计算机研究与发展
Journal of Computer Research and Development
DOI:10.7544/issn1000-1239.2019.20170412
56(4):742-754, 2019

# 一种高效解决间接转移的反馈式静态二进制翻译方法

王 军 庞建民 傅立国 岳 峰 张家豪
(数学工程与先进计算国家重点实验室(战略支援部队信息工程大学) 郑州 450002)
(wj_xd@foxmail.com)

# An Efficient Feedback Static Binary Translator for Solving Indirect Branch

Wang Jun, Pang Jianmin, Fu Liguo, Yue Feng, and Zhang Jiahao
(*State Key Laboratory of Mathematical Engineering and Advanced Computing* (*Strategic Support Force Information Engineering University*), *Zhengzhou* 450002)

**Abstract** In order to solve the problem of indirect branch efficiently in static binary translation, a feedback static binary translation method is proposed, with two-level address mapping table to realize the fast mapping of indirect branch target address. This method can solve the problem of less code optimization and more redundant code in existing linear traversal translation. Firstly, the two-level address mapping table is used to address the code location quickly, using array address to store the target platform code block address in the order of the source platform base block start address and using array index to save the index position of the basic block start address in array address. Then, the monitoring feedback mechanism is added to the target executable program to carry on the code discovery, and the uncertain indirect branch target address would be returned so that the source code can be divided to new basic blocks and re-translated. The feedback static binary translation framework FD-QEMU is implemented based on QEMU(quick emulator), an open source binary translator. As the experimental results on SPEC2006 and NBENCH show, compared with QEMU, the speedup ratio of FD-SQEMU (feedback static QEMU) is 3.97 and 6.94 times on average; compared with SQEMU, a static translator with all instructions' address mapping originally proposed by our group, the average acceleration ratio of FD-SQEMU is 1.18 times, and the maximum speedup ratio is 1.36 times, which verifies the effectiveness of the framework and method proposed in this paper.

**Key words** binary translation; static binary translation; indirect branch; translator QEMU; feedback translation framework FD-SQEMU

**摘 要** 为了在追求程序执行效率的同时解决静态二进制翻译中的间接转移问题,针对现有间接转移问题处理方法中线性遍历翻译方式代码优化较少、冗余代码较多的缺陷,提出了基于基本块翻译的反馈式静态二进制翻译方法,并结合二级地址映射表实现了间接转移目标地址的快速映射.首先,在目标可执行程序运行过程中添加监控反馈机制解决代码发现问题,对未确定的间接转移地址进行反馈,以便对源程序重新划分基本块并重新翻译执行;然后构造二级地址映射表,借助二级地址映射快速解决代码定位问题.在开源二进制翻译平台 QEMU(quick emulator)上实现了反馈式静态二进制翻译框架 FD-SQEMU(feedback static QEMU),并基于 SPEC2006 和 NBENCH 测试集进行测试,与 QEMU 相比,

**收稿日期**:2017-06-09;**修回日期**:2018-04-08
**基金项目**:国家自然科学基金项目(61472447,61802433)
This work was supported by the National Natural Science Foundation of China (61472447, 61802433).
**通信作者**:庞建民(jianmin_pang@126.com)

FD-SQEMU 平均加速比分别达到 3.97 倍和 6.94 倍;与课题组之前提出的保存源程序指令全地址的静态 SQEMU 翻译器相比,FD-SQEMU 的平均加速比达到 1.18 倍,最高加速比达到了 1.36 倍,验证了提出的框架和方法的有效性.

**关键词**　二进制翻译;静态二进制翻译;间接转移;翻译器 QEMU;反馈式翻译框架 FD-SQEMU

**中图法分类号**　TP391

二进制翻译[1]的核心目标是将源指令集的指令序列通过代码翻译操作,翻译成等价的其他指令集的指令序列.二进制翻译已广泛用于软件安全分析[2-3]、程序行为分析[4]、软件逆向工程、系统虚拟等领域,并已成为软件移植[5]的主流技术.如 FX!32 把 x86 平台下的 Win32 应用程序移植到 Alpha 平台[6-7];昆士兰大学开发的跨平台静态二进制翻译器 UQBT(University of Queensland binary translator),可以支持不同源平台应用程序到目的平台的翻译;动态二进制翻译器 QEMU (quick emulator) 同时支持进程级和系统级程序翻译,并已成功实现对国产龙芯平台的支持[8].

主流的二进制翻译可以分为动态二进制翻译、静态二进制翻译和动静结合的二进制翻译[9].动态二进制翻译[10]是一种即时翻译(边翻译边执行)技术,在运行目标代码时能很好地解决代码发现和代码定位问题[11-12],但是动态二进制翻译需要在动态翻译源平台二进制可执行程序的同时执行翻译生成的目标平台二进制程序,翻译时间和代码优化时间占据程序执行时间,所以动态二进制翻译不能采用深度优化的方法.即便目前有很多优化方法,如热路径优化[13]、寄存器映射[14]、多线程并行优化[8]等,在一定程度上提高了动态二进制翻译的效率,但动态二进制翻译效率低的问题仍然比较突出.静态二进制翻译在不运行程序的情况下,静态分析源平台可执行文件,提取程序指令进行翻译,能够采用较复杂的代码分析和优化算法,能生成高质量的代码,程序执行效率较高;而且静态二进制翻译方式一次翻译多次执行的特性也十分适用于高性能计算程序的翻译,但是静态二进制翻译难以处理下段的代码发现和代码定位问题[15].

代码发现问题是指在静态条件下难以获取间接转移(具体可分为间接跳转和间接调用[16])目标地址的问题,间接转移指令的目标地址可能是一个寄存器或者存储器的值;代码定位问题是指难以在程序执行时确定源平台转移目的地址对应的目标平台代码地址.

目前现有的解决静态二进制翻译中由间接转移引起的代码发现和代码定位问题的方法,无论是解释器法还是地址映射表法,都存在一个共同的缺陷:生成代码质量不高且含有较多的冗余代码.为了更高效地解决静态二进制翻译面临的代码发现问题,本文对课题组之前实现的基于动态二进制翻译器 QEMU 的静态二进制翻译器 SQEMU[17](static QEMU)进行了优化和改进,提出了以基本块为单位进行翻译的反馈式静态二进制翻译框架(feedback static QEMU, FD-SQEMU),并提出了二级地址映射表法,更快速地实现了代码定位.为了解决基本块翻译引入的代码定位问题,设计了高效的二级地址映射表来快速定位转移指令目标地址.在翻译过程中,使用数组结构 *Address* 保存源平台每个基本块对应的目标平台的代码基本块起始地址,并使用索引数组 *Index* 保存源平台指令地址在 *Address* 中对应的索引位置,从而根据 *Index* 和 *Address* 数组快速得到源平台指令块在目的平台上的对应地址.为了解决静态二进制翻译中的代码定位问题,本文引入了反馈机制,在静态翻译目标程序时对间接转移指令添加监控反馈代码,之后运行翻译后的目标程序,依据监控反馈信息[6]确定间接转移目标地址进行代码发现.之后,依据反馈的间接转移目标信息对源程序重新进行基本块划分,然后继续翻译、执行、反馈,重复此过程直至程序能正常执行结束.

本文的主要贡献有 3 个方面:

1) 针对现有静态二进制翻译中由间接转移指令引起的代码发现和代码定位问题解决方法效率较低的缺陷,提出了基于基本块翻译的反馈式静态二进制翻译框架 FD-SQEMU,以基本块方式进行翻译能更好地进行代码优化,生成更高质量的目标代码.

2) 针对 SQEMU 线性逐行遍历翻译方式不能兼顾程序上下文以及引入大量冗余代码的缺陷,改进了地址映射方式,提出了二级地址映射表法,更快速地解决了静态二进制翻译中代码定位问题.

3) 提出了新的代码发现的思路,利用动态执行、监控、反馈进行代码发现,更有效地解决了数据段代码问题.另外,该方法虽然是针对基于 QEMU 的静态二进制翻译框架 SQEMU 实现的,但具有相当强的通用性,不限于目标平台和指令形式,对降低二进制翻译后的代码膨胀率有重要意义,有很强的可扩展性.

## 1 相关工作

目前,静态二进制翻译在不同的平台已有不同的实现,本节首先对静态二进制翻译中间接转移处理的相关方法进行介绍,然后重点对二进制翻译系统 HBT(hybrid binary translation)和 SQEMU 的工作方式及其优缺点进行介绍.

### 1.1 常见的静态二进制翻译中间接转移处理方法

一种主流的解决静态二进制翻译中间接转移问题的方法是引入解释器.

目前已有成熟的静态二进制翻译系统引入解释器模块来处理二进制翻译中的间接转移问题,例如跨平台静态二进制翻译器 UQBT 系统[7].UQBT 系统先将源平台二进制文件 $M_s$ 抽象为机器无关语言 HRTL(higher-level register transfer language),之后对 HRTL 中间语言进行优化,生成目的平台可执行文件.在将 HRTL 中间语言翻译为目标机器指令遇到间接转移指令时,UQBT 系统使用解释器再次使用源二进制文件解释执行获得 $M_s$ 到 $M_t$ 的地址映射.

使用解释器法处理间接转移存在着 2 个问题:1)处理速度较慢;2)需要重复使用源二进制文件的代码段.

另一种主流的解决静态二进制翻译中间接转移问题的方法是地址映射表法,地址映射表实质上替代了上述解释器获取间接转移目标的功能.

比较成熟的使用地址映射表法处理间接转移问题的二进制翻译系统如 Shen 等人[18]提出的基于 LLVM(low level virtual machine)的动静结合的二进制翻译系统 HBT 和卢帅兵等人[17]提出的基于 QEMU 的静态翻译器 SQEMU.

### 1.2 基于 LLVM 的动静结合的二进制翻译系统 HBT

HBT 系统将动态翻译器作为目标代码的一个动态库链接入二进制文件,在翻译源程序完毕开始执行时若遇到间接转移指令,则查找地址映射表确认分支目标地址,如未找到,则启动动态二进制翻译器进行处理.可见,地址映射表结构的设计直接关系到系统执行的效率.

另外,HBT 采用 LLVM switch 指令,生成一系列的 if-else 指令,这些指令需要比较和跳转才能定位到目标跳转地址,该方法的又一个局限性是不可能将所有的目标地址放入 LLVM switch 列表.实际上,HBT 仅将返回地址、函数指针、函数入口点作为地址映射表项,对 switch 语句产生的间接跳转指令并没有统一的解决方法,在实现中仅对 ARM 平台二进制文件进行了处理.

### 1.3 基于 QEMU 的静态翻译器 SQEMU

SQEMU 是基于 QEMU 改造的一个静态翻译器,其实质是采用地址映射表法处理间接转移问题,其具体翻译过程如图 1 所示,其中 TCG(tiny code generator)为 QEMU 自带的中间表示.

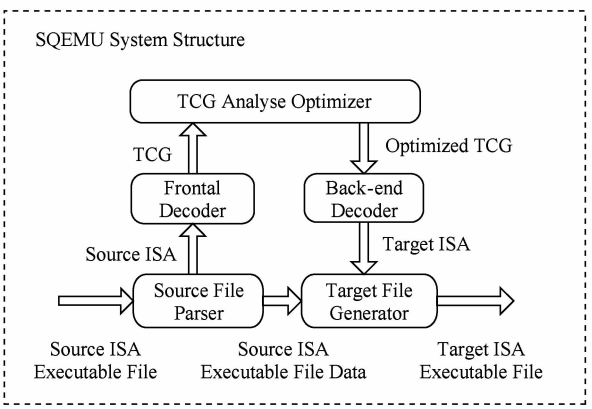



Fig. 1 Framework of SQEMU

图 1 SQEMU 框架设计

SQEMU 分离了翻译时间、代码优化时间和目标程序执行时间[17],使得目标程序在翻译中及翻译后能够采用不同层次的优化算法,生成更高质量的执行代码,这也是静态二进制翻译器的优势.首先,SQEMU 线性遍历、逐行翻译源程序,并使用地址索引映射表记录每一条源平台指令对应的目的平台指令块起始地址,成功解决了静态二进制翻译中由间接跳转和间接转移指令引起的代码发现和代码定位问题.另外,SQEMU 实现了对本地库函数的封装[19],在目标平台模拟源平台参数传递和返回规则,实现了目标程序调用本地库函数的功能,提高了程序执行效率.

但是,为解决静态二进制翻译面临的代码发现和代码定位问题,SQEMU 采用线性遍历、逐行翻译的方法,通过生成每一条 x86 指令对应的标签来辅助进行代码定位。该方法存在 2 个问题:1)逐行翻译的方式忽略了程序上下文信息,无法借助基本块

优化算法等对翻译代码进行有效优化;2)逐行翻译引入了较多的冗余代码,导致代码膨胀率较高,程序整体执行效率较低.

为解决 SQEMU 逐行翻译无法进行块内优化以及引入较多冗余代码、目标程序执行效率较低的问题,本文引入了基本块翻译方式.但与此同时,基本块翻译方式,又带来了代码发现和代码定位的问题.为解决代码发现问题,本文提出反馈式二进制翻译框架,将代码发现放入程序执行过程,在程序执行时进行监控,利用执行反馈情况获取程序间接转移目标地址;为与反馈框架相匹配并高效解决代码定位问题,本文改进了 SQEMU 地址索引映射表,提出了基于基本块首地址的二级地址映射表和映射方法.

## 2　反馈式静态二进制翻译框架 FD-SQEMU

静态翻译是将源平台的二进制可执行文件完全翻译到目标机器上,通常一次翻译后就可以多次在目标平台上运行,并且因为其翻译过程耗费的时间不计入执行时间,所以静态翻译可以花费大量的时间用于生成代码的优化,而且静态翻译可以利用程序以往执行的记录进行优化,以获得更高的效率.

但是,静态二进制翻译在翻译带有间接转移指令的程序时,由于无法获得运行时信息,可能无法获得间接跳转和间接调用的目标地址,无法准确定位到目标平台上该转移地址,进而无法正确实现源二进制程序的功能,不能保证程序翻译的等价性.逐行翻译方式,虽然能够很好地进行代码定位,因为每一行源代码均对应一个目的平台代码块,直接或者间接跳转的任意地址均可以被准确定位,但是该方法无法进行块内优化且程序效率较低.而基本块翻译方式只记录了基本块的首地址,对于转移目的地址在基本块中间的间接跳转或者间接调用指令则无法处理.

为了确定间接跳转和间接调用的目的地址,本文设计并实现了反馈式静态二进制翻译框架.其基本思想是,先翻译源二进制程序,进而执行、监控,在遇到无法确定目标地址的间接跳转或者间接调用时,终止执行并反馈回跳转的目标地址;然后将该跳转目标地址作为一个新的基本块起始地址,重新对源平台二进制文件进行基本块划分,重新翻译,直至能完整实现源二进制文件的功能,实现程序等价性翻译[20].

本文设计的反馈式静态二进制翻译框架如图 2 所示,主要包括 8 个模块:预处理模块、源文件解析器、前端解码器、TCG 分析优化器、后端编码器、目标文件生成器、动态执行模块和反馈地址处理模块.

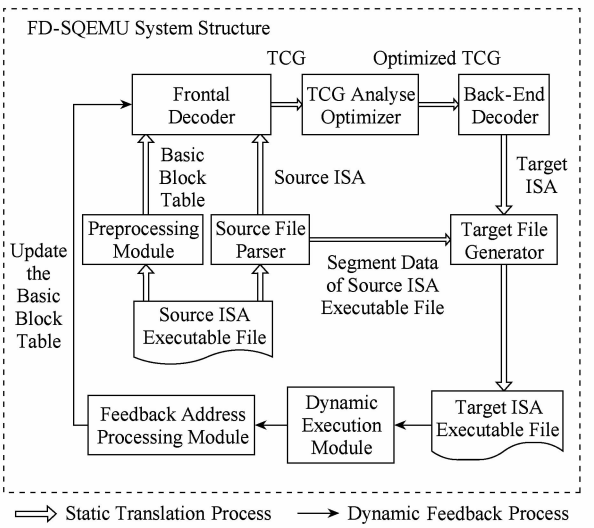



Fig. 2　Framework of FD-SQEMU

图 2　FD-SQEMU 框架设计

1) 预处理模块.分析源平台可执行文件,获取源程序的代码段,对程序的代码段进行基本块划分,生成基本块信息表 $T_b$,基本块划分方式与《编译与反编译技术》这本书中介绍的相同.

2) 源文件解析器.分析源平台可执行文件,获取程序入口地址,提取代码段、数据段、只读数据段、初始化段、动态链接表、全局符号表等.代码段,用于结合基本块信息表进行静态翻译;数据段和只读数据段用于翻译后程序的执行;动态链接表和全局符号表用于对系统函数和用户函数的定位以及使用.

3) 前端解码器.以基本块为单位对源平台指令解码.FD-SQEMU 采用了 QEMU 前端,保留了 QEMU 能对多源平台如 x86,ARM,MIPS 等平台的指令解码功能,并根据译码器分析出的指令生成相对应的 TCG 指令,在遇到间接转移指令时,在该间接转移指令前加上异常处理指令.

4) TCG 分析优化器.对中间表示无关语言 TCG 进行平台无关化优化,如活性分析、块内寄存器分配、块内代码优化等.

5) 后端编码器.依据 TCG 中间表示生成目标平台机器指令.在目标代码层次,根据目标平台特点采取可行的优化措施,如用户函数调用与返回时冗余操作的消除、调用库函数时寄存器快速传递参数等.

6) 目标文件生成器. 重新组合从源文件分析器获得的可执行文件的数据段、只读数据段和全局符号表信息,从 TCG 生成器获得的代码段、初始化段对应的 TCG 中间表示序列,生成 TCG 对应的目标平台可执行 elf 文件,需要处理库函数接口、间接跳转和间接调用指令.

7) 动态执行模块. 提供执行环境,保障由目标文件生成器生成的目标机器代码执行. 若程序完整地正确执行,则说明程序翻译完成;若执行遇到间接转移地址找不到的情况,则返回该转移目标地址,此处返回的地址是源平台可执行文件中的跳转目标地址.

8) 反馈地址处理模块. 该模块依据动态执行模块返回的跳转目标地址,对源平台可执行文件重新划分基本块,并更新基本块信息表 $T_b$,以供系统重新翻译.

实质上,该翻译框架依旧是基于 QEMU[21] 设计的,继续沿用了 QEMU 的前端文件分析和 TCG 中间表示,并继承了 QEMU 的跨平台特性[22]. 另外,在 FD-SQEMU 中,对库函数的处理继续采用了 SQEMU 中库函数封装方法以及库函数本地化调用方法,以提升程序的运行效率.

## 3 间接转移指令的处理

间接转移指令在静态分析时无法确定该指令跳转的目标地址,针对不同运行环境或者运行对象,其转移目标可能不同,本文通过动态执行反馈机制获取间接转移目标地址,进而结合二级地址映射表来确定目的平台上转移目标地址. 下面对二级地址映射表的构造和查找过程以及间接转移目标地址的获取进行详述.

### 3.1 相关定义与符号约定

本文定义下述符号表示翻译过程中涉及的概念.

$M_s$:源平台;

$M_t$:目的平台;

$I_s$:源机器平台指令;

$A_{si}$:源平台间接转移目标地址;

$B_s$:源平台二进制文件;

$B_t$:使用 FD-SQEMU 翻译源平台二进制文件 $B_s$ 到 $M_t$ 所得的二进制文件;

$B_0, B_1, \cdots, B_n, \cdots$:源平台划分的各基本块,$B_0$ 表示从程序入口开始的第 1 个基本块;

$A_0, A_1, \cdots, A_n, \cdots$:源平台划分的各基本块的起始地址(指令块第 1 条指令的地址);

$LB_0, LB_1, \cdots, LB_n, \cdots$:源平台指令基本块对应的目标平台指令块;

$LA_0, LA_1, \cdots, LA_n, \cdots$:源平台划分的各基本块的起始地址对应的目标平台指令块起始地址.

### 3.2 间接转移处理过程

对于间接转移来说,通过静态分析确定间接转移的目标地址是非常难的问题,而且该间接转移指令可能跳转到程序的任何一个位置,本文采用反馈方式,借助动态执行 profile 信息获取间接转移目标地址,具体获取过程算法流程如图 3 所示:

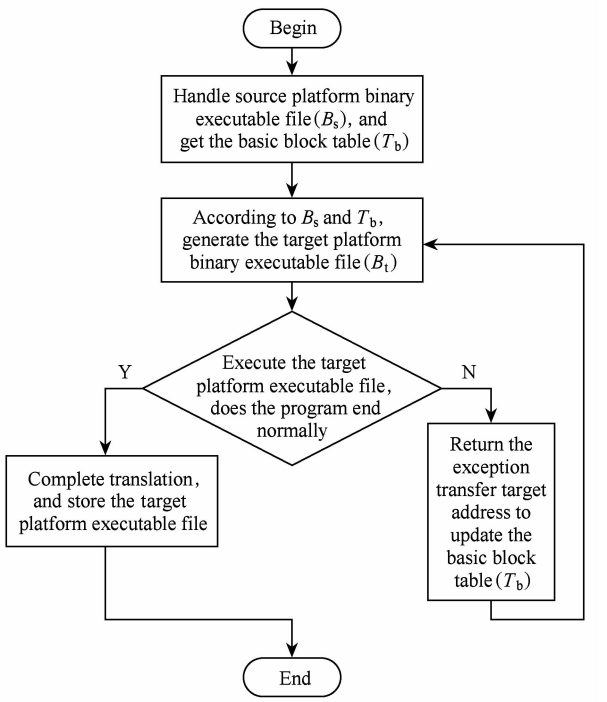



Fig. 3 Solving process of indirect branch

图 3 间接转移处理过程

1) 预处理模块对源平台二进制文件 $B_s$ 进行基本块划分,获取各基本块起始和结束地址信息,存入基本块信息表 $T_b$,初始基本块划分算法如算法 1 所示.

2) 根据源平台二进制文件 $B_s$ 以及基本块信息表 $T_b$,结合静态二进制翻译器生成目标平台二进制文件 $B_t$,并在生成 $B_t$ 的过程中在间接转移指令前添加异常处理反馈指令进行监控反馈,详细算法见 3.4 节的算法 4.

3) 将生成的目标平台二进制文件 $B_t$,在动态执行模块执行,如果程序正常结束,则说明程序已翻译完成,可存储供后续使用;如程序找不到转移目标地址 $A_{si}$(说明该地址不是已划分基本块首地址),异常结束,则返回该转移目标地址供反馈模块处理.

4) 反馈处理模块依据程序异常结束抛出的转移目标地址 $A_{si}$,对源平台二进制文件 $B_s$ 重新进行基本块划分,并更新基本块信息表 $T_b$;之后,依据更新后的基本块信息表 $T_b$,重新对源平台二进制文件进行翻译,待再次翻译完成后重新执行;重复此过程,直至程序能正常执行、正常结束,完成该源程序的翻译.

**算法 1.** 初始基本块划分算法.

功能:对 x86 源代码进行初始基本块划分;

输入:x86 汇编代码;

输出:基本块信息表 $T_b$.

① 获取每个基本块的入口语句:程序的第 1 条语句,能由跳转语句或者函数调用语句转移到的语句,紧跟在条件转移或者函数调用语句后的语句.

② 确定每条入口语句所属的基本块:由该入口语句到下一条入口语句、转移语句或者停语句之间的语句序列.

③ 删除掉未被纳入到基本块中间的语句,并根据基本块起止地址信息生成基本块信息表 $T_b$.

在对源程序重新进行基本块划分时,重点便是更新基本块信息表 $T_b$,主要是更新该表中基本块的起始地址和结束地址.在反馈模块更新基本块信息表 $T_b$ 时,如发现某一转移目标地址 $A_{si}$ 在某一基本块 $B_x$ 的中间(该地址大于基本块起始地址,小于该基本块结束地址),则将该基本块 $B_x$ 划分为 2 个基本块,并以转移目标地址 $A_{si}$ 为下一基本块的起始地址,该转移目标地址上一条指令的地址为上一基本块的结束地址,以图 4 QUICKSORT 程序的核心代码片段为例进行说明.

如图 4 所示,如在原先划分的起始地址 $A_x$ 为 0x4007f5、结束地址是 0x400899 的基本块 $B_x$ 中有一间接转移目标地址 $A_{sin}$ 为 0x40080d,则将原来的基本块拆分为 2 个基本块,这 2 个基本块的起止地址分别是 0x4007f5～0x400809,0x40080d～0x400899,并据此更新基本块信息表 $T_b$,为源平台二进制程序后续的静态翻译提供基本块划分信息.

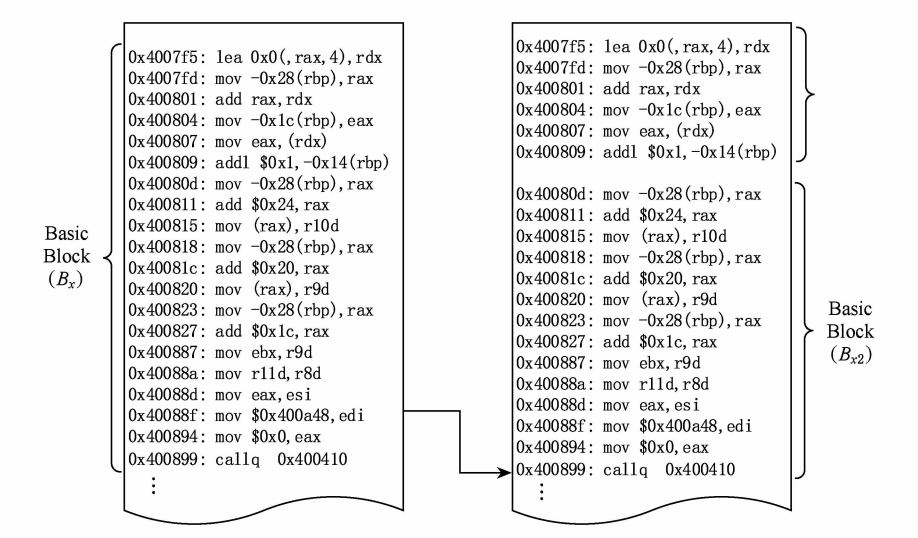



Fig. 4　Re-division of basic block

图 4　基本块重划分

针对间接转移目标地址的确定,本系统将其作为基本块首地址进行处理,并存入二级地址映射表,以方便转移地址的快速查找及定位.

### 3.3　地映射表的构造

在确定源平台间接转移目标地址后,其就成为一基本块的起始地址,为了保证程序的逻辑关系以及正确翻译,在翻译时必须保证各基本块首地址映射的正确性.

为了解决 $M_s$ 指令块地址到 $M_t$ 指令块地址的映射问题,需要保存所有 $M_s$ 基本块对应的 $M_t$ 基本块的地址,并采用合适的方式来保证映射过程的高效性,以提高程序的效率及质量.

地址映射表的构造是在翻译阶段完成的.首先使用源文件解析器,分析源可执行文件的各个段数据,提取出包含指令的代码段、初始化段、附加段信息,然后使用前端解码器按照地址从小到大(即 $A_0$,$A_1$,…,$A_n$,…)的顺序逐块翻译 $M_s$ 指令到 TCG 中间表示,再借助后端编码器翻译为 $M_t$ 指令.在生成目标代码的过程中,如在翻译以 $A_n$ 为起始地址的一个指令块 $B_n$ 时,在生成目标代码首部加上标签 $LA_n$,并且使用数组 $Address$ 保存此标签值,此标签值即目标地址信息.另外,使用 $A_n$ 相对 $A_0$ 的偏移

量作为索引,能够快速定位目的地址.

图 5 给出了在翻译中二级地址映射表的构造过程及源平台和目标平台指令块地址映射信息.

图 5 详细给出了存储源平台代码块首地址到目的平台代码块首地址信息的二级地址映射表的构造过程,其中二级地址映射表的具体生成算法如算法 2 所示.在算法 2 中,使用数组 $Address$ 存储源平台 $M_s$ 指令块翻译为 $M_t$ 平台指令块的首地址,使用数组 $Index$ 存储与 $M_s$ 指令块对应的 $M_t$ 指令块在数组 $Address$ 中的索引位置.

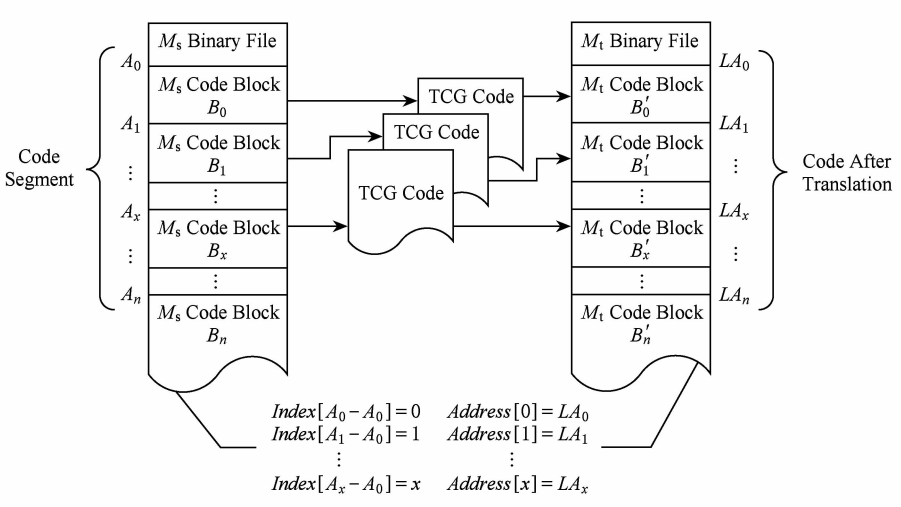



Fig. 5 Construction of two-level address mapping table

图 5 二级地址映射表的构造过程

**算法 2.** 地址映射表的构造算法.

功能:构造源程序地址到目标平台程序地址的映射表;

输入:包含基本块 $B_n$ 起始地址 $A_n$ 和结束地址 $E_n$ 的基本块信息表 $T_b$;

输出:地址数组 $Address$ 和索引数组 $Index$.

① FOR 每一个$\{A_0,A_1,\cdots,A_n,\cdots\}$ 中的 $A_x$

② 分析起始地址为 $A_x$ 的基本块 $B_x$ 中的操作码和操作数,并将分析结果翻译成 TCG 中间代码;

③ 将TCG 中间代码翻译至目标机器代码,并使用 $LA_x$ 记录目标机器代码块的起始地址,令 $Address[x]=LA_x$,$Index[A_x-A_0]=x$;

④ END FOR

在构造二级地址映射表后,本文设计了间接转移分支指令目的地址的查找算法,如算法 3 所示.

**算法 3.** 间接转移分支指令目的地址查找算法.

功能:获得间接转移分支指令的目的地址;

输入:$M_s$ 源平台程序起始地址和 $M_t$ 目标平台上间接转移指令的目的地址;

输出:目标平台 $M_t$ 上间接转移目标地址.

① 获取 $M_s$ 源平台程序起始地址 $A_x$ 和源程序间接转移目标地址 $A_0$;

② 计算 $A_x$ 相对于 $A_0$ 的偏移地址:$A_x-A_0$;

③ 返回 $M_t$ 目标平台上间接转移目标地址:

$$Address[Index[A_x-A_0]].$$

如算法 3 所示,结合二级地址映射,采用算法 3,利用源平台程序基本块起始地址信息,仅需 3 条指令即可定位到间接转移目标地址.

下面举例说明该二级地址映射表构造及查找算法的执行效果.当 $M_s$ 为 x86 平台、$M_t$ 为某国产平台时,基本块 $B_0$ 的起始地址 $A_0=$0x400530,前 3 个指令块的第 1 条指令分别为 xor,pop,mov,对应的 TCG 中间表示代码块和生成的申威国产平台指令块如图 6 所示.

在目标平台二进制文件运行遇到间接转移指令如 call *%rax 时,以间接转移指令目标地址 rax 为 0x4005ce 为例,使用间接转移分支指令目的地址查找算法仅使用 3 条指令即可获得标签 L_0x4005ce 的值,然后转移到该标签的位置.具体定位间接跳转过程为:首先依据源平台间接转移地址 0x4005ce,访问 $Index$[0x4005ce−0x400530]即 $Index$[158],获得以 0x4005ce 为起始地址的代码块对应的目的平台代码块起始地址 L_0x4005ce 在数组 $Address$ 中的位置 1;继而访问 $Address$[1]获得 L_0x4005ce 的值,即定位到间接转移分支目的地址.

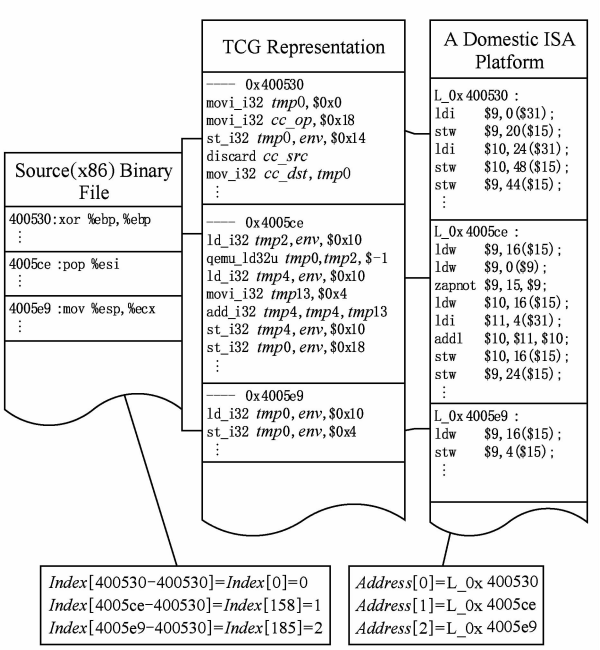



Fig. 6　Example of translating x86 ISA to national ISA

图 6　x86 指令块翻译为某类国产平台的指令举例

从算法 2 可以看出，*Address* 保存了所有 $M_s$ 指令块对应的目标代码块地址，*Index* 保存了该所有 $M_s$ 指令块在 *Address* 中的索引位置. 如此设计，一是思路简单明了、利于实现，二是查找目标平台间接转移目标地址效率较高，以地址索引映射表查找方法、以最小的查询代价，解决了间接跳转和间接调用的代码发现问题.

### 3.4　间接转移目标地址的反馈

实际上，在对源平台二进制文件进行基本块划分后，经过翻译对于大部分间接调用地址基本都能够在二级地址映射表中查询得到，这是因为间接调用一般都是调用的子函数，而子函数的入口地址已经被处理为某一基本块的起始地址，并且该地址已经存储在二级地址映射表中，在动态执行时可以查询得到；而对于部分间接跳转指令，因为跳转指令地址的不确定性，其可能跳转到程序的任何位置，比如原先划分的基本块中间某地址，此时该目标地址的确定是重中之重，这也是反馈机制存在的必要性. 本文通过借助 QEMU 的 Helper 机制在翻译源平台可执行程序时添加异常处理反馈代码，采用监控程序执行的方式获取间接转移目标地址，该间接转移地址反馈算法如算法 4 所示：

**算法 4.** 间接转移地址反馈算法.

输入：$M_s$ 源平台上指令 $I_s$，如 callq *%rax 和 jmpq *0x200c04(%rip)指令；

输出：$M_s$ 源平台上不是块首地址的间接转移目标地址.

① 分析 $M_s$ 源平台上间接转移指令 $I_s$，获取操作码 *opcodes* 和操作数 *oprands*.

② IF $I_s$ 是间接转移指令

③　IF $Index[oprands-A_0]\leqslant 0$

④　　将该操作码 *oprands* 写入间接转移地址存储文件 IndirectAdd；

⑤　　程序终止；

⑥　END IF

⑦ ELSE

⑧　将源平台指令 $I_s$ 翻译成 TCG 中间表示；

⑨　将TCG 中间表示翻译成 $M_t$ 目标平台本地代码；

⑩ END IF

下面举例说明算法 4 的工作原理，依旧以图 4 中的间接调用指令 callq *%rax 为例，在对这条指令进行翻译时，根据如算法 4 所示的间接转移地址反馈过程，在实际将该指令翻译成目标平台指令前，首先检查该间接转移的源平台目标地址(rax 寄存器中的值 0x4005ce)是否已经在二级地址映射表中，根据二级地址映射的构造方法，此时 $Index[0x4005ce-A_0]>0$，则 $Address[Index[0x4005ce-A_0]]=$L_0x4005ce 即为 callq *%rax 间接调用目标平台的跳转地址，此时使用翻译器正常对该指令翻译即可.

以间接跳转指令 jmpq *0x200c04(%rip)为例，若 $Address[Index[0x200c04(\%rip)-A_0]]==0$，则说明该跳转目标地址未存储在数组 *Address* 中，此时即跳转目标地址未定，需在 jmpq *0x200c04(%rip)指令执行前将 0x200c04(%rip)中的值通过文件进行反馈，并终止程序的执行，否则可能会造成不可预知的异常终止情况；待将该间接跳转的源目标平台转移地址确定后，即可更新基本块信息表 $T_b$，并在翻译时将该间接转移的源平台目标地址和目标平台地址存入数组 *Address*，如此即可在程序执行时准确定位该间接转移的目标地址.

## 4　实验和结果

下面对 FD-SQEMU 进行测试，测试的结果分别与 QEMU 和 SQEMU 进行比较. 测试内容主要包括 3 个方面：1)正确性测试；2)性能测试；3)反馈次数测试. 考虑到程序执行的局部性原理，性能测试中又分为热代码测试和整体性能测试 2 个方面.

热代码测试主要针对循环体和递归函数,对循环热代码的测试主要依托 NBENCH 测试集进行,对递归热代码的测试主要通过 FIBONACCI,N-QUEEN,QUICKSORT,MERGESORT 进行测试;整体性能测试主要通过选取 SPEC2006 测试集中的部分程序进行测试;与此同时,在各个对象的测试中,记录每个测试程序含有的所有及需要反馈处理的间接调用和间接跳转个数.

令 $T_{\text{FD-SQEMU}}$,$T_{\text{QEMU}}$,$T_{\text{SQEMU}}$ 分别表示 FD-SQEMU,QEMU,SQEMU 执行翻译后可执行程序的执行时间,定义速度比:

$$Ratio_1=\frac{1/T_{\text{FD-SQEMU}}}{1/T_{\text{QEMU}}}=\frac{T_{\text{QEMU}}}{T_{\text{FD-SQEMU}}},$$

$$Ratio_2=\frac{1/T_{\text{FD-SQEMU}}}{1/T_{\text{SQEMU}}}=\frac{T_{\text{SQEMU}}}{T_{\text{FD-SQEMU}}}.$$

### 4.1 实验环境

$M_s$ 为 x86 体系架构的机器,$M_t$ 为国产某处理器,主频 1.4 GHz,内存 4 GB,操作系统为 Fedora,内核版本 3.8.0,编译器为 gcc,版本 4.5.3.

QEMU 与 FD-SQEMU 运行在 $M_t$ 机器上.测试用例是 NBENCH 版本 2.2.3,手动实现的主流递归算法和 SPEC2006 测试集[23].

### 4.2 正确性测试

针对 FD-SQEMU 的正确性测试主要分为 2 个部分:1)指令翻译测试;2)实际程序翻译测试.

指令翻译测试借助 QEMU 自带的 test-i386 测试集,重点对 x86 架构用户态的指令进行了测试.依据 x86 指令分类情况,具体的测试结果如表 1 所示:

**Table 1 Correctness Test of Instruction Translation**

**表 1 指令翻译的正确性测试**

| Test Instruction | Test Result |
|---|---|
| Data Movement Instructions | √ |
| Destination Address Transfer Instruction | √ |
| Arithmetic Instruction | √ |
| Logical Instruction | √ |
| String Instructions | √ |
| Program Transfer Instructions | √ |
| Shift and Bit Instructions | √ |

在保证了单条指令翻译的正确性后,对实际程序翻译进行了正确性测试.具体测试过程如下:首先,在 x86 平台上编译 SPEC CPU2006 和 NBENCH 测试集等测试程序并运行记录结果;然后,在目标平台上运行翻译后的可执行程序,并与 x86 上运行结果相比对.部分程序的测试结果如表 2 所示:

**Table 2 Correctness Test of Program**

**表 2 实际程序的正确性测试**

| Test Case | Test Result |
|---|---|
| NUMERIC_SORT | √ |
| STRING_SORT | √ |
| BITFIELD | √ |
| FP EMULATION | √ |
| FOURIER | √ |
| IDEA | √ |
| HUFFMAN | √ |
| NUEURAL NET | √ |
| LU DECOMPOSITION | √ |
| FIBONACCI | √ |
| QUICKSORT | √ |
| MERGESORT | √ |
| N-QUEEN | √ |
| BZIP2 | √ |
| MILC | √ |
| SPECRAND | √ |
| MCF | √ |

Notes: "√" mean that the test cases have passed the correctness verification.

表 2 的实验结果表明,翻译后的目标程序执行结果与源 x86 平台上程序执行结果相同,验证了本文提出的翻译框架和算法能够进行正确翻译,做到了程序翻译的功能等价.

### 4.3 循环热代码性能测试

NBENCH 测试集的主要功能是通过计算一定时间内(一般是 5 s)单项测试代码块的循环迭代次数来评价系统性能,其中每一个测试块都是典型的热代码,具体的 NBENCH 测试集各部分功能如表 3 所示:

**Table 3 NBENCH Tasks**

**表 3 NBENCH 测试任务**

| Test Case | Tasks |
|---|---|
| NUMERIC_SORT | Sort an array of long integers |
| STRING_SORT | Sort an array of strings of arbitrary length |
| BITFIELD | Execute a variety of bit manipulation functions |
| FP EMULATION | A small software floating-point package |
| FOURIER | A numerical analysis routine for calculating series approximations of waveforms |
| ASSIGNMENT | A well-known task allocation algorithm |
| IDEA | A text and graphics compression algorithm |
| HUFFMAN | A relatively new block cipher algorithm |
| NUEURAL NET | A small but functional back-propagation network simulator |
| LU DECOMPOSITION | A robust algorithm for solving linear equations |

为了展示 FD-SQEMU 处理热代码的能力,我们分别用 FD-SQEMU,SQEEMU 和 QEMU 翻译系统对 NBENCH 测试集进行测试,并将测试结果进行对比.针对 NBENCH 测试集,FS-QEMU 和 SQEMU 相对于 QEMU 的性能提升情况如图 7 所示:

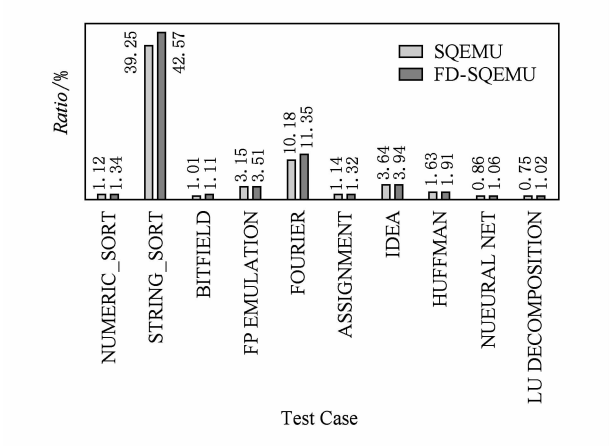



Fig. 7　Speedup ratio on NBENCH

图 7　FD-SQEMU,SQEMU 在 NBENCH 测试集上的加速比

如图 7 所示,对于不同的测试项目,SQEMU 和 QEMU 展示了各自的性能优势;而 FD-QEMU 保持了 SQEMU 和 QEMU 两个的性能优势,并有一定程度上的性能提升.对于 STRING_SORT 功能测试,因为字符串操作频繁使用 strcmp,strncmp 等字符串相关库函数,FD-SQEMU 和 SQEMU 分别获得 42.57 倍和 39.25 倍的加速比;类似地,因为 FOURIER,IDEA,FP EMULATION 功能测试使用了 pow,sin,cos,memmove 等库函数,FD-SQEMU 和 SQMEU 获得相应的加速比,并且针对这些功能的测试,FD-SQEMU 相对于 SQEMU 有 12%左右的加速比,这是因为 FD-SQEMU 在继续沿用 SQEMU 库函数本地化的基础上,采用了基本块的翻译方式,在翻译时进行了更多的优化,提高了生成代码的质量;对于 NUEURAL NET 和 LU DECOMPOSITION 功能测试,SQEMU 运行反而比 QEMU 要慢,但此时 FD-SQEMU 相比于 SQEMU 有较高的加速,最高加速可达到 36%,因为 SQEMU 在单条翻译 NEURAL NET 和 LU DECOMPOSITION 时缺少上下文关系,并引入了较多的冗余指令,而 FD-SQEMU 采用基本块翻译,一定程度上保留了源可执行文件的上下文关系,并在翻译中使用了少量的块内优化方法,所以对于 NUEURAL NET 和 LU DECOMPOSITION 功能测试,FD-SQEMU 相比于 QEMU 有少量的加速,相对于 SQEMU 的性能提升较为明显.对于 NBENCH 测试集,FD-SQEMU 相对于 QEMU 的平均加速比为 6.913,相对于 SQEMU 的平均加速比为 1.16.

从 NBENCH 的测试结果看,FD-SQEMU 继承了 SQEMU 的优点,并有效去除了 SQEMU 逐行翻译引入的部分冗余指令,FD-SQEMU 相比于 SQEMU 的平均性能提升了 16.14%.

### 4.4　递归热代码性能测试

递归算法是会反复执行的热代码,递归中对单一或某些函数的反复调用会产生大量的函数调用和返回指令,能否高效地处理函数调用,直接影响目标程序执行的效率,本文对典型的递归算法进行了测试,并给出了 FD-SQEMU 和 SQEMU 相比于 QEMU 的加速比,具体情况如图 8 所示:

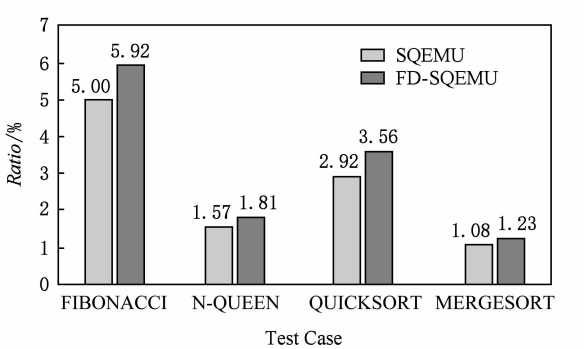



Fig. 8　Speedup ratio on recursive algorithms

图 8　FD-SQEMU,SQEMU 对 QEMU 在递归算法的加速比

如图 8 所示,FD-SQEMU 继承了 SQEMU 对递归热代码的处理能力,并在性能上有一定的提升,平均性能提升了 17.25%.这是因为 FD-SQEMU 继承了 SQEMU 的本地栈作为影子栈[24]实现用户自定义函数本地化调用的思想,有效减少了函数调用和返回时代码切换的消耗,保证了相对 QEMU 的性能提升;在此基础上 FD-SQEMU 使用基本块方式翻译,一是有效去除了 SQEMU 逐行翻译中的冗余代码并对代码进行了一定程度的块内优化,在翻译代码膨胀率上 FD-SQEMU 整体比 SQEMU 降低了约 20%.

### 4.5　整体性能测试

在对 FD-SQEMU 整体性能进行测试时,使用的是 SPEC2006 测试集.SPEC2006 测试集中的程序基本都是实际应用中常用到的程序,该测试集的执行结果能够反映出一个翻译系统的整体性能.图 9 展示了对 SPEC2006 中部分应用进行测试时 FD-SQEMU 和 SQEMU 相对 QEMU 的性能提升情况.

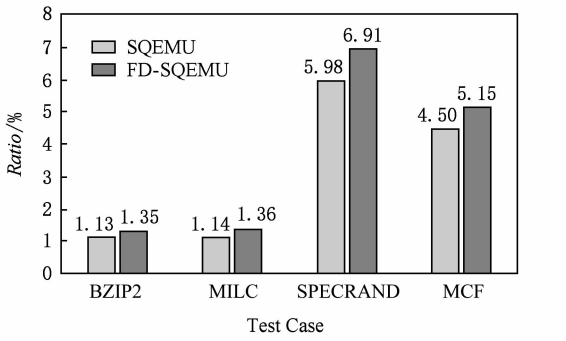



Fig. 9 Speedup ratio on part of SPEC2006

图 9 FD-SQEMU, SQEMU 在 SPEC2006 测试集上的加速比

如图 9 所示, 对于 BZIP2, MILC 程序, FD-SQEMU 和 SQEMU 相比于 QEMU 获得了 1 倍多的加速, 这是因为这些实际应用中大量代码迭代次数少, 函数重复调用较少, FD-SQEMU 和 SQEMU 在目标代码执行阶段不需要再翻译和代码维护, 消除了 QEMU 的代码块切换的时间, 提高了目标平台代码的执行效率; 在 SPEC 2006 测试集中 SPECRAND 和 MCF 程序获得了较高的加速比, 对于 SPECRAND 程序, FD-SQEMU 相比于 QEMU 的加速比高达 6.91, 这是因为 SPECRAND 程序反复使用到了产生随机数的库函数, 对于 MCF 程序同样也是如此.

与 SQEMU 相比, 对于 SPEC2006 中调用库函数较少的程序进行测试, FD-SQEMU 比 SQEMU 整体性能提升了 19.1%; 对于反复调用库函数的 SPEC2006 程序进行测试, FD-SQEMU 对 SQEMU 性能仅提升了约 15%, 这是因为 FD-SQEMU 相比于 SQEMU 在对库函数处理上并没有较大的改进. 总体上, 对部分 SPEC2006 程序的测试, FD-SQEMU 相比于 SQEMU 整体性能平均提升了 16.75%.

### 4.6 反馈情况测试

作为反馈式静态二进制翻译器 FD-SQEMU, 在翻译程序时所需反馈次数也是其性能的重要衡量标准.

在测试时, 针对可能存在的因参数不同执行路径不同的情况, 本文做了多种参数的测试, 并将各测试实例代码段中所有间接跳转和间接调用以及需要通过反馈处理的间接跳转和间接调用情况进行统计, 如表 4 所示.

如表 4 中所示, FD-SQEMU 对各测试项的反馈运行次数情况并不相同, MCF, FIBONACCI, QUICKSORT, MERGESORT, N-QUEEN, SPECRAND 中反馈次数为 0, 这是因为在这些应用程序中的间接跳转和间接调用目标地址都为某一函数或者基本块的首地址, 不存在跳转到某一基本块中间的情况, 此时在数组 *Address* 中可以查询得到该跳转目标地址在目标平台上的指令地址, 不需要反馈执行; 而对于 NBENCH 测试集的 BZIP2 和 MILC, 因为这些程序中部分间接跳转和间接调用目标地址在原先划分的基本块中间, 所以需要通过反馈来获得该地址并对原先基本块重新划分, 以方便将该地址存入二级地址映射表中进行间接转移目标定位. 实际上在 NBENCH, BZIP2, MILC 中虽然有较多的间接跳转和间接调用指令, 但是需要进行反馈处理的并不多, 其中大部分间接转移目标地址为某一子程序的起始地址或者某一基本块的起始地址. 实际完成源程序翻译所需的反馈次数与该程序中间接转移目标地址在之前划分的基本块中间的转移指令数成线性关系. 最初的基本块信息表 $T_b$ 的生成, 直接影响着程序需要执行、反馈、重翻译的次数.

**Table 4 Translation Feedback Situation of FD-SQEMU**

**表 4 FD-SQEMU 翻译反馈情况**

| Test Case | Indirect Jump Instructions Sum /Solved by Feedback | Indirect Call Instructions Sum/Solved by Feedback | Number of Feedback |
|---|---|---|---|
| NBENCH | 9/3 | 4/0 | 3 |
| FIBONACCI | 2/0 | 2/0 | 0 |
| QUICKSORT | 2/0 | 2/0 | 0 |
| MERGESORT | 2/0 | 2/0 | 0 |
| N-QUEEN | 2/0 | 2/0 | 0 |
| BZIP2 | 6/2 | 20/1 | 3 |
| MILC | 5/2 | 9/2 | 4 |
| SPECRAND | 0/0 | 5/0 | 0 |
| MCF | 1/0 | 5/0 | 0 |

## 5 总 结

本文在分析现有静态二进制翻译中解决间接转移问题方法的基础上, 针对现有间接转移问题处理方法中线性遍历翻译方式代码优化较少、冗余代码较多的缺陷, 对基于动态 QEMU 的静态二进制翻译器 SQEMU 进行了改进, 提出了基于基本块翻译的反馈式静态二进制翻译框架 FD-SQEMU, 并结

合二级地址映射表实现了间接转移目标地址的快速映射.首先,FD-SQEMU 在继承 SQEMU 库函数本地化的基础上,引入基本块翻译方式,解决了 SQEMU 逐行翻译无法兼顾程序上下文进行块内优化以及引入大量冗余代码的缺陷;接着,采用反馈机制,并结合新的二级地址映射表及查找算法,以较高效率解决了静态二进制翻译中间接跳转和间接调用中的代码发现及定位的问题;最后,通过与 SQEMU 和 QEMU 的性能对比实验,验证了反馈式静态二进制翻译器 FD-SQEMU 的正确性和有效性.

但在反馈式静态二进制翻译中,针对源二进制程序的最初基本块信息表 $T_b$ 的建立直接关系到反馈次数,如何根据源二进制程序特性建立更合适的起始基本块信息表以减少反馈次数还需要进一步分析;另外,在该翻译过程中,采用的代码优化较少,还可利用现有的代码优化方法进一步对翻译代码进行优化,以生成更高质量的目标代码,提高目标代码的执行效率.

## 参　考　文　献


[1] Altman E R, Kaeli D, Sheffer Y. Welcome to the opportunities of binary translation [J]. Computer, 2000, 33(3): 40-45

[2] Shan Zheng, Guo Haoran, Pang Jianmin. BTMD: A framework of binary translation based malcode detector [C] //Proc of the 4th Int Conf on Cyber-Enabled Distributed Computing and Knowledge Discovery. Los Alamitos, CA: IEEE Computer Society, 2012: 39-43

[3] Ma Jinxin, Li Zhoujun, Hu Chaojian, et al. A reconstruction method of type abstraction in binary code [J]. Journal of Computer Research and Development, 2013, 50(11): 2418-2428 (in Chinese)
(马金鑫, 李舟军, 忽朝俭, 等. 一种重构二进制代码中类型抽象的方法[J]. 计算机研究与发展, 2013, 50(11): 2418-2428)

[4] Zhao Tianlei, Tang Yuxing, Fu Guitao, et al. Accelerating program behavior analysis with dynamic binary translation [J]. Journal of Computer Research and Development, 2012, 49(1): 35-43 (in Chinese)
(赵天磊, 唐遇星, 付桂涛, 等. 利用动态二进制翻译加速应用程序行为特征分析[J]. 计算机研究与发展, 2012, 49(1): 35-43)

[5] Chen Jingsong, Lan Xuhui, Wei Zhong. Software transplant based on instruction simulation [J]. Electronic Instrumentation Customer, 2010(1): 55-57

[6] Chernoff A, Herdeg M, Hookway R, et al. FX! 32: A profile-directed binary translator [J]. IEEE Micro, 1998, 18(2): 56-64

[7] Cifuentes C, Emmerik M V. UQBT: Adaptable Binary Translation at Low Cost [M]. Los Alamitos, CA: IEEE Computer Society, 2000

[8] Liao Yin. Dynamic binary translation modeling and parallelization research [D]. Hefei: University of Science and Technology of China, 2013 (in Chinese)
(廖银. 动态二进制翻译建模及其并行化研究[D]. 合肥: 中国科学技术大学, 2013)

[9] Liu Xiaonan. Research on key technologies of binary translation for the domestic CPU [D]. Zhengzhou: PLA Information Engineering University, 2014 (in Chinese)
(刘晓楠. 面向国产处理器的二进制翻译关键技术研究[D]. 郑州: 解放军信息工程大学, 2014)

[10] Li Jianhui, Ma Xiangning, Zhu Chuanqi, et al. Research on dynamic binary translation and optimization [J]. Journal of Computer Research and Development, 2007, 44(1): 161-168 (in Chinese)
(李剑慧, 马湘宁, 朱传琪, 等. 动态二进制翻译与优化技术研究[J]. 计算机研究与发展, 2007, 44(1): 161-168)

[11] Jia Ning, Yang Chun, Wang Jing, et al. SPIRE: Improving dynamic binary translation through SPC-indexed indirect branch redirecting [C] //Proc of the 9th Int Conf on Virtual Execution Environments. New York: ACM: 2013: 1-12

[12] Hiser J D, Williams D, Hu Wei, et al. Evaluating indirect branch handling mechanisms in software dynamic translation systems [J]. ACM Transactions on Architecture and Code Optimization, 2011, 8(2): Article 9

[13] Sun Tingtao, Yang Yindong, Yang Hongbo, et al. Return instruction analysis and optimization in dynamic binary translation [C] //Proc of the 4th Int Conf on Frontier of Computer Science and Technology. Los Alamitos, CA: IEEE Computer Society, 2009: 435-440

[14] Liao Yin, Sun Guangzhong, Jiang Haitao, et al. All registers direct mapping method in dynamic binary translation [J]. Computer Applications & Software, 2011, 28(11): 21-24

[15] Cifuentes C, Malhotra V M. Binary translation: Static, dynamic, retargetable? [C] //Proc of the 18th Int Conf on Software Maintenance. Piscataway, NJ: IEEE, 2002: 340-349

[16] Chen Long, Wu Chenggang, Xie Haibin, et al. Using graph match method to resolve multi-way branch in binary translation [J]. Journal of Computer Research and Development, 2008, 45(10): 1789-1798 (in Chinese)
(陈龙, 武成岗, 谢海斌, 等. 二进制翻译中解析多目标分支语句的图匹配方法[J]. 计算机研究与发展, 2008, 45(10): 1789-1798)

[17] Lu Shuaibing, Pang Jianmin, Shan Zheng, et al. Retargetable static binary translator based on QEMU [J]. Journal of Zhejiang University, 2016, 50(1): 158-165 (in Chinese)

(卢帅兵, 庞建民, 单征, 等. 基于 QEMU 的跨平台静态二进制翻译系统[J]. 浙江大学学报, 2016, 50(1): 158-165)

[18] Shen Bor-yeh, You Jyun-Yan, Yang Wuu, et al. An LLVM-based hybrid binary translation system [C] //Proc of the 7th Int Symp on Industrial Embedded Systems (SIES). Piscataway, NJ: IEEE, 2012: 229-236

[19] Yang Hao, Tang Feng, Xie Haibin, et al. Library function disposing approach in binary translation [J]. Journal of Computer Research and Development, 2006, 43(12): 2174-2179 (in Chinese)
(杨浩, 唐锋, 谢海斌, 等. 二进制翻译中的库函数处理[J]. 计算机研究与发展, 2006, 43(12): 2174-2179)

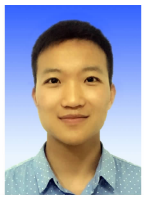
[20] Song Fangmin. Proof of equivalence of program [J]. Journal of Computer Research and Development, 1989, 26(4): 31-38 (in Chinese)
(宋方敏. 程序的等价性证明[J]. 计算机研究与发展, 1989, 26(4): 34-38)

[21] Bellard F. QEMU, a fast and portable dynamic translator [C] //Proc of the 2nd Conf on USENIX Technical. Berkeley, CA: USENIX Association, 2005: 41-46

[22] Luo Yan. Research on Dynamic Binary Translation and Optimization based on QEMU [D]. Hangzhou: Zhejiang University, 2013 (in Chinese)
(罗艳. 基于 QEMU 的动态二进制翻译优化研究[D]. 杭州: 浙江大学, 2013)

[23] Henning J L. SPEC CPU2006 benchmark descriptions [J]. ACM SIGARCH Computer Architecture News, 2006, 34(4): 1-17

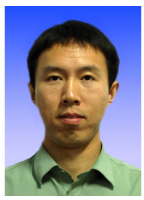
[24] Ince T, Yamada K, Caprioli P, et al. Technologies for shadow stack manipulation for binary translation systems: US, 9477453 B1 [P]. 2016-10-25



**Wang Jun**, born in 1992. PhD candidate. His main research interests include computer architecture and high performance computing.

**Pang Jianmin**, born in 1964. PhD, professor, PhD supervisor. Senior member of CCF. His main research interests include computer architecture, information security and high performance computing.

**Fu Liguo**, born in 1989. PhD. His main research interests include advanced compilation and high performance computing.



**Yue Feng**, born in 1985. PhD. Member of CCF. His main research interests include advanced compilation and high performance computing.

**Zhang Jiahao**, born in 1991. Master. His main research interests include computer architecture and high performance computing.