第 35 卷 第 8 期 计 算 机 工 程 2009 年 4 月
Vol.35 No.8 Computer Engineering April 2009
·软件技术与数据库· 文章编号：1000—3428(2009)08—0122—03 文献标识码：A 中图分类号：TP315

保证 Java 精确异常的指令调度技术

张 垚
(复旦大学并行处理研究所，上海 201203)

摘 要： Java 语言的精确异常要求和 Java 程序中频繁出现的异常检测严重阻碍或限制了指令调度在 Java 本地代码编译中的应用，从而减少了代码的指令级并行度。提出的算法可以使指令调度打破 Java 精确异常要求，能最大程度地发挥作用，并在有效提高代码执行效率的同时确保精确异常要求在异常发生时不被破坏。实验结果证明该算法的有效性和正确性。

关键词： Java 语言；指令调度；精确异常

Instruction Scheduling Technology for Java Precise Exception

ZHANG Yao
(Parallel Processing Institute, Fudan University, Shanghai 201203)

【Abstract】 Many existing instruction scheduling algorithms are severely hindered or disabled by Java precise exception model and the frequent exception checks, which substantially reduce the amount of instruction level parallelism for compiled Java. This paper presents an algorithm which helps instruction scheduling break the precise exception constraints to achieve significant speedup while ensuring the precise exception constraints. The experimental results prove the usefulness and correctness of the proposed algorithm.

【Key words】 Java; instruction scheduling; precise exception

Java 作为一种通用语言得到了越来越多的应用和重视，然而相对较差的执行效率却阻碍了它在诸多领域的进一步发展。产生这一问题的一个主要原因是 Java 语言规范定义的精确异常要求[1]和程序中频繁出现的异常检测[2-3]严重限制和阻碍了很多指令调度算法[4]在 Java 本地代码编译中的应用，使编译器无法充分发掘代码的指令级并行度(Instruction Level Parallelism, ILP)并有效利用硬件资源，从而大大降低了 Java 代码的执行效率。

1 Java 的异常处理与精确异常要求

把可能在Java程序执行期抛出异常的指令称为潜在异常指令(Potential Exception Instruction, PEI)，Java 本地代码编译器通常使用条件跳转指令来表示 PEI，从而显示检测异常是否发生[3]。若异常条件满足，称当前 PEI 抛出异常，控制流跳转到某个与当前 PEI 对应的异常处理模块 EGTB(Exception Generation and Throwing Block)，生成并抛出对应异常。然后根据异常种类，找到对应的异常处理单元(exception handler)入口，执行其代码并处理抛出的异常；若没有异常发生，则程序正常执行。此外，Java 语言规范引入精确异常要求，其可理解为[2]：(1)程序状态一致性。当某个异常发生时，优化程序中对应的异常处理单元入口的可见程序状态必须和未优化的原程序一致。(2)异常抛出顺序一致性。在多个异常的成立条件都满足的前提下，优化程序的异常抛出顺序必须和未优化原程序一致。

2 精确异常要求与指令调度之间的矛盾

为了保证 Java 精确异常的要求，编译器在处理 PEI 之间关系、PEI 与修改程序状态指令 PSMI(Program State Modification Instruction)，即修改程序非临时变量的指令之间关系时，必须引入精确异常相关依赖关系[2]。

为保证程序状态一致性，编译器在 PEI 与 PSMI 之间引入程序状态依赖[2]，使 PSMI 不会被调度到其之前(或之后)的 PEI 之前(或之后)。即当某个 PEI 发生异常时，在其之前的所有 PSMI 都被执行完毕而在其之后的所有 PSMI 都未被执行，从而保证当该 PEI 抛出异常时，与其对应的异常处理单元入口可见程序状态与原程序一致。

为保证异常抛出顺序一致性，编译器在相邻 PEI 之间引入异常顺序依赖[2]，阻止 PEI 之间相对位置因调度而产生变化，确保 PEI 按照原程序中规定顺序先后执行。

精确异常相关依赖的引入严重限制和阻碍了指令调度的应用。频繁出现的通过条件跳转指令实现的 PEI 把原来的基本块拆分为尺寸更小的基本块(PEI 成为所在基本块的尾指令)，PSMI 的调度被限制在这些单个基本块内部。同时基本块尺寸的减小也使编译器在单个基本块内无法发掘更好的 ILP。如何能在打破精确异常相关依赖进行指令调度的同时又保证 Java 精确异常要求，是将现有指令调度算法应用于 Java 并大大提高 ILP 和代码执行效率的关键。本文提出一种保证 Java 精确异常的指令调度技术，它允许指令调度打破 Java 精确异常要求最大程度地发挥作用，并在异常发生时通过一定的机制确保精确异常要求不被破坏。实验结果显示了该算法为基准程序带来的明显加速并证明了算法的有效性。

3 Java 精确异常与指令调度算法的统一

该部分描述了如何在保证精确异常要求的同时打破各种精确异常相关依赖进行指令调度，达到两者的统一。

3.1 基本定义与转换

对于 PEI p(被命名为 p 的 PEI)，定义其异常处理单元可见程序状态集合 $EHVV(p)$(Exception Handler Visible Variables)为包含与 p 潜在抛出的异常对应的异常处理单元入口处所有

作者简介： 张 垚(1983－)，男，硕士研究生，主研方向：系统软件
收稿日期： 2008-07-24 **E-mail：** 052053011@fudan.edu.cn

可见变量的集合，这些变量构成了该异常处理单元入口的可见程序状态。使用文献[1]中提出的计算异常处理单元入口活跃变量的方法计算 *EHVV*(*p*)。对某个修改程序变量 *v* 的 PSMI*v* 而言，当其从位置 *i* 被调度到位置 *j* 时，定义它的被污染 PEI 集合 *PPEIS*(Polluted PEI Set)包含所有同时满足下列条件的 PEI *p*：(1)变量 *v* 属于 *EHVV*(*p*)；(2)*p* 的位置位于 *i* 与 *j* 之间；(3)当顺控制流调度 PSMI*v*(即 *i*<*j*)时，*p* 按原始程序执行顺序应在 PSMI*v* 之后执行(调度后却先于 PSMI*v* 执行)。当逆控制流调度 PSMI*v*(即 *i*>*j*)时，*p* 按原始程序执行顺序应在 PSMI*v* 之前执行(调度后却后于 PSMI*v* 执行)。对于 *PPEIS* 中的所有 PEI 而言，当其抛出异常时，其对应的异常处理单元入口的可见程序状态都会因指令调度而被破坏。

对于 Java 编译代码中传统的异常检测-抛出-处理方式，做了如图 1 所示的转换：插入异常预处理块 EPBB(Exception Preprocessing Basic Block)。当位于基本块尾部的 PEI 异常检测失败时，控制流首先进入 EPBB 并执行其中的代码，随后执行 EGTB 代码抛出异常。某些 Java 编译代码通过捕获执行时产生的硬件异常来隐式地捕获空指针访问和除零 2 种异常，通过插入显示的异常检测来判断指针与除数是否为零，可将隐式异常检测转变为能够被转换后的异常检测-抛出-处理方式统一处理的形式。

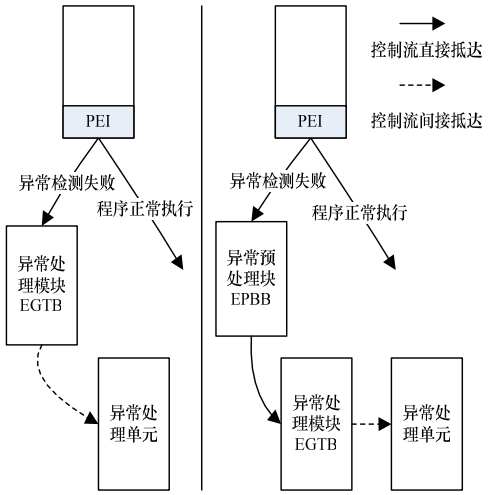

图 1　对异常检测-生成-处理方式的转换

3.2　指令调度与程序可见状态的统一

针对非 PSMI 的调度和 PSMI 的基本块内调度不会破坏异常处理单元入口可见的程序状态，同样打破精确异常相关依赖进行基本块间的 PSMI 调度。若根据调度计算出的 *PPEIS* 为空，此调度也不会破坏异常处理单元入口的可见程序状态，所以下文中只考虑 PSMI 被调度且对应的 *PPEIS* 非空的情况。PSMI 根据其修改的程序变量驻留位置的不同分为修改驻留在寄存器中变量的 PSMI 与修改驻留在内存中变量的 PSMI 2 种类型。

3.2.1　顺控制流调度的处理

打破程序状态依赖将某段指令顺控制流调度，如果被调度的指令中存在某 PSMI，并且与其对应的 *PPEIS* 非空，那么对属于该 *PPEIS* 的每个 PEI *p*，当 *p* 抛出异常时，本该在 *p* 之前执行完毕的这个被调度的 PSMI 此刻实际上还没有被执行，其修改结果对 *p* 的异常处理单元不可见，从而破坏了 *p* 对应的异常处理单元可见程序状态。通过在 EPBB 中插入补偿代码可以有效解决这个问题。假设当前某段指令被调度，则计算其中所有 PSMI 对应的所有 *PPEIS*，对于所有存在于这些 *PPEIS* 中的 PEI，将当前被调度的指令分别插入它们各自对应的 EPBB 中。EPBB 中的补偿指令按其出现在原始程序中的先后顺序排序，由此确保所有应该在抛出异常前完成却因调度而未被执行的代码在 EPBB 中被依次执行，进而保证了异常处理单元的可见程序状态的正确。此方法对 2 种类型的 PSMI 都有效。

3.2.2　逆控制流调度的处理

打破程序状态依赖逆控制流调度某 PSMI*v*，若根据调度计算出的 *PPEIS* 非空，则对该 *PPEIS* 中的每个 PEI *p* 而言，当 *p* 抛出异常时，*v* 的值会被 PSMI*v* 提前修改进而破坏了精确异常对程序状态一致性的要求。这里提出一种称为变量分裂-合并的方法，可以有效解决这个问题。

变量分裂-合并的本质是在 PSMI*v* 的被调度的新位置区分它所修改的变量 *v* 与 *p* 对应的异常处理单元入口可见的变量 *v*，即将其分裂为相互独立的 2 个变量，并称前者为 *v*'(PSMI*v* 在新位置变为 PSMI*v*')。由于 *v*'不再属于 *EHVV*(*p*)，因此 PSMI*v*'不影响 *p* 对应的异常处理单元入口可见的 *v*。为表示 *v* 与 *v*'代表不同的变量，可以为 *v*'分配新的位于堆栈上的内存空间，以达到变量分裂的目的。在 PSMI*v* 原来位置和 PSMI*v* 原来所在基本块结束位置之间用 *v*'替换 *v*，并在基本块末尾(跳转指令之前)将变量合并，将 *v*'赋值给 *v*，使后续执行能够及时得到 *v* 的正确值，在合并之后 *v*'的生存期结束。对于修改驻留在寄存器中变量的 PSMI*v* 而言，虽然 *v*'被分配了新的栈上空间，但在实际使用中只要将 PSMI*v*'修改的结果放在与 *v* 所驻留的寄存器不同的其他寄存器中即可，变量合并时进行寄存器拷贝操作。若发生寄存器溢出(register spill)，再将 *v*'的值存入新的栈上空间，并在需要时从该空间载入 register fill(如果变量分裂只将被调度的 PSMI*v* 做简单的寄存器重命名，则发生寄存器溢出时，它的修改结果会被错误地写回 *v* 所在内存，从而破坏程序状态)。当逆控制流调度的是修改驻留在内存中变量的 PSMI*m* 时，变量分裂-合并会将原来的一次内存写操作变为 2 次，这样处理没有任何意义，因此，规定不允许 PSMI*m* 进行这样的逆控制流调度。

当 PSMI*v* 从原始位置 *i* 逆控制流被调度到位置 *j* 时，算法步骤如下(定义 PSMI*v* 原来所在基本块为当前基本块)：

(1)根据调度计算 *PPEIS*，若集合为空则结束。

(2)分裂变量 *v*，生成 *v*'并将 PSMI*v* 替换为 PSMI*v*'，将位置 *i* 到当前基本块末尾之间所有对 *v* 的使用和定义变为对 *v*'的使用和定义。若其中存在把 *v*'值存回主存的操作，则其目的地址仍为 *v* 的内存地址；当 *v* 在当前基本块出口活跃时，生成变量合并指令 *v*=*v*'，并将其插入到该基本块的末尾(跳转指令之前)。

(3)对 PSMI*v* 的依赖前驱，若依赖关系为输出依赖或反依赖，则将其转移到 *v*=*v*'上；若依赖关系为流依赖，则转移到 PSMI*v*'上。

(4)对于 PSMI*v* 的依赖后续：1)若依赖关系为输出依赖并且后续指令不在当前基本块中，则将其转移到 *v*=*v*'上；若后续指令在当前基本块中，则建立该指令和 *v*=*v*'之间关于 *v*'的流依赖，同时在 PSMI*v*'与该指令之间建立关于 *v*'的输出依赖。2)若依赖关系为流依赖并且后续指令不在当前基本块中，则将其转移到 *v*=*v*'上；若后续指令在当前基本块中，则在 PSMI*v*'与该指令之间建立关于 *v*'的流依赖。

(5)删除 PSMI*v* 及其相关的所有依赖。

变量分裂确保被调度的 PSMI*v*'的执行结果不会影响异常处理单元入口可见变量 *v*，从而在相应异常抛出时保证了程序状态的一致性。变量合并使 PSMI*v*'的执行结果能够及时准确地被后续执行使用，保证后续程序的正确执行。更新相关依赖在保证依赖关系正确性的同时，也使对合并指令的调度成为可能。

3.3 指令调度与异常抛出顺序的统一

PEI 因调度在控制流某个方向上产生的移动可以认为是该 PEI 跨越的指令在控制流相反方向上的移动。因此，若对某 PEI 的调度没有因跨越其他 PEI 而打破异常顺序依赖，则可以使用 3.2 节中的方法保证精确异常要求；否则，PEI 之间先后执行顺序的改变可能破坏异常抛出顺序的正确性，通过推迟被调度 PEI 所对应的异常的生成抛出并使用补偿代码可以解决这一问题。

定义指令 *i* 的原始位置是 *i* 在程序还未做任何指令调度前所在的位置。如果 *i* 是逆控制流调度某 PSMI*v* 所产生的 PSMI*v*'，则它的原始位置为该 PSMI*v* 的原始位置，对应的变量合并指令 *v*=*v*'的原始位置为其生成后被插入的位置。当逆控制流调度 PEI *p* 跨越包括某些 PEI 的一段指令时，把同时满足以下条件的所有指令 *i* 按照其原始位置的先后不做任何调度地插入 *p* 所对应的 EPBB 中：(1)*i* 的原始位置小于 *p* 的原始位置(*i* 应先于 *p* 执行)；(2)*i* 的当前位置大于 *p* 的当前位置(*p* 因调度会先于 *i* 执行)。所有应该在 *p* 之前执行却被调度到 *p* 之后执行的代码(包括那些被 *p* 跨越的 PEI)都按其在原程序中的先后顺序出现在补偿代码中。在 *p* 抛出异常并执行对应 EPBB 中的补偿代码时，*p* 所对应的异常还未生成并抛出。假设 *e* 是在执行 EPBB 中补偿代码时抛出的第一个异常，由于补偿代码本身不做任何指令调度且所有指令按原始执行的先后顺序执行，则 *e* 就是按精确异常要求必须被真正抛出的异常，将其抛出即可；若补偿代码中没有任何 PEI 抛出异常，则抛出 *p* 对应的异常。特别需要注意的是，如果 *p* 对应的 EPBB 中已经存在 3.2.1 节所描述的补偿代码，则必须先删除这些代码，再插入新的补偿代码，因为后者已经包含前者。

当顺控制流调度 PEI *p* 跨越包括某些 PEI 的一段指令时，可以认为这些指令被逆控制流调度到 *p* 之前，首先运用 3.2.2 节中提到的方法进行必要的变量分裂-合并，然后对这段指令中包含的每个 PEI 运用之前提到的方法生成补偿代码，由此既保证程序可见状态在 PEI 异常处理单元入口的正确性，又确保了正常的异常抛出顺序。

3.4 指令调度顺序的影响

对某条已被调度指令的再次调度可能影响之前调度所产生的补偿代码和变量分裂-合并操作的正确性。同样，对某条指令的调度也可能影响之前已被调度的某些 PEI 的补偿代码的正确性(比如某条 PEI 先被调度到某 PSMI 之前，随后该 PSMI 又被调度到该 PEI 之前)。为避免算法过于复杂并消除调度顺序的影响，在指令调度全部完成的基础上根据指令间相对位置的改变进行变量分裂-合并和补偿代码的生成。最后，删除所有空 EPBB，并将对应的没有被调度影响的 PEI 的异常检测-抛出-处理恢复成传统方式。

4 实验数据

实验选取的 5 个 Java 基准程序都具有如下特点：(1)程序热点中存在大量异常检测；(2)冗余异常检测消除不能有效消除这些异常检测。这些特点极大地限制或阻碍了指令调度算法的应用，进而影响程序执行效率。

笔者在 Sun Hotspot Server Compiler 1.4.2 中实现了本文提出的算法，使编译器能够打破精确异常相关依赖进行指令调度，进而充分发掘代码的 ILP 并提高代码执行效率。实验测试平台为：双 1.4 GHz 安腾 2 处理器，1 GB 主存，RedHat Linux AS2.1。表 1 给出了不同基准程序在应用本文算法后获得的净加速比，进而证明了算法的有效性。

表 1 基准程序获得的净加速比

基准程序	净加速比/(%)
Linpack	55
FFT	38
SparseCompRow	53
LU	72
SOR	90

5 结束语

本文提出的保证 Java 精确异常的指令调度技术允许指令调度算法打破精确异常要求的限制，最大程度地发挥作用来提高 ILP 和代码执行效率。同时通过补偿代码和变量分裂-合并等方式保证经优化调度的代码在异常发生时不会破坏 Java 精确异常要求，实验结果表明算法达到了很好的效果。

参考文献

[1] Sun Microsystems. The Java Language Specification[EB/OL]. (2007-08-20). http://java.sun.com/docs/books/jls/.

[2] Gupta M, Choi J D, Hind M. Optimizing Java Programs in the Presence of Exceptions[C]//Proc. of the 14th European Conference on Object-oriented Programming. Cannes, France: Springer, 2000: 422-446.

[3] Arnold M, Hsiao M, Kremer U, et al. Instruction Scheduling in the Presence of Java's Runtime Exceptions[C]//Proc. of the 12th International Workshop on Languages and Compilers for Parallel Computing. CA, USA: Springer, 2000: 18-34.

[4] Muchnick S S. Advanced Compiler Design and Implementation[M]. CA, USA: Morgan Kaufmann Pub, 1997.

编辑 顾逸斐