

PowerPC 汇编程序的反编译研究

刘延昭,赵银亮,武万杰

(西安交通大学 计算机科学与技术系,陕西 西安 710049)

摘要:反编译技术将二进制程序或汇编程序转换成可读性较好的高级语言代码,在代码理解、代码维护和代码安全验证等方面具有重要作用。文中介绍了一种基于 PowerPC 汇编程序的反编译软件框架及其关键技术。该软件框架主要包括由汇编程序加载、指令系统的语义描述和汇编指令的解码所组成的前端,由数据流分析、类型分析和控制流分析所组成的中间端以及负责代码生成的后端。采用的关键技术有 switch 语句翻译,代码复制消除 goto 语句和指令习语(instruction idiom)翻译等。实验结果表明,反编译生成的高级语言程序在结构、可读性等方面都有所增强,对于辅助代码理解有指导意义。

关键词:PowerPC 汇编程序;结构恢复;习语分析;反编译框架

中图分类号:TP311

文献标识码:A

文章编号:1673-629X(2015)08-0001-08

doi:10.3969/j.issn.1673-629X.2015.08.001

Decompile Research of PowerPC Assembly Program

LIU Yan-zhao,ZHAO Yin-liang,WU Wan-jie

(Department of Computer Science and Technology,Xi'an Jiaotong University,
Xi'an 710049,China)

Abstract:Decompilation is the process of converting binary program or assembly program into high level code with good readability, which plays a vital role in code understanding, code maintenance and code safety verification. In this paper, introduce a decompilation framework based on PowerPC assembly program and its main techniques. The framework consists of the frontend, which includes the loading of the assembly programs, the semantic description and the decoding of the instruction set, the middle end, which includes data flow analysis, type analysis and control flow analysis, and the back end, which is responsible for code generation. The main techniques consist of translation of switch sentence, goto sentence of elimination by code duplication and translation of instruction idiom. Experimental results show that the generated high level code improves in both structure and readability, there's a guiding significance in code understanding.

Key words:PowerPC assembly program; structure recovery; instruction idiom analysis; decompilation framework

0 引言

反编译是软件逆向工程的重要组成部分,其核心是将低级代码转换成可读性较好的高级代码,在代码理解、代码维护、代码安全验证等方面都有着重要作用。在代码理解方面,反编译产生的高级代码更容易让人理解程序的逻辑和算法;在代码维护方面,当源码不可用时,仍然可以通过反编译手段,重新构建软件;在代码安全验证方面,反编译技术可以用来验证编译器产生的目标代码,检查软件中是否存在恶意代码等。随着嵌入式系统的迅速扩张,系统的安全性、扩展性越

来越重要,而反编译有望发挥重要作用。

目前,国内外有许多反编译框架。DCC^[1]利用基本编译的技术,如数据流分析、类型分析和死代码删除等研究反编译问题,为反编译的研究奠定了理论和技术基础。Hex-Rays Decompiler^[2]是基于流行的交互式反汇编工具 IDA(Interactive DisAssembler)^[3]开发的反编译插件。目前,能够将 x86 可执行程序 and ARM 可执行程序翻译成可读性较好的类 C 程序。IDC(Interactive DeCompilation)^[4]是一款交互式的反编译器,在反编译过程中,允许用户提供必要的指导信息,去除

收稿日期:2014-09-25

修回日期:2014-12-26

网络出版时间:2015-07-21

基金项目:陕西省科技计划项目(2014K05-04)

作者简介:刘延昭(1988-),男,硕士研究生,研究方向为逆向工程与反编译;赵银亮,教授,博士生导师,研究方向为程序语言及编译系统、并行计算与机器学习。

网络出版地址: <http://www.cnki.net/kcms/detail/61.1450.TP.20150721.1448.052.html>

反编译过程中的二义性和模糊性,增强高级代码的精确性和可读性。SecondWrite^[5]将 x86 二进制程序转换成 LLVM IR,利用 LLVM 编译器的优化遍译,重新优化程序,生成效率更高的二进制程序或者可读性较好的类 C 程序。Dagger^[6]是基于 LLVM 的反编译框架,可以用来进行程序重写、二进制转换翻译、指令集模拟和反编译的研究。Retargetable Decompiler^[7]利用指令系统描述语言 (ISAC) 来建模不同的体系结构,然后通过语义提取工具,将二进制程序转换成 LLVM IR,动态生成反编译器,实现可重定向的反编译器。Boomerang^[8]是一款开源的,支持多种二进制文件格式和多种体系结构的反编译软件,其前端的灵活设计使得扩展新体系结构和新文件格式相当容易,后端针对统一的中间表达式做反编译和代码生成。

文中提出通过扩展 Boomerang 反编译器,构建基于汇编程序的反编译器。扩展 Boomerang 的加载器,使其能够加载汇编程序;扩展 Boomerang 的解码器,使其能够解码汇编指令,并通过指令习语分析,提高中

间代码的抽象度,从而提高生成代码的可读性;在控制流分析方面,提出控制流图的重构算法,如 switch 语句的规整和代码复制消除 goto 语句等。最终,优化 PowerPC 汇编程序的反编译器的输出结果。

1 反编译框架

现有的绝大多数反编译框架都是基于昆士兰大学的 C. Cifuentes 等的研究框架^[1]提出的。该反编译框架由加载器、解码器、数据流分析、类型分析、控制流分析和代码生成六部分组成。加载器根据汇编的文法,解析汇编程序;解码器依据每条汇编指令的语义,将其分解成一系列微操作,并构造 RTL (Register Transfer Language) 中间表达式;以 RTL 为中间表达式的程序,经过数据流分析、类型分析和控制流分析,被转换成结构良好的高级中间表达式,具备循环和分支等高级结构;最终,通过代码生成类 C 程序。

基于汇编指令的反编译软件框架如图 1 所示。

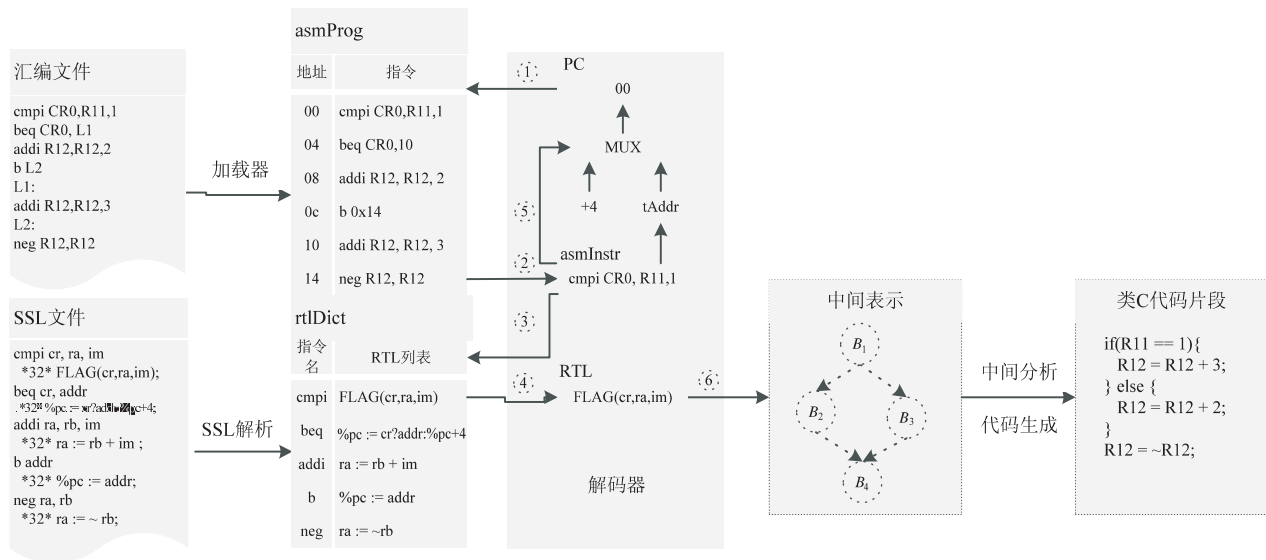


图 1 反编译框架

图 1 中基于汇编的反编译器框架由三大部分组成:前端、中间端和后端。前端由加载器、SSL (Semantic Specification Language)^[9]解析单元和解码器构成;中间端由数据流分析、类型分析和控制流分析构成;后端主要包括代码生成。

(1) 前端部分。

加载器根据汇编语言文法,将汇编程序读入到反编译程序中,并且组织成相应的数据结构,如符号表、符号地址表、函数地址表、跳转链表、指令链表等。

SSL 解析单元是 SSL 语言的词法和语法分析工具,将 SSL 文件中描述的寄存器信息、标志位函数和指令解码信息等组织成相应的数据结构。其中,最重要的数据结构如图 1 中的 RTL 词典 rtlDict,该词典在

解码阶段使用。

首先,解码器根据 PC 值,从 asmProg 中获取一条指令并存储到 asmInstr 中;接着,解码器以指令 asmInstr 的操作码为关键字,到 rtlDict 字典中查询其语义描述集合—RTL 列表,并将 RTL 列表存储到 RTL 中;其次,解码器根据指令 asmInstr 的类型去更新 PC 值,如果是顺序指令,则将 PC 修改成 PC+InstrLength (asmInstr),其中 InstrLength (asmInstr) 为指令 asmInstr 的长度;否则,解码器要根据符号地址去符号地址表中查询其具体地址 tAddr,并将其赋给 PC 作为下一次取指令的地址;最后,解码器根据控制流重构算法^[10]实时地构造出汇编程序的控制结构。通过上述四步,最终将平坦的汇编指令序列转换成具有控制结构的中间表

达式。

(2) 中间端部分。

中间端是反编译流程中最重要的部分,该部分主要包括数据流分析、类型分析和控制流分析。数据流分析^[8]通过活跃变量分析来消除无用代码,确定被调用过程体的参数和返回值等,通过到达定值分析,进行表达式传播和表达式组合等;类型分析^[11]从机器指令的操作码、库函数的签名和常数的值等多处获取基本类型信息,然后利用类型推导规则推导其他变量的基本类型,从而使得生成的高级代码的可读性更强;控制流分析根据结构化算法^[12],将控制流图中的节点按照其在控制流图中的位置分成不同的类别,如顺序代码块、分支代码块和循环代码块等。

文中 2.1 节的 switch 控制流图的规整以及 2.2 节的通过代码复制消除 goto 语句的关键技术都是在控制流图上,利用一定的规则,将其转化成更加规整的流图,以便生成结构更加良好、可读性更高的高级代码。

(3) 后端部分。

后端^[1]通过遍历控制流图,依据每个基本块所属高级结构的类型,分别生成顺序、分支和循环的代码,并最终输出。整个代码生成过程是自顶向下,从程序到过程体,到控制流图,到基本块,到语句,到表达式,逐层深入的过程。

当然,为了进一步提高程序的可读性,可能要对某些地址值进行重新命名,例如将全局地址命名成带有 global 前缀的名字,将局部地址命名为带有 local 前缀的名字,将参数命名成带有 var 前缀的名字等;为了回溯汇编代码,以便进行跟踪比对,可以在高级程序的分支或循环结构处、特殊操作处标注汇编指令的地址范围等;在输出 switch 结构时,需要将跳往同一基本块的多个 case 分支进行合并,从而有效减少不必要的 goto 语句。

2 反编译关键技术

传统的反编译关键技术研究主要集中在数据流分析、类型分析和控制流分析等方面,而忽略了解码阶段的优化。2.1 节针对 IDA 的反汇编结果没有有效地构建出 switch 语句的控制流图这一现象入手,通过分析反汇编结果,重构跳转表,最终在解码阶段成功地构造出 switch 语句的控制流图,对于后续分析的完整性具有重要意义。同时,为了得到可读性较好的 switch 语句,在控制流分析前,需要对程序的控制流图进行规整,其算法将在 2.1 节给出。2.3 节分析指令习语,在解码阶段有效地检测并还原指令本质含义,为后续分析提供抽象程度更高的中间表达式,有利于生成可读性更高的代码。Goto 语句的使用往往使程序的逻辑结构混乱不清^[13],2.2 节以程序控制流图为基础,分析反编译过程中 goto 语句对应的子图模式,构造相关的子图模式检测规则,通过代码复制的手段,消除相应的子图,减少生成代码中的 goto 语句数目。

2.1 Switch 语句翻译方法

Switch 结构对于理解程序的逻辑至关重要,如果在程序解码过程中不能够有效地恢复 switch 结构,会使程序的控制结构不完整,后续的数据流分析,类型推导和控制流分析不能正确地进行,反编译结果无意义。

不同的编译器处理 switch 语句的方法^[14]各有所异,即便是同一个编译器在处理不同类别的 switch 语句时,方法也有所差别。基于搜索策略的方法将 switch 实现成一系列 if-equal 的分支嵌套语句;基于跳转表策略的方法将所有跳转偏移组成跳转表,通过 case 值索引各个表项来实现 switch 语句;基于组合策略的方法根据 case 值的分布,对于不同段的跳转既可以采用基于搜索策略的处理方式,也可以采用基于跳转表的处理方式。本节着重分析基于跳转表策略的 switch 结构的构造和规整。

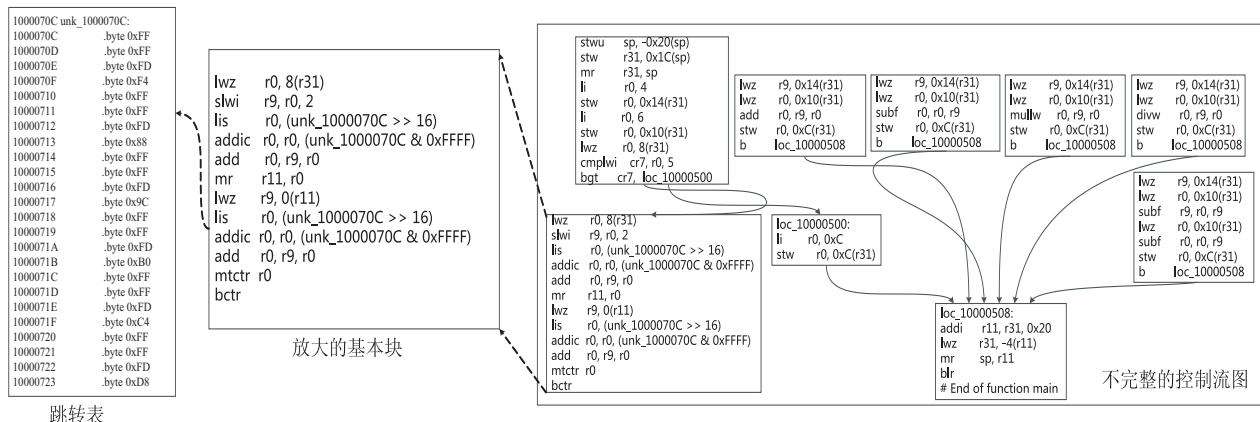


图 2 IDA 建立起的 switch 控制流图

IDA 在分析带有跳转表的 PowerPC 程序时,没有有效地构建出其控制流图,如图 2 所示,从包含 bctr 指

令的基本块,并没有到相应 case 基本块的有向边。因此,在具体分析到 PowerPC 中的间接跳转指令 bctr 时,

需要向前分析,找到跳转表的地址 baseAddr 。在图 2 中,需要到地址 $\text{baseAddr} = 0x1000070C$ 处找到相应的跳转表,在跳转表中每 4 个字节组成一个字,该字构成了相对于 baseAddr 的偏移,将该字与 baseAddr 相加得到最终的跳转地址。其计算方法可以用式 (1) 来实现。

$$\text{realAddr} = \text{baseAddr} + m[\text{baseAddr} + 4 * \text{idx}] \quad (1)$$

式中, idx 表示跳转表的索引,也是 switch 的分支值, $\text{idx} = 0$ 代表 default 分支; $m[\text{Addr}]$ 表示从内存地址 Addr 处取值; realAddr 是最终的跳转地址。

根据式 (1) 计算表明,从 bctr 指令所在的基本块出发,有跳往首地址分别为 $0x10000500$, $0x10000494$, $0x100004a8$, $0x100004bc$, $0x100004d0$ 和 $0x100004e4$ 的基本块的边,在解码阶段需要通过相应算法,恢复出完

整的 switch 控制流图。在反编译基于跳转表策略生成的 switch 语句时,关键是根据 mtctr 指令、 bctr 指令和跳转表,在程序的控制流图中构造出 N-way 的分支结构,如此完整的基于控制流图的中间表达式再经过控制流分析之后会自动地将 N-way 的分支结构构造成为 switch 结构,最终经过代码生成得到 switch 语句。

N-way 结构的构建主要集中在解码阶段,解码器根据被解码指令,指令的操作码以及指令所在的上下文,适当地构造出 N-way 结构的控制流图以及 switch 语句的分支变量。图 3(c) 给出了如何根据 switch 指令模式和跳转表,完成 N-way 控制流图的构建,其中的关键部分是跳转表的查找,跳转表的构建在加载部分完成,以便解码阶段能够方便地得到 N-way 分支结构中的各个跳转地址,构建完整的 N-way 控制流图。

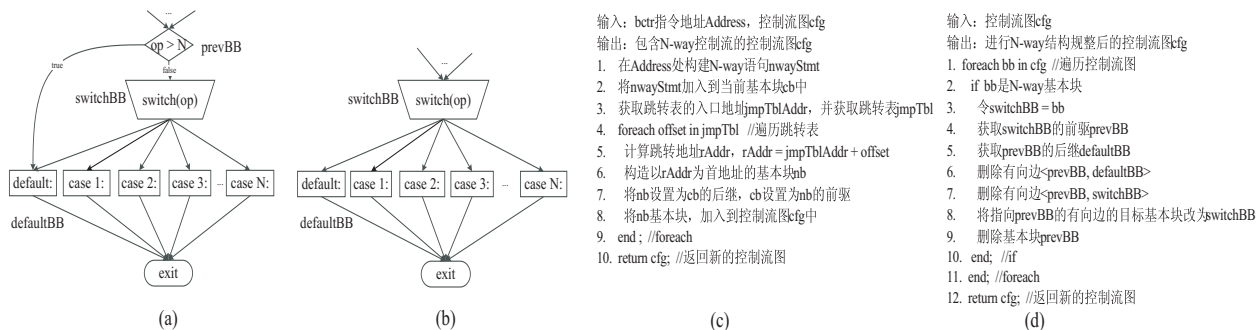


图 3 Switch 结构构建以及规整算法

图 3(c) 列出了 N-way 控制流图构建算法,即 switch 相关指令的解码以及对应控制流图的构建过程。但是,通过图 3(c) 构建起来的 N-way 控制流图并不是完整的 switch-case 语句的控制流图。如图 3(a), 根据图 3(c), 只构造出了由 $\{\text{switchBB}, \text{defaultBB}, \text{case } 1, \dots, \text{case } N, \text{exit}\}$ 等基本块组成的控制流图。但是,完整的 switch-case 语句的控制流图应由 $\{\text{prevBB}, \text{switchBB}, \text{defaultBB}, \text{case } 1, \dots, \text{case } N, \text{exit}\}$ 等基本块组成。图 3(a) 的控制流图,从执行角度来看可以提高执行速度;但是,当被转换成高级代码时,从阅读角度来说,不易于理解。因此,必须将其规整,规整算法如图 3(d) 所示。

图 3(a) 是从汇编程序中构建起的完整的 N-way 控制流图,基本块 defaultBB 有两条入边,分别是 $\langle \text{prevBB}, \text{defaultBB} \rangle$ 和 $\langle \text{switchBB}, \text{defaultBB} \rangle$ 。边 $\langle \text{prevBB}, \text{defaultBB} \rangle$ 是编译器为优化 switch 语句的执行效率和执行速度而引入的,对于反编译没有太大的价值;而边 $\langle \text{switchBB}, \text{defaultBB} \rangle$ 是正常的 switch 语句到 default 分支的执行路径,必须在分析过程中保留。如果在控制流分析之前不将边 $\langle \text{prevBB}, \text{defaultBB} \rangle$ 和基本块 prevBB 删除,那么生成的代码中,往往在 switch 结构外嵌套 if 结构,影响代码理解;因此,将 N-way 控制流图

尽早地规整化成图 3(b) 的模式,对于后续的控制流分析和生成代码的可读性都非常重要。N-way 控制流图规整的核心是对图 3(a) 中控制流图的数据结构的修改,其过程如图 3(d) 所示。

经过 N-way 控制流图的规整算法处理之后,最终生成的 switch 语句的可读性大大增强,去除了嵌套在 switch 结构外层的 if 结构,同时也消除了从 switch 的 default 分支跳往 if-else 处的 goto 语句。在真实的程序中,当有多个 case 分支对应于同一个基本块时,在生成 case i2, case i3, ..., case in 的代码时,通常会引入一个 goto 语句,该语句的跳转目标是 case i1 的起始位置。这样的结构也影响正常的阅读逻辑,必须在反编译过程中进行解决。最简单的解决方案是在代码生成的时候,先统计出每个 case 分支中基本块对应的 case 变量的数目;然后,输出各个 case 变量;最后,输出对应的基本块代码。通过上述方法,在实际中可以大大减少 switch 语句中 goto 语句的数目,优化代码结构,提高可读性。

2.2 代码复制消除 goto 语句

程序中的 goto 语句使程序的逻辑混乱不清。同样,在反编译的结果中也要避免过多生成 goto 语句。在分析了大量产生 goto 语句的过程体的中间表达式

后,发现 goto 语句的引入可以从控制流图上找到对应的结构。

图4展示了有可能产生 goto 结构的控制流图子图模式,消除 goto 结构的方法以及由简单复制引起的级联复制。图4(a)是实际中遇到的一个例子,具有这种结构的程序,如果不进行表达式组合,在生成高级代码时会产生 goto 语句。理想情况下,不同分支的基本块

之间不应当有边相关联,即在控制流图中不应当存在有向边 $\langle B_1, B_2 \rangle$ 。图4(a)的控制流图是因在编译时期,编译器对于组合条件表达式采用短路算法而引入的,文献[1]给出了四种条件表达式组合的子图模式,但归结起来可用图4(a)概括。如果 B_0 和 B_1 的条件表达式是与逻辑,则 B_2 是 false 跳转基本块;如果 B_0 和 B_1 的条件表达式是或逻辑,则 B_2 是 true 跳转基本块。

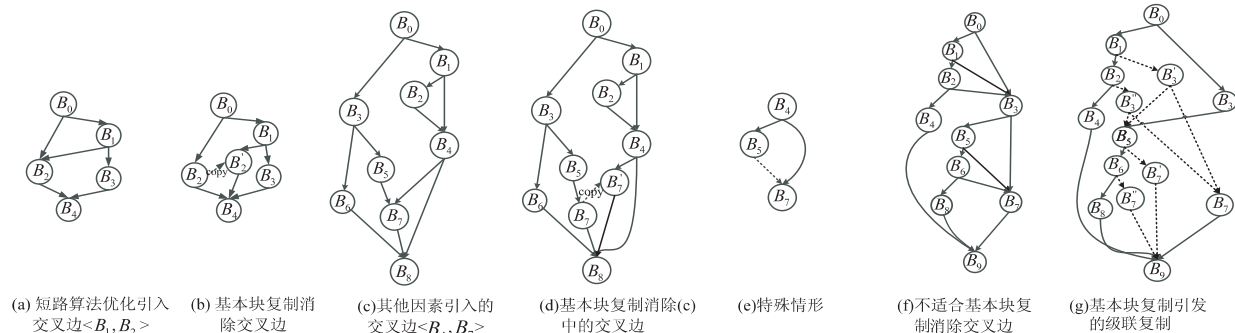


图4 代码复制消除 goto 语句示意

如果通过代码复制,仅消除图4(a)引入的 goto 结构,那么采用文献[1]中提到的子图模式检测算法已经可以满足需求。但是,有时候程序的控制流图比较复杂,如图4(c)中所示的结构,有向边 $\langle B_4, B_7 \rangle$ 并非组合条件表达式采用短路算法而引入的结构。但是,同样由于该有向边的存在,会在生成代码中引入 goto 语句。因此,必须在代码生成之前将该有向边消除。

观察图4(a)和(c),可以总结出一些特征,利用这些特征在控制流图中检测 goto 结构,并通过代码复制的方法消除 goto 结构,如图4(b)和(d)所示的方法。

以图4(c)为例,得到如下四条启发式规则:

- (1) 基本块 B_4 是分支节点,即 $\text{outDegree}(B_4) = 2$;
- (2) 基本块 B_7 是汇合且非分支节点,即 $\text{inDegree}(B_7) \geq 2 \ \&\& \ \text{outDegree}(B_7) = 1$;
- (3) 基本块 B_7 是基本块 B_4 的后继节点,即 $B_7 \in \text{succ}(B_4)$;
- (4) 基本块 B_4 不是基本块 B_7 的前向支配节点,且 B_7 不是 B_4 的后向支配节点。

启发式规则(1)和(2)是对图4(c)中的基本块 B_4 和 B_7 的度的数值关系的描述;规则(2)中的条件限制 $\text{outDegree}(B_7) = 1$ 非常重要,如果不做这一限制,在某些情况下,将会引入更多的 goto 结构,如图4(f)和(g)所示。

对于图4(f)中的控制流图,如果不限制被复制基本块的出度,则会复杂化程序的控制结构,增加控制流分析复杂度,影响生成代码的可读性。图4(f)中有四条交叉边,分别是 $\langle B_1, B_3 \rangle$, $\langle B_2, B_3 \rangle$, $\langle B_5, B_7 \rangle$ 和 $\langle B_6, B_7 \rangle$ 。图4(g)是未对被复制基本块的类型进行

限制而采用基本块复制得到的程序控制流图,在原有的四条交叉边被消除的同时,又引入了新的交叉边。例如,有向边 $\langle B_3, B_5 \rangle$, $\langle B_3', B_5 \rangle$ 和 $\langle B_3'', B_5 \rangle$ 等边又成为新的交叉边,这些边在随后的迭代过程中要继续被消除,引发大量的复制。这样的级联复制会使程序的控制流图比较复杂,生成的代码的冗余度大大增加,并有可能引入更多的 goto 语句,进而影响最终生成的程序的可读性。因此,必须对被复制基本块的类型进行限制。

规则(3)描述了存在一条有向边 $\langle B_4, B_7 \rangle$;规则(4)则排除如图4(e)中的子图模式,图4(e)中的有向边 $\langle B_4, B_7 \rangle$ 满足启发式规则(1)、(2)和(3),但是该有向边并不是交叉边。

整个 goto 结构检测算法遵从规则(1)到规则(4)的检验顺序,即整个判断逻辑是与的关系。逻辑判断式为

$$\text{isGoto} = \text{pCond}(1) \wedge \text{pCond}(1) \wedge \text{pCond}(1) \wedge \text{pCond}(1)$$

$\text{pCond}(i)$ 表示第 i 条规则。如果 $\text{isGoto} = \text{true}$,则边 $\langle B_4, B_7 \rangle$ 是交叉边,否则是正常的边。

通过迭代地使用上述四点启发式规则,可以很好地在控制流图中检测出 goto 结构。当 goto 结构被检测出之后,按照图4(d)将基本块 B_7 复制一份,得到孤立的基本块 B_7' ;接着,修改有向边 $\langle B_4, B_7 \rangle$ 为有向边 $\langle B_4, B_7' \rangle$;最后,将 B_7 的出边赋给 B_7' ,作为 B_7' 的出边。

代码复制的时间非常关键,不能够在解码结束后立刻进行代码复制,过早的复制会导致被复制的基本块中的被定值变量在数据流分析时处于不同的上下文中,从而在代码生成时,导致代码不一致,影响代码理

解。例如,假设在图 4(c)的基本块 B_7 中定义了变量 x ,如果在解码结束,立即进行代码复制,那么变量 x 就存在两份,记为 x_{B_7} 和 x_{B_7} 。那么在进数据流分析中, x_{B_7} 所处的上下文是 $\langle B_0, B_3, B_5, x_{B_7}, B_8 \rangle$,而 x_{B_7} 所处的上下文是 $\langle B_0, B_1, B_2, B_4, x_{B_7}, B_8 \rangle$ 。最终,经过数据流分析之后,可能导致 $x_{B_7} \neq x_{B_7}$ 。代码复制的最佳时间是在控制流分析之前,这样既能保证代码的一致性,同时还能保证程序的结构清晰。

总之,通过代码复制可以消除部分 goto 语句,增加代码的可读性。但是,这种策略适合于复制顺序基本块,而不适合复制分支基本块;如果待复制的基本块是分支基本块,则最好采用条件表达式组合的方式消除 goto 语句。另外,代码复制的时间应当在控制流分析之前进行,过早的复制会导致代码的不一致性。

2.3 指令习语分析与重构

编译器在生成代码时,会生成固定的指令序列流来解释高级语言中的某些操作,从而提高可执行代码

的运行速度,减少可执行代码的体积和模拟目标体系结构不支持的操作。例如,用软件模拟的方式,在不支持浮点运算的平台上实现浮点运算,通常将这些固定的指令序列流叫做指令习语。文献[15]在中间表达式 LLVM IR 上检测指令习语,通过重构中间表示形式,最终得到可读性较好的目标代码。但是,从指令习语分析的有效性和精确度来说,应当将指令习语的检测提前至解码阶段,毕竟不同的指令系统的指令习语序列有所差异。

分析 SPEC CPU 2006 中的部分测试用例的汇编代码,总结出如表 1 的指令习语及其对应的指令序列,习语主要集中在判断语句、跳转语句、乘除法和取模运算等方面。在指令序列中,可能由于编译器的代码优化,例如代码移动等,在指令序列中插入其他指令,在分析时要根据情况将这些指令提出,继续后续指令序列的匹配。

表 1 指令习语及分布情况

指令习语		汇编程序中对应的指令序列	SPEC CPU 2006 测试集指令习语统计情况				
编号	名称		bzip2	mcf	sjeng	h264ref	lbm
1	判断语句	比较指令; mfcrr; rlwinm	41/1.1	2/0.04	65/0.1	174/0.07	6/0.07
2	跳转语句	比较指令; 控制转移指令	1 381/3.8	205/4.3	2 807/5.2	7 953/3.2	74/0.9
3	乘以 2^n	slwi	1 434/4.0	68/1.4	1 975/3.7	12 108/4.8	389/4.8
4	乘以非 2^n	mulli	6/0.02	23/0.5	421/0.8	1 305/0.5	314/3.9
5	除以 2^n	srawi	60/0.2	18/0.4	163/0.3	1 810/0.7	8/0.09
6	除以非 2^n	lis; ori; mulhw; srawi; subf	2/0.01	0/0.0	0/0.0	2/0.000 8	0/0.0
7	求 2^n 的模	srawi; addze; slwi; subf	1/0.005	0/0.0	1/0.002	180/0.07	0/0.0
8	求非 2^n 的模	lis; ori; mulhw; srawi; subf; mulli; subf	0/0.0	0/0.0	0/0.0	0/0.0	0/0.0

不同编译器针对不同的体系结构,在不同的优化级别中生成的指令习语是不同的。表 1 中的分析都是针对 powerpc-linux-gcc-4.4 交叉编译器的 -O0 级优化代码而言的。判断语句的指令习语与跳转语句的指令习语的主要区别在于判断语句是顺序语句,而跳转语句是控制转移语句。因此,在解码到比较指令时,一定要继续检查后续指令,以便区分当前的比较指令是判断语句的一部分,还是跳转语句的一部分,从而构造合适的中间表达式。

在普通的乘法运算上,主要考虑乘数和除数是否是 2 的幂次方。如果是,则采用运算速度相对快的移位操作;否则,采用复杂的操作序列实现乘除运算。从高级语言的可读性来看,乘除法的可读性要高于移位操作。因此,在解码阶段要用乘除法语句代替移位语句。至于除以非 2^n 的计算,以及其取模运算为什么这么复杂,可以参考文献[16]。在解码相关指令时,首先要根据指令序列流中的操作数,计算出除数;然

后,再构造出除法或取模运算语句。

为了进一步了解程序中指令习语的分布情况,统计 SPEC CPU 2006 中部分测试集,得到的指令习语数量统计,如表 1 所示。表格中反斜杠前的数字为相应指令习语的个数,而反斜杠后的数字表示指令习语在程序中所占百分比例。

可以看出,各种指令习语在不同的程序中广泛存在,这种高密度的复杂运算,如果在解码阶段不进行有效地分析和替换,必将影响后续的分析以及生成代码的可读性。

指令习语的检测属于解码的重要内容,每解码一条指令,都要分析其附近地址的指令,检测附近几条指令是否有效地组成一个指令习语。如果是,则将这几条指令联合起来翻译,构造一个语义更加清晰的中间表达式;否则,按照单条指令进行解码,并构造其中间表达式。

表 1 中的指令习语,一般都出现在一个基本块中,

有些时候指令习语是程序控制流图中的一个子图,此时的指令习语检测比较困难。在反编译某编译器生成的 PowerPC 汇编程序中,64 位长除法或长乘法由将近 20~30 条汇编指令来解释,在计算过程中要判断除数是否为零和计算结果是否溢出等,这些指令组成了复杂的控制流子图。在实际的反编译过程中,需将该控制流子图进行消减,以简化程序分析。

图 5 的左栏上部分列出的是执行 64 位长乘法的

01. mulhw R0, R29, R27	Ld00e4:	R20=0x0;
02. mullw R8, R29, R27	14. li R11, -0x1	R2=0x7fffffff;
03. mr R7, R0	15. lis R12, 0x8000	R4=ROTL(((R29-R25)*R27),0x11)&0xfffe0000
04. rlwinm R4,R8,0x11,0xf,0x1f	16. ori R12, R12, 0x0	ROTL(((R29-R25)*R27),0x11)&0x1ffff&0x1ffff;
05. rlwimi R4,R7,0x11,0x0,0xe	17. cmpw CR0, R3, R11	R3=((R29-R25)*R27)>>0xf;
06. srawi R3, R7, 0xf	18. bne CR0, Ld00fc	if ((R3<=R20) (R4 <=R21)) {
07. cmpw CR0, R3, R20	19. cmplw CR0, R4, R12	R11=0xffffffff;
08. bne CR0, Ld00d4	Ld00fc:	R12=0x80000000;
09. cmplw CR0, R4, R21	20. bge CR0, Ld0108	if ((R3>=R11) (R4>=R12)) {
Ld00d4:	21. lis R31, 0x8000	R31=R4;
10. ble CR0, Ld00e4	22. b Ld010c	} else {
11. lis R31, 0x7fff	Ld0108:	R31=0x80000000;
12. ori R31, R31, 0xffff	23. addi R31, R4, 0x0	}
13. b Ld010c	Ld010c:	} else {
	...	R31=0x7fffffff;
		}
		//R31 中存储着最终的计算结果
R31=(R29-R25)*R27; //64 位长乘法		

图 5 指令习语的作用对比

在实际的反编译过程中,指令习语可以在解码的时候通过指令流匹配的方式识别出来。具体的指令序列流的模式与编译器,目标体系结构和优化级别等紧密相关,可以通过建立指令序列流模板库的方法来实现不同指令序列流的识别,模板库的构造方法可以参考文献[17]。

3 实验

测试程序分别取自 SPEC CPU 2006 标准测试集和 Olden 标准测试集^[18]。首先,利用 powerpc-linux-gcc-4.4 交叉编译器,采用-O0 优化,将测试程序编译成汇编程序。然后,通过实验,分别评价反编译器的整体功能以及两种关键技术对生成代码在结构上的改善。

测试计算机的 CPU 为 Intel Core Quad 3.10 GHz,内存为 8 GB,操作系统为 Windows 7。由于在文献中未查找到 PowerPC 汇编程序的反编译器,因此,对比实验暂时不能进行。

为了评价反编译的整体效果,定义两个评价指标—膨胀率和压缩率。膨胀率是反编译行与源码行的比值,压缩率是汇编行和反编译行的比值。

根据以上两种评价指标,得到 Olden 测试集和 SPEC CPU 2006 测试集的平均膨胀率约为 2.2,平均压缩率约为 2.0。反编译结果在代码行数上整体要少于

汇编程序,右栏列出的是没有经过指令习语分析的反编译结果,左栏的下部分列出的是经过习语分析之后的反编译结果。未经过指令习语分析的翻译结果比较复杂,如此复杂的代码块,对于理解整个程序的逻辑没有实质性的作用,反而转移分析人员的理解重心;而经过指令习语分析后的翻译结果,简洁直观,代码长度得到有效缩减。

汇编代码行数。反编译器能够将 SPEC CPU 2006 中的一些大程序进行处理,足见反编译器在性能方面的可靠性。

表 2 中的 switch 语句数目对比一栏中,反斜杠后面的数字代表汇编程序中 switch 语句的个数,反斜杠前面的数字代表反编译结果中 switch 语句的个数;同样,对于 goto 语句数目对比一栏中,反斜杠后面的数字代表没有进行代码复制时,goto 语句的数目,反斜杠前的数字代表代码复制后,goto 语句的数目。从表中可以看出,反编译器翻译出了所有的 switch 语句,并且消除了 11.2% 的 goto 语句。说明了 switch 语句翻译方法和代码复制消除 goto 语句等方法的有效性。

4 结束语

针对 PowerPC 汇编程序,通过扩展 Boomerang 开源软件,构造可以反编译 PowerPC 汇编程序的反编译器。利用汇编程序中的跳转表和对 N-way 控制流图的规整,可以有效地反编译出结构良好的 switch 语句。通过仔细分析生成代码中 goto 语句对应的控制流图模式,构建相应流图的检测和规整算法,有效地删除 goto 语句。统计汇编程序中的指令习语的模式,构造指令习语的检测算法,在解码阶段将指令习语替换成可读性更高的中间表达式,有助于提高程序的可读性。

表 2 反编译器整体功能评价

应用程序	过程数	源码行	汇编行	反编译行	switch 语句数目对比	goto 语句数目对比
bisort	3	350	905	399	0/0	4/4
health	17	504	1 921	771	0/0	3/3
mst	16	428	1 207	669	0/0	4/4
perimeter	12	484	1 157	791	0/0	9/20
power	17	622	2 257	877	0/0	5/5
treeadd	4	245	251	125	0/0	0/0
voronoi	45	1 151	4 302	2 137	0/0	9/9
bzip2	122	8 293	35 995	19 597	4/4	531/552
h264ref	593	51 578	249 791	122 864	12/12	1 898/2 255
lbm	22	1 155	8 076	2 166	0/0	22/22
mcf	24	2 685	4 743	2 424	0/0	57/61
sjeng	144	13 847	53 617	27 251	17/17	705/721

参考文献：

[1] Cifuentes C. Reverse compilation techniques[D]. Queensl- and;Queensland University of Technology,1994.

[2] Hex-rays decompiler[EB/OL]. 2014. [https://www. hex - rays. com/products/decompiler](https://www.hex-rays.com/products/decompiler).

[3] Eagle C. The IDA pro book[EB/OL]. 2014. [http://www. ida- book. com/](http://www.ida-book.com/).

[4] Fonseca J M R. Interactive decompilation[D]. Wales;Univer- sity of Wales,2006.

[5] Kapil A,Matthew S,Khaled E,et al. A compiler-level inter- mediate representation based binary analysis and rewriting system[C]//Proceedings of the 8th ACM European confer- ence on computer systems. Prague, Czech Republic; ACM, 2013.

[6] Bougacha A. Dagger[EB/OL]. 2014. [http://dagger. repzret. org/](http://dagger.repzret.org/).

[7] Luk U,Jakub K. Design of an automatically generated retar- getable decompiler[C]//Proceedings of the 2nd international conference on circuits,systems,communications and comput- ers. [s. l.]:[s. n.],2011.

[8] van Emmerik M J. Static single assignment for decompilation [D]. Queensland;University of Queensland,2007.

[9] Duke R,Rose G,Smith G. Object-Z;a specification language advocated for the description of standards [J]. Computer Standards & Interfaces,1995,17(5):511-533.

[10] Kästner D,Wilhelm S. Generic control flow reconstruction from assembly code[C]//Proc of SIGPLAN. [s. l.]:ACM,2002.

[11] Mycroft A. Type-based decompilation,programming languages and systems[M]. [s. l.]:Springer,1999:208-223.

[12] Cifuentes C. Structuring decompiled graphs[C]//Proc of in- ternational conference on compiler construction. [s. l.]:[s. n.],1996.

[13] Dijkstra E W. Go to statement considered harmful[M]. [s. l.]:Springer,2002.

[14] Wienskosi E. Switch Statement case reordering FDO[C]// Proc of GCC. [s. l.]:[s. n.],2006.

[15] Kroustek J,Pokorny F. Reconstruction of instruction idioms in a retargetable decompiler [C]//Proc of FedCSIS. [s. l.]: IEEE,2013.

[16] Warren H S. Hacker’s delight[M]. [s. l.]:Addison-Wesley Longman Publishing Co,2002.

[17] Chen Gengbiao, Qi Zhengwei, Huang Shiqiu, et al. A refined decompiler to generate C code with high readability [C]// Proc of 17th working conference on reverse engineering. [s. l.]:IEEE,2010.

[18] Olden Benchmark Suite[EB/OL]. 2014. [http://www. cs. princeton. edu/~ mcc/olden. html](http://www.cs.princeton.edu/~mcc/olden.html).

PowerPC汇编程序的反编译研究

作者：[刘延昭](#)，[赵银亮](#)，[武万杰](#)，[LIU Yan-zhao](#)，[ZHAO Yin-liang](#)，[WU Wan-jie](#)
作者单位：[西安交通大学 计算机科学与技术系, 陕西 西安, 710049](#)
刊名：[计算机技术与发展](#)
英文刊名：[Computer Technology and Development](#)
年，卷(期)：2015(8)

引用本文格式：[刘延昭](#).[赵银亮](#).[武万杰](#).[LIU Yan-zhao](#).[ZHAO Yin-liang](#).[WU Wan-jie](#) [PowerPC汇编程序的反编译研究](#)[期刊论文]-[计算机技术与发展](#) 2015(8)