计算机研究与发展 DOI:10.7544/issn1000-1239.2017.20151110
Journal of Computer Research and Development 54(9): 1931-1944, 2017

二进制翻译中冗余指令优化算法

谭 捷 庞建民 单 征 岳 峰 卢帅兵 戴 涛
(数学工程与先进计算国家重点实验室 郑州 450002)
(jessie_tanjie@hotmail.com)

Redundant Instruction Optimization Algorithm in Binary Translation

Tan Jie, Pang Jianmin, Shan Zheng, Yue Feng, Lu Shuaibing, and Dai Tao
(*State Key Laboratory of Mathematical Engineering and Advanced Computing*, *Zhengzhou* 450002)

Abstract Binary translation is a main method to implement software migration. Dynamic binary translation is limited by dynamic execution and cannot be deeply optimized, resulting in low efficiency. Traditional static binary translation has difficulty to deal with indirect branch, and conventional optimization methods mostly affect in the intermediate code layer, paying less attention to a large number of redundant instructions that exist in the target code. According to this situation, this paper presents a static binary translation framework SQEMU and a target code optimization algorithm to delete redundant instructions based on the framework. The algorithm generates an instruction-specific data dependence graph (IDDG) based on the analysis of target codes, then combines liveness analysis with peephole optimization using IDDG, and effectively removes redundant instructions in target codes. Experimental results show that using the optimization algorithm for target codes, the execution efficiency is significantly increased, the maximal increase up to 42%, and the overall performance test shows that the optimized translation efficiency of nbench is increased by about 20% on average, and it is increased about 17% of SPEC CINT2006 on average.

Key words binary translation; redundant instruction; liveness analysis; peephole optimization; SQEMU frame

摘 要 二进制翻译是实现软件移植的主要方法.动态二进制翻译受动态执行限制而不能深度优化导致效率较低而传统的静态二进制翻译难以处理间接分支,且现有的优化方法大部分集中在中间代码层,对目标码中存在的大量冗余指令较少关注.针对这一现状,提出一种静态二进制翻译框架SQEMU,基于该框架提出了一种对目标码冗余指令进行删除的优化算法.该算法通过分析目标码生成指令特定数据依赖图(instruction-specific data dependence graph, IDDG),再利用该图将活性分析和窥孔优化的2种理论相结合,有效删除目标码中的冗余指令.实验结果表明,利用该算法对目标码优化后,其执行效率得到显著提升,最大提升可达42%,整体性能测试表明,优化后nbench测试集翻译效率提高约20%,SPEC CINT2006测试集翻译效率提高约17%.

收稿日期:2015-12-21;**修回日期**:2016-06-02

基金项目:国家自然科学基金项目(61472447);国家“八六三”高技术研究发展计划基金项目(2009AA012201);“核高基”国家科技重大专项基金项目(2009ZX01036-001-001)

This work was supported by the National Natural Science Foundation of China (61472447), the National High Technology Research and Development Program of China (863 Program) (2009AA012201), and the National Science and Technology Major Projects of Hegaoji (2009ZX01036-001-001).

关键词 二进制翻译;冗余指令;活性分析;窥孔优化;SQEMU 框架

中图法分类号 TP314

二进制翻译技术是实现软件移植的主要方法,现已广泛应用于遗产代码移植、系统安全防护等国内外诸多领域,是解决不同体系结构处理器之间软件移植的主要途径,如开源的跨平台动态二进制翻译器 QEMU[1],已被移植到多种处理器平台,其中包括国产龙芯平台[2].

二进制翻译可按翻译的时机不同分为动态二进制翻译和静态二进制翻译.动态二进制翻译是一种即时翻译技术,它是在目标程序运行时动态生成可执行代码,代码优化占用程序执行时间,其翻译过程受动态执行的限制而不能进行深度细致优化[3].另外,动态二进制翻译需要在执行所生成目标代码的同时,完成加载分析、运行环境设置、实时翻译、代码缓存管理、代码块切换等工作.一些技术如热路径优化、寄存器映射、多线程优化等提高了动态二进制翻译的效率,但仍未解决动态二进制翻译效率偏低的问题.静态二进制翻译在不运行目标程序的情况下,静态分析可执行程序,提取指令进行翻译,能采用复杂的代码分析和优化算法,它有充足的时间进行完整细致的优化,生成代码质量较高,运行效率较高.静态二进制翻译还可以利用程序以往执行的记录进行优化,即 profiling,获取更好的优化结果.

QEMU 是一个二进制动态翻译器[1],针对中间代码实施简单的活性分析以得到优化后的代码.但由于其优化算法本身占用运行时间并且提高生成代码的质量并不能减少基本块切换和生成代码维护开销,优化效果并不理想.为了提高翻译后代码段的质量,在二进制翻译中针对编译器设计采取了很多复杂的优化,将 LLVM 和 QEMU 结合的 HQEMU[4]就是典型的例子,HQEMU 在生成的中间代码层上针对改进的 LLVM 编译器进行优化,比如寄存器优化和标量运算矢量化等.与 QEMU 相比,这种优化导致了整体翻译效率的下降.为了降低优化开销,HQEMU 抛弃了在多核系统中的其他硬件线程或内核的优化,这又导致系统吞吐量明显降低.由于优化算法本身的开销,QEMU 在 TCG 中间表示层未能实现有效的优化.一些基于 LLVM 的动态二进制翻译器[5-9]虽然能够生成较高质量代码,但是翻译和块切换开销在运行时仍然存在.现有静态二进制翻译难以处理间接分支,而动态二进制翻译效率低,为处理以上问题,本文采用基于 QEMU 的静态二进制翻译框架 Static-QEMU(SQEMU)[10],SQEMU 能够分离翻译时间、代码优化时间与目标程序执行时间,并且代码优化的时间不受运行时的限制,能够采用不同层次的优化算法,使得 SQEMU 能够生成更高质量的目标代码.

因此,本文以 SQEMU 作为静态二进制翻译平台,对生成的目标代码中常规冗余指令和冗余存取 LOAD/STORE 指令进行优化,通过分析目标码生成指令特定数据依赖图(instruction-specific data dependence graph, IDDG),再利用该图将活性分析和窥孔优化的 2 种理论相结合,提出基于 IDDG 的冗余指令删除优化算法,有效删除目标码中冗余指令,以达到提高目标代码执行效率的目的.

本文的主要工作有 4 点:

1) 针对动态二进制翻译器 QEMU 因其翻译过程受动态执行的限制而不能进行深度细致优化等缺点,以及现有静态二进制翻译器难以处理间接分支的现状,提出一种基于 QEMU 的静态二进制翻译框架 SQEMU,SQEMU 能够分离翻译时间、代码优化时间与目标程序执行时间,并且代码优化的时间不受运行时的限制,能够采用不同层次的优化算法,使得 SQEMU 生成的目标代码质量更高.

2) 将 SQEMU 后端生成的目标代码作为输入参数,利用指令寄存器之间存在的数据依赖关系,生成指令特定数据依赖图(IDDG),将图理论灵活运用到冗余代码优化过程中,拓展了处理冗余代码的手段.

3) 利用 IDDG 分别对常规冗余代码进行活性分析、对冗余访存指令进行窥孔优化,将 2 种理论相结合,双重优化的思路避免了因方法单一而导致优化不够完备的弊端.实验结果表明,以 SPEC CPU2006 和 nbench 测试集为例,该算法对代码执行效率提升显著,最大提升可达 42%,整体性能测试表明优化后 nbench 测试集翻译效率平均提高约 20%,SPEC CINT2006 平均提高约 17%.

4) 本文提出的算法虽然是针对基于 QEMU 的静态二进制翻译框架 SQEMU 后端目标码实现的,但该算法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.

1　基于 QEMU 的静态翻译框架——SQEMU

QEMU 系统是目前较为先进的可实现多种异构平台映射的动态二进制翻译系统,具有支持平台多样、翻译效率相对较高、开源易移植等优点[11].QEMU 为实现多源多目标虚拟机,采用将源二进制代码翻译为 TCG 中间代码再翻译为目标代码的方式,可以实现将 X86,ARM,MIPS 下的 ELF 格式可执行文件翻译为 TCG 中间表示,再翻译为目标代码.

本文设计了基于 QEMU 的静态二进制翻译框架 Static-QEMU(SQEMU),采用 QEMU 的前端分析和 TCG 中间表示,继承了 QEMU 跨平台的特性.与 QEMU 和基于 LLVM 的动态二进制翻译器[5-6,8-9]相比,SQEMU 分离翻译、代码优化与目标程序执行阶段,使得在同等优化手段下 SQEMU 能够更高效.并且,由于代码优化时间不受运行时的限制,能够采用不同层次的优化算法生成高质量的执行代码.

SQEMU 包括前端源指令提取、TCG 中端分析优化、后端目标代码生成 3 个阶段,基本设计结构如图 1 所示.其中,源文件解析器、前端解码器构成 SQEMU 前端,负责将源平台二进制代码(本文即 X86 代码)翻译成中间代码 TCG(tiny code generator),TCG 中端分析优化器对中间表示进行平台无关优化,后端翻译器负责将 TCG 中间代码翻译成目标平台的二进制代码(本文中为 Alpha 代码).前端和 TCG 中端分析优化器采用 QEMU 的相应模块,删除了 QEMU 中控制中心、缓存管理和执行模块,后端添加了目标文件生成器.

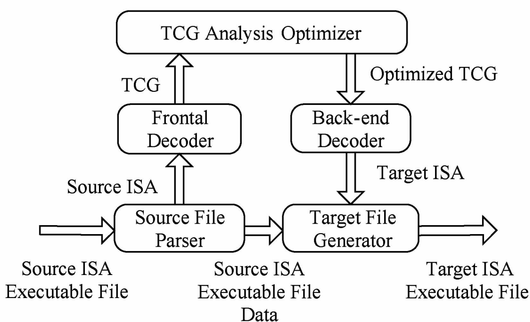


Fig. 1　Framework of SQEMU

图 1　SQEMU 框架设计

其中,前端解码器逐条对源平台指令进行解码,SQEMU 使用了 QEMU 的指令译码部分,根据译码器分析出的指令,生成相同语义的 TCG 指令.传统的优化策略仅在 TCG 分析优化器中针对 TCG 中间代码层进行平台无关优化[12],如活性分析和块内寄存器分配.而事实上由于仅靠软件翻译,一条 X86 指令翻译后经 TCG 中间代码优化后仍会生成多条 Alpha 代码,其中含有大量冗余指令和冗余访存操作,执行效率很低.针对这一现状,本文在目标代码层实现冗余代码优化.

2　冗余指令的发现

SQEMU 系统后端生成的 Alpha 代码的冗余是影响代码膨胀的最直接因素,通过分析 SQEMU 后端生成的 Alpha 代码,原 SQEMU 系统并没有对目标码进行冗余删除的优化,其中存在少量含有非活跃变量的常规指令和大量冗余访存指令,本节着重描述冗余访存指令.

冗余访存指令是指在该访存指令之前,有对同一个内存地址的读或者写指令,并且该地址的值仍保留在同一寄存器中,那么当前访存指令即可定义为冗余访存指令.在静态二进制翻译器 SQEMU 后端产生的目标码中,如果参与运算的寄存器值先用 ldl 指令从内存中取出,运算结束后再使用 stl 指令存入内存中,这样,当这 2 条指令之间存在数据相关性时,就会产生访存冗余的情况;或先使用 stl 指令将运算结束后的寄存器值存入内存中,当遇到下一条需要从相同位置取出同一寄存器值的 ldl 指令前,并没有对该寄存器值进行重新赋值,此时也会产生访存冗余,即存在类似图 2 所示的汇编程序段.

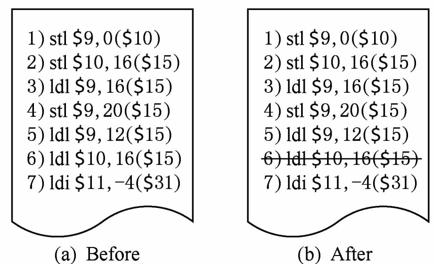


Fig. 2　Redundant code examples

图 2　冗余代码示例

在图 2 所示的目标码中,寄存器$10 在语句 2)中被存入内存 16($15)的位置,而语句 6)需要从同一内存位置 16($15)取出相同变量,由于语句 2)的源寄存器和语句 6)的目标寄存器相同,而在 2 条语句之间并没有对寄存器$10 和内存中 16($15)改变的指令,所以语句 6)中的 ldl 为冗余访存指令,该语句可在后续的优化过程中被直接删除.

通过对 SQEMU 的后端编码器按 TCG 中间表示生成的目标平台代码进行语义分析,通过提取操作码和寄存器,总结归纳出其执行特性,发现其中含有大量如表 1 所示的冗余指令对.

1) stl-ldl 型. 如果匹配到的 stl 和 ldl 指令之间并没有对同一寄存器进行重新赋值,且偏移量不变,则 ldl 指令为冗余指令.

2) stl-stl 型. 如果匹配到的 2 条 stl 指令之间并没有对同一寄存器进行重新赋值,且偏移量不变,则第 2 条 stl 指令为冗余指令.

3) ldl-ldl 型. 如果匹配到的 2 条 ldl 指令之间并没有对同一寄存器进行重新赋值,且偏移量不变,则第 2 条 ldl 指令为冗余指令.

4) ldl-stl 型. 如果匹配到的 ldl 和 stl 指令之间并没有对同一寄存器进行重新赋值,且偏移量不变,则 stl 指令为冗余指令.

Table 1 Redundancy Types of Instruction

表 1 访存指令冗余类型

stl-ldl Type	stl-stl Type	ldl-ldl Type	ldl-stl Type
stl *ra*,disp(*rb*)	stl *ra*,disp(*rb*)	ldl *ra*,disp(*rb*)	ldl *ra*,disp(*rb*)
ldl *ra*,disp(*rb*)	stl *ra*,disp(*rb*)	ldl *ra*,disp(*rb*)	stl *ra*,disp(*rb*)

3 冗余指令的优化思路

目标码中使用大量的临时寄存器来处理指令中数据运算与传递,针对目标平台后端生成的 Alpha 代码的特点,将活性分析和窥孔优化理论应用到 SQEMU 的后端代码优化过程中,以达到删除冗余代码,降低代码膨胀率的目的. 活性分析理论通常用于编译器中以判断临时变量的活跃,对基本块的正确划分是活性分析的前提. 正确划分基本块的关键就是确定基本块的首地址,基本块首地址按照一定规则确定,例如程序入口点、分支指令的下一条指令、分支指令的目标地址等. 其中,在处理分支指令的目标地址时情况较复杂,首先分析 X86 指令,找到分支指令,并分析其后的目标地址,以分支指令的目标地址作为基本块的首地址. 若间接跳转指令为 call 指令,由于其目的地址一定是某函数的首地址,而该函数的首地址往往在上一个函数的 return 指令之后,则根据划分规则,必然会被确定为基本块首地址;若分支指令 jump 是直接跳转指令,则其目的地址一定是一个常数,能够从静态二进制翻译后得到的目标代码中直接获取到;若 jump 为间接跳转指令,由于间接跳转指令的目标地址依赖于程序运行时寄存器的值,在静态二进制翻译中无法确定该指令的目标地址,因此间接跳转指令的目标地址确定问题成为静态二进制翻译的关键问题之一.

间接跳转指令的存在会导致静态翻译模式自动翻译某些程序时失败,但当静态翻译成功翻译时,生成程序的正确性是可以保证的. 解决间接跳转指令目标确定问题是为了完备的代码挖掘,即使用间接转移指令的目标地址来定位后继指令位置以确定基本块. 针对间接跳转指令的目标地址确定问题,课题组相继进行了不懈探索:吴伟峰提出一个改善其完备性的亚纯静态二进制翻译框架[13],该框架基于静态二进制翻译,并为翻译器提供此待翻译二进制程序对应的制导文件,翻译时根据制导信息提取系统(control and guide information picking system, CGIPS)提供的信息,有效解决间接跳转、间接调用和自修改代码等制约静态二进制发展的翻译完备性问题;卢帅兵提出在 SQEMU 中使用源地址索引映射表来确定间接跳转指令目标地址[10],以解决在静态二进制翻译中目标地址依赖于程序运行时寄存器的值,且在多次运行时分支目标地址可能改变而无法确定该指令目标地址的问题,但 SQEMU 是针对逐条指令翻译,破坏了基本块的整体性,无法采用针对基本块中冗余访存指令的优化算法.

结合实验室目前的研究成果,在实际处理间接跳转指令时,本文在静态二进制翻译框架 SQEMU 的基础上,使用文献[13]提出的执行路径逆向构造算法和特定路径的控制执行算法,利用线性扫描反汇编工具 objdump 处理待翻译程序,获得逆向构造所需汇编指令,进而定位出静态二进制翻译中影响基本块划分的间接跳转指令目标地址.

按照上述分析和算法,能够确定基本块首地址,进而全面正确地划分基本块.

3.1 针对常规冗余指令的优化

在对冗余访存指令优化前,先根据指令特性以及相邻目标码间寄存器的使用关系,通过分析目标码生成指令特定数据依赖图 IDDG. 它以 TCG 中间表示生成的汇编码 qemu.asm 作为输入,将目标码顺序生成相对应的指令相关节点,每个节点由相应指令的信息(操作码、寄存器、立即数等)组成. 一旦指令相关节点确立,节点之间的数据依赖关系则根据源寄存器和目的寄存器之间的使用关系被同时确立,且这 2 个节点用定向性依赖边相连. 因此,整个 IDDG 包括指令特性节点和依赖边.

在生成数据依赖图以前,先给出以下相关定义:

定义 1. 输出变量集合 $Out(S)$. 输出变量代表着对寄存器的写操作,由语句 S 的所有输出变量组成的集合称为 S 的输出变量集合.

定义 2. 输入变量集合 $In(S)$. 输入变量代表着对寄存器的读操作,由语句 S 的所有输入变量组成的集合称为 S 的输入变量集合.

定义 3. 引用变量集合 $Use(S)$. 表示语句 S 中被引用到的变量的集合.

定义 4. 定值变量集合 $Def(S)$. 表示语句 S 中被定值变量的集合.

定义 5. 依赖关系. 对于计算机程序,当事件或动作 A 必须先于事件 B 而发生时,称 B 依赖于 A.

定义 6. 数据依赖关系. 依赖关系是由于读/写或计算/使用同一数据而引起的,称这种关系为数据依赖关系. 已知语句 S 和 T,若存在变量 x 使之满足下述条件之一,则称语句 T 依赖于语句 S,记为 $S\delta T$;否则,语句 S 和 T 之间不存在依赖关系.

1) 若同时有 $x\in Out(S)$,$x\in In(T)$,且 T 使用由 S 计算出的 x 的值,则称 T 流依赖于 S,记为 $T\delta^f S$,用弧线表示.

2) 若同时有 $x\in In(S)$,$x\in Out(T)$,但 S 使用 x 的值先于 T 对 x 的定值,则称 T 反依赖于 S,记 $S\delta^a T$,用带×的弧线表示.

3) 若同时有 $x\in Out(S)$,$x\in Out(T)$,且 S 对 x 的定值先于 T 对 x 的定值,则称 T 输出依赖于 S,记为 $S\delta^o T$,用带◦的弧线表示.

活性分析理论是一种全局数据流分析,在此,我们将活性分析理论用于基本块内任意路径数据流分析,从全局范围来看一个变量是活跃的,如果存在一条路径使得该变量被重新定值之前它的当前值还要被引用.

针对 SQEMU 后端代码的活性分析算法是以基本块为单位逆向线性扫描生成的目标码,分析目标码并生成指令相关节点,确定节点之间的数据依赖关系即源寄存器和目的寄存器之间的使用关系,根据源寄存器在执行本条指令后基本块后续指令是否改变寄存器的值来判断寄存器的活跃性. 通过对目标码的活性分析,就能识别出当前值不再活跃的那些寄存器. 若该条语句中寄存器均失去活性,则可判定为冗余代码.

令 S 为基本块内一条语句,定义 $F(S)$ 为基本块内语句 S 的前驱语句集合,$N(S)$ 为语句 S 的后继语句集合,则根据定义 6 中规定的数据依赖关系一定存在至少一个寄存器 x 使得:

$$S\delta^f i\text{,即 } x\in Out(i)\cap In(S);$$

$$j\delta^f S\text{,即 } x\in Out(S)\cap In(j).$$

其中,$i\in F(S)$,$j\in N(S)$.

定义 $LiveIn(S)$ 为在语句输入变量集合中为活跃变量的集合,定义 $LiveOut(S)$ 为语句输出变量集合中为活跃变量的集合. 其中,$LiveIn$ 和 $LiveOut$ 并不是相互独立的,则有:

$$\bigcup LiveOut(i)=\bigcup LiveIn(j),\tag{1}$$

其中,$i\in F(S)$,$j\in N(S)$.

也就是说,一个基本块内,某条指令之前的语句其目的寄存器是活跃的仅当该条指令后继指令源寄存器为活跃的. 如果该指令没有后继,则其 $LiveOut$ 为空.

根据定义 3,$Use(S)$ 是一个集合常量,其值由语句 S 唯一确定,易得,如果 $x\in Use(S)$,则 $x\in LiveIn(S)$,即:

$$LiveIn(S)\supseteq Use(S).\tag{2}$$

根据定义 4,$Def(S)$ 也是一个集合常量,其值由语句 S 唯一确定,如果一个寄存器在语句 S 的输出变量集合中为活跃的且 $x\notin Def(S)$,则它在 S 的输入变量集合中一定为活跃值,即:

$$LiveIn(S)\supseteq LiveOut(S)-Def(S).\tag{3}$$

通过分析可知,一个寄存器在语句输入变量中为活跃的,则一定有:或者它在语句的 Use 中,或者它在语句的输出变量中为活跃的且在语句中没有被重新定值. 因此,有等式:

$$LiveIn(S)=Use(S)\cup(LiveOut(S)-Def(S)),\tag{4}$$

式(4)对于基本块内每条语句均成立.

若对于语句 S 中被定值的变量集合不属于语句 S 输出变量中活跃变量的集合,则这条定值指令可判定为冗余指令,即:

$$LiveOut(S)\cap Def(S)=\varnothing.\tag{5}$$

设静态二进制翻译目标码中指令规则为 Π,定义指令输入变量数、输出变量数、变量总数等性质,$\Pi in(S)$ 表述语句 S 这条指令输入变量数,$\Pi out(S)$ 表示该指令输出变量数,$\Pi all(S)$ 表示该指令变量总数,N 表示基本块内语句总数.

① 如果 $\Pi out(S)=0$,指令只引用变量,并未对变量定值;

② 否则,该指令对变量重定:

$$\Pi all=\sum_{S=N}^{0}\Pi all(S),\tag{6}$$

其中，$\Pi all(S)=\Pi out(S)\cup\Pi in(S)$.

若存在 $j\delta^{f}S$，则有：

$$Def(j)=Def(S)-\Pi(S)*Def(S),\quad(7)$$

$$Use(j)=Use(S)+\Pi(S)*Def(S),\quad(8)$$

其中，“$*$”表示对$\Pi(S)$的重定义.

设 $Use^{\sim}(S)$ 表示语句 S 在按照指令规则语句中寄存器被引用集合，由式(8)可得：

$$Use(j)=Use^{\sim}(S).\quad(9)$$

首先设定对活跃寄存器 x 重新定值的语句 j 是距离源寄存器所在语句 S 最短的语句. 依据冗余指令的判定公式可得：

$$Def(j)\cap\sum_{l=S+1}^{j}Use^{\sim}(x)=(Def(j))\cap\sum_{l=S+1}^{j}((\Pi(x)*Def(x))+Use(x))=Def(j)\cap\sum_{l=S+1}^{j}(\Pi(x)*Def(x))\cup\left(Def(i)\cap\sum_{l=S+1}^{j}Use(x)\right).\quad(10)$$

若 $Def(j)\cap\sum_{l=S+1}^{j}Use^{\sim}(x)=\varnothing$，则语句 S 可判定为冗余语句. 由设定可知，依赖边(S,j)是对活跃寄存器 x 重新定值的最短路径，所以，在语句 S 和语句 j 之间，不存在任意语句 i，使得$\Pi(i)*Def(i)\neq\varnothing$，即对于任意 $x\in(S+1,j)$，均 $\sum_{l=S+1}^{j}(\Pi(x)*Def(x))=\varnothing$成立. 所以式(10)可化简为

$$Def(i)\cap\sum_{l=S+1}^{j}Use(x).\quad(11)$$

若在依赖边(S,j)中没有任意语句 i 引用寄存器 x，即 $\sum_{l=S+1}^{j}Use(x)=\varnothing$，则 $Def(j)\cap\sum_{l=S+1}^{j}Use^{\sim}(x)=\varnothing$，所以语句 S 中的寄存器可判定为失去活性寄存器，而语句 S 即为冗余指令.

综上所述，若存在语句 T 使得 $S\delta^{o}T$，且在 S 与 T 之间并没有对寄存器使用或重定值，则 S 语句中的寄存器被判定为失去活性，该指令为冗余访存指令.

失去活性的寄存器所在指令用“#”标识，不再翻译成目标指令. 将生成的指令相关节点以基本块为单位存储在数组 char *bb*[][*BBCOLUMS*]中，用指针 STLD_INFO **temp* 逆向顺次移动. 为标识基本块内寄存器的所有使用情况，开辟一个大小为基本块寄存器总体数量的数组空间，并初始化为 1，表示基本块内寄存器全部为非活跃. 然后利用指针 STLD_INFO **temp* 对生成的指令相关节点进行分析，判断指令目标寄存器对应在数组空间是否标记为 1：

1）若全为 1，即寄存器已经失去活性，该指令即可视为冗余访存指令；

2）若不全为 1，将指令的源寄存器和目的寄存器对应在数组空间的值标记清 0，表明寄存器是活跃的.

依次循环迭代直到基本块入口点，完成整个基本块的寄存器活性分析.

活性分析的关键即确定活跃变量，先明 Alpha 目标平台的寄存器特性，寄存器使用规则如表 2 所示. 特殊寄存器如$26，$27，$29，$30，$31 在活性分析时始终标记为活，即标记为 0. 具体如算法 1.

Table 2 Usage Rule of Register

表 2 寄存器使用规则

Register	Usage Rule
$9-$14	Be peserved across procedure calls.
$15 or $fp	Contain the frame pointer.
$16-$21	Be used to pass the first six integer type actual arguments.
$26	Contain the return address.
$27	Contain the procedure value and be used for expression evaluation.
$29 or $gp	Contain the global pointer.
$30 or $sp	Contain the stack pointer.
$31	Always have the value 0.

算法 1. 活性分析算法.

① 初始化基本块内寄存器；
  将所有特殊寄存器标记为活，即 0；
  将一般寄存器标记为死，即 1；
  设基本块总共有 n 条语句；
② for ($i=n-1$; $i\geqslant 0$; $i--$)
③ 　提取语句 S 的 $In(S)$，$Out(S)$；
④ end for
⑤ if ($Out(S)$是活性的)
⑥ 　把 $Out(S)$标记为死，把 $In(S)$标记为活性；
⑦ end if
⑧ if ($Out(S)$为特殊寄存器)
⑨ 　把 $Out(S)$标记为活性；
⑩ else
⑪ 　删除当前语句 S；
⑫ end if

从 SQEMU 系统后端生成的类 Alpha 代码中选取一段用于活性分析,如图 3 所示:

(a) Before

```
1) ldl $10,8($31)
2) addl $9, $13, $11
3) stl $9,32($15)
4) ldl $13,32($15)
5) ldl $9,8($31)
6) stl $10,40($9)
7) addl $9, $10, $11
8) stl $11,32($15)
```

(b) After

```
1) ldl   R1,reg1
2) addl  R2,R3,R4
3) stl   R2,reg2
4) ldl   R3,reg2
5) ldl   R2,reg1
6) stl   R1,reg3
7) addl  R2,R1,R4
8) stl   R4,reg2
```

Fig. 3 Redundant code of common instruction

图 3 常规指令冗余代码

图 3(a)表示任意选取的一段目标码,图 3(b)表示将目标码中寄存器进行符号化后所得指令片段. 根据目标平台 Alpha 指令访存指令格式,如 ldl *ra*, disp(*rb*),其中,*ra* 表示目标平台寄存器,disp(*rb*)表示目标平台内存地址空间,在 SQEMU 系统中,目标平台开辟一块空间用来模拟 X86 寄存器,因此 disp(*rb*)也可视为寄存器. 如图 3 所示,为描述清晰,将寄存器\$9,\$10,\$11,\$13 分别命名为 R_2,R_1,R_4,R_3,将内存地址空间按顺序标记为 reg_1,reg_2,reg_3. 活性分析时开辟的寄存器总数和即 7 个整形数组空间,并初始化为 1,表示 7 个寄存器起始状态均为非活跃的,逆向扫描并依据操作码对指令的输入变量集合 *In* 和输出变量集合 *Out* 进行着色. 图 4 表示逆向分析寄存器活跃性的图着色过程,其中黑色为非活跃寄存器,白色表示活跃寄存器,阴影表示已判定为冗余代码中的寄存器.

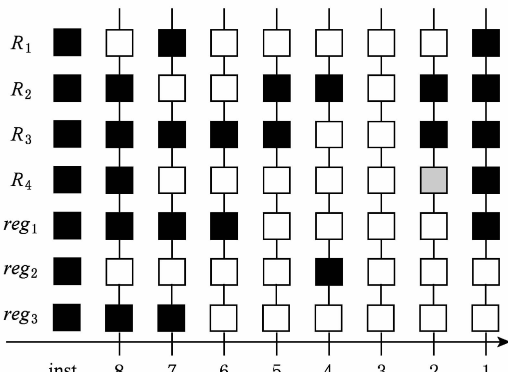


Fig. 4 Activity analysis diagram coloring process

图 4 活性分析图着色过程

图 4 中 inst7 寄存器 R_4 为输出变量,通过逆向活性分析发现,R_4 在语句 2)中被重新赋值,且在这 2 条指令之间寄存器 R_4 并未被引用,判定输出变量 R_4 已不活跃,所以 R_4 所在的语句 2)为冗余指令,可在后续优化过程中直接删除.

3.2 针对访存冗余指令的优化

通过对目标码中寄存器进行活性分析,删除一定的失去活性的寄存器和冗余代码,但对 SQEMU 生成的后端 Alpha 代码分析时发现这种冗余代码并不常见,利用活性分析技术进行优化对整体翻译性能提升程度有限,无法从全局做到代码最优化,且仍存在如冗余代码发现所述大量冗余访存指令. 代码产生器依次逐条将中间代码翻译为目标代码时,通常使目标代码中产生冗余指令或者不太优的结构. 在目标代码级上,可以利用窥孔优化(peephole optimization)有效处理冗余代码,改进代码质量.

窥孔优化是一种局部优化方法,其基本原理是通过考察编译器所生成的目标代码中一小段相邻指令(称为窥孔),比如一个基本块中的目标码,通过整体分析和指令转换,把这些指令替换为更短和更快的一段指令,以此来提高代码质量.

美国 Stanford 大学的静态二进制翻译器利用窥孔优化对目标代码实施高质量的等价替换,获得了更为优化的执行效率. 窥孔优化一般包括冗余访存指令的删除、不可达代码的删除、控制流优化、强度削弱等. 由于窥孔优化需要对目标码进行若干遍处理,开销较大,较少应用在动态二进制翻译中,是静态二级制翻译和编译器的常用优化手段. 虽然代码转换限于局部,只需很少访问内存,但可能会带来很大性能提升.

从 SQEMU 系统生成的后端 Alpha 代码中随机抽取一段目标码如图 5 所示:

(a) Before

```
1) ldl $9,32($15)
2) ldi $10,8($31)
3) addl $9, $10, $9
4) ldl $10,40($15)
5) stl $10,0($9)
6) stl $9,32($15)
7) stl $10,40($15)
8) ldl $9,32($15)
9) sub $9, $10, $9
10) stl $9,32($15)
```

(b) After

```
1) ldl   R1,reg1
2) ldi   R2,reg2
3) adll  R1,R2,R1
4) ldl   R2,reg3
5) stl   R2,reg4
6) stl   R1,reg1
7) stl   R2,reg3
8) ldl   R1,reg1
9) sub   R1,R2,R1
10) stl  R1,reg1
```

Fig. 5 Register renaming

图 5 寄存器重命名

如图 5 所示,为描述清晰,将寄存器\$9,\$10 分别命名为 R_1,R_2,将内存地址空间按出现顺序标记为 reg_1,reg_2,reg_3,reg_4. 根据定义,图 5 中语句的输入输出变量集合如表 3 所示.

Table 3 Input and Output Variables Set

表 3 输入输出变量集合

Statement	$In(S_i)$	$Out(S_i)$
S_1	reg_1	R_1
S_2	reg_2	R_2
S_3	R_1, R_2	R_1
S_4	reg_3	R_2
S_5	R_2	reg_4
S_6	R_1	reg_1
S_7	R_2	reg_3
S_8	reg_1	R_1
S_9	R_1, R_2	R_1
S_{10}	R_1	reg_1

图 6 左图所示为一段后端 Alpha 代码，将寄存器和立即数顺次编号，生成指令相关节点，并逐条分析确定指令相互之间数据依赖关系即源寄存器与目的寄存器之间的使用关系，如果前一个节点的目的寄存器被用作后一个的源寄存器，将这 2 个节点用定向性依赖边相连. 根据定义，若 $T\delta^f S$，则用弧线表示；若 $S\delta^a T$ 用带×的弧线表示；若 $S\delta^o T$，用带 o 的弧线表示. 得到如图 6 左图所示的指令特定数据依赖图. 其中，第 3 个指令相关节点中源寄存器 \$9 与 \$10 分别为第 1 个与第 2 个指令相关节点中的目的寄存器，即寄存器 \$9 和 \$10 作为语句 S_3 的输入变量分别为语句 S_1 和语句 S_2 的输出变量，所以语句 S_3 和语句 S_1、语句 S_2 之间存在流依赖关系，且语句 S_3 输出依赖于语句 S_1. 再如语句 S_7 关于寄存器 \$10 流依赖于语句 S_4，且这 2 个语句之间并不存在对寄存器 \$10 重赋值的情况，即语句 S_4 和语句 S_7 之间并没有寄存器 \$10 的输出依赖，语句 S_7 的访存操作是没有任何意义的，所以第 7 个指令相关节点可标记为冗余节点. 同理，第 8 个指令相关节点也可标记为冗余节点. 删除冗余节点后可简化为图 6 右图中数据依赖图.

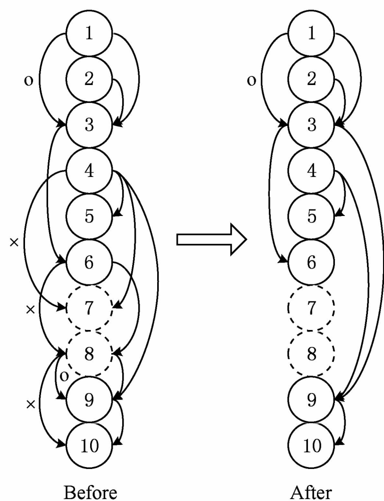


Fig. 6 Instruction-specific data dependence graph

图 6 指令特定数据依赖图

指令相关节点包含了指令的特性，如操作码、立即数段、寄存器段，一个节点 S_i 与预先定义的 LOAD/STORE 指令格式相匹配，根据 IDDG，由节点间定向性依赖边可对节点 S_i 与其目标节点 S_j 之间的数据依赖关系依据冗余指令匹配类型进行以下分类判定：

判定 1. 若冗余访存指令类型为 stl-stl 型或 ldl-ldl 型，则一定存在寄存器 x 与寄存器 y，其中：

$$\begin{cases} x\in In(S_i)\cap In(S_j) \\ y\in Out(S_i)\cap Out(S_j) \end{cases}，且\ S_i\delta^o S_j.$$

如果 2 条语句 S_i 与 S_j 之间存在语句 S_p 对寄存器重新赋值，即 $y\in Out(S_p)$ 且 $x\notin In(S_p)$，同时满足 $S_p\delta^o S_j$，则 2 条语句 S_i 与 S_j 不是冗余访存指令；如果 3 条语句之间不存在对寄存器重新赋值的指令，则依据活性分析结论，一定存在一条语句 S_q 使用寄存器 y 作为输入变量，否则，语句 S_i 与 S_j 为冗余访存指令，可删除 S_j.

即若冗余访存指令为 stl-stl 型或 ldl-ldl 型时，有：

$$R=(y\in Out(S_p)\cap x\notin In(S_p))\cup(y\in In(S_q)),$$

其中 $i<p,q<j$，当 $R=\varnothing$ 时，语句 S_i 与 S_j 可判定为冗余访存指令.

判定 2. 若冗余访存指令类型为 stl-ldl 型或 ldl-stl 型，则一定存在寄存器 x 与寄存器 y，其中：

$$\begin{cases} x\in In(S_i)\cap Out(S_j) \\ y\in Out(S_i)\cap In(S_j) \end{cases}，且\begin{cases} S_i\delta^a S_j \\ S_j\delta^f S_i \end{cases}.$$

如果 2 条语句 S_i 与 S_j 之间存在语句 S_p 对寄存器重新赋值，即 $y\in Out(S_p)$ 且 $x\notin In(S_p)$，同时满足 $S_p\delta^o S_j$，则 2 条语句 S_i 与 S_j 不是冗余访存指令；如果 2 条语句之间不存在对寄存器重新赋值的指令，则依据活性分析结论，一定存在一条语句 S_q 使用寄存器 x 作为输入变量，否则，语句 S_i 与 S_j 为冗余访存指令，可删除 S_j.

即若冗余访存指令为 ldl-ldl 型时，有：

$$R=(y\in Out(S_p)\cap x\notin In(S_p))\cup(y\in In(S_q)),$$

其中 $i<p,q<j$，当 $R=\varnothing$ 时，语句 S_i 与 S_j 可判定为冗余访存指令.

4　冗余指令优化算法

本文提出的优化方法是基于IDDG实现的，将针对常规冗余指令活性分析与冗余访存在指令窥孔优化相结合，实现对目标码冗余指令的优化删除算法.

根据第2节和第3节的分析，SQEMU系统生成后端Alpha代码仍存在大量冗余，影响编译效率，对此，本文提出一种静态二进制翻译冗余目标代码删除优化算法，对基本块内冗余代码进一步分析. 在经过控制流分析和数据流分析之后，其方法为只需要记录每次循环迭代基本块内一条ldl或stl指令和寄存器被重写指令序号，每当记录一条新的ldl和stl指令时，遍历已经记录的ldl和stl指令，2条匹配的ldl或stl指令之间若不存在寄存器被重写的情况，即可判断为冗余指令，过程如下：

1）从SQEMU系统后端生成的Alpha代码中顺次获取相应的指令信息，如操作码、寄存器和立即数等，根据指令特性以及相邻目标码间寄存器的使用关系，生成指令相关节点和指令特性数据依赖图IDDG，对基本块内寄存器进行活性分析，失去活性的寄存器所在指令用“#”标识，不再翻译成目标指令. 将生成的指令相关节点以基本块为单位存储在数组char *bb*[][*BBCOLUMS*]中，创建一双链表记录基本块内ldl和stl指令序号，指令类型和全局变量的偏移量.

2）依次获取目标码中指令的操作码和操作数，将除ldl和stl指令之外被赋值寄存器的结构体ST_TEMP插入到全局链表数组*stTmp*[32]中，由于SQEMU系统后端仅涉及\$0～\$31共32个寄存器，所以全局链表数即寄存器数，ST_TEMP结构体按指令中被改变的寄存器值插入到全局链表相应位置.

算法流程如图7所示：

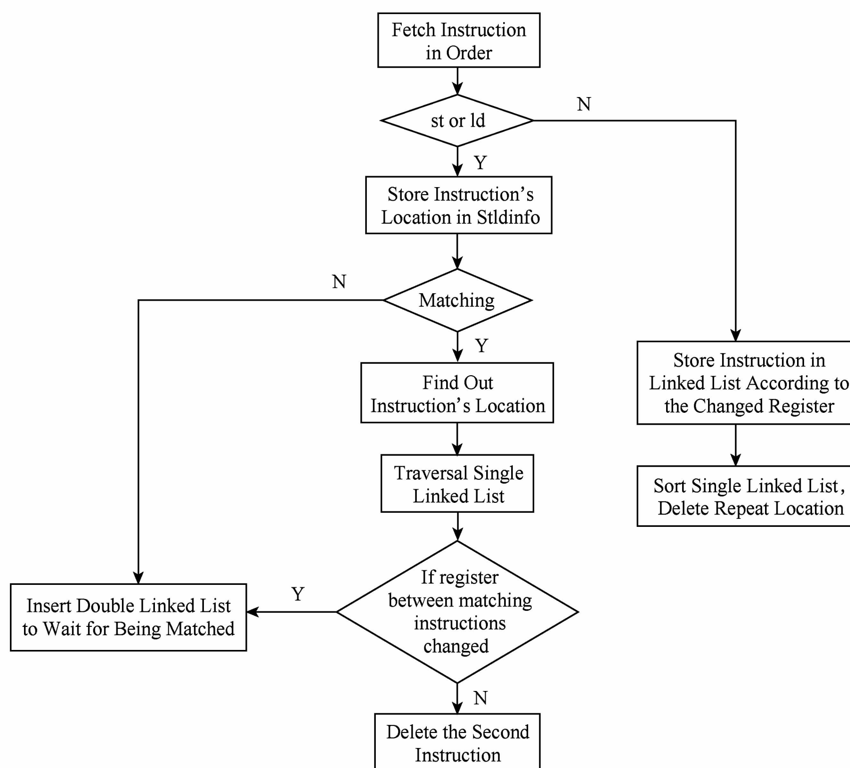


Fig. 7　Algorithm flow chart

图7　算法流程图

分别初始化ldl和stl指令的双链表和非ldl或stl指令的单链表，依次分析获取到的指令和寄存器值，确定语句*S*的输入变量集合*In*(S)和输出变量集合*Out*(S)，再判断获取到的指令类型是否为ldl或stl指令. 若获取到的指令非ldl或stl指令，根据相应指令特点，则记录输出变量即指令中被改变的寄存器值，依据寄存器值将此语句的序号插入单链表数组中；若获取到的指令是ldl或stl指令，遍历存储ldl和stl指令的双链表，顺次检查匹配双链表中指令的操作码、寄存器值和立即数，判断双链表中的指令与待插入的ldl或stl指令对应变量是否相等，若不相等，直接把指令插入到双链表中，若双链

表中存在指令与待插入指令相匹配,则依据冗余指令对的类型分2种情况讨论:

① stl-stl 型或 ldl-ldl 型,即双链表中已存在的指令与待插入指令为同一指令;

② stl-ldl 型或 ldl-stl 型,首先根据 IDDG 确定双链表中已匹配的指令节点与待插入指令节点之间的数据依赖关系,设双链表中已匹配的指令节点为 S_i,待插入的指令节点为 S_j,则根据上文提到的冗余节点的判定条件可知,若 S_i 流依赖于 S_j 且 S_j 反依赖于 S_i,则2条指令存在是冗余访存指令的可能.

若出现以上2种情况之一,下一步可依据 IDDG 确定2条指令节点之间的数据依赖关系,判断匹配的2条语句之间是否存在其他指令节点与这条指令节点形成输出依赖关系,即2条语句之间是否存在对寄存器重新赋值的新指令,同时遍历该寄存器所在单链表,依据指令序号确认这条新指令是否在2条匹配的指令之间.若存在,则2条已匹配的指令并不是冗余访存指令,可将待插入指令直接插入双链表中,待后续指令与其匹配;若不存在,则待插入指令即可判断为冗余访存指令,同时用“#”标识该条指令,进而完成整个冗余访存指令删除的优化过程.

5 实验与分析

为验证冗余访存优化算法的实际优化效果,采用上述 SQEMU 作为静态二进制翻译平台,通过正确性测试、整体性能测试和测试数据分析对提出的算法进行评估.

5.1 实验环境

1) 测试集.在 SQEMU 上运行基准测试集 SPEC CPU2006 和 nbench-2.2.3 benchmark suite (QEMU 官方网站推荐用于评测性能的测试程序).所有的执行速度(iteration/s)都是5次测试的算术平均值.

2) 源平台.32位 X86 平台,Fedora11 Linux 2.6.29.4,gcc 4.4.0.

3) 目标平台.64位 Alpha 平台,中标麒麟 Linux 3.8.0,gcc 4.3.0.

5.2 正确性测试

在 X86 平台上编译 SPEC CPU2006 和 nbench 测试集,启动优化后的 SQEMU,输入编译好的 X86 测试集可执行文件,从调试信息获取执行结果.实验显示本算法翻译执行的结果与 SPEC CPU2006 和 nbench 正确执行时的结果一致,说明本算法能够进行正确的翻译,具有较高可信度.

5.3 整体性能评价

为体现冗余访存优化算法对 SQEMU 整体性能的提高效果,分别统计算法改进前后 SQEMU 翻译执行 nbench 和 SPEC CPU2006 测试集性能指标.

对 nbench 测试集进行测试,记 SQEMU 优化前执行性能指标为 I_1,SQEMU 优化后性能指标为 I_2,性能指标单位为 iteration/s,即每秒循环次数.加速比记作 I_2/I_1.

对基准测试集 SPEC CPU2006 进行测试,由于 C++ 程序间接调用情况更复杂更频繁,是影响静态二进制翻译性能的主要瓶颈之一,未选取 C++ 程序进行对比.且不同语言程序所需编译器不同,为增加数据可比性,对 SPEC CPU2006 采用其中 C 程序作测试用例.分别统计算法改进前后 SQEMU 翻译执行 SPEC CPU2006 所用时间,记 SQEMU 优化前执行时间为 T_1,SQEMU 优化后执行时间为 T_2,加速比记作 T_1/T_2.

从上述测试结果中可以发现,基准测试集 SPEC CPU2006 和 nbench 测试集在优化后执行效率明显提升,测试用例不同其性能提升的效果也不同.由表4及图8中数据可得,通过采用针对冗余访存指令的优化算法可使得 nbench 测试集性能提升 6%~42%,且平均性能提升约为 20%;由图9中数据发现使用优化算法后 SPEC CPU2006 测试集性能提升 1%~32%,SPEC CINT2006 平均性能提升约 17%,SPEC CFP2006 平均性能提升仅约为 5%.从实验结果中不难发现,该算法具有明显的优化效果.

Table 4 nbench Test Results

表4 nbench 测试结果

nbench Tests	I_1	I_2
NUMERIC_SORT	14.544	19.970
STRING_SORT	6.319	7.721
BITFIELD	3.33E+06	4.72E+06
FP EMULATION	1.482	1.578
FOURIER	2.18E+03	2.46E+03
ASSIGNMENT	0.439	0.510
IDEA	60.916	69.696
HUFFMAN	19.645	22.415
NUEURAL NET	0.192	0.204
LU DECOMPOSITION	4.535	5.315

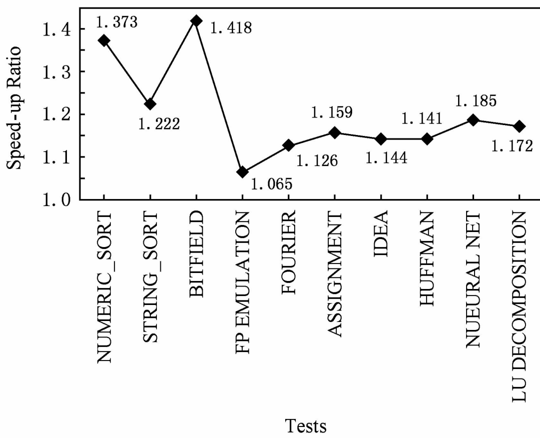


Fig. 8　nbench speed-up ratio

图 8　nbench 加速比

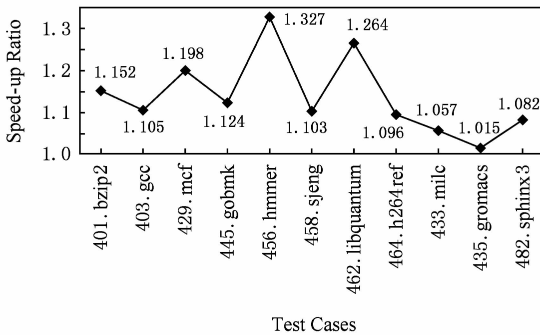


Fig. 9　SPEC CPU2006 speed-up ratio

图 9　SPEC CPU2006 加速比

5.4　测试数据分析

从表 4 针对 nbench 测试集得出的测试结果以及图 8 和图 9 中分别针对 nbench 和 SPEC CPU2006 测试用例所得加速比中发现，测试用例不同，加速比也不同，该冗余指令优化算法对不同程序的优化程度也不尽相同. 由此推断，优化效果与程序自身指令特性和程序任务相关.

分别统计 nbench 和 SPEC CPU2006 测试集中选取的测试用例在优化前和优化后的指令总数，得到如图 10、图 11 所示柱状图. 分析柱状图中指令总数，并设优化效益为 γ，则有公式：

$$\gamma=\frac{\text{优化前的指令总数}-\text{优化后的指令总数}}{\text{优化后的指令总数}},$$

不难发现，nbench 执行时间的加速比与指令的优化效益成正相关，由此可推断，冗余指令越多，被优化掉的指令数也越多，优化效益 γ 也越大，最终获得的执行时间的加速比也越大，即冗余指令删除算法的效果更明显. 而 SPEC CPU2006 由于其规模较大、测试集较为复杂、间接跳转指令数较多等原因，执行时间的加速比与指令的优化效益 γ 并不完全正相关.

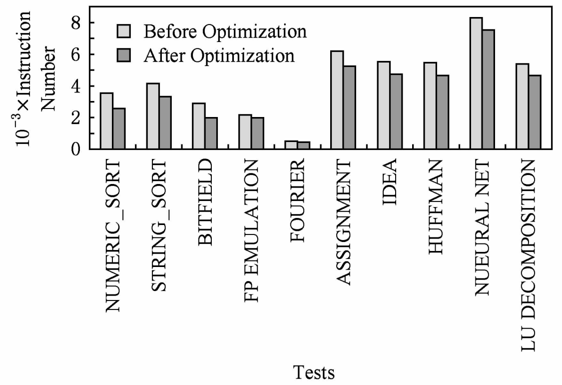


Fig. 10　nbench instruction number

图 10　nbench 指令数

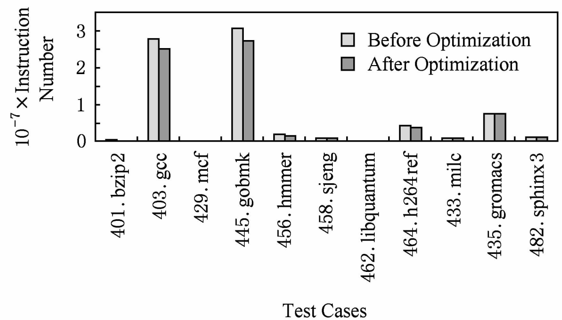


Fig. 11　SPEC CPU2006 instruction number

图 11　SPEC CPU2006 指令数

首先分析 nbench 测试用例任务，如表 5 所示：

Table 5　nbench Tasks

表 5　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.

下面对 nbench 测试加速比最小的测试用例 FP EMULATION 与所得加速比最大的前 3 个测试用例 BITFIELD，NUMERIC_SORT，STRING_SORT 进行深入分析.

1) 查看加速比最小的 FP EMULATION 源码发现,该测试用例内包含 3 个主要函数:*DoFPUTransIteration*, *TrapezoidIntegrate*, *thefunction*. 分别统计 3 个函数优化后各自的指令数,并计算其占 FP EMULATION 指令总数的百分比,如图 12 所示. FP EMULATION 测试用例中主要占用执行时间的函数是 *DoFPUTransIteration*,且其指令数也最多,而 *DoFPUTransIteration* 函数中主要操作为模拟基本数学运算,主要涉及 *memmove*,与优化算法关系不大.

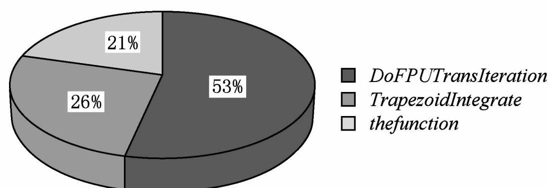


Fig. 12 Ratio of FP EMULATION instruction number

图 12 FP EMULATION 指令数比例

2) 对 BITFIELD 测试用例使用冗余指令删除算法优化,其所得加速比最大即达到 1.418. BITFIELD 是一系列位操作函数,位操作需要大量使用通用寄存器,而本文提出的冗余指令删除优化算法对通用寄存器优化效果最为明显,且 BITFIELD 测试用例中冗余指令数目较多,所以优化效果最好.

3) NUMERIC_SORT 和 STRING_SORT 加速比分别为 1.373 和 1.222,NUMERIC_SORT 实现对 32 位整型数组排序的任务,而 STRING_SORT 实现的任务是对任意长度字符串数组的排序,这 2 个测试用例使用整型寄存器和通用寄存器较多,如果 2 条指令使用同一寄存器,则极大可能产生冗余存取指令. 而冗余指令删除优化算法主要针对冗余存取指令,所以优化效果也比较好.

其次分析基准测试集 SPEC CPU2006,利用执行路径逆向构造算法和特定路径的控制执行算法定位出阻碍静态二进制翻译间接跳转指令的目标地址. 针对基准测试集 SPEC CPU2006 分别统计出其间接跳转指令数和间接调用指令数,如表 6 所示.

从 5.3 节整体性能评价中得知,通过采用针对冗余访存指令的优化算法可使 nbench 测试集平均性能提升约为 20%,SPEC CINT2006 平均性能提升约 17%,SPEC CFP2006 平均性能提升仅约为 5%. SPEC CPU2006 整体测试优化效益低于 nbench 测试集,一方面由于 SPEC CPU2006 测试集复杂度较高,另一方面从定位到的阻碍静态二进制翻译间接跳转指令的目标地址来看,SPEC CPU2006 含有较多间接分支指令,而 nbench 热路径中不存在间接跳转指令. 间接分支指令影响基本块的划分,例如,当间接跳转指令跳转到原有基本块内部时,根据基本块划分规则,该基本块需要进行重新划分,导致基本块规模减小,而本文所述冗余访存指令优化算法是基于基本块进行优化,基本块规模减小将导致优化程度随之降低. 通过分析表 6 中间接分支指令数测试结果与图 9 中 SPEC CPU2006 加速比发现,优化加速比与间接跳转指令数大致成反比例关系.

Table 6 Indirect Branch Instruction Test Results

表 6 间接分支指令数测试结果

SPEC CPU2006	Test Case	Indirect Jump Instruction	Indirect Call Instruction	Indirect Branch Instruction
SPEC CINT 2006	401. bzip2	3	19	22
	403. gcc	15	18	33
	429. mcf	5	1	6
	445. gobmk	10	30	40
	456. hmmer	0	1	1
	458. sjeng	21	3	24
	462. libquantum	1	0	1
	464. h264ref	15	367	382
SPEC CFP 2006	433. milc	9	5	14
	435. gromacs	33	285	318
	482. sphinx3	5	7	12

从测试用例类型来看,SPEC CINT2006 平均性能提升程度接近 nbench 测试集,而 SPEC CFP2006 远低于 nbench. 由于 SPEC CFP2006 浮点测试用例中热代码主要是浮点指令,SQEMU 使用函数模拟浮点指令的功能,该算法对此类指令优化效果较差.

从上述分析可得,本文提出的冗余指令优化算法对不同测试用例得到的优化效果也不尽相同,主要取决于程序自身指令特性和程序任务. 待优化的程序冗余指令数目越多,在执行过程中主要使用整型寄存器和通用寄存器,则会取得较好的优化效果. 而间接分支指令和浮点操作也会影响实际优化效果,间接跳转指令和浮点操作越少则取得的优化效果将越好.

6 相关研究

文献[14-15]中针对 TCG 使用大量临时变量来处理指令数据运算与传递,QEMU 引入寄存器活性

分析优化技术和存储转发优化技术来删除中间表示中存在的冗余变量，减少中间表示指令，降低代码的膨胀率，但是中间表示优化对整体翻译性能提升程度有限且仍存在冗余指令.

文献[4]中为了提高翻译后代码的质量，使用改进LLVM编译器针对生成的LLVM中间代码进行优化，其中包括增加寄存器优化和标量运算矢量化等，和QEMU相比，这种优化过程却导致整体翻译效率损失超过60倍，为了降低优化开销，HQEMU卸载了在多核系统中的其他硬件线程或内核的优化过程，严重降低了系统吞吐量.

文献[2]中引用活性分析技术对QEMU后端MIPS冗余MOVE指令提出了删除算法，优化了目标代码的冗余指令，但仅仅针对冗余MOVE指令，优化方法过于单一，不能取得较完备的优化效益.

文献[16]中针对动态翻译时高速缓存负荷数倍膨胀导致翻译器性能下降的问题，提出基于指令高速缓存与数据高速缓存访问负荷动态均衡的软硬件协同翻译方法，该方法主要为处理器设计高速缓存负荷平衡状态和负荷转化通道，通过软硬件协同配合的方式把调度器地址转换操作在指令高速缓存上产生的负荷转化到数据高速缓存，有效提高了数据高速缓存的利用率和动态翻译器性能.但并未提及对大量冗余代码造成代码膨胀导致翻译器性能下降的处理.

7　结束语

本文提出了一种基于静态二进制翻译器SQEMU的冗余代码优化算法，该算法针对X86指令使用SQEMU翻译生成多条Alpha目标代码的过程中，含有大量冗余操作的缺陷，利用指令特定数据依赖图结合活性分析和窥孔优化思想，分别针对常规冗余指令和冗余访存指令进行优化.实验结果表明，经优化后，在64位Alpha目标平台上利用SQEMU翻译X86程序其运行速度得到可观提升，该算法优化效益最高可达到42%.该算法具有相当强的通用性，很多二进制翻译框架存在访存指令的冗余问题，不限于目标平台和指令形式，例如文献[13]中提到的二进制翻译系统UQBT和文献[17]提到的一个动态翻译结合解释执行的二进制翻译系统DigitalBridge，由于其对源平台指令的每一次访问寄存器都需要进行访存操作，必然存在冗余访存指令，效率较低.所以该算法对跨平台应用程序的移植具有极高的现实意义，对降低二进制翻译后的代码膨胀率和推动国产处理器的发展有重要意义.

参　考　文　献

[1] Bellard F. QEMU, a fast and portable dynamic translator [C] //Proc of the 9th IEEE Working Conf on Reverse Engineering. Piscataway, NJ: IEEE, 2002: 35-44

[2] Song Qiang. Research of optimization for binary translator QEMU based on Godson [D]. Hefei: University of Science and Technology of China, 2012 (in Chinese)
(宋强. 基于龙芯的二进制翻译器QEMU优化研究[D]. 合肥：中国科学技术大学，2012)

[3] 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)

[4] Hong Ding-Yong, Hsu Chun-Chen, Yew Pen-Chung, et al. HQEMU: A multi-threaded and retargetable dynamic binary translator on multicores [C] //Proc of CGO'12. New York: ACM, 2012: 104-113

[5] Shen B, You J, Yang W, et al. An LLVM-based hybrid binary translation system [C] //Proc of the 7th IEEE Int Symp on Industrial Embedded Systems. Piscataway, NJ: IEEE, 2012: 229-236

[6] Lyu Yihong, Hong Dingyong, Wu Taiyi, et al. DBILL: An efficient and retargetable dynamic binary instrumentation framework using LLVM backend [C] //Proc of the 10th ACM SIGPLAN/SIGOPS Int Conf on Virtual Execution Environments(VEE). New York: ACM, 2014: 141-152

[7] Zhang Xiaochun, Guo Qi, Chen Yunji, et al. HERMES: A fast cross-ISA binary translator with post-optimization [C] //Proc of the 13th Annual IEEE/ACM Int Symp on Code Generation and Optimization (CGO). New York: ACM, 2015: 246-256

[8] Jeffery A. Using the LLVM compiler infrastructure for optimised, asynchronous dynamic translation in Qemu [D]. Adelaide, Australia: the University of Adelaide, 2009

[9] Chipounov V, Candea G. Dynamically translating x86 to LLVM using QEMU, EPFL-TR-149975 [R]. Lausanne, Switzerland: Swiss Federal Institute of Technology in Lausanne, 2010

[10] 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）

[11] Filipe de A G，Fernanda L K，Jose E P S，et al. Soft error injection methodology based on QEMU software platform [C] //Proc of the 15th Latin American Test Workshop (LATW). Piscataway，NJ：IEEE，2014：1-5

[12] Shao Yuanhua. Research and implementation of instruction optimization technique based on QEMU emulator [D]. Chengdu：University of Electronic Science and Technology，2013 (in Chinese)
（邵院华．基于 QEMU 仿真器的指令优化技术的研究与实现[D]．成都：电子科技大学，2013）

[13] Wu Weifeng. Research on completeness of static binary translation and analysis of code [D]. Zhengzhou：PLA Information Engineering University，2012 (in Chinese)
（吴伟峰．静态二进制翻译完备性及代码分析研究[D]．郑州：解放军信息工程大学，2012）

[14] Payer M，Gross T R. Generating low overhead dynamic binary translators [C] //Proc of the 3rd Annual Haifa Experimental Systems Conf. New York：ACM，2010：1-14

[15] Guha A，Hazelwood K，Soffa M L. Memory optimization of dynamic binary translators for embedded systems [J]. ACM Trans on Architecture and Code Optimization，2012，9(3)：1-29

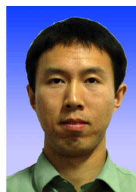
[16] Li Zhanhui，Liu Chang，Meng Jianyi，et al. Cache load balancing oriented dynamic binary translation [J]. Journal of Computer Research and Development，2015，52(9)：2105-2113 (in Chinese)
（李战辉，刘畅，孟建熠，等．基于高速缓存负荷均衡的动态二进制翻译研究[J]．计算机研究与发展，2015，52(9)：2105-2113）

[17] Wang Wenwen，Wu Chenggang，Bai Tongxin，et al. A pattern translation method for flags in binary translation [J]. Journal of Computer Research and Development，2014，51(10)：2336-2347 (in Chinese)
（王文文，武成岗，白童心，等．二进制翻译中标志位的模式化翻译方法[J]．计算机研究与发展，2014，51(10)：2336-2347）

Tan Jie，born in 1991. PhD candidate. Her main research interests include binary translation and high performance computing.

Pang Jianmin，born in 1964. PhD，professor，PhD supervisor. Senior member of CCF. His main research interests include high performance computing and information security (jianmin_pang@hotmail.com).

Shan Zheng，born in 1977. PhD，associate professor. Senior member of CCF. His main research interests include high performance computing and information security (zzzhengming@163.com).



Yue Feng，born in 1985. PhD. Student member of CCF. His main research interests include dynamic compiling and system virtualization (firstchoiceyf@163.com).

Lu Shuaibing，born in 1990. Master. His main research interests include binary translation and high performance computing (yeaxxx@163.com).

Dai Tao，born in 1990. Master. His main research interests include software reverse engineering (daitworld@126.com).