代码生成阶段的循环不变量外提

杨书鑫 薛丽萍 张兆庆
(中科院计算技术研究所先进编译组 北京100080)

摘 要 循环不变量外提是一种传统的优化算法。在现代编译器中,循环不变量通常在编译器的中端中被删除或外提。中端的中间表示是目标处理机无关的,而编译器的后端的中间表示是目标处理机相关的。尽管中端的优化十分有效,但是从中端的中间表示向后端的中间表示转化的过程中会引入许多循环不变量。因此,有必要在后端再进行循环不变量外提。由于在指令调度的过程能够比较容易地决定一个循环不变量是否需要外提,我们将这一个阶段集成到指令调度中。为了降低指令调度的复杂性,我们把循环不变量的识别和外提区分开来。"识别"独立进行,而决定是否"外提"并实施"外提"则集成到指令调度阶段中。

我们在开放源码编译器ORC的代码生成模块中具体实现了本文所介绍的算法。实验结果显示,在代码生成阶段的循环不变量外提能够提高目标代码1%的性能。我们的代价模型避免了78%的循环不变量不必要地外提到循环之外。

关键词 循环不变量,代码生成,部分冗余删除,指令调度

Loop Invariant Code Motion in Code Generator

YANG Shu-Xin XUE Li-Ping ZHANG Zhao-Qing
(Advaneed Edit Group, Institute of Computer Technology, CAS, Beijing 100080)

Abstract Loop invariant code motion (LICM) is a traditional optimization. In modern compiler, it is normally performed by the phases of middle-end. The immediate representation (IR) of middle-end is machine independent. And it is machine specific in the back-end. The loop invariants are created when the IR are lowered from middle-end to back-end. Hence, we need to perform the LICM in the back-end. We integrate the LICM into the instruction scheduling phase. The reason behind this integration is that it is easy to model the cost of elimination during the phase of scheduling. In order to alleviate the already complex scheduler, we decouple the loop invariant identification from the "code motion" operation. The identification part is a standalone phase and the "code motion" part is integrated into scheduling.

We have implemented the proposed LICM in the code generator of open source compiler ORC. The result shows the LICM can boost the performance by 1% and obviate the needs of unnecessary motions of 78% loop invariants to their enclosing loops.

Keywords Loop invariant, Code generation, Partial redundancy elimination, Instruction scheduling

1 引言

循环不变量外提(loop invariant code motion)是一种传统的优化算法。循环不变量可能是"死代码"或者是完全冗余表达式或者是部分冗余表达式。在现代编译器中,死代码通过死代码删除优化阶段予以删除;完全冗余的表达式可以通过Value numbering[1]优化阶段予以删除;部分冗余的表达式可以通过部分冗余删除[2]阶段予以消除。这3个优化阶段通常都安排在编译器的中端,它们的中间表示通常是SSA[3]形式。

尽管中端的变换已经把大多数的循环不变量都外提到循环之外,但是在编译器后端的代码生成阶段还会有一些需要外提的循环不变量。这是由于:编译器中端的中间表示是目标处理机无关的,而后端的代码生成阶段是直接针对目标处理机的,从中端的中间表示转换成后端的中间表示的过程中会引入循环不变量。

在代码生成阶段进行循环不变量外提与在编译器中端进行循环不变量外提的差异在于:

(1)编译器中端并不刻意把循环不变量外提到循环之外,而是在进行其他优化(例如部分冗余删除)的同时自动地把循环不变量外提到循环之外。代码生成阶段的中间表示一般不是SSA式,不易在这个阶段进行在全局(指整个控制流)上进行中端的一些优化,因此,代码生成阶段的循环不变量外提必须刻意地识别循环不变量,然后决定是否把它外提到循环之外。

(2)如果中间表示是SSA式,优化阶段只需要考虑被编译代码的数据流就可以了,反之优化阶段既要考虑被编译代码的数据流也要考虑它的控制流。

循环不变量外提的代价是寄存器压力增大,也增加了循环之外的代码。实际上,并非所有的循环不变量都值得外提,"外提"反而可能得不偿失。由于利用指令调度已有算法和数据结构能够很方便地判断是否需要"外提",我们把循环不变量外提集成到全局指令调度中。此外,为了减小指令调度的复杂性,我们把循环不变量的识别和外提这个过程分开,识别过程独立进行而外提则合并到全局无环指令调度的时刻。

本文第2节介绍理解本文所必要的背景知识;第3节在文[4]的基础上讨论如何识别循环不变量;第4节论述如何在全

局无环指令调度的同时进行循环不变量外提；第5节报告实验结果；第6节总结并结束全文。

2 背景知识

我们把循环不变量外提集成到全局指令调度过程中。尽管全局指令调度有许多，但它们通常会维护一些相同或类似的量。为了便于说明，我们在这一节中介绍一个调度原型。下文的描述将基于这个原型。事实上，我们并不试图描述全局调度原型，而只描述它一个类似于 list 调度[5,6]的局部调度，作者认为，一个局部指令调度的原型就可以帮助理解本文了。

分别用 $Etime(i)$和 $Ltime(i)$表示指令 i 尽可能早的发射时间和尽可能迟的发射时间。如果 $Etime(i)$和 $Ltime(i)$相等，称指令 i 处在关键路径上。若指令 j 数据依赖于 i，$Latency(i,j)$表示两者之间的依赖延迟。指令 i 的依赖高度 $DH(i)$定义为：

$$DH(i)=\begin{cases} i\text{的执行时间} & \text{(没有指令数据依赖于 } i\text{)} \\ \max(DH(j)+Latency(i,j)) & \text{(否则)} \end{cases}$$
$$j\text{ 数据依赖于 } i$$

如果一条指令所有数据依赖的指令都被调度，这条指令称为就绪指令或候选指令。调度器维护一个拍数(称为当前拍)，其初始值为0。调度实际上是把指令映射为拍数的过程。这个过程是从所有的候选指令中选出 $Etime()$不大于当前拍的候选指令在从中选出优先级最高的一个。如果当前拍不能容纳(由于目标处理机资源所限)任何其它指令或任何指令的 $Etime()$都大于当前拍，当前拍递增1。上述过程重复进行直到所有指令都被调度。

候选指令 i 的优先级用一个序偶(e_1,e_2)表示，$\langle e_1,e_2\rangle=\langle Ltime(i)-Etime(i),DH(i)\rangle$。设指令 i 和 j 的优先级分别是$\langle e_1^i,e_2^i\rangle$和$\langle e_1^j,e_2^j\rangle$，那么指令 i 的优先级比 j 高当且仅当 $e_1^i<e_1^j$ 或 $e_1^i=e_1^j$且 $e_2^i>e_2^j$。

3 识别循环不变量

文[4]已经详细描述了如何识别循环不变量。但是要把文[4]应用到实际场合需要考虑别名(alias)带来的影响。此外，标注存放循环不变量的变量是否需要引入临时变量会给将来的“外提”提供方便。

3.1 别名

别名是由于 store 指令或子函数调用引起的。文[4]不考虑别名的情况，因此不能够直接应用当循环中有 store 和子函数调度用的场合。在文[4]的基础上，考虑别名的影响十分简单，只需要在当文[4]判断一条指令 i 所计算的表达式是循环不变量时，判断如下两点即可：

(1)如果循环中存在子函数调用，且存在一个子函数 c，在 c 中定义 i 的源操作数，那么 i 所计算的表达式不是循环不变量。

(2)如果 i 是取数指令，且在循环中存在一条 store 指令与其别名，那么 i 所计算的表达式不是循环不变量。

上述判断是在①待定为循环不变量的表达式和②循环体内所有的 store 及子函数调用之间进行的。这是因为：如果循环中存在 store /子函数调用的话(设为指令 s)，那么对于循环体中任意一条指令 i，至少存在一条路径 L，在 L 上 s 先于 i 执行。如果 i 和 s 别名，而 i 又以循环不变量外提到循环体之外的话，那么 i 和 s 至少在路径 L 上违反了两者之间的依赖关系。例如，在图1中，尽管块 B_2中的指令 *store* 和 B_3中指令 load 在循环体内部互相不可达，但是在路径 L：$B_1\rightarrow B_2\rightarrow B_4\rightarrow B_1\rightarrow B_3$上 store 先于 load 执行。如果两者别名而 load 又作为循环不变量外提到循环体之外，那么在路径 L 上，它们的数据依赖关系被违反了。

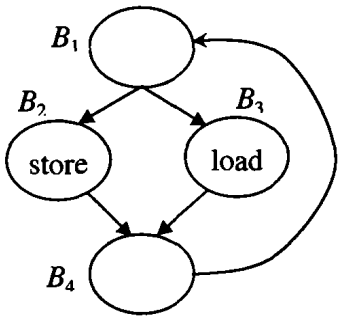

图1

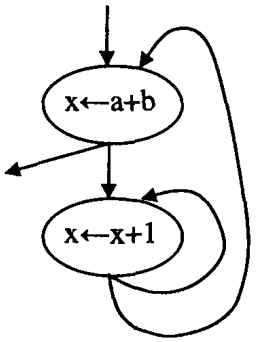

图2

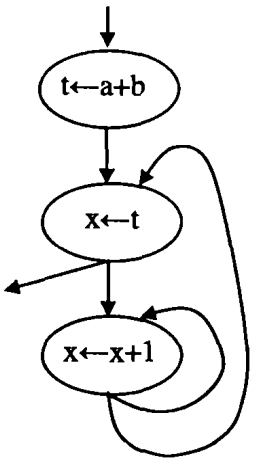

图3

3.2 引入临时变量

在文[4]算法中，每一条计算循环不变量的指令都予以标志。当中间表示不是 SSA 式的情况下，用于存放循环不变量的变量本身可能不是循环不变量。例如在图2中，表达式 $a+b$ 是循环不变量，但是这个循环不变量被存放到一个不是循环不变量的变量 x 上，因此如果要把 $a+b$ 外提，那么需要引入一个临时变量 t，如图3所示。

由于我们把循环不变量的“识别”和“外提”分开，其中“外提”在全局无环指令调度时进行。由于指令调度并不一定在整个循环上进行(它有可能只在循环体对应的控制流的子图上进行)，在指令调度阶段判断一个变量是否是循环不变量并不很容易。所以，在“识别”阶段需要对每一条指令作2个标记，其一标记该指令是否计算循环不变量，另外一个标记目标变量是否是循环不变量。

4 在全局指令调度时外提

循环不变量外提的代价是增加了寄存器压力，延长了循环外的代码的长度。而事实上并非所有的循环不变量都有必要外提。“外提”反而会得不偿失。由于在指令调度时在进行“外提”的代价评估比较方便，我们把外提这个过程集成到指令调度中。

4.1 代价模型

“外提”的代价评估就是决定循环不变量外提的结果是否提高目标代码的性能。我们分几种情况讨论。

情况1：大循环体

假设循环不变量的结果存放于变量 x 中，如果这个循环不变量被外提，那么 x 所对应的 *live range* 将与所有在循环体中所有 *live range* 干涉。由于上述原因，在一个大循环上进行循环不变量外提会大大增加寄存器压力。

循环体的“大小”可以用它所包含“执行频率较高”的指令的条数来衡量。我们不应该用循环体所包含的指令条数来衡量循环体的“大小”。例如在图4中（有向边旁边的数字表示它的执行概率），尽管图示的循环有很多基本块，但是执行频率较高的仅2个，即块1和块2。尽管外提块1和块2中的循环不变量会增加互相干涉的 *live range* 个数，但是在寄存器分配器会在执行频率很低的块3和块4附近 *spill* 或 *split* 部分 *live range*[7]，从而降低了相互干涉的 *live range* 个数。情况2中将讨论如何界定“执行频率较高”的指令。

上述讨论得到这个结论：如果循环体很“大”，不外提任何循环不变量。

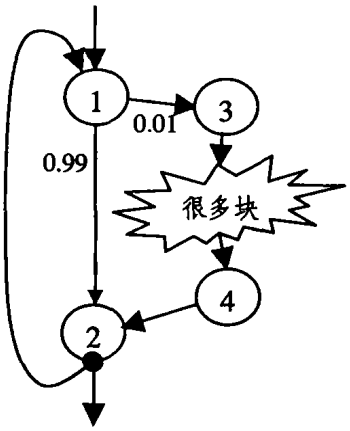

图4

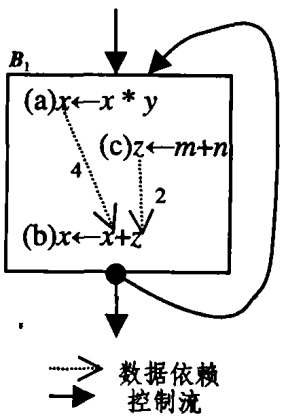

图5

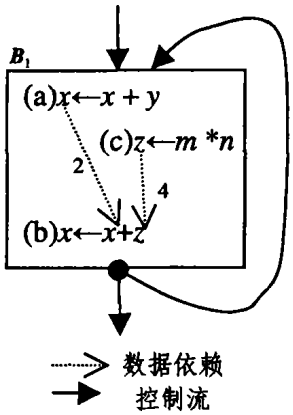

图6

情况2：计算循环不变量的指令的执行频率低

计算循环不变量的指令的执行频率低外提不会对性能的提高有任何帮助。如何界定执行频率高和低的指令呢？我们只能借助一些通过实验得到的经验规则了：

规则1：如果在编译时刻不知道循环头的频率，那么如果在循环体内从循环头计算循环不变量的指令 i 所在的基本块的到达概率小于0.2，我们认为指令 i 的执行频率低。

规则2：如果指令 i 的执行频率（在编译时刻就知道或通过动态 profile 获知）小于100000，那么我们认为指令 i 的执行频率低。

情况3：计算循环不变量的指令不在关键路径上

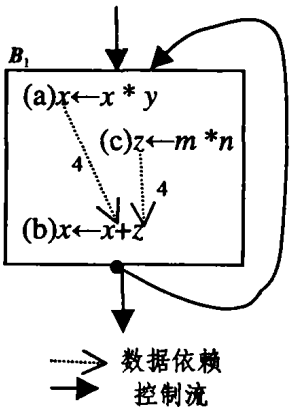

图7

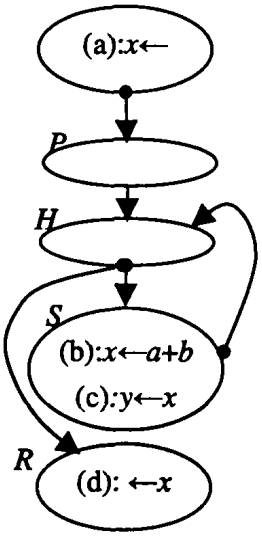

图8

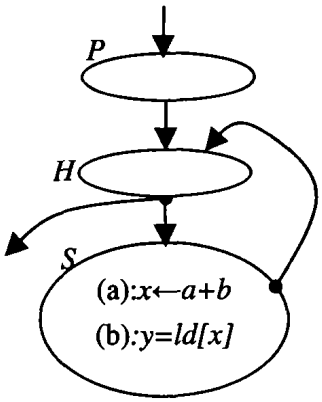

图9

如果计算循环不变量的指令 i 不在关键路径上，显然，由于没有必要循环不变量的计算“外提”到循环之外，指令 i 可以利用处理机空闲的功能部件执行。例如在图5中，指令(a)和(b)之间有4拍的延迟，指令(c)和(b)之间有2拍的延迟。在这个例子中，指令(a)和(b)处在关键路径上，它们分别只能在第0拍和第4拍发射。尽管(c)计算的表达式 $m+n$ 是循环不变量，但是(c)不处在关键路径上，它可以不用外提到循环之外，而是可以被调度到第1或2拍发射。

情况4：计算循环不变量的指令在关键路径上

假设计算循环不变量的指令 i 在关键路径上，且 i 的依赖高度为 $DH(i)$。另设在所有的候选指令中，有 n 条候选指令的依赖高度和 $DH(i)$ 相等。

(a) n = 1。这种情况说明把 i 移到循环之外能够缩短执行路径。例如在图6中，指令(c)计算循环不变量，DH((c)) = 4+1=5(设(b)的执行时间为1拍)，且(c)在关键路径上。另外一个候选指令(a)的依赖高度仅为 DH((a)) = 2+1=3。因此，外提(c)能够把循环体的执行时间由原先的5拍变成3拍。

(b) n > 1。在这种情况下，指令 i 是否需要外提依赖于目标处理机的发射宽度。例如在图7中，(c)计算循环不变量。指令(a)和(c)具有相同的优先级。当且仅当目标处理机能够同时发射的乘法指令小于2时，(c)才有必要外提。在具体实现上，如果调度器有多个具有相同优先级的候选指令可供选择时，调度器优先调度不是计算循环不变量的指令。如果最终计算循环不变量的指令 i 被滞后(即迟于预计的最迟发射时间)，那么决定将 i 外提。

上述讨论忽略了引入临时变量的开销。如果调度器决定外提某一个循环不变量时，还要继续考虑外提是否需要引入临时变量。如果答案是肯定的而且复制临时变量的指令的执行时间不小于计算循环不变量的时间，那么对应的循环不变量不外提。例如，若加法指令的执行时间不大于复制指令，那么图2中的循环不变量 $a+b$ 就没有必要外提(图3)。

4.2 合理性检查

假设计算循环不变量的指令所在的基本块为 S，且循环不变量被外提到它的前置块(preheader)[8] P 中。P 支配(dominate) S，而 S 可能后支配(post-dominate) P，也可能不后支配 P。从指令调度的角度看，如果 S 不后支配(post-dominate) P，那么把计算循环不变量的指令 i 移到 P 中这个行为应该被看作是控制投机(control speculation)的。因此，应该避免控制投机会带来的两个问题：(1)覆盖(kill)活跃变量；(2)产生异常。

算法1 Kill_live_var

输入：循环 L，计算循环不变量的指令 i：$x \leftarrow E$
输出：true 或 false

```
S←i 所在的基本块;
P←循环 L 的前置块;
if S 后支配 P;then
  return false;
endif
for each M in{L 的出口基本块}do
    for each N in {M 的直接后继}do
        if N 不在 L 中的∧变量 x 在 N 的入口处活跃;then
            return true;
        endif
    endfor
endfor
return false
```

4.2.1 *避免覆盖活跃变量* 我们以图8为例。表达式 $a+b$ 是循环不变量，且存放 $a+b$ 的变量 x 本身也是循环不变量。由于指令(b)所在的基本块 S 不后支配前置块 P，因此把指令(b)移到前置块 P 中的行为是控制投机。这个控制投机的结果是覆盖了在基本块 R 入口处活跃的变量 x。在这种情况下，我们考虑对 x 重命名(renaming)，重命名的方法与3.2节介绍的“引入临时变量”完全一致。

对于循环 L 中的计算循环不变量的指令 i：$x \leftarrow E$，算法1判断外提 i 是否会覆盖活跃变量 x。如果答案是肯定的，算法1返回 true，否则 false。

4.2.2 *避免产生异常* 一些可能产生异常的指令，例如除法、取数等指令，一般不能够被控制投机，除非在编译时刻能够确保这些指令在程序任何可能输入都不会产生异常。因此，如果计算循环不变量的运算如果可能产生异常，首先必须要判断“外提”这个行为是控制等价的代码移动还是指令投机。如果是投机，我们必须确保被“外提”的循环不变量的计算在任何可能输入都不会产生异常，否则不外提该循环不变量。

例如在图9中，设 $a+b$ 是循环不变量，因此表达式 $ld[x]$ 也是循环不变量。由于指令(b)所在的基本块 S 不后支配循环的前置块 P，把 $ld[x]$ 外提到前置块 P 是投机的而不是控制等价的。$ld[x]$ 可能由于所访问的虚地址 x 无效而触发异常，而这个异常在循环不变量外提之前可能不会发生(循环的迭代个数为0，控制流根本没有经过块 S)。

5 实验报告

我们在开放源码编译器 ORC[9] 上实现了本文所介绍的算法。ORC 是 Intel 系统软件实验室和中科院计算所的联合项目。目前，它支持的目标处理机有 Itanium-1[10] 和 Itanium-2[10]。ORC 具有十分丰富的优化阶段。它的中端由全局标量优化(WOPT)和过程间分析(IPA)两个模块组成。在 WOPT 中，以死代码和完全冗余形式存在的循环不变量分别在死代码删除和 value numbering 这两个阶段中予以删除。而以部分冗余形式存在的循环不变量则会在部分冗余删除阶段中自动“外提”。

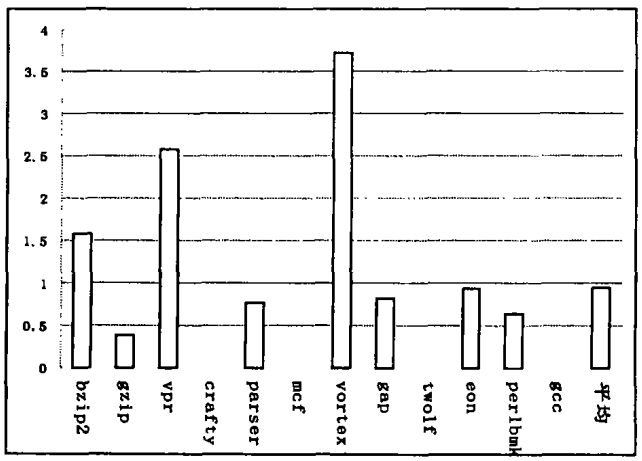

图10

(下转第165页)

会使重构变得更加实用，并期待着在不久的将来，重构技术真正在软件工程师团体内流行起来。

参考文献

1 Fowler M. Refactoring: Improving the Design of Existing Code. Addison Wesley Longman, Inc., Reading, Massachusetts, 1999
2 Beck K. Extreme Programming Explained. Upper Saddle River: Addison Wesley, 1999
3 Emden E van, Moonen L. Java quality assurance by detecting code smells. In: Proc. of the 9th Working Conf. on Reverse Engineering. IEEE Computer Society Press, Oct. 2002
4 Antoniol G, Fiutem R, Cristoforetti L. Using metrics to identify design patterns in object-oriented software. In: 5th Intl. Symposium on Software Metrics, March 1998
5 Simon F, Steinbruckner F, Lewerentz C. Metrics based refactoring. in CSMR, 2001. 30～38
6 Astels D. Refactoring with UML. http://www.agilealliance.org/articles/articles/RefactoringWithUML.pdf
7 Boger M, Sturm T, Fragemann P. Refactoring Browser for UML. http://www.xp2002.org/atti/Boger-Fragemann--Refactoring-BrowserforUML.pdf
8 Simon F, Löffler S, Lewerentz C. Distance Based cohesion measuring. In: proc. of the 2nd European Software Measurement Conf. (FESMA)99, Technologist Institute Amsterdam, 1999
9 Chidamber S R, Kemerer C F. A metrics suite for object oriented design. IEEE Transactions on Software Engineering, 1994
10 Bieman J M, Ott L M. Measuring functional cohesion: [Technical Report CS-93-109]. Michigan Technological University, 1993
11 Bieman J M, Kang B K. Measuring Design-Level Cohesion. IEEE Transac- tions on Software Engineering, 1998, 24(2)
12 Agile Modeling website. www.agilemodeling.com

（上接第161页）

ORC 的中间表示 WHIRL 是层次化的。WOPT 阶段的中间表示包括 WHIRL 的中层和低层。在 WOPT 之后，目标代码将经历代码生成(CG)阶段。这一个阶段主要由针对目标处理机的优化构成，它的中间表示非常接近目标机的指令。在 WOPT 的中间表示向 CG 的中间表示递降的过程中，会引入一些循环不变量。我们在代码生成阶段实现的循环不变量外提就是为了删除这些循环不变量。

SPEC 公司的 CPU2000[11]的整形测试例被用于评估在代码生成阶段的循环不变量外提(LICM)的性能。运行测试例的机器为 Itanium-2服务器。它具有 SMP 结构，每一个处理机的频率为896Mhz。

LICM 加速比如图10所示。在这里加速比被定义为：(无 LICM 时测试例的运行时间－有 LICM 时测试例的运行时间)/ 无 LICM 时测试例的运行时间 × 100%。Vortex 和 vpr 的循环小而多，因而，LICM 对这两个测试例子上能够取得比较显著的效果。gcc 和 crafty 的运行频率比较高的循环一般都比较大，循环不变量的外提的收益偏小。

表1

测试例	Loop#	Skip-Loop#	Skip-Loop%	LI#	Skip-LI#	Skip-LI%
Bzip2	180	115	64	198	116	59
Gzip	247	178	72	115	91	80
Vpr	426	298	70	277	208	75
crafty	462	367	79	191	180	94
mcf	50	22	44	45	37	82
parser	787	382	49	327	280	86
gap	2008	1765	88	350	247	71
perlbmk	808	685	85	115	79	69
eon	478	382	80	54	53	98
vortex	226	165	73	311	236	76
twolf	992	759	77	255	181	71
gcc	2839	2317	82	678	558	82
平均			72			78

表1反映了我们的代价模型的性能。从左往右各列的意义是：测试例、测试中的循环个数、被忽略的循环个数(由于循环体过大或循环的执行频率过低，详见4.1节中情况1和情况2)、被忽略的循环所占的百分比，在没有被忽略的循环中的循环不变量个数以及循环不变量中不被外提的个数，这些不外提的循环不变量所占的百分比。我们的实验结果，大约有72%的循环由于它们的循环体过大或者由于它们的循环迭代的个数较少而不进行循环不变量外提。在进行循环不变量外提的循环中，平均有78%的循环不变量不需要外提到循环之外。其中，由于 eon 的基本块较大，eon 中的循环不变量基本不需要外提到循环之外。

总结 在现代编译器中，循环不变量外提通常在中端进行。由于中端和后端的中间表示存在差别，当中端的中间表示向后端的中间表示递降时会引入循环不变量。在后端再次进行循环不变量外提是很有必要的。我们把不变量外提集成到指令调度的过程中以充分利用调度器已有的数据结构进行代价评估。实验显示我们的循环不变量外提能够提高目标代码1%的性能。在调度的同时进行循环不变量外提能够避免78%的循环不变量外提到循环之外。

参考文献

1 Click C. Global code motion/global value numbering. In: Proc. of the ACM SIGPLAN '95 Conf. on Programming Language Design and Implementation, La Jolla, California, June 1995. 246～257
2 Knoop J, Ruthing O, Steffen B. Lazy code motion. In: Proc. of the ACM SIGPLAN '92 Conf. on Programming Language Design and Implementation. SIGPLAN Notices, 1992, 27(7): 224～234
3 Cytron R, Ferrante J, Rosen B K, Wegman M N, Zadeckm F K. Efficiently Computing Static Single Assignment Form and the Control Dependench Graph. ACM Transactions on Programming Languages and Systems, 1991, 13(4): 461～486
4 Muchnick S S. Advanced Compiler Design and Implementation. Morgan Kaufmann Publishers, San Francisco, California. 397～406
5 Coffman E G Jr. Computer and Job Shop Scheduling Theory. Wiley, New York, 1976
6 Muchnick S S. Advanced Compiler Design and Implementation. Morgan Kaufmann Publishers, San Francisco, California. 539～543
7 Chow F C, Hennessy J L. The priority-based coloring approach to register allocation, ACM Trans. on Programming Languages and Systems, 1990, 12(4): 501～536
8 Muchnick S S. Advanced Compiler Design and Implementation. Morgan Kaufmann Publishers, San Francisco, California, 193
9 ORC team. ORC v2.0 suite. http://sourceforge.net/projects/ipf-orc, 2001～2003
10 Intel Corp. http://www.intel.com/design/itanium/itanium/index.htm.
11 Standard Performance Evaluation Corporation. http://www.spec.org/cpu2000, 2003