一个基于混合并发模型的 Java 虚拟机*

杨 博, 王鼎兴, 郑纬民

(清华大学 计算机科学与技术系,北京 100084)
E-mail: yangbo1997@yahoo.com
http://hpc.cs.tsinghua.edu.cn

摘要: 从解释执行到及时编译的转变极大地提高了 Java 程序的运行速度.但是,现有的 Java 虚拟机还有待进一步的改进.提出了一种新的 Java 虚拟机编译与执行模型——混合并发模型 HCCEM(hybrid concurrent compilation and execution model).该模型通过多线程控制方式将字节码的编译与执行过程相重叠,从而获取加速的效果.另外还给出了基于 HCCEM 的 Java 虚拟机 JAFFE 的设计方案,并就实现中的执行模式切换、异常处理以及层次线程等问题进行了讨论.实验结果表明,HCCEM 能够有效地提高 Java 程序的执行速度.

关 键 词: 混合并发;Java 虚拟机;异常处理;多线程
中图法分类号: TP311 文献标识码: A

Java 语言的平台无关性和严格的安全性检查措施以及字节码的精简性都使得 Java 语言成为网络计算最具前途的编程语言.然而,Java 与 C,C++以及 Fortran 等语言相比的低性能成为阻碍其发展的重要因素.为了提高 Java 程序的运行速度,人们提出了及时编译技术 JIT(just-in-time)[1],通过在运行时将字节码编译成本地的机器指令来提高 Java 的运行效率,从而大大缩小了与其他语言的性能差距.现在绝大多数的 Java 虚拟机中都采用了及时编译技术.

虽然及时编译是一种十分有效的技术,但是它仍然有值得改进的地方.本文提出了一种新的 Java 程序执行模式——混合并发模式 HCCEM(hybrid concurrent compilation and execution model).作为解释和及时编译的一种混合体,基于 HCCEM 的 Java 虚拟机通过编译和执行并发与重叠来提高 Java 程序在运行时的性能.本文第 1 节描述混合并发模型.第 2 节给出基于 HCCEM 的 Java 虚拟机 JAFFE 的设计方案.第 3 节就基于 HCCEM 的 Java 虚拟机实现中的一些关键问题进行讨论.实验结果在第 4 节给出.最后是本文的总结和进一步工作介绍.

1 混合并发模型 HCCEM

在一个基于 JIT 的 Java 虚拟机中,编译模块以类似于中断处理例程的方式进行工作.当一个程序在执行过程中遇到一个尚未编译的方法时,程序的执行将被暂停同时编译模块被激活,开始对该方法的编译.只有等到编译模块生成了所需的本地代码之后,程序的执行才会被恢复,所以,在基于及时编译模式的 Java 虚拟机中不存在对字节码的解释执行.

HCCEM 是在连续编译[2]思想的基础上提出来的,它与及时编译模式在以下两个方面存在不同:(1) 混合并发编译模式中的编译过程由另外一个独立的线程来控制,因此可以与程序的执行并发地进行,而不像在 JIT 中那样被间歇性地激活.所谓并发是指编译与执行可以并发进行;(2) 与 JIT 在执行模式上的单一性不同,混合并

* 收稿日期: 2000-09-01; 修改日期: 2001-04-17
基金项目: 国家自然科学基金资助项目(69873023)
作者简介: 杨博(1976－),男,陕西商州人,博士,主要研究领域为并行处理,网络计算技术;王鼎兴(1937－),男,江苏吴县人,教授,博士生导师,主要研究领域为并行处理与分布式系统;郑纬民(1946－),男,浙江鄞县人,教授,博士生导师,主要研究领域为并行处理与分布式系统.

发编译模式中保留了解释执行模块,因此能够在及时编译后执行和解释执行之间作出选择.所谓混合是指程序在运行过程中可能同时出现解释执行和编译生成的本地码执行两种状态.程序在 JIT 下的执行方式与在 HCCEM 下的执行方式的区别如图 1 所示.

(a) JIT (b) HCCEM

T: compiling①, E: native code execution②, I: interpreting③, W: waiting④.

①编译,②本地码执行,③解释执行,④等待.

Fig.1 Difference of execution mode between JIT and HCCM

图 1 程序在 JIT 和 HCCM 模式下执行方式的不同

根据混合并发模式在结构上的特点,与及时编译模式相比,它具有以下优势:

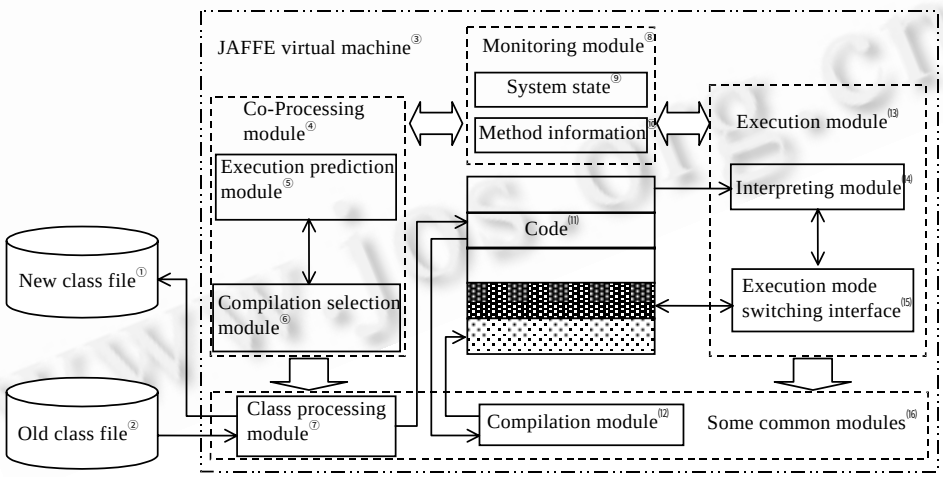
(1) 对于某些方法而言,解释执行的性能可能要好于及时编译[3].基于 HCCEM 的 Java 虚拟机由于保留了解释执行模块,可以在解释和编译后执行之间进行权衡,以选择较优的执行方式.

(2) 现有的基于 JIT 的 Java 虚拟机中采用单线程控制方式,因而无法充分利用底层平台的计算资源.随着 SMP 工作站的普及,HCCEM 的多线程控制方式能很好地支持多处理器系统.即使是在单处理器系统中,也可以利用因等待产生的处理器空闲来进行方法的编译.

(3) 由于采用了多线程控制方式,在后台进行的及时编译由一个独立于执行线程的线程来控制,消除了及时编译模式中可能因编译所造成的停顿时间过长而无法满足程序实时性要求的顾虑,所以在后台进行的编译可以花更多的时间来优化程序,从而得到性能更好的本地执行码.

2 JAFFE 虚拟机设计

我们所设计的基于 HCCEM 的 Java 虚拟机 JAFFE 的结构如图 2 所示.



①新类文件,②旧类文件,③JAFFE 虚拟机,④协同处理模块,⑤执行预测模块,⑥编译选择模块,⑦类处理模块,⑧监控模块,⑨系统状态,⑩方法信息,⑪代码,⑫编译模块,⑬执行模块,⑭解释执行模块,⑮执行模式切换接口,⑯一些公用模块.

Fig.2 Structure of JVM based HCCEM

图 2 基于混合并发编译模式的 Java 虚拟机结构图

其中各部分描述如下:

执行模块与一个传统的 Java 虚拟机类似,负责 Java 程序的实际运行.根据 HCCEM 的特点,执行模块中包含了一个解释执行子模块.这样,当在执行过程中遇到一个本地码还未准备就绪的方法时,可以在解释执行该方法

和启动及时编译这两者之间进行选择.执行模式切换接口完成不同执行方式下参数的传递和结果的返回.

协同处理模块由一个独立于执行模块的线程来控制,该模块的目的在于尽量将那些可以与程序的执行同步进行的操作(如对类文件的处理以及对字节码方法的编译)调度起来.其中的编译选择子模块根据一定的编译选择策略从待编译方法中选择一个方法进行上述的操作.由于 Java 程序的执行过程具有动态链接的特点,很多类只是在用到它时才会通过网络下载到执行端,所以本次运行中将会用到的类和方法在开始运行时是不可知的.为了尽快得到那些将被使用的方法(也就是需要编译的方法),我们在协同处理模块中设计了一个执行预测子模块,它以类似于处理器中的指令预取部件的方式工作,对正在被执行的方法的字节码进行分析,进而预测即将被用到的方法集,使得对这些方法的编译能够尽快开始.

类处理模块负责读取类文件,并在内存中建立类的内部表示,其中包括用于存放代码的结构.编译模块对原本以字节码形式存在的方法进行编译,最后将编译产生的本地执行码写入用于存放代码的内部数据结构中,供以后使用.

监控模块本身没有任何活动的代码,而只是一些共享访问的数据结构,其中存放着协同处理模块和执行模块都需要的一些信息.通过监控模块,协同处理模块和执行模块之间可以相互通信.例如,协同处理模块中的编译选择子模块有时要基于执行过程中收集到的方法信息作出选择.这些信息由执行模块收集并存放于监控模块中,供协同处理模块使用.

3 JAFFE 虚拟机实现

我们在拥有开放源代码的 KaffeVM 的基础上实现了基于 HCCEM 的 Java 虚拟机原型系统 JAFFE.有关 Kaffe 的细节,见文献[5].本节将集中讨论 JAFFE 实现过程中因混合并发模型的引入而产生的几个问题.

3.1 执行模式的切换

在基于混合并发编译模式的 Java 虚拟机中,执行模式之间的切换包括两个方面:一方面,当一个正以解释执行方式运行的方法调用一个本地执行码已经准备就绪的方法时,需要从解释执行到本地码执行的切换;另一方面,当一个正在以本地码执行方式运行的方法调用一个本地执行码还未准备就绪的方法时,就可能发生从本地码执行到解释执行的切换.其中,从本地执行码到解释执行的切换是 HCCEM 独有的,任何已有的 Java 虚拟机中都不曾出现类似的切换.在图 3 的虚拟机设计中,每次执行模式的转换都将激活执行模式切换接口,其主要工作是处理参数的传递和结果的返回.由于这些对栈和寄存器的操作比较琐碎,有关细节本文不再给出.

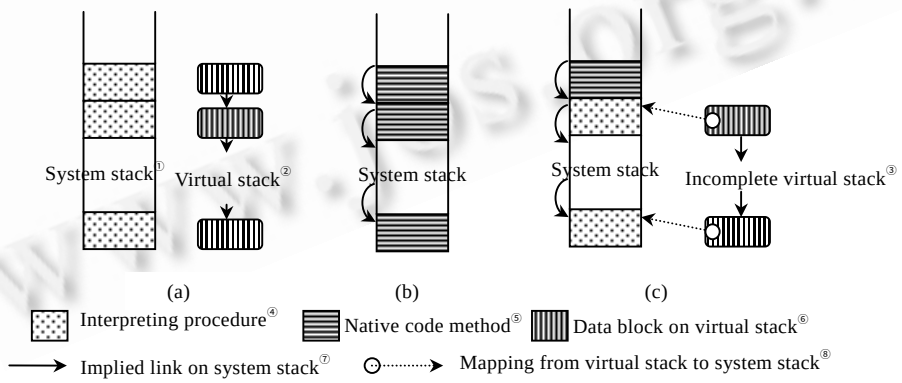


Fig.3 Different exception processing in Java virtual machines

图 3 不同的 Java 虚拟机中的异常处理

3.2 异常情况的处理

在 Java 虚拟机中,对异常情况的处理步骤分为异常的触发、在栈上搜索处理该异常的 Java 方法以及将执

行退回到所找到的 Java 方法并对异常情况进行处理.由于 JAFFE 的运行栈上可能同时存在着本地码方法和解释执行过程的活动记录,如何在这样一种新情况下正确定位异常处理方法则是需要解决的问题.

正确定位异常处理方法的关键在于维护一个 Java 方法的调用序列,只要有这样一个序列,一旦异常发生,就可以沿着这一序列向前回溯,同时对回溯过程中遇到的每一个方法进行考察,直到找到一个对该异常进行了处理的方法.在解释执行的 Kaffe 中,对每个 Java 方法的解释执行都是由一个独立的解释执行过程来完成的,方法的调用序列是保存在记录方法各种相关信息的虚拟栈中(如图 3(a)所示).它的缺点是每次进入和退出一个解释执行过程时都要对虚拟栈进行额外的操作,但考虑到解释执行过程中其他更加费时的操作,这些额外的操作不会对性能有太大的影响.但是,在 JIT 模式中这些额外操作对性能的影响将不再是可以忽略的.所以,在及时编译的 Kaffe 中,回溯在系统栈上进行.在正常执行的情况下,不做任何操作,而一旦发生异常,就利用 Java 方法在系统运行栈上的活动记录之间的隐式链表结构(这主要是通过指向前一个活动记录的栈顶基指针 BP 来链接的)向前回溯查找(如图 3(b)所示).这种解决方案在正常执行的情况下无须任何额外的操作,保证了及时编译模式的高效性.

上述两种解决方案在其各自的执行模式中都能很好地解决搜索异常处理方法的问题.但是在混合并发编译模式中,这两种方式将不可行.混合并发编译模式对高性能的要求使得构建一个完整的反映 Java 方法调用情况的虚拟栈的解决方案不再适用.而由于系统运行栈上可能同时存在着解释执行过程和本地码方法的活动记录,上述的扫描系统运行栈的解决方案无法区分这两种活动记录,从而无法正确对针对这两种活动的记录分别进行处理.

JAFFE 的解决方案是上述两个方案的一种综合,它能够很好地处理混合并发编译模式的特殊性.具体的措施如下:在正常执行过程中,对本地码方法和解释执行过程分别按照上述两个解决方案中的方法进行处理,即对本地码方法什么也不做,对解释执行过程则维护一个虚拟栈,需要改动的是将解释执行过程在系统运行栈上的栈顶基指针作为一种标志也保存到虚拟栈.当一个异常被触发时,我们所面对的是一个不完整的虚拟栈和一个包含完整的 Java 方法调用序列,但对其中的解释执行过程和本地码方法未加区分的系统运行栈.由于虚拟栈对应于所有活动于系统运行栈上的解释执行过程,并且还保存有它们在系统运行栈上的栈顶基指针,我们可以把虚拟栈上的每一个数据块映射到系统栈上,从而区分系统运行栈上的解释执行过程和本地码方法(如图 3(c)所示).这样,回溯查找工作就可以在系统栈上进行了.这种解决方案只对解释执行过程才维护其在虚拟栈上的数据块,而解释执行的方法仅占很小的比例,因此不会增加太多的运行时开销.另外,它的回溯查找过程比起及时编译模式下的解决方案要复杂,多了一步从虚拟栈到系统栈的映射,但是由于异常情况很少发生,这种在极少数情况下的相对低效对系统的整体性能不会有什么影响.

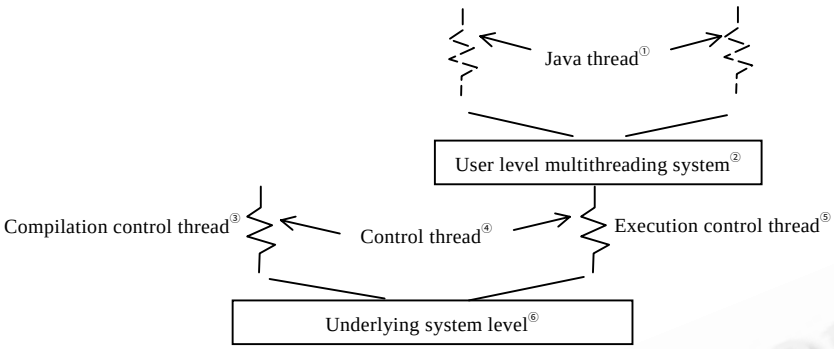
3.3 层次化的混合多线程

JAFFE 使用多线程来实现编译和执行的并发操作,由于 Java 本身也支持多线程,因此在 JAFFE 中将有两种类型的线程同时在活动.本节将就 JAFFE 中的混合多线程的组织和调度问题进行讨论.

3.3.1 混合多线程系统的组织与调度

JAFFE 中的 Java 线程和用于控制并发编译操作的线程无论从实现的功能,还是对调度的要求都有很大不同.若将这两种线程放在一起调度,一旦控制线程的优先级设得太高,就会妨碍真正程序的执行;而一旦控制线程的优先级设得太低,又可能始终得不到调度,使得编译操作无法真正并发起来.所以综上所述可以看出,对这两种线程进行有区别的处理是比较合理的选择.

在 JAFFE 中,我们设计并实现了一个具有层次结构的混合多线程系统.其结构如图 4 所示.对于控制编译和执行之间并发操作的线程,我们用系统级的线程来实现它们,唯有如此,在拥有多个 CPU 的计算机上,底层平台的计算能力才能得到充分利用.至于那些 Java 线程,则沿用 Kaffe 中原有的用户级线程,而整个 Java 多线程系统是构建在分管执行的系统级线程上的.这样一种混合多线程系统在结构上比较清晰,用系统级线程实现的控制线程的调度依靠具体的操作系统,并通过操作系统的支持而充分利用多个 CPU;而由用户级线程实现的 Java 线程,其调度依然保持很大的自由性,它们相互之间的切换对在后台并发进行的编译操作没有影响.



①Java 线程,②用户级多线程系统,③控制编译的线程,④控制线程,⑤控制执行的线程,⑥底层系统级多线程系统.

Fig.4 Hierachical hybrid multithreading system in JAFFE

图 4 JAFFE 中层次化的混合多线程系统

3.3.2 混合多线程系统的同步

JAFFE 中除了 Java 线程之间的同步以外,还需要有调度编译的控制线程和 Java 线程之间的同步.由于 JAFFE 是一个具有层次结构的多线程系统,所以一个很自然的想法就是采用层次结构的加锁方案,即将原来的一次加锁改为两步加锁,首先,利用系统线程库中提供的线程加锁接口对系统级线程进行加锁;然后,若是从 Java 线程中发起的加锁,还需要利用虚拟机的加锁方法再进行一次加锁.这种层次结构的加锁方案有以下几个缺点:

(1) 变一步加锁为两步加锁,增加了一次系统级线程的加锁,降低了系统性能;

(2) 系统级线程加锁时用到的互斥变量需事先申请,要为 Java 中的每个对象锁都申请一个互斥变量是不现实的,惟一可能的解决方案就是采取类似哈希锁的思想,事先申请一定数目的互斥变量,然后动态分配给加锁线程.这样,在每次加锁前都要有一个到互斥变量池中申请互斥变量的操作,降低系统性能;

(3) 在这种加锁方案中,每次加锁至少都需要进行一次系统级线程的加锁.这种系统级线程的加锁与使用硬件指令的快锁相比在执行时间上相差一个数量级以上,而一旦加锁不成功引起线程调度,对性能的影响则更为可观.

基于上述分析,我们在实现过程中没有采用这种层次结构的加锁方案,而是在快慢混合锁[5]的基础上进行了修改,以满足混合并发模型和低开销的需要.对于从调度编译的控制线程发起的加锁请求,我们用该线程的线程号来进行快锁加锁,而不再使用局部变量的地址.由于系统级线程的线程号都是很小的整数,所以不会和局部变量的地址或分配出来的锁的基地址相同.其算法见下.

加锁算法.

```
If (由控制编译的线程发起)
        while(1){
            用硬件指令和线程号进行快锁加锁;
            if (加锁成功) return;
          }
else{
    用硬件指令和局部变量进行快锁加锁;
    if (加锁成功)  返回;
      else {
        while(1) {
            取得原来锁指针;
            用硬件指令置锁指针为慢锁处理中标志;
            if (置标志不成功) {
                等待一段时间,若有其他活动线程则切换;
                continue;
              }
            if (原锁指针为线程号)   锁指针复原;
        }
if (原来锁指针已经指向一个真正的锁) {
    将本线程加到该锁的等待线程队列;
    锁指针复原;
    本线程进入等待状态,进行线程切换;

}
else {
    重新分配一个锁;
    if (原来锁指针  != NULL) {
            将本线程加到该锁的等待线程队列;
            将锁指针指向该锁;
            本线程进入等待状态,进行线程切换;
    }
    else  将锁指针指向该锁;
}
}
```

解锁算法.

```
If (由控制编译的线程发起) {
  while(1){
      用硬件指令解锁;
      if (解锁成功) return;
    }
 }
if (锁指针中为快锁) {
      用硬件指令进行解锁;
      if (解锁成功) return;
}
do {
      取得原来锁指针;
      用硬件指令置锁指针为慢锁处理中标志;
      if (置标志不成功)
            等待一段时间,若有其他活动线程则切换;
} while (置标志不成功);
if (锁指针所指的锁的等待队列非空) {
      从锁的等待队列中取出第 1 个线程;
      锁指针复原;
      将所取下来的线程恢复到就绪态,
      并进行线程调度;
}
else {
   释放该锁;
   将锁指针清零;
}
```

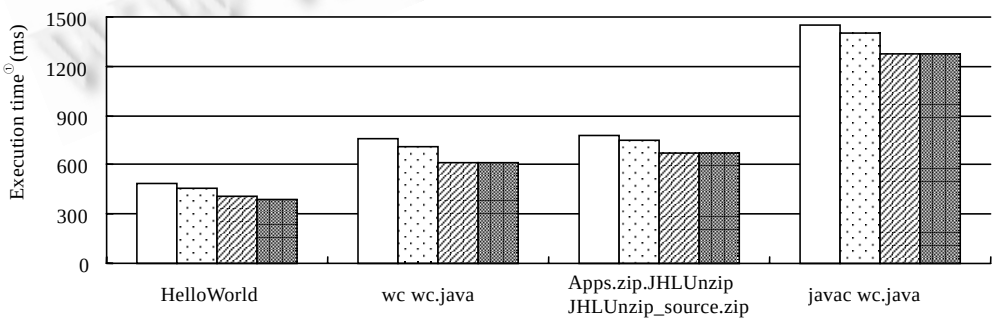
需要特别说明的是:(1) 不能让调度编译的控制线程进入可能造成用户级线程切换的慢锁加锁阶段,否则将引起错误.所以对由它发起的加锁,采用死循环方式,直到快锁加锁成功为止.(2) 对于由 Java 线程发起的加锁,若此时该锁正被控制线程占用,也不能进入慢锁阶段,而只能死循环等待控制线程解锁,否则,控制线程在解锁时将不知道如何唤醒等待在该锁上的用户级线程.

在我们的加锁方案中,采用死循环来加锁.这种方法在多 CPU 的计算机上,当系统负载不太大的时候,其效果要优于让线程休眠然后再唤醒的传统方法.采用死循环方式加锁在实现上也简单得多.

4 实验结果

本节我们给出基于 HCCEM 的虚拟机 JAFFE 针对一些程序的性能测试结果.实验环境为一台拥有双 CPU 的 Ultra2 工作站,操作系统为 Solaris2.5.2. 首先,每个被测的 Java 程序分别在解释和 JIT 方式下执行,以获取程序中用到的各个方法的解释执行时间、及时编译时间、本地码执行时间以及调用的顺序和次数.这些信息将作为同 JIT 比较的基础以及 HCCEM 用于确定编译顺序的历史信息.测试用例选用了 4 个程序,其中 HelloWorld 为 Java 程序设计的入门程序;wc 是用于统计一个文件中的单词数和行数;Apps.zip.jHLUnzip 为解压缩 ZIP 文件的程序,Javac 则是将 Java 源码编译为字节码的程序.

测试结果如图 5 所示,每组的第 1 列代表采用及时编译模式的执行时间.第 2 列代表引入解释执行和及时编译之间的选择,但使用单线程控制的执行时间.由此我们可以看出,对于某些方法来说,解释执行的效果确实要比编译执行好.后面两列都是采用混合并发编译模式的执行时间,不同之处在于第 3 列采用的是最短优先策略,第 4 列则是使用历史信息按照方法被调用的顺序进行编译.从实验结果来看,使用后一种编译选择策略要略好于前一种,但是,两者之间的差距很小.造成这一现象的主要原因在于现在还没有规模较大的公认的 Java 基准测试程序,因此上面使用的测试用例相对来说都较小,使得基于历史信息的编译策略的优势没有很好地得到体现.从总体上看,采用混合并发编译模式的执行时间与及时编译模式相比,平均可以缩短 20%的运行时间.



①执行时间.

Fig.5 Performace test result of JAFFE

图 5 Jaffe 的性能测试结果

5 总结与进一步的工作

混合并发模型 HCCEM 通过使用多线程控制将编译过程和执行过程重叠来提高程序的运行速度.本文就该模型进行了讨论,并给出了基于该模型的 Java 虚拟机 JAFFE 的设计和实现方法.实验结果表明,该模型能够充分利用处理机资源,有效地提高 Java 虚拟机的性能.

通过深入分析可知,虽然 HCCEM 可以重叠编译和执行过程以提高程序的执行速度,但是这种提高是有限的,因为程序的编译时间在程序总的执行时间中所占的比例相对较少,尤其是对于一些计算密集型的应用程序,使用的方法很少但被反复执行.对于这些程序,HCCEM 将不会带来大的性能改善.但是,由于 HCCEM 有着动态编译的特点,而且采用多线程控制,因此可以在后台对程序进行 JIT 下无法实施的比较费时的优化工作,而且这种优化可以结合程序的运行时信息,从而大幅度地提高本地码的质量.在今后的工作中,我们将针对这一情况,在编译模块中集成多种优化方法,以进一步提高程序的运行效率.

References:

[1] Cramer, T. Compiling Java just in time. IEEE Micro, 1997,17(3):36~43.

[2] Plezbert, M.P. Continuous compilation for software development and mobile computing [MS. Thesis]. Sever Institute of Washington University, 1996.

[3] Plezbert, M.P., Cytron, R.K. Does “Just in Time”=“Better Late than Never”? In: Proceedings of the 24th Annual SIGPLAN-SIGACT Symposium on Principle of Programming Languages. Paris: ACM Press, 1997.

[4] Lindholm, T. , Yellin, F. The Java Virtual Machine Specification. Addison-Wesley, 1996.

[5] Kaffe OpenVM 1.0.5. Transvirtual Technologies Inc., 1999. ftp://ftp.transvirtual.com/pub/kaffe/.

[6] Budimlic, Z., Kennedy, K. Optimizing Java: theory and practice. Concurrency: Practice and Experience, 1997,9(6):445~463.

[7] Bik, A.J.C., Gannon, D.B. Automatically exploiting implicit parallelism in Java. Concurrency: Practice and Experience, 1997,9(6):579~619.

A Java Virtual Machine Based on Hybrid Concurrent Compilation and Execution Model*

YANG Bo, WANG Ding-xing, ZHENG Wei-min

(*Department of Computer Science and Technology, Tsinghua University, Beijing* 100084, *China*)

E-mail: yangbo1997@yahoo.com

http://hpc.cs.tsinghua.edu.cn

Abstract: The change from interpretation to just-in-time compilation has improved the performance of Java dramatically. However, further amelioration is still possible. In this paper, a hybrid concurrent compilation and execution model (HCCEM) is proposed, which possesses the potential to surpass JIT by overlapping the production of native code with program execution through multithreaded control. The design of JAFFE, a Java virtual machine based on HCCEM, and some critical problems are also discussed in the implementation such as mode switch, exception handling and hierarchical multithreads. The experimental result shows that HCCEM can improve the execution speed of Java programs efficiently.

Key words: just-in-time compilation; hybrid concurrent; Java virtual machine; exception handling; hierarchical multithreads

* Received September 1, 2000; accepted April 17, 2001
Supported by the National Natural Science Foundation of China under Grant No.69873023