# 数据访问相似性编译分析工具的设计与实现*

刘逊韵[1]，吴俊杰[1]，唐玉华[1]

[1](国防科学技术大学高性能计算国家重点实验室， 湖南 长沙 410073)

**摘 要** 处理器和存储器之间的速度差距一直是计算机系统的性能瓶颈。相似性描述了程序的多个执行体中对应的多个数据单元内容之间的关系，可用于优化多个执行体对存储器的占用量。本文设计和实现了自动求解相似数据的编译分析工具，该工具以 C 语言编写的程序源代码为输入，输出每个程序点的相似数据集合，为后续的相似性优化技术提供支持。实验结果验证了该工具的正确性和有效性。

关键词 相似性；差异传播模型；编译分析工具

**中图法分类号** **** DOI 号：*投稿时不提供 DOI 号*

# Design and Implementation of a Tool for Automatic Similarity Compile Analysis *

LIU Xun-Yun[1], WU Jun-Jie[1], TANG Yu-Hua[1]

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

**Abstract** The speed gap between the processors and memories has been one of the system performance bottlenecks. similarity describes the relationship of the values of corresponding data elements in multiple execution entities of one program. It is used to optimize the occupation of memory resource for multiple execution entities. The purpose of this paper is to design and implement an automatic similarity compile analysis tool based on similarity theory. The tool takes program written in C language as input, output similar data set for each program point, and provides technical support for similarity optimization. The experimental results show that all similar data can be detected properly.

**Key words** similarity; differences propagation; compiler analysis tool

## 1 引言

处理器和存储器的速度差异一直是计算机系统最严重的性能瓶颈之一[1]。随着多核处理器计算性能地不断提高，访存优化技术成为了当今研究的热点[2]。为减少和避免核与核通信时对片外存储器的访问。访存优化技术的重要命题就是要高效地利用片上共享存储结构的容量资源。

在多核系统的运行中，会常见到同时运行的多个执行体（进程或线程）都派生自同一个程序的情况。程序中的大部分变量在不同的执行体（特别是进程）中对应不同的存储单元。但当某个程序变量对应的多个存储单元具有相同的值时，那么这个变量称为相似数据，对相似数据的访问具有相似性[3]。

相似数据在片上共享 Cache 中占用了不同的位置，严重影响了共享存储结构的利用率。本文依据相似数据量化分析理论，运用数据流分析方法[3]，采用编译手段给出了源代码中各程序点的相似数据信息。其具体贡献如下：

收稿日期：年-月-日*投稿时不填写此项*；最终修改稿收到日期：年-月-日 *投稿时不填写此项*. 本课题得到国家自然科学基金(No.61003082)资助.
刘逊韵，男，1989年生，硕士生，E-mail: xunyunliu@gmail.com，主要研究领域为高性能计算
吴俊杰，男，1981年生，博士，E-mail: junjiewu@nudt.edu.cn，助理研究员，主要研究领域为计算机体系结构
唐玉华，女，1962年生，研究员，E-mail:yhtang62@163.com，主要研究领域为计算机体系结构
第1作者手机号码(投稿时必须提供，以便紧急联系，发表时会删除): 13787299675, E-mail: xunyunliu@gmail.com

1)基于正则表达式和有限状态自动机方法，依托 FLEX 快速词法分析程序产生器来构建自动编译分析工具。对面向 C 语言源代码的差异传播模型进行了具体实现，定量地求解了相似数据。

2)引入模块化的设计方法及分遍（PASS）编译的思想，词法分析模块识别出程序中出现的变量、语句以及基本块，并对其识别结果进行可视化。差异传播模块则在词法分析模块的分析结果上应用差异传播模型得出相似数据。

3)通过实验测试了编译分析工具的正确性与有效性，定量地分析了测试程序中的差异传播速度，传播能力的具体特征。

本文第 2 节介绍了编译分析工具所依赖相似性理论；第 3 节介绍了工具的总体结构；第 4、第 5 节分别介绍了词法分析模块与继生差异求解模块的具体实现方法；最后在第 6 节通过实验证实了工具的正确性，并对结果进行了分析。

# 2 预备知识

## 2.1 相似性

当同一个程序派生多个执行体时，如果这些执行体内和程序中某个变量对应的多个数据单元在同一段代码的执行过程中拥有相同的值，那么对这些数据单元的访问具有相似性。相似性描述了程序的多个执行体中对应的多个数据单元内容之间的关系。在图 1 例程 hello.c 中，同一个程序被执行了两次，两次执行的进程中变量 d 的取值相同。虽然两个进程中变量 d 的虚地址相同，但物理地址不同，在主存中对应不同的存储单元。也就是说，进程间变量 d 的访问具有相似性。

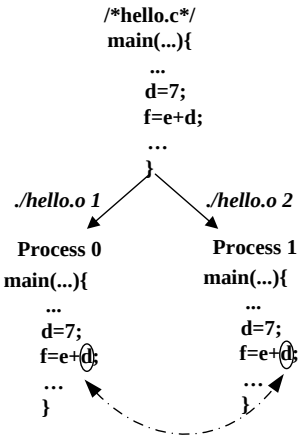


图 1 访问相似性举例

数据访问的相似性在程序中普遍存在。与相似性有关的优化技术也已经非常广泛。如 UNIX 系统中 copy-on-write 的存储管理机制[4,5]、虚拟机的存储优化共享技术[6-8]、支持相似数据合并的多核合并 Cache 技术等等[9]。

## 2.2 差异传播模型

差异传播模型是求解相似数据的静态数据流分析方法[3]。差异是与相似互补的概念，如果程序多个执行体间变量 a 的取值不同，那么变量 a 是差异数据。当差异数据来自于不同的输入或参数时，差异数据为原生差异，如图 2 中的变量 a。原生差异在程序中会沿着数据的依赖关系和表达式赋值进行传播，从而产生继生差异。

| | |
|---|---|
| **1** input a | **1** input a |
| **2** assert_diff(a) | **2** assert_diff(a) |
| **3** b=a+1 | **3** if a>0 then |
| | **4** b=0 |
| | **5** else |
| | **6** b=1 |

(a)数据流生差异 (b) 控制流生差异

图 2 继生差异分类图

由程序中表达式赋值引起的继生差异称之为数据流生差异，如在图 (2a)的第 3 行，变量 *a* 的差异将通过表达式赋值传播给变量 *b*；由不同的控制流路径引起继生差异称之为控制流生差异，如在图(2b)的第三行，差异变量 *a* 出现在分支语句的条件表达式中，因此不同的执行体可以有不同的执行路径，而两个分支中变量 *b* 被赋予了不同的值，所以变量 *b* 将变成控制流生差异。

原生差异与继生差异共同组成了差异数据的集合。在求解方法上，我们可以使用数据流分析的方法求解程序中每个点的差异信息。

数据流分析作为一项编译时使用的技术，它能根据程序代码静态地分析程序的结构和静态地收集变量的引用情况，并通过代数的方法在编译时确定变量的定义和使用，来收集计算机程序在不同点计算的值的信息。进行数据流分析的最简单的一种形式就是对控制流图的某个节点建立数据流方程，然后通过迭代计算，反复求解，直至到达不动点。

从结构上看，差异传播模型结构图如图 3 所示：

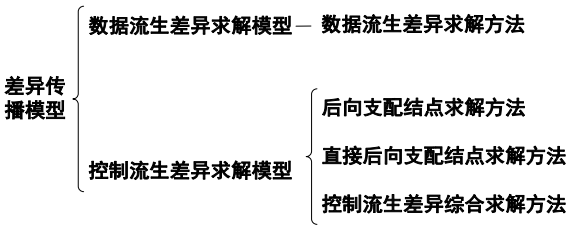


图 3 差异传播模型结构图

数据流生差异求解模型用于求解数据流生差异。它以语句为基本处理单元，采用正向工作集算法求解数据流方程，对于语句 *S*，其输入差异集合和输出差异集合满足如下数据流方程：

$$OUT_{IN(S)}(S)=GEN_{IN(S)}(S)\cup(IN(S)-KILL_{IN(S)}(S))$$

$$IN(S)=\bigcup_{S_p\in \mathrm{Pred}(S)} OUT_{IN(S_p)}(S_p)$$

其中，程序运行到语句 *S* 之前的程序点时的差异数据集合记为 *IN(S)*；程序运行到语句 *S* 之后的程序点时的差异数据集合记为 *OUT(S)*；语句 *S* 将新生成的差异数据集合记为 *GEN(S)*；语句 *S* 将消除的差异数据集合记为 *KILL(S)*。若语句 *S* 的引用变量列表里有任一个变量为差异数据，那么该条语句的所有赋值变量共同组成了 *GEN(S)*。否则，该条语句的所有赋值变量共同组成了 *KILL(S)*。

以图(2a)的第三条语句 *S*："b=a+1"为例，*IN(S)*={*a*}，*GEN(S)*={*b*}，*KILL (S)*=∅，*OUT(S)*={*a*，*b*}，Pred(S)={"assert_diff(a)"}。

控制流生差异求解模型则包含三个有严格执行顺序的过程：

1)后向支配结点求解方法

后向支配结点指到达程序出口的每条路径都必定经过的基本块。后向支配结点求解方法用于求解分支基本块的后向支配结点，求解后向支配结点的数据流方程如下所示：

$$IN^{node}(B)=\{B\}\cup OUT^{node}(B)$$

$$OUT^{node}(B)=\bigcap_{B_s\in Succ(B)} IN^{node}(B_s)$$

式中 $IN^{node}(B)$即为分支基本块 *B* 的后向支配结点集合，与数据流生差异求解方法不同的是该方法采用了逆向工作集算法。

2)直接后向支配结点求解方法

用于从基本块B的后向支配结点列表中求解直接后向支配结点。即在执行距离上最接近 B 的基本块。

3)控制流生差异综合求解方法

当控制流图中的某个结点B存在分支差异时，根据B的直接后向支配结点B´可以找到分支差异影响的基本块。控制流生差异求解方法将这些基本块中所有被定值的数据都标记成差异数据

最后，继生差异求解方法综合考虑了数据流生差异和控制流生差异的相互作用。因此差异传播模型的工作流程可概括为：首先运用逆向数据流分析方法，求解程序控制流图中各个结点的后向支配结点集，再从后向支配结点集中求解直接后向支配结点，尔后运用正向数据流分析方法求解数据流生差异，对含差异数据的分支语句，计算控制流生差异（从其本身到其直接后向支配结点之间所有被定值的变量）。求解数据流生差异与求解控制流生差异构成一个循环，直至没有新的差异数据产生，循环才结束。

## 3 工具的总体结构

在本文的具体实现中，编译分析工具以标记了原生差异的 C 源程序为输入，输出内容包括待测程序识别的变量表、语句表、基本块表及各程序点的差异信息与相似信息。其总体组成结构如图 4 所示：

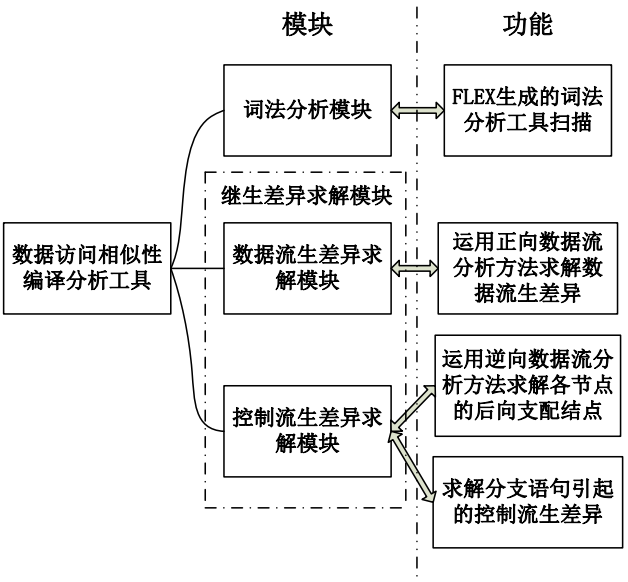


图 4 工具总体组成结构图

其中，词法分析模块运用 FLEX 构建的词法分析模块对源代码进行扫描，识别出代码中包含的变量、语句、基本块及其相互关系。将代码抽象为变量表、语句表、基本块表和一些标志量，为第二阶段继生差异求解做出铺垫。它对应 FLEX 快速词法分析程序产生器的正则表达式的处理规则。

而继生差异求解模块则基于C语言对差异传播模型进行了具体实现。

# 4 词法分析模块

我们采用 FLEX（The Fast Lexical Analyzer）作为词法分析模块的生成工具，FLEX 的输入文件中包含若干对规则，每一条规则由一个正则表达式和其相应的 C 源代码组成，其中 C 源代码描述了对该词法模式所需采取的识别动作。FLEX 的输出为一个名为“lex.yy.c”的 C 文件，该文件可以被编译为可执行的词法分析程序。程序运行时分析输入文件中出现的正则表达式，每找到一个，就执行其相应的 C 代码[10]。

根据差异传播模型对程序源代码中出现的变量，定值表达式以及数据的依赖关系的识别要求，我们书写了 18 条 FLEX 规则，这些规则按识别的词法模式分类如表 1 所示:

表 1 FLEX 规则分类表

| 规则类型 | 规则处理对象（正则表达式） |
|---|---|
| 类型声明符 | “int” |
| 关键字 | “if”、“else”、“for”、“break”、“continue” |
| 差异断言符 | “assert_diff” |
| 函数名 | “abs”、“sqrt”、“max”、“min” |
| 变量名 | {letter}({letter}\|{digit})* |
| 标识符 | “;”、“=”、“{”、“}”、“\n” |
| 通配符 | “.”(匹配所有未被上述规则处理的对象) |

其中类型声明符与函数名这两类符号在规则中有着大致相同的处理流程，因此可以根据待测程序的要求来进行添加和删除。

## 4.1 识别变量

规则中变量名正则表达式部分完成了变量的划分，其对应的 C 代码部分将负责变量表的维护。变量表应包含下列子项：

表 2 变量表包含的子项

| 子项名 | 记号 | 说明 |
|---|---|---|
| 变量名 | NAME | 记录该变量的变量名 |
| 执行体号 | BRACKET | 记录定义该变量的执行体在执行体列表中的序号 |
| 定义域起始句序号 | RANGE_BEGIN | 记录变量定义域的起始句序号 |
| 定义域终止句序号 | RANGE_END | 记录变量定义域的终止句序号 |

上述各项填写时机并不相同。执行体号与定义域起始句序号在声明该变量时填写，即在将新变量加入变量表时，将其执行体号填写为当前句所在执行体号，定义域起始句序号填写为当前句在语句表中的序号；而定义域终止句序号将在执行体结束的时候填写，即填写为执行体的最后一条语句序号。

简要来说，当词法分析模块扫描到一个变量名(identifier) 时的处理流程如图 5 所示：

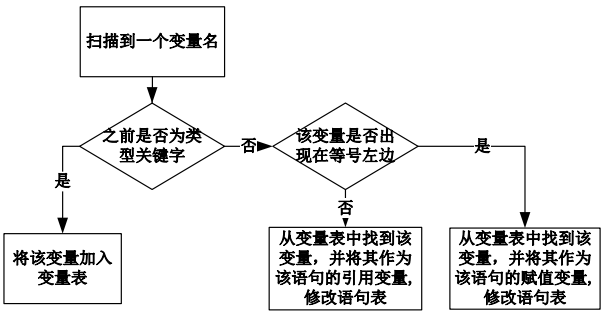


图 5 变量识别流程图

## 4.2 识别语句

在语句的划分上，普通语句以“;”作为结尾；if 语句、for 语句以最外层的“)”作为结尾；“else”不认为是一条语句，仅当做识别基本块的标志。语句表以语句为粒度，一条语句占据一项，按代码中出现的顺序在表中排列。对于语句 S，其在语句表中包括下列子项：

表 3 语句表中包含的子项

| 子项名 | 记号 | 说明 |
|---|---|---|
| 基本块号 | *BELONG* | 记录该语句属于的基本块块号 |
| 语句类型 | *TYPE* | 记录语句的类型 |
| 前驱语句列表 | *STATE_PRED* | 记录该语句的前驱语句序号 |
| 后继语句列表 | *STATE_SUCC* | 记录该语句的后继语句序号 |
| 赋值变量列表 | *DEFINE* | 记录该语句赋值的变量（变量表中的序号） |
| 引用变量列表 | *USE* | 记录该语句使用的变量（变量表中的序号） |
| 输入差异列表 | *IN(S)* | 程序运行到语句 S 之前的程序点时的差异数据集合 |
| 输出差异列表 | $OUT_{IN(S)}(S)$ | 程序运行到语句 S 之后的程序点时的差异数据集合 |
| 差异生成 | $GEN_{IN(S)}(S)$ | 运行语句 S 将新生成的差异数据集 |

| 列表 | | 合 |
|---|---|---|
| 差异消除列表 | $KILL_{IN(S)}(S)$ | 运行语句 S 将消除的差异数据集合 |

填写基本块号、语句类型与前驱后继关系的时机通常是扫描到该句第一个语法单位后。此时在语句表中新增一项，代表识别到一条新的语句。同时填写新语句的前驱关系，并修改其前驱语句的后继列表。确定前驱后继逻辑关系的规则如下:

一条语句如果是某个基本块的第一条语句，那么它的前驱为该基本块所有前驱基本块的最后一条语句。否则，其前驱就为上一条语句。

### 4.3 识别基本块

基本块的划分遵循下列两个基本原则：

**划分原则一**：当程序出现分支语句或将进行跳转时，需要划分一个新的基本块。

**划分原则二**：当程序出现被跳转的语句时，需要划分一个新的基本块。

基本块表以基本块为粒度，按代码中出现的顺序在表中排列。对于一个基本块 B，其在基本块表中包含下列子项：

表 4 第一类定序关系示意图

| 子项名 | 记号 | 说明 |
|---|---|---|
| 包含的语句列表 | STATES | 记录该基本块包含的语句（语句表中的序号） |
| 分支标记 | IS_BRANCH | 标记该基本块是不是分支基本块 |
| 前驱基本块列表 | BNODE_PRED | 记录该基本块的前驱基本块序号 |
| 后继基本块列表 | BNODE_SUCC | 记录该基本块的后继基本块序号 |
| 后向支配结点列表 | PDOM(B) | 见差异传播模型 |
| 严格后向支配结点列表 | SPDOM(B) | 见差异传播模型 |
| 直接后向支配结点列表 | IPDOM(B) | 见差异传播模型 |

基本块表的前四项子项的填写时机为扫描到旧基本块的末尾。当扫描待测程序，根据划分规则应当划分出一个新的基本块时，则将该基本块作为新项加入基本块表，并确定其前驱关系。基本块的定序比语句的定序复杂。首先，约定包含 if(else)、for 的基本块为分支基本块，其分支的目的称为分支目的基本块；记大括号内的程序段为执行体，把即将跳出执行体的基本块称为出口结点。那么，基本块的定序关系可以分为两大类：

**第一类定序关系**：分支基本块与其分支目的基本块的前驱后继关系。

**第二类定序关系**：出口结点与其跳转目的基本块的前驱后继关系。

在第一类定序关系中，分支基本块与分支目的基本块都是固定的，如表 5 所示

表 5 第一类定序关系示意图

| 语句类型 | 分支基本块（前驱） | 分支目的基本块（后继） |
|---|---|---|
| if(…){A} | if(…)所属基本块 | 执行体 A 内第一个基本块；执行体 A 后第一个基本块 |
| if(…){A}else{B} | if(…)所属基本块 | 执行体 A 和执行体 B 内的第一个基本块 |
| for(…){A} | for(…)所属基本块 | 执行体 A 内第一个基本块；执行体 A 后第一个基本块 |

而在第二类定序关系中，出口结点是一个继承的概念。当出现循环嵌套，且内层执行体出现在外层执行体的出口位置上时，外层的执行体要继承内层执行体的出口结点列表。

表 6 第二类定序关系示意表

| 语句类型 | 出口结点（前驱） | 跳转目的基本块（后继） |
|---|---|---|
| if(…){A} | 执行体 A 的出口结点 | 执行体 A 后第一个基本块 |
| if(…){A}else{B} | 执行体 A 的出口结点；<br>执行体 B 的出口节点 | 执行体 B 后第一个基本块 |
| for(…){A} | 执行体 A 的出口结点 | for(…)所属基本块 |
| for(…break;) | break 语句所属基本块 | for 执行体后第一个基本块 |

## 5 继生差异求解模块

### 5.1 数据流生差异求解模块

本模块的目的是求解通过表达式传播的数据差异，它接收语句表为输入，以语句为基本处理单元，修改语句表中差异列表信息。

**算法 1**. 数据流生差异求解算法.

输入：语句表 **struct** STATEMENT st[]，原生差异列表

输出：语句表中各语句的差异列表子项 $OUT_{IN(S)}(S)$

Step1：构建一个空队列 queue<**int**> q；

Step2：将待测程序的入口语句，即 st[0]加入队列；

Step3：从队列头取出一条语句 S，根据数据流方程计算其输出差异数据，然后将其后继语句加入队列尾；

Step4：若队列非空，则转 Step3 继续执行，否则算法执行完毕

由于待测程序之中有可能出现 for 语句，而 for 执行体中出口结点的最后一条语句都是要往前（for 语句）无条件跳转的。因此就将出现要加入队列尾的语句在当前处理语句之前，之后造成数据流生差异求解算法循环处理的情况。这种情况我们称之为回跳。为了使差异求解算法在有限步内终止，for 执行体内出口结点语句的回跳次数不应超过执行体内赋值语句的数目。

本算法采用的数据结构如表 7 所示：

表 7 算法 1 数据结构表

| 变量名 | 含义 | 变量名 | 含义 |
|---|---|---|---|
| queue<**int**> q | 工作集 worklist | **int** b | worklist 头语句 |
| **struct** STATEMENT st[] | 语句表 | **int** ncount | 执行体内赋值语句数 |
| **int** st_num | 语句总数 | **int** st[i].count4_1 | 第 i 条语句的回跳计数 |

算法的伪代码实现如下:

***算法代码***

```
    Input: 词法分析模块输出的语句表
    Output: 程序中所有语句的差异信息
    /* IN(S) 是语句 S 入口的差异信息 */
    /* OUT(S) 是语句 S 出口的差异信息 */
1   queue<int> q;          //声明工作集 worklist
2   for (语句表中的所有语句)
3       st[i].out_num=0; /* 初始化所有语句的出口差异信息 */
4   st[0].in_num=0;    /*S0 是控制流的入口语句，IN(S0) ← ∅; */
5   q.push(0);          // worklist←{S0};
6   while (q.empty()!=1) // 如果队列非空
7   {
8           int b;
9           b=q.front();
10          q.pop();     //从 worklist 中取出队列头语句 S
11          if (b!=0)
12          {   // 如果 S ≠ Sentry ， IN(S) ← ⋃ OUT(Sp)
                                              Sp∈Pred(S)
13              for (对语句 S 的所有前驱语句)
14                计算这些前驱语句输出差异的并集;
15          }
16  if(st[b].type==1||st[b].type==4|| st[b].type==6||st[b].type==7)
17        {//变量声明语句、if 语句、break、continue 语句的差
18        //异生成和消除集合为空
19              st[b].gen_num=0;
20              st[b].kill_num=0;
21          }
22          else if (st[b].type==2) //如果是变量赋值语句
23          {
24              //计算 GEN(S) 和 KILL(S);
25              if (st[b].gen_num==st[b].define_num)
26                  st[b].kill_num=0;
27              else
28              {   //查看该语句是否使用了输入差异数据
29                  int flag=0;
30                  for (int i=0;i<st[b].use_num;i++)
31                  {
32                    进一步查找输入差异集合，是否有相同数据;
33                  }
34                  if (flag==1) //该语句使用变量里包含差异数据
35                  {
36                    将赋值变量列表拷贝至差异生成列表;
37                  }
38                  else  //该语句使用变量里不包含差异数据
39                  {
40                    将赋值变量列表拷贝至差异消除列表;
41                  }
42              }
43          }
44        // NEW_OUT(S) ← GEN(S) ∪(IN(S) —KILL(S));
45          int new_out[100];
46          int new_outn=0;
47          int temp[100];
48          int tn=0;
49          从语句的输入差异集合中减去语句的消除差异集合;
50          上一步所得集合与语句的生成集合合并;
51      // 如果 NEW_OUT(S) ≠ OUT(S) ,则 OUT(S) ← NEW_OUT(S);
52          if (equal(new_out,new_outn,st[b].outs,st[b].out_num)==0)
53              copy(new_out,new_outn,st[b].outs,st[b].out_num);
54        //添加当前处理语句的后继进入工作集 worklist
55          for (int i=0;i<st[b].succ_num;i++)
56          {
57              if (st[b].succ[i]<b) //如果属于回跳
58              {
59                  st[b].count4_1++; //回跳计数自增 1
60                  //寻找该执行体内有多少赋值语句
61                  int ncount=0;
62                  //找到起始语句
63                  int m=st[b].succ[i];
64                 //找到终止语句，即下一基本块的头语句
65                  int end=bac[bac[st[m].belong].succ[1]].states[0];
66                  for (int k=m+1;k<end;k++)
67                  {
68                      if (st[k].type==2)
69                          ncount++;
70                  }
71                      if (st[b].count4_1>ncount)
72                          continue;   //如果回跳计数超过了限制，
73                                      //则不再加入队列
74              }
75             // worklist ← worklist ∪{Ssucc};
76                  int flag=0;
77                  for( int j=0;j<q.size();j++)
78                  {
79                          int tmp=q.front();
80                          q.pop();
81                          if(tmp==st[b].succ[i])
82                          {
83                                  flag=1;
84                                  q.push(tmp);
85                          }
86                          else
87                                  q.push(tmp);
88                  }
89                  if (flag==0)
90                    q.push(st[b].succ[i]); //若队列中未出现该
91                                           //语句，则将其加入队列
92          }
93      }
94      for (int i=0;i<st_num;i++)
95          st[i].count4_1=0; //算法执行完毕后，将各语句的回跳计
96                            //数清空，以便下次调用
```

## 5.2 控制流生差异求解模块

### 5.2.1 后向支配结点求解算法

后向支配结点求解算法用于求解分支基本块的后向支配结点。它接收基本块表为输入，输出为填写基本块表中后向支配结点列表子项。

**算法 2**. 后向支配结点求解算法.

输入：基本块表 **struct** BASIC bac[]

输出：基本块表中后向支配结点列表子项 *PDOM(B)*

预操作：基本块表中各基本块的前驱后继关系都已正确填写

Step1：构建一个空队列 queue<**int**> q；

Step2：将待测程序的出口基本块，即 bac[end]加入队列；

Step3：从队列头取出一条基本块 B，根据数据流方程计算其后向支配结点集合，然后将其前驱基本块加入队列尾；

Step4：若队列非空，则转 Step3 继续执行，否则算法执行完毕；

同样，待测程序中有可能出现 for 语句，for 语句作为一个单独的基本块，其前驱包括循环体的出口结点。如果不限制 for 语句加入队列的次数，算法亦将进入一个死循环。因此 for 语句仅加入处理队列一次，在出队的处理中，将其后向支配结点集合置为循环执行体后第一个基本块的后向支配结点集合与 for 语句本身的并集。

在具体实现上此算法与算法 1 类似。不同之处在于数据流方程求解部分应修改为如下代码：

```
1   if (b!=bac_num)
2    {// 如果 B ≠ Bexit，则 OUT^node(B) ← ⋂_{Bs∈Succ(B)} IN^node(Bs)
3       for (int i=0;i<bac[b].succ_num;i++)
4       {
5       对分支基本块 b 的所有后继基本块，求其后向支配结点的并集
6       }
7       // NEW_IN^node(B) ←{B}∪OUT^node(B);
8       int new_in[100];
9       int new_inn=bac[b].out_pdom_n;
10      for (i=0;i<bac[b].out_pdom_n;i++)
11          new_in[i]=bac[b].out_pdom[i];
12        insert(new_in,new_inn,b);
13      // 如果 NEW_IN^node(B) ≠ IN^node(B) ,
14      // 则 IN^node(B) ←NEW_IN^node(B);
15     if (equal(new_in,new_inn,bac[b].pdom,bac[b].pdom_num)==0)
16          copy(new_in,new_inn,bac[b].pdom,bac[b].pdom_num);
17      //如果是 for 语句，则按 for 语句处理规则处理
18      if (st[bac[b].states[0]].type==5)
19      {
20       将 for 语句的后向支配结点集合置为循环执行体后第一个基本块的后向支配结点集合与 for 语句本身的并集;
21          bac[b].count4_2++;//for 语句处理计数加 1
22     }
```

作为逆向工作集算法，应将当前处理的基本块的前驱基本块加入工作集队列。即将算法 1 第 73 行以后的添加工作集部分修改为如下代码：

```
1     for (i=0;i<bac[b].pred_num;i++)
2        {
3           //for 语句只在队列中处理一次
4            if (bac[bac[b].pred[i]].count4_2>0)
5                    continue;
6           // worklist ← worklist ∪{Bp};
7            int flag=0;
8            for( int j=0;j<s.size();j++)
9            {
10                int tmp=s.front();
11                s.pop();
12                if(tmp==bac[b].pred[i])
13                {
14                    flag=1;
15                    s.push(tmp);
16                }
17                else
18                    s.push(tmp);
19           }
20           if (flag==0)
21               s.push(bac[b].pred[i]); //若队列中未出现该基
22                                       //本块，则将其加入队列
23       }
```

5.2.2 直接后向支配结点求解算法

该算法从分支基本块的后向支配结点列表中找出直接后向支配结点。

对一个分支基本块 B，此算法首先从 B 的后向支配结点集合中删去自己，再任意取出两个结点，考察这两个结点的相互后向支配关系，保留其中被后向支配的结点，删去另外一个，直至集合中只剩下一个结点为止。该结点即为直接后向支配结点。

5.2.3 控制流生差异求解算法

此算法同样不涉及数据流方程求解，它通过宽度优先搜索的手段，找出受分支差异影响的基本块。并将这些基本块中所有被定值的数据都标记成差异数据。

## 5.3 继生差异综合求解算法

继生差异综合求解算法构建了继生差异求解模块的整体框架。其具体步骤如下所示：

**算法 3**. 继生差异综合求解算法.

输入：程序的基本块表 **struct** BASIC bac[]

输出：每个程序点的差异信息

预操作：直接后向支配结点算法已被顺利执行

Step1：调用后向支配结点求解算法求解各基本块的后向支配结点集合;

Step2：调用直接后向支配结点求解算法求解各基本块的直接后向支配结点集合;

Step3：调用数据流生差异求解算法求解各条语句的差异信息;

Step4：

**for** 基本块 B 是基本块表中的分支结点 **do**

**if** *B* 存在分支差异 **and** B 没有被标记为 have_visited **then**

调用控制流生差异求解算法求解分支差异影响的基本块，标记出控制流生差异;

将 B 标记为 have_visited;

将 B 中产生分支的语句加入数据流生差异求解算法的队列 q;

以 q 为队列调用数据流生差异求解算法，求解新产生的数据流生差异;

**end if**

**end for**

Step5： 若循环产生了新的差异数据，则转 Step4，否则算法执行结束

# 6 实验

## 6.1 实验准备

开展实验所处的环境如表 8 所示，

表 8 实验环境配置

| | | |
|---|---|---|
| 开发环境 | flex-2.5.4a-1 for windows+ Microsoft Visual Studio 6.0+ Java SDK 1.6 + Eclipse3.0 | |
| 硬件配置 | CPU | Inter Pentium Dual CPU 1.73GHz 1MB L2 Cache |
| | 内存 | 2GB |
| 软件平台 | 操作系统 | Microsoft Windows 7 (32 位) |
| | 运行环境 | Java Runtime Environment 1.6 |

实验采用的基准测试程序为 SPEC2006 (Standard Performance Evaluation Corporation) 中 464.h264ref 的 Get_Reference_Pixel 函数。h.264/AVC 是一种崭新的视频压缩标准，由 ITU(International Telecommunications Union 国际电信同盟) 的 VCEG(Video Coding Experts Group 视频编码专家委员会)提出。该标准将代替现在广泛使用的 MPEG-2 标准，在下一代 DVD 和视频传输等应用中普及[11][12]。

Get_Reference_Pixel(imgpel **imY, int y_pos, int x_pos) 函数位于源代码文件 decorder.c 中。decorder.c 包括了实现基于“率失真优化技术”的相关解码函数，其中 Get_Reference_Pixel 用来找到未采样参照系中的像素(y,x)[13]。

## 6.2 测试指标

在对基准测试程序进行实验之前，我们首先提出几个重要的评测指标，以反映差异传播的特性及编译分析工具的性能。值得注意的是，所有的评测指标都是针对某种原生差异输入的情况下确定的。若将原生差异输入情况记为 $A$，那么所有的指标函数都应该包含参数 $A$，为书写的方便，下文的讨论中均将参数 $A$ 省略。

1) 相似数据比例 $P(t)$

其中，参数 $t$ 代表程序的第 $t$ 条语句，$P(t)$代表在该程序点的相似数据占全体变量集合 $S$ 的比例。

2) 相似数据变化率 $R(t_1,t_2)$

对程序中的两条语句 $t_1$，$t_2$，如果 $t_2$ 所属的基本块是 $t_1$ 所属基本块的后向支配结点（即语句 $t_1$ 后有一条到语句 $t_2$ 的程序执行路径），我们称语句 $t_2$ 为语句 $t_1$ 的执行后继，那么记 $R(t_1,t_2)$为这两条语句的相似数据变化率。其计算公式如下：

$$R(t_1,t_2)=\frac{S\times(P(t_2)-P(t_1))}{[t_2,t_1]}$$

其中， $S$ 表示全体变量集合，$[t_2,t_1]$表示从语句 $t_1$ 到语句 $t_2$ 的程序执行路径上的语句条数。该指标主要用来衡量整个程序中差异数据传播的速度。

3) 原生差异数据的传播能力 $D(x,t)$

在程序第 $t$ 条语句声明的原生差异 $x$ 可以通过差异传播，在 $t$ 语句的后继语句中产生继生差异。$D(x,t)$即表示在整个待测基准程序中，由 $x$ 直接或间接决定的差异数据的总数，用来衡量一个原生差异在程序中的影响因子。其具体的计算公式为：

$$D(x,t)=\sum_{i=t+1}^{n}D_i$$

其中 $D_i$ 代表待测程序第 $i$ 条语句的差异信息集合中由原生差异 $x$ 直接或间接决定的差异数据数目。$n$ 代表待测程序的最后一条语句序号。

4) 继生差异求解模块循环次数 C

继生差异求解算法由若干个循环构成，每一个循环包含一个数据流生差异求解算法和控制流生差异求解算法的相互调用。该指标用来衡量这种相互调用的次数。

5) 编译分析工具执行时间 T

编译分析工具执行的时间从输入基准测试程序后开始计时，经过词法分析模块，继生差异求解模块的处理，到预备输出结果为止，使用 gettimeofday 函数计时[14]。不包括工具输入输出时间。

## 6.3 实验方案

### 6.3.1 输出词法分析结果

首先输出词法分析模块的结果，并对其进行可视化。词法分析模块在测试基准程序中识别出 19 个变量，92 条语句与 51 个基本块。其识别的语句类型如表 9 所示：

表 9 语句类型统计表

| 语句类型 | 条数 |
|---|---|
| 变量声明语句 | 19 |
| 变量赋值语句 | 51 |
| if 语句 | 12 |
| for 语句 | 10 |
| continue 语句 | 0 |

| | |
|---|---|
| break 语句 | 0 |

这些语句均有特定的前驱后继关系，我们对这些语句进行可视化。

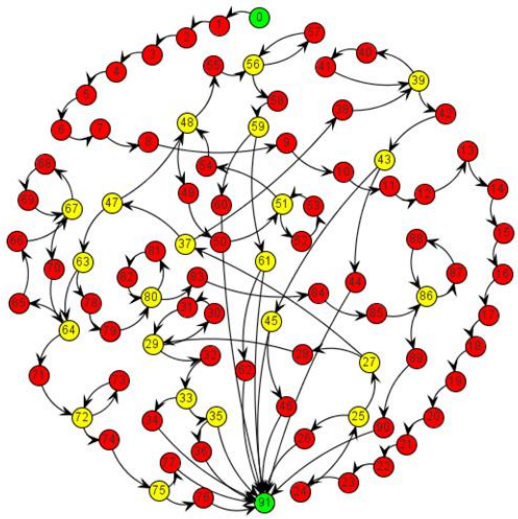

图 6 语句流程图

图 6 为语句流程图，22 个浅色顶点分别代表 22 条分支语句（12 条 if 语句，10 条 for 语句）。每条分支语句有两条出边，对应两条分支目的语句。

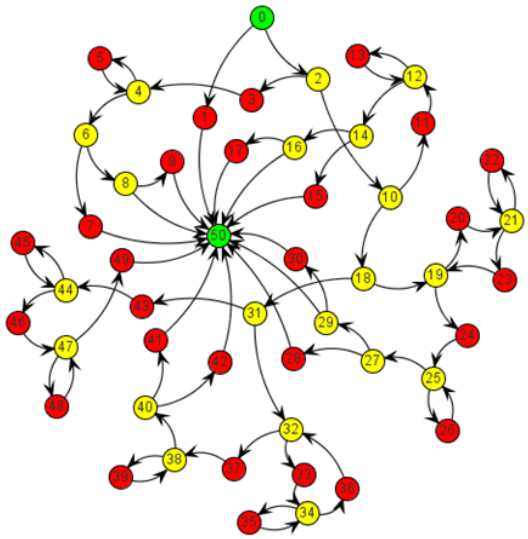

图 7 基本块流程图

图 7 为基本块流程图，在识别语句的基础上，词法分析模块共识别出 51 个基本块，其中 21 个为分支基本块，在图 7 中用浅色顶点表示。

### 6.3.2 选取考察程序点

测试根据不同的原生差异输入将会有不同的结果，我们将不同的原生差异输入称之为不同的测试情形。我们约定情形一中函数的宽、高(width，height)这两个变量为原生差异，情形二中函数的高(height)变量为原生差异。

为了保证程序考察的连续性，以及满足指标 2、3 的计算条件。选取的程序点序列中满足后一个程序点为前一个程序点第一个相似信息发生变化的执行后继。故相邻的两个程序点相似信息互不相同。

表 10 选取程序点一览表

| 程序点编号 | 语句序号 | 所属基本块号 | 语句内容 |
|---|---|---|---|
| 1 | 17 | 0 | "int max_imgpel_value;" |
| 2 | 25 | 0 | "maxold_x = width-1;" |
| 3 | 26 | 0 | "maxold_y = height-1;" |
| 4 | 81 | 43 | "pres_y = max(0,min(maxold_y,pres_y));" |
| 5 | 82 | 44 | "for(x=-2;x<4;x++)" |
| 6 | 85 | 46 | "result1 = max(0, min(max_imgpel_value, (result+16)/32));" |
| 7 | 91 | 49 | "result2 = max(0, min(max_imgpel_value, (result+16)/32));" |
| 8 | 93 | 50 | "**return** result;" |

### 6.3.3 指标分析

由于情形二的原生差异输入集合为情形一的原生差异输入集合的子集，导致后者所有程序点的相似信息集合均为前者对应集合的子集。这说明，差异传播模型具有传递性质。

图 8 为情形一与情形二的相似数据比例图。这两个情形的相似数据比例随着程序的执行逐渐降低，但一直保持着一个较大的比例（≥37%）。这说明对相似数据进行优化是可行的，且具有较大的优化空间。

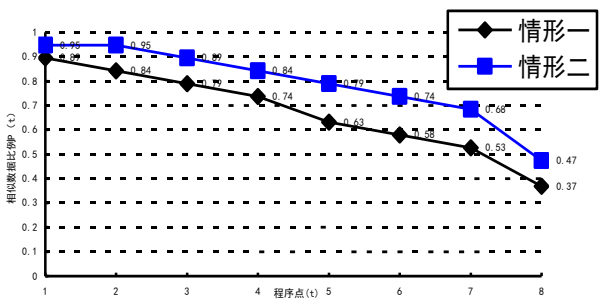


图 8 相似数据比例图

根据相似数据变化与相邻程序点间隔的语句

条数，我们还能够计算出相邻程序点的相似数据变化率。由图 9 可知相似数据的变化率并不稳定。这也就是说程序中差异数据的传播速度不一，其传播速度取决于原生差异的差异传播能力。

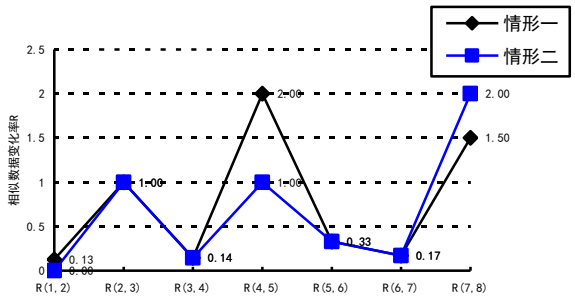


图 9 相似数据变化率图

指标 3 衡量了某个原生差异数据的传播能力。我们依据其计算公式分别考察单个原生差异的传播能力

表 11 单个原生差异的传播能力示意表

| 原生差异变量名 x | 原生差异声明的位置 t | 其传播产生的差异数据总数 D(x,t) |
|---|---|---|
| height | 16 | 347 |
| COEF | 16 | 235 |
| max_imgpel_value | 16 | 110 |
| pres_x | 16 | 66 |
| height | 25 | 68 |
| pres_x | 32 | 8 |

其中传播产生的差异变量指在程序中各点的差异信息中出现过的变量。而差异数据总数指这些变量出现的次数之和。通过研究指标 3 我们发现，不仅在同一位置声明的不同原生差异变量的传播能力有差异，声明位置不同的同一原生变量，其传播能力也将有所区别。

其中传播产生的差异变量指在程序中各点的差异信息中出现过的变量。而差异数据总数指这些变量出现的次数之和。通过研究指标 3 我们发现，不仅在同一位置声明的不同原生差异变量的传播能力有差异，声明位置不同的同一原生变量，其传播能力也将有所区别。

计算在不同情形下程序的执行时间(精确到 us)，得出结果如下：

表 12 编译分析工具性能分析表

| | 词法分析模块 | 继生差异求解模块 | 继生差异模块中循环次数 |
|---|---|---|---|
| 情形一： | <1 us | 16070 us | 2 |
| 情形二： | <1 us | 15635 us | 2 |

通过对指标 4、5 的分析，我们发现词法分析模块耗时较少，工具总执行耗时均在 15ms 左右。因此本文所构建的编译分析工具的工作时间开销是可以接受的。

## 7 结束语

本文在相似性分析的理论基础上，设计和实现了一个数据访问相似性编译分析工具。首先本文依托 FLEX 构建了词法分析模块，完成了识别变量、识别语句、识别基本块等任务，之后本文在差异传播模型的相关理论基础上，通过书写 FLEX 中用户附加 C 语言部分，实现了数据流生差异求解算法、后向支配结点求解算法、直接后向支配结点求解算法与控制流生差异求解算法。并对算法中出现的回跳现象与针对 for 语句的处理规则进行了研究。词法分析模块与继生差异求解模块相配合，共同完成了识别差异数据与相似数据的目的。

最后，本文选取基准测试程序验证了工具的正确性与有效性。在下一步的工作中，应着手对找到的相似数据进行共享优化。减少共享 Cache 和共享主存中的数据占用量，最终提高运行多个串行程序副本时的系统吞吐率。

## 参 考 文 献

[1] John L. Hennessy, David A. Patterson. Computer architecture (4th ed.): a quan-titative approach[M]. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc, 2007.

[2] Win. A. Wulf. Hitting the Memory Wall: Implications of the Obvious[M]. Chicago, USA: University of Chicago Press, 1994.

[3] 吴俊杰. 层次存储的访问分析与优化方法研究——重用性、相似性与亲和性[C].长沙, 中国: 国防科学技术大学博士毕业论文, 2009.

[4] Uresh Vahalia. UNIX 系统内幕(英文版)[M]. 北京, 中国: 人民邮电出版社, 2003.

[5] Jonathan M. Smith, Gerald Q. Maguire. Effects of

copy-on-write memory management on the response time of UNIX fork operations. Computing Systems[M]. USA: Computing & Electronic Books, 1988.

[6] Daniel G. Bobrow, Jerry D. Burchfiel, Daniel L. Murphy, Raymond S. Tomlinson. TENEX, a paged time sharing system for the PDP - 10[J]. Commun ACM. 1972, 15(3): 135~143.

[7] E. Zayas. Attacking the process migration bottleneck[J]. SIGOPS Oper Syst Rev.1987, 21(5): 13~24.

[8] Carl A. Waldspurger. Memory resource management in VMware ESX server[J]. SIGOPS Oper Syst Rev. 2002, 36(SI): 181~194.

[9] Susmit Biswas, Diana Franklin, Alan Savage, Ryan Dixon, Timothy Sherwood, Frederic T. Chong. Multi-execution: multicore caching for data-similar executions. ISCA'09: Proceedings of the 36th annual international symposium on Computer architecture[C]. New York, NY, USA: ACM, 2009: 164~173.

[10] M.E.Lesk, E.Schmidt. Lex – A Lexical Analyzer Generator. (DB/OL). http://dinosaur.compilertools.net/lex/, 1989[2010-01-17].

[11] Wiegand, T. Sullivan, Bjontegaard, Luthra. Overview of the H.264/AVC video coding standard[J]. Circuits and Systems for Video Technology, 2003, First Quarter 27~41.

[12] Ostermann.T, Bormans.S, List.B. Video coding with H.264/AVC: tools, performance, and complexity[J]. Circuits and Systems Magazine. 2004, First Quarter 7~28

[13] Schwarz.H, Marpe.D, Overview of the Scalable Video Coding Extension of the H.264/AVC Standard[J], Circuits and Systems for Video Technology, Sept. 2007, 1103 - 1120

[14] The IEEE and The Open Group. The Open Group Base Specifications Issue 6. (DB/OL).http://pubs.opengroup.org/onlinepubs/009695399/functions/gettimeofday.html, 2004[2010-01-20].