CN43-1258/TP
ISSN 1007-130X

计算机工程与科学
COMPUTER ENGINEERING & SCIENCE

2012年第34卷第2期
Vol. 34, No. 2, 2012

文章编号:1007-130X(2012)02-0073-05

GCC编译器中间代码层控制流扩充研究*

Research on the Expanding Intermediate Code for Control Flow Checking Based on GCC

何 涛[1],周会平[2],贾丽丽[2],王发鸿[1]

HE Tao[1], ZHOU Hui-ping[2], JIA Li-li[2], WANG Fa-hong[1]

(1. 国防科学技术大学继续教育学院,湖南 长沙 410073;
2. 国防科学技术大学计算机学院,湖南 长沙 410073)

(1. School of Further Contin Education, National University of Defense Technology, Changsha 410073;
2. School of Computer Science, National University of Defense Technology, Changsha 410073, China)

摘 要:本文首先对CFCSS(控制流检错算法)进行了介绍,对GCC编译器的运行流程进行了简要分析,再次给出了在GCC编译器中扩充CFCSS算法的具体方法,最后通过故障注入实验对扩充后的GCC进行了有效性验证。实验表明,扩充了CFCSS算法的GCC编译器所编译的程序在运行过程中具有控制流检错能力。这为我们下一步的故障定位和故障恢复提供了有力的支持,为解决星载计算机的运行故障奠定了基础。

Abstract: The paper introduces for CFCSS, briefly analyzes the running routine of the GCC compiler, gives the concrete methods for how to extend the CFCSS algorithm in the GCC compiler and then compiles a single C language program in the extended GCC. Finally, we perform the CFCSS algorithm's valid verification by fault injection experiments. The experiment shows the new program can check the error of the control flow when it runs, which is compiled by the GCC compiler after extending the CFCSS algorithm. The above conclusion gives great support to positioning and recovering the faults in the next step and lays a good foundation for solving the computer running faults in the space.

关键词:GCC;中间层代码;优化;控制流检错算法

Key words: GCC; intermediate code; optimization; control flow error detection algorithm

doi: 10.3969/j.issn.1007-130X.2012.02.014

中图分类号:TP314 **文献标识码**:A

1 引言

在外太空运行的计算机受到来自宇宙射线的辐射,其运行环境非常恶劣,故障也比在地面高出许多,再加上维修困难,因而对计算机的可靠性提出了非常严格的要求。本文通过对GCC编译器总体结构的简要分析,并结合CFCSS算法自身的特点,提出了在GCC编译器中扩充CFCSS算法的方案,从而有效解决了软件在太空中运行时发生控制流错误时的故障检测问题,并且为下一步的故障定位和故障恢复提供了有效的依据。

2 CFCSS算法介绍

CFCSS算法是采用标签分析方法检测控制流错误的典型算法。其基本原理是:在编译时将程序划分为基本块,并为每个基本块分配唯一的静态标

* 收稿日期:2010-03-24;修订日期:2010-06-10
通讯地址:410073 湖南省长沙市国防科学技术大学继续教育学院二队
Address: School of Further Education, National University of Defense Technology, Changsha, Hunan 410073, P. R. China

签，程序运行时根据当前控制流计算出一个动态标签，并比较动态标签和静态标签是否一致，不匹配则说明控制流出现错误。

算法将程序划分为基本块后，为每个基本块 b_i 分配不同的静态标签 s_i，并基于程序控制流图为每个基本块 b_i 计算出一个标签差量 d_i，$d_i = s_i \wedge s_j$（s_i 为当前块 b_i 的静态标签，s_j 为 b_i 前驱块的静态标签，下文中出现如 s_i 和 s_j 不特别说明则和此处代表意思相同，不再赘述）。在入口块处将动态标签 G 初始化为 $G=s_1$，差量标签初始化为 $d_1=0$，此后，算法在每个基本块 b_i 开始处插入如下两条指令：

$$G = G \wedge d_i \tag{1}$$

$$br\ G \neq s_i\ \text{error} \tag{2}$$

第一条指令用于更新动态标签，第二条指令用于检测当前动态标签是否和静态标签相等，如果不相等则报错。

虽然上述过程通过为每个基本块分配静态标签并插入检测指令，可以检测出控制流中的非法分支，但是还有一种特殊情况需要考虑，就是存在多扇入结点的情况。当基本块 B 有两个或者两个以上的前驱块时 B 的差量标签便无法确定，此时算法引入运行时调整标签 D，在多扇入结点开始处，更新后的动态标签 G 还需要与 D 进行异或运算。调整标签 D 的确定方法为，首先从多扇入节点 b_i 的前驱节点集合 $\{b|b_1,b_2,b_3,\cdots,b_n\}$ 中任选一个节点 b_i 作为基准节点，其次将基准节点的调整标签初始化为 $D=0$，最后将其他前驱节点 b_j 的调整标签 D 设为基准节点 b_i 的静态标签和 b_j 的静态标签异或的结果。多扇入节点 B 的差量标签 d 则必须取基准节点的静态标签和 B 的静态标签做异或。当前驱节点的调整标签 D 都确定了，在更新动态标签 G 后让 G 再和调整标签 D 做异或，最后比较 G 是否和当前块的静态标签值相等即可。

3　GCC 编译器运行流程分析

由于 CFCSS 算法是针对基本块进行操作的，因此我们需对 GCC 编译器的中间层代码进行认真分析，尤其是和控制流图创建相关的代码。GCC 编译器执行流程如图 1 所示。

通过上图可以看出 GCC 编译器的执行流程如下：

（1）前端处理（预处理、词法分析、语法分析、语义分析等）将源程序转化为中间代码树 GENERIC；

（2）c_genericize 函数将 GENERIC 树又转化为高级 GIMPLE 树；

（3）execute_pass_list(all_lowing_passes)函数将高级 GIMPLE 转化为低级 GIMPLE 树；

（4）优化器 execute_pass_list(pass_early_local_passes_sub)生成 SSA(Static Single Assignment，简称 SSA)，在此优化过程中会划分基本块并且创建控制流图；

（5）SSA 通过 RTL 生成器生成 RTL 树，并进行优化；

（6）RTL 最后经 INSN 生成器转化为 INSN 并优化后进入 GCC 后端，生成相应平台的可执行文件或者汇编代码。

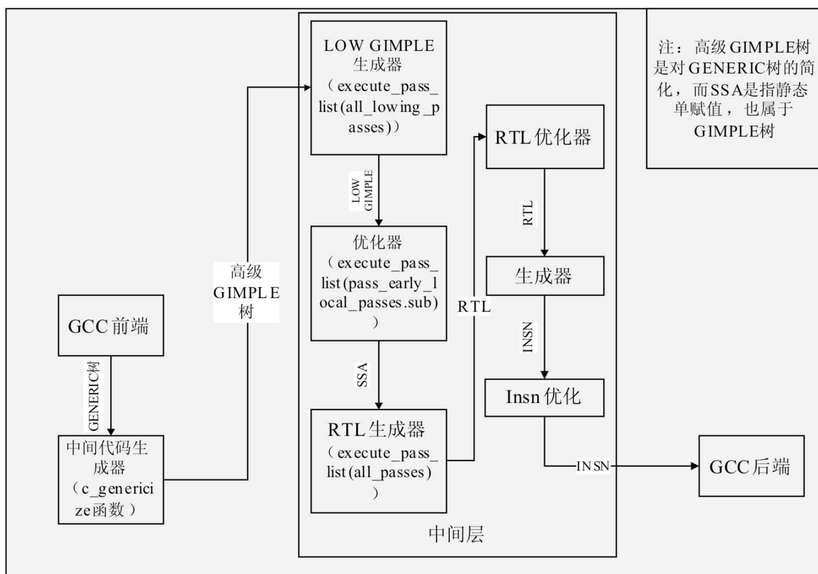

图 1　GCC 编译器执行流程图

4　CFCSS 算法在 GCC 编译器中的实现

4.1　插入点的选择

经过上面的算法分析可知，CFCSS 是针对基本块进行操作的，所以要将该算法扩充到 GCC 编译器中必须选择合适的插入点。经过上面对 GCC 编译器执行流程的分析可知，GCC 编译器中主要包含下面几种不同的中间语言表示形式，分别为 GENERIC 树、GIMPLE 树（包含高级 GIMPLE 和低级 GIMPLE 以及 SSA）和 RTL 树。这三种中间语言都是与前端语言无关的。其中 GENERIC 树是将前端语言直接翻译过来后形成的中间树，而 GIMPLE 则是简化了的 GENERIC 树的集合，在 GENERIC 转化为 GIMPLE 的过程中将 GENERIC 树中比较复杂的语句都转化为了多个比较简单的语句，其中的计算结果用临时变量来保存。RTL 树是通过低级 GIMPLE 树转化而来的，将低级 GIMPLE 树转化为 RTL 树的主要目的是进行优化。

由于 CFCSS 算法是针对控制流进行操作的，而控制流的创建是在形成 GIMPLE 之后，因此在 GENERIC 树上扩充算法不必考虑。如果靠后选择则需要构造 RTL 树，而 RTL 树的形成是在优化遍中完成的，在优化遍中存在函数的嵌套调用和大量的函数循环调用，这对算法的正确性检测和跟踪带来极大的不便。对控制流图的创建是在形成低级 GIMPLE 树之后通过函数 execute_pass_list (pass_early_local) 实现的，通过对 GCC 运行流程以及三种中间语言各自特点的分析，结合 CFCSS 算法自身特点，CFCSS 算法的插入点应该选择在 execute_pass_list (pass_early_local) 之后，RTL 生成器之前，即图 1 中的优化器和 RTL 生成器之间。

4.2　算法实现

对于算法中所涉及到的语句，可以将其构造成低级 GIMPLE 树的格式来插入每个基本块的语句链表中，算法实现源程序函数调用关系如图 2 所示。

图 2 中所绘函数均为实现 CFCSS 算法所新编的函数，在 insert_cfcss 中调用的 GCC 编译器自带的函数在上图中没有绘出。insert_cfcss 为插入 CFCSS 算法的入口函数，通过 insert_cfcss 调用其子函数来实现在 GCC 编译器中插入 CFCSS 算法。下面分别就每个函数的功能做如下介绍：

（1）make_label_string 函数的作用是根据所给的字符创建标签名称，其子函数 num_to_string 的作用是将参数传入的数字转化为字符，并存入指定的字符数组中，比如数字 123 经过 num_to_string 处理后则转化为‘1’‘2’‘3’。

（2）create_gimple_modify_node 函数的作用则是创建一个赋值表达式，在表达式的左侧为一个无符号的变量，表达式右侧为一无符号的常数，比如 $s_1=1$。insert_cfcss 调用此函数的目的是初始化每个基本块的静态标签，并且将入口块的差量标签 d_1 初始化为 0。

（3）create_balance_label_node 函数的作用是将当前块的静态标签和前驱块的静态标签做异或，并将其值赋值给当前块的差量标签。例如 $d_i=s_i \wedge s_j$ 这样的表达式则由该函数创建，如果当前块为多扇入块，则将边存储数组中下标为 0 的那条边的源块默认为基准块。insert_cfcss 中调用该函数的目的是计算每个基本块的差量标签。

（4）initial_capital_g 函数的作用则是对动态标签 G 进行初始化，使其值等于入口块的静态标签值，即 $G=s_1$。其调用的子函数 create_var_decl_node 的作用为创建一个变量声明，用在此处是为

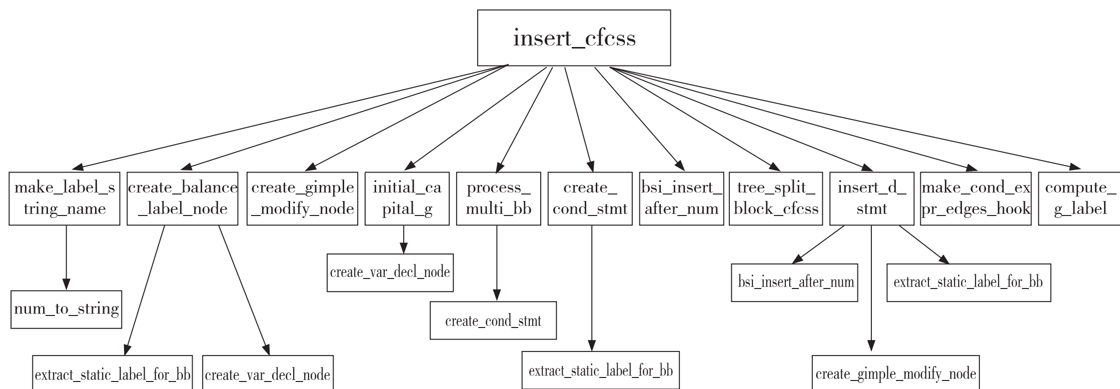

图 2　CFCSS 算法实现函数调用关系图

了声明动态标签 G。

(5)compute_g_label 函数的作用是将动态标签 G 和差量标签 d 做异或,并将异或的值赋值给 G,即 $G=G \wedge d_i$。

(6)create_cond_stmt 函数的作用是创建一个条件语句,用来判断当前的动态标签是否和当前块的静态标签相等,即 $G==s_i$。

(7)insert_d_stmt 函数的作用是计算调整标签 D 的值,即 $D=0$ 或者 $D=s_i \wedge s_j$(s_i 为前驱块的静态标签,s_j 为基准块的静态标签),并将形成的表达式语句插入前驱块的语句链表中。其中调用 bsi_insert_after 函数的作用是将语句插入基本块的语句链表中的某条语句后。extract_static_label_for_bb 函数的作用是从入口块的语句链表中提取基本块 b_i 对应的的静态标签 s_i。此处调用函数 create_gimple_modify_node 是为了对调整标签 D 做初始化,即 $D=0$。

(8)process_multi_bb 函数的作用是创建 $G=G \wedge D$ 和条件检测语句 $G==s_i$,并将创建好的两条语句加入多扇入块的语句链表中。

(9)tree_split_block_cfcss 函数的作用是将基本块在条件检测语句处拆分为两个基本块。

(10)make_cond_expr_edges_hook 函数是一个钩子函数,其作用是调用 GCC 自带的函数 make_cond_expr_edges,从而为拆分后的基本块重新创建边,从而达到更新控制流图的目的。

在算法实现过程中 insert_cfcss 函数为扩充 CFCSS 算法的入口函数,该函数的执行过程如下:

(1)循环调用 create_gimple_modify_node 函数将 $s_1=1, s_2=2, \cdots, s_n=n$ 插入入口块的语句链表中。

(2)循环调用 create_gimple_modify_node 函数和 create_balance_label_node 函数将 $d_1=0$ 插入入口块的语句链表的第 n(n 为控制流图中基本块的数量)条语句后,将 $d_i=s_i \wedge s_j$ 插入其余块的第一条语句。

(3)循环调用 initial_capital_g 函数和 compute_g_label 函数将 $G=s_1$ 插入入口块语句链表的第 $n+1$ 条语句后,将 $G=G \wedge d_i$ 插入其余块的第二条语句。

(4)调用 create_empty_bb 函数创建一个不包含语句的空块,并命名为 error_bb,然后在 error_bb 中插入一个表示出错后跳入位置的标签 error_label。

(5)循环调用 create_cond_stmt 函数,将条件检测语句 $G==s_i$ 和 then_label 加入单扇入块的第三和第四条语句。

(6)循环调用 insert_d_stmt 将 $D=0$ 或者 $D=s_i \wedge s_j$(s_i 为基准块的静态标签,s_j 为除基准块外其余前驱块的静态标签)插入多扇入块的前驱块的语句链表中,如果其前驱块为单扇入块并且不是入口块,则将 D 的赋值语句插入其前驱块的第五条语句;如果其前驱块为入口块,则将 D 的赋值语句插入第 $n+2$ 条语句后;如果其前驱块为多扇入块,则将 D 的赋值语句加入前驱块的第二条语句后。

(7)循环调用 process_multi_bb 函数将 $G=G \wedge D$ 和 $G==s_i$ 以及 then_label 加入当前多扇入块的第三、第四和第五条语句。

(8)调用 tree_split_block_cfcss 函数将每个插入条件检测语句的基本块在 then_label 语句处分解为两个基本块。

(9)调用 make_cond_expr_edges_hook 函数为分解后的基本块创建新的边,并用 cleanup_dead_labels 函数将控制流图中无用的标签语句删除。

通过上面的九个步骤 CFCSS 算法已经在 GCC 编译器中完成扩充。

5 实验测试

为了评价 CGCC(扩充了 CFCSS 算法后的 GCC 编译器)的性能,采用故障注入的方法验证 CGCC 的控制流检测能力。故障注入实验使用 SimpleScalar 模拟容错程序运行的底层硬件平台——SimpleScalar/x86 体系结构。故障注入实验是在 Intel x86 - Fedora Linux 8.0 下进行。

实验过程是分别通过 GCC 和 CGCC 对示例程序进行编译,得到 X86 格式的具有容错能力的汇编代码,将容错汇编代码汇编后即得到目标代码。最后在目标代码中利用故障注入工具随机注入故障,检验容错算法的故障检测能力和算法扩充的正确性。

故障注入分为静态注入和动态注入两部分。静态注入指在程序运行之前向程序代码段注入由于一次位翻转所导致的控制流错误,包括把非程序控制指令改成程序控制指令,修改程序控制指令的目标地址、操作码或操作数等。动态注入指在程序运行过程中,在随机的时刻随机翻转 PC 寄存器中的一位。根据对程序的不同影响把注入的故障分为五大类:

(1)Correct:注入的故障没有影响程序的正常

执行;

(2)Exception:注入的故障被 SimpleScalar 检测出,例如访问非法的内存地址;

(3)Timeout:注入的故障导致程序不能在指定的时间内结束执行;

(4)Wrong:程序正常退出但结果错误;

(5)Detected:检测算法成功检测到的故障。

实验使用 MM(矩阵乘法)、BS(冒泡排序)和 PI(蒙特卡罗法求 π)三个程序作为测试用例。

通过上述的实验方法进行实验,每个版本的程序运行 1 000 次,每次注入一个故障,静态注入和动态注入的比例为 4 ∶ 1,得到的实验数据如表 1 所示。

表 1　故障注入实验数据

	GCC			CGCC		
	MM	BS	PI	MM	BS	PI
Correct	134	340	211	380	483	408
Exception	414	351	382	252	223	250
Timeout	177	74	98	47	19	27
Wrong	275	235	309	151	100	102
Detected	0	0	0	170	175	213
FailRate(%)	27.50	23.50	30.90	15.10	10.00	10.20
Overhead(%)	0	0	0	12.60	31.22	28.82

表 1 第 8 行的失效率(FailRate)是导致错误结果的故障所占比例。第 9 行的性能开销(Overhead)是实现控制流检测版本的 GCC 多消耗的执行周期数与原始 GCC 版本所需周期数的比值。

从表 1 可以看出,CGCC 对这三个示例程序的平均性能开销为 12.6%、31.22%和 28.82%,失效率分别为 15.10%、10.00%和 10.20%,失效率明显低于未扩充 CFCSS 算法的 GCC,并且三个程序的平均性能开销仅为 24.21%,在可承受范围内。故障注入实验表明,CFCSS 算法在 GCC 编译器中间代码层的扩充是成功的,经过 CGCC 编译后的目标代码具有控制流错误检测能力。

6　结束语

本文通过对 CFCSS 算法的研究和对 GCC 编译器的分析,提出在 GCC 编译器的中间代码层扩充 CFCSS 算法,并通过文中介绍的方法实现了 CFCSS 算法在 GCC 编译器中间代码层的扩充,最后用故障注入实验对 CGCC 编译器编译出的目标代码进行了验证。验证结果表明,通过扩充的编译器编译出的目标代码具有控制流检测能力,在性能开销可接受的范围内 CGCC 的控制流错误检测的失效率明显低于原始的 GCC。

由于 CGCC 并没有过程间的控制流检测能力,在下一步的工作中尝试解决过程间控制流错误的检测。

参考文献:

[1]　GCC Documents Version 4.3.0[EB/OL].[2009-12-20]. http://GCC.gnu.org/software/GCC/GCC-4.3.0.

[2]　周永彬. 单粒子效应下星载处理平台的容错设计与测试验证:[博士学位论文][D]. 长沙: 国防科技大学,2008.

[3]　李建立. 空间辐射环境下软件实现的硬件故障检测技术研究:[硕士学位论文][D]. 长沙:国防科技大学,2008.

[4]　Gros X E, Lowden D W. A Probabilistic Pproach to Data Fusion Non-Destuctive Structural Integrity Assesment[D]. University of Texas at El Paso,2005.

[5]　Perry F,Mackey L W,Reis G A,et al. Fault-Tolerant Typed Assembly Language[C]// Proc of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation,2007:42-53.

[6]　宫经刚,华更新. 星载容错计算机故障分析及总线级故障注入[C]//空间电子学学术年会论文集,2006:637-643.

[7]　Goloubeva O. Software Implemented Hardware Fault Tolerance[M]. Sprinler Press, 2006:124-136.

[8]　朱鹏. 星载 SAR 控制软件故障注入技术研究:[硕士学位论文].[D]. 中国科学院研究生院,2004.

[9]　Fox D B, Frail D A, Price A P. The Afterglow of GRB 050709 and the Nature of the Short-Hard γ -ray Bursts[J]. Nature, 2005, 437(1):845-850.

[10]　Rufenacht H, Hiemstra D M, Ronge R. Single Event Upset Characterization of the ESP603 Single Board Space Computer with the PowerPC603 Processor Using Proton Irradiation[J]. Radiation Effects Data Workshop, 2005, 11(15):65-69.

何涛(1982 -),男,陕西渭南人,硕士生,研究方向为软件容错和编译技术。**E-mail**:hetao0016066@yahoo.com.cn

HE Tao, born in 1982, MS candidate, his research interests include software fault tolerance, and compiler technology.

周会平(1972 -),男,湖北天门人,博士,副教授,研究方向为自然语言处理和编译技术。**E-mail**:icent@qq.com

ZHOU Hui-ping, born in 1972, PhD, associate professor, his research interests include natural language processing, and compiler technology.