

一种使用静态分析的汇编代码缺陷检测方法

邱 景, 苏小红, 马培军

(哈尔滨工业大学 计算机科学与技术学院, 150001 哈尔滨)

摘 要: 针对当前缺乏汇编代码自动化审查工具的情况,对汇编代码人工审查方法进行研究,提出了一种基于静态分析的汇编代码缺陷检测方法.该方法中,在控制流线性化后,运用特征识别处理间接寻址跳转,采用结点克隆处理延迟条件分支,使用调用序列处理存在递归函数的过程间控制流图的构造.在此基础上,实现了 ADSP SHARC 汇编代码检测工具,并进行了静态分析测试和缺陷检测测试.测试结果表明,该方法可以有效地检测汇编代码中的不可退出点、循环、寄存器、以及内存访问缺陷.

关键词: 汇编代码;静态分析;缺陷检测;延迟分支

中图分类号: TP313

文献标志码: A

文章编号: 0367-6234(2013)02-0053-07

Defect detection for assembly codes based on static analysis

QIU Jing, SU Xiaohong, MA Peijun

(School of Computer Science and Technology, Harbin Institute of Technology, 150001 Harbin, China)

Abstract: Aiming at the present situation that needs to develop a code review tool for assembly codes, this paper studies the procedure of manual code reviews and proposes a method to detect defects in assembly codes based on static analysis. After the control flow linearization, compiler patterns are used to solve indirect jumps, and node cloning is used to recover the control flow of the delayed branch. In the construction of inter-procedural control flow graph, the recursive function is in-lined by means of a call trace with limited depth. The prototyping tool for ADSP SHARC assembly codes is realized finally. Experimental results show that the tool can effectively detect defects in loops, registers, and memory accessing.

Key words: assembly code; static analysis; defect detection; delayed branch

目前在 DSP 平台上编程多使用汇编语言与 C 语言^[1],而当前对于汇编代码的缺陷检测的相关研究文献则比较少,传统的缺陷检测方法在分析日益复杂的现代处理器汇编代码时面临诸多挑战.这些挑战主要来自处理器架构相关的并行操作指令,人工优化或编译器高度优化产生的指令流水线代码带来的复杂的控制流和数据流关系.

本文选用 ADSP SHARC 汇编语言来描述汇编代码的缺陷检测,其中代码控制流分析部分借

鉴文献[2]的方法. ADSP SHARC 处理器系列广泛应用与通信、图像处理、生物医学、自动控制等领域.尽管如此,本文方法同样可以应用与其他汇编语言的缺陷检测.

1 汇编代码缺陷检测模型

汇编代码缺陷检测模型如图 1 所示.汇编代码经过语法分析后生成抽象语法树,然后经过代码线性化处理,生成控制流图.再使用路径简化处理,精简流图.为了过程间分析,构造了 ICFG(过程间控制流图).数据流分析使用了常数传播、到达定值、活跃变量分析,最后使用预定义的检测规则进行缺陷检测生成报告.

本文主要研究汇编代码的静态分析和缺陷检测,其可以应用在 C/C++ 或二进制文件检测上.

收稿日期: 2011-12-01.

基金项目: 国家自然科学基金资助项目(61173021).

作者简介: 邱 景(1982—),男,博士研究生;

苏小红(1966—),女,教授,博士生导师;

马培军(1963—),男,教授,博士生导师.

通信作者: 邱 景, qj@sse.hit.edu.cn.

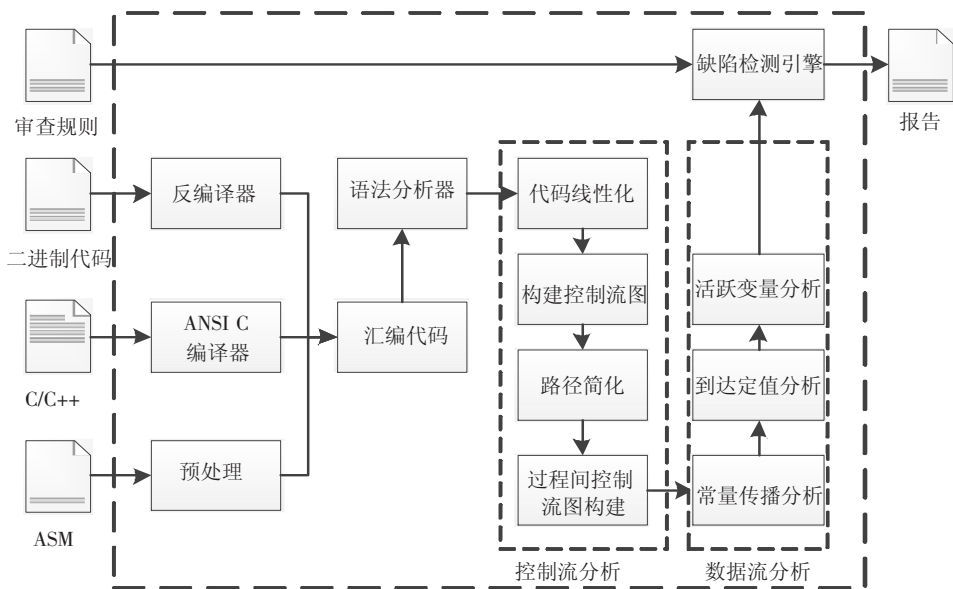


图 1 汇编代码缺陷检测模型

2 汇编代码的静态分析

为处理多个代码文件间对变量或函数的交叉引用,本文为每一个文件设置一个分析上下文.分析上下文里包含了单个文件静态分析的所有结果,包括符号表、控制流信息等.文件中用 global 声明的函数或变量可以在各个上下文间可见.

2.1 线性化

ADSP SHARC 可以在一个指令周期内完成 3 个计算.有并行操作的指令不方便后续的静态分析,因此需要进行线性化.

线性化过程分为:1)为代码中的每条指令添加序号,并增加条件标记属性;2)对于并行操作的指令,按并行操作分裂.分裂后的各指令序号和原来相同,并按执行条件分配条件标记.

如这样的指令“n: IF EQ JUMP (PC, 2), ELSE R0 = 1, DM(I0, M0) = R1;”线性化结果如表 1 所示.

表 1 线性化结果

序号	操作	条件标记
n	IF EQ	
n	JUMP(PC, 2)	T
n	R0 = 1	F
n	DM(I0, M0)	F

2.2 控制流分析

2.2.1 间接寻址跳转

间接寻址跳转指令是指如 JUMP (Md, Ic) 这样的指令.在静态分析中,由于无法确定间接寻址地址,理论上必须假设 JUMP (Md, Ic) 跳转目标为程序中任意指令.由于假设跳转到任意位置会

极大增加后续分析负担,所以多数情况下仅假设可能跳转到程序中有标号的位置.

某些情况下,可以将跳转目标范围缩小.图 2 中代码为含有 switch/case 结构的 C 代码经 ANSI C 编译器编译出的汇编代码.

```
M4 = R2;
I4 = . SWITCH.0;
I12 = DM( M4, I4 );
JUMP( M13, I12 ) ( DB );
R0 = M6;
NOP;
.P36L2: R0 = 2;
.P36L7: JUMP( PC, _EXIT ) ( DB ); RFRAME; NOP;
.P36L3: JUMP( PC, . P36L7 ) ( DB ); R0 = 3; NOP;
.P36L4: JUMP( PC, . P36L7 ) ( DB ); R0 = 4; NOP;
.SWITCH.0:
    . VAR = . P36L7;
    . VAR = . P36L2;
    . VAR = . P36L3;
    . VAR = . P36L4;
.SWITCH.0. END;
```

图 2 间接寻址跳转

对于图 2 中的 JUMP (M13, I12),如果用高级语言描述的话,相当于 JUMP PM[M13 + DM[R2 + . SWITCH.0]].可以看出,. SWITCH.0 相当于数组中的基址,R2 是偏移.因此 JUMP (M13, I12) 实际跳转地址只能是. P36L7, . P36L2, . P36L3, . P36L4. 这里的识别并没有采取文献[2]中程序切片的方法,而是直接采用特征识别方法.特征代码和特定的编译器相关.ADI Visual DSP + + 5.0 特征代码如下:

```
I4 = jump_table; // 跳转表位置
I12 = DM( M4, I4 );
JUMP( M13, I12 );
```

文献[2]采用了程序切片的技术,需要精确计算数据流,而计算数据流需要精确的控制流图.因此需要反复迭代分析控制流和数据流.对于图2代码,文献[2]最后虽然能够成功得到I12的切片,但是由于M13值未知,还是不能肯定最终跳转位置.相比程序切片方法,特征识别法方法简单、分析速度快.

2.2.2 延迟分支(Delayed Branch)

ADSP SHARC 执行延迟分支时,2个延迟槽(Delay Slot)中指令先被执行,然后再执行分支指令.延迟分支指令分为:普通延迟分支指令和条件延迟分支指令.

1) 普通延迟分支.

图3为DSP执行 $R0 = R2$, $R1 = R3$,最后执行foo处指令.

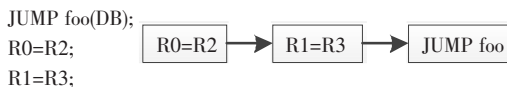


图3 普通延迟分支

2) 条件延迟分支.

为讨论方便,假设指令op延迟槽数为 k ,且 $n+1$ 到 $n+k$ 都是顺序执行指令.

情况1 IF cond op (DB).

当cond成立时,op延迟 k 周期执行,执行 $n+1, \dots, n+k$,执行op分支目标处指令,....当cond不成立时,执行 $n+1, n+2, \dots$.

情况2 IF cond op (DB), op2.

当cond成立时,op延迟 k 周期执行,op2立刻执行,执行 $n+1, \dots, n+k$,执行op分支目标处指令,....当cond不成立时,执行 $n+1, n+2, \dots$.

情况3 IF cond op (DB), ELSE op2.

当cond成立时,op延迟 k 周期执行,op2不执行,执行 $n+1, \dots, n+k$,执行op分支目标处指令,....当cond不成立时,执行op2,执行 $n+1, n+2, \dots$.

由上述讨论可知,上述3种情况控制流图如图4~6所示.情况2中op2为空时退化成情况1.图中仅当op为CALL时,虚线存在.

情况1中延迟槽中指令可合并(图中 $n+1 \dots k$ 结点),因为无论cond是否满足,都要先执行 $n+1 \dots k$ 指令.对于情况2、3由于 $n+1 \dots k$ 指令可能和op2有数据依赖,故无法合并.为了保证控制流信息的完整性,需要复制 $n+1 \dots k$ 指令到2条路径中,这就是结点克隆法.对于延迟槽的处理文献[2]仅仅给了简单的文字描述:基本块中的延迟分支增加边到其他基本块中,不再假设基本

块的末尾是控制流操作.

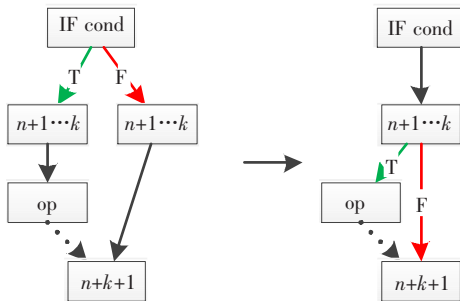


图4 情况1控制流图

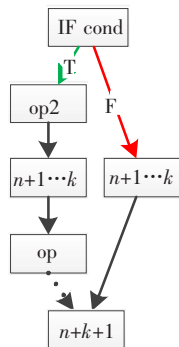


图5 情况2控制流图

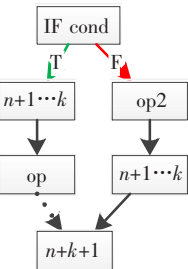


图6 情况3控制流图

控制流分析完成后,进行控制流路径简化,主要是删除NOP, JUMP/CJUMP, IDLE/IDLE16等指令.

2.3 过程间控制流图ICFG

在汇编语言中,函数多使用寄存器或全局变量传递参数,因此使用过程间分析可以得到更精确的分析结果^[3].

本文采用的过程间分析方法是函数内联,把一个函数调用替换为被调用函数的函数体.ICFG的构造分为:构造函数调用图和函数内联.

调用图中如果存在循环,就有可能发生递归调用,从而使函数内联无法结束.解决办法是记录调用序列限制内联深度.这样处理递归程序会损失分析精度,但是本文认为在大部分情况下只要内联深度足够,分析结果是可以接受的.算法如下:

输入:函数 f

输出:函数内联过的函数 f'

```
void inline(Function f, Stack call_trace) {
```

```

if( call_trace.count (f) = N//f 内联过 N 次?
    return;
call_trace.push (f);
for( call_side cs:f ) {
    Function p' = cs.target;
    inline (p', call_trace);
    Function cloned = p'. clone();
    修改 cs 的前驱的后继到 cloned.Entry;
    修改 cs 的后继的前驱到 cloned.Exit;
}
清除 f 的调用点记录;
call_trace.pop();
}

```

在算法里,函数内联时,堆栈 call_trace 压入此函数,内联后堆栈弹出栈顶.内联中,如果发现此函数已经在 call_trace 中出现过 N 次,说明已达指定深度,停止内联.

2.4 数据流分析

本文使用了 3 种数据流分析技术:到达定值方程、活跃变量分析和常量传播.具体的计算方法及本文 gen, kill, use, def, UD 集合的定义见文献[4].在计算中,假设程序中所有代码都是可达的,并忽略分析内存相关的指令.在搜集数据流信息中,充分考虑指令的副作用,不考虑对状态寄存器的影响.所谓指令副作用是指指令执行后除了操作数外对整个系统状态带来的影响,譬如算术运算指令会影响状态寄存器.

对于图 7 中代码,当执行 1→3 时,3: R1 = R3 中 R3 可能未初始化.如果仅计算文献[4]中 IN 和 OUT,有 $IN[3] = \{1, 2\}$, R3 的 UD 链为 $\{2\}$ 非空,发现不了此处缺陷.

本文参考了文献[5]中的方法,在到达定值计算中,将文献[4]中的 IN, OUT 定义为 MAY_IN, MAY_OUT, 增加 MUST_IN, MUST_OUT.

定义 1 $MUST_IN[s]$ 为肯定能到达 s 入口的定值点集合, $MUST_OUT[s]$ 为肯定能到达 s 出口的定值点集合.

$MUST_OUT$ 由 2 部分组成: s 中产生的定值点,还有肯定到达 s 入口而没有被 s 杀死(重定义)的定值点,即

$$MUST_OUT[s] = (MUST_IN[s] - kill[s]) \cup gen[s].$$

如图 8 所示, $MUST_IN[s]$ 由 2 部分组成:

- 1) $MUST_OUT[p1] \cap MUST_OUT[p2]$;
- 2) $p1, p2$ 如果都对某一变量进行定值,那么 $p1, p2$ 都到达 s .

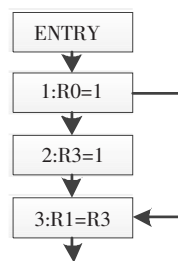


图 7 可能使用未初始化变量的例子

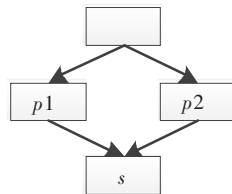


图 8 控制流图

故计算公式为

$$MUST_IN[s] = \left(\bigcap_{p \in pred(s)} MUST_OUT[p] \right) \cup \Phi(pred(s)).$$

其中

$$\Phi(s) = \begin{cases} \emptyset, & \text{if } \bigcap_{p \in S} def[p] = \emptyset; \\ s, & \text{otherwise.} \end{cases}$$

可以看出, $MUST_OUT$ 的计算和文献[5]一致,在 $MUST_IN$ 的计算中,本文使用了活跃变量分析结果 def.

定义 2 $MUST_UD[s]$ 为肯定到达 s 引用变量定值点的定值点集合.

和 UD 链的计算一样, $MUST_IN[s]$ 中对 s 引用变量的定值点都在 $MUST_UD$ 链中.

对于图 7 中代码有, $MUST_IN[3] = \{1\}$, $MUST_UD[3] = \emptyset$, 故 $R1 = R3$ 有可能引用未初始化的寄存器 R3.

3 汇编代码缺陷检测方法

3.1 未初始化的变量及无用代码

计算到达定值方程后,计算每条语句的 use 集合中每个变量的 UD 链.如果为空,则说明该变量没有定值. $use[s]$ 中变量 v 的 UD 链 = $IN[s]$ 中对 v 的定值点集合.

无用代码的检测由活跃变量分析的结果检出:如果 $def[s]$ 中存在变量 v 不在 $OUT[s]$ 中,则说明 v 在 s 执行后被 kill 了.

不可达代码的检测首先从各个中断函数开始遍历构造 ICFG,然后检测是否有没有遍历到的结点,如果有说明此结点可能不会被执行.由于有间接寻址跳转的存在,因此可能会产生误报.

3.2 不可退出点

定义 3 对于控制流图中结点,如果结点存在 1 条通向 Exit 点有向边路径,则称此结点称为可退出点,反之称为不可退出点.

不可退出点来自于死循环和除死循环外的无 RTS/RTI 的路径.

1) 死循环.

定义 4 控制流图中的循环是一个包含满足以下性质的头结点 h 的结点集合 S .

性质 1 S 中每个结点都有 1 条通向 h 的有向路径.

性质 2 从 h 到 S 中任意结点,都有 1 条有向路径.

性质 3 除 h 之外,不存在任何从 S 外的结点到 S 内其他结点的边.

定义 5 循环的入口结点是有一个前驱位于循环外的结点.

定义 6 循环的出口结点是有一个后继位于循环外的结点.

定义 7 没有出口结点的循环称为死循环.

图 9 给出了存在死循环的函数例子. 图中控制流有两条, $1 \rightarrow 2$ 以及 $1 \rightarrow 3 \rightarrow 4 \rightarrow 3 \rightarrow 4 \dots$, 后者是一个无限长执行路径. 出现此错误的原因可能是程序员编程时在点 4 后的分支指令写错跳转标签.

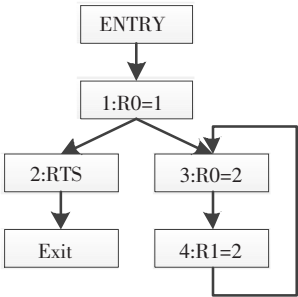


图 9 有死循环的代码

定理 死循环所有结点必然属于不可退出点.

证明 假设死循环 S 中存在可退出点 v , 由退出点定义可知, v 有 1 条通向 Exit 的有向边路径. 因为 Exit 没有后继, 所以由循环的定义可知, Exit 不属于任何循环, 即 Exit 属于循环外结点. 由出口结点定义可知, v 为 S 的出口结点. 由于 S 有出口结点, 则 S 不属于死循环, 与假设矛盾.

由定理可知, 死循环可产生不可退出点.

2) 除死循环外的无 RTS/RTI 的路径.

对于图 10 中代码, $f: R0 = 1$ 是不可退出点. 在实际运行中, 处理器执行 f 后会继续执行后续地址里的指令, 从而产生不可预料的结果.

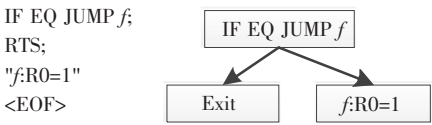


图 10 不可退出点

下面给出不可退出点检测算法.

输入: 函数 f

输出: 函数 f 中的所有不可退出点路径入口结点

```
check( Function  $f$  ) {
    work_list. push (  $f$ . Exit );
    while ( ! work_list. empty ( ) ) {
        Node  $s$  = work_list. top ( );
        for ( Node  $p$ :  $s$ . predecessors ) {
             $p$ . visited = true;
            work_list. push (  $ps$  );
        }
    }
    for( Node  $s$ : Function. nodes ) {
        if (  $s$ . visited )
            for ( Node  $ss$ :  $s$ . successors )
                if ( !  $ss$ . visited )
                    report (  $ss$ , Error. DeadLoop );
    }
}
```

算法中, 第 1 步从 Exit 点开始逆向 DFS 遍历函数 CFG. 图中未被访问的结点都是不可退出点. 第 2 步, 检测所有访问过的结点的直接后继结点 ss 是否未被访问, 如果是输出 ss . 第 2 步完成的是不可退出点路径入口的检测, 即控制流进入 ss , 则必无法正常退出函数.

3.3 循环相关缺陷

循环相关缺陷主要是循环交错, 同一指令终止, 以及最后 3 条指令不能为分支指令 (JUMP, CALL, RTS/RTI), 计数循环倒数第 3 条指令不能写计数器, 写 CURLCNTR 指令的下一条指令不能是 IF NOT LCE 等.

循环交错程序可以编译运行, 不过却可能不是程序员本意.

对任意 2 个循环 P_1, P_2 , 有
 $(P_2. head > P_1. head) \wedge$
 $(P_2. head < P_1. tail) \Leftrightarrow P_1, P_2$ 交错,
 $P_2. tail = P_1. tail \Leftrightarrow P_1, P_2$ 同一地址终止.

3.4 延迟分支相关缺陷

1) 可改成延迟分支.

非延迟指令 JUMP、CALL、RTS/RTI 在取指的时候不会执行后续的 2 条指令, 而是执行 2 条

NOP. 可以在适当条件下将非延迟指令改写成延迟指令,以避免流水线 2 个时钟的浪费.

2) 延迟分支使用 JUMP、CALL、RTS/RTL.

延迟分支先被执行,循环指令在流水线上展开后,会将 JUMP 的目标地址覆盖掉,从而执行不了跳转.

3) 禁止使用 IDLE.

控制流执行到 IDLE 会一直等待,直到有中断发生. 因此如果在延迟分支中使用 idle 指令,那么处理器一直处于空闲状态直到有中断发生.

3.5 内存相关缺陷

3.5.1 DAG 寄存器读写

以下 2 种指令处理器会执行,但是会产生不正确的结果.

1) 保存 DAG 寄存器到内存,使用同一个 DAG 间接寻址,指令会写错误的数据或者更新错误的索引寄存器.

2) 从内存加载 DAG 寄存器,使用同一个 DAG 间接寻址,指令会加载 DAG 寄存器或者更新索引寄存器.

3.5.2 寄存器的 2 个写操作

如果有 2 个对同一寄存器的写操作在一个周期内发生,只有高优先级的写操作会真正发生.

4 过程与结果分析

工具使用 Java 编写,可以运行在支持 Java 的任何平台上. 实验环境: JDK 1.7, Win 7 SP1 64 bit, Intel Core2 Q9300, 8 G RAM.

4.1 静态分析测试

本文对 OpenSSL 1.0.0.e 中的对称加密算法, DSPstone^[7], bzip2 1.0.6, gzip 1.2.4 及 libquantum 1.1.0 进行了静态分析测试. 编译器为 ADI Visual DSP + +5.0 中的 cc21k.exe,编译参数使用-S 输出汇编代码,其他使用默认值. 静态分析测试结果如表 2 所示,数据流分析结果未列出.

由表 2 中数据可以看出,路径简化在此次实验中效果并不明显,简化率 $452/165\ 167 = 0.27\%$. 间接跳转中 CASE 语句引起的跳转表全部被识别,验证了特征识别法的准确性,但仍有部分间接跳转无法处理. 经检查,无法处理的间接跳转都类似以下代码片段:

```
.P54L14;  
R4 = DM(11,I4);  
R2 = DM(10,I4);  
I12 = R2;  
R2 = I6; I6 = I7; JUMP(M13,I12)(DB);
```

DM(I7,M7) = R2; DM(I7,M7) = PC;

JUMP(PC, .P54L16);

其中的 JUMP(M13,I12) 确实静态无法确定跳转地址.

表 2 静态分析测试结果

文件名	指令数	延迟分支	路径简化	间接跳转	耗时/s
openssl crypto	100 438	1 720	136	293(0)	663.9
DSPstone	12 418	367	223	114(0)	6.5
bzip2	25 925	1 051	32	200(20)	33.0
libquantum	13 024	940	2	176(3)	5.0
gzip	13 362	676	59	127(4)	11.0

4.2 缺陷检测测试

缺陷检测测试使用了 ADI 公司官方给出的示例程序集 ADSP-21065L Audio Demos^[6]和本文手工编写的缺陷测试集作为样本. 由于后者测试全部通过,这里不列出. Audio Demos 检测结果如表 3 所示. 由于篇幅所限,表 3 中数据为各个项目的第 1 个子项目结果. 这些手工编写的汇编项目都使用了共同的内存和寄存器初始化函数,检测工具从中断向量表里的各个中断开始进行过程间分析. 表中括号数值为误检数. 误检率 = $36/515 = 7.0\%$. 由于汇编代码人工审查代价比较高,因此本文没有计算漏检率.

检测未发现循环缺陷、内存缺陷和未初始化寄存器缺陷. 检测结果中的死循环,都是类似以下等待用户操作的代码:

```
wait_forever;  
call wait_flag1;  
bit tgl1 ustat1 0x3F;  
dm(IOSTAT) = ustat1;  
jump wait_forever;
```

未初始化寄存器缺陷仅在 WaveMusic Demo 中发现 1 处,是 WaveMusic.asm 中 72 行的“R1 = DM(I0, 1)”,经人工审查,此处 I0 确无代码对它进行初始化.

无用赋值的误检来自于两方面:1) 程序未考虑循环数据缓冲区操作中会隐式使用基址寄存器,所以把如 Digital Delay Effects 中的 ADT&Slapback.asm 119 行“B6 = w;”基址寄存器 B6 的赋值当作无用赋值了;2) 分析时是每个中断分别进行分析的,没有考虑中断处理间的关联. 在测试中的代码里,都是 reset 中断里初始化,然后无限循环等待另外一个中断. 由于没作中断间分析,所以 reset 中断里的初始化就会被误报.

表 3 Audio Demos 缺陷检测结果

项目	循环缺陷	未使用标号	死循环	无用赋值	未初始化寄存器	延迟分支缺陷	内存缺陷	行数	耗时/ms
Chorus Effects	0	12	1	3(2)	0	18	0	2 027	662
DeTune Effects	0	15	1	9(8)	0	15	0	1 981	635
Digital Delay Effects	0	11	1	2(1)	0	15	0	1 965	613
Digital Reverberation Effects	0	27	1	6(5)	0	25	0	2 061	1 012
Reverb Building Blocks ToolKit	0	11	1	2(1)	0	14	0	1 816	617
Dynamic Range Control Effects	0	13	1	1	0	17	0	1 917	615
Flanger	0	12	1	3(2)	0	21	0	2 019	633
Guitar Distortion	0	17	1	3(2)	0	17	0	2 232	657
Phaser Effect	0	14	1	5(4)	0	13	0	2 161	625
Pitch Shifter	0	9	1	1	0	8	0	1 985	610
Rotating Speaker	0	15	1	3(2)	0	17	0	1 950	629
Tremolo	0	14	1	2(1)	0	17	0	1 893	636
Vibrato	0	12	1	3(2)	0	20	0	2 003	640
WaveMusic Demo	0	10	1	8(5)	1	12	0	1 841	673
Digital Filters	0	11	1	5(1)	0	11	0	1 787	615

5 结 论

1) 汇编代码的控制流静态分析中,间接跳转和延迟分支的处理是难点. 本文在 ADSP SHARC 分析中采用编译器特征方法识别间接跳转,使用结点克隆处理延迟条件分支.

2) 过程间分析技术可以提高代码静态分析精度,本文使用函数内联实现过程间分析. 对于递归的函数记录调用序列限制内联深度.

3) 定义并计算 MUST_UD 链检测代码里未初始化变量. 定义不可退出点检测由于误写跳转标签引起的死循环及无返回指令的函数. 对于循环、延迟分支、内存相关缺陷按照语言规范进行检测.

参考文献

[1] BERMUDO N, KRALL A, HORSPOOL N. Control flow graph reconstruction for assembly language programs with delayed instructions[C]//Proceedings of the 2005 Fifth IEEE International Workshop on Source Code Analysis and Manipulation(SCAM' 05). Washington, DC: IEEE Computer Society, 2005: 107 – 118.

[2] KASTNER D, WILHELM S. Generic control flow reconstruction from assembly code[C]//Proceedings of

the Joint Conference on Languages, Compilers and Tools for Embedded Systems: Software and Compilers for Embedded Systems. New York, NY: ACM, 2002: 46 – 55.

[3] SCHLICH B, LOLL J, KOWALEWSKI S. Application of static analyses for state space reduction to microcontroller assembly code[C]//Proceedings of the 12th International Conference on Formal Methods for Industrial Critical Systems. Berlin, Heidelberg: Springer-Verlag, 2008: 21 – 37.

[4] AHO A V, ULLMAN J D. Principles of compiler design thoery[M]. Boston: Addison-Wesley, 1980.

[5] 张广梅, 李晓维. 数据流相关软件故障的静态检测[J]. 计算机辅助设计与图形学学报, 2005, 17(11): 2477 – 2483.

[6] ADI. Code example [EB/OL]. http://www.analog.com/en/processors-dsp/sharc/products/code-examples/21065L_audio_demos/resources/fca.html, 2011.

[7] ZIVOJNOVIC V, VELARDE J M, SCHLAGER C, *et al.* DSPstone: a DSP-oriented benchmarking methodology [C]//Proceedings of the International Conference on Signal Processing and Technology (ICSPAT' 94). Dallas, [s. n.]: ICE, 1994: 715 – 720.

(编辑 张 红)