计算机科学 2000Vol. 27№. 4

Java 指令集结构的研究*)

Research of Java Instruction Set Architecture

戴 葵 秦 莹 陈 虎 胡守仁
(国防科技大学计算机学院 长沙 410073)

Abstract Java virtual machine is a machine architecture for executing programs. It is designed to be secure, modular and portable. However, the object-oriented features of the architecture also have a tremendous impact on the performance of the virtual machine. In order to better understand the performance of instructions in the Java virtual machine, this paper gives a benchmark suite based on Web applets in order to measure the performance of instructions. Testing results show that most of the execution time in the virtual machine is spent on memory management and garbage collection for objects. Using these results, this paper suggests improvements to be incorporated into future implementations.

Keywords Java virtual machine, Instruction set architecture, Performance

1 引言

Java 是一种编程语言，用其编写的程序具有安全、模块化和可移植等特点。当前 Java 在 Internet 有广泛应用，在网站主页的 HTML 代码中嵌入 Java 类文件，可以增强界面的动画效果，这种类文件称为小程序(Applet)，它是 Java 源程序的可执行代码。当浏览器访问包含小程序的主页时，相应的类文件从服务器传送到在客户机上运行的 Java 虚拟机(JVM)上，由 JVM 生成相应的类对象，并执行相应的方法。Netscape 浏览器中就包含这种 JVM。

然而，Netscape 浏览器为实现 Java 也付出了较大的代价，针对 Java 虚拟机(JVM)编译的 Java 程序代码其性能比相应的为本地机器结构编译的 C++ 程序代码要低得多，所以研究 Java 程序的特点，提出对 JVM 实现的改进措施具有非常重要的现实意义。

本文选择了一组小程序作为测试基准程序，通过分析这些小程序在 JVM 上的运行特性，讨论了 JVM 实现的性能瓶颈，并给出了一些克服这些瓶颈的方法，这些结果为 JVM 的高效实现提供了明确的目标。

2 Java 虚拟机(JVM)

JVM 是实现 JVM 指令集的一组功能单元的集合。这些功能单元一般由软件实现，也可由硬件实现，本文主要研究由软件实现的 JVM。JVM 指令由表示指令编码的一个字节、及不同大小和数目的参数组成，并以字节代码(bytecode)的形式进行引用，JVM 的结构如图 1 所示。

JavaAPI 由对象组成，为 Java 应用程序提供标准的服务；字节代码解释器对指令进行解码，并调用相应的功能单元；对象管理的功能则是分配新的对象，并对类、对象和方法进行解析；存储器用来存储局部变量；整数和浮点操作由算术逻辑部件 ALU 完成，而所有的操作和结果均是基于堆栈实现的。

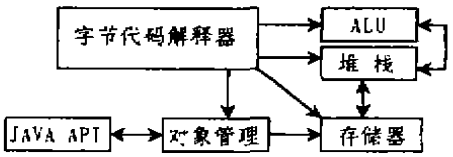


图 1 JVM 的结构

2.1 Java 程序的运行

一个 Java 程序就是一个类，有其自己的方法和数据块。程序的运行是以方法为单位进行的，通过方法调用完成程序功能。每个程序(类)的运行都由调用该类的 main 方法开始。

*)本文受国家自然科学基金资助(批准号 69803013)。戴葵 博士，副教授，研究领域为新型计算机体系结构、量子计算和量子计算机、神经网络等。秦莹 硕士研究生，研究领域为计算机体系结构。陈虎 博士研究生，研究领域为计算机体系结构。胡守仁 教授，博士导师，研究领域为计算机体系结构等。

字节码解释器对方法中的指令进行解码，并调用相应的功能模块完成指令。当字节码解释器遇到方法调用指令时，将在存储器中为方法的局部变量分配一个固定大小的存储块，这个块就称之为帧(frame)。帧由局部数据区、一个操作数堆栈和四个32位寄存器组成。方法调用时，JVM将相应的帧加入到一个帧堆栈中。方法调用返回时，JVM从帧堆栈中弹出方法相应的帧，转回到调用该方法的帧中。

2.2 面向对象的结构

JVM和传统结构的最大不同之处在于它将面向对象的特点融入了存储模型和指令集中。一个对象是由命名的数据块(称为成员)和代码块(称为方法)组成的实体，Java程序的全局存储空间完全由这种对象组成，这一点尤其不同于多数传统结构中的连续固定大小可寻址存储模型。JVM还提供了分配新的对象、访问对象的数据成员和调用对象方法的指令，Java程序的所有指令均是在基于某个特定的对象和方法所组成的语境下运行。

JVM中有一类分配对象指令，对象管理负责生成对象并返回对该对象的引用，这个引用可以被后续的指令使用，或是保存在当前的帧中。

JVM没有提供释放对象指令，而是依靠一个垃圾回收器来回收不再使用的对象。一个对象将一直保持在全局存储器中，直到在帧栈中再没有任何帧或在常数池中再没有任何对象引用它为止，此时，该对象就成为了"垃圾"。垃圾回收器将负责收回它占有的资源。

理论上，对象管理对应用程序来说应该是透明的，实际上垃圾回收器还是要和应用程序竞争宿主系统的资源。

2.3 Java应用程序接口(API)

JVM包含许多称为API的标准服务，这些API提供了线程、字符串、图形用户接口和其它系统功能，并以类库的形式提供给用户。尽管这些服务的实现随宿主系统不同而差异很大，但是程序能够通过生成适当的类对象来使用特定的API服务，并能够跨平台访问API对象。

Java的API是一种服务的抽象模型。API中同样包含能直接访问宿主系统的类，这些类包括一些称为本地(native)方法的特殊方法，通过调用宿主机的API完成所需的功能。

Java应用程序API的实现方式保证了Java程序跨平台的特性，并且具有扩展性好的特点，通过增加API类库，可为程序提供更多的服务。

3 JVM性能研究

我们采用了基准程序测试方法来研究JVM的性能。首先收集一些小程序，由这些小程序构成基准程序组；在Java字节代码解释器中加入测量装置；基准程序组中的小程序在修改后的Java解释器上运行，测量装置收集性能数据；最后处理数据，得到结论。

3.1 小程序基准程序组

我们从这样几个方面来选择基准程序：①计算密集型：涉及大量的算术运算，和经典的计算算法；②图形密集型：使用API的图形类来完成动画和图像处理任务；③存储器密集型：在程序运行过程中涉及大量的数据访问，通常是对矩阵的运算。选择的测试基准程序组如表1所示。

表1 基准程序组

小程序名称	计算密集型	图形密集型	存储器密集型	说明
CaffeineMark	√	√	√	商用Java浏览器基准程序
Dot3D	√	√	√	由点组成的3D表面动画
Liny	×	√	×	跳动线条屏幕保护器
Trans	√	√	√	利用周期函数的实时图像变换
Snowflake	√	√	×	一种DSP算法的可视化程序
Hanoi	√	×	√	Hanoi塔基准程序
Waves	√	×	×	波叠加基准程序

3.2 字节代码解释器中的测量装置

在Java字节代码解释器中，主循环的结构如下：

```
do {
    取下一个字节代码;
    根据字节代码执行某一动作;
} While(小程序还没有运行结束);
```

我们在指令的解释和指令的执行结束之间插入了测量代码，可以测量指令的执行时间。多数指令的执行时间是微秒级，为了保证测量的准确性，设置了高分辨率的定时器。

3.3 测量方法

基准程序组中的小程序运行时，测量装置主要收集这些信息：操作码；指令类型；执行时间。

指令类型可明确哪些类型的操作是经常用到的，这里Java的指令被分为如下类型：

·堆栈型：由于 Java 是堆栈类型的机器，有许多指令是完成堆栈的压入和弹出，而压入和弹出的操作数的大小各不相同。

·存储器型：这种类型的指令功能主要是在当前 Java 帧中的局部变量和堆栈之间移动数据。

·算术逻辑运算类型：Java 中所有的算术运算操作均是对栈顶的一到两个操作数进行处理，并将其计算结果返回堆栈的顶部。

·分支类型：这类指令通常包含常用的条件和无条件分支指令。

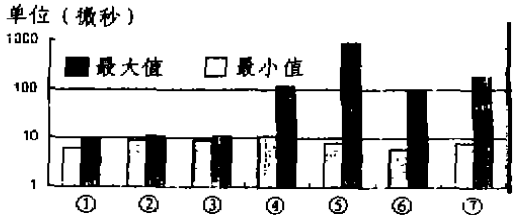
·对象类型：Java 在字节代码级提供了大量面向对象的特点，并有大量的字节代码支持在对象上的操作，Object 典型的功能包括生成一个新的对象、数组处理、在数据域中提取和存储数据，以及唤醒对象的方法。

·其它类型：这种类型主要包括数据类型转换指令和其它操作指令。

除了 Java 指令集标准所描述的字节代码之外，Java 解释器还提供了对某些指令的优化，这些优化指令以 quick 作为后缀。我们的分析中也认真考虑了这种类型的指令。

3.4 测量结果分析

3.4.1 *指令执行* 指令的执行时间测量结果如图 2 所示。



①堆栈操作；②存储器操作；③ALU 操作；④分支操作；⑤对象操作；⑥其它；⑦总平均

图 2 指令的执行时间统计结果对数图

从图 2 可以看出不同指令类型的执行时间有很大的不同，传统的机器指令，如存储器类型、堆栈类型和 ALU 类型的指令的时间大约都是在 2—10 微秒之间，而对象类型的指令则没有任何规律，其执行时间大约是传统机器指令类型的 2—5 个数量级。

在对象中提取和设置域值、调用对象方法、生成新的对象、处理向量等对象指令的执行需要查找大量的表格，方法和域的名字解析也需要搜索表格，查表需要大量的时间；另外，对象的存储器管理也比较费时。

分支指令的执行也比较耗时，主要是由于其中的方法返回指令造成的。一个方法执行完毕，并将控制返回给前一个方法时，需要清空帧堆栈。

测试还表明，几乎在所有的基准程序中，对象指令几乎占据了所有的执行时间，所以为了提高 Java 程序的性能，必须要优化对象指令的执行。

3.4.2 *对象分配指令分析* 由上面测试可知，对象分配指令（new 类指令）是对程序执行时间影响最大的指令，我们对其实现进行了深入分析。基本的 new 指令从堆中分配存储空间，优化的 new_quick 指令则直接在该类的常数池中分配存储空间。如果在常数池中没有遇到足够容量的存储器，就按照标准的 new 指令来分配存储器。new_quick 指令的平均执行时间约为 8 微秒，而相应的 new 指令的平均执行时间大约是 10—100 微秒，并且多数情况执行的是 new_quick 指令。最坏情况下存储分配的执行时间是 100ms，这种最坏情况主要因为垃圾回收。

通常，Java 的垃圾回收器以低优先级的线程执行，但是如果存储分配超出了存储空间范围，垃圾回收器马上被调度执行。图 3 表明了在存储器密集型基准程序中存储分配的执行时间。

一般存储器分配的时间在 10 微秒以内，但是在空间不足时，存储器分配需要暂停，等待垃圾回收器回收存储空间，满足分配，因而使得其执行时间上升了近 5 个量级。图中曲线的尖峰表明了垃圾回收的时刻。

JVM 规范对垃圾回收器的实现并没有任何描述。SUN 公司当前采用的垃圾回收策略是"标记清扫"策略，基于这种策略，垃圾回收器扫描整个存储器，并标记当前可见的对象，并清扫那些没有被标记的存储器。这种实现十分简单，但比较低效。

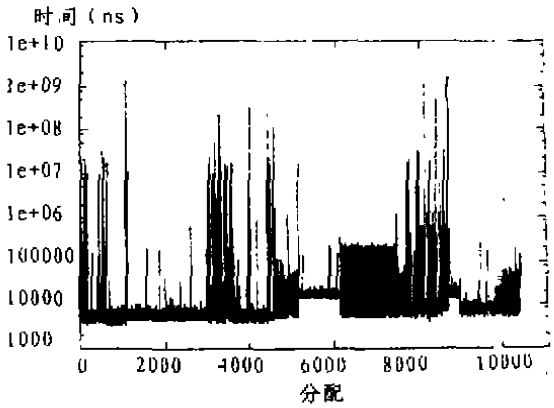


图 3 存储器分配指令的执行时间

讨论 从上面的分析可以看到，Java 结构面向对象程序模型的最显著的特点，也是其性能低效的最主

要的原因。一般，有两种提高性能的策略，一是提高对象指令和对象管理系统的效率，二是降低 Java 程序的特性的使用而提高性能。

JVM 中对象指令最低效的部分在于对象的存储管理，特别是新的对象分配，针对存储管理进行改进是提高效率的有效手段。根据当前 JVM 对 new_quick 指令优化实现的思想，可以得到一种对象分配的优化方法。为所有对象开设一个全局池，具有较短生命周期的临时对象和诸如例外、中断之类的高优先级对象能够有效地在这种全局池上进行分配。垃圾回收策略将急剧降低某些分配的性能，采用高效的垃圾回收算法可以大大提高效率。如采用"世代回收技术"对多线程应用十分有效。

JVM 极大地依赖于诸如存储器管理等操作系统、本地机器结构的支持，Java 必须花费时间和空间将其服务转化成为宿主系统所使用的模型、在操作系统级别实现一个自动存储器管理器将会提高 Java 的性能。

提高 Java 程序执行性能的另外措施是在编译时尽量减少对象指令的数目，在 C++ 编译器中，这种优化称之为插入(inlining)。插入通过将调用替换为方法的实际代码而实现，所以是直接执行代码，没有额外的方法调用开销，可以采用同样的"方法展开"优化 Java 中方法调用的开销。Java 程序中的字符串类、数字类和例外实例通常生成临时对象以调用方法。在这种情况下，插入方法代码和对象构造器的代码，可以免去费时繁琐的对象分配和方法调用，降低在程序中所需要的垃圾回收次数，达到提高性能的目的。

即时编译技术也是提高 Java 程序执行的有效手段，即时编译器将 Java 字节代码转化成本地机器代码，显然本地代码的执行比解释执行要快，减轻存储器管理压力的另外一种技术是对类和对象进行压缩，Netscape Navigator 当前就是采用压缩格式读取和使用其 Java API 类，这种压缩降低了对存储器的需求，然而这种技术对性能究竟有多大的影响还需进一步进行研究。

参 考 文 献

1 Java(tm) Virtual Machine Specification Sun Microsystems, Inc ,1995

2 Gosling J, McGilton H. The Java Language Environment, A White Paper. Sun Microsystems, Inc. ,1995

3 Patterson D A, Hennessy J L. Computer Architecture: A Quantitative Approach, 2nd Edition. Morgan Kaufmann Publishers, Inc. ,1995

(上接第 22 页)

3 自调节软件往何处去

上边我们已经简要介绍了自调节软件的一些概况，它的出现已经引起了人们相当大的关注，因此有必要对于它的今后发展谈谈我们的看法。

自调节软件是一个崭新的事物，尽管目前它还处于幼年时期，但它代表了人们在软件上所追求的方向。因此可以肯定地说，某一个具体的自调节软件不一定会永久地存在下去—它肯定会被性能更好的自调节软件所代替，但是作为一个整体，自调节软件必将成为人们追求的目标，就如同人们曾经努力去研制可搬家的软件一样。在 21 世纪里，人们一定会努力去研制自调节软件，以期这样的软件可以充分利用它在上边运行的硬件的所有能力。

其次，适应于自调节软件的工作，或许在计算机的硬件结构上也有必要有一个调整，以便使这种软件的调节可以更容易进行。比如，在寄存器的数量的设置上，或许就应该考虑增加寄存器的数量才能使计算得以更快进行，以解决上边谈到的处理器处理的快速与存储器存取相当缓慢之间的不匹配。

第三个问题是，目前的自调节软件几乎都是面向科学计算的，因此我们自然要说科学计算确实很重要，但在今天，计算机的应用领域已经是如此广泛，以致使科学计算在整个计算机应用所占有的份额已降到较低的地位。我们希望见到适用于各种各样需求的自调节软件，特别是希望能有适应于网上工作或在分布式系统上工作的自调节软件，如果那样，自调节软件就会发挥更大的作用。

第四个问题是，我们看到，无论是 ATLAS 还是 FFTW，其工作原理都是基于分而治之的思想，因此，也就有这样一个问题，当软件所处理问题不具有可分性，是否能构造出自调节的软件，或者在这样一个问题中，如何去找出可分的部分来，针对这个部分来作些自调节。显然，这种情况，也仍值得我们加以研究。

参 考 文 献

1 Cipra B. Self-Tuning software Adapts to Its Environment. Science, 1999,286(5437)

2 Thomas W. Parsons Introduction to Algothms in Pascal. John Wiley & Sons, 1995