptoshlibasn1_parse	1	5 235	6 948	1 713	10.3	10.7	57.1	34.2
openssl/libcryptoshlibtasn_prn	1	1 543	2 121	578	4.2	4.5	13.1	6.9
ctags/libctags_a-json	2	3 526	4 022	456	4.0	4.1	24.5	22.3
openssl/libcryptoshlibtasn_enc	1	1 783	2 569	786	8.2	10.9	24.9	13.4
openssl/libcryptolibbn_mul	2	1 467	1911	261	5.1	7.1	13.0	11.5
ctags/libctags_a-html	1	4 149	5 771	1 622	4.0	4.9	43.2	40.2
ctags/libctags_a-protobuf	1	8 503	13 405	4 902	18.7	25.1	145.8	143.3
ctags/libctags_a-vhdl	1	4 368	5 231	863	14.8	15.7	104.7	47.5
openssl/opensslbinpkcs12	1	2 364	2 424	60	**6.8**	**6.6**	40.9	39.6
gsl/francis	1	6 888	24 477	17 589	6.3	15.7	63.9	67.6
hello/quotearg	1	2 757	8 409	5 652	22.2	29.7	84.2	128.6
gnugo/filllib	1	3 616	5 680	2064	**5.0**	**4.9**	158.7	142.8
openssl/libcryptolibtasn_dec	2	4 737	6 395	1 439	26.2	34.9	131.8	48.7
coreutils/lbracket	1	3 314	4 373	1 059	13.1	15.1	96.9	38.3
ctags/libctags_a-ldscript	1	4 962	5 902	940	4.4	4.6	63.3	44.8
findutils/libgnulib_aquotearg	1	6 189	20 649	14 460	59.4	96.7	249.3	244.8
diffutils/analyze	1	12 439	17 115	4 676	69.1	76.7	191.3	138.4
byacc/lalr	1	10 672	10 826	154	9.9	10.0	188.5	187.2
openssl/libcryptoshlibbn_exp	1	6 477	13 486	7 009	5.5	10.3	45.0	47.8
ctags/libctags_a-ttcn	1	12 073	14 451	2 378	48.2	54.6	799.4	785.6
tar/create	2	7 219	13 540	4 677	81.9	114.3	542.7	309.4
coreutils/echo	1	9 070	28 454	19 384	104.7	164.2	409.6	339.9
coreutils/pwd	1	8 742	28 126	19 384	106.7	166.2	443.6	343.5
gsl/hyperg_1F1	1	6 552	8 058	1 506	26.0	28.7	221.4	222.2
coreutils/cat	1	12 513	31 897	19 384	132.6	197.5	981.0	537.0
gnuit/git	1	37 660	38 076	416	155.5	163.0	2 574.4	2 577.0
coreutils/seq	1	9 553	28 937	19 384	118.2	177.4	529.1	529.1
gnubg/makebearoff	2	35 815	36 934	1 108	62.4	62.9	930.4	479.4
ctags/readtags-es	6	10 509	11 343	405	57.9	63.4	326.4	172.7
gnugo/montecarlo	2	21 399	23 991	2 523	137.9	194.2	1 153.7	977.9
pkg-config/libglib_2_0_la-gmain	5	13 657	22 103	2 706	**32.3**	**34.7**	598.7	440.9
bc/number	1	19 228	20 995	1 767	83.7	91.6	1 392.7	1 221.2
Average	#	9 343	14 644	5 041	45.2	59.4	395.1	324.5
Ratio	#	−56.7%	#	#	−23.9%	#	+21.8%	#

表4中,ADSCA-T和HSCA-T是时间指标,分别代表了使用两种不同的算法进行依赖簇检测的时间;ADSCA-S和HSCA-S是空间指标,代表了两种方法在检测过程中所使用的内存;SCC代表程序模块中出现递归调用的强连通分量个数.Full代表完整进行过程间分析的遍历的指令数;Reduce代表应用了自适应计算进行优化后,遍历的指令数;Max代表应用了自适应计算进行优化后,单个SCC最大削减的遍历指令数;Average是两种方法对应指标的平均值;Ratio是ADSCA相对于HSCA在时间和空间上减少或增加的百分比,以及Reduce相对于Full约简的指令数量的百分比.

实验结果表明,与HSCA相比,ADSCA可以约简56.7%的冗余指令计算,分析效率方面平均可以提高23.9%,但是空间占用方面增加21.8%.原因在于,构建

自适应计算图,将函数摘要作为自适应计算图的节点保存,以及应用自适应计算中的变化-传播过程,会带来额外的空间和时间占用.

具体地,对于约简指令较大的程序(如coreutils/echo),这些程序时间效率的提升较为显著.然而,对于约简指令量较小的程序(如gnugo/filllib)ADSCA的时间表现与HSCA相近.另一方面,对于pkg-config/libglib_2_0_la-gmain,尽管其Reduce相对于Full有了显著减少,但是其SCC共有5个,Max仅为2 706,SCC数量的增多占用了额外的时间空间进行自适应计算图的构建,并且对于每个SCC来说实际减少的计算量有限,削弱了利用自适应计算提高分析效率的作用.

由以上实验结果可以得出,ADSCA相对于HSCA的性能提升主要受应用自适应计算之后实际指令消减数的影响.除此之外,由于构建自适应计算图需要消耗一定的时间,每个SCC内的减少的冗余计算能否平衡应用自适应计算图所需的额外时间也是影响ADSCA性能的因素.总的来说,ADSCA对于约简指令量较大的程序表现出明显的时间效率提升,而对于约简指令量较小的程序表现与HSCA相近.

6　总结与展望

为了解决传统基于系统依赖图的依赖簇检测方法构图过程复杂,对大型程序进行分析的时间空间损耗大的问题,本文提出一种轻量级且高效的依赖簇检测方法以及相配套的自适应计算依赖簇检测优化策略,能够有效提高分析的效率并减少空间损耗.经相关实验证明,本文提出的基于高阶函数的依赖簇检测方法相比系统依赖图的检测方法能够有效提高分析效率并节约分析所需空间.在未来的工作中,我们将本文提出的方法进行扩展,使其能够支持对多线程程序以及面向对象编程的检测.

参考文献

[1] BINKLEY D. Dependence cluster causes[C]//Scalable Program Analysis. Dagstuhl: Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2008: 1-13.

[2] HARMAN M, BINKLEY D, GALLAGHER K, et al. Dependence clusters in source code[J]. ACM Transactions on Programming Languages and Systems, 2009, 32(1): 1-33.

[3] ISLAM S S, KRINKE J, BINKLEY D, et al. Coherent dependence clusters[C]//Proceedings of the 9th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering. New York: ACM, 2010: 53-60.

[4] 程克. 基于形式概念分析的依赖簇检测方法研究[D]. 北京: 北京化工大学, 2015.
CHENG K. Research on Formal Concept Analysis Based Dependence Cluster Detection[D]. Beijing: Beijing University of Chemical Technology, 2015. (in Chinese)

[5] ACAR U A, BLELLOCH G E, HARPER R. Adaptive functional programming[J]. ACM Transactions on Programming Languages and Systems, 2006, 28(6): 990-1034.

[6] 陈洪龙, 李仁发. 自适应演化软件研究进展[J]. 计算机应用研究, 2010, 27(10): 3612-3616, 3621.
CHEN H L, LI R F. Survey on self-adaptive evolution software[J]. Application Research of Computers, 2010, 27(10): 3612-3616, 3621. (in Chinese)

[7] 连瑞琦, 张兆庆, 乔如良. 一种增量式数据流分析方法[J]. 计算机研究与发展, 2002, 39(2): 136-141.
LIAN R Q, ZHANG Z Q, QIAO R L. An incremental method of data-flow analysis[J]. Journal of Computer Research and Development, 2002, 39(2): 136-141. (in Chinese)

[8] 王留帅. 基于函数摘要的C++过程间静态分析研究[D]. 成都: 电子科技大学, 2017.
WANG L S. Summary-Based Interprocedure Analysis for C++ [D]. Chengdu: University of Electronic Science and Technology of China, 2017. (in Chinese)

[9] KIM S K, VENET A J, THAKUR A V. Deterministic parallel fixpoint computation[J]. Proceedings of the ACM on Programming Languages, 2019, 4(POPL): 1-33.

[10] BINKLEY D, HARMAN M. Locating dependence clusters and dependence pollution[C]//21st IEEE International Conference on Software Maintenance (ICSM'05). Piscataway: IEEE, 2005: 177-186.

[11] GALINDO C, KRINKE J, PÉREZ S, et al. Field-sensitive program slicing[M]//Software Engineering and Formal Methods. Cham: Springer International Publishing, 2022: 74-90.

[12] LERCH J, SPÄTH J, BODDEN E, et al. Access-path abstraction: scaling field-sensitive data-flow analysis with unbounded access paths (t)[C]//2015 30th IEEE/ACM International Conference on Automated Software Engineering. Piscataway: IEEE, 2015: 619-629.

[13] 姜淑娟, 徐宝文, 史亮, 等. 一种基于异常传播分析的依赖性分析方法[J]. 软件学报, 2007, 18(4): 832-841.
JIANG S J, XU B W, SHI L, et al. An approach to analyzing dependence based on exception propagation analysis [J]. Journal of Software, 2007, 18(4): 832-841. (in Chinese)

[14] 张健, 张超, 玄跻峰, 等. 程序分析研究进展[J]. 软件学

报, 2019, 30(1): 80-109.
ZHANG J, ZHANG C, XUAN J F, et al. Recent progress in program analysis[J]. Journal of Software, 2019, 30(1): 80-109. (in Chinese)

[15] MARLOW S, NEWTON R, PEYTON JONES S. A monad for deterministic parallelism[J]. ACM SIGPLAN Notices, 2011, 46(12): 71-82.

[16] 张迎周, 徐晨晨, 竺殊荣. 一种参数化的改进SDG程序切片方法[J]. 南京邮电大学学报(自然科学版), 2017, 37(6): 75-80, 89.
ZHANG Y Z, XU C C, ZHU S R. Parametric method for improving SDG-based program slicing[J]. Journal of Nanjing University of Posts and Telecommunications (Natural Science Edition), 2017, 37(6): 75-80, 89. (in Chinese)

作者简介

杨嘉毅 男, 1997年1月出生于江苏省徐州市. 南京邮电大学硕士毕业生. 主要研究领域为程序分析、函数式编程.
E-mail: 18013977230@163.com

张迎周 男, 1978年1月出生于安徽庐江. 南京邮电大学教授. 主要研究方向为软件与信息安全.
E-mail: zhangyz@njupt.edu.cn

李俊锋 男, 1999年5月出生于新疆维吾尔自治区奎屯市. 南京邮电大学硕士生. 主要研究方向为程序分析.
E-mail: 1617736692@qq.com

马 锐 男, 2000年7月出生于江苏淮安. 南京邮电大学硕士生. 主要研究方向为程序分析.
E-mail: 1607414781@qq.com

汪全盛 男, 1999年12月出生于江苏省南京市. 南京邮电大学硕士生. 主要研究领域为程序分析.
E-mail: 1366257509@qq.com

薛渝川 男, 1999年1月出生于江苏省徐州市睢宁县. 南京邮电大学硕士生. 主要研究方向为程序分析.
E-mail: 1842688656@qq.com