利用 Java即时编译器自动外提循环中的同步操作

张 昱[1,2],史成荣[1,2]

[1] (中国科学技术大学 计算机科学与技术学院,安徽 合肥 230027)

[2] (安徽省计算与通信软件重点实验室,安徽 合肥 230027)

E-mail: yuzhang@ustc.edu.cn

摘 要: 同步开销是影响并行程序性能的一个重要方面,如果同步操作出现在循环中,将会使这种影响进一步扩大.为了降低循环中同步操作的开销,本文提出一种利用即时编译器外提 Java程序中循环内同步操作的优化算法,并在实际的 Java虚拟机中实现.该算法在保证程序语义不变的前提下,大量减少运行时实际执行的同步操作数量,降低同步开销,并能保证外提变换后同步代码块不会太大而降低程序的并发度.实验结果表明该算法能提高程序的整体性能,并且不降低程序的可扩放性.

关 键 词: 即时编译器;同步优化;循环;代码外提

中图分类号: TP311 **文献标识码**: A **文 章 编 号**: 1000-1220(2009)12-2414-06

Automatically Hoisting Synchronization Operations from Loops with Just-in-time Compiler for Java

ZHANG Yu[1,2], SHI Cheng-rong[1,2]

[1] (School of Computer Science & Technology, University of Scince & Technology of China, Hefei 230027, China)

[2] (Anhui Province Key Laboratory of Software in Computing and Communication, Hefei 230027, China)

Abstract: Synchronization overhead is an important aspect influencing performance. If synchronization operations are in loops, the influence to performance will be enlarged. To reduce the overhead of synchronization operations in loops, an algorithm on hoisting synchronization operations from loops with Just-in-time Compiler for Java is put forward and implemented in an existing Java VirtualMachine. The algorithm can greatly decrease synchronization operations at run-time so that the synchronization overhead is reduced, and can ensure the concurrency of the program not be reduced without changing the semantics of the program. The experiment results show that the algorithm can improve the whole performance and not reduce the scalability.

Key words: just-in-time compiler; synchronization optimization; loop; loop hoisting

1 引 言

Java语言[1]支持线程间同步的语言级结构是用监视器(monitor)实现的 synchronized语句和 synchronized方法. Java中每个对象有对应的一个监视器,线程可以对其加锁(lock)或解锁(unlock);每次只有一个线程可以持有一个监视器上的锁,其他试图对这个监视器加锁的线程都将被阻塞直到它们获得这个监视器上的锁.虽然同步可以保证 Java程序中各线程对共享数据的正确访问,但是同步也带来许多开销,从而影响 Java虚拟机(JVM)[2]运行 Java程序的性能.

为降低同步开销,除改良同步原语的实现[3,4]之外,一些研究者从程序分析的角度提出一些解决算法:例如,基于逃逸分析[5-9]分析出局部于线程的对象,消除在这些对象上的同步操作;对作用在同一监视器 O上的形如"lock(O)…lock(O)…unlock(O)…unlock(O)"的嵌套同步或者是形如"lock(O)…unlock(O)…lock(O)…unlock(O)"的顺序同步,通过变换和削减同步操作[10-13]来消除重复的同步开销.然而,这些算法多采取保守的静态程序分析技术,对循环中的代码只分析一次,因此丢失了对循环内同步操作进行优化的一些机会.

本文重点研究对 Java程序中循环内同步操作的优化改进,其基本思想是将作用在循环不变的监视器上的一些同步操作外提出循环以减少重复的同步开销,外提所做的程序变换必须保证程序语义不被改变.基于这种思想,本文提出一种基于即时编译器的同步操作外提变换算法,给出了可以外提的同步操作应具备的条件,重点分析和解决了在优化变换中因多态、动态类载入、异常、JVM及其内存模型规范等导致的安全和性能问题.笔者在开源的 Java SE项目 Apache Harmony[14]中实现了本文的算法,并用 specjbb2005进行了测试,实验结果表明所提算法能将 specjbb2005的吞吐量提高 1%-3%.

2 预备知识

JVM主要通过即时编译器编译待执行方法的字节码得到本地码,然后再运行本地码,其中热点方法会多次被不同优化级别的即时编译器重编译.即时编译器一般以多个遍(pass)

收稿日期: 2008-07-22 基金项目: Intel公司研究基金项目资助;国家自然科学基金项目(60673126)资助. 作者简介:张 昱,女,1972年生,博士,副教授,CCF会员,研究方向为程序设计语言理论与实现技术,特别是并行语言设计、编译,并行程序分析和验证等;史成荣,男,1981年生,硕士研究生,研究方向为程序分析与同步优化.

组成的流水线形式依次对当前编译的方法进行字节码载入、翻译到中间表示(IR)、在 IR上进行优化变换,最后生成本地码.本文的同步外提优化最终即实现为流水线上的一个遍.

在字节码级,语言级的 synchronized语句变换成显式含有 monitorenter(加锁)和 monitorexit(解锁)指令的指令序列;而 synchronized方法则是在其常量池条目中设置有同步标志, invokevirtual虚方法调用指令隐式地根据被调用的方法是否是同步的,再做相应的处理.本文只考虑对显式同步操作的外提变换.为便于描述,这里统一用 lock(O)和 unlock(O)表示对监视器对象 O的加锁和解锁.

Java语言的同步模型是与异常机制结合的,对于程序中的每个同步块(不论是对同步方法的虚调用,还是包含在一对显式同步操作中的代码), JVM 都要确保其有相应的异常处理来支持在同步块运行出现异常时能释放锁.多态、动态类载入、异常、Java内存模型等特性使得对同步操作的外提变换远比传统的代码外提变换(即把循环不变运算放到循环的前面)要复杂得多.

同步操作的外提变换本质上属于循环优化,它是以一个方法的 IR上的控制流图(简称流图)和流图中的循环为基础开展的.流图是具有单源点的有向图,图中每个节点对应 IR中的一个基本块,而节点间的有向边则表示基本块间可能的控制流向.通过分析流图中节点的控制关系,可以找出流图中的所有循环.每个循环由流图中的一组形成环的节点组成.流图中的所有循环按其间的嵌套关系形成一棵循环树,其中,除根节点外,每一节点对应流图中的一个循环,而流图中所有顶层循环都是根节点的孩子.

3　循环内可以外提的同步操作所具备的条件

定义 1　一个循环含有潜在可外提同步操作的必要条件是:如果它至少存在一对相配对的 lock(O)操作和 unlock(O)操作(简称一对同步操作),且这对操作作用的监视器 O是循环不变的.

下面分析满足定义 1的循环所具有的几种情况.为简便起见,这里引入伪流图(节点不一定对应一个基本块,边主要反映正常的控制流向,且忽略同步块所引出的异常边)来表示代码片段的控制流.

情况 1　循环中包含一对作用在循环不变的监视器上的同步操作,

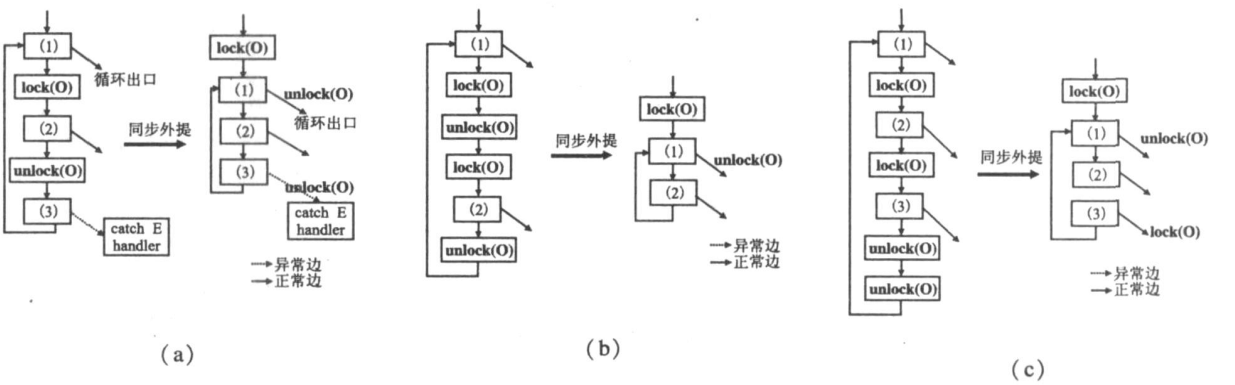


图 1　含潜在可外提同步操作的循环

Fig. 1　Loops containing hostalbe synchronigation operation

且无其他同步操作或者其他同步操作都作用在其他监视器上并位于这对同步操作内.

此时将这对同步操作外提到循环外.图 1(a)示意外提前后的伪流图,其中 lock(O)单独成为节点被提到原循环入口节点(1)之前, unlock(O)则单独成为节点加到每个退出循环且弧尾节点不在锁定区域中的边上(详见第 4节和 5.2节).

情况 2　循环中顺序包含多对作用在同一循环不变的监视器上的同步操作.

如图 1(b)所示,此时在外提变换时,可以删去循环内的这些同步操作,而在循环入口之前增加 lock(O)节点,并按在情况 1中介绍的方法对循环的每个出口进行变换处理.

情况 3　循环中包含多对相互嵌套的同步操作,且它们都作用在同一个循环不变的监视器上.

如图 1(c)所示,此时在外提变换时,除参照情况 2中的做法,还要进一步保证各个出口的锁深度(即某监视器被加锁的次数)是否正确.例如,图 1(c)中为保证锁深度的正确性,变换时在节点(3)的循环出口边增加了 lock(O)节点.

定义 2　当一个循环 L满足定义 1中的条件时,若它同时还满足以下条件之一,则 L内的同步操作是不可外提的(具体原因参见第 5节):

条件 1　L含有作用在多个监视器上的同步操作,且这些同步操作不出现在 L内的同步块中;

条件 2　L含有内嵌循环,且内嵌循环不出现在 L内的同步块中;

条件 3　L含有对 volatile变量的读或写指令,且这些指令不出现在 L内的同步块中;

条件 4　L含有方法调用指令,且这些指令不出现在 L内的同步块中;

条件 5　共享变量作为循环控制变量.

4　同步操作外提变换算法

同步操作外提变换算法是以方法的流图 CFG和循环树

LT为输入的,以变换后的流图 CFG为输出.整个算法分可外提同步操作信息收集和 CFG变换两个阶段.收集的信息包括:

(1) 表 LMList元素为二元组〈loop, mon〉, loop表示一个可外提同步操作的循环,mon是 loop中可外提同步操作作用的监视器;

(2) 表 EMList元素是二元组〈exitEdge, mon〉, exitEdge是某循环的退出边,mon是可外提出该循环的同步操作作用的监视器;

(3) 集合 SyncInstSet:可外提同步操作集.

两阶段的算法描述分别见算法 1和算法 2.

算法 1. 可外提同步操作的信息收集算法

```
BEGIN
unhoistLoopSet = {}; //存放不可外提的循环集
for each node node in CFG do{
//取 node所属的循环层数 num和最内层循环 loop
  <num, loop> = getLoopsForNode(LT, node);
  if ( num <=0) continue; // node不在循环中
  if (loop in unhoistLoopSet) continue;
  st1 = getLoopEntryLockState(loop);
  if (num >1) {
    ploop = getParentLoop(loop); //取上一层循环
    stp = getLoopEntryLockState(ploop);
    st2 = getNodeEntryLockState(node);
    if (stp == st2) //定义 2中的条件 2
      insert loop into unhoistLoopSet;
  }
  for each instruction inst in node do{
    if(inst is shared variable access instruction)
      insert loop into unhoistLoopSet; //条件 5
    else if (inst is sync instruction) {
      st2 = getInstLockState(instruction);
      if(st1 == st2 && loop contains other sync instruction)
        insert loop into unhoistLoopSet; //条件 1
      else if(loop not contains other sync instruction)
        insert inst into SyncInstSet;
    } else if(inst is volatile access, method call) {
      st2 = getInstLockState(inst);
      if (st1 == st2) //定义 2中的条件 3和 4
        insert loop into unhoistLoopSet;
} } }
for each instruction sync in SyncInstSet do{
  loop = getInnerLoop(LT, sync); //取所在最层循环
  if (loop in unhoistLoopSet)
    remove sync from SyncInstSet;
  else insert〈loop, monitor on sync〉into LMList;
  }
  for each edge exitEdge of CFG do
  if (exitEdge is an exit edge of a loop in LMList)
    insert〈exitEdge, mon〉into EMList;
END
```

算法 2 CFG变换算法

```
BEGIN
for each〈loop, mon〉in LMList do {
  start = getLoopEntry(loop); //取 loop的入口节点
  create a CFG node newnode;
  insert instruction "lock(mon)" into newnode;
  add an edge from newnode to start;
  for each inedge inEdge of start do{ //处理各入边
    if (inEdge is a back edge) continue; //是回边
    change head of inEdge into newnode; //改弧头
  } }
  for each〈exitEdge, mon〉in EMList do {
    //取 exitEdge的弧头节点被 mon锁定的次数
    depth = getLockDepth(exitEdge);
    if (depth == 0)
      insert instruction "unlock(mon)" on exitEdge;
    else if (depth > 1)
      insert (depth-1) "lock(mon)" instructions on exitEdge;
    }
    for each instruction sync in SyncInstSet do
      remove sync instruction from CFG;
END
```

5 正确性和性能分析

同步操作的外提变换会锁定一些原本未被锁定的代码.外提变换必须确保锁定原本未被锁定的代码不会改变程序的语义,还需要保证程序的性能.本节给出本文采用的策略并通过例子来分析说明.

5.1 避免死锁

两个线程以不同次序对多个监视器加锁可能导致死锁,因此本文所提算法不对含有作用在多个监视器上的同步操作且这些同步操作不出现在同步块内的循环进行外提.循环中若出现如方法调用、volatile变量的访问、共享变量作为循环控制变量等情况,也都有可能因同步操作外提而导致死锁.为避免产生新的死锁,所提算法采取保守做法,不外提包含上述情况的同步操作,即总结出前面定义 2中的各个条件.下面分别举例来分析说明.

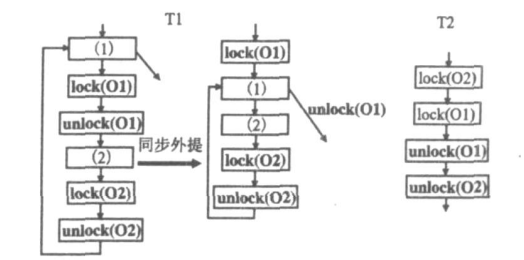


图 2 多个监视器上的同步操作可能导致死锁的情况

Fig. 2 Case that may cause deadlock: synchronization operations on different monitors

图 2展示了含有作用在多个监视器上的同步操作的循环发生死锁的情况.在外提线程 T1中的 lock(O1)和 unlock(O1)之前,T1与 T2的并行执行不会发生死锁.但外提之后,

当 T1准备执行 lock(O2)而 T2已执行完 lock(O2)但尚未执行 lock(O1)时, T1、T2会因为相互等待对方释放持有的锁而进入死锁状态.

当循环包含因动态类载入和多态等导致的无法解析的方法调用时,将无法预知实际执行的方法.如果实际执行的方法包含的同步操作与循环中的同步操作不是作用在同一个监视器上,则与图 2类似,也可能导致死锁.

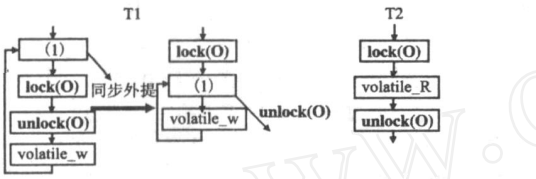


图 3　包含 volatile操作的循环可能导致死锁的情况

Fig. 3　Case that may cause deadlock: volatile operatios existing in loop

Java内存模型[2]不仅保证 volatile变量间的可见性,还保证 volatile变量与其他变量间的可见性,因此 volatile变量常用作表明某组操作已完成的守卫.如图 3,假定 T2中 volatile_W操作负责写一个 volatile变量,而 T1中 volatile_R操作对该 volatile变量以检查 T2中的 volatile_W操作是否已执行,若没有执行则等待,否则继续执行.在外提 T1中的 lock和 unlock之前, T1与 T2的并行执行不会发生死锁;但外提之后,若 T1已执行完 lock并等待 T2完成对 volatile变量的写,而 T2正准备执行 lock时,便进入死锁状态.

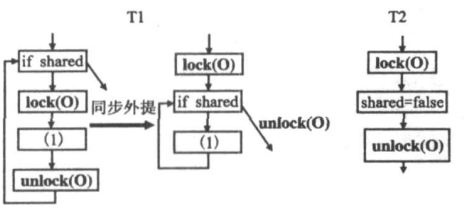


图 4　循环控制变量为共享变量可能导致死锁的情况

Fig. 4　Case that may cause deadlock: loop variable is a shared variable

如果循环控制变量是一个共享变量,外提同步操作也有可能导致死锁.图 4中, T1包含一个以共享变量 shared (shared是某类的静态域,初始为 true)为循环控制变量的循环.外提 T1中的 lock和 unlock之前, T1与 T2的执行不会发生死锁;但外提之后,当 T1执行了 lock而未执行 unlock,如果此时 T2还未执行 lock操作,则 T1会一直在循环中等待 T2将 shared变量改成 false,而 T2则因未获得锁而一直等待 T1退出循环并释放锁,从而进入死锁.

5.2　异常处理

变换阶段会在退出循环的边上插入 lock或 unlock等指令来保证同步外提不改变程序语义以及遵循 JVM 规范的规定.对于正常的循环退出边,如图 5(a)中节点 (1)到节点 (3)的边,只需创建包含 unlock(O)指令的新节点,再将新节点插入到该退出边两端的节点之间即可.但是对于异常退出边,如图 5(b)中从节点 (1)到节点 (3)的边,由于只有在发生特定的异常时才会走特定的异常边,故只有发生类型为 E的异常时控制流才会从节点 (1)沿着异常边走向异常处理节点 (3).如果用图 (a)的方式插入 unlock操作就会改变程序的语义,这时就必须创建一个新的异常处理节点,在新节点中插入一条 catch E指令,以捕获节点 (1)发生的类型为 E的异常,然后才插入 unlock指令,最后还要在新创建的节点中插入一条 rethrow E指令,将节点 (1)中发生的类型为 E的异常重新抛出,以保证执行完 unlock操作后,控制流能转到节点 (3),维持程序原来的语义.此外,由于外提作用在同一监视器上的嵌套同步,为保证外提后锁深度的正确性,在一些循环退出边上新增的节点中还需要插入 lock指令.

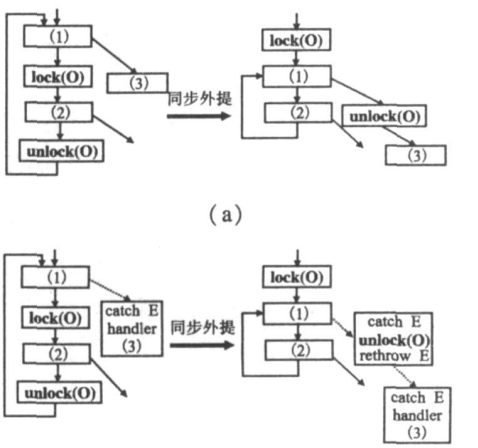


图 5　正常边和异常边上插入 unlock 操作

Fig. 5　Insert "unlock" operations on normal edges and exceptional edges

5.3　性　能

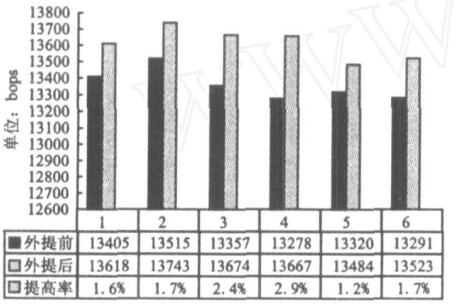
同步优化必须考虑同步开销和并发度的权衡.降低同步开销的同时往往会降低程序的并发度,为此必须在这两者间有好的折中.本文的算法将循环体内的多个同步操作合并成一个,大大减少了同步操作的数量,但同时也粗化了同步范围,降低了程序的并发度.为避免这种粗化影响性能,算法并不将那些原本不在同步块内的循环和方法调用进行粗化.因为循环的运行时间一般都较长,而方法调用,尤其是无法解析的方法,其长度往往无法预测.

同步操作一般用于保护多线程对共享数据的访问.因此在多数情况下,有同步代码块的代码长度都比较短.基于这样的假设,对于同步块中的循环或方法调用等,可以包含在粗化后的同步块中.

6　实验结果与分析

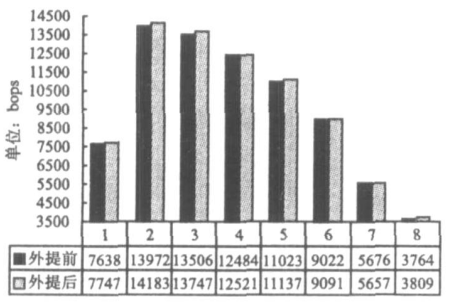
笔者在开源的 Apache Harmony(一个 JVM 实现)的即时编译器中以优化遍的形式实现了该算法,记为 syncopt遍. Apache Harmony的即时编译器在编译一个方法时,经常会利用代码插桩等手段在字节码或中间表示上插桩一些代码以在运

行时收集信息,并利用这些信息决定是否用更高优化级别的即时编译器对当前方法重新编译.为了保证较快的启动速度,虚拟机在第一次执行一个方法时一般不会用较高优化级别的编译器对其进行编译,根据在运行时收集到的信息,如果发现当前方法是热点方法才会启动较高优化级别的编译器对其进行编译,包含循环特别是迭代次数多的循环的方法经常被认为是热点方法,需要用更高优化级别的即时编译器对其重编译,因此,虽然外提循环中的同步操作作为一种较高级别的优化,与其他高级别的优化一样,在方法首次被执行时不会被启动,但是它被启动的概率相对其他级别的优化会高很多.



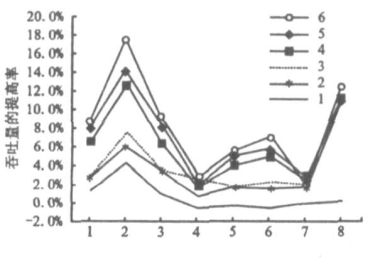
	1	2	3	4	5	6
■外提前	13405	13515	13357	13278	13320	13291
□外提后	13618	13743	13674	13667	13484	13523
□提高率	1.6%	1.7%	2.4%	2.9%	1.2%	1.7%

(a)



	1	2	3	4	5	6	7	8
■外提前	7638	13972	13506	12484	11023	9022	5676	3764
□外提后	7747	14183	13747	12521	11137	9091	5657	3809

(b)



(c)

注:bops: business operation per second

图6 同步外提前后 specjbb2005 的吞吐量对比

Fig. 6 Comparison of specjbb2005 throughputs before and after hoisting

本实验使用的平台为 AMD Athlon(tm) 64 * 2 Dual Core Processor 3800 + 2.0GHz, 896MB 内存,操作系统为 Windows XP.测试用例为 specjbb2005.

我们分别用不含 syncopt遍以及含 syncopt遍的 JVM 对 specjbb2005进行了多次运行测试,结果见图 6.图 6(a)列出了 6组运行的总体吞吐量,深色柱体和数据表的第 1行是外提前 (即无 syncopt遍)的吞吐量,浅色柱体和数据表的第 2行为外提后 (即有 syncopt遍)的吞吐量,数据表第 3行是外提后吞吐量的提高率.从中可见,算法对吞吐量的提高可达到 1%-3%.

图 6(b)为 (a)中第 4组的明细情况,即线程数为 1到 8时同步操作外提前后的吞吐量,横坐标表示运行时的仓库数,即启动的线程数.从中可见,除线程数为 7时外提后的吞吐量略有下降,其它情况下都有不同程度的提高.

图 6(c)给出了同步操作外提后 6次运行中线程数分别为 1到 8时吞吐量提高率的变化趋势,横坐标是线程数,纵坐标是吞吐量的提高率.可以看出从线程数为 2起,随线程数的增加,吞吐量的提高率有下降的趋势,这表明在外提同步操作后,由于同步块增大致使程序的并发度下降;但是吞吐量的提高率基本上都在 0以上.由此可见,虽然程序的并发度略有下降,但外提同步操作后减少的同步开销远超过并发度降低对性能的影响.

7 相关工作

一些研究者通过改良同步原语的实现来降低 Java虚拟机同步开销,将对象的一部分用来存放锁信息能提高锁的效率,[3]利用对象中的 24个位保存锁信息,简单、存储开销小、容易维护,对无竞争锁的操作可以加快速度,但需要比较多的原子操作;[4]基于线程局部性 (thread locality),对一个给定的对象上的锁,一般都会由一个特定的线程来申请和释放,为一个锁保留一个线程,该线程申请该锁就非常快,不需要任何原子操作.大多数研究者从程序分析的角度,以静态程序分析为主,结合运行时信息,分析出程序中的冗余同步操作并将其删除,[12]在 Intel的 ORP(Open Runtime Platform)上实现了一个逃逸分析框架并运用于同步消除优化中,并提出了一些传播算法,如 bottom-up和 up-down算法等,用于在对象上传播逃逸状态;[8, 11]的逃逸分析算法引入了一个称为连接图 (Connection Graph)的程序抽象用以确立对象、对象引用之间的可达性.逃逸分析被映射到连接图上的可到达问题.连接图可以很容易地为每个方法建立相关结构,并能在不同的方法调用上下文中被利用以避免重复计算. PowerPC/AIX体系结构平台下的 IBM 高性能静态 Java编译器 (HPCJ)中实现了这类算法,并运用其分析结果进行栈中分配与同步消除的优化;[9]不仅对静态域不可到达的对象作了同步消除,还对整个程序中只有一个线程访问的静态域可到达对象也作了同步消除;[13]引入了一种基于等价类 (equivalence-class-based)的表示用以确立对象、对象引用之间的关系,目标是仅保留被多于一个线程同步的对象上的同步操作;[5]对各个线程进行时序分析,对于被多个线程访问并且在多个线程中被同步的对象,如果不同线程中在该对象上的同步操作存在时间上的先后顺序,这样的同步操作也可以删除;[6]提出一套同步变换的方式,是编译器进行同步优化的基础;[7]提出 2种降低锁开销的技术:数据锁粗化 (data lock coarsening)和计算锁粗化 (computation lock coarsening);[10]提出一种动

态反馈技术,使系统能自动评价相同代码的不同同步优化策略,然后为当前环境选择最优的优化策略;[14]在即时编译器中分析 Java程序的中间表示,查找不必要的同步指令,包括删除嵌套同步操作的内层同步操作以及合并同一对象监视器上的多个连续同步操作.

上面提到的这些工作都没有涉及对程序性能影响最大的循环的优化,本文的算法专门针对循环进行同步优化,而且针对 Java虚拟机中的即时编译器,需要考虑编译时间以及虚拟机规范和 Java特性.

8 结束语

本文提出的基于 JVM 即时编译器的同步外提优化算法是过程内的.它在保证变换前后程序语义的基础上,能大量减少运行时实际执行的同步操作的数量,同时能在降低同步开销和降低程序并发度之间进行良好的折中.它不仅能方便地与其他同步优化算法集成在一起使用,而且能适用于多种架构的编译器,如基于调用图等全局信息的编译器、诸如 Harmony的逐方法编译的流水线编译器,等等.

References:

[1] James Gosling, Bill Joy, Guy Steele, et al. Java™ language specification, third edition[S]. Addison Wesley Professional, June 14, 2005.

[2] Tim Lindholm, Frank Yellin. The Java™ virtual machine specification, second edition[M]. Prentice Hall PTR, April 1999.

[3] Bacon D, Konuru R, Murthy C, et al. Thin locks: featherweight synchronization in Java[C]. PLDI'1998, Montreal, June 1998.

[4] Kiyokuni Kawachiya, Akira Koseki, Tamiya Onodera. Lock reservation: Java locks can mostly do without atomic operations[C]. OOPSLA'2002, Seattle, November, 2002.

[5] Wu Ping, Chen Yi-yun, Zhang Jian. Effective synchronization removal in concurrent Java programs[J]. Journal of Software, 2005, 16(10): 1708-1716.

[6] Ruf E. Improving the precision of equality-based dataflow analyses[C]. SAS'2002, Madrid, Spain, Sep. 2002, 247-262.

[7] Jong-Deok Choi, Manish Gupta, et al. Stack allocation and synchronization optimizations for Java using escape analysis[J]. ACM Transactions on Programming Languages and Systems, Nov. 2003, 25(6): 876-910.

[8] Wang Lei, Sun Xi-kun. Escape analysis for synchronization removal[C]. SAC'2006, 1419-1423.

[9] Ruf E. Effective synchronization removal for Java[C]. PLDI'2000. New York, 2000, 208-218.

[10] Pedro Diniz, Martin Rinard. Synchronization transformations for parallel computing[C]. POPL'1997, 1997, 187-200.

[11] Pedro C Diniz, Martin C Rinard. Lock coarsening: eliminating lock overhead in automatically parallelized object-based programs[C]. JPDC'1998, 1998, 49(2): 218-244.

[12] Pedro C Diniz, Martin C Rinard. Eliminating synchronization overhead in automatically parallelized programs using dynamic feedback[C]. TOCS'1999.

[13] Mark Stoodley, Vijay Sundaresan. Automatically reducing repetitive synchronization with a just-in-time compiler for Java[C]. CGO'2005.

[14] http://harmony.apache.org, 2004.

附中文参考文献:

[1] 吴 萍,陈意云,张 健.并发 Java程序同步操作的有效删除[J].软件学报,2005,16(10):1708-1716.