第 33 卷 第 7 期 计 算 机 工 程 2007 年 4 月
Vol.33 No.7 Computer Engineering April 2007
·软件技术与数据库· 文章编号：1000—3428(2007)07—0087—02 文献标识码：A 中图分类号：TP311

汇编程序覆盖测试中虚拟插桩的实现

王学东，汪文勇
（电子科技大学计算机科学与工程学院，成都 610054）

摘 要：在对汇编语言源程序的测试工作中，由于其非结构化的设计思想以及众多的跳转指令，使得插桩非常困难。该文在汇编嵌入式软件程序流程图自动生成的研究基础上，提出了以 PC 值为基准条件的断点设置、检测机制以及虚拟插桩机制。并结合特定的测试用例在仿真测试平台上得以实现，最后通过实验检验了其正确性。

关键词：汇编程序；Lex；Yacc；虚拟插桩；语句覆盖；分支覆盖

Implementation of Virtual Instrumentation in Coverage Test of Assembly Code

WANG Xuedong, WANG Wenyong
（School of Computer Science & Engineering, University of Electronic Science and Technology of China, Chengdu 610054）

【Abstract】 It is very difficult to make implementation in the assembly code because of its none structural design idea. Based on the research of embedded assembly program flow chart automatic building, the breakpoint setting and checking mechanism, virtual instrumentation mechanism based on the PC values are proposed. The mechanisms are realized and verified in the emulation testing platform with special test case.

【Key words】 Assembly program; Lex; Yacc; Virtual instrumentation; Statement coverage; Branch coverage

程序插桩是实现覆盖测试的重要手段，它用来捕获程序执行过程中变量值的变化情况，也可以用来检测程序的分支覆盖和语句覆盖。在保持被测试程序原有逻辑完整性基础上，在程序中插入一些探针(又称探测仪)。这些探针本质上就是信息采集的代码段，可以是赋值语句或采集覆盖信息的函数调用。通过探针的执行并输出程序的运行特征数据。基于这些特征数据分析，可以获得程序的控制流及数据流信息，进而得到覆盖测试等动态信息。

汇编语言是一种非结构化的程序设计语言，汇编语言源程序中往往采用很多条件和无条件跳转指令，使得程序结构复杂，程序的执行路径难于判断，其插桩的实现相当困难。本文在研究汇编语言指令结构的基础上，提出了一种虚拟插桩机制，在汇编语言源程序实现了真实插桩的功能，并达到了覆盖测试的目的。

1 虚拟插桩前的基础工作

根据文献[1]对汇编语言程序结构的分析(本文对汇编语言的讨论均基于 Intel 8051 单片机指令系统)，定义了 5 种类型的汇编指令单链表，然后在这 5 类单链表之间建立起了关联，并掌握了被测程序的流程。以此为基础进行下述的研究。

1.1 基于 Lex 的词法分析的实现

汇编程序以行为程序逻辑单位，当分析完一行语句时，提交该行语句的如下属性：定义标号，操作符，操作数，是否有引用标号，语句行号，指令起始地址，指令类型，前一指令类型。其中，必须弄清楚每条汇编指令在经汇编器汇编后得到的机器码在内存中所占字节数，而对指令机器码的字节排布进行计算将在语法分析部分进行。所以在词法分析部分要提供能够影响汇编后程序机器码码在内存中排布的有关信息。51 标准指令集和伪指令中的汇编起始指令 org，数据定义指令 db、dw 会影响机器代码的内存排布。

汇编源程序按顺序汇编，如果开始没有 org，则机器码从 0000h 开始组织，遇到标准指令，则统计该指令经汇编后所得机器码所占用字节数，pc 值增加，遇到数据定义指令 db,dw，则根据该指令定义的数据数量统计其占用的字节数，如 db 02h，03h 则占用两字节空间，pc 值增加。程序中遇到 org 时，改变当前 pc 值为 org 指令对应的值。除顺序指令之外的程序控制转移指令单独分析。

1.2 基于 Yacc 的语法分析的实现

语法分析是分析完一行就提交。用以下几个全局变量记录一条语句的各种属性和统计信息。

(1)记录当前指令的起始 pc 值，定义为 unsigned short u_pc_next = 0；初始化为 0，当语法分析程序分析完一条汇编源程序代码后 u_pc_next 会加上该条源代码所占的字节数。

(2)指令操作符记录(助记符字符串，指令类型)，定义为

```
struct tag_instruction_node{
char *instruction;
UCHAR instruction_type; //指令类型
}instruction_node = {NULL, SUC_TYPE_NULLTYPE};
```

它是分析辅助结构，当语法分析程序分析完一条汇编源程序代码后将被清空。

(3)操作数结点，定义了操作数的个数，操作数和指令所占用的字节数。

```
struct tag_operands_node{
int count_of_operand; //操作数个数(不包括标号)
char *operands[OPERAND_NO];   //操作数地址
UCHAR bytes; //操作数占用的内存字节数
```

作者简介：王学东(1973－)，男，硕士生，主研方向：软件测试；汪文勇，硕士、副教授

收稿日期：2006-03-28 **E-mail：**dongge@tom.com

```
UCHAR determined_by_instruction;
//是否由指令完全确定所占用内存字节数
}operands_node = { 0, {0}, 1, FALSE};
```

操作数结点也是分析辅助结构，当语法分析程序分析完一条汇编源程序代码后将被清空。

当获得每条汇编源程序代码经汇编器汇编后其机器码所占用的字节数后，就可以计算出每条指令的绝对地址，也就可以掌握每条指令对应的程序记数器所指内存单元的值。

2 虚拟插桩的实现

常规的程序自动插桩方法是在被测源程序的插桩点处插入赋值语句或函数，待其实际运行后再检测运行结果，以此来判断被测代码的执行情况。但是由于单片机无操作系统的支持，并且存储器的管理完全由程序员负责，如果采取这种方式，就产生了对插桩代码中的变量的存储问题，因此不可避免地会对被测汇编源程序的程序员对单片机的存储资源的分配使用方案造成破坏。

为避免上述情况发生，根据仿真器(本文所提的测试是基于仿真环境下进行的)的断点机制以及对 CPU 内部程序记数器指针的仿真，提出了一种虚拟插桩机制，在被测源程序的插桩点处不用插入任何代码，这样就不涉及到常规插桩方法中对插桩代码中的变量的存储问题。

虚拟插桩机制是在被测源程序的每一个插桩点处设置断点，当被测程序运行到此处时，则抛出程序记数器所指单元的值(PC 值)将其记录下来。程序运行结束后，得到一个 PC 值的输出流，通过文献[1]的 5 类单链表与之建立的对应关系可得到被测程序的实际执行路径，从而计算出各类覆盖指标。

2.1 虚拟插桩的基础数据结构及其他基础设施

(1)根据词法语法分析后得到的被测汇编源程序中可执行语句的 5 个单链表，单链表中的每个节点记录了该节点所对应的待测汇编源程序中相应语句块的起始 PC 值。5 个单链表将插桩过程中使用。

(2)以 5 个单链表为基础数据结构生成的对应于整个待测汇编源程序的语句级程序流程图。这个流程图是在被测源程序被插桩并且仿真运行后，进行覆盖率分析时使用。而且在 5 个单链表节点的数据结构中有些数据成员是在流程图的生成过程中得到赋值的，例如一个条件转移节点的左右分支后继节点的指针就是在此过程中得到的，以这两个指针为基础，可以进一步得到其左右分支后继节点所对应语句块的起始 PC 值。

(3)确保在仿真器的实现中提供以 PC 值为基准条件的断点设置及检测机制。

2.2 插桩的实现与覆盖测试指标的计算

首先遍历 5 个单链表，以每个节点中记录的对应语句块起始 PC 值为断点设置条件来设置断点，这里设置的断点就是我们所说的“虚拟桩”。在被测汇编源程序的仿真运行过程中，一旦某个断点被触发，就记录当前的 PC 值，待整个被测源程序的仿真运行结束后，就能得到一个程序实际运行路径的 PC 记录流。然后再以此为基础数据进行测试结果分析。

2.2.1 分支覆盖率分析

遍历条件转移单链表，为每个条件转移节点建立一个分支覆盖信息节点，所有的分支覆盖信息节点用链表组织，形成一个分支覆盖信息节点的单链表。分支覆盖信息节点的数据结构如下：

```
struct    branch_cover_info{//分支覆盖信息节点
    struct branch_cover _info *pNext;
unsigned short    pc_self; //条件转移节点对应语句块起始 pc
unsigned short    pc_left; //左分支节点对应语句块起始 pc
    unsigned short    pc_right; //右分支节点对应语句块起始 pc
    UCHAR     left_covered_flag; //左路分支覆盖标记
    UCHAR     right_covered_flag; //右路分支覆盖标记      };
```

(1)初始化该链表，初始化过程中，两个分支覆盖标记取值赋为零，其余数据成员取值都能在对应的条件转移单链表的节点中取得。

(2)对 PC 记录流维护一个读取指针，从该 PC 记录流中读取一个记录值。

(3)将读取出来的 PC 值与分支覆盖信息节点链表中每个节点的 pc_self 相比较，如果搜索不到某个分支覆盖信息节点的 pc_self 与该 PC 值相等，略过读出的该 PC 值，读取指针前移以读取下一个记录。如果搜索到某个分支覆盖信息节点的 pc_self 与该 PC 值相等，则说明这个 PC 值代表一个条件转移节点，读取指针前移以读取下一个记录，将新读取出来的 PC 值与该分支覆盖信息节点的 pc_left 和 pc_right 相比较，如果和其中某个相等，就对相应的那一路分支置覆盖标记，即对 left_covered_flag 或者是 right_covered_flag 赋非零值，然后读取指针前移以读取下一个记录。

(4)对 PC 记录流中的每个记录均作第(2)步的处理。

(5)统计分支覆盖信息节点链表的节点总数，以此乘以 2(每个条件转移节点就是一个两路分支节点)，得到总分支数 total_branch。

(6)统计分支覆盖信息节点链表中 left_covered_flag 或者是 right_covered_flag 标记不为零的节点数，得到被覆盖分支总数 covered_branch。

(7)用 covered_branch 除以 total_branch 得到分支覆盖率。

2.2.2 语句覆盖率分析

遍历 5 个单链表，为每个节点建立一个语句覆盖信息节点，所有的语句覆盖信息节点用链表组织，形成一个语句覆盖信息节点的单链表。语句覆盖信息节点的数据结构如下：

```
struct statement_cover_info{//语句覆盖信息节点
        struct cover_node *pNext;
  unsigned short    pc_self; //节点对应语句块起始 pc
UINT    start_line; //节点起始行号
   UINT end_line; //节点结束行号
    UCHAR     statement_covered_flag; //节点覆盖标记      };
```

(1)初始化该链表，在初始化过程中，覆盖标记取值赋为零，其余数据成员的值都能在 5 个单链表的对应节点中取得。

(2)对 PC 记录流维护一个读取指针，从该 PC 记录流中读取一个记录值。

(3)将读取出来的 PC 值与语句覆盖信息节点链表中每个节点的 pc_self 相比较，如果搜索不到某个语句覆盖信息节点的 pc_self 与该 PC 值相等，则略过该 PC 值，读取指针前移以读取下一个记录。如果搜索到某个语句覆盖信息节点的 pc_self 与该 PC 值相等，则说明该节点所对应的语句块在程序执行流程中被覆盖过，对这个语句覆盖信息节点的数据成员 statement_covered_flag 赋非零值，然后读取指针前移以读取下一个记录。

(4)对 PC 记录流中的每个记录均作第(3)步的处理。

(5)遍历语句覆盖信息节点链表的节点，用节点的 end_line 减 start_line 再加 1，得到每个节点对应语句块的语句条数，据此计算整个被测源程序的总语句数 total_statement。

遍历语句覆盖信息节点链表的节点，对覆盖标记 statement_covered_flag 为非零值的节点用其 end_line 减 start_line 再加 1，得到这个节点对应语句块的语句条数，据此计算整个被测源程序的被覆盖的语句数 covered_statement。

(6)用 covered_statement 除以 total_statement 得到语句覆盖率。

2.3 实验结果

在我们实现的仿真测试平台上，实现了本文提出的以 PC 值为基准条件的断点设置及检测机制、虚拟插桩机制以及相应的分析方法，并作了大量的实验。

在实验中采用一个已投入使用的实用串口通信控制系统作为实验对象。在该系统中，提取出 73 个可独立运行的并且

（下转第 98 页）