

即时编译器中的轻量级指令调度算法

史晓华¹, 刘 超¹, 金茂忠¹, 郭 鹏²

(1. 北京航空航天大学软件工程研究所, 北京 100083; 2. 英特尔微处理器研究院, 北京 100086)

摘 要: 介绍了一种为即时编译器和时空受限系统设计的轻量级线性复杂指令调度算法。该算法进行指令调度时, 不基于传统的 DAG 图或表达式树, 而是基于一种独创的数据结构扩展关联矩阵, 其时间复杂性在最坏情况下也能与全部指令长度构成严格的线性关系, 仅占用不到 1 KB 的内存空间。该算法已被 Intel 为 Xscale 设计的高性能 J2ME 虚拟机 XORP 采用为即时编辑器中的缺省指令调度算法。

关键词: 指令调度; 即时编译器; Java 虚拟机

Lightweight Instruction Scheduling Algorithm
for Just-in-time CompilerSHI Xiao-hua¹, LIU Chao¹, JIN Mao-zhong¹, GUO Peng²

(1. Software Engineering Institute, Beijing University of Aeronautics and Astronautics, Beijing 100083;

2. Intel Microprocessor Technology Labs, Beijing 100086)

【Abstract】 This paper presents a novel lightweight algorithm of instruction scheduling for reducing the pipeline stalls on RISC micro-architecture with super pipelines. The algorithm is designed for and implemented in a J2ME JIT(just-in-time) compiler in memory constraint and time critical system. It is not based on DAG(directed acyclic graphs) or expression trees, but a novel data structure, namely EDM(extended dependency matrix). The algorithm has strict linear time complexity to the code length in the worst cases. It consumes only about 1KB constant memory space, to fit the small memory footprint for J2ME configurations. This algorithm has been chosen as the default instruction scheduling solution in the JIT compiler by Intel's high-performance J2ME JVM, namely XORP.

【Key words】 instruction scheduling; just-in-time compiler; Java virtual machine

本文以 Intel XScale 微处理器为目标硬件平台, 介绍一种应用于 J2ME(Java 2 platform, micro edition)即时编译器(just-in-time compiler)中, 针对时空受限系统设计的、独创的轻量级指令调度算法(lightweight instruction scheduling, LIS)。采用权威 J2ME 基准程序集 EEMBC 测试表明, 该算法平均可以削减超过 25.3%由数据关联造成的流水线停顿, 平均提高 7.14%以上的程序运行速度。实践发现, 该算法的平均编译时间不到 list scheduling 算法的一半, 代码的整体优化效果也与前者相当。

1 相关工作

许多有关指令调度的重要研究成果都讨论了如何减少数据关联造成的流水线阻塞, 同时有效提高程序运行的指令级并行度(instruction level parallelism, LP)。Proebsting^[1]提出了将指令选择、指令调度组合在一起的算法, 该算法是基于表达式树的。Kurlander^[2]将上述算法应用在 DAG 图上, 并评估了 Goodman 和 Hsu 的 IPS 算法的效率^[3]。list scheduling^[4,5]是在编译器中广泛采用的指令调度算法。实践发现, 该算法的时间复杂性可以接近线性, 在较坏的情况下接近 $O(N^2)$, 其中的 N 为指令长度, 但这仍会给对编译时间高度敏感的 J2ME 即时编译器带来一些问题。例如, 此类指令调度算法可能会减缓 Java 应用程序在启动阶段的运行速度, 从而降低整个系统的效率, 并给用户以程序运行“发涩”的感觉。

2 XScale 的超流水线结构

如图 1 所示, 在 XScale 的单发射超流水线^[6]上, 内存访

问、数据运算和乘法指令可以分别进行处理。不同流水线上的指令之间如果不存在数据关联关系, 就可能乱序完成。

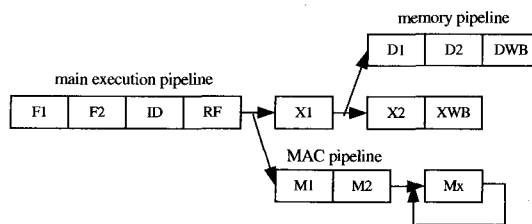


图 1 XScale 的 RISC 核心超流水线结构

例如, 下列指令将依次进入主流线, 但可能不按照进入主流线的次序完成执行, 指令 I_1 可能在指令 I_0 之前完成执行。

I_0 : ldr R1, [R0]

I_1 : add R2, R2, R3

3 轻量级指令调度算法

3.1 扩展关联矩阵

LIS 算法的实现依赖于一个独创的定长数据结构, 扩展

基金项目: 国家自然科学基金资助项目(60573084); 英特尔研究基金资助项目

作者简介: 史晓华(1973—), 男, 博士, 主研方向: 编译系统, 体系结构, 运行时系统, 嵌入式软件环境; 刘 超, 教授; 金茂忠, 教授、博士生导师; 郭 鹏, 硕士、研究员

收稿日期: 2007-04-23 **E-mail:** shixiaohua@gmail.com

关联矩阵(extended dependency matrix, EDM)。LIS 通过检查 EDM 的内容,选择合适的指令将它们合法地移动到其他指令之前,以减少流水线阻塞造成的运行效率降低。LIS 并不会完全打乱原有指令的次序,而是通过有限的调度,尽快找到合适的候选指令,填补流水线上由数据关联造成的空洞。

图 2 给出一段典型的字节码程序以及即时编译器在实施指令调度优化前生成的机器码。假设 I_2 中的内存访问指令将命中 L1 数据缓存并且读取内存指令的典型反应时间在命中 L1 数据缓存的前提下为 3 个时钟周期,那么整段代码 4 条指令的执行时间为 4 个时钟周期加上指令 I_3 运行时由于数据关联造成的 2 个时钟周期的流水线阻塞,共计 6 个时钟周期。

字节码	生成代码
iload_1	
iload_2	
iadd	I_0 : add R0,R5,R6
iload 3	
iload 4	
isub	I_1 : sub R1, R7,R8
aload_0	
getfield #5	I_2 : ldr R2, [R4, 0x4]
iadd	I_3 : add R3, R2, R1

图 2 字节码与生成代码

对于图 2 中编译生成的代码,相应的 EDM 表示在图 3 之中。EDM 左侧是一个数据关联关系矩阵,其中,“-1”表示横纵行列对应的 2 条指令不存在关联关系,如果该值非负,则表示这 2 条指令之间可能存在的流水线阻塞的最大时钟周期数。例如,在行 I_3 和列 I_2 交汇处的“2”表示: I_3 可能需要在流水线上等待 2 个时钟周期直到 I_2 执行完成。左侧的矩阵内容可以非常容易地在生成代码阶段得出。

	I_0	I_1	I_2	I_3	Stl	Cyl	Ceil	UP	DWN
I_0	\				0	1	-1	-1	4
I_1	-1	\			0	2	-1	-1	3
I_2	-1	-1	\		0	3	-1	-1	3
I_3	-1	0	2	\	2	6	2	2	4

图 3 扩展关联矩阵

图 3 中虚线右侧的矩阵内容是由左侧计算所得,下文将详述其各项数值的计算公式。为了简化问题,假设所有指令的 issue latency 为 1, EDM 包含的最大指令个数为 M 。

对于可能被阻塞在流水线上的指令 I_n , 其对应的 $Ceil_n$ 和 Stl_n 值将被同时计算出。首先找到阻塞 I_n 的指令, 将其下标记录在 $Ceil_n$ 中, 同时计算 I_n 可能被阻塞的时钟周期数, 记录在 Stl_n 中。引入临时变量 $t_{nk}=I_{nk}-(n-k-1)$, 表示当代码执行到 I_n 时, 由指令 I_k 存在数据关联关系而导致的流水线阻塞的最大时钟周期长度。 t_{nk} 小于等于 0 时表明流水线不会阻塞。

$$Ceil_n = \begin{cases} k, t_{nk} = \text{MAX}\{t_{n0}, \dots, t_{n(n-1)}\} \text{ and } t_{nk} \geq 0 \\ -1, n=0 \text{ or } \forall t_{nk} \in \{t_{n0}, \dots, t_{n(n-1)}\} \in t_{nk} < 0 \end{cases} \quad (1)$$

$$Stl_n = \begin{cases} t_{nk}, t_{nk} = \text{MAX}\{t_{n0}, \dots, t_{n(n-1)}\} \text{ and } t_{nk} \geq 0 \\ 0, n=0 \text{ or } \forall t_{nk} \in \{t_{n0}, \dots, t_{n(n-1)}\} \in t_{nk} < 0 \end{cases} \quad (2)$$

“Cyl”列中的数值表明根据估算,从 I_0 开始到执行完某条指令的总钟周期,其公式如下。该值等于前一条指令的 Cyl 值加上该指令对应的 Stl 值,再加上指令本身执行需要消耗的时钟周期(假设均为 1)。

$$Cyl_n = \begin{cases} Cyl_{n-1} + Stl_n + 1, n > 0 \\ 1, n = 0 \end{cases} \quad (3)$$

“UP”和“DWN”的 2 列数据记录了一条指令可以被合法移动的范围。即任意一条指令在其对应的 UP 和 DWN 范围内移动,都不会违反原有的数据关联关系。例如,指令 I_0 的 UP 值为 -1, DWN 值为 4, 意味着 I_0 可以被合法地移动到第“-1”条指令之后,即 I_0 之前,第 4 条指令之前,即 I_3 之后)的任意位置上。

对于每条指令,它的 UP 值等于 EDM 左侧矩阵该指令所在行中从右向左第 1 个非负值下标,或者 -1, 当该指令所在行的值均为 -1, 见式(1)。例如, I_3 所在行从右向左第 1 个非负值出现在 I_2 所在的列,那么 I_3 的 UP 值便为 2, 对于指令 I_1 和 I_2 , 它们所在的行中均未出现非负值,这 2 条指令的 UP 值均为 -1。

$$UP_n = \begin{cases} k, I_{nk} > 0 \text{ and } \forall I_{nj} \in \{I_{n(k+1)}, \dots, I_{n(n-1)}\} I_{nj} < 0 \\ -1, \forall I_{nk} \in \{I_{n0}, \dots, I_{n(n-1)}\} I_{nk} < 0 \end{cases} \quad (4)$$

每条指令的 DWN 值等于该指令所在列中从上到下第 1 个非负值的下标,或者 EDM 中最后一条指令的下标加 1, 当该指令所在列均为 -1, 见式(2)。例如, I_1 和 I_2 所在列的第 1 个非负值的下标为 3, 它们的 DWN 值便为 3。 I_0 所在列中未出现非负值, 它的 DWN 值为 $3+1=4$ 。

$$DWN_n = \begin{cases} k, I_{kn} > 0 \text{ and } \forall I_{jn} \in \{I_{0n}, \dots, I_{(k-1)n}\} I_{jn} < 0 \\ m+1, I_{kn} \in \{I_{(n+1)n}, \dots, I_{Mn}\} I_{kn} < 0 \end{cases} \quad (5)$$

至此,整个 EDM 构造完成。从中可以发现, Stl 值非 0 的指令在执行时会出现流水线阻塞,即 I_3 在执行时会出现 2 个时钟周期的流水线空洞。在介绍算法之前,通过观察 EDM, 容易得出这样的调度方案,将 DWN 值都大于 I_2 下标的 I_0 和 I_1 安全地移动到 I_2 之后,可以减少 I_2 和 I_3 之间由于数据关联所造成的阻塞 2 个时钟周期,如图 4 所示。由此,这 4 条指令的执行时间在遵循前文假设的前提下可以缩短为 4 个时钟周期,这也是任何指令调度方案可以达到的最优结果。

I_2 :	ldr R2, [R4, 0x4]
I_0 :	add R0, R5, R6
I_1 :	sub R1, R7, R8
I_3 :	add R3, R2, R1

图 4 调度完成的结果

3.2 LIS 算法

LIS 算法的核心就是前文描述的 EDM 矩阵。该算法的设计思想是,当被调度的基本块中存在正数 Stl 值时,在导致流水线阻塞的指令周围,寻找合适的候选指令,将它们移动到被阻塞指令之前,减少流水线上的运行停顿;如果被调度基本块中所有 Stl 值均为 0, 则无须进行指令调度。

下面给出了 LIS 算法的伪代码,下文将结合图 2 中的代码实例和图 3 中的 EDM 矩阵,解释该算法的运行机制。

```

for (every  $Stl_n > 0$ ) {
     $t = Stl_n$ ;
    for ( $m = Ceil_n - 1$ ;  $m \geq 0$  &&  $t > 0$ ;  $m--$ ) {
        if ( $I_m$  belongs to other  $Stl\_E$ ) break;
        if ( $I_m$  has been moved before) continue;
        if ( $DWN_m > Ceil_n$ ) {
            move  $I_m$  after  $Ceil_n$ ;
             $t = t - \text{issue-latency}(I_m)$ ;
        } /*if*/
    } /*for*/
    for ( $m = n + 1$ ;  $m \leq \text{last instruction}$  &&  $t > 0$ ;  $m++$ ) {
        if ( $I_m$  belongs to other  $Stl\_E$ ) break;
    }
}

```

```

if(  $I_m$  has been moved before) continue ;
if(  $UP_m < n$  ){
    move  $I_m$  before  $I_n$ ;
     $t = t - \text{issue-latency}(I_m)$ ;
}/*if*/
}/*for*/
}/*for*/

```

首先 LIS 对于每一个出现在 EDM 中大于 0 的 Stl_n , 都从 $Ceil_n$ 之前的指令依次开始寻找是否有合适的候选指令可以被合法地移动到标为 $Ceil_n$ 的指令之后。需要注意的是, 对于 $Ceil_k$ 不为 0 的指令, 要将下标从 $Ceil_k$ 到该指令的所有指令定义为一个“关联指令子集” Stl_E_k 。不论该指令的 Stl 值是否为正数, LIS 都不会把指令从这样的子集中移去。这是为了避免以下情况: 尽管将某条指令移动后可以减少目的区域附近指令的流水线阻塞, 但会造成该指令原来所在区域新流水线的停顿。这也是设计 LIS 的宗旨: 合理地将优化效率与时间复杂性和空间复杂性进行折中, 以便得出最适合应用环境的解决方案。

当某条指令的 DWN 值大于 $Ceil_n$, 该指令便可合法地移动到 $Ceil_n$ 之后。当有多条指令需要移动时, 这些指令在目的区域中仍将保持原有顺序。每当一条指令被移动到 $Ceil_n$ 之后, 便从 Stl_n 中减 1 (假设所有指令的 issue latency 均为 1 个时钟周期), 当 Stl_n 为 0, 或者遍历到某个 Stl_E 子集中, 向前方寻找合适指令的循环结束。

如果该循环完成时 Stl_n 仍为正数, LIS 将依次访问 I_n 之后的指令, 寻找其中 UP 值大于 n , 并且不属于任何 Stl_E 子集的指令。如果存在, 该指令将被移动到指令 I_n 之前。同样, 当多条指令需要移动时, 这些指令在目的区域中仍保持原有顺序。每当一条指令被移动到 I_n 之前, 便从 Stl_n 中减 1, 当 Stl_n 为 0, 或者遍历到某个 Stl_E 子集中, 向后方寻找合适指令的循环结束。

从上文可以发现, 任何一条不出现在 Stl_E 子集中的指令最多被访问 2 次。下面讨论 LIS 的正确性。根据 UP 和 DWN 的定义, 任何一条指令被移动到其 UP 值之前或者 DWN 值之后, 都不会打破原有的数据关联关系。根据 LIS 算法的描述, 当 LIS 准备将指令 I_m 移动到 $Ceil_n$ 之后时, 不论算法进行到哪一步, 出现在 I_m 与 $Ceil_n$ 之间的所有指令, 都将是算法启动前原有的全部或一部分指令, 并且这些指令的原有顺序也未经改变。同理, 将指令 I_m 移动到 I_n 之前时, I_m 与 I_n 之间的所有指令也将是算法启动前原有的全部或一部分指令。因此, 这样的移动不会破坏原有代码的数据关联关系。

这意味着, 在 LIS 调度过程中, 除了更改当前 Stl 的数值, 不需要回溯更新任何 EDM 中其它的数据内容。这也是 LIS 具有较低复杂性的原因之一。

根据上述 LIS 算法, 结合图 2 和图 3 中的例子, 可以发现只有 Stl_3 大于 0, 并且只有一个关联指令子集 Stl_E_3 , 该子集包括指令 I_2 和 I_3 。在 LIS 调度过程中, LIS 算法中的历史变量 t 首先被赋予 Stl_3 的值, 等于 2。算法从 $Ceil_3$, 即指令 I_2 之前开始寻找合适的候选指令。因为 DWN_1 大于 2, 所以 I_1 先被移动到 I_2 之后, 同时 t 减去 I_1 的 issue latency, 即 1 个时钟周期。因为 $t=1$ 仍然大于 0, LIS 继续向前寻找。 DWN_0 为 4, 意味着它也可以被合法地移动到 I_2 之后。根据前文约定, I_0 被移动后将出现在 I_2 与 I_1 之间, 仍保持与 I_1 固有的指令顺序。

此时 t 再减去 1 后等于 0。LIS 算法将从内循环中退出。由于 EDM 中不再有其他大于 0 的 Stl , 对该基本块的调度结束。调度结果同图 4 中经过手工整理的代码段完全一致。原有代码中 2 个时钟周期的流水线阻塞被削减, 并且这也是任何其他调度算法能够做到的最优解。

3.3 时间及空间复杂性分析

LIS 算法具有较低的时间复杂性由多方面的原因决定:

(1) 根据对 LIS 算法的描述, 任何指令在调度时最多被访问 2 次。与其他调度算法不同, 在 LIS 调度过程中, 除了更改当前 Stl 的数值, 不需要回溯更新任何 EDM 中其它的数据内容。保证了 LIS 算法的时间复杂性在最坏的情况下仍为严格的 $O(N)$, 其中, N 为指令长度。

(2) 算法围绕着大于 0 的 Stl 展开, 根据统计, 尽管这样的指令会对整个程序的运行造成较大影响, 但其数量仅占全部指令的 15% 左右 (统计数据见第 4 节), 即 LIS 算法的实际复杂性与 Stl 大于 0 的指令数量构成线性关系。

(3) 当设定 L1 缓存的访问时间为 3 个时钟周期时, 如果由于访问内存导致 Stl 大于 0, 那么该值一定小于等于 2, 即 LIS 算法需要为每个大于 0 的 Stl 寻找的候选指令最多为 2 条。

(4) 只有 Stl 大于 0 的基本块才参与调度。这与很多其他调度算法不同。如果编译器已经生成了没有流水线空洞的代码, 自然不需要再进行指令调度。

对于虚拟机中每个独立运行的线程, LIS 算法需要的内存空间主要是 EDM 占用的空间。将调度窗口的长度限制在 16 条指令时, EDM 的大小一般不超过 1KB。这对于一个在运行时间和空间上都有较严格要求的 J2ME 虚拟机即时编译器而言, 大小适中。

4 性能评测

实现 LIS 的 XORP 即时编译器对于 Java 函数采用“初次调用, 即时编译”的策略, 即任何 Java 函数都不会被编译, 直到它第 1 次被调用。以下所有评测数据都基于这样的前提。XORP 运行在嵌入式 Linux 上, 硬件环境采用 XScale PXA255 CPU, SRAM 为 64MB, CPU、BUS 和内存主频分别为 333MHz、166MHz 以及 84MHz。XORP 运行时设定的堆大小为 8MB。

本文不讨论 Java 字节码在解释器, 或者解释器和编译器混合模式下的运行情况。

下面将从权威的 J2ME 基准评测程序 EEMBC^[7] 中选择 6 个具有代表性的应用程序进行 LIS 算法性能的评估。

4.1 list scheduling 的参考实现

本文实现了一个 list scheduling 算法作为评测参考。该算法采用了文献[4,5]中介绍的启发式规则, 不与寄存器分配联动, 并且采用同 LIS 相同的输入和输出条件。

该算法直接将编译器生成的机器码通过一个 register scoreboard 建立 DAG 图, 然后从该图的根节点开始遍历, 依次选择具有最早执行时间和最小延迟时间的节点, 同时, 将被选择节点的子节点的当前时间和最早执行时间进行更新。完成遍历时, 依次选择的节点就形成了调度结果。

4.2 编译时间和实际运行性能评测

之前分析过, LIS 算法的实际时间复杂度与同被阻塞在流水线上的指令长度有关, 而与全部指令长度无关。从图 5 可以发现, 通过静态统计可以得出编译后 EEMBC 全部指令的长度是被阻塞指令长度的 6.8 倍, 对于某些测试程序, 例如 kXML, 这个比例上升到了 8.31。

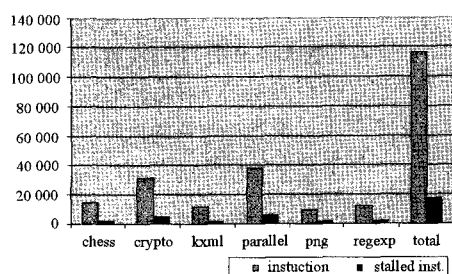


图5 总指令长度与阻塞指令长度

对于即时编译器而言,它的编译时间将计入 Java 程序的总运行时间。从图 6 中可以看到,包含 list scheduling 模块的编译时间占据了 EEMBC 第 1 次循环运行总时间的 25.3%,并且 list scheduling 占据了全部编译时间的 15.9%以及整个运行时间的 4%。图 7 展现了 list scheduling 和 LIS 在不同测试程序上花费的编译时间。可以看出, LIS 较 list scheduling 在运行速度上具有明显优势。list scheduling 平均花费的编译时间是 LIS 的 2.25 倍。对于 kXML 或者 PNG decoding 这样的程序, list scheduling 比 LIS 慢 3 倍左右。

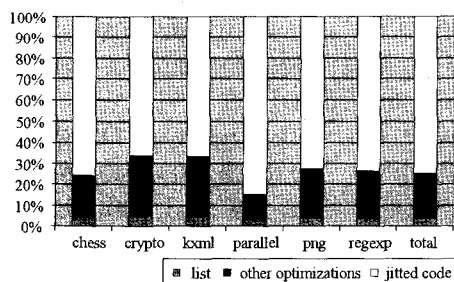


图6 编译与运行时间

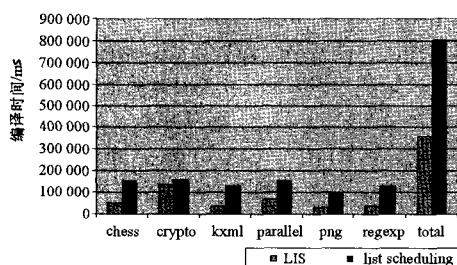


图7 LIS 和 list scheduling 的编译时间

根据图 8, LIS 削减的流水线阻塞平均达到 list scheduling 的 82.6%。该数据通过 XScale 上的 PMU 模块动态取得。对于 kXML 而言, LIS 的优化效率高于 list scheduling。通过 LIS 的优化, EEMBC 运行过程中平均 25.3%的流水线停顿被削减, Cryptography 有 41.4%的流水线停顿被削减。但即便经过指令调度,也无法彻底消除应用程序中的流水线停顿,原因之一是 XScale 不具有 L2 缓存,无论采用何种优化技术,都无法完全避免内存访问中出现大量的 L1 Cache Miss。

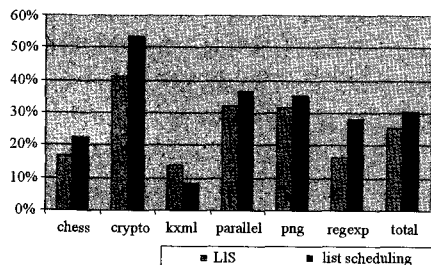


图8 削减的流水线阻塞

图 9 展示了通过 LIS 优化后, EEMBC 运行性能的提高。缺省状态下 EEMBC 将运行 5 个相同的循环, 第 1 次循环时, 被调用的 Java 函数都将被编译, 并且此后对这些函数的访问都不再需要即时编译器的参与。图 9 中同时展示了 EEMBC 循环 5 次和仅循环 1 次的性能提高数据。

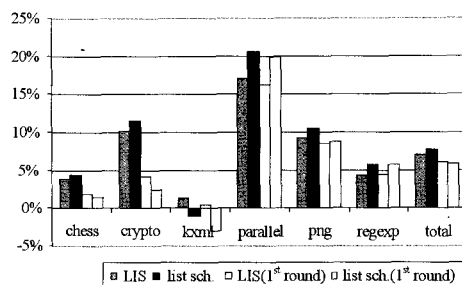


图9 EEMBC 运行性能的提高

从图 9 中可以看到, 在 5 次循环的情况下, LIS 使 EEMBC 运行速度平均提高了 7.14%, 其中, Parallel 被提高了 17.15%。在循环 1 次的情况下, 由于编译时间将占据整个运行时间的较大比例, LIS 在整体上超越了 list scheduling 大约 2%。根据 EEMBC 的运行速度上看, 即便循环 5 次的情况下, LIS 也能够达到与 list scheduling 相当的优化效果。

5 结论

目前在嵌入式平台上, J2ME 虚拟机不再仅仅依赖速度非常缓慢的解释器运行 Java 程序的字节码。主流的高端 J2ME 虚拟机, 例如 IBM WME J9、Sun CLDC-HI 都采用了优化的即时编译器。但是, 由于能耗、成本等原因, 诸如手机这样的嵌入式设备, 在可以预见的将来仍将对软件运行环境保留一些比较苛刻的限制。

对于 XScale 这样具有超流水线结构的处理器而言, 由于芯片本身不具备超标量和指令乱序发射等能力, 指令调度只能依靠编译器来完成。本文通过介绍一个新的轻量级指令调度算法 LIS, 尝试在综合考虑优化效果、时间复杂性和空间复杂性的情况下, 探寻如何在时间与空间受限的系统中设计即时编译器优化算法。

参考文献

- 1 Proebsting T A, Fischer C N. Linear-time, Optimal Code Scheduling for Delayed-load Architectures[C]//Proceedings of the ACM Conference on Programming Language Design and Implementation, Toronto, Ontario, Canada. 1991.
- 2 Kurlander S M, Proebsting T A, Fischer C N. Efficient Instruction Scheduling for Delayed-load Architectures[J]. ACM Transactions on Programming Languages and System, 1995, 17(5): 740-776.
- 3 Goodman J R, Hsu W C. Code Scheduling and Register Allocation in Large Basic Blocks[C]//Proceedings of the 2nd International Conference on Supercomputing, St. Malo, France. 1988.
- 4 Muchnick S S. Advanced Compiler Design and Implementation[M]. USA: Academic Press, 1997.
- 5 Smotherman M, Krishnamurthy S, Aravind P S, et al. Efficient DAG Construction and Heuristic Calculation for Instruction Scheduling[C]//Proceedings of the 24th Annual International Symposium on Microarchitecture. 1991: 93-102.
- 6 XScale™ Microarchitecture Technical Summary[Z]. (2003-01-01). <http://download.intel.com/design/intelxscale/XScaleDatasheet4.pdf>.
- 7 EEMBC Java Benchmarks[Z]. (2007-02-01). <http://www.eembc.org>.