基于包含的指针分析优化技术综述

陈聪明[1),2)] 霍 玮[1)] 于洪涛[1),2)] 冯晓兵[1)]

[1)](中国科学院计算技术研究所计算机系统结构重点实验室 北京 100190)

[2)](中国科学院研究生院 北京 100190)

摘 要 指针分析是程序分析和编译优化的基础,针对基于包含的指针分析算法的改进一直是指针分析领域研究的热点之一.文中从该指针分析算法改进的两类技术来总结近二十年来相关的研究工作,包括在线优化技术如约束图上的强连通分量的检测和消除等和离线优化技术如变量替换等.通过实验对比了 7 种较有影响力的分析算法和三种离线优化算法,并从性能、内存开销等方面进行了评述和总结.文章最后阐述了基于包含的指针分析今后潜在的研究方向.

关键词 指针分析;Andersen 风格;指向集;约束图;流不敏感

中图法分类号 TP311 **DOI 号**: 10.3724/SP.J.1016.2011.01224

A Survey of Optimization Technology of Inclusion-Based Pointer Analysis

CHEN Cong-Ming[1),2)] HUO Wei[1)] YU Hong-Tao[1),2)] FENG Xiao-Bing[1)]

[1)](*Key Laboratory of Computer System and Architecture*, *Institute of Computing Technology*, *Chinese Academy of Sciences*, *Beijing* 100190)

[2)](*Graduate University of Chinese Academy of Sciences*, *Beijing* 100049)

Abstract Pointer analysis is the basis of program analysis and compiler optimization. Improvement of inclusion-based pointer analysis is one of the most important research areas in pointer analysis. This paper summarizes two classes of improvement methods of inclusion-based analysis in recently twenty years, including: on-line optimizing technology such as online cycle detection and elimination etc, off-line optimization such as variable substitution. The authors compare seven principal analysis algorithms and three offline optimizing algorithms by significant experiments; evaluated each algorithm from performance and memory consumption. The authors also discuss some further improvements and research points of inclusion-based pointer analysis in the end of this paper.

Keywords pointer analysis; Andersen-style; points-to set; constraint graph; flow-insensitive

1 引 言

指针是众多编程语言中广泛使用的一种特殊的数据类型.一个指针变量用于保存一个程序对象的内存地址,由此能间接地操作这个程序对象.在许多编程语言尤其是 C 和 C++中,指针的使用非常灵活,例如指针既可用于表示动态分配的存储对象又可作为函数参数传递数组或者结构体对象等等.与指针相关的基本分析包括指针分析和别名分析,其

收稿日期:2010-07-02;最终修改稿收到日期:2011-03-13.本课题得到国家"八六三"高技术研究发展计划项目基金(2008AA01Z115)、国家"九七三"重点基础研究规划项目基金(2011CB302504)、国家核高基重大专项基金(2009ZX01036-001-002)和国家自然科学基金创新研究群体科学基金(60921002)资助.**陈聪明**,男,1985 年生,博士研究生,主要研究方向为程序分析以及编译优化技术等.E-mail: chencongming@ict.ac.cn. **霍 玮**,男,1981 年生,博士,主要研究方向为程序分析以及编译优化技术等. **于洪涛**,男,1981 年生,博士研究生,主要研究方向为程序分析以及编译优化技术等. **冯晓兵**,男,1969 年生,研究员,博士生导师,主要研究领域为编译优化技术以及二进制翻译技术等.

中指针分析是指分析一个指针所有可能指向的内存位置，包括程序中的全局变量、局部变量、动态数据对象等等.

指针分析是一类特殊的数据流问题[1]，它是其它静态程序分析的基础，但指针使用的灵活性导致了指针分析的复杂性，实际上精确的指针分析是一个不可判定问题[2-3]，所以实际的指针分析算法都是近似且保守的，须在效率和精度之间进行折衷. 在过去的近三十年间，指针分析一直是程序分析领域的研究重点之一，至今仍很活跃. 指针分析研究的内容主要集中在分析精度和时空开销之间的取舍. 精度方面，主要指流敏感性(flow-sensitivity)和上下文敏感性(context-sensitivity)，一般而言，流敏感分析方法的精度明显好于流不敏感的分析方法，在上下文敏感性上也有同样的特点. 精度不同，对应的指针分析算法的差别也较大，指针分析领域的大多数研究工作都是在保证精度的前提下研究如何提升分析算法的效率.

流不敏感的指针分析普遍使用在开源或者产品级高级编译器中，其中主要有两类：基于包含(inclusion-based)[4]的指针分析和基于合并(unification-based)[5]的指针分析. 基于包含的指针分析是一种基于约束集(constraint set)求解[6-8]的流不敏感的指针分析方法，由 Andersen[4] 于 1994 年在他的博士论文中首次提出. 该指针分析又称为基于子集(subset-based)的指针分析或者基于约束的(constraint-based)指针分析，在指针分析领域后来也被称之为 Andersen 风格(Andersen-Style)的指针分析，其算法的时间复杂度为 $O(n3)$，其中 n 指程序中参与分析的变量数；而基于合并的指针分析是由 Steensgaard[5] 在 1996 年提出的一种指针分析方法，又称为基于等价(equivalence-based)的指针分析，或者也称之为 Steensgaard 风格(Steensgaard-Style)的指针分析，其复杂度接近于线性复杂度. 在近三十年来的指针分析研究中，很大部分的研究工作都是集中在对上述两种指针分析方法的改进. 相对于基于合并的指针分析，对基于包含的指针分析的研究更活跃同时也更具影响力：一方面是因为基于包含的指针分析方法精度更高而更能满足其它程序分析的需要；另一方面基于包含的指针分析由于其分析算法本身的特点使得其效率提升的空间更大. 目前开源编译器(如 GCC 和 LLVM)均使用的是融合了优化改进技术的基于包含的指针分析. 所以，我们相信对基于包含的指针分析的优化技术进行详细的总结和分析具有十分重要的意义，这也是本综述的主要出发点.

在指针分析领域，有多篇综述文章对指针分析领域的相关研究工作做了较为全面的分析和总结，如文献[9-11]等等. 但是这些综述都是从宏观角度对各种指针分析方法及相关工作作横向的介绍，并没有就某一类具体的指针分析算法做深入探讨，缺乏对分析算法具体的改进措施的介绍，亦没有详实的可供参考的实验数据. 与其它综述不同的是，本综述关注的是基于包含的指针分析及相关研究工作，主要涵盖了近二十年来对基于包含的指针分析算法进行优化的相关工作，并通过详细的实验数据对各种优化方法进行对比和分析.

本文第 2 节介绍基于包含的指针分析基本概念和算法；第 3 节总结在线优化技术相关的算法，并通过实验进行对比和分析；第 4 节总结离线优化相关的技术，并通过实验进行对比和分析；第 5 节简要介绍指针分析通用改进技术，包括精度提升和指向集表示；最后一节总结全文同时阐述指针分析领域今后潜在的研究方向.

2　基于包含的指针分析算法

基于包含的指针分析将指针值看作是一种约束关系，这种约束关系直观上来说是指一种集合包含关系. 基于包含的指针分析算法建立在集合约束分析之上，它将指针分析分为两个阶段：约束生成(constraint generation)和约束求解(constraint resolution). 约束生成使用一种约束规范语言来表示实际的程序；约束求解根据生成的约束使用迭代方法求解以得到问题的最小解. 关于约束分析的详细介绍和应用可以参考文献[4].

2.1　约束生成

在约束生成过程中，基于包含的指针分析将每一条赋值语句看作一个约束(许多高级语言都支持强制类型转换，在指针分析算法中不考虑变量的类型信息，因此此处以及下文对指针变量和非指针变量不加区分)，并按照约定规则为每一条语句生成对应的约束，将整个程序转换成一个约束集合. 基于包含的指针分析的约束系统可以抽象为表 1 第 2 列所示的 4 条规则：其中[addr]称为基本约束，[trans]称为简单约束，[deref1]和[deref2]都是复杂约束. τ_1，τ_2，τ_3 是约束变量，约束生成过程中为每个程序变量 v 赋予一个约束变量 τ，约束变量 τ 表示 v 的抽象

存储位置，τ 的值表示 v 所指向的存储位置集合. 例如对于指针变量赋值语句 $p=q$，其对应的约束为 $\tau_1 \supseteq \tau_2$，表示 p 所指向的存储位置集合(简称为指向集)包含 q 的指向集. 符号“{}”代表取地址操作，“*”代表指针解引用操作. 表 1 中最后两列显示了由示例语句生成的初始约束以及基于上述 4 条约束规则进行约束求解之后的结果. 此处为简单起见，用同一个字母表示程序变量与对应的约束变量，如语句 $p=\&a$ 中的程序变量 p 与对应约束($p \supseteq \{a\}$)中约束变量 p. 此处需要指出的是，语句的顺序并不影响约束的生成以及约束求解结果，这也是流敏感的分析与流不敏感的分析之间的差异.

表 1 约束规则以及约束求解

示例语句	约束规则		约束生成	约束求解结果
$p=\&a;$	$\dfrac{\tau_1=\&\tau_2}{\tau_1 \supseteq \{\tau_2\}}$	[addr]	$p \supseteq \{a\}$	
$p=q;\ q=\&a;$	$\dfrac{\tau_1 \supseteq \{\tau_2\}\quad \tau_3 \supseteq \tau_1}{\tau_3 \supseteq \{\tau_2\}}$	[trans]	$q \supseteq \{a\}\quad p \supseteq q$	$p \supseteq \{a\}$
$q=\&r;\ p=*q;$	$\dfrac{\tau_2 \supseteq \{\tau_3\}\quad \tau_1 \supseteq *\tau_2}{\tau_1 \supseteq \{\tau_3\}}$	[deref1]	$q \supseteq \{r\}\quad p \supseteq *q$	$p \supseteq r$
$p=\&r;\ *p=q;$	$\dfrac{\tau_1 \supseteq \{\tau_3\}\quad *\tau_1 \supseteq \tau_2}{\tau_3 \supseteq \tau_2}$	[deref2]	$p \supseteq \{r\}\quad *p \supseteq q$	$r \supseteq q$

基本约束和简单约束是构建初始约束图(constraint graph)的基础. 约束图是一种有向图，如图 1 所示，图中的节点表示程序变量，有向边表示简单约束. 子图(a)为示例程序对应的初始的约束图，初始约束图的建立分为三步：首先为程序中的每个变量建立一个节点，然后根据基本约束标注节点的指向集，最后为每一个初始的简单约束建立一条有向边. 所有复杂约束是进行约束求解的驱动，因为复杂约束会导致新的简单约束产生. 子图(b)是示例程序约束求解后的结果，其中虚线边为约束求解过程中在约束图中新添加的有向边，各节点指向集元素的改变在图中用粗体标出. 具体的约束求解算法在下文给出.

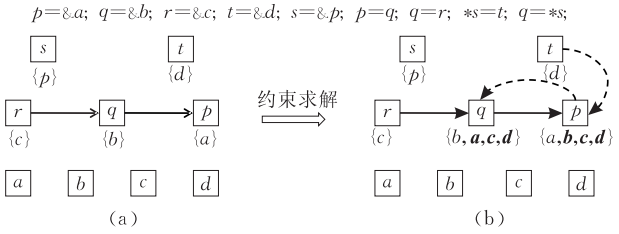


图 1 约束图以及约束求解结果

2.2 约束求解

约束求解基本算法如图 2 所示，其中输入为初始的约束图，输出为求解之后的约束图，符号 $pts(x)$ 表示变量 x 的指向集. 算法使用基于工作集(work-list)的迭代求解方法，算法 6～19 行表示一次迭代过程中的所有操作，该过程主要分为两步：处理两类复杂约束(7～15 行)和传递指向集(16～19 行). 处理复杂约束的过程中可能会向约束图中添加新边(10 行和 14 行)，这会导致新的指向集传递过程(16 行)；而指向集的更新(17 行)会导致工作集的更新(19 行)从而进入下一次的迭代过程. 当工作集为空时，算法终止. 约束图中的 n 个节点之间最多有 n^2 条有向边，因此不论是处理复杂约束还是传递指向集，这两步操作中理论上最多能有 n^2 个节点添加到工作集中，且工作集中节点数是有限的，因此算法必然能在有限次迭代之后终止.

```
1.  Input: directed graph G=⟨V,E⟩
2.  Output: graph G after constraint solving
3.  Let W be an empty worklist;
4.  W←V
5.  while W≠∅ do
6.    choose n from W
7.    for each v∈pts(n) do
8.      for each constraint a⊇*n do
9.        if v→a∉E then
10.         E←E∪{v→a}
11.         W←W∪{v}
12.     for each constraint *n⊇b do
13.       if b→v∉E then
14.         E←E∪{b→v}
15.         W←W∪{b}
16.   for each n→z∈E do
17.     pts(z)←pts(z)∪pts(n)
18.     if pts(z) changed then
19.       W←W∪{z}
```

图 2 基本的约束求解算法[19]

提升基于包含的指针分析算法效率的改进措施本质上并没有减小指针分析算法的复杂度，而是通过显著地减少 n 来获得性能的提升. 这些措施主要可以分为两类：一类是在线优化，主要是约束图上强连通分量(strongly connected components)的检测和消除，所谓在线是指优化在约束求解过程中进行.

约束求解以复杂约束为驱动进行迭代，其间不断有新边加入到约束图中，通过检测和合并不断更新的约束图上的强连通分量可以显著地减少约束图上冗余的指向集传递，同时有效地降低约束求解过程中的迭代次数；另外一类称为离线优化（Offline Optimization），是指在约束生成之后约束求解之前进行预处理，通过变量替换和合并强连通分量等策略减小约束图的规模从而减少约束求解的时空开销. 以上两种技术是对基于包含的指针分析算法的最为重要的改进. 此外还有一些研究工作集中在提升分析的精度上，如将初始的基于域的（Field-Based）的基于包含的指针分析算法[4]改进成域敏感（Field-Sensitive）的算法；或者做上下文敏感的改进，提升原始算法过程间分析的精度；另外还有采用更为有效的数据结构表示指向集，以此提升分析的效率等等. 本文主要从在线优化和离线优化两方面对基于包含的指针分析的相关改进技术做一个总结和评述.

3　在线优化

在约束求解过程中采用的在线优化技术概括起来可以分为：约束图上强连通分量的检测和消除、指向集传递过程中的优化、工作集节点的迭代求解顺序.

3.1　约束图上强连通分量的检测和消除

约束图上强连通分量的检测和消除（Online Cycle Detection and Elimination，为行文方便，以下简称 OCD）是显著提高基于包含的指针分析算法效率的一项重要改进技术. 在约束求解过程中不断有新边加入到约束图中，可能在约束图上产生新的强连通分量，该优化通过合并节点来消除强连通分量.

OCD 之所以能够提高约束求解的效率，其本质原因在于基于包含的指针分析的约束图中的每一条边实质上代表的是一个偏序关系，例如图 3 所示.

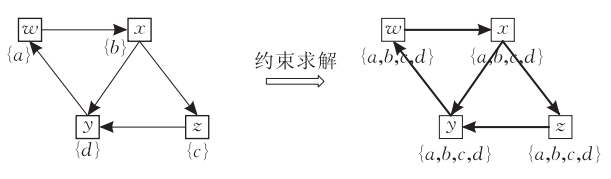


图 3　强连通分量中的指向集传播

图中各节点的初始指向集在其下方标出，如 w 的初始指向集为$\{a\}$. 约束求解时，指向集基于有向边进行传递，虽然各节点的初始指向集不同，但由于节点 w、x、y、z 构成了一个强连通分量，在约束求解之后各个节点的指向集都是$\{a,b,c,d\}$. 换言之，同在一个强连通分量中的各个节点在约束求解之后的指向集必定是相同的①. 将同在一个强连通分量中的节点合并，既可以减少指向集在其内部冗余的传递，以提升迭代求解的效率，同时还可以有效地降低存储的开销，因为合并之后的节点只需维持一个指向集. 实际上，在图算法领域已有一些经典的算法能够高效地检测有向图上的强连通分量，如 Tarjan 的算法[12]和 Nuutila[13]的算法，其中后者是对前者的改进，指针分析中很早就使用了这些算法，所以 OCD 的难点并不在于如何检测强连通分量，而是如何选择检测的时机以及如何控制整个算法的开销等.

根据对强连通分量检测的规模和频度，OCD 的改进算法大致可以分为两类：

一类是在有新边加入到约束图中的时候执行 OCD，这种方式检测的频度较高，且检测的规模一般只限于由该边两端顶点出发的所有可达路径. 例如：Fähndrich 等人[14]首次提出在约束图上做 OCD 可以显著提升约束求解的效率，他们基于路径可达性来查找约束图上的强连通分量. 具体而言，约束图中每加入一条新边 $x \rightarrow y$，算法从节点 x 开始反向执行一遍深度优先（Depth-First）遍历，实际上是为了判断是否存在始于节点 y、终于节点 x 的路径，如果存在这样一条路径 p，则 p 和有向边 $x \rightarrow y$ 将构成一个强连通分量. 但是该方法的问题在于，从节点 x 开始的路径条数如果是指数级的，则遍历的开销将会不可控，因此算法在此基础上增加了一个遍历的终止条件. 终止条件虽然控制了算法的整体开销，但是弊端是导致算法无法检测出当前约束图上所有的强连通分量. Fähndrich 等人工作的重要意义在于其算法显示了 OCD 能够有效地提高基于包含的指针分析的效率. 在此之后，他们又做了进一步的提高分析效率的尝试，采用映射合并（Projection Merging）[15]的方式减少约束图上的冗余，与 OCD 算法结合在当时可以分析 50 万行左右的程序.

继 Fähndrich 等人之后，Heintze 和 Tardieu 提出了一种动态迁移闭包的算法[16]，以下简称为 HT 算法. HT 算法基于子集图进行约束求解，子集图是一种类似于约束图的有向图，是对约束图的扩展，图上同时包含指针变量和指针解引用变量，且子集图上的有向边是反偏序方向的. HT 算法并不在迭代

① 限于篇幅，具体的证明未在本文中列出，可参考文献[14].

过程中传递指向集,在处理复杂约束需要用到当前节点指向集时,通过子集图上的可达性来获取,同时对可达路径进行标记,如果有路径构成强连通分量,则进行合并. HT 算法非常高效,并且颇具影响力,之后 Whaley 等人[17]将此算法推广到分析 Java 程序. HT 算法与 Andersen[4]所提的初始算法一样,都是基于域(field-based)的.

此后,Pearce 等人[18]提出了一种检测强连通分量的新算法,以下简称 PKH1 算法. 该算法的特点是其并不在当约束图上每加入一条新边之后就检测强连通分量,而是动态地维持一个约束图节点的拓扑序. 如果新边加入后改变了原有的拓扑序,则说明约束图上可能存在强连通分量,此时才进行检测,并且重新生成新的拓扑序. 整体而言,相对于 Fähndrich 的算法,Pearce 等人所提出的 OCD 算法对于规模较大的程序,分析效率的提升效果明显,但是整体而言相对于 HT 却不够高效.

Harderkopf 等人[19]提出了惰性强连通分量检测(Lazy Cycle Detection,LCD)算法. LCD 算法的基本思想是基于这样一个假设:如果一条有向边上的两个结点指向集相同,则这条边可能在一个强连通分量上. 因此算法在传递指向集的过程中,首先会判断当前所处理的有向边两端节点的指向集是否相同,基于判断的结果进行强连通分量的检测. 但是这个假设并不是在所有情况下都成立,最理想的情况是能成功检测到强连通分量,否则就将已经检测过的有向边添加到一个集合中,以避免之后迭代过程中的重复计算,但是如此一来,算法就无法保证能够检测出约束图上所有的强连通分量.

Pereira 等人[20]以 Hardekopf 等人的工作为基础,提出了横向传播和纵向传播算法(Wave Propagation, Deep Propagation,以下简称 WP 算法和 DP 算法). 其中的 DP 算法开始先采用 Nuutila 的强连通分量检测算法[13]在整个约束图上做 OCD,之后对所有节点执行指向集传递操作,最后处理复杂约束,在添加完新边之后,算法以指定的起始节点和终止节点做深度优先遍历,同时在遍历的过程中传递指向集. 例如对于复杂约束 $p=*q$,以 p 为起始节点和终止节点做深度优先遍历,如果最后能够到达 p,则说明新边添加后出现了新的强连通分量,可以将其合并;否则仅仅传递指向集. 同理,对于复杂约束 $*p=q$,则以 p 中更新的指向集元素为起点,q 为终点遍历. 算法重复上述步骤直到约束图不再改变.

另外一类算法则是在整个约束图上执行 OCD,一般的频度是每次迭代检测一次,能检测到当前约束图上所有强连通分量. 代表性的工作包括:

Pearce 等人提出的另一个改进算法[21],以下简称为 PKH2 算法. 算法嵌套两层循环,外层循环每次迭代开始之前在整个约束图上首先执行一遍强连通分量检测,在此过程中可以附带得到约束图上节点的拓扑序;之后内层循环按照拓扑序逐个处理约束图上的节点. 总体上算法对强连通分量检测的频度不够,一定程度上制约了分析效率,因此还有提升空间. 除此之外,Pearce 等人还从图算法角度进一步研究了 OCD 的相关改进技术[22-23].

Harderkopf 等人[19]还提出了混合式的强连通分量检测(Hybrid Cycle Detection, HCD)算法. HCD 算法的思想是在约束求解之前,基于离线约束图先做一遍预处理. 所谓的离线约束图和 HT 算法所提的子集图类似,通过在离线约束图上预先找出可能存在的强连通分量为后续的约束求解提供辅助信息,即对形如 $*p=q$(或者 $q=*p$)这样的复杂约束,在离线约束图中,节点 $*p$ 和节点 q 构成一个环,此环的存在意味着 p 的指向集中的所有元素对应的节点将会和节点 q 构成强连通分量,因此在约束求解时就可将 p 的指向集中的元素所代表的节点和节点 q 直接合并,这样省去了遍历约束图查找强连通分量的开销. HCD 算法的创新之处本质上可以归为一种离线的优化,不过因为离线收集的信息主要是用于约束求解过程中的 OCD,因此本文仍然将其归为一种 OCD 算法,HCD 算法也可以与其它算法相结合使用.

Pereira 等人[20]提出的 WP 算法实际上是在 PKH2 算法[21]的基础上对其进行了改进. 算法的主要思想是将处理复杂约束添加新边的过程与传播指向集的过程分离. 算法的主要步骤与 DP 算法类似,只不过在第 3 步处理完所有复杂约束之后不再进行深度优先遍历,所以,WP 算法与 DP 算法效率之间的不同之处在于,DP 算法能够以更小的代价在新边添加之后就能检测出一些潜在的强连通分量.

3.2 指向集传递过程中的优化

在约束求解每次迭代的指向集传递过程中,有时并不需要将当前节点完整的指向集传递给其后继结点,而只需传递本次迭代相较于前次迭代新增加的指向集元素即可,这就是差异传播(Difference

Propagation)的思想. Pearce 等人在 PKH1 算法[18]中提出了差异传播技术. 所谓差异,可以量化为约束求解过程中一个变量在两次迭代之间指向集元素的变化,如果本次迭代和前次迭代中,当前变量的指向集元素没有更新,则无需向其后继节点传递指向集. 差异传播可以一定程度上减少因约束图上冗余的指向集传递所带来的时空开销,有助于提升分析算法的效率. 首先将此技术应用到指针分析领域的是 Lhoták 和 Hendren[24],此外,在 PKH2 算法以及 WP 算法和 DP 算法中也均有使用.

3.3　工作集节点的选取顺序

在基于包含的指针分析算法的迭代过程中,不难发现工作集节点的选取顺序会影响分析的开销. 以图 1 子图(a)中的初始约束图为例,如果在求解之初从工作集中选取的节点为 q,经有向边 $q\rightarrow p$ 将 q 的指向集$\{b\}$传到节点 p,而在某一个时刻,当迭代进行到节点 r 时,指向集经由有向边 $r\rightarrow q$ 传递导致节点 q 的指向集被更新,因此算法又将处理一遍节点 q. 如果采用拓扑序先后选择节点 r、q、p,则可以避免上述冗余计算的部分. 实际上,已有许多研究工作是关于如何在有向图上按照拓扑序处理节点[25-27],但这些都只是针对于静态的有向图. 结合基于包含的指针分析的特点,由于复杂约束的存在,导致在约束求解的过程中不断有新边加入,换言之,约束图上的拓扑序是动态变化的,如果每次迭代都重新生成拓扑序,则会影响算法的执行效率. 一种比较简单有效的选择方式是最近最少使用策略(Least Recently Fired,LRF),也就是最近最少访问到的节点具有最高的优先级,因此最先得到处理. Pearce[18]、Hardekopf[19]、Pereira[20] 等人的分析算法中都采用了这种策略.

3.4　实　验

上文介绍了比较有代表性的改进算法,为了对以上算法有一个较为全面的评价,本文对上述的 HT[16]、PKH1[18]、PKH2[21]、LCD[19]、LCD+HCD[19]、WP[20]、DP[20] 7 个分析算法做了较为全面的对比实验,所有算法在精度上没有差别,都是上下文不敏感、域不敏感的(出于实验公平性上的考虑,将 PKH2 修改成域不敏感的算法).

所用的测试用例为表 2 所示的 6 个测试程序,Emacs 是 Linux 系统中常用的文本编辑器,Ghostscript 是一个 PS 文本浏览工具,Gimp 是 Linux 系统中的一个绘图软件,Insight 是基于 GDB 调试器的一个图形化用户界面程序,Wine 是 Windows 操作系统的模拟器,而 Linux 是 Linux 操作系统内核. 表中第 3 列为程序的规模,其中 Linux 和 Wine 是两个超过百万行的程序. 第 4 列是各测试程序初始的约束规模,包括上文提到的所有 4 类约束. 实验过程中约束生成部分的数据获取采用了与 LCD[19]、DP[20] 等算法类似的流程,这些约束通过 CIL[28] 前端生成,同时出于指向集存储效率上的考虑,将每一个约束变量映射成一个整形数值,将变量之间的约束转换成整型值之间的约束关系,将生成的约束写入约束文件中,之后各个分析算法读取约束文件进行约束求解.

表 2　实验所用测试用例

测试程序	简称	规模/万行	初始约束规模	离线优化后约束规模
Emacs-21.4a	emacs	16.9	83213	27122
Ghostscript-8.15	gs	24.2	169312	80071
Gimp-2.2.8	gimp	55.4	411783	125203
Insight-6.5	inst	60.3	243404	99245
Wine-0.9.21	wine	133.8	713065	199465
Linux-2.4.26	lnx	217.2	574788	231290

实验所用硬件平台为:Intel 四核 CPU E5430 (2.66GHz) x 2,16GB DDR3 内存,操作系统为 Redhat 企业版 5.1. 测试之前,用 gcc 编译各算法源程序,编译优化选项为 O3,同时由于实验机器是 64 位的,编译时设置选项"-m32"用来生成 32 位模式下的可执行码,以此与各算法文献中的实验方法保持一致,此外,实际测试的程序都是单线程的. 所有测试所得数据都是取 3 次运行结果的算术平均值.

由于初始生成的约束规模较大,而各算法的效率不一,对于较大的测试程序,某些算法可能无法完成分析. 因此实验过程中采用了离线变量替换[29]算法对初始约束进行了离线优化(优化后的约束规模如表 2 第 5 列所示),以加快分析算法的效率,提高测试结果的显示度.

表 3 显示了上述 7 种分析算法运行时间的对比数据,图 4 是以 HT 算法为基准,各算法相对于 HT 算法时间开销的对比图示,图中虚线表示 HT 算法归一化后的数据,GeoM 表示几何平均值.

表 4 显示了上述 7 种分析算法的内存开销对比数据,图 5 是以 HT 算法为基准,各算法相对于 HT 算法内存开销的对比图示,图中虚线表示 HT 算法归一化后的数据.

表 3 7 种分析算法的时间开销对比

测试程序	时间开销/s						
	HT	PKH1	PKH2	LCD	LCD+HCD	WP	DP
emacs	0.92	31.85	1.57	2.04	0.68	0.80	0.68
gimp	52.64	651.09	92.18	30.15	12.92	34.93	65.98
gs	9.20	167.92	13.82	9.60	6.35	7.88	18.66
inst	35.23	977.70	102.79	51.10	15.10	94.42	48.49
lnx	342.13	8998.36	1048.23	218.91	83.10	697.73	414.03
wine	1408.66	12650.30	1848.37	822.33	450.65	707.55	1674.91

表 4 7 种分析算法的内存开销对比

测试程序	内存开销/MB						
	HT	PKH1	PKH2	LCD	LCD+HCD	WP	DP
emacs	21.84	20.37	22.91	20.63	20.41	26.45	17.81
gimp	353.66	409.65	433.97	425.14	422.88	549.73	356.69
gs	96.41	94.06	105.35	94.98	95.30	146.11	96.43
inst	261.21	234.63	246.51	235.54	234.86	377.93	233.56
lnx	998.04	981.83	1012.21	1002.60	985.66	1473.82	1011.04
wine	2093.43	1675.02	1776.78	1750.16	1735.54	2420.19	1561.18

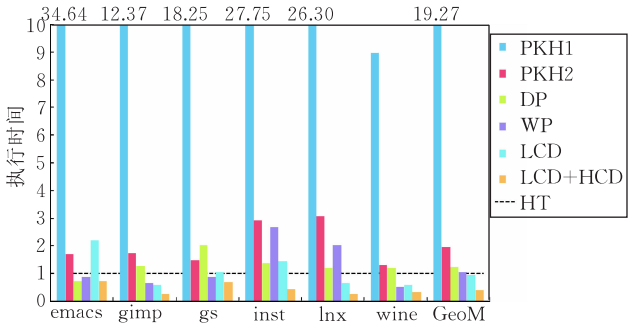


图 4 算法执行时间对比(基于 HT 算法的归一化)

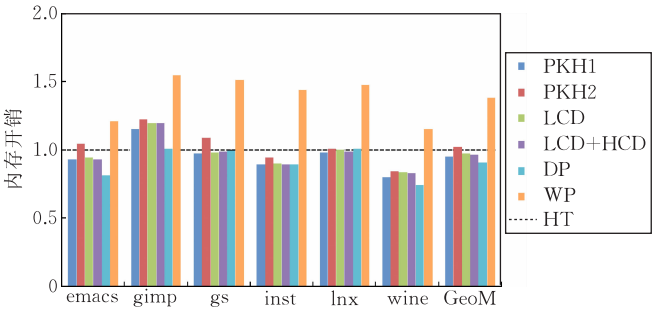


图 5 算法内存开销对比(基于 HT 算法的归一化)

以上数据统计的是约束求解的时空开销,不包括文件读取过程中的开销.

整体上,PKH1 的性能明显差于其它算法,其原因在于算法每加入新边即检测强连通分量并且重新生成拓扑序的开销较大,且算法无法保证检测出约束图上的所有强连通分量. HT 算法虽然提出较早,但是却比较高效,原因主要在于算法勿需传递指向集,在计算可达性获取指向集的同时能够隐式地检测到子集图上的绝大部分强连通分量. LCD 和 LCD+HCD 算法的效率要好于 HT 算法,其中 LCD+HCD 算法尤为明显,这也正好显示了 HCD 离线优化的作用. WP 算法作为对 PKH2 算法的改进,虽然对于强连通分量的检测频度没有增加,但是调整了算法的整体结构,性能相对于后者仍有不小提升.

在内存开销上,除 WP 之外,其它算法之间几无差别,原因在于 WP 算法在分析过程中约束图中的每一个节点保留两个指向集:本次迭代中的指向集和前次迭代后的指向集,用以计算两次迭代之间指向集的差异,一定程度上增加了内存的开销.

4 离线优化

所谓离线优化,是指在约束生成之后约束求解之前对约束做精简. 优化手段都是以变量替换的方式将多个具有相同某种属性的变量用一个变量表示,从而减少约束变量的规模以及与之相关的约束的数量,达到显著减少后续约束求解时空开销的目的. 约束求解之后,要得到替换之前程序的指针分析信息,只需经过一个反替换的过程即可,替换和反替换的过程通过维持一个简单的映射关系即可实现. 可进行替换的变量的属性主要有两种:指针等价(pointer equivalent)和位置等价(location equivalence).

4.1 指针等价

对于任意变量 x、y,经过约束求解后,如果变量 x 的指向集中的元素和变量 y 的指向集中的元素相同,则称 x 和 y 是指针等价的. 由所有指针等价的变量构成的变量集合称为指针等价集(pointer equivalence sets). 指针等价意味着确定了指针等价集中一个变量的指向集,也就能确定该集合中其它变量的指向集,因此可用一个变量来替换指针等价集中的所有变量. 替换之后约束变量的规模显著减

少，而且也可以减少指向集在指针等价集中变量之间的冗余传播，这是因为指针等价集中的部分变量也可能在一个强连通分量上. 上文提到，在一个强连通分量上的变量的指向集相同，因此指针等价变量的查找算法与 OCD 类似，都须做强连通分量的检测. 此外，理论上的证明[29]确保了这种替换不会影响约束求解结果的正确性.

Rountev 等人[29]提出了离线变量替换(Offline Variable Substitution，OVS)算法，该算法复杂度为线性 $O(n)$，其中 n 表示源程序中的变量数. 算法基于由所有约束构建的子集图(subset graph)，子集图是根据基本约束、简单约束和所有复杂约束构建的有向图，子集图中的节点包括被取地址的变量以及指针解引用的变量，这一点与约束图不同，也不同于上文提到的 HT 算法中的子集图，但是节点之间的偏序关系与前两者是一致的.

OVS 算法首先合并子集图中的强连通分量，得到有向无环图. 之后在新图上按照拓扑序逐个处理节点，给每个节点附上一个整数值标号，在此过程中算法区分直接节点和非直接节点，所谓直接节点，是指其指向集结果完全取决于它在子集图中的所有前驱节点. 如果是非直接节点则为其赋上新的标号，如果是直接节点，则分情况讨论：如果它没有前驱，则附上标号 0；如果有前驱且所有前驱节点的标号相同，则给当前节点也赋上同样的标号；如果有前驱但是前驱节点的标号不一致，则给当前节点赋上新的标号. 最终标号相同的变量隶属于同一个指针等价集. 此外，OVS 算法还能附带发现空指针(Non-Pointer)，即指向为空的变量，所有标号为 0 的变量即属于这一类. 与空指针相关的约束可以预先删除，因为其指向集必定为空，所以约束求解过程中这些约束必然不会发生指向集的传递. OVS 算法非常高效，线性时间的复杂度下，平均能够将约束规模减少 64%.

Hardekopf 等人[30]在 OVS 的基础上做了进一步的改进，提出了基于 Hash 的值标号(Hash-based Value Numbering，下文简称为 HVN)方法. 算法根据约束建立所谓的离线约束图(offline constraint graph)，离线约束图与 OVS 子集图类似，唯一的不同之处在于，对于形如 $p \supseteq q$ 的简单约束，子集图中包含节点以及有向边$(q \rightarrow p)$、$(*q \rightarrow *p)$，而离线约束图中仅仅包含节点以及有向边$(q \rightarrow p)$，因此形式上，离线约束图规模会更小. HVN 算法与 OVS 算法最大不同之处在于：假设离线约束图上的两个结点分别为 p 和 q，设节点 p 的所有前驱节点的标号值集合为 S_p，节点 q 的所有前驱节点的标号值集合为 S_q，如果 $S_p = S_q$，则节点 p 和节点 q 有相同的标号. 但是在 OVS 的算法中，节点 p 和节点 q 被赋上了不同的标号，因此 HVN 相对于 OVS 能找到更多指针等价的变量. 同时，Hardekopf 还提出了 HRU(HVN with deReference and Union，HR 和 HU 合称为 HRU)算法. HR 算法是基于指针变量与解引用变量之间的对应关系，而 HU 算法则是基于离线约束图上的抽象指向集的传递，两者都是对 HVN 的扩展，对 HVN 的分析结果做更深层次的精化. HRU 的复杂度为 $O(n^4)$. 虽然复杂度更高，但相对于 OVS 而言，HRU 显著增加了发现指针等价变量的机会.

根据 Hardekopf 等人[30]的分析，在基于包含的指针分析最终结果中，仍有大量满足指针等价的变量未被检测出来，指针等价变量的查找算法还有较大的提升空间，因此之后仍然有相关的研究工作致力于更大程度挖掘程序中指针等价的变量. 例如 Simon 等人[31]借鉴了同似性(Bisimilarity，简称 BSM)的概念和思想[32]，提出了一种新的技术用于查找指针等价的变量. 他们基于超级图(Superset Graph)和模拟图(Simulation Graph)来判断图中节点是否是同似的(Bisimilar)，如果满足该条件，则节点所代表的变量就是指针等价的，BSM 算法的复杂度为 $O(n^3)$. BSM 也仍有可改进的空间，例如基于更好设计和定义的模拟图应该可以检测出更多指针等价和位置等价的变量，此外，BSM 不仅能够作为一种离线优化的算法，同时该方法也可与 OCD 一样在约束求解过程中执行，但是由于其较高的复杂度，其执行频度必须控制得更小. 整体上，Simon 的方法不仅相对于 HRU 等算法找出了更多指针等价的变量，而且一定程度上拓宽了离线优化算法改进的思路.

值得一提的是，指针等价的思想不仅仅在基于包含的指针分析中作为离线优化的策略，在整个指针分析领域也有广泛使用[5,33-34].

4.2　位置等价

对于任意变量 x、y，如果 $x \in pts(z)$，则一定有 $y \in pts(z)$ 成立；反之，如果 $y \in pts(z)$，则一定有 $x \in pts(z)$ 成立时，我们称满足上述条件的变量 x 和变量 y 是位置等价的. 换言之，位置等价的变量必定同时存在于某个或者某些变量的指向集中. 同样，找出位置等价的变量的目的也是为了进行替换，一个指向集中所有位置等价的变量可以用一个变量表示，从而可以显著减少算法的空间开销. 但是与指针等价不同的是，直接节点所代表的变量不会被其它

变量间接引用到,但是位置等价的变量则有可能,因此在约束求解过程中还需用到变量替换过程中的映射关系.

查找位置等价变量的算法(简称为 LE 算法)由 Hardekopf 等人[30]首先提出. LE 算法的复杂度为 $O(n)$,算法本身并不复杂,它基于 HRU 算法的分析结果,在离线约束图上对形如 $\&a$ 的节点计算其所有后继结点的标号集合,具有相等的标号集合的这种节点即被赋上相同的位置等价标签. BSM 算法[31]也能找出位置等价的变量,但是效果不如 LE 算法. 在指针分析之外,Liang 等人[35]也将位置等价的思想用于程序切片等数据流分析的优化.

4.3 实 验

本文从上述离线优化算法中选取了其中较具代表性的工作进行对比,包括 OVS[29]、HRU[30]、HRU+LE[30] 3 种算法. 各优化算法基于初始的约束文件进行优化,之后在优化的基础上采用 LCD 算法[6]进行分析,采用 LCD 的原因在于,基于包含的指针分析算法的复杂度较高,而 LCD 是目前效率较高的分析算法. 整个过程的开销包括优化部分的开销和分析的开销. 为显示优化的效果,实验中也加入初始的约束文件一并分析(在表 7 和表 8 中以"ORIGIN"标示),但是这部分由于没有做离线优化,因此只有分析的开销.

实验平台配置和测试用例与第 3.4 节相同. 实验流程同样是由 CIL 前端生成各个测试用例初始的约束,并将其写入文件中,之后采用不同的离线优化算法对初始约束进行优化,并统计优化过程中的开销;在离线优化的基础上,调用 LCD 算法进行约束求解,并统计求解过程中的开销. 表 5 和表 6 显示了各算法优化过程中的时间和空间开销. 表 7 和表 8 显示了分析过程中的时空开销,其中由于未经优化的初始约束规模过大,对 linux 和 wine 两个测试用例的初始约束的分析都未能完成(OOM 表示 Run-Out of Memory).

图 6 显示了各优化算法优化之后的约束文件相对于初始未经优化的约束文件的比例,比例越小说明优化效果越明显.

表 5 各离线优化算法的性能

测试程序	时间开销/s		
	OVS	HRU	HRU+LE
emacs	0.15	0.33	0.37
gimp	1.08	3.21	3.56
gs	0.32	1.97	2.22
inst	0.58	4.26	4.61
lnx	1.42	8.79	9.65
wine	2.48	9.48	10.60

表 6 各离线优化算法的空间开销

测试程序	空间开销/MB		
	OVS	HRU	HRU+LE
emacs	22.04	24.31	24.98
gimp	109.93	129.49	133.25
gs	49.67	60.53	62.12
inst	69.57	141.05	143.25
lnx	173.68	358.63	364.96
wine	198.48	606.86	614.52

表 7 采用 LCD 分析算法的性能

测试程序	时间开销/s			
	ORIGIN	OVS	HRU	HRU+LE
emacs	13.79	2.04	0.0093	0.0043
gimp	77.35	29.98	5.15	0.040
gs	28.49	9.63	7.11	0.24
inst	140.82	50.88	11.69	1.61
lnx	OOM	220.84	62.95	2.31
wine	OOM	820.31	469.60	2.25

表 8 采用 LCD 分析算法的空间开销

测试程序	空间开销/MB			
	ORIGIN	OVS	HRU	HRU+LE
emacs	195.61	20.63	2.52	2.12
gimp	3120.44	425.13	145.95	8.42
gs	594.49	94.98	54.62	8.69
inst	2071.40	235.54	111.25	13.46
lnx	OOM	1002.60	389.40	37.07
wine	OOM	1750.16	1120.93	42.79

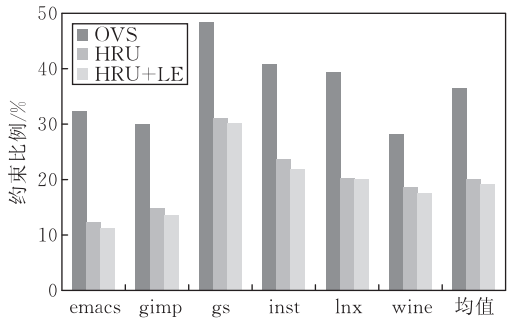


图 6 离线优化之后的约束相对于初始约束的比例

图 7 显示了 LCD 分析算法结合不同的离线优化算法的运行时间对比,需要强调的是,此处的时间开销由各离线优化算法的优化时间加上 LCD 分析算法的执行时间所组成. 各算法基于 LCD+HRU

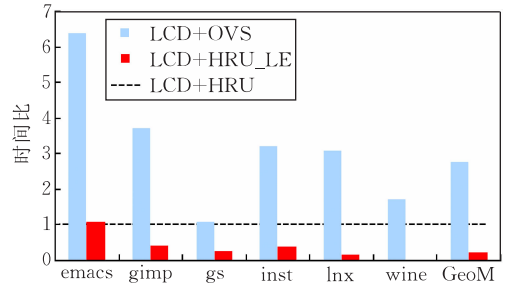


图 7 时间开销对比

做了归一化，如图中虚线所示，由于对 linux 和 wine 两个测试程序的初始约束文件的分析未能完成，因此没有将 LCD+ORIGIN 的组合列入图中. 更小规模的约束图意味着更小开销的迭代求解，从图中可以明显看出各优化算法所获得的性能提升.

如图 8 所示，相对于 OVS，HRU 发现并且合并了更多指针等价的变量，显著减少了内存开销，平均使用的内存只占 OVS 所使用的 57%. 由于增加了对位置等价变量的合并和替换，HRU+LE 算法相对于 HRU 算法也减少了一部分内存开销，但是除了较大规模的程序(如 wine)之外，减少的幅度并不大. 由于离线优化和在线分析两个过程可以独立进行，所以此处统计的内存开销是取自两部分的最大者. 与其它算法不同的是，HRU+LE 算法优化过程中的空间开销显著高于分析部分的空间开销，这也是图 8 中各算法之间对比度没有图 7 显著的原因.

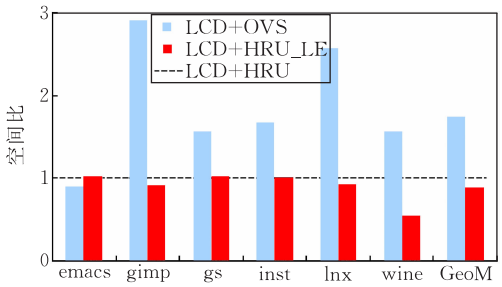


图 8　空间开销对比

5　通用改进技术

本章简要介绍指针分析的通用改进技术，主要包括精度提升和指向集表示. 这些技术在一定程度上具有普适性，不仅适用于基于包含的指针分析同样也适用于其它指针分析方法.

5.1　精度提升

Andersen[4] 所提出的初始算法本质上是流不敏感的，同时也是基于域(field-based)的和上下文不敏感的，但可以结合域敏感、上下文敏感等提升其分析精度.

广义上，域敏感的分析包括区分结构体(或共用体、对象等)的不同域、函数指针的处理、堆建模、指针运算、常量处理等等，其中最关键的问题是如何区分和处理结构体(共用体)变量的不同域. 在指针分析领域，对于结构体(或共用体、对象等)域的处理通常有 3 种策略：

(1) 域不敏感(field-insensitive). 对域不加区分，例如将 $s.a$，$s.b$ 等同视为同一个变量 s；

(2) 基于域的(field-based). 区分域，但是不区分含有相同域的不同结构体(或联合体)实例. 例如将 $s1.a$ 和 $s2.a$ 视为同一变量，而将 $s1.a$ 和 $s1.b$ 视为两个不同变量；

(3) 域敏感(field-sensitive). 区分域，同时区分不同结构体实例.

基于包含的指针分析的多数工作都是域不敏感或者基于域的，相对而言，域敏感[21,36-38] 的算法无论是在设计还是实现上都比上述两种方式更为复杂，对于指针使用极为灵活的 C 语言尤其如此. 在针对 C 的基于包含的域敏感指针分析研究方面，Pearce 等人[21] 提出的算法较具影响力，该算法在域敏感方面的改进主要集中在两方面：首先是对于函数指针的处理，在 Pearce 的扩展规则中，变量加上偏移值就可以对应到不同的域变量，其算法用函数的第一个形参地址代表当前函数的基地址(类似的，结构体的基地址用第一个域地址表示)，同时在推导规则中对偏移的范围作了限定，以防止函数指针之间的类型转换所带来的参数不匹配的问题；其次是对复杂域操作的刻画，类似于对域取地址这样的操作必须做特殊处理，将约束图中的有向边扩展为加权有向边，权重表示偏移值. Pearce 算法的复杂度为 $O(n^4)$，其中 n 表示约束图中节点数量，其文献中的实验结果显示域敏感能大幅度提升基于包含的指针分析的精度. 此外，域敏感还可进一步精化，于洪涛等[39] 将域的区分与机器模型的数据布局结合起来，使得域敏感的分析在特定机器上能达到更高的精度，该技术虽然基于 Steensgaard 指针分析，但也可扩展到其它指针分析方法.

上下文敏感是提升指针分析精度的重要方法，但其难点主要在于如何处理由于区分大量调用点的上下文而导致的空间爆炸问题. 广义上的上下文敏感分析有两种实现方式.

第 1 种实现方式称为调用链(calling-string)或者 k-CFA，由函数调用的调用栈信息构成不同的调用上下文，直观上，调用链对应的是调用栈中的一个函数序列. 在实现中，往往对调用链的长度(假设为 k)进行控制，即只分析最初始的 k 个调用点的信息，以获得分析效率和精度之间的折中. 该实现方式结合其它技术可以获得较好的效率和精度，如 Whaley 等人[40-41] 提出了基于过程克隆的上下文敏感的指针分析算法，能够利用上下文不敏感的分析达到上下文敏感的效果.

第 2 种实现方式称为转移函数(transfer function),本质上是过程内区域(region)分析向过程间的扩展.转移函数反映了函数入口参数和出口参数之间的关系,给定函数入口参数中与指针相关的信息,由转移函数直接计算函数出口参数中所包含的指针信息.程序分析之前,遍历调用图对每一个函数构建其转移函数,之后以函数不同的调用点的上下文信息为输入应用转移函数得出指针分析的结果,以实现上下文敏感的分析.大部分上下文敏感的指针分析都采用了这种实现方式[42-44].黄波等人[45]提出了一种指针指向信息在过程间的传播方法,该方法通过对函数调用点处指针指向信息内的表达式进行标记的策略一次性求得需要映射到被调函数中的指针指向信息.实验结果显示,该方法分析精度与 Wilson 等人[44]算法相当.此外,刘强[46]、孙洪浩等人[47]也对上下文敏感指针分析算法的改进做了有益尝试.

除域敏感和上下文敏感之外,还有其它提升指针分析精度的方法和策略,如堆建模、对象敏感等,但限于篇幅且非本文重点关注的方面,因此不再赘述.

5.2 指向集表示

在指针分析中,一般采用位向量(bit vector)或者二元决策图(Binary Decision Diagram,BDD)来表示指向集,二者都能有效地降低存储开销,但在存储和操作效率上还是有一些差异.位向量是使用广泛的传统的数据结构,已有不少开源的实现,如 GCC 中的 bitmap、Open64 编译器中的 SPARSE_BV 等.Berndl 等人[48]首次在指针分析中引入了 BDD,用于表示约束图和指向集,之后 Whaley 等人[40]以及 Hardekopf 等人[19,30]也在指针分析中使用 BDD.BDD 是一种以紧凑形式表示数据的数据结构,且能够进行比较快速的操作,目前已有许多不同的 BDD 表示和实现库.BDD 也被广泛用于程序分析领域中,如用于形状分析[49-50]和谓词抽象[51]等.

图 9 显示了分别采用 BDD 和位向量表示指针指向集的结果.其中,BDD 将变量分为两个域:变量 a,b,c 属于变量域 V,指向对象 x,y,z 属于指向域 P.这里为简单起见,用两个 bit 表示一个元素,例如用 00 表示 a 和 x,01 表示 b 和 y,10 表示 c 和 z.因此指向关系 a 指向 x 可以编码为 0000,c 指向 z 编码为 1010,其它指向关系以此类推.最终的表示结果如子图(a)所示,图中只含有 5 个 BDD 节点(结果 0 和 1 的表示除外).用位向量表示指向集如子图(b)所示,图中给每一个变量一个标号,如假设变量 x、y、z 的标号分别为 24、25、26.相对而言,BDD 表示虽然不如位向量直观方便,但是存储开销更小.

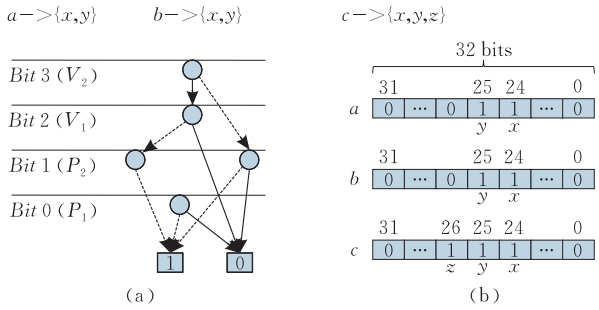


图 9 用 BDD 和位向量分别表示指向集

BDD 表示的大小取决于所采用的变量的顺序,但决定 BDD 表示中变量的最优顺序是一个 NP 难的问题[52],所以在工程实现的时候,往往是找一个相对较优的顺序,尽量减少空间开销,最大程度上发挥 BDD 的作用.文献[48]中对此问题做了详细阐述.

一般而言,二元决策图相对于位相量来说,在存储效率上优于后者,在操作效率上不如后者.图 9 只从存储开销上做了简单对比,两者之间的具体差异可以参考文献[19,30,48]中给出的相关对比数据.

6 总结和展望

指针分析是程序分析和编译优化的基础,也一直是程序分析领域的热点研究问题.到目前为止,已经有相当多的研究工作关注于指针分析的算法改进,主要可以分为两类:流敏感指针分析算法的改进和流不敏感指针分析算法的改进.本文主要回顾和总结了近二十年来关于流不敏感的基于包含的指针分析以下两方面的工作:在线优化和离线优化,并选取其中 7 种较具影响力的在线优化算法和 3 种离线优化算法通过实验进行了对比和评述.

虽然结合现有的各种优化技术,基于包含的指针分析算法的效率已经得到了很大提升,但随着计算机硬件的发展、软件规模的增长以及应用需求的变化,对指针分析的要求也在改变.我们认为,基于包含的指针分析未来潜在的研究方向包括如下几个方面:

(1)分析效率的进一步提升.一方面是离线优化与在线分析的结合,HCD[19]是一个值得借鉴的范例,它将离线预先分析所得到的信息用于在线的优化;另一方面,离线的优化还有可提升的空间,特别

是如何以高效的方法找到更多指针等价的变量，在这方面，BSM[31]将同似(bisimilarity)的概念与指针等价、位置等价联系起来，提供了一种离线优化的新思路.此外，无论是离线优化还是在线优化，其本质上都是在一个有向图(约束图)中查找满足特定属性(指针等价、位置等价、强连通分量)的节点和边并将其合并的过程，因此，将来的研究也许可以结合图论中的先进算法或技术进行更深入的优化.

(2)客户分析需求驱动.一般而言，指针分析精度越高，对后续的客户分析(client analysis)越有利.但问题在于一方面全敏感(流敏感、上下文敏感结合域敏感等)的指针分析的开销太大，另一方面不同的客户分析对指针分析精度的要求也可能是不同的.在这种情况下，需求驱动(demand-driven)的指针分析[53-56]可以作为一种解决方案，它只在客户分析有需求(query)的时候做必要的分析，其最大的优势在于避免了对全程序进行分析所带来的较大时空开销，同时可以适用逐步精化的指针分析[53]，对不同的客户需求采用不同精度的分析，此外相同需求的分析结果还可以保存起来复用以减少重复分析的代价.

(3)指针分析并行化.多核为并行提供了天然丰富的计算资源，相对于优化而言，并行所带来的效率提升更为显著，但是对于基于包含的指针分析算法而言，并行化的关键问题是要解决如何处理不规整的数据结构同时采取何种并行方式以避免并行任务之间的数据依赖.Méndez-Lojo 等[57]对 Java 程序上的基于包含的指针分析做了并行化的尝试，在八核机器上的平均加速比能达到 3 左右，并行所带来的效率提升相当可观.

(4)并行程序的指针分析.并行程序的指针分析是并行程序分析和检测的基础，如静态数据竞争(race condition)的检测等，该方向的研究仍然较为初步但需求却由于多核时代的来临而变得日益迫切.实际上已有相关的针对并行程序指针分析的研究，如文献[58]是针对多线程的 CILK 程序所设计的指针分析方法，该算法整体是流敏感的，但是对于并行区域中的代码则做流不敏感的处理.未来的研究需要关注如何结合并行程序的固有特点进一步提升分析的精度，与针对并行程序的流分析框架相结合可能是一个研究趋势.

(5)指针分析与软件检测.软件的可靠性日益受到重视，因此软件检测、程序验证等研究领域也日益成为热点.编译器由于具有强大的程序分析能力，可以作为软件检测的一个良好的支撑平台.与服务于传统的编译优化不同，服务于软件检测的指针分析有其新的特点[41,59-61].例如，对于优化而言，指针分析的结果首先必须正确以保证优化的正确，因而往往是保守的；但对于软件检测而言，指针分析的结果则可以相对激进一些，因为在有可能减少漏报的同时增加一些误报信息并不会影响检测工具本身的功能.

参　考　文　献

[1] Aho A V, Lam M S, Sethi R, Ullman J D. Compilers: Principles, Techniques, and Tools. 2nd Edition. Boston: Pearson/Addison Wesley, 2007

[2] Landi W. Undecidability of static analysis. ACM Letters on Programming Languages and Systems, 1992, 1(4): 323-337

[3] Ramalingam G. The undecidability of aliasing. ACM Transactions on Programming Language and Systems, 1994, 16(5): 1467-1471

[4] Andersen L O. Program analysis and specialization for the C programming language[Ph. D. dissertation]. University of Copenhagen, DIKU, Denmark, 1994

[5] Steensgaard B. Points-to analysis in almost linear time//Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. St. Petersburg Beach, FL, 1996: 32-41

[6] Aiken A. Set constraints: Results, applications and future directions//Proceedings of the Workshop on Principles and Practice of Constraint Programming. Rosario, Orcas Island, Washington, USA, 1994: 326-335

[7] Aiken A. Introduction to set constraint-based program analysis. Science of Computer Programming, 1999, 35(2-3): 79-111

[8] Heintze N. Set-based analysis of ML program//Proceedings of the ACM Conference on Lisp and Functional Programming. Orlando, Florida, USA, 1994: 306-317

[9] Hind M. Pointer analysis: Haven't we solved this problem yet? //Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program Analysis for Software Tools and Engineering. Snowbird, Utah, USA, 2001: 54-61

[10] Hind M, Pioli A. Which pointer analysis should I use?//Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis. Portland, Oregon, USA, 2000: 113-123

[11] Ryder B G. Dimensions of precision in reference analysis of object-oriented programming languages//Proceedings of the 12th International Conference on Compiler Construction. Warsaw, Poland, 2003: 126-137

[12] Tarjan R. Depth-first search and linear graph algorithm. SIAM Journal on Computing, 1972, 1(2): 146-160

[13] Nuutila E, Soisalon-Soininen E. On finding the strongly connected components in a directed graph. Information Processing Letters, 1993, 49(1): 9-14

[14] Fähndrich M, Foster J S, Su Z, Aiken A. Partial online cycle elimination in inclusion constraint graphs. ACM SIGPLAN Notices, 1998, 33(5): 85-96

[15] Su Z, Fahndrich M, Aiken A. Projection merging: Reducing redundancies in inclusion constraint graphs//Proceedings of the 2000 Symposium on Principles of Programming Languages, Boston, Massachusetts, 2000: 81-95

[16] Heintze N, Tardieu O. Ultra-fast aliasing analysis using CLA: A million lines of C code in a second. ACM SIGPLAN Notices, 2001, 36(5): 254-263

[17] Whaley J, Lam M S. An efficient inclusion-based points-to analysis for strictly-typed languages//Proceedings of the 9th Symposium on Static Analysis (SAS). London, UK, 2002: 180-195

[18] Pearce D J, Kelly P H J, Hankin C. Online cycle detection and difference propagation: Application to pointer analysis. Software Quality Journal, 2004, 12: 311-337

[19] Hardekopf B, Lin C. The ant and the grasshopper: Fast and accurate pointer analysis for millions of lines of code//Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation, San Diego, California, USA, 2007: 290-299

[20] Pereira F M Q, Berlin D. Wave propagation and deep propagation for pointer analysis//Proceedings of the 2009 International Symposium on Code Generation and Optimization. Washington, DC, USA, 2009: 126-135

[21] Pearce D J, Kelly P H J, Hankin C. Efficient field-sensitive pointer analysis of C. ACM Transactions on Programming Languages and Systems, 2007, 30(1): 1-42

[22] Pearce D J. Some derected graph algorithms and their application to pointer analysis[Ph. D. dissertation]. London UK: Imperial College, 2004

[23] Pearce D J, Kelly P H J, Hankin C. A dynamic algorithm for topologically sorting directed acyclic graphs. Journal of Experimental Algorithmics (JEA), 2006, 11(1): 1-24

[24] Lhoták O, Hendren L J. Scaling java points-to analysis using SPARK//Proceedings of the 12th International Conference on Compiler Construction. Warsaw, Poland, 2003: 153-169

[25] Bourdoncle F. Efficient chaotic iteration strategies with widening//Proceedings of the Conference on Formal Methods in Programming and their Applications. Novosibirsk, Russia, 1993: 128-141

[26] Chen L L, Harrison W L. An efficient approach to computing fixpoints for complex program analysis//Proceedings of the ACM Conference on Supercomputing. Manchester, England UK, 1994: 98-106

[27] Jones J A, Harrold M J, Stasko J. Efficient evaluation of circular attribute grammars. ACM Transactions on Programming Languages and System, 1990, 12(3): 429-462

[28] Necula G C, McPeak S, Rahul S P, Weimer W. CIL: Intermediate language and tools for analysis and transformation of C programs//Proceedings of the 11th International Conference on Compiler Construction, In Computational Complexity. Grenoble, France, 2002: 213-228

[29] Rountev A, Chandra S. Off-line variable substitution for scaling points-to analysis. ACM SIGPLAN Notices, 2000, 35(5): 47-56

[30] Hardekopf B, Lin C. Exploiting pointer and location equivalence to optimize pointer analysis//Proceedings of the International Static Analysis Symposium (SAS). Kongens Lyngby, Denmark, 2007: 265-280

[31] Simon L. Optimizing pointer analysis using bisimilarity//Proceedings of the International Static Analysis Symposium (SAS). Los Angeles, CA, 2009: 222-237

[32] Dovier A, Piazza C, Policriti A. An efficient algorithm for computing bisimulation equivalence. Theoretical Computer Science, 2004, 311(1-3): 221-256

[33] Shapiro M, Horwitz Su. Fast and accurate flow-insensitive points-to analysis//Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. Paris, France, 1997: 1-14

[34] Das M. Unification-based pointer analysis with directional assignments//Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation. Vancouver, Canada, 2000: 35-46

[35] Liang D L, Harrold M J. Equivalence analysis and its application in improving the efficiency of program slicing. ACM Transactions on Software Engineering and Methodology (TOSEM), 2002, 11(3): 347-383

[36] Yong S H, Horwitz S, Reps T. Pointer analysis for programs with structures and casting//Proceedings of the ACM Conference on Programming Language Design and Implementation(PLDI). Atlanta, Georgia, USA, 1999: 91-103

[37] Chandra S, Reps T. Physical type checking for C//Proceedings of the ACM Workshop on Program Analysis for Software Tools and Engineering (PASTE). Toulouse, France, 1999: 66-75

[38] Johnson R, Wagner D. Finding user/kernel pointer bugs with type inference. University of California, Berkeley, California: UCS/CSD-04-1308, 2004

[39] Yu Hong-Tao, Zhang Zhao-Qing. An aggressive field-sensitive unification-based pointer analysis. Chinese Journal of Computers, 2009, 32(9): 1722-1735(in Chinese)
(于洪涛, 张兆庆. 激进域敏感基于合并的指针分析. 计算机学报, 2009, 32(9): 1722-1735)

[40] Whaley J, Lam M S. Cloning-based context-sensitive pointer alias analysis//Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation (PLDI). Washington, DC, USA, 2004: 131-144

[41] Avots D, Dalton M, Livshits V B, Lam M S. Improving software security with a C pointer analysis//Proceedings of the 27th International Conference on Software Engineering. St Louis, Missouri, USA, 2005: 332-341

[42] Fähndrich M, Rehof J, Das M. Scalable context-sensitive flow analysis using instantiation constraints//Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation, Vancouver. British Columbia, Canada, 2000: 253-263

[43] Whaley J, Rinard M. Compositional pointer and escape analysis for Java programs//Proceedings of the 14th ACM SIGPLAN conference of Object Oriented Programming, Systems, Languages, and Applications. Denver, CO, USA, 1999: 187-206

[44] Wilson R P, Lam M S. Efficient context-sensitive pointer analysis for C programs//Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation. La Jolla, CA, USA, 1995: 1-12

[45] Huang Bo, Zang Bin-Yu, Wei Jun-Yin, Zhu Chuan-Qi. Context-sensitive interpeocedural pointer analysis. Chinese Journal of Computers, 2000, 23(5): 477-485(in Chinese)
(黄波，臧斌宇，韦俊银，朱传琪. 上下文敏感的过程间指针分析. 计算机学报，2000，23(5)：477-485)

[46] Liu Qiang. Study on inter-procedural analysis techniques in the presence of pointer aliasing[Ph. D. dissertation]. Graduate University of Chinese Academy of Sciences (Institute of Computing Technology), Beijing, 1998(in Chinese)
(刘强. 考虑指针别名的过程间分析技术研究[博士学位论文]. 中国科学院研究生院(计算技术研究所)，北京，1998)

[47] Sun Hong-Hao, Wang Shu-Yi, Lin Xiao-Bin, Wang Xiao-Fei. Improvement of inter-procedural pointer analysis algorithm. Computer Engineering, 2009, 35(1): 90-92, 104(in Chinese)
(孙洪浩，王树义，林晓斌，王晓飞. 过程间指针分析算法的改进. 计算机工程，2009，35(1)：90-92，104)

[48] Berndl M, Lhoták O, Qian F, Hendren L, Umanee N. Points-to analysis using BDDs. ACM SIGPLAN Notices, 2003, 38(5): 103-114

[49] Manevich R, Ramalingam G, Field J, Goyal D, Sagiv M. Compactly representing first-order structures for static analysis//Proceedings of the 9th International Static Analysis Symposium. Madrid, Spain,2002: 196-212

[50] Kahveci T Y, Bultan T. Automated verification of concurrent linked lists with counters//Proceedings of the 9th International Static Analysis Symposium. Madrid, Spain, 2002: 69-84

[51] Ball T, Rajamani S K. Bebop: A symbolic model checker for boolean programs//Proceedings of the 7th International SPIN Workshop on Model Checking of Software. California, USA, 2000: 113-130

[52] Meinel C, Theobald T. Algorithms and Data Structures in VLSI Design. New York, USA, 1998

[53] Manu Sridharan, Rastislav Bodik. Refinement-based context-sensitive points-to analysis for Java//Proceedings of the Programming Language and Design Implementation. Ottawa, Ontario, Canada, 2006: 387-400

[54] Sridharan M, Gopan D, Shan L, Bodik R. Demand-driven points-to analysis for Java//Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications. San Diego, CA, USA, 2005: 59-76

[55] Heintze N, Tardieu O. Demand-driven pointer analysis//Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation. Snowbird, Utah, USA, 2001: 24-34

[56] Zheng X, Rugina R. Demand-driven alias analysis for C//Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Language. San Francisco, CA, USA, 2008: 197-208

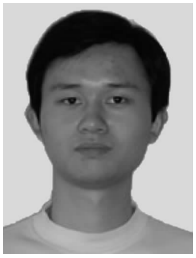
[57] Mario Méndez-Lojo, Augustine Mathew, Keshav Pingali. Parallel inclusion-based points-to analysis//Proceedings of the ACM International Conference on Object-Oriented Programming Systems Language and Applications. Reno/Tahoe, Nevada, USA, 2010: 428-443

[58] Rugina R, Rinard M C. Pointer analysis for structured parallel programs. ACM Transactions on Programming Languages and Systems, 2003, 25: 10-116

[59] Zhang Wei, Lu Qing-Ling, Li Mei, Gong Yun-Zhan. Research on memory leak faults testing method based on pointer analysis. Application Research of Computers, 2006, 23(10): 22-24(in Chinese)
(张威，卢庆龄，李梅，宫云战. 基于指针分析的内存泄露故障测试方法研究. 计算机应用研究，2006，23(10)：22-24)

[60] Zhang Ming-Jun, Luo Jun. Pointer analysis algorithm in static buffer overflow analysis. Computer Engineering, 2005, 31(18): 41-43, 107(in Chinese)
(张明军，罗军. 缓冲区溢出静态分析中的指针分析算法. 计算机工程，2005，31(18)：41-43，107)

[61] Chen Yi-Yun, Hua Bao-Jian, Ge Lin, Wang Zhi-Fang. A pointer logic for safety verification of pointer programs. Chinese Journal of Computers, 2008, 31(3): 372-380(in Chinese)
(陈意云，华保健，葛琳，王志芳. 一种用于指针程序安全性证明的指针逻辑. 计算机学报，2008，31(3)：372-380)



CHEN Cong-Ming, born in 1985, Ph. D. candidate. His research interests include program analysis and compiler optimization.

HUO Wei, born in 1981, Ph. D.. His research interests include program analysis and compiler optimization.

YU Hong-Tao, born in 1981, Ph. D.. His research interests include program analysis and compiler optimization.

LI Feng, born in 1985, Ph. D. candidate. Her research interests include program analysis and compiler optimization.

FENG Xiao-Bing, born in 1969, Ph. D., researcher, Ph. D. supervisor. His research interests include compiler optimization and binary translation.

Background

Pointer analysis is the basis of other static program analysis, it was one of the most important research areas of program analysis in the past 30 years, and it is still very active nowadays. The basic research of pointer analysis focuses on the trade-off between precision and performance.

Inclusion-based pointer analysis and unification-based pointer analysis are the most important two algorithms of flow-insensitive pointer analysis. Inclusion-based pointer analysis has a higher time complexity compared with unification-based pointer analysis, but it also has better precision, and there's more improvement room for inclusion-based pointer analysis. A lot of research in pointer analysis is about improving the performance of inclusion-based pointer analysis and a lot of improving techniques and algorithms have been proposed. We introduced and evaluated several important improving algorithms in this paper mainly from two aspects, which were online optimization and offline optimization, these algorithms were the most representative work in related research, in addition, representation of points-to set and precision promotion were also discussed. We compared seven online optimizing algorithms and three offline optimizing algorithms by significant experiments. We also discussed some further improvements and research points of pointer analysis in this paper. To the best of our knowledge, this paper is the first optimization techniques survey of inclusion-based pointer analysis.

This work is supported by a Chinese National High Technology Research and Development Grant (2008AA01Z115), a Chinese National Basic Research Grant (2011CB302504), a Chinese National Science and Technology Major Project (2009ZX01036-001-002), a Project supported by the Foundation for Innovative Research Groups of the National Natural Science Foundation of China (Grant No. 60921002). These projects aimed at improving software reliability and security by program analysis, program analysis methodologies including: control-flow analysis, data-flow analysis, program slicing and so on. Most of these analyses are based on the result of pointer analysis.

We have been focusing on static program analysis for several years, while pointer analysis is one of our most important research areas. We have improved the precision of unification-based pointer analysis by combining low level data layout information within a specified machine, we also proposed an new flow-sensitive and context-sensitive pointer analysis algorithm which was accepted by CGO'2010. Since inclusion-based pointer analysis algorithm is commonly used in most open-source or commercial compilers, such as GCC, LLVM etc, and the primary improvements related to this pointer analysis are optimization techniques for reducing analysis overhead in the recent twenty years, we believe that a survey of these optimizations will really make sense, which also can be the basis of our future research.