DOI:10.7544/issn1000-1239.2019.20170871
56(8): 1783-1791, 2019

动态二进制翻译中库函数处理的优化

傅立国 庞建民 王 军 张家豪 岳 峰
(数学工程与先进计算国家重点实验室 郑州 450001)
(flg_njlg@163.com)

Optimization of Library Function Disposing in Dynamic Binary Translation

Fu Liguo, Pang Janming, Wang Jun, Zhang Jiahao, and Yue Feng
(*State Key Laboratory of Mathematical Engineering and Advanced Computing*, *Zhengzhou* 450001)

Abstract In the research of cross-platform migration without source code, efficiency is the main bottleneck restricting the development of dynamic binary translation technology. Using the method of disposing the local function can effectively improve the performance of binary translation by jacketing and replacing. However, in practical applications, as the number of library function calls in the source program or the number of the library function supported by translators increasing, the benefit of using disposing the local function is on decrease. The querying useless overhead in library function disposing grows which weakens and reduces the optimization effect of the method of disposing the local function. In order to address this kind of problem, a method is proposed based on the properties combined with dynamic and static translation. It is based on the characteristic of dynamic binary translation and the using of disposing the local function. The overhead for the query decreases with the method by preprocessing the query table and realizing the query with Hash function. It can map the source program addresses to corresponding processing function rapidly. Realized on a dynamic translator QEMU, the optimization method is implemented and tested. Experiments verify the effectiveness of this method to reduce query overhead in the process of using disposing the local function with dynamic translation.

Key words dynamic binary translation; local function disposing; query overhead optimization; static pre-processing; Hash function

摘 要 在无源跨平台移植的研究中,效率是制约动态二进制翻译技术发展的主要瓶颈.使用本地函数封装替换的翻译处理方式可以有效提高二进制翻译的性能.然而在实际应用中,随着源程序中库函数调用次数或者翻译器支持库函数数量的增长,库函数处理过程的无用查询开销随之增长,减弱了该方法的优化效果.针对此类问题,结合动态二进制库函数处理过程内在动静结合的性质,提出了将查询信息静态预处理,使用散列函数实现查询过程的优化方法,实现了源程序中库函数地址到相对应处理函数的快速映射,降低了查询开销.基于动态二进制翻译器 QEMU 实现并测试了优化方法,通过实验验证了该方法降低库函数处理过程中查询开销的有效性.

关键词 动态二进制翻译;库函数处理;查询开销优化;静态预处理;散列函数

中图法分类号 TP314

收稿日期:2017-11-20;**修回日期**:2019-04-09

基金项目:国家自然科学基金项目(61472447)
This work was supported by the National Natural Science Foundation of China (61472447).

通信作者:庞建民(jianmin_pang@126.com)

动态二进制翻译技术作为一种特殊的即时编译技术,既可以用于跨架构的程序移植,也能够用于程序特征分析和性能调优,因而在体系结构优化、程序性能优化、安全分析以及软件移植的研究中备受关注.随着虚拟化技术的兴起,在跨指令集架构的虚拟化以及全软件虚拟化的项目中,二进制翻译技术得到了更广泛的应用[1].

然而较低的效率限制了该技术解决跨架构无源移植问题的实用性.为了提高动态二进制翻译的效率,针对源平台的体系架构特征,充分利用宿主平台的软硬件资源成为相关研究的核心点.

不论是使用指令模拟技术,还是使用软硬件协同又或者是纯软件模拟,翻译的正确性和翻译的效率总是各种翻译方法研究的要点[2-3].在动态二进制翻译效率优化的研究中,主要针对 2 部分内容进行展开:1)提高翻译过程的效率,即加快源指令的分析和本地指令的生成;2)改善翻译方法,使生成的本地代码执行更加高效[4-6].

2 类优化既相互联系也相互制约.简单的翻译方式所生成的代码执行效率相较而言比较低,而翻译过程中优化本地代码生成的开销也会削减执行时代码质量提高获取的收益[7].

针对这种情况,使用库函数处理的方法可以有效地提高动态二进制翻译执行的效率[8-9].该方法对源程序中库函数的执行进行了本地函数的封装和替换处理.该方法精简了原有的翻译过程,将函数指令序列翻译代价转换为一条函数调用指令的代价,并且利用源码编译优化的优势.相较于动态翻译中基于指令语义解析生成的代码,本地编译优化的代码具有更高的执行效率.因此,作为一种动静结合的翻译方式,库函数处理可以有效地提高动态二进制的效率.

本质上,库函数处理过程类似于共享库函数的动态链接过程.当源程序中调用的库函数数量以及翻译器支持的库函数数量较多时,这种查询链接过程的开销会降低库函数处理应有的收益,从而影响动态二进制翻译的整体效率和实际应用效果.

针对当源程序中库函数调用次数以及翻译器支持的库函数数量较多时,库函数处理机制中查询链接过程的开销过大问题,本文基于库函数处理的性质,提出了静态预处理散列查询表的优化方法.基于动态二进制翻译器 QEMU 实现了库函数处理的机制,然后在此基础上进行了优化方法的实现,最后通过实验验证了该优化方法的有效性[10].

1 动态二进制翻译中的库函数处理

1.1 共享库函数的动态链接

程序执行时共享库函数的动态连接过程如图 1 所示:

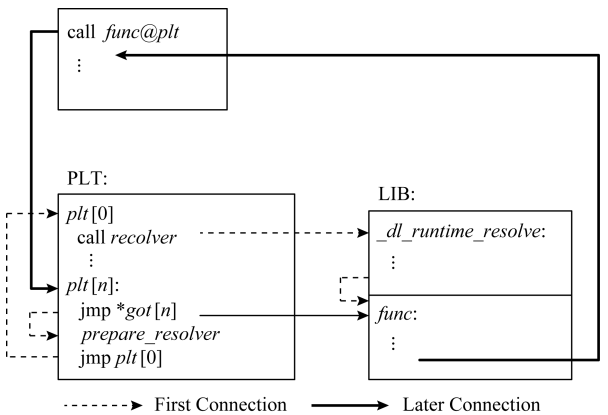


Fig. 1 The dynamic connection process of shared library functions

图 1 共享库函数的动态连接过程

图 1 中的粗实线和虚线描述了第 1 次重定位和调用函数 *func* 的过程.二进制程序执行到 call *func*@*plt* 指令时,指令中的目标地址 *func*@*plt* 指向 PLT(procedure linkage table)段的第 *n* 个入口即地址 *plt*[*n*].进入 PLT 段对应的入口后,程序执行指令 jmp **got*[*n*],跳转的目标地址取决于 GOT(global offset table)段第 *n* 个项中存储的地址.

若是第 1 次调用该入口对应的函数,*got*[*n*]中存储的是 jmp **got*[*n*]指令下一条指令的地址,此时程序按照地址连续执行,准备函数名解析相关的参数,然后通过执行指令 jmp *plt*[0],调用解释链接器(即函数*_dl_runtime_resolve*)解释器不仅会完成函数 *func* 的首次调用,也会将其在 LIB 段中实际的存储地址回填 *got*[*n*].

此后程序对函数 *func* 的调用,当执行到 PLT 段入口的 jmp **got*[*n*]指令时,即可直接跳转到函数 *func* 在 LIB 中的入口地址,实现对函数的有效调用.函数重定位后的执行过程由图 1 中粗实线和细实线构成.

综上可知,共享库函数的动态链接过程完成了一次基于外部链接函数 PLT 段入口地址的查询,即实现了函数 *func* 入口地址 *func*@*plt* 到其在 LIB 段中实际存储入口地址的映射.

1.2 动态二进制翻译中的库函数处理

类似于处理器的执行过程,动态二进制翻译过程可以概括为查找、翻译、执行 3 个子过程[11].而库函数处理区别于一般的二进制翻译过程主要体现在查找和翻译 2 个过程中.

图 2 给出了动态二进制翻译过程中库函数 2 种不同的翻译处理方式.

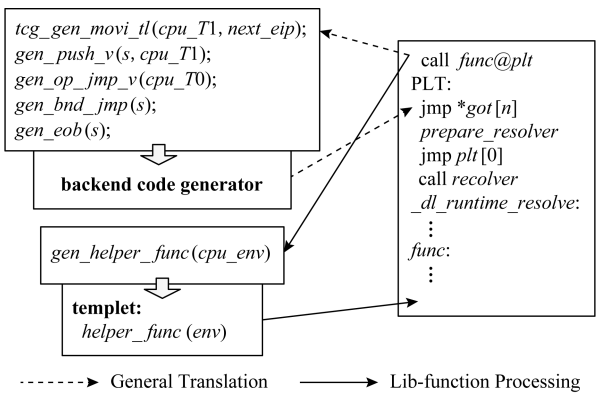


Fig. 2 Dynamic translation of shared library functions

图 2 共享库函数的翻译过程

图 2 右侧描述了图 1 中库函数 *func* 第 1 次被调用时的指令序列;图 2 左侧是对右侧指令序列的动态翻译处理过程.

左侧虚线箭头描述的是对指令 call 一般的动态二进制翻译处理过程,先将 call 指令解释成右上的 5 条中间指令,然后由后端代码生成器将中间指令转换为本地代码。执行完生成的本地代码后,会继续获取待翻译的指令进行翻译,以此反复.

左侧实线箭头描述的是使用库函数处理方法对指令 call 的翻译处理过程.对函数调用地址 *func*@*plt* 进行分析,可以将其解释成特殊的中间表示.在翻译器执行时,解释中间表示 *gen_helper_func* 需要调用翻译器内建的模板函数 *helper_func*,该模板函数是与源平台函数 *func* 功能相同的本地实现,但同时也封装了对应的传参和返回值回写机制,以此作为对左侧 *func* 函数指令序列本地翻译的结果.

图 2 右侧的指令序列从功能上可以分为函数解析重定位和函数执行 2 部分.从指令 call *func*@*plt* 开始到函数 *func* 之前,这部分指令完成了函数解析重定位的运行环境的维护工作,与原应用的逻辑功能无关.按照一般的翻译处理方式,序列中的每条指令都会被翻译成中间语言表示,然后再被后端转换成本地代码.而库函数处理的翻译方式,在翻译器内部实现了函数的解析重定位,再调用翻译器提供的模板函数(即封装好的函数)作为函数 *func* 在本地翻译结果的执行.

在使用库函数处理的方式翻译 call 指令时,需要使用基于共享库函数 PLT 段入口地址的语义信息.因此,在翻译时需要先解析该地址的语义,查询对应的函数名,然后在翻译器支持的模板函数中查找相应的处理函数,最终完成共享库函数调用全过程的翻译.

由于语义分析的层次更高,使用本地编译优化的模板处理函数,其代码的执行效率远远超过一般指令序列动态翻译所生成代码的执行效率,同时还节约了相应指令序列的翻译时间.

综上可知,在库函数处理的翻译过程中,有 2 次查询的操作:1)根据 call 指令的目标地址判断是否是 PLT 段的入口,并根据入口的序号查询函数的名称;2)根据函数的名称查询翻译器模板函数中是否有对应函数的处理函数.当源程序中 call 指令的数量或者翻译器模板函数的数量增长时,查询开销会随之增长.最差的情况就是,翻译器模板函数没有对应的处理函数时,此时没有任何收益却额外添加了 2 次查询的开销.

2 库函数处理过程的解析

一般的翻译处理方法和库函数处理的翻译方法,其差异性主要体现在 2 个方面:

1) 语义解析的层次不同.一般的翻译处理是基于指令的语义解析;库函数处理的翻译过程是基于函数的语义解析,即在翻译 call 指令时需要结合跳转指令的目标地址解析调用的函数,继而完成函数语义的解析.

2) 指令流的执行语义有所改变.在一般的翻译过程中,源程序函数解释链接重定位的过程中对应的指令序列是被逐一翻译的;在基于函数语义的翻译过程中,源程序链接重定位过程的指令序列并未被翻译,函数的解析过程被当做 call 指令解析的一部分,翻译执行的代码是模板函数中已经本地化封装的处理函数.

使用库函数处理的动态二进制翻译模式,在翻译 call 指令时,结合跳转目标地址的解析以确定调用函数的名称,然后根据翻译器对该函数的支持情况确定翻译的方式.

因此库函数处理即在普通翻译模式中对 call 指令的翻译添加了一类特殊处理.

2.1 库函数处理的 call 指令翻译

库函数处理中 call 指令的翻译流程如图 3 所示：

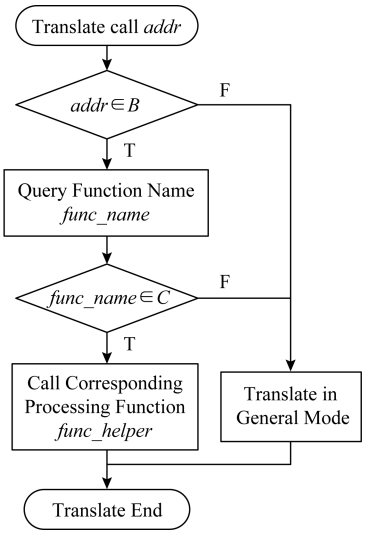


Fig. 3 Translation of call instruction in library function disposing

图 3 库函数处理中 call 指令的翻译

在图 3 中，B 表示源程序中外部链接共享函数的 PLT 段入口地址的集合，C 表示翻译器中支持库函数处理的库函数集合名称集合.如图 3 所示，在一次完整库函数处理的翻译过程中，需要 2 次查询判断：

1) 查询 $addr$ 对应的函数名 $func\_name$；

2) 查询名为 $func\_name$ 的库函数对应的处理函数 $func\_helper$.

不论哪次查询失败，该 call 指令的翻译都会采用一般的翻译模式.查询的本质即完成了从跳转指令目标地址到翻译处理函数的映射.

设源程序中指令 call 调用的目标地址集合为 A，设一般翻译执行模式的处理函数为 d，那么库函数处理的查询过程完成了集合 A 到集合 $C\cup\{d\}$ 的映射.该映射过程用函数 SA 表示.则有：

$$\forall a\in A, SA(a)\in(C\cup\{d\}). \tag{1}$$

根据式(1)中元素 a 映射的结果可以对集合 A 中的元素划分：

$$\begin{cases}\forall a\in F, SA(a)\in C;\\ \forall a\in\overline{F}, SA(a)\in\{d\};\\ A=F+\overline{F}, F\cap\overline{F}=\varnothing.\end{cases} \tag{2}$$

其中，SA 表示 call 指令跳转目标地址到 call 指令翻译处理函数的映射，因为库函数翻译处理函数和其处理的库函数名称是一一对应的，式(2)中 C 表示的是翻译器中库函数翻译处理函数的集合.

2.2 库函数名称的解析

集合 B 中的库函数的名称作为集合 A 向集合 C 映射的过渡，是动态二进制翻译过程中源程序地址语义和翻译处理函数语义的唯一标识.

因此从集合 A 向集合 B 映射的过程，实质上是对 call 指令地址语义的解析.除了需要判断目标地址是否属于 PLT 段的入口，还需要解析其对应的函数名.

在确定应用二进制接口(application binary interface, ABI)规范下，外部链接函数名称的解析过程是固定的.以 elf(executable and linkable format)格式为例，其共享库函数名称解析过程如图 4 所示[12].

首先根据函数 $func$ 在 PLT 段入口序号 n 在 RELA.PLT 段中索引查询符号名的索引，然后依次在 DYN.SYM 段、SYM.TAB 段和 STR 段中索引查询，最终得到外部链接函数 $func$ 名称的字符串.

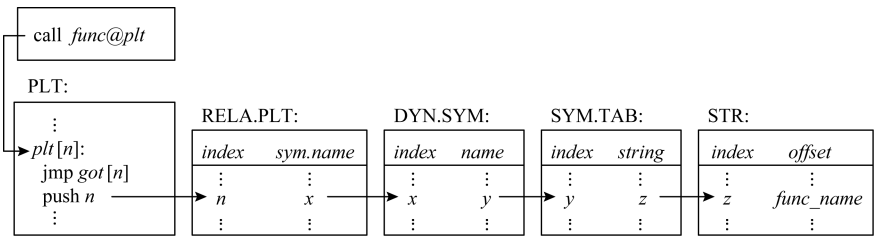


Fig. 4 The name analyzing of shared library function

图 4 共享库函数的名称解析

3 优化的分析及实现

库函数处理使用本地库函数封装，在不改变封装方式和实现方法的情况下，这部分过程的执行代价是固定不变的.因此本文的分析和优化主要针对 call 指令翻译处理的判断确认过程.

根据以上的分析，在使用库函数处理的翻译过程中，查询过程主要完成了函数名称的解析以及对应处理函数的查询.

因此查询的优化主要从 2 个方面进行:1)优化查询表的组织结构;2)优化查询实现的方法.

3.1 查询表的优化

对于查询成功的跳转目标地址而言,在 2 次查询(PLT 段入口地址查询、翻译器支持库函数处理的函数名称查询)以及一次解析(使用 PLT 段入口地址解析函数名称)过程中,需要使用若干张查询表.特别是共享库函数名称解析的过程中,需要基于文件规范从多个表格中进行多次索引查询获取函数名称.

针对一个确定的源程序,其外部链接函数名称解析所需的数据是静态可得的.因此,PLT 段入口地址和函数名称的对应关系是能够进行静态分析的.静态的解析过程与动态链接重定位的解析过程相同,使用文件中若干个段中的索引和数据获取描述函数名称的字符串.类似的解析过程在类似 objdump 的反汇编工具中均有实现和应用.

针对一个确定的翻译器,其支持库函数处理的共享库函数也是确定的.这些函数的名称和其处理函数的名称也是一一对应且静态可知的.

在翻译执行前,通过静态预分析源程序和翻译器的相关信息,就可以获取库函数处理过程中所需的若干查询表.因此,可以在预分析的基础上对查询表进行合并.最终获得 PLT 段入口地址到 call 指令翻译处理函数的查询表.

按照匹配的查询方式,查询的平均开销和每次查询表中项目的规模成正比.将查询表合并后,查询表的规模始终和 PLT 段的入口数相同,避免了之前查询过程中匹配查询中的额外的开销.

3.2 查询方法的优化

在程序执行过程中,call 指令调用的目标地址是随机且离散的.然而对于任意符合相应规范的二进制可执行文件,其 PLT 段的构成总是有规律的.

通过源程序的静态预分析,每个源程序中的 PLT 段入口地址是静态可知且地址偏移相等.设源程序 PLT 段中有 m 个入口,可以通过构建一个简单的散列函数将任意调用地址 $addr$ 散列到 $m+1$ 个槽中.

由于库函数处理过程对被处理的库函数具有较为严格的限制,对于大部分的源程序而言,查询失败的概率远大于查询成功并进行处理的概率.因此在构建查询算法的过程中,应当优先针对查询失败的情况进行处理,从整体上降低平均的查询开销.

根据之前的分析,调用外部链结函数的 call 指令其目标地址一定是 PLT 段入口,因此跳转目标地址不在 PLT 段内的地址一定会查询失败.其次,跳转目标地址减去 PLT 段首地址不能整除一个记录长度的地址一定查询失败.最后,将整除 PLT 段记录长度的结果作为 PLT 段入口地址查询的索引值.为保持查询的完备性,将查询失败的索引值置为 −1.

根据索引值,可以获取该 call 指令翻译的处理函数.在静态预处理的查询表中,根据翻译器的支持情况,将处理函数的函数指针填到相应索引对应的值中,而翻译器不支持的 PLT 段入口地址索引以及索引 −1 对应的值则为普通 call 指令翻译处理的函数指针.

3.3 优化的查询实现及其算法描述

查询表的优化及查询算法的优化实现如图 5 所示.

图 5 描述了查询表的优化,即将 TAB_A,TAB_B,TAB_C,TAB_D 合并成 TAB_E 的过程.其中表 TAB_A 可通过对二进制程序静态分析获取;表 TAB_B 描述了可封装翻译处理函数名 $func\_name$ 和翻译器模板函数入口 $helper\_pointer$ 的对应关系;通过 TAB_A 和 TAB_B 即可实现库函数处理的 2 次查询.

TAB_C 对 PLT 段的入口 plt_entry 进行了排序,目的是为了可以实现指令 call 目标地址变量 $addr$ 在 plt_entry 中的快速查询;TAB_D 则是对模板函数入口 $helper\_pointer$ 进行了排序,因为在翻译器执行期间表 TAB_D 是常驻内存的,并且

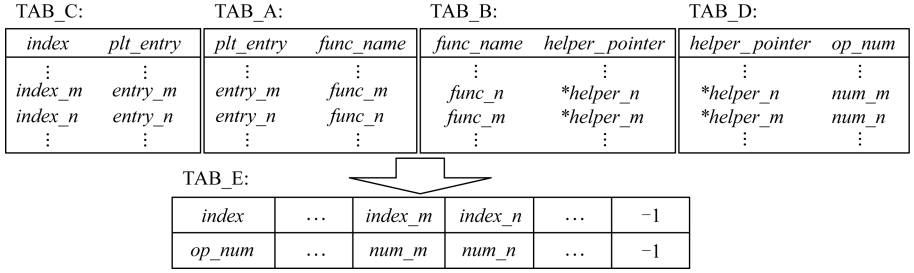


Fig. 5 Optimized query table and query processing

图 5 优化的查询表及查询过程

翻译器自身构建的一些特殊指令辅助翻译的实现也是通过查询该表调用对应模板函数实现的,构建使用模板函数调用号 *op_num* 实现模板函数的调用可以与实验翻译器自身的执行机制更好的融合.

通过索引表的合并,最终得到表 TAB_E,完成了入口地址索引 *index* 到模板函数地址索引号的对应表.索引 *index* 中记录值−1 表示无效的索引号;对应项 *op_num* 中的记录值−1,表示无对应的模板函数,需要调用一般的动态翻译处理过程.查询表的合并过程可以在翻译器加载源平台二进制程序的过程中完成,进而减少了原本在翻译过程中进行动态查询的开销.

经过查询表和查询过程的优化,将原有的多次查询过程转换为 1 次计算和 2 次索引取值的过程,即对指令 call 的目标地址变量 *addr* 使用函数 *hash* 进行计算,使用函数 *tab_e* 和函数 *tab_d* 实现对表 TAB_E 和表 TAB_D 的索引取值,整个映射过程为

$$helper\_pointer = tab\_d[tab\_e[hash(addr)]]. \quad (3)$$

使用变量 *index* 表示翻译过程中模板函数的调用号,优化的查询算法描述如算法 1 所示:

算法 1. 优化的查询算法.

功能:构造源程序地址到目标平台程序地址的映射表;

输入:call 指令目标地址 *addr*;

输出:对应模板函数入口地址 *helper_pointer*.

① 初始化数组 *tab_e*,获取 *plt_start* 起始地址数值和长度 *length*;
② for each *addr*
③ if *addr* 不在 *plt* 段范围内
④ *index*=−1;
⑤ else
⑥ *index*=*tab_e*[(*addr*-*plt_start*)/*length*];
⑦ end if
⑧ if *index*=−1
⑨ 使用一般方式翻译指令 call;
⑩ else
⑪ 调用 * *tab_d*[*index*]处的模板函数.
⑫ end if
⑬ end for

4 实验与结果

静态预处理动态查询的查询表,将匹配查询替换为计算查询,针对动态二进制翻译中库函数处理中查询过程优化的有效性是具有理论依据的.为了进一步验证优化的效果,在动态二进制翻译器 QEMU 上实现了库函数处理的翻译机制,再在此基础上实现了本文提出的优化方法,最后在课题研究所使用的 SW 平台上进行了实验验证.关于库函数处理机制的正确性验证在课题组之前的研究中已经通过了相关的正确性测试[13-14].

4.1 实验环境

实验测试环境的环境如表 1 所示:

Table 1 Binary Translation Environments

表 1 二进制翻译环境

Items	Source Platforms	Target Plantforms
OS	Fedora 2.6.27.5-117.fc10.i686	NeoKylin 4.0.0
CPU	Intel® Core™2 Quad CPU Q9500@2.83 GHz	SW410
Compiler	gcc-5.1.1	gcc-5.1.1

测试平台采用的是国产申威 SW410 处理器,是一款面向桌面平台场景应用的处理器,是神威太湖之光系统采用的处理器 SW20610[15] 的精简版.SW410 较 SW20610 只保留了 4 个主频为 1.4 GHz 主核,精简了对应的从核架构.

实验使用的翻译器为 qemu-2.5.1,通过修改,支持标准 C 库中 math.h,stdio.h,string.h 下定义的大部分函数以及其他部分库函数的翻译处理.

实验使用的测试集包括 qemu 官网推荐的性能评测用例 nbench 以及标准性能测试集 spec cpu2006.具体的测试用例包括 nbench-2.2.3 benchmark suite 中的所有用例,如表 2 所示[16];以及 spec cpu2006 中所有的 C 语言程序,如表 3 所示[17]:

Table 2 nbench Description

表 2 nbench 测试用例简介

Test Case	Tasks
NUMERIC_SORT	Sorts an array of long integers
STRING_SORT	Sorts an array of strings of arbitrary length
BITFIELD	Executes 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
LUDECOMPOSITION	A robust algorithm for solving linear equations

Table 3 spec cpu2006 Description

表 3 spec cpu2006 测试用例简介

Test Case	Tasks
401.bzip2	Compression
403.gcc	C Compiler
429.mcf	Combinatorial Optimization
445.gobmk	Artificial Intelligence: go
456.hmmer	Search Gene Sequence
458.sjeng	Artificial Intelligence: chess
462.libquantum	Physics: Quantum Computing
464.h264ref	Video Compression
433.milc	Physics: Quantum Chromodynamics
470.lbm	Fluid Dynamics
482.sphinx3	Speech Recognition

4.2 实验结果

实验对每个测试用例翻译执行的时间进行 3 组测试,分别为一般的翻译方式、未经优化查询的库函数处理翻译方式和使用优化查询的库函数处理翻译方式.通过计算可得无优化查询库函数处理翻译方式的加速比用 Original 表示,优化查询过后库函数处理翻译方式的加速比用 Optimized 表示.

测试结果如图 6 和图 7 所示.在未优化查询过程时,部分用例库函数处理的收益小于查询开销,因此整体呈现负加速.为了形成鲜明对比,图 6 和图 7 中的纵轴使用自然数指数为坐标.当数据柱高于 1 时未正加速,低于时为负加速.对比图 6 和图 7 中的测试结果,从整体上看,本文优化方法对 spec 测试用例的效果要优于对 nbench 测试用例的优化效果,这是测试集自身的特性造成的.nbench 测试集主要是针对处理器性能进行了的测试,其中需要函数处理翻译实现的代码较少,因此查询的次数较少,可优化的空间也小.spec 测试集针对的是一些典型算法实现效率,调用函数的次数较多,库函数处理查询的次数较多,优化的空间更大.

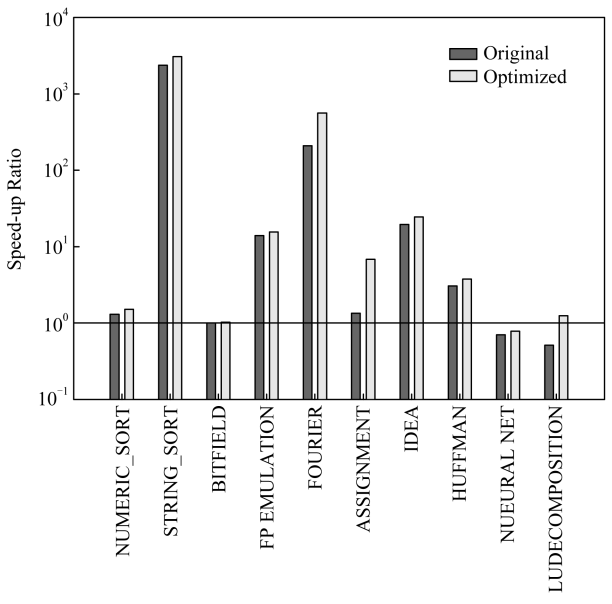


Fig. 6 Speed-up ratio contrast of nbench

图 6 nbench 加速比对照

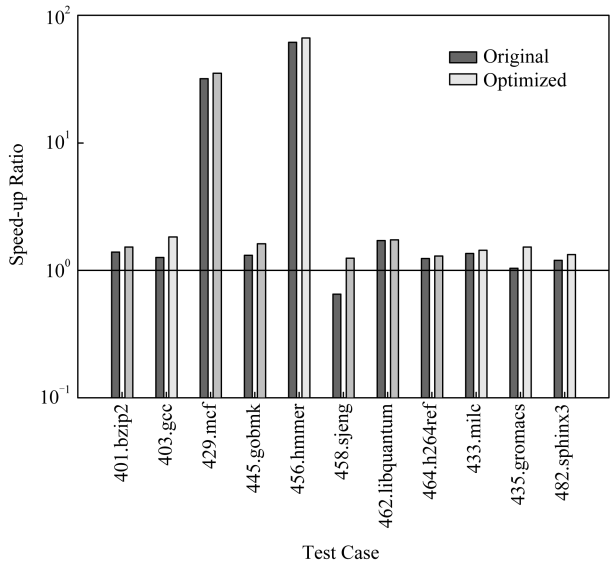


Fig. 7 Speed-up ratio contrast of spec2006

图 7 spec2006 加速比对照

4.3 结果分析

通过 nbench 和 spec2006 中 2 组加速比的对照,可以发现,经过查询优化的库函数处理翻译方式,相较于未优化的翻译方式,平均加速比有明显的提升.对于部分有负加速的测试用例,其提升效果更佳明显.例如 nbench 中的 nueural_net 和 lu_decomposition 用例以及 spec2006 中的 sjeng 用例.

查询优化的效果对于负加速的例子优化较为明显.这是对本文优化方法有效性的有力佐证.因为在未优化的库函数处理翻译模式下,因为查询的开销过大,当查询失败时,库函数处理过程完全成为翻译过程的额外开销,此时相较于一般的翻译处理方式,库函数处理过程会显现出负加速.因此,降低库函数处理过程中查询的开销,使得即使查询失败额外开销也不明显,这样可以使得有限的库函数处理获得的收益尽可能地不被查询开销消耗.

而其他例子的加速效果和用例的 2 个特征有关:1)库函数调用次数以可以处理的库函数比例;2)用例程序整体的指令规模.

测试结果中,npb 中 string_sort 和 fourier 用例以及 spec2006 中的 mcf 和 hmmer 用例具有较好的加速比.相较于其他用例,这几个用例在翻译执行的过程中,可处理库函数指令的规模在程序整体的指令规模中占比较高.spec006 中翻译可处理库函数在源程序执行过程中指令数占源程序执行的指令总数比例如图 8 所示:

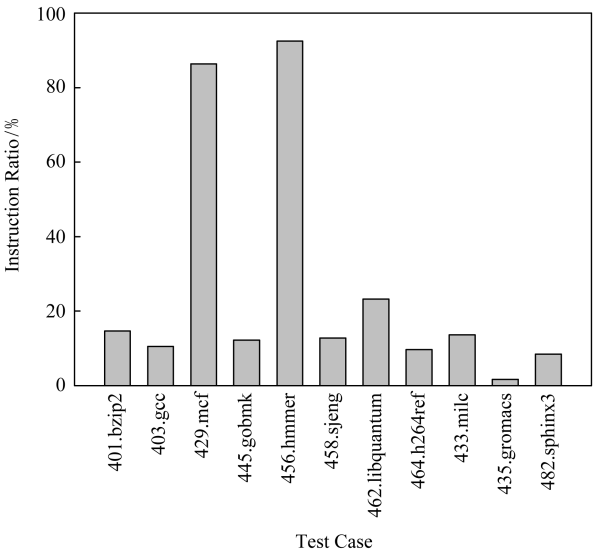


Fig. 8 Instruction ratio of disposing library functions to whole

图 8 可处理库函数指令占比

由此可见,库函数处理的优化加速类似于并行加速,执行过程中可处理库函数的翻译执行过程属于可优化的部分,因此这部分程序的比例决定了库函数处理优化加速的效果.而查询过程的优化即降低了库函数处理决策的开销,从而保证了此翻译方式的优化效果.

5 相关研究

动态二进制翻译中的库函数处理在函数解析的层面上和软件逆向技术的需求是一致的;在语义解析的基础上完成本地代码生成,因此动态二进制翻译可以归类于即时编译技术的一类具体应用.

在语义解析时,由于库函数处理进行了函数级的语义解析,因此可以实现更有效的即使编译以达到更高效的翻译效率.

在同架构跨操作系统的移植系统中,比较知名的应用有 wine.在跨架构的 aa 软件移植以及虚拟化系统中,QEMU 成为研究的主流平台[18-19].

然而,在相关研究中,源程序语义的解析,语义元素在本地实现的优化始终是研究的重点.库函数处理的翻译方式在解析和本地实现 2 个层面都有所优化,因此具有较好的应用前景[20-21].

6 总 结

本文针对动态二进制翻译中库函数处理翻译模式中的查询开销,给出了一种优化的查询方法.该方法充分解析了库函数处理过程内在动静结合的性质,通过将查询信息静态预处理,使用散列函数实现查询过程等手段,实现了源程序中库函数地址到相对应处理函数的快速映射,降低了查询开销.通过基于动态二进制翻译器 QEMU 的实验验证,该方法针对库函数查询开销的优化是有效的.

参 考 文 献

[1] Tan Yusong, Wu Qingbo. A study of virtualization and operating system technologies [J]. Computer Engineering & Science, 2011, 33(4): 62-68 (in Chinese)
(谭郁松, 吴庆波. 虚拟化与操作系统辨析[J]. 计算机工程与科学, 2011, 33(4): 62-68)

[2] Chen Jiuye, Yang Wu, Shen Boye, et al. Automatic validation for binary translation [J]. Computer Languages Systems & Structures, 2015, 43(C): 96-115

[3] Tang Feng, Wu Chenggang, Zhang Zhaoqing, et al. Exception handling in application level binary translation [J]. Journal of Computer Research and Development, 2006, 43(12): 2166-2173 (in Chinese)
(唐锋, 武成岗, 张兆庆, 等. 二进制翻译应用级异常处理[J]. 计算机研究与发展, 2006, 43(12): 2166-2173)

[4] William Y J, Bauman M A, Kao Fengjung, et al. Operand and limits optimization for binary translation system: America, US9021454 [P]. 2015-04-28

[5] Liang Yi, Shao Yuanhua, Yang Guowu, et al. Register allocation for QEMU dynamic binary translation systems [J]. International Journal of Hybrid Information Technology, 2015, 8(2): 199-210

[6] Hawkins B, Demsky B, Bruening D, et al. Optimizing binary translation of dynamically generated code [C] //Proc of the 13th Int Symp on Code Generation and Optimization. Piscataway, NJ: IEEE, 2015: 68-78

[7] 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)

[8] 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)

[9] Xie Haibin, Zhang Zhaoqing, Wu Chenggang, et al. Classified method of disposing library function in binary translation [J]. Application Research of Computers, 2008, 25(4): 1057-1059 (in Chinese)
(谢海斌,张兆庆,武成岗,等. 二进制翻译中系统库函数的分类处理方法[J]. 计算机应用研究, 2008, 25(4): 1057-1059)

[10] Fabrice B. QEMU, a fast and portable dynamic translator [C] //Proc of USENIX Annual Technical Conf. Berkeley, CA: USENIX Association, 2005: 41-46

[11] 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)

[12] TIS Committee. Tool interface standard executable and linking format specification [OL]. [2018-02-20]. http://refspecs.linuxbase.org/elf/elf.pdf

[13] Tan Jie, Pang Jianmin, Shan Zheng, et al. Redundant instruction optimization algorithm in binary translation [J]. Journal of Computer Research and Development, 2017 (in Chinese)
(谭捷,庞建民,单征,等. 二进制翻译中冗余指令优化算法[J]. 计算机研究与发展, 2017, 54(9): 1931-1944)

[14] Dai Tao, Shan Zheng, Lu Shuaibin, et al. Register allocation algorithm of dynamic binary translation based on priority [J]. Journal of Zhejiang University, 2016, 50(7): 1338-1346 (in Chinese)
(戴涛,单征,卢帅兵,等. 基于优先级动态二进制翻译寄存器分配算法[J]. 浙江大学学报, 2016, 50(7): 1338-1346)

[15] National Research Center of Parallel Computer Engineering Technology. Compile system user manual of Sunway TaihuLight [OL]. [2018-02-20]. http://www.nsccwx.cn/ceshi.php?id=12 (in Chinese)
(国家并行计算机工程技术研究中心. 神威太湖之光编译系统用户手册[OL]. [2018-02-20]. http://www.nsccwx.cn/ceshi.php?id=12)

[16] Mayer U F. nbench is a byte CPU benchmark [OL]. [2018-02-20]. http://linux.softpedia.com/get/System/Benchmarks/nbench-1374.shtml

[17] Standard Performance Evaluation Corporation. SPEC CPU© 2006 [OL]. [2018-02-20]. http://www.spec.org/cpu2006/

[18] WINEHQ. WINE [OL]. [2018-02-20]. https://www.winehq.org/

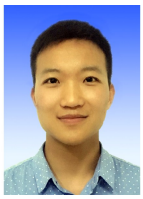
[19] Daniel B. QEMU: A Multihost Multitarget Emulator [M]. Houston, TX: Belltown Media, 2006

[20] Qi Ning, Fu Wen, Zhao Rongcai. Study of library functions recognizing technology in binary translation [J]. Journal of Computer Applications, 2006, 26(4): 233-235, 238 (in Chinese)
(齐宁,付文,赵荣彩. 二进制翻译中的库函数识别技术研究[J]. 计算机应用, 2006, 26(4): 233-235, 238)

[21] Liang Yingchao, Shang Yunhai, Li Chunqiang. Packing automation of library function in dynamic binary translation [J]. Computer Applications and Software, 2015, 32(6): 14-16 (in Chinese)
(梁英超,尚云海,李春强. 动态二进制翻译的库函数包装自动化[J]. 计算机应用与软件, 2015, 32(6): 14-16)

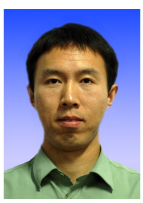
Fu Liguo, born in 1989. PhD. His main research interests include advanced compilation and high performance calculation.

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 calculation.



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

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



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