IRDiff：基于 LLVM 中间表示的代码差异分析

胡文颖，梁洪亮
（北京邮电大学计算机学院，北京，100876）

摘要：代码差异分析是多版本程序分析领域中重要的研究问题之一。已有的工作，包括基于代码行的比较或基于抽象语法树的代码分析，往往会受到代码移动的影响，使得对比过程中，由于对比代码的错位而导致差异分析的不精确。本文提出一种基于 LLVM 中间表示语言的源代码差异分析算法。该算法首先将待比较程序的两个版本的源代码转换成 LLVM 中间表示语言，利用 LLVM 中间表示的语法结构、控制流以及数据流信息，在程序两个版本的控制流图上寻找同构节点，得到一系列同构子图以及差异节点集合，从而得到程序两个版本间的代码差异。利用中间表示语言的层次化结构特性以及控制流信息，能够避免因代码移动导致的对比代码错位问题，并且结合控制依赖、数据依赖以及类型信息，可以提高代码差异分析的准确性。基于该算法，我们实现了一个代码差异分析工具 IRDiff，并使用 SIR 程序集进行实验评估，实验表明该工具可以较好地分析 C/C++程序不同版本之间的代码差异。

关键词：代码演化分析；源代码差异；控制流图；中间表示；程序理解
中图分类号：TP311.5

IRDiff：Source Code Differencing based on LLVM IR

Hu Wenying, Liang Hongliang
(*Computer School, Beijing University of Posts and Telecommunications, Beijing* 100876)

Abstract: Source Code differencing is one of the most important research issues in multi-version program analysis. Existing work, including text differencing or abstract syntax tree-based code analysis, is often affected by code movement, making the comparison inaccurate due to the misalignment of the code. This paper proposed a new approach of source code difference analysis based on LLVM intermediate representation language (LLVM IR). The two versions of the source code are as input transformed to the LLVM IR. Then based on the syntax structure, control flow and data flow information represented by LLVM IR to find the isomorphic nodes in the two versions of the control flow graph. A series of isomorphic sub-graphs and a set of difference nodes are obtained, and the code difference between two versions of the program is obtained. By using LLVM IR's hierarchical structure and control flow information, the problem of code misalignment caused by code movement can be avoided, and the accuracy of code variance analysis can be improved by combining control dependency, data dependence and type information, etc. Based on this algorithm, we implemented a code variance analysis tool, called IRDiff, and used the SIR benchmark for experimental evaluation. Experiments show that the tool can better analyze the code differences between different versions of C / C ++ programs.

Keywords: source evolution analysis; control flow graph; program comprehension; graph differencing; intermediate

0 引言

软件演化研究领域重点关注如何更好地了解软件系统发展的问题，包括对软件的外部环境、执行需求演化的理解以及对软件源代码改进的理解，在本文中，我们将侧重于对后者的研究。在传统的源代码改进分析中，通常利用代码行或抽象语法树(AST)进行匹配[1,2,3,4]，结合推导编辑脚本的程序分析两个代码之间的差异。本文工作首先将 C/C++的源代码转换成

作者简介：胡文颖（1993-），女，研究生，主要研究方向：可信软件，智能系统
通信联系人：梁洪亮（1972-），男，副教授、硕导，主要研究方向：可信系统，智能软件. E-mail: hliang@bupt.edu.cn

LLVM 中间表示(IR)，利用 LLVM 提供的控制流图(CFG)和数据流信息，提出一种基于 LLVM IR 的细粒度代码演化分析算法。LLVM 中间表示程序提供一系列包含控制流和数据流信息的函数层信息。利用这些信息，我们可将整个程序划分成多个函数，将整个程序的代码差异分析问题分解成函数层的代码差异分析问题，新版本程序中新增或删除的函数可以直接判定为差异代码。利用函数名信息匹配新旧版本程序内的函数，保证分析对象在语法层上是等价的，减少因代码文本移动对代码演化分析的影响。

本文提出的代码演化分析算法主要包含两个部分。第一部分是基于 LLVM 中间表示语言语法的差异判定准则。第二部分是控制流图的同构子图搜索算法，该算法分两个阶段进行。第一阶段从程序两个版本的函数入口开始，沿着控制流图寻找高度递减的最大同构子图。第二阶段利用第一阶段的输出，将两个版本的控制流图划分为多个对应的独立子图，独立子图的边界均为第一阶段中所得的同构节点，子图内部均为第一阶段未找到同构节点的节点，在独立子图内借助控制依赖找寻更多的同构子图。

本文的主要贡献如下：

1） 首次提出了一个基于 LLVM 中间表示语言的代码差异分析方法。该方法使用基于 LLVM IR 语法的差异判定准则与同构子图搜索算法，实现了细粒度的代码差异分析。

2） 基于上述方法，实现了一个分析工具 IRDiff，能够分析 C/C++程序的不同版本间的代码差异。以 SIR 基准测试集中的 4 个 C 程序(共 8 组版本)为实验对象，对 IRDiff 进行了实验评估。实验结果表明 IRDiff 能够得到精确的代码差异分析结果。

1 LLVM IR 差异判定准则

1.1 LLVM 中间表示语言

LLVM 中间表示(IR)是一种基于静态单赋值形式的程序中间表示语言，能够为多种高级语言提供类型安全且灵活的表示。该中间表示是 LLVM 各个编译策略阶段通用的代码表示语言，包含丰富的代码信息。LLVM IR 程序由模块(Module)组成，每个模块都是输入程序的一个翻译单元。模块的结构如图 1 所示，每个模块包含一系列函数(Function)、全局变量以及符号表入口，并且可能与 LLVM 链接器组合完成合并函数和全局变量的定义、处理前置声明、合并符号表入口等处理。模块中的一个函数定义包含一个基本块(BasicBlock)列表，该列表中的基本块组成该函数的控制流图。一个基本块包含一个代表该基本块的符号表入口的独有标签以及一个指令(Instruction)列表。每个指令列表的末尾指令称为终止指令，表示一个基本块的出口，末尾指令包括返回指令、分支指令等等。

1.2 差异判定准则

针对 LLVM IR 的层次化结构，本文对各个层次的结构提出不同的差异判定准则，用于代码演化的分析。总的判定流程可归纳为如下几个步骤：

1. 输入一个原始版本程序对应的 IR 模块(M)和一个修改后的比较版本程序对应的 IR 模块(M')
2. 比较 M 和 M'中的函数，通过 IR 模块层提供的函数列表，将两个模块中包含的独有函数包含的基本块集分别添加到集合 U 和 U'中，将两个模块中匹配的函数添加到映射表 F 中。
3. 对于映射表 F 中的每个匹配对(f,f')：

a) 执行第一阶段的同构子图搜索算法，将得到的同构基本块对添加到映射表 Matched 中；

b) 执行第二阶段的同构子图搜索算法，将得到的同构基本块对添加到映射表 Matched 中；

4. 将 f 和 f'中未找到匹配的基本块分别添加到集合 U 和 U'中

5. 最后输出一个表示原始程序独有基本块的集合 U，比较程序独有基本块集合 U'和两个版本程序的匹配代码块映射表 Matched。

依据 LLVM IR 的层次化结构，首先获得模块层的代码差异，区分两个版本中独有的函数和匹配的函数。以基准版本为标准，基准版本中的独有函数即比较版本中删除的函数，而比较版本中的独有函数即比较版本中新增的函数。利用函数名匹配函数的前提是在代码演化中不考虑函数名更新的操作。函数名匹配的函数是后续层次的代码差异分析对象。本文设计一种利用函数控制流图的同构子图搜索算法，进一步分析函数内部的代码差异，找到两个版本之间同名函数间的同构基本块和独有基本块，同构基本块表示两个版本之间的共有代码，以基准版本为标准，基准版本中的独有基本块即比较版本中删除的代码内容，而比较版本中的独有基本块即比较版本中的新增代码内容。在同构子图搜索算法中同时结合了 IR 程序基本块层以及指令层的代码差异分析准则。

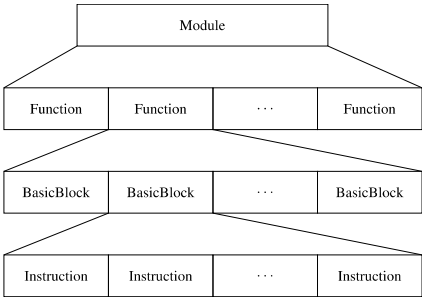

图 1 LLVM IR 基本结构图

Fig. 1 The structure of LLVM IR

1.3 模块层差异

模块层提供当前模块包含的函数定义、全局变量以及常量。

通过匹配函数名，可以获取待比较的两个模块中包含的相同函数以及各自的独有函数。相同函数需进一步执行函数层差异判定，而独有函数则可直接判断为某版本的独有代码，构成模块层代码的差异。

为了得到准确的分析结果,对于全局变量和常量差异的分析推迟到指令层。即，在指令中使用到全局变量和常量时，才对它们进行差异判定。

1.4 函数层差异

LLVM 的函数定义包含函数的函数名，返回值类型，参数表。函数体包含一系列基本块。根据模块层差异的分析结果，对同名的两个函数的函数体进行差异性判定。在 LLVM 程序中，函数包含的一系列基本块根据控制流信息将构成对应的函数层控制流图。受 B. Fluri 提出一种 AST 差异算法[3]的启发，本文提出一种面向控制流图的函数差异判定算法，利用同构子图的搜索策略分析两个函数之间的差异代码，构成函数层代码的差异。

1.5 基本块层差异

基本块中包含一个指令列表。由于包含相同指令的基本块在一个控制流图中可能会出现多次，因此分析两个基本块之间的差异或者基本块之间的等价性需要两个步骤，第一个步骤是判定基本块内部指令集的等价关系。第二个步骤是判定基本块的前驱和后继的等价关系。

1.6 指令层差异

1.6.1 LLVM IR 的类型系统

在展开对指令层代码差异分析之前，我们先简要的描述 LLVM IR 提供的类型系统。LLVM 程序提供丰富的类型信息，包括基础类型，如整数、浮点数、指针等，以及复合类型，如数组、结构体、类等。对于如下的代码片段：

```
public:
    int a;
    int *p;
    struct B *t;
}
struct B{
    char *b;
    float s[5];
}
```

对应地，在 LLVM 的模块中将生成如下复合类型声明：

```
%class.A = type {i32, i32*, %{“struct.B”*} }
%struct.B= type {i8*,[5 x float]}
```

LLVM 中的指令均以寄存器的形式存储，并且都带有相应的类型信息。因此，指令层的差异判定的首要步骤就是判断两个指令的类型是否等价。对于基础类型，可以直接根据 LLVM 提供的类型标识符进行比较；对于复合类型，将进一步地根据复合类型中的元素类型的等价性进行判定。这样分析的好处是，能分析出由于类或结构体声明改动引起的不可见的代码改动。

1.6.2 指令差异

LLVM 共有 31 种指令类型，可分为如下 5 大类。

终止指令：ret/br/switch/…

二元操作符：add/sub/mul/div/and/or/…

内存分配与访问：alloca/load/store/...

类型转换：zext/sext/bitcast/…

其他：cmp/call/phi/…

每种指令都有自己独有的语法，通用的语法结构是由操作符、指令类型以及若干操作数组成，因此通用的指令差异判定准则通过操作符、指令类型以及各个操作数的差异性进行判定。对于指令的操作数和指令类型的差异判定，由于 LLVM 为它们提供唯一的标识，因此我们直接利用这些唯一标识判定两个指令操作符和指令类型的差异性。根据操作数包含的类型、定义以及存储的内存空间等信息，操作数的差异性判定有以下几种判断方式：

对于单操作数的指令，则直接判断指令各自操作数的不等价性，包括类型差异，定义差

异以及存储空间差异等等。这些差异性的判定对于所有的操作数差异性判定均适用；

对于多操作数的指令，每个操作数还有一个索引信息，表示它在指令中的顺序和位置。对于多操作数指令的操作数差异性判定有如下两种不同的处理方式：

对于操作数的顺序改变不会影响运行结果的指令，例如 add/mul/and/or 等指令，在判别过程中忽略操作数所在的位置信息。例如指令:

%1 = i64 add i64 %a, i64 %b （1）

%1 = i64 add i64 %b, i64 %a （2）

在进行差异判定的时候，会将指令（1）的操作数集合 ops1:{%a,%b}中的每个元素和指令（2）的操作数集合 ops2：{%b,%a}中的每个元素分别进行判定，当且仅当 ops1 中有元素与 ops2 中的所有元素都存在差异时，才会判定指令（1）与指令(2)存在差异。

对于操作数的顺序改变会影响运行结果的指令，例如 div/rem 等指令，或是拥有固定含义的指令，例如 getelementptr/store/call/phi 等指令，则必须考虑操作数的位置信息，进行一对一的判定。例如指令：

%2 = i64 div i64 %c, i64 %d (3)

%2 = i64 div i64 %d, i64 %c (4)

在进行差异判定时，若指令（3）中的%c 与指令（4）中的%d，或指令（3）中的%d 与指令（4）中的%c 存在差异，则指令（3）和指令（4）存在差异。

除了上述的通用的指令差异判定准则之外，还有一些特殊的指令需要特殊处理：

1) br 指令

br 指令是一个基本块的终止指令，它包含三个操作数，分别表示条件表达式，真分支的基本块标志以及假分支的基本块标志。在判断 br 指令本身的差异性时，并不考虑后继基本块的差异性，而是根据 br 指令各自的条件表达式的差异性进行判定。若条件表达式无差异，将进一步判断指令各自的真分支跳转基本块与假分支跳转基本块的差异性，若存在差异则直接判定后继基本块的的差异性。但后继基本块的的差异性并不会影响 br 指令本身的差异性。

2) load 指令

load 指令用于从内存中读值，它包含一个操作数，表示指向待访问内存空间的指针。在判断待访问空间的类型以及空间大小的差异性之后，还需要进一步的判定当前内存空间中存放的值是否存在差异。因此在判定 load 指令的差异性时，我们利用 LLVM 提供的数据流图，收集当前内存空间中所有可能存放的值，若可能存放的值存在差异则判定 load 指令存在差异。

3) icmp/fcmp 指令

icmp/fcmp 指令用于表示条件表达式，其中 icmp 用于整数间的比较而 fcmp 用于浮点数间的比较。它们包含三个操作数，第一个操作符表示不同的比较操作符，LLVM 中一共支持 10 种比较操作符的表示，后两个操作数分别表示比较操作符两端的值。在比较两个 cmp 指令的差异性时，除了判断在相同的比较操作符中，左右两端的值各自的差异性之外，还需考虑不同比较操作符可能的无差异性。例如指令：

%3 = icmp ugt i64 %e, %f (5)

%3 = icmp ult i64 %f, %e (6)

表示 (e>f)与(f<e)两个表达式，若指令（5）和（6）中的%e 与%f 无差异，则这两条指令中并不存在差异。

2 函数层同构子图搜索算法

LLVM 程序中的函数由一系列基本块组成，函数的控制流图记录着这些基本块之间的控制依赖关系。面向控制流图的函数差异判定算法的核心是控制流图中的同构子图搜索算法，该算法分为两个阶段进行：

第一阶段，我们使用一种贪婪的自顶向下算法，结合基本块差异判定准则，寻找两个函数各自控制流图中的高度递减的同构子图，将同构子图各个基本块一一对应，存储在映射表中。

第二阶段，利用第一阶段所得的映射表，将控制流图划分成多个一一对应的独立子图，结合基本块差异判定准则，找出子图中包含的同构子图，并动态的更新各个独立子图的边界。

该算法的思想基于如下观察和开发实践：在人工分析代码差异时，先尽可能多的搜索未修改的代码片段。当未修改的代码片段确定之后，我们可以确信利用未修改节点，可将剩余代码划分成几个独立的子片段，两个程序间的子片段两两对应，即程序 P 区域 A 内的节点若在 P’中有同构节点，则一定位于 P’对应于区域 A 的区域 A’中，而不会出现在其他区域内。由此可以缩小搜索区域，实现进一步的差异搜索。

2.1 高度相等的同构子图搜索算法

定义 1，控制流图中基本块的高度 height(bb)：1）对于一个无后继的基本块 bb，height(bb) = 1; 2) 对于其他基本块 bb，height(bb)=1 + max({height(s)| s∈succ(bb)});3）若路径中出现循环节点，循环入口节点高度设为 1.

自顶向下的算法使用了一种高度优先列表，该列表包含一系列依照高度递减排列的节点。与该数据结构相关的操作包括:1)L->push(n)将一个包含基本块以及对应高度信息的节点 n 插入列表 L 中，2)L->peekMax() 返回列表 L 存储的最高基本块高度，3)L->pop() 将从列表 L 中取出表中所有高度等于 peekMax()返回值的节点，4)L->open(n)将节点 n 中包含的基本块的所有后继基本块构成新的列表节点插入列表 L 中。算法细节见算法 1。该算法的目的是找出输入的两个控制流图 cfg1 和 cfg2 中最大高度的同构子图。

具体的算法流程如下描述：该算法从控制流图的入口基本块开始，并判断两个基本块是否同构。如果 cfg1 和 cfg2 的入口基本块不同构，则考虑基本块各自的后继节点。当前的同构基本块判定仅考虑基本块内部的指令是否等价而不考虑基本块的前驱以及后继，所以可能会找到多对同构基本块，因此在当前阶段，若找到一对同构基本块，则将该映射添加到候选映射表中。该表将在后续利用基本块的前驱节点与后继节点的信息进行进一步的同构判定，再将确定且一一对应的同构基本块映射添加到结果映射表中。输入参数 minHeight 指定了当前算法考虑的最低的基本块高度，通常设为 1，即考虑至无后继的基本块。在确定了一对同构基本块之后，通过宽度优先的方式遍历基本块的后继基本块，若后继基本块满足内部指令的等价，则直接将对应的后继映射添加到结果映射表中，并继续判定同构后继的后继节点，直到没有后继或没有同构后继为止。

```
算法 1：自顶向下阶段算法
输入：源控制流图 cfg1 和目标控制流图 cfg2，最低基本块判
断高度 minHeight，两个空的高度优先列表 L1,L2，一个空的
候选映射表 C 和结果映射表 M.
输出：结果映射表 M
1 L1->push(entry(cfg1));
2 L2->push(entry(cfg2));
3 while min(L1->peekMax(), L2->peekMax()) > minHeight do
4       if L1->peekMax() != ls->peekMax() then
5               if L1->peekMax() >L2->peekMax() then
6                       foreach n ∈L1->pop() do L1->open(n)
7               else
8                       foreach n ∈L2->pop() do L2->open(n)
9       else
10              N1 = L1->pop();
11              N2 = L2->pop();
12              foreach (n1, n2) ∈ N1 X N2 do
13                      if isEqualBlock(n1, n2) then
15                              C.insert(n1, n2);
16              foreach n1 ∈N1 | (n1, nx) ∈C do
17                      if(C.getRange(n1)==1)
18                              C.move(n1, C[n1],M);
19                      else
20                              (n1, n2)= isEqualDeps(n1,SN2);
21                              C.move(n1,n2,M);
22                              M.add(isEqualBlock(ns1,ns2)|ns1∈
desc(ns1), ns2∈desc(n2));
23      foreach n1 ∈N1 | (n1, nx) □ C∪M do L1->open(n1);
24      foreach n2 ∈N2 | (nx, n2) □ C∪M do L2->open(n2);
```

图 2　自顶向下阶段算法

Fig. 2　The Algorithm of the first phase

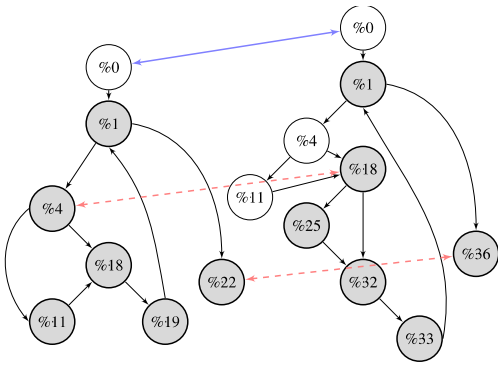

图 3　close_files 函数在 sed(v1,v2)中的控制流图

Fig. 3　the CFG of sed(v1,v2)/close_files()

图 3 展示了 SIR 程序集中 sed 程序 v1 和 v2 版本中 close_files 函数的控制流图，左下为 v1 版本的控制流图，右上对应于 v2 版本。图中的每个节点表示一个基本块，节点中的“%n”对应于中间表示中的基本块标识。图 3 中用红色短线连接起来的节点{(%4,%18),(%22,%36)}表示中第一阶段找到的高度相等的同构基本块。当找到如(%4,%18)这样一对同构基本块之后，将进一步分析%4 的后代基本块（直接后继与间接后继）与%18 的后代的映射关系，得到最大高度的同构子图。最终，第一阶段将输出如下映射表

{(%4,%18),(%11,%25),(%18，%32),(%19,%33),(%1,%1), (%22,%36)}。在图中为灰色底表示的节点。这些节点组成了 close_files 函数在 v1 版本与 v2 版本之间高度相等的最大同构子图。

2.2 子图分割与图内基本块匹配的算法

经过第一阶段的分析之后，整个控制流图被已映射的节点划分为几个独立的子图。根据控制流图的特性可以确定，同构基本块一定位于两个对应的独立子图中，而不会跨越子图。算法 2 详细描述子图的初始边界生成算法以及自底向上的同构子图匹配算法。

当前阶段的输入是两个待比较的控制流图 cfg1 和 cfg2 以及第一阶段的输出结果 M。根据映射表 M，getRange()可以得到 cfg1 的初始独立子图集,该操作的算法细节详见图 5，其中组成边界的基本块均为映射表 M 中的同构基本块对，子图内部的节点为起始节点的后继节点以及后继的后继等，且为终止边界节点的前驱节点以及前驱的前驱，且均未有任何匹配的基本块。因此，利用映射表 M 以及 getRange()所得的子图集，getMatchedRange()可以得到 cfg2 中对应的子图集，该操作的算法细节详见图 6。然后利用宽度优先遍历的策略，从起始节点开始继续自顶向下地遍历 cfg1 中的节点，与 cfg2 中对应的子图区域中的节点进行比较。若找到一对内部指令等价，前驱与后继均同构的基本块对，则直接添加到结果映射表中，并将 cfg2 对应的子图区域的起始边界更新为匹配基本块的兄弟节点与后继节点组成的新边界。cfg1 中的后续访问节点的潜在匹配点一定在新的子图区域中。若找到一对内部指令等价，但前驱或后继不同构，则将该匹配对添加到候选映射表中。在整个自底向上阶段完成后，利用与第一阶段相同的方法确定候选映射表中确切的同构基本块，并移入结果基本块中。

算法 2：子图分割与图内基本块匹配的算法

输入：控制流图 cfg1,cfg2; 结果映射表 M;一个空的候选映射表 C;

输出：结果映射表 M

```
1.    range=getRange(cfg1,M);
2.    foreach range1 in range
3.        range2=getMatchedRange(range1,M);
4.        foreach block1 ∈ {BFS(cfg1) in range1}
5.            foreach block2 ∈ {BFS(cfg2) in range2}
6.                if(isSameBlock(block1,block2)
7.                    if(isSameDeps(block1,block2)
8.                        M.insert(block1,block2);
9.                        update(range2);
10.                   else
11.                       C.insert(block1,block2);
12.   foreach n1 ∈N1 | (n1, nx) ∈C do
13.       if(C.getRange(n1)==1)
14.           C.move(n1, C[n1],M);
15.       else
16.           (n1,n2)= isEqualDeps(n1,SN2);
17.           C.move(n1,n2,M);
18.   return M;
```

图 4 子图分割与图内基本块匹配的算法

Fig. 4 The Algorithm of sub-graph separation and graph matching

在两个阶段完成后，将得到一个记录函数控制流图上所有匹配的基本块对。剩余的未匹配的基本块则为两个比较函数各自独有的基本块。这些基本块是高级语言层的修改在 LLVM 中间表示语言上的表示。根据中间表示语言和高级语言的对应关系，同样可以获得程序在高级语言层上的修改位置和相关的信息。

继续分析图 3 中所示的程序。根据第一阶段所得的映射表，v1 版本的控制流图可得到一个独立子图{%0}，v2 版本可得到两个独立子图{{%0}，{%4，%11}}，其中 v1:{%0}与 v2:{%0}

相对应而 v2:{%4,%11}在 v1 版本中无对应子图。因此 v2 版本中的%4 和%11 均可直接判定为新增基本块。而 v1:%0 与 v2:%0 通过基本块差异判定准则进一步分析，得到两个基本块为同构基本块的结果。至此，对程序控制流图的同构子图搜索算法结束，在 sed 程序的 v2 版本中，函数 close_files 相比较于 v1 版本，新增了两个基本块的代码%4,%11。同时，根据 LLVM IR 提供的调试信息，我们可以得到对应的新增源码位置，为源码的 513 和 514 行。

```
1.  procedure getRange(cfg,M)
2.      getBeginPoint(exit(cfg),M,begin);
3.      while(!begin.empty())
4.          start=begin.front();
5.          queue.push(cur)
6.          while(!queue.empty())
7.              end=queue.front();
8.              if(M.find(start)==false)
9.                  foreach  succ ∈ succ(cfg,end) do
10.                     if M.find(succ) == true
11.                         range[start].insert(succ);
12.                         getBeginPoint(succ,M,begin);
13.                     else
14.                         queue.push(succ);
15.             else
16.                 queue.push(succ(cfg,cur));
17.             queue.pop();
18.     return range;
```

图 5　getRange () 算法

Fig. 5 The Algorithm of Function getRange()

```
1.  procedure getBeginPoint(start,M,begin)
2.      queue.push(start);
3.      if(succ(start).size() == 0)
4.          begin.push(start);
5.      while(!queue.empty())
6.          cur=queue.front()
7.          if M.find(cur) == true
8.              foreach  succ ∈ succ(cfg,cur) do
9.                  if M.find(pred)==false
10.                     begin.push(end);
11.             else
12.                 queue.push(succ(cfg,cur));
13.         else
14.             queue.push(pred(cfg,cur));
15.         queue.pop();
16.     return begin;
```

图 6　getBeginPoint()算法

Fig. 6　The Algorithm of Function getBeginPoint()

3 评估

3.1 实验对象以及实验环境

我们从 SIR 程序集中选择了{schedule, sed, grep, gzip}四个程序作为实验对象。选择 SIR benchmark 中的程序为实验对象的原因是，1)该 benchmark 收集了一个程序的多个版本，从中能较为方便的选择对比函数；2)该 benchmark 中有很多程序是将一个程序的多个文件的代码整合成一个文件形成的，因此它们包含大量的不影响代码实际执行逻辑的代码差异，能够较为充分的测试 IRDiff 的性能。实验对象的具体信息罗列在表 1 中。我们的实验环境是一

台运行 Ubuntu 14.04 的台式机，配备一个 4 核 3.40GHz 的 Intel(R) Core(TM) i7-6700 CPU 以及 8GB 的内存空间。

3.2 研究问题

通过对 IRDiff 的实验评估，我们希望研究以下 2 个问题：

问题 1：IRDiff 能否得到正确的代码演化结果？

问题 2：IRDiff 的时间开销有多少？

表 1 实验对象

Tab. 1 Subjects

程序	源版本行数	目标版本行数
schedule(v1,v2)	292	292
schedule(v3,v4)	292	291
sed(v1,v2)	5473	9784
sed(v3,v4)	7067	7070
gzip(v1,v2)	4521	5048
gzip(v3,v4)	5059	5178
grep(v1,v2)	9366	9943
grep(v3,v4)	10032	10073

行数。其中可能包含一些注释代码，而在上一节的实验对象描述中，只包含有效的代码行数而忽略了注释，因此 Diff 得到的差异行数可能会多于两个比较程序的代码总和。而且，由于分析对象的特殊性，Diff 的分析过程中并不能匹配相应的函数，因此部分结果几乎将比较程序的所有代码都认作是差异代码，例如 sed(v1,v2)，gzip(v3,v4)和 grep(V3,v4)。后四列为 IRDiff 所得到的结果。第四列是两个版本中独有函数的个数，在模块层差异分析过程中，直接将两个程序中的独有代码标记为差异代码。有些程序对中并不包含独有函数，说明在两个版本中未有函数的增删。如 gzip(v1,v2)，gzip-v1 和 gzip-v2 之间共有 17 个独有函数，总计 490 行。最后两列是 IRDiff 在分析两个程序中的同名函数时，搜索到的两个程序的独有基本块个数以及映射到源码层的代码总行数。由表 2 我们可以得到这样的一个结论，利用 IRDiff 逐层对 LLVM 程序进行代码差异分析，能够消除如代码位移等不影响代码实际语法的文本改动对代码差异分析的影响，更细粒度的得到代码差异分析结果。

表 2 IRDiff 与 Diff 的比较

Tab 2 The comparison between IRDiff and Diff

程序	Diff		IRDiff				
	差异块	差异行	差异 函数(个)	差异 函数行	差异 基本块（个）	差异基本块 行数	差异 行数
schedule(v1,v2)	2	6	-	-	4	6	6
schedule(v3,v4)	1	5	-	-	2	4	4
sed(v1,v2)	460	12232	-	-	1660	85	85
sed(v3,v4)	5	17	-	-	16	15	15
gzip(v1,v2)	66	1002	17	90	278	323	813
gzip(v3,v4)	620	13153	-	-	1204	235	235
grep(v1,v2)	281	1750	6	70	342	670	940
grep(v3,v4)	1155	13661	-	-	439	479	479

3.2.1 IRDiff 的时间性能

表 3 展示了 IRDiff 分析的耗时情况，我们对每组程序进行 10 次分析，表中数字是对总耗时取平均值之后所得的结果。总体上来说，程序的规模越大，IRDiff 的耗时越多。从表 3 中可知，分析 sed(v1,v2)程序对的耗时最多。通过分析结果之后，我们发现，当代码文本中存在越多的增删修改时，对生成的控制流图的影响越大，由此，第一阶段所能匹配的最大同构子图的规模越小，使得第二阶段划分的独立子图规模增大，增加了第二阶段的分析负担，从而导致分析的耗时越多。

结合表 2 和表 3，我们可以发现在分析 gzip(v3,v4)时，得到的总差异基本块个数相对较多，但与 sed(v1,v2)组相比，耗时却相对较少。通过分析 gzip-v3 和 gzip-v4 的代码，我们发现 gzip-v3 和 gzip-v4 中的函数较多，平均函数规模较小，使得第二阶段的负担较轻，从而耗时较少。

表 3 IRDiff 的耗时

Tab. 3 Time consuming of IRDiff

程序	运行时间
schedule(v1,v2)	0.130 s
schedule(v3,v4)	0.125 s
sed(v1,v2)	6.059 s
sed(v3,v4)	0.951s
gzip(v1,v2)	0.944 s
gzip(v3,v4)	2.158 s
grep(v1,v2)	1.619 s
grep(v3,v4)	1.681 s

4 相关工作

源代码间的差异分析是多版本程序中一个长期研究的基础问题。现有的方法大多依赖于词法、语法或语义分析技术实现代码间的差异分析。根据分析对象的不同还可以进一步将方法分为面向文本、面向树结构以及面向图结构的差异分析技术。本文提出的算法是属于面向图结构基于语法的源代码差异分析。下面将以分析对象为分类依据，展示现有的各类相关工作。

以文本行为分析单元的程序多个版本间代码差异分析技术是代码演化分析中最常见的粒度之一。使用这种粒度的算法往往有运行速度快且与语言无关的优势。GNU diff[5,6]文本差异分析中最出色的工具之一。利用最长子序列算法计算文本修改，得到代码的修改、插入和删除等编辑信息。但这些算法最主要的问题就是无法提供细粒度的差异分析。程序的一行代码往往可以分解成多个更细粒度的编程元素，并且每行代码之间存在一些关联关系，并不是完全独立的，diff 并没有很好的利用这些信息，使得得到的结果精度低，误报高。

Chawathe 等人提出的计算编辑脚本的算法[7]是树结构差异分析中最出色的算法之一。但该算法的分析对象为表示 Latex 文档的树结构，并不能很好的适用于常见的编程语言的 AST 树结构。G. Dotzler 等人提出了针对 AST 不同的 AST 节点匹配算法 [1,2,3,4]，结合树结构编辑距离的算法[7,8]实现代码比对。M. Hashimoto、J. W. Hunt 等人将 AST 树与语义信息结合，生成相应的语义图实现细粒度的代码差异分析 [8,9,10,11,12,13,14]。其中同时结合面向对象编程语言的特征，通过匹配类、接口、方法等方式，将程序划分成不同的独立片段，有针对性地进行代码比对分析，而不是仅仅通过代码的文本位置进行比对[9,10,11,14]。

S. Horwitz[15] 提出了一种利用等价行为判定将程序分割成多个集合，针对这些行为等价的集合进行语义和文本差异识别的技术。其中的分割算法使用程序依赖图，即结合程序依赖图和静态单赋值形式形成的程序图，但图中并不包含语义信息，例如类型信息等。并且该方法仅适用于不包含函数或进程的语言。D. Binkley[16]改进 S. Horwitz[15] 的算法，将技术扩展应用于包含进程的语言。H. Palikareva[17]通过分析修改前后程序输入与输出之间的依赖关系分析两个程序间的语义差异。M. Pawlik[18]在工作中描述了一个程序的语言是如何通过部分函数模拟的，并且设置布尔代数来识别不同程序代码间的增加以及删除的编辑操作。S. Raghavan[19]利用抽象解释技术，利用关系抽象域模拟包含语义差异的关系，并提出一个贪心搜索算法计算两个程序间的最小抽象语义差异。BMAT[20]是面向二进制程序的匹配工具，利用局部控制流信息和基本块相似性判断技术实现原始版本与修改版本间的基本块匹配。

5 讨论与未来工作

首先，由于 IRDiff 需将源代码转换成 LLVM 中间表示,，因此使用 IRDiff 时必须提供可编译的完整源码，而无法对两个代码片段进行分析。其次，工具对输入有一个隐含的限制，即两个对比函数需有一定程度的相似性。若两个函数完全不同，使得第一阶段几乎找不到任何匹配对，使得第二阶段所需处理的独立子图与整个控制流图相比，规模并没有实质上的缩小，并且由于无法得到确切的匹配对，子图边界收缩缓慢，分析的耗时将增大。最后，虽然在分析过程中的最小对比单元为指令层代码，但当前 IRDiff 的匹配单元是一个基本块而非指令，而一个基本块可能会包含多行源码，因此生成的差异报告精度不足。后续的工作将研究如何将匹配单元细化至指令层，提高分析精度。

6 结语

软件演化分析中一个重要的研究问题是在一个程序的多个版本之间识别代码的改变。现有的工作或仅依据文本信息跟踪源代码文件之间的改动，忽略程序中包含的语法信息；或利用 AST 进行代码文本匹配，而未能正确的处理由于代码移动带来的代码比对错位的影响。本文利用 LLVM 中间表示语言及函数控制流图，首先通过函数匹配，从函数层确定正确的代码对比范围，避免了错位对比带来的错误差异分析结果。其次，在函数内的分析过程中，利用最大同构子图搜索算法，找到函数内的程序共有代码以及差异代码，分析源码的改动情况。基于上述算法，我们实现了 C/C++程序的代码分析工具 IRDiff，并利用 SIR benchmark 对工具的有效性和实用性进行了评估。实验结果表明 IRDiff 能较准确的找出代码之间的差异，且时间开销不大，对万行级别的代码分析时间在 10 秒之内。

[参考文献] (References)

[1] G. Dotzler and M. Philippsen, 'Move-optimized source code tree differencing'[A], 2016, pp. 660-671
[2] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus,Fine-grained and accurate source code differencing[A], in ASE[C], 2014.
[3] Fluri, Beat et al. Change Distilling:Tree Differencing for Fine-Grained Source Code Change Extraction.[J] IEEE Transactions on Software Engineering 33 (2007): n. pag.
[4] M. Hashimoto and A. Mori, Diff/TS: A Tool for Fine-Grained Structural Change Analysis [A], in WCRE[C], 2008.
[5] J. W. Hunt and T. G. Szymanski,A Fast Algorithm for Computing Longest Common Subsequences[C], 2000
[6] J. Krinke, Identifying Similar Code with Program Dependence Graphs[C], in WCRE, 2001
[7] P. Bille, 'A survey on tree edit distance and related problems' [J], Theor. Comput. Sci., vol. 337, pp. 217-239, 2005.

[8] M. Pawlik and N. Augsten, RTED: a robust algorithm for the tree edit distance [C], Proceedings of the VLDB Endowment, vol. 5, no. 4, pp. 334-345, Jan. 2011.
[9] T. Apiwattanapong, A. Orso, and M. J. Harrold, 'A differencing algorithm for object-oriented programs'[C], Proceedings. 19th International Conference on Automated Software Engineering, 2004., pp. 2-13, 2004.
[10] T. Apiwattanapong, A. Orso, and M. J. Harrold, 'JDiff: A differencing technique and tool for object-oriented programs' [C], Automated Software Engineering, vol. 14, pp. 3-36, 2006
[11] J.-R. Falleri, M. Huchard, M. Lafourcade, and C. Nebut, 'Metamodel Matching for Automatic Model Transformation Generation' [C], in MoDELS, 2008.
[12] J. C. King, 'Symbolic execution and program testing' [J], Communications of the ACM, vol. 19, no. 7, pp. 385-394, Jan. 1976.
[13] Z. Xing and E. Stroulia, 'UMLDiff: an algorithm for object-oriented design differencing' [C], in ASE, 2005.
[14] S. Raghavan, R. Rohana, D. Leon, A. Podgurski, and V. Augustine, 'Dex: A Semantic-Graph Differencing Tool for Studying Changes in Large Code Bases' [C], in 20th International Conference on Software Maintenance (ICSM 2004), 11-17 September 2004, Chicago, IL, USA, 2004, pp. 188-197.
[15] S. Horwitz, 'Identifying the Semantic and Textual Differences Between Two Versions of a Program' [C], in PLDI, 1990
[16] D. Binkley, 'Using Semantic Differencing to Reduce the Cost of Regression Testing'[C], 1992.
[17] H. Palikareva, T. Kuchta, and C. Cadar, 'Shadow of a doubt: testing for divergences between software versions' [C], presented at the Proceedings of the 38th International Conference on Software Engineering, 2016, pp. 1181-1192.
[18] V. Berzins, 'Software Merge: Semantics of Combining Changes to Programs' [J], ACM Trans. Program. Lang. Syst., vol. 16, pp. 1875-1903, 1994.
[19] W. Miller and E. W. Myers, 'A File Comparison Program' [J], Softw., Pract. Exper., vol. 15, pp. 1025-1040, 1985.
[20] Z. Wang, K. Pierce, and S. McFarling, ‘BMAT - A Binary Matching Tool for Stale Profile Propagation’ [J], J. Instruction-Level Parallelism, vol. 2, 2000.