ISSN 1000-9825, CODEN RUXUEW
Journal of Software, Vol.18, No.7, July 2007, pp.1603−1611
DOI: 10.1360/jos181603

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel/Fax: +86-10-62562563

基于动态反馈的标志位线性分析算法*

唐 锋[1,2+], 武成岗[1], 冯晓兵[1], 张兆庆[1]

[1](中国科学院 计算技术研究所 先进编译实验室,北京 100080)

[2](中国科学院 研究生院,北京 100049)

EfLA Algorithm Based on Dynamic Feedback

TANG Feng[1,2+], WU Cheng-Gang[1], FENG Xiao-Bing[1], ZHANG Zhao-Qing[1]

[1](Advanced Compiler Technology Laboratory, Institute of Computing Technology, The Chinese Academy of Sciences, Beijing 100080, China)

[2](Graduate University, The Chinese Academy of Sciences, Beijing 100049, China)

+ Corresponding author: Phn: +86-21-23072493, Fax: +86-21-51534159, E-mail: tf@ict.ac.cn, http://www.ict.ac.cn

Tang F, Wu CG, Feng XB, Zhang ZQ. EfLA algorithm based on dynamic feedback. *Journal of Software*, 2007,18(7):1603−1611. http://www.jos.org.cn/1000-9825/18/1603.htm

Abstract: Binary translation is applied for the legacy code porting. Binary code can be executed in different hardware platforms by binary translation. If the source platform uses condition code to change the execution flow, it is an important performance issue to handle the condition code translation. This paper presents the algorithm of Eflag linear analysis. The complexity of the algorithm is linear and the algorithm reduces much of the flag computing and increases the performance of the dynamic execution. Through dynamic profiling, the algorithm solves to eliminate the Eflag calculation in the basic indirect jump block. Some integer test cases are analyzed in spec 2000. The experimental results prove the efficiency of the EfLA (Eflag linear analysis) for large calculation program.

Key words: dynamic optimizing; binary translation; condition code optimizing; dynamic feedback; linear analysis

摘 要: 二进制翻译可以用于解决遗产代码的迁移问题,也可以实现不同硬件平台之间软件的通用.如果源平台通过标志位进行条件跳转,那么如何处理标志位就成为翻译中的一个重要问题,对翻译的代码质量起着决定性作用.提出标志位线性分析算法,复杂度为线性,基本上能够消除所有的标志位冗余计算,提高了动态执行的效率.基于动态profiling技术,消除了间接跳转的基本块标志位冗余计算.分析了spec 2000中的大部分整点测试例子,实验结果表明,EfLA(Eflag linear analysis)算法对于大运算量的程序是非常有效的.

关键词: 动态优化;二进制翻译;标志位优化;动态反馈;线性分析

中图法分类号: TP314 文献标识码: A

1 引 言

1.1 二进制翻译

新开发的微处理器须有丰富的软件支持,才能得以推广和应用.若一个处理器未能赢得足够的市场份额,那

* Supported by the National Natural Science Foundation of China under Grant No.60403017 (国家自然科学基金)
Received 2006-04-17; Accepted 2006-08-21

么软件厂商一般不会去冒险为它开发新的应用软件.上述两个方面互相制约,限制了处理器的革新.二进制翻译是将原有软件迁移到新处理器上的一种有效手段,为新处理器和原有的软件资源之间搭建了一座联系的桥梁.

二进制翻译作为代码迁移的重要方法,得到了广泛重视.从 20 世纪 80 年代第一个二进制翻译系统出现至今,在此领域已经取得了许多研究成果,并相继研制出实验性和商用的系统.1987 年,HP 公司开发了最早的一个商用二进制翻译系统,将 HP3000 的客户转移到新的 PA 体系结构上.从 1992 年开始,DEC 公司开发了一系列二进制翻译器,用来将 VAX/VMS,MIPS/Unix,Sparc/Unix 以及 x86/WinNT 上的代码翻译到他们新开发的 Alpha 机器上,其中以 FX!32[1]最为有代表性.1996 年,IBM 公司开发的 Daisy[2]是利用二进制翻译调度 PowerPC 代码到超长指令字(VLIW)处理器,增加并行性.1999 年,IBM 公司开发 BOA[3]系统,动态地翻译了 PowerPC 的整个系统,用简单指令实现原语义,简化硬件.2000 年,Transmeta 公司发布了 Crusoe 处理器芯片和动态翻译代码软件 Code morphing[4,5],用来在 VLIW(very long instruction word)内核的硬件平台上运行 x86 代码,甚至包括操作系统.2003 年,Intel 公司开发了 IA32(Intel architecture 32)执行层软件 IA32 EL[6],通过软件方法在 IA64 机器上运行 IA32 的应用程序.2004 年, Transitive 推出了多平台应用程序级的翻译 QuickTransit[7].

1.2 二进制翻译的性能需求

效率对于二进制翻译系统而言是非常关键的,是衡量翻译器质量除了正确性之外最重要的指标,因为如果被翻译程序在新的目标平台上的运行速度比在源平台上下降很多,那么最终该产品会被用户所抛弃.现在,个人计算机市场基本上是被 x86 体系结构所占据,而服务器市场 x86 也已经占据了大部分的市场份额,在 x86 体系结构下有着丰富的应用软件.因此,研究以 x86 为源平台的二进制翻译具有很重要的现实意义.

像 Intel 和 AMD 的 x86 系列、DEC 的 VAX 以及 Motorola 的 680x0 系列处理器等均采用复杂指令集体系结构(CISC),内部设有专用的标志寄存器以记录标志位.一部分机器指令会影响这些标志位,即对标志位定值.同时,有一部分指令会根据标志位执行不同的语义,即对标志位进行引用.精简指令集(reduced instruction set computing,简称 RISC)体系结构(例如 MIPS,Alpha)上一般没有相应的标志寄存器.如果要实现由 CISC(complex instruction set computing)到 RISC 的二进制翻译,则需要通过一系列指令来模拟源 CISC ISA(instruction set architecture)的标志位寄存器[6].

在拥有标志寄存器的处理器中,绝大多数的运算指令都会对标志位定值,虽然这些定值并不一定会使用到.如果对源指令中标志位的每次定值都进行翻译,那么所产生的目标代码的效率就会大幅度下降.平均而言,在翻译所产生的代码中,对每一个标志位的定值所需要的代码与翻译一条指令基本功能的代码数量差不多,而每条指令都会对多个标志位进行修改,那么,如果完全把标志位定值翻译出来,而这些定值并没有被使用,那么浪费就显而易见了.所以,对于标志位进行定值引用分析,删除冗余计算,对提高翻译系统性能是非常重要的.

1.3 标志位处理对性能的影响

由前文可知,标志位对翻译性能具有重大影响,下文将以具体实例说明标志位处理对提升性能的帮助.

以 x86 处理器为例[8,9],它有一个特殊的寄存器 EFLAG,由 6 个状态标志位、一个控制标志位以及若干系统标志位组成.其中最常用的状态标志位是 CF,PF,AF,ZF,SF,OF.AF 和 PF 用于早期的 8 位 x86 指令,现在对其几乎不再引用,所以本节对标志位的说明只针对 OF,SF,ZF,CF 这 4 个标志位.

以下考虑 x86 的一个基本块 *B*0(以这种类型结束的基本块在 spec 2000 的测试例子中占了绝大部分),两个后继分别为 *B*1,*B*2:

```
例 1:B0: sub ecx,edx              /*ecx=ecx−edx,并根据计算结果置标志位*/
         je x86_target_label      /*如果 zf=1,则跳转成功,控制转向目标*/
    B1: add eax,eax
        …
    x86_target_label:
    B2: sub eax,ebx
        …
```

在翻译过程中,翻译第 1 条算数指令 sub *ecx*,*edx* 总共用了 30 条目标机器指令,而其中真正翻译减法行为的

只有 2 条,其余 28 条都是为了翻译标志位的变化.虽然 *je* 用到了 *zf*,翻译 *zf* 用了 4 条指令,如果后继的基本块都不再使用此次定值的标志位,那么,其余 3 个标志位的翻译都是冗余的.

令总共翻译指令条数为 *Total_Instr_Num*,真正用来翻译语意的指令条数为 *Core_Instr_Num*,翻译后面会用到的标志位的定值的指令条数为 *Used_Flag_Instr_Num*,那么,指令翻译的冗余率为

$$Redu=(Total\_Instr\_Num-Core\_Instr\_Num-Used\_Flag\_Instr\_Num)/Total\_Instr\_Num.$$

应用上述公式,对于 sub 这条指令的翻译冗余率为(30−2−4)/30=80%.

本文提出的标志位线性分析算法 EfLA(Eflag linear analysis)在算法复杂度为线性的前提下,消除了绝大多数的标志位冗余赋值.EfLA 算法适用于所有 CISC 机器需要标志位模拟的系统,特别适合需要实时分析的系统.

本文第 2 节介绍相关工作.第 3 节给出标志位线性分析优化算法.第 4 节给出实验数据以及分析结果.第 5 节给出结论以及未来工作.

2 相关工作

二进制翻译分为动态、静态和动静结合 3 种模式.静态翻译系统对于标志位的处理,只需考虑标志位的处理效果,即冗余计算的消除率,而无须考虑标志位处理本身的开销.而动态翻译系统,对上面两方面需要综合考虑,寻找平衡点.

Fx!32 是一个动静结合的翻译系统,结合了解释执行和静态翻译两种模式.对于静态翻译,采用了数据流分析的方法,基本上可以完全消除标志位的冗余计算,但是数据流分析本身开销比较大.不过,这个过程是在静态时进行,不会影响到运行时性能.动态执行需要解释器支持,解释器采用了延迟计算的方法,即对于每条指令把需要用到计算标志位的信息(包括操作数、操作码等信息)先存储起来,而不是马上计算标志位,等到需要使用标志位的时候再通过原先存储的标志位信息计算标志位.

Queensland 大学开发的 UQBT(the University of Queensland binary translator)[10]多源到多目标的静态翻译器在 ISA(instruction set architecture)相关的底层中间表示转化到与 ISA 无关的高层中间表示的过程中,通过数据流分析去除冗余定值.对于引用标志位,则按照一定模式转化为相应的中间表示.虽然这种方式翻译出来的代码质量较高,但是不具有广泛的适应性,只能对高级语言编译下来、没有经过优化的代码才有效,对于手写汇编或者经过优化模式被打乱之后的代码就无能为力.

Intel 的 IA32 EL[6]构建不完全数据流图,一般包含 1~20 个基本块,在不完全流图上进行数据流分析.流图内部基本可以消除冗余计算,但到基本块边界就会产生冗余计算.部分流图在对数据流进行分析时,开销会比完全流图的分析小,但是个别情况效果会略差.例如边界情况无法分析,就有可能比完全流图产生更多的冗余计算.

Transmeta 公司开发的 Crusoe 芯片[4,5]采用了硬件支持标志位的方法,即它也设置了一个类似 x86 的标志寄存器,可以把每个指令的标志位影响记录下来.该系统采用硬件实现标志位计算,可以实时地把每一条指令对标志位的影响都记录下来而不会影响执行速度.

中国科学院计算技术研究所开发的 Digital Bridge 第 1 版中,采用了及时计算与延迟计算相结合(instant computing and delayed computing,简称 ICDC)算法以及数据流分析和延迟计算相结合(data flow and delayed computing,简称 DFADC)算法[11].这些算法本身开销很小,因为只对基本块内部进行分析,基本块间采用延迟计算.因基本块间没有分析,所以虽然没有导致冗余计算,但却导致了冗余存储.EfLA 算法受到了 DFADC 算法的影响,基本块内分析也采用了下次使用信息[12]算法,但是对于基本块间标志位的处理还是有非常大的区别的.文献[11]中采用了延迟计算,而 EfLA 算法则分析后继基本块,化基本块间标志位关系为基本块内标志位关系.后文会给出两者详细的对比分析.同时,基于动态反馈的 EfLA 算法很好地消除了间接跳转基本块的标志位冗余计算.

3 EfLA 算法

3.1 概 述

标志位分析的理想目标就是把所有的无用定值全都找出来,而后在翻译指令的时候,不必理会这些无用定

值,仅对活跃定值进行翻译.

如果以一个基本块为分析单位,那么在基本块内部,可以通过线性扫描发现基本块内的无用定值,但是到了基本块的边界,如果不进行基本块间的数据流分析,就很难发现边界的无用定值.为了处理基本块的边界,需要分析该基本块的后继对标志位的影响,然后根据后继对标志位的影响,决定对边界的处理.例如,在一个基本块的边界最后定值某一个标志位,假设该基本块只有两个后继,在每个后继中对该标志位的引用之前都存在定值,这样就可以判断该基本块中最后对该标志位的定值是一个无用定值.

如果一个基本块对标志位没有影响,只要它的所有后继都对标志位定值,那么其效果也等同于该基本块对标志位定值.假设基本块 $b0$ 有 n 个后继,每个后继也都有 n 个后继,如果总共有 i 层后继,$b0$ 为第 0 层,那么总共有 $1+n+n^2+\ldots+n^i=(n^{i+1}-1)/(n-1)$个基本块.在最坏的情况下,只有第 i 层后继才对标志位影响,其余 $b0$ 以及第 1 层~第 $i-1$ 层后继都没有对标志位影响,那么,需要遍历所有的基本块才能发现 $d0$ 中的定值为无用定值.最坏情况下,随着后继层数的增大,遍历基本块的复杂度是呈指数级的.但是,根据程序的实际情况,EfLA 算法作一定简化之后,就能保持线性的复杂度,并且能够消除绝大多数的标志位冗余计算.简化基于这样一个事实:对于标志位的定值如果没有分析出来是一个无用定值,那就多了一次冗余计算,但是不会影响正确性.对于简化之后算法分析无用定值的效率,后文结合实际程序将会作相关说明.

基本块内可以使用线性算法分析标志位定值引用关系,一次扫描就可以得出标志位的活跃注销信息,开销非常小.对于边界的处理,通常的方法是使用全局数据流分析,但是开销太大,可以把所有后继基本块对标志位的影响转化成一条虚拟指令附加在当前基本块中,该虚拟指令只对分析标志位定值引用有效.

3.2 标志位线性分析算法

设置一个数据结构,用于记录源机器的每条指令 I 对标志位的影响,其形式如下:

def 域,*use* 域以及 *status* 域.

其中,*def* 标示 I 是否对标志位定值,*use* 标示 I 是否对标志位引用,*status* 标示 I 对该标志位 *def* 是无用或者是活跃.通过 EfLA 算法,能够得到当前指令标志位的 *def* 是无用还是活跃,翻译时以此为依据,只翻译活跃的定值.

EfLA 算法以一个基本块为处理单元,主要有以下 4 个步骤:

Step1:分别计算后继基本块对标志位的影响关系,产生后继基本块的标志位影响虚拟节点.

Step2:计算所有后继对当前基本块标志位的影响关系,为当前基本块添加虚拟指令节点.

Step3:初始化最后一条指令(虚拟指令)的 *status*(活跃信息),表明该指令标志位如果存在定值则是活跃的.

Step4:基本块内标志位定值引用关系线性分析.

3.2.1 计算后继基本块对标志位的影响

```
对于每一个标志位
{/*第 1 层后继*/
    标志位影响节点=该基本块中第 1 条对该标志位的影响(use 或者 def 或者 use&&def);
    if (该基本块对该标志位没有影响){/*根据前文所述,这一步应该递归或者迭代直到发现对标志位有
                                       影响的后继基本块.但是 EfLA 算法作了简化,只寻找到第 2 层
                                       后继就停止*/
    for (所有后继){/*第 2 层后继*/
        后继标志位影响节点=该基本块中第 1 条对该标志位的影响(use 或者 def 或者 use&&def);
    } end_for
    计算所有后继对当前基本块标志位的影响;/*算法如第 3.2.2 节所示,这是为了通过第 2 层后继计算对
                                            第 1 层后继的标志位的影响*/
    } end_if
}
```

上述算法是为了计算当前第 1 层后继基本块对标志位的影响,如果该后继基本块对标志位没有影响,那么

就需要计算后继的后继(即第 2 层后继)基本块对标志位的影响,然后把所有第 2 层后继基本块对标志位的影响附加到第 1 层后继基本块.EfLA 算法只遍历第 2 层后继就结束,如果还没有发现对标志位的影响,就只能保守地认为对标志位的影响为 *use*.EfLA 算法并没有寻找所有后继直到找到有影响的基本块,这是为了算法效果和算法开销的平衡,在后文实验数据部分我们给出详细的分析.

应当注意,当流图中出现了环,并且正好所处的环路径上对标志位没有影响时,EfLA 算法不必对其特殊处理;但是,如果算法采用递归分析所有后继基本块对标志位的影响,为了让递归能够收敛,一旦当一个基本块被第 2 次遍历到,并且该基本块还是没有对标志位有影响时,就终止该层递归,认为该基本快标志位进行了引用.

3.2.2 计算所有后继对当前基本块标志位的影响

对于每一个标志位{

temp_flag_affect.use=0;

temp_flag_affect.def=1;

for(所有后继){

/*所有的后继都对定值注销,虚拟节点才能具有定值的性质.

只要后继中有一个引用,虚拟节点就必须具备引用的性质*/

temp_flag_affect.use|=后继虚拟指令标志位.*use*

temp_flag_affect.def &=后继虚拟指令标志位.*def*

}

把 *temp_flag_affect* 附加到当前基本块的虚拟节点上;

}

3.2.3 基本块内标志位定值引用关系线性分析

经过前面几个步骤,已经把虚拟指令节点附加到了当前基本块上,把基本块间的标志位分析转化为基本块内的标志位分析.采用下次引用信息算法[12]处理基本块内部的标志位分析.算法伪码如下:

```
从后往前扫描每一条指令(最后一条为标志位虚拟指令)
{
    对于每一个标志位
    {
        当前指令标志位的 status=后一条指令标志位的 status;
        if (这条指令定值标志位)
            当前指令标志位的 status=0; /*无用定值*/
        if (这条指令使用标志位)
            当前指令标志位的 status=1; /*定值活跃*/
    }
}
```

3.3 基于动态反馈的EfLA算法

如前所述,EfLA 算法需要确切知道基本块后继的个数,对于间接调用,这个条件是较难满足的.文献[13]中提出了可以把间接调用提升为 switch case 语句,这样,一部分间接调用的后继就可以得到,利用 EfLA 算法就可以把冗余标志位计算消除.但是对于无法提升的间接调用,我们在利用 EfLA 算法时,只能进行保守处理,因此有可能会导致冗余的标志位计算.假设 *A* 的一个后继为 *B*,*B* 是以间接跳转来结束的基本块,那么只要 *B* 对标志位有影响,对 *A* 进行标志位线性分析的时候就不会产生冗余计算.只有当 *B* 对标志位没有影响时,使用 EfLA 算法才有可能导致 *A* 中标志位的冗余计算.对标志位没有影响的基本块基本上都是与 *plt* 表相关的,在 x86 中,主要是为了库函数调用所需要的,因为其前躯是以 call 结束的基本块,所以,根据标志位不会跨越函数使用的原则,也不会产生多余的标志位计算.我们将在实验结果中分析这类情况.

还有一类情况是 *A* 本身就是一个以间接跳转结束的基本块,这样,我们在完全 profiling 之前就很难确切得知后继个数,如果仍然使用上述 EfLA 算法,就只能作保守处理,冗余的消除效果不是很好.因此,我们可以对其进

行变换,加入标志位处理 stub.假设有 n 个标志位处理,就把所有的后继按照对标志位的影响分为 2^n 种类型:对 0 个标志位定值 C_n^0,对 1 个标志位定值 C_n^1,…,对 n 个标志位定值 C_n^n.因此,需要加入 2^n 个 stub,分别处理所有的标志位处理类型.因为翻译代码在执行之前,所以对于 A 的某一个后继 A_i 的第 1 次执行,是默认为对 0 个标志位定值;但是执行到 A_i 之后,我们就可以对 A_i 分析,得出它对标志位的影响情况,反馈给 A 被翻译的代码,等到第 2 次执行从 $A \rightarrow A_i$ 的时候,就是根据 A_i 真正对标志位的影响计算标志位了.

处理间接跳转 A 的基于动态反馈的算法如下:

(1) 初始化一张空表 succ2stub(该表需要以后根据动态执行反馈回填),表项为后继地址以及不同标志位类型 stub 入口地址.stub 的作用就是计算相应的标志位以及跳转到相应的后继.

(2) 在基本块的最后插入查询 succ2stub 的代码,如果该后继在 succ2stub 中,则执行相应的 stub 代码,计算相应的标志位;如果该后继不在 succ2stub 中,则进入保守处理 stub,计算所有标志位.

(3) 执行翻译好的 A 的代码之后,如果后继 A_i 不在 succ2stub 中,则运用第 3.2.1 节中所提到的算法得到 A_i 对标志位的影响,根据 A_i 对标志位的影响,填写 succ2stub 表.

运用上述算法之后,对于间接调用的基本块,其后继的第 1 次执行可能会存在冗余的标志位计算.但是通过动态反馈,在第 2 次以后从该基本块到该后继的执行就不会存在冗余的标志位计算问题了.

3.4 算法效果比较

3.4.1 与 FX!32[1]和 IA32 EL[6]比较

FX!32 所采用的静态全局数据流分析或者 IA32 EL 所采用的局部数据流分析虽然能够消除绝大部分的冗余计算,但都还有一小部分无法消除,因为在数据流分析中,对于边界的情况只能做保守的处理,认为边界上的值是活跃的.在 FX!32 中,因为静态无法得知后继,所以间接跳转的边界部分的标志位计算如果存在冗余是没有办法消除的.对于 IA32 EL 的局部数据流分析,流图的边界也没有进行处理.

我们考虑例 2,$B0$ 以间接跳转结束,它的所有后继为 $B0$,$B1$,$B2$,$B3$:

例 2:$B0$: sub *ecx*,*ebx*
Jmp**ebx*
$B1$: add *ecx*,*edx*
…
$B2$: cmp *ecx*,*ebx*
…
$B3$: neg *eax*
…

FX!32 所采用的静态数据流分析,或者 IA32 EL 所使用的局部数据流分析,对于 $B0$ 的边界都无法处理.FX!32 通过静态数据流分析是根本无法得知 $B0$ 的后继是哪些基本块的,而对于 IA32 EL,遇到间接跳转,就划分为流图的边界,不再考虑后继对当前基本块的影响,所以它们对 $B0$ 进行保守处理,认为标志位在离开 $B0$ 之后仍然是活跃的.由例 1 可知,翻译 sub 指令使用了 30 条指令,其中用于标志位处理的为 28 条.根据对后继基本块的分析,28 条标志位的处理完全都是冗余计算.应用 FX!32 或者 IA32 EL 所采用的算法,翻译 sub 指令的冗余计算率为 28/30=93%;而使用基于动态反馈的 EfLA 算法,在后继第 1 次执行的时候,比 FX!32 或者 IA32EL 的算法多了 10 条查表的开销,在第 2 次执行之前分析得到后继基本块都在 *use* 标志位之前已经对标志位 *def*,真正翻译 sub 指令的时候只使用了 2 条指令,再加上查表的开销,总共只需 12 条指令.所以,如果后继能够执行到第 2 遍,那么其总开销就会低于 FX!32 或者 IA32 EL 所采用的算法.根据我们后面的实验结果作统计分析,这样的条件总是能够满足的.

为了量化地进行比较,我们假设计算所有的标志位需要 n 条指令,查表的开销需要 m 条指令,后继 B_i 的执行次数为 b_i,假设 B_i 能够把所有的标志位计算都注销,那么,使用基于反馈的 EfLA 算法较之静态数据流分析能够节省的指令条数是 $n \times b_i - (m \times b_i + n)$,因此,只要 $b_i >= n/(n-m)$,就能保证 B_i 使用 EfLA 算法优于静态数据流分析.实际上, $n/(n-m)$的值很小,一般小于 2.所以一般情况下,若某一个后继执行次数大于 2,就值得使用 EfLA 算法.

由此可见,EfLA 算法在处理流图边界情况时,只要后继的执行次数大于 2,就能优于数据流分析或者局部数据流分析.当然,在流图内部,所有算法的处理效果都是相同的,都能够把所有的冗余计算消除.

3.4.2 与 ICDC 和 DFADC[11]比较

在文献[11]中使用的方法,基本块间使用延迟计算,每个基本块末尾需要多条指令存储计算标志位的信息,虽然比计算标志位开销小,但这些存储也已是很大的开销了.

对于例 2 而言,除了翻译 sub 需用 2 条指令以外,还需要以下的指令存储计算标志位的源料:

```
sw    m_ecx,eflag_material_src1        /*存储第 1 个源操作数*/
sw    m_ebx,eflag_material_src2        /*存储第 2 个源操作数*/
sw    result, efalg_material_dest      /*存储结果操作数*/
addi temp,r0,opcode                    /*opcode 是翻译为可知的一个常量,放入 temp 寄存器中,因为 RISC
                                         机器没有直接往内存中存储立即数的指令,所以需要 temp 中转*/
sw    temp,eflag_material_opcode       /*存储该指令操作码*/
addi temp,r0,instruction_size          /*instruction_size 是指该指令操作数位数的属性,也是翻译时可知的
                                         一个常量*/
sw    temp,eflag_material_size         /*存储指令大小的属性*/
```

虽然把计算标志位的材料存了起来,但是后继的基本块都没有使用,所以这些存储都是冗余的.而且使用延迟计算,每次使用标志位之前需要判断标志位是需要重新计算还是可以直接使用,这也增加了不少开销.

而 EfLA 算法进行了基本块间的数据流分析,消除了全部存储冗余,使性能得到提升.EfLA 算法本身的开销会因为计算后继基本块标志位的使用情况而高出 DFADC 算法,但是生成的本地码质量高于后者所产生的本地码质量,整体的性能还是高于 DFADC 算法,后文我们会给出具体的实验数据,以证明 EfLA 的整体性能优于 DFADC 算法.

4 实验数据分析

4.1 基于动态反馈的EfLA实验数据

数据流分析,需要建立控制流图.如果通过迭代解数据流方程,则最坏算法复杂度为 $O(n^2)$,n 为节点数目.而 EfLA 算法是线性的,所以算法本身的开销小于 FX!32 所采用的非线性的数据流分析以及 IA32 EL 所采用的非线性局部数据流分析.表 1 中给出了 spec 2000 test 测试集中的一些测试用例关于间接跳转的数据.

Table 1 Information for indirect jump block

表 **1** 间接跳转基本块信息

	plt	*plt_exec*	*jmp_1*	*jmp_2*	*jmp_3*	*jmp_4*	*jmp_5*	*ind_num*	*ind_exec_num*
bzip2	34	399	17	1	3	0	0	2	355
gzip	38	56 630	16	1	1	1	0	2	355
mcf	39	8 006	16	2	1	0	0	2	362
parser	19	2 065 020	16	1	1	0	0	3	368
crafty	59	7608	16	2	2	0	2	105	9 639 882
twolf	56	28 016	17	1	1	0	0	2	362
vortex	66	5 311 422	47	3	1	3	0	65	2 503 467
vpr	66	595 760	33	3	2	0	0	26	118 710
gap	54	1 275 593	19	1	1	1	0	34	429 209

我们统计了对标志位没有影响的基本块,发现都是 *plt* 中的基本块,根据第 3.3 节中的分析,*plt* 表中的基本块对于标志位计算可以特殊处理,不会因为对标志位没有影响而使前驱产生冗余计算.在表 1 中,*plt* 和 *plt_exec* 分别是 *plt*(对标志位没有影响)的基本块数目和这些基本块总共执行的次数.

jmp_x(x=1,…,5)是指间接跳转的基本块执行某一个后继 x 次,*ind_num* 是指间接跳转的基本块执行某一个后继的执行次数多于 5 次的基本块数目,*ind_exec_num* 是指后继执行次数大于 5 的所有间接跳转总共执行次数.根据第 3.4.1 节中的分析,绝大多数间接跳转的基本块执行某一个后继的次数多于 2 次,即使我们把这个域值扩

大到 5(因为考虑到并不是所有后继都能把间接调用的基本块的标志位计算注销,所以我们把域值扩大到 5),对于 spec test 测试集而言,使用动态反馈的 EfLA 算法比静态所采用的全局数据流分析或者 IA32 EL 所采用的局部数据流分析能节省更多的机器指令.

4.2 EfLA算法实验数据

文献[11]提出的方法在基本块内部分析,开销只有线性 $O(n)$,n 为基本块指令条数.EfLA 比文献[11]中的算法多了分析后继基本块对标志位的开销.根据分析基本块后继的层数,开销和算法的效果会有不同的变化,下面给出 3 种情况下 spec cpu 2000 测试例子删除冗余标志位计算的不同效果.表 2 给出了 test 测试集下各个程序执行的基本块数目(*b_num*)、对标志位有定值的基本块数目(*b_def_num*)、在基本块的结束使用标志位的数目 *use_num*、分析一层后继冗余计算标志位的数目(*R*1)、分析一层后继,如果该后继对标志位没有影响且该后继只有一个后继,则冗余计算标志位数目(*R*2)、分析两层后继冗余计算标志位数目(*R*3).因为基本块对标志位的定值一般都是对 6 个标志位定值,所以我们可以这样计算基本块间的标志位冗余计算:(*Rx*/*b_def_num*×6).

Table 2 The effect for the depth of successors

表 **2** 后继层数对标志位的影响

	b_num	*b_def_num*	*use_num*	*R*1	*R*2	*R*3
bzip2	2 712	1 925	2 276	2 102	160	4
gzip	2 746	1 966	2 026	1 941	160	4
mcf	2 889	2 041	2 052	2 057	172	4
parser	7 435	4 844	4 407	7 103	180	6
crafty	7 534	5 608	5 818	5 875	294	120
twolf	7 152	5 159	5 925	7 512	174	6
vortex	14 133	9 052	8 665	4 305	198	30
vpr	6 201	4 185	5 131	2 731	211	43
gap	10 802	7 601	6 515	11 479	198	30

从表 2 可以得出,经过一层后继分析,有 17.95%左右的基本块边界冗余标志位计算;分析两层后继的最坏复杂度有可能达到 $O(m^2)$,m 为后继的个数;如果只对一个后继的后继进行分析,那么最坏复杂度仍然是 $O(m)$.但是,通过这样的线性分析,可以降低到 0.87%左右的冗余标志位计算.因为在实际程序中,绝大多数的后继个数为 1~2 个,所以分析两层后继,一般情况下也就是分析 4 个基本块,把冗余计算控制在 0.088%以内.所以 EfLA 算法中采用了这种策略.如果为了完全消除冗余计算,即使一般后继只有 2 个,但是遍历 n 层后继之后,算法复杂度将变为 $O(2^n)$,两相权衡之下,应当放弃递归遍历所有后继基本块的算法.

对于文献[11]提到的算法,因为基本块间采用延迟计算,所以每个基本块就会增加 7 条冗余的存储指令,对于所有的基本块也就增加了 7×*b_def_num* 指令,而采用 EfLA 算法分析两层后继只需比其增加少量开销,就能把这些冗余基本消除.表 3 给出了 EfLA 算法和 DFADC 算法的程序执行时间比较.

Table 3 DFADC Vs EfLA

表 **3** DFADC Vs EfLA

	DFADC (s)	EfLA (s)	Speed up (%)
bzip2	359.64	343.77	4.41
gzip	106.13	102.65	3.28
crafty	346.7	317.22	8.50
mcf	5.75	5.4	6.09
parser	152.65	142.81	6.45
twolf	38.61	37.5	2.87
vortex	858.05	767.76	10.52
vpr	3 008.17	2 982.27	0.86
gap	94.65	85.79	9.36

由表 3 可以看出,EfLA 算法使程序执行速度平均提高了 5.82%,这是 spec cpu 2000 test 的测试集;当使用 ref 的测试集时,平均性能提高在 15%左右.ref 比 test 性能提高较多的原因是 ref 更多的时间是在执行本地码,它的执行速度对代码翻译质量的依赖性更强,EfLA 算法本身虽然比 DFADC 开销大一点,但是,它指导翻译生成的代码效率比 DFADC 高,所以对于 spec 这类大运算量的程序而言,最终执行开销还是 EfLA 算法明显优于 DFADC.

5 总 结

本文提出了标志位线性分析算法,简化了基本块间的标志位分析,采取分析后继的方法减小了算法开销.尤其是对于每条算术指令都能对标志位定值的 x86 这类 CISC 体系结构,基本能把标志位冗余计算消除,从而提高了翻译代码的质量.统计结果表明,这种简化是切实可行的,是具有普遍适用性的,可以在二进制翻译或者模拟器中有效地加快运行速度.

在实际应用中,特别是对高级语言编译的程序,对标志位的引用都使用固定的模式,模式匹配和标志位处理联合起来,将会使翻译的效果更加高效.今后的工作就是要把这两者有机地结合起来,互相补充.

致谢 在此,我们向对本文的工作给予支持和建议的同行,尤其是中国科学院计算技术研究所先进编译室二进制翻译组的同仁表示感谢.

References:

[1] Chernoff A, Herdeg M, Hookway R, Reeve C, Rubin N, Tye T, Yadavalli SB, Yates J. FX!32: A profile-directed binary translator. IEEE Micro, 1998,18(2):56−64.

[2] Ebcioglu K, Altman K. Dynamic binary translation and optimization. IEEE Trans. on Computers, 2001,50(6):529−548.

[3] Gschwind M, Altman ER, Sathaye S, Ledak P, Appenzeller D. Dynamic and transparent binary translation. Computer, 2000,33(3):54−59.

[4] Halfhill R. Transmeta breaks x86 low-power barrier. Microprocessor Report, 2000,14(2).

[5] Klaiber A. The technology behind Crusoe™ processors. Transmeta Corporation White Paper. 2000.

[6] Baraz L, Devor T, Etzion O, Goldenberg S, Skaletsky A, Wang Y, Zemach Y. IA-32 execution layer: A two-phase dynamic translator designed to support IA-32 applications on Itanium based systems. In: Proc. of the 36th Annual IEEE/ACM Int'l Symp.on Microarchitecture (MICRO-36). San Diego: IEEE-CS Press, 2003.

[7] http://www.transitive.com/

[8] Intel. Intel architecture software developer's manual Vol. 1: Basic architecture. 1999.

[9] Intel. Intel architecture software developer's manual Vol. 2: Instruction set reference. 1999.

[10] Cifuentes C, Van Emmerik M. UQBT: Adaptable binary translation at low cost. Computer, 2000,33(3):60−66.

[11] Ma XN, Wu CG, Tang F, Feng XB, Zhang ZQ. Two condition code opitimization approaches in binary translation. Journal of Computer Research and Development, 2005,42(2):329−337 (in Chinese with English abstract).

[12] Aho AV, Sethi R, Ullman JD. Compilers: Principles, Techinques, and Tools. Addison-Wesley, 1986. 608−623.

[13] Cifuentes C, Van Emmerik M. Recovery of jump table case statements from binary code. In: Proc. of the Int'l Workshop on Program Comprehension. Pittsburgh: IEEE-CS Press, 1999. 192−199.

附中文参考文献:

[11] 马湘宁,武成岗,唐锋,冯晓兵,张兆庆.二进制翻译中的标志位优化技术.计算机研究与发展,2005,42(2):329−337.

唐锋(1979－),男,上海人,博士生,主要研究领域为二进制翻译,编译优化.

冯晓兵(1969－),男,博士,研究员,CCF 高级会员,主要研究领域为并行编译技术,二进制翻译,相关工具环境.

武成岗(1969－),男,博士,副研究员,CCF 高级会员,主要研究领域为二进制翻译,编译优化.

张兆庆(1938－),女,研究员,博士生导师,主要研究领域为编译优化技术.