doi:10.11887/j.cn.202104019 http://journal.nudt.edu.cn

基于中间表示规则替换的二进制翻译中间代码优化方法*

李 男[1,2],庞建民[1,2]
(1. 战略支援部队信息工程大学,河南 郑州 450001;
2. 数学工程与先进计算国家重点实验室,河南 郑州 450002)

摘 要:动态二进制翻译在实现多源到多目标的程序翻译过程中,为屏蔽不同源平台间的硬件差异引入中间代码,采用内存虚拟策略进行实现,但同时带来中间代码膨胀问题。传统的中间代码优化方法主要采用对冗余指令进行匹配删除的方法。将优化重点聚焦在针对特殊指令匹配的中间表示规则替换上,提出了一种基于中间表示规则替换的二进制翻译中间代码优化方法。该方法针对中间代码膨胀所呈现的几种典型情景,描述了中间表示替换规则,并将以往应用在后端代码优化上的寄存器直接映射策略应用在此处。通过建立映射公式,实现了将原来的内存虚拟操作替换为本地寄存器操作,从而降低了中间代码膨胀率。使用SPEC CPU2006测试集进行了实验,验证了此优化方法的正确性和有效性。测试用例在优化前和优化后的执行结果一致,验证了优化方法的正确性;优化后测试用例的中间代码平均缩减率达到32.59%,验证了优化方法的有效性。

关键词:动态二进制翻译;中间代码;内存虚拟策略;代码膨胀;中间表示规则;寄存器映射
中图分类号:TP314 **文献标志码**:A **文章编号**:1001-2486(2021)04-156-07

Intermediate code optimization method for binary translation based on intermediate representation rule replacement

LI Nan[1,2], *PANG Jianmin*[1,2]
(1. PLA Strategic Support Force Information Engineering University, Zhengzhou 450001, China;
2. State Key Laboratory of Mathematical Engineering and Advanced Computing, Zhengzhou 450002, China)

Abstract: In the process of realizing multi-source to multi-target program translation, the dynamic binary translation uses intermediate code to shield the hardware differences between different source platforms, and the memory virtual strategy is adopted to achieve the goal. As a result, it brings about the problem of intermediate code expansion. Traditional intermediate code optimization methods usually use the method of matching and deleting redundant instructions. The intermediate representation rule replacement for special instruction matching was focused on, and an intermediate code optimization method for binary translation based on intermediate representation rule replacement was proposed. This method described the corresponding intermediate representation replacement rules for several typical scenarios of intermediate code expansion, and the register direct mapping strategy used in the back-end code optimization was applied here. By establishing mapping formula, the memory virtual operation was replaced by local register operation, thus reducing the expansion degree of intermediate code. The SPEC CPU2006 test suite was used to carry out the experimental, and the experimental results verify the correctness and effectiveness of this optimization method. The results before and after optimization are consistent, which verifies the correctness of the optimization method, meanwhile, the average reduction rate of intermediate code for each case after optimization is 32.59%, which verifies the effectiveness of the optimization method.

Keywords: dynamic binary translation; intermediate code; memory virtual strategy; code expansion; intermediate representation rule; register mapping

二进制翻译技术[1-2]是作为程序等价变换工具产生并发展起来的,被定义为一种机器上的指令序列到另一种机器上指令序列的等价转换过程。它在丰富国产平台软件生态的过程中,发挥了重要作用。

二进制翻译按照实现方式主要包含静态翻译和动态翻译[3]。静态翻译采用的是一种先翻译后执行的“离线翻译”方式,实现了“一次翻译,多次执行”,其翻译过程与执行过程彼此独立,翻译时间不计入系统时间。动态翻译采用的是一种边

* **收稿日期**:2020-11-09
基金项目:国家自然科学基金资助项目(61802433)
作者简介:李男(1977—),男,辽宁锦州人,副教授,博士,硕士生导师,E-mail:linan_happy@126.com

翻译、边执行的“捆绑式”工作方式,可以实现多源到多目标的程序翻译,由于可以获得运行时信息,因此动态翻译能够解决静态翻译难以解决的动态地址发现和自修改代码的问题。但是,动态翻译本身需要占用部分系统时间。翻译优化问题一直是动态翻译的研究热点[4-6]。本文在不引起歧义的情况下,将待翻译二进制文件的生成平台称为源平台,将动态二进制翻译器运行的平台称本地平台或目标平台。

1　问题的提出

动态二进制翻译在实现多源到多目标的程序翻译过程中,为了屏蔽不同源平台间的硬件差异,引入了中间代码,可以将任何源平台的指令先转化成中间代码,再转化成目标平台指令。在中间代码生成过程中,动态二进制翻译采用了内存虚拟机制,借助临时变量和环境变量,将对源平台特定寄存器的访问操作转化为与寄存器无关的内存访问操作。临时变量用来暂存变量,一般使用 tempX 的形式来表示,例如 temp0,temp1 等;环境变量是一种全局变量,用来模拟源平台的 CPU 环境,负责将临时变量存储到本地内存,一般使用 env 的形式来表示。env 有多种实现形式,例如 X86 平台对应的 env 数据结构如图 1 中的 CPUX86State 所示,其成员变量 *regs*[*CPU_NB_REGS*]用来模拟 X86 平台的寄存器,此时常量 *CPU_NB_REGS* = 9。

```
typedef struct CPUX86State
{
    target_ulong regs[CPU_NB_REGS]
        …
} CPUX86State
```

图 1　X86 平台下的 env 结构

Fig. 1　env structure on X86 platform

利用内存虚拟机制,二进制翻译器在处理与源平台寄存器有关的指令时,借助 env 将其转化成对本地内存空间的操作。图 2 展示了 X86 平台的立即数加法被翻译到中间代码的过程。源平台第 1 条 movl 汇编指令是立即数传送指令,使用了通用寄存器 eax,经过内存虚拟机制,被翻译为 2 条中间代码:第 1 条中间代码是 movi_i32 temp0,$0x1,表示将立即数 0x1 暂存于临时变量 temp0 中;第 2 条中间代码是 st_i32 temp0 env,$0x0,表示将 temp0 存储于由环境变量 env 和偏移量 0 指定的本地内存位置。由环境变量 env 与偏移量构成的虚拟内存位置等价表示了源平台的位置,实现了去源平台化,这就是内存虚拟的含义。

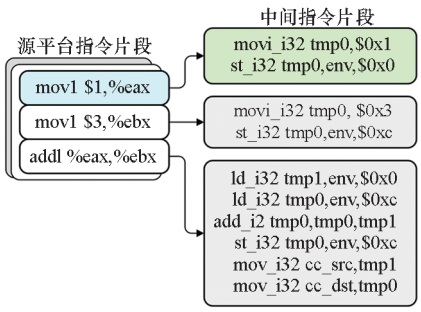

图 2　立即数加法的翻译过程

Fig. 2　Translation process of immediate addition instruction

内存虚拟策略在屏蔽源平台硬件差异的同时,会带来中间代码的膨胀问题。在图 2 的示例中,源平台的前 2 条指令被分别翻译为 2 条中间代码,第 3 条指令被翻译为 6 条中间代码。总的代码膨胀达到 3.3 倍。造成中间代码膨胀的原因在于中间代码生成机制依赖于频繁的内存模拟操作。

在针对中间代码优化的相关研究中,主要采用的方法有两类:第一类与后端冗余指令相关,文献[7]提出了线性扫描冗余 ldM 和 stM 指令的匹配删除算法来删除冗余指令;文献[8]引用活性分析技术对快速模拟器(Quick EMUlator, QEMU)[9]后端无内部互锁流水级的微处理器(Microprocessor without Interlocked Piped Stages, MIPS)冗余 MOVE 指令提出了删除算法。另一类方法与寄存器分配策略相关,文献[10]详细分析了 QEMU 下的寄存器管理机制,但并未提出自己的优化方案;文献[11]实现了 X86 平台到龙芯平台的寄存器映射,映射的寄存器范围限于源平台的 eax、esp、ebp 三个寄存器和龙芯平台的 S4、S5、S6 三个寄存器;文献[12]提出了一种在寄存器映射过程中进行裁剪的方法,借助裁剪函数实现了 PowerPC 平台到 ALPHA 平台的寄存器映射;文献[13]实现了 X86 平台下的 8 个通用寄存器向 MIPS 平台的映射;文献[14]提出一种基于优先级的动静结合寄存器映射优化算法,首先根据源平台寄存器的统计特征进行静态全局寄存器映射,然后通过获取基本块中间指令需求确定寄存器分配的优先级,进行动态寄存器分配。

基于上述研究,本文将特殊指令匹配与寄存器直接映射思想应用在二进制翻译中间代码的优化过程中,对造成中间代码膨胀的典型场景进行分析归纳,找出特定指令动作的组合,通过建立相

应的映射规则,将内存虚拟表示转化为对本地寄存器的直接操作,从而提高翻译效率。

2 优化方法

2.1 中间表示函数

动态二进制翻译使用内存虚拟策略时,中间代码的生成是通过调用中间表示函数(序列)实现的,一次内存模拟操作需要调用一个中间表示函数序列,由多个中间表示函数的组合进行实现。例如,对于x86平台下的数据传送指令MOV *Im*, *reg*,内存虚拟机制会调用两条中间表示函数 *gen_op_movl_T0_im*( )和 *gen_op_movl_reg_T0*( )生成中间代码。表1给出了这两条中间表示函数的功能描述和定义。

表1 中间表示函数示例

Tab.1 Example for intermediate representation function

函数定义	功能描述

static inline void *gen_op_movl_T0_im*(*int32_t val*)
{
tcg_gen_movi_tl(*cpu_T*[0], *val*);
}
	将立即数暂存于临时变量 *CPU_T*[0](tmp0)中

static inline void *gen_op_mov_reg_T0*(*int ot*, *int reg*, *TCGv t0*)
{
switch(ot) {
…
case OT_LONG:
tcg_gen_st32_tl(*t0*, *cpu_env*, *offsetof*(*CPUState*, *regs*[*reg*]) + *REG_L_OFFSET*);
break;
…
}
	将立即数存入对应于源寄存器的本地内存区域

中间表示函数 *gen_op_movl_T0_im*( )用来实现将立即数暂存于临时变量 *CPU_T*[0]中,中间表示函数 *gen_op_movl_reg_T0*( )用来实现将立即数存入对应于源寄存器的本地内存区域。类似于这样的中间表示函数还有很多,它们的函数名称都以gen_op_作为前缀,以特定的指令动作名称作为后缀。基于中间表示函数,可以进一步对相关机器指令操作,特别是造成中间代码膨胀的寄存器操作进行描述。

2.2 基于中间表示函数的特定指令动作描述

在图2的示例中,源平台的代码翻译到中间代码后,反复出现了立即数提取指令movi_i32,加载指令ld_i32,以及存储指令st_i32等。通过大量的代码分析得出,中间代码膨胀主要与四种寄存器操作相关,包括立即数提取操作、加载操作、存储操作和临时变量操作,本文将上述四种造成中间代码膨胀的特殊指令操作称为特定指令动作。

使用中间表示函数可以对这四种特定指令动作进行描述,并进一步抽象,方便后续的优化工作。例如,立即数提取操作可以使用中间表示函数类 *gen_op_movl_A_im* 表示,并可进一步简化表示为 $op\_mov\_im(A, Im)$。

四种特定指令动作的描述如下:

1)立即数提取操作:$op\_mov\_im(A, Im) \in \{gen\_op\_movl\_A\_im\}$。

2)加载操作:$op\_ld(A, \alpha) \in \{gen\_op\_movl\_A\_reg(\alpha)\}$。

3)存储操作:$op\_st(\alpha, A) \in \{gen\_op\_movl\_reg(\alpha)\_A\}$。

4)临时变量操作:$op(A)$, $op(A, B)$。第一个参数 A 用于存储运算后的结果。

其中,$A, B \in \{T0, T1, A0\}$。

以此为基础,分析造成中间代码膨胀的典型场景,找到与其相关的特定指令动作组合,然后运用寄存器直接映射思想,使用少量的本地寄存器操作进行等价替换,从而降低中间代码膨胀。

3 优化方法的实现

在实现将内存虚拟操作映射到本地寄存器的过程中,需要解决好两个问题:一是要保证寄存器在映射过程中的一致性。二是要建立等价的中间表示替换规则。

针对第一个问题,首先需要构建合适的映射公式,同时,还要保证源平台每一个待映射的寄存器都能够映射到本地平台的某个寄存器上。当本地平台的寄存器数量远大于或等于源平台的寄存器数量时,可以选择本地平台的部分寄存器进行映射。当本地平台的寄存器数量小于源平台的寄存器数量时,可以采用活性分析技术加以解决。3.1节给出了实现过程。

针对第二个问题,需要找到与中间代码膨胀相关的特定指令动作的组合,通过构建相应的映射规则,实现将原有的内存虚拟操作转化为对本地平台的寄存器操作。3.2节给出了实现过程。

3.1 映射公式的构建

本文研究的源平台为X86平台,目标平台是国产申威平台,源平台包含9个通用寄存器,目标平台包含32个通用寄存器。目标平台寄存器的数量远多于源平台寄存器的数量,满足映射的前

提条件,因此,适用于本文的优化方法。

原有的内存虚拟策略是借助环境变量 *env* 通过调用使用 $env \rightarrow regs[\alpha]$ 实现的,本文在实现内存虚拟操作到本地寄存器映射时,引入临时变量数组 $reg[i]$,建立了式(1)和表 2 所示的映射关系。

$$env \rightarrow regs[\alpha] \rightarrow reg[i] \tag{1}$$

表 2　X86 平台与国产申威平台的寄存器映射关系

Tab. 2　Register mapping between X86 platform and domestic SW platform

X86 平台寄存器	映射变量	申威平台寄存器
eax	$reg[0]$	r7
ecx	$reg[1]$	r8
edx	$reg[2]$	r9
ebx	$reg[3]$	r10
esp	$reg[4]$	r11
ebp	$reg[5]$	r12
esi	$reg[6]$	r13
edi	$reg[7]$	r14
eip	$reg[8]$	r15

在式(1)中,临时变量数组 $reg[i]$ 的自变量取值范围由源平台寄存器的数量决定,例如,源平台 X86 平台包含 9 个通用寄存器,则 $reg[i]$ 中 i 的取值为 $0 \leqslant i \leqslant 8$。而目标申威平台包含 32 个通用寄存器,仅需要选用其中的部分寄存器进行本地映射即可。本文选用申威平台下的 r7 到 r15 作为本地寄存器进行映射,表 2 显示了具体的映射关系。

3.2　中间表示替换规则的建立

2.2 节中已经提到中间代码的膨胀往往和一些特定指令动作的组合相关,针对中间代码膨胀呈现出的几种典型场景,应用式(1),可以建立以下 4 条基本替换规则:

替换规则 C1:如果存在相邻的特定指令动作 $op\_mov\_im(A, Im)$ 和 $op\_st(reg(\alpha), A)$ 其中 $A \in \{T0, T1, A0\}$,则可以使用 $op\_mov\_im(reg[\alpha], Im))$ 替代 $op\_mov\_im(A, Im)$ 和 $op\_st(reg(\alpha), A)$。

C1 的典型应用场景:将立即数移动到虚拟内存。

分析:将立即数 *Im* 提取到中间变量 *A*,然后将中间变量 *A* 存储到虚拟内存 $reg(\alpha)$ 的操作序列,这与将立即数 *Im* 存储到寄存器 $reg(\alpha)$ 的操作功能等价。

替换规则 C2:如果存在相邻的特定指令动作 $op(A)$、$op\_ld(B, reg(\alpha))$、$op(B, A)$ 和 $op\_st(reg(\alpha), B)$,其中 A、$B \in \{T0, T1, A0\}$,则可以使用 $op(reg[\alpha], A)$ 替代 $op\_ld(B, reg(\alpha))$、$op(B, A)$ 和 $op\_st(reg(\alpha), B)$。

C2 的典型应用场景:虚拟内存与立即数的计算结果存入同一虚拟内存。

分析:将虚拟内存 $reg(\alpha)$ 提取到中间变量 *B*,然后将中间变量 *B* 和 *A* 的计算结果临时存储到 *B* 中,再将 *B* 存储到虚拟内存 $reg(\alpha)$ 的操作序列,这与将虚拟内存 $reg(\alpha)$ 和中间变量 *A* 的计算结果存储到虚拟内存 $reg(\alpha)$ 的操作功能等价。

替换规则 C3:如果存在相邻的特定指令动作 $op\_ld(A, reg(\alpha))$、$op(A)$ 和 $op\_st(reg(\alpha), A)$,其中 $A \in \{T0, T1, A0\}$,则可以使用 $op(reg[\alpha])$ 替代 $op\_ld(A, reg(\alpha))$、$op(A)$ 和 $op\_st(reg(\alpha), A)$。

C3 的典型应用场景:仅操作同一虚拟内存数据。

分析:将虚拟内存 $reg(\alpha)$ 提取到中间变量 *A*,然后对 *A* 进行相关计算并保存计算结果到 *A*,再将 *A* 写回虚拟内存 $reg(\alpha)$ 的操作序列,这与将虚拟内存 $reg(\alpha)$ 计算后存储到自身的操作功能等价。

替换规则 C4:如果存在相邻的特定指令动作 $op(A)$、$op\_ld(A, reg(\alpha))$ 和 $op\_st(reg(\beta), A)$,其中 $A \in \{T0, T1, A0\}$,则可以使用 $mov\_reg\_reg(reg[\beta], reg[\alpha])$ 替代 $op(A)$、$op\_ld(A, reg(\alpha))$ 和 $op\_st(reg(\beta), A)$。

C4 的典型应用场景:仅涉及不同虚拟内存间的数据移动。

分析:将虚拟内存 $reg(\alpha)$ 提取到中间变量 *A*,然后将 *A* 的值存储到虚拟内存 $reg(\beta)$ 的操作,这与将虚拟内存 $reg(\alpha)$ 存储到虚拟内存 $reg(\beta)$ 的操作功能等价。

在上述 4 条基本替换规则基础上,还可以衍生出以下 2 条扩展的替换规则:

替换规则 C5:如果存在相邻的特定指令动作 $op\_ld(A, reg(\alpha))$、$op(B, A)$ 和 $op\_st(reg(\beta), B)$,则可以使用 $op(reg[\beta], reg[\alpha])$ 替代 $op\_ld(A, reg(\alpha))$、$op(B, A)$ 和 $op\_st(reg(\beta), B)$。

替换规则 C6:如果存在相邻的特定指令动作 $op\_ld(A, reg(\alpha))$,$op\_ld(B, reg(\beta))$,$op(B, A)$ 和 $op\_st(reg(\beta), B)$,其中 $A, B \in \{T0, T1, A0\}$,则可以使用 $op(reg[\beta], reg[\alpha])$ 替代 $op\_ld(A, reg(\alpha))$、$op\_ld(B, reg(\beta))$、$op(B, A)$ 和 $op\_st(reg(\beta), B)$。

上述规则提到的相邻特定指令动作并不要求位置连续,但必须存在于同一基本块,这是由语义环境一致性要求决定的。通过以上 6 条替换规则,可以实现功能等价条件下的寄存器直接映射策略,将原有内存模拟操作替换为本地寄存器操作。

4 优化方法的应用

下面分别以翻译源平台 X86 下的立即数加法指令和栈操作指令为例，阐述中间表示替换规则的运用。

图 3 展示了应用替换规则后，X86 平台的立即数加法指令被翻译到中间代码的优化过程。源平台中第一条 movl 指令应用替换规则 C1 后，将立即数 0x1 直接存入 eax 寄存器对应的本地寄存器 *reg*[0]中；类似地，源平台第二条 movl 指令应用替换规则 C1 后，将立即数 0x3 存入 ebx 寄存器对应的本地寄存器 *reg*[3]中；源平台第三条 addl 指令应用替换规则 C6 后，使用本地寄存器 *reg*[0]和 *reg*[3]完成相应操作。最终，中间代码由优化前的 10 条指令简化为优化后的 5 条，中间代码规模缩减为原来的 50%。

同样，图 4 展示了应用替换规则后，X86 平台的栈操作被翻译到中间代码的优化过程。应用替换规则后，中间代码数量由优化前的 29 条指令简化为优化后的 16 条，规模减少了 44.8%。

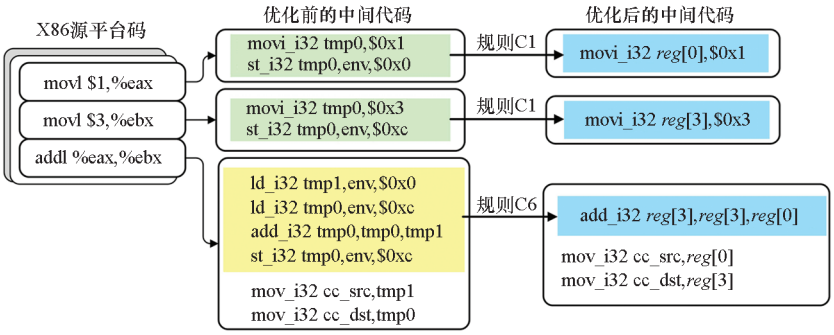

图 3 立即数加法指令的中间代码优化示例

Fig. 3 Example of intermediate code optimization for immediate addition instruction

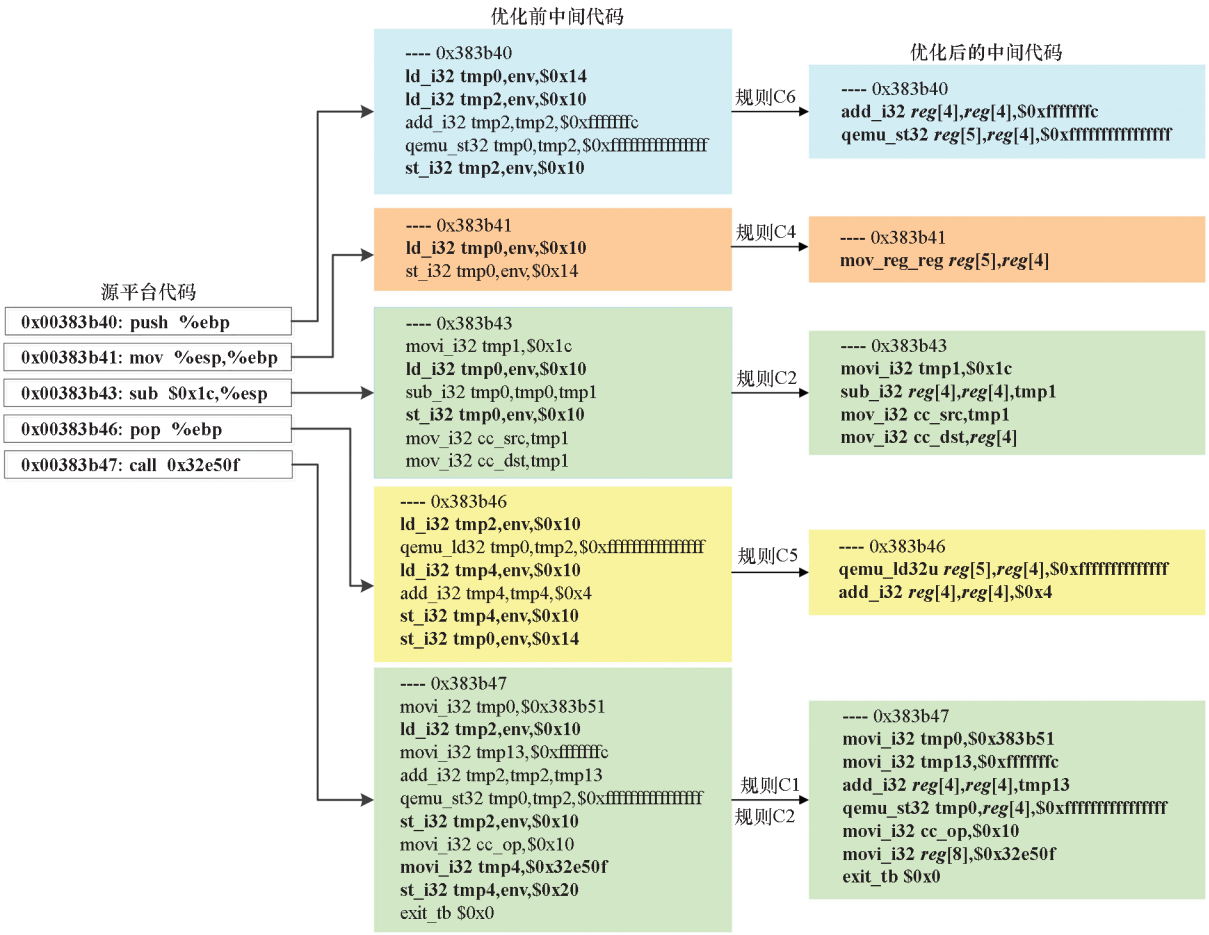

图 4 栈操作指令的中间代码优化示例

Fig. 4 Example of intermediate code optimization for stack operation instruction

5　实验部分

将提出的中间代码优化方法在开源二进制翻译器 QEMU1.7.2[15] 进行了实现。使用如表 3 所示的实验环境,源平台采用的是 X86 – 64 架构处理器,本地平台采用的是国产申威 411[16] 处理器。

表 3　实验环境

Tab.3　Test environment

参数名称	源平台	本地平台
处理器	Intel(R) Core(TM)2 Quad CPU Q9500 @ 2.83 GHz	SW411 processor
操作系统	Fedora 2.6.27.5 – 117.fc10.i686	NeoKylin 3.8.0
编译器	gcc – 4.3.2	gcc – 4.5.3
主频	2.5 GHz	1.6 GHz
内存	4 G	4 G
外存	500 G	500 G

采用了正确性测试和性能测试两种方法来验证优化方法的正确性和有效性。正确性测试通过对比优化前和优化后的执行结果是否一致来进行验证,性能测试通过计算优化后的中间代码指令数量相对于优化前的中间代码指令数量的压缩率来进行验证,压缩率越高优化效果越好。测试用例选取了 SPEC CPU2006[17] 中的 CINT2006 的测试用例,如表 4 所示。

5.1　正确性测试

实验使用 SPEC CPU2006 进行了正确性测试,采用的测试方法是比较优化前后测试用例的执行结果是否一致。表 5 显示了选取的测试用例和测试结果,测试结果表明优化方法的正确性达到 100%。

5.2　性能测试

实验借助 QEMU 翻译 SPEC CPU2006 测试用例过程中产生的中间代码日志文件,通过对比优化前后日志文件中的中间指令数量,测试中间代码优化的效果。为了更清楚地说明测试效果,引入了代码缩减率 R,如式(2)中所示:

$$R = 1 - N_{opt}/N_{ori} \times 100\% \qquad (2)$$

式(2)中,N_{ori}代表优化前的中间指令数量,N_{opt}代表优化后的中间指令数量,R 值越高表明中间代码优化效果越好。

实验记录了每个测试用例的 R 值,测试结果见图5。结果表明,中间表示规则的建立对于中间代码的优化效果作用明显,各用例的代码缩减率 R 平均达到 32.59%。可以看出,不同用例的加速效果差别较大,优化效果最好的用例是

表 4　测试用例说明

Tab.4　Test case specification

测试用例	说　明
Perlbench	PERL 编程语言,负载由三个 script 组成
bzip2	压缩,负载包括 6 个部分:2 个图片,1 个程序,1 个 tar 包,1 个 HTML 文件,1 个混合文件,分别使用了 3 个不同的压缩等级进行压缩和解压缩
gcc	C 编译器,对 9 组 C 代码进行了编译
mcf	组合优化,用于大型公共交通中的单站车辆调度的程序
gobmk	人工智能:围棋
hmmer	基因序列搜索,使用隐马尔可夫模型(Hidden Markov Models, HMMS)基因识别方法进行基因序列搜索
sjeng	人工智能:国际象棋
libquantum	libquantum 是模拟量子计算机的库文件
h264ref	视频压缩使用两种配置对两个 YUV 格式源文件进行 H.264 编码
omnetpp	离散事件仿真,包括约 8 000 台计算机和 900 个交换机/集线器,以及混合了各种速率的大型 CSMA/CD 协议以太网络模拟
astar	寻路算法,实现了 2D 寻路算法 A* 的三种不同版本
xalancbmk	XML 处理,XML 文档/XSL 表到 HTML 文档的转换

表 5　中间代码优化正确性测试

Tab.5　Correctness test of intermediate code optimization

用例名称	执行结果是否一致
perbench	是
bzip2	是
gcc	是
mfc	是
gobmk	是
hmmer	是
sjeng	是
libquantum	是
h264ref	是
omnetpp	是
astar	是
xalancbmk	是

xalancbmk,其 R 值达到了 37.31%;优化效果最差的用例是 gcc,其 R 值为 30.68%。通过分析用例源代码发现,xalancbmk 用例中包含了大量的寄存器操作,使得中间表示替换规则可以被充分使用,所以优化效果明显;gcc 用例只包含少量的寄存器移位操作,替换规则应用较少,所以优化作用有限。

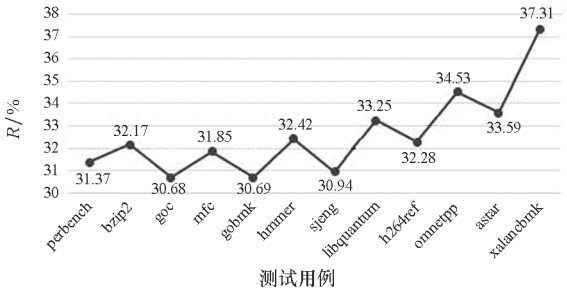

图5 中间代码优化性能测试

Fig. 5 Performance test of intermediate code optimization

6 结论

针对动态二进制翻译过程中的中间代码膨胀问题,本文对内存虚拟策略的实现机制进行了深入分析,提出了一种基于中间表示规则替换的二进制翻译中间代码优化方法,在对中间表示函数进行分类归纳的基础上,建立了针对特定指令函数匹配的中间表示替换规则,对于能够匹配规则的特定指令动作,应用建立的映射公式,使用少量的本地寄存器操作替代原有的内存虚拟操作,进而达到优化中间代码的目的。

另外,本文研究的基础平台是 QEMU,具备多源到多目标的二进制翻译功能,因此,本文提出的方法具有一定的泛化能力。同时,需要进一步完善等价替换规则,从而取得更好的翻译效果。

参考文献(References)

[1] ALTMAN E R, KAELI D, SHEFFER Y. Welcome to the opportunities of binary translation[J]. Computer, 2000, 33(3): 40 – 45.

[2] GSCHWIND M, ALTMAN E R. Dynamic and transparent binary translation[J]. Computer, 2000, 33(3): 54 – 59.

[3] CIFUENTES C, MALHOTRA V. Binary translation: static, dynamic, retargetable? [C] // Proceedings of International Conference on Software Maintenance, 1996: 340 – 349.

[4] 傅立国, 庞建民, 王军, 等. 动态二进制翻译中库函数处理的优化[J]. 计算机研究与发展, 2019, 56(8): 1783 – 1791.
FU Liguo, PANG Jianmin, WANG Jun, et al. Optimization of library function disposing in dynamic binary translation[J]. Journal of Computer Research and Development, 2019, 56(8): 1783 – 1791. (in Chinese)

[5] 王荣华. 动态二进制翻译优化研究[D]. 杭州: 浙江大学, 2013.
WANG Ronghua. Research on dynamic binary translation optimization[D]. Hangzhou: Zhejiang University, 2013. (in Chinese)

[6] HAWKINS B, DEMSKY B, BRUENING D, et al. Optimizing binary translation of dynamically generated code[C]// Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization. IEEE, 2015: 68 – 78.

[7] 戴涛, 单征, 岳峰, 等. 一种动态二进制翻译中间表示变量活性分析改进算法[J]. 小型微型计算机系统, 2016, 37(2): 395 – 400.
DAI Tao, SHAN Zheng, YUE Feng, et al. An improved algorithm for activity analysis of intermediate representation variables in dynamic binary translation[J]. Journal of Chinese Computer Systems, 2016, 37(2): 395 – 400. (in Chinese)

[8] 宋强, 陈香兰, 陈华平. 动态二进制翻译器 QEMU 中冗余指令消除技术研究[J]. 计算机应用与软件, 2012, 29(5): 67 – 69.
SONG Qiang, CHEN Xianglan, CHEN Huaping. Optimization technique of redundant instructions elimination in dynamic binary translator QEMU [J]. Computer Applications and Software, 2012, 29(5): 67 – 69. (in Chinese)

[9] BARTHOLOMEW D. QEMU: a multihost, multitarget emulator[J]. Linux Journal, 2006, 2006(145): 3.

[10] LIANG A, GUAN H, LI Z X. A research on register mapping strategies of QEMU[C]// Proceedings of 2th International Symposium on Intelligence Computation and Applications, 2007.

[11] 蔡嵩松, 刘奇, 王剑, 等. 基于龙芯处理器的二进制翻译器优化[J]. 计算机工程, 2009, 35(7): 280 – 282.
CAI Songsong, LIU Qi, WANG Jian, et al. Optimization of binary translator based on GODSON CPU[J]. Computer Engineering, 2009, 35(7): 280 – 282. (in Chinese)

[12] 文延华, 唐大国, 漆锋滨. 二进制翻译中的寄存器映射与剪裁的实现[J]. 软件学报, 2009, 20(12): 1 – 7.
WEN Yanhua, TANG Daguo, QI Fengbin. Register mapping and register function cutting out implementation in binary translation[J]. Acta Software Sinica, 2009, 20 (12): 1 – 7. (in Chinese)

[13] 廖银, 孙广中, 姜海涛, 等. 动态二进制翻译中全寄存器直接映射方法[J]. 计算机应用与软件, 2011, 28(11): 21 – 24.
LIAO Yin, SUN Guangzhong, JIANG Haitao, et al. All registers direct mapping method in dynamic binary translation[J]. Computer Applications and Software, 2011, 28(11): 21 – 24. (in Chinese)

[14] 王军, 庞建民, 傅立国, 等. 二进制翻译中动静结合的寄存器分配优化方法[J]. 计算机研究与发展, 2019, 56(4): 708 – 718.
WANG Jun, PANG Jianmin, FU Liguo, et al. A dynamic and static combined register mapping method in binary translation [J]. Journal of Computer Research and Development, 2019, 56(4): 708 – 718. (in Chinese)

[15] BELLARD F. QEMU, a fast and portable dynamic translator[C]// Proceedings of USENIX Annual Technical Conference, 2005: 41 – 46.

[16] SUNWAY 411 [EB/OL]. (2017 – 10 – 8)[2019 – 12 – 14]. http://www.swcpu.cn/show – 190 – 256 – 1.html.

[17] Standard Performance Evaluation Corporation. SPEC CPU 2006[EB/OL]. (2018 – 01 – 09) [2019 – 12 – 14]. http://www.spec.org/cpu2006/.