高技术通讯 2014 年 第 24 卷 第 12 期:1228 ~ 1237

doi:10.3772/j.issn.1002-0470.2014.12.003

编译队列监视下的 Size-Speed 动态编译调度算法①

傅 杰[*,**,***] 廖 彬[*,**,***] 陈新科[*,**,***]

靳国杰[****] 章隆兵[*,**,***] 王 剑[*,**,***]②

([*]中国科学院大学 北京 100049)

([**]计算机系统结构国家重点实验室 北京 100190)

([***]中国科学院计算技术研究所 北京 100190)

([****]龙芯中科技术有限公司 北京 100095)

摘 要 针对动态编译影响虚拟机启动性能和响应速度的问题,研究了动态编译系统的优化技术,提出了编译队列监视下的 Size-Speed 动态编译调度算法。该 Size-Speed 调度算法以程序中方法的运行速度和方法本身的大小为参数计算调度的优先级,使得调度既能适应程序的动态行为,又能提高动态编译的吞吐量。此外,在调度的同时进行编译队列监视,通过跟踪编译队列中方法的活跃度,及时删除队列中不再活跃的方法,进一步降低了动态编译的开销。实验结果表明,该算法能够显著提升系统的启动性能和响应速度。在 DaCapo 的测试中,虚拟机总体性能提升了 12.4%,单项性能的最大提升幅度高达54.3%。该算法通用性强,实现代价低,适用于绝大多数动态编译系统。

关键词 虚拟机,动态编译,编译调度,启动性能,响应速度,优化

0 引 言

云计算环境下,高度异构的设备和复杂的网络环境对程序的可移植性和安全性提出了更高的要求[1]。基于虚拟机技术实现的语言,如 Java 和 C# 等,使得程序具备了“一次编译,到处运行”的高度的可移植性。同时虚拟机内部所提供的多种运行时检查机制增强了程序运行的安全性[2]。因此,虚拟机技术在云计算环境中得到了广泛的应用。采用 Java 等基于虚拟机实现的语言所编写的程序,通常以平台无关的中间代码形式(如字节码)发布[2]。中间代码必须通过虚拟机来执行。虚拟机的执行引擎包括解释器(interpreter)和即时(just-in-time,JIT)编译器。解释器结构简单,易于实现,但执行速度非常慢。为了提升虚拟机的性能,人们引入了 JIT 编译器。JIT 编译器的编译行为发生在程序运行期间,故又称为动态编译[3]。动态编译是决定虚拟机性能最为关键的因素。

动态编译会引入额外的运行时开销[4],影响虚拟机的启动性能和响应速度。对于目前广泛使用的智能手机、平板电脑等手持设备,较低的启动性能将严重影响用户的使用体验。而在交互式环境下,如

① 863 计划(2012AA010901,2012AA011002,2012AA012202,2013AA014301),国家自然科学基金(61221062,61100163,61133004,61173001,61232009,61222204)和国家“核高基”科技重大专项课题(2009ZX01028-002-003,2009ZX01029-001-003,2010ZX01036-001-002,2012ZX01029-001-002-002)资助项目。

② 男,1987 年生,博士生;研究方向:虚拟机,计算机系统结构;联系人,E-mail:fujie@ict.ac.cn

(收稿日期:2014-08-15)

网络游戏、订票系统和 KTV 点唱系统等,较低的响应速度通常构成整个系统的应用瓶颈。因此,对虚拟机的动态编译系统进行优化,提升系统的启动性能和响应速度具有十分重要的意义,已逐渐成为虚拟机的研究重点[4-6]。本文研究了动态编译系统的优化技术,提出了编译队列监视下的 Size-Speed 动态编译调度算法。实验结果表明,该算法可以显著提升虚拟机的启动性能和响应速度。

1　相关工作

20 世纪 90 年代,Hölzle 等人[7]在设计 Self-93 虚拟机时已开始关注如何提升虚拟机的启动性能和响应速度。他们设计并实现了著名的分层编译系统。该系统包含一个非优化的 JIT 编译器和一个优化的 JIT 编译器。非优化的编译器编译速度快,但生成的机器代码质量不高;优化的编译器能生成高质量的机器代码,但是编译速度较慢。程序中所有的方法在第一次运行时,均由非优化的编译器迅速进行编译,而优化的编译器仅编译程序中频繁执行的方法。Self-93 虚拟机能够根据程序运行时收集的信息,自主判断何时对哪些方法采用优化的编译器进行编译。虽然分层编译系统能有效提升虚拟机性能,但该方法要求系统中存在多个 JIT 编译器,实现代价高,并且还需要设计一系列复杂的控制和状态转换算法[8]。

文献[9,10]提出了一种相对简单的动态编译优化技术。其主要思想是提升编译线程的优先级,使得编译线程优先被操作系统调度。文献[10]还规定了在多线程环境下编译线程对 CPU 占用率的下限,优先满足编译线程对系统资源的需求。上述优化能加快虚拟机动态编译的速率,降低方法在编译队列中的等待延迟。然而该方法总是将系统资源优先分配给编译线程,在编译负荷较大(如系统启动或程序行为突变)时,应用程序本身的执行将受到抑制。此外,该方法还要求操作系统内核提供特定的线程调度支持,缺乏通用性,因而其适用范围受到限制。

近年来,有人提出通过动态编译调度来优化虚拟机的启动性能和响应速度[4-6]。由于动态编译调度具有简单、有效、通用性强等特点,已成为研究人员关注的焦点。一种直观的想法是基于方法大小(size)的调度。该调度算法优先选择较小的方法进行编译,从而使所有方法在编译队列中的平均等待时间最小,最大限度地提升动态编译的吞吐量。但是基于 size 的调度没有考虑程序的动态行为,且总是延迟大方法的编译,因而不满足对外界变化快速响应的要求。Stadler 等人[6]又提出了基于方法运行速度(speed)的编译调度算法。该算法在调度时实时计算编译队列中各个方法的运行速度,优先编译运行速度较快的方法。虽然基于 speed 的调度考虑了程序的动态行为,但却不能充分降低方法在编译队列中的平均等待时间,忽视了对系统中编译吞吐量的优化。

针对上述研究的不足,本文提出了编译队列监视下的 Size-Speed 动态编译调度算法,使得调度既能适应程序的动态行为,又能提高虚拟机动态编译吞吐量。本文首先展示了动态编译的基本过程。然后分析了动态编译调度模型,并在此基础上提出了编译队列监视下的 Size-Speed 调度算法。最后通过对比实验,分析和验证了本文算法的有效性。

2　动态编译的基本过程

本文的实验工作在 Oracle JDK8[11] 的 HotSpot 虚拟机上进行。HotSpot 是一款广泛使用的高性能 Java 虚拟机,其执行引擎由解释器和即时编译器[12,13]组成。图 1 展示了 HotSpot 虚拟机动态编译的基本过程。程序运行时,所有方法均由解释器开始执行。HotSpot 只翻译执行频度较高、对程序性能影响较大的方法,即热点方法[3,12,13],以减小动态编译的开销。图 1 中,λ 表示虚拟机识别热点方法的

速率,μ 表示虚拟机编译方法的速率。对任意方法 m,虚拟机内部维护两个计数值 ic __count(m) 和 bc __count(m),其中 ic __count(m) 表示对方法 m 的调用次数,bc __count(m) 表示方法 m 内部循环的迭代次数。当上述两个计数值之和超过 HotSpot 中设定的编译阈值时,方法 m 则被识别为热点方法,并被加入编译队列中等待编译。随后编译线程按先来先服务(first come first service,FCFS)的策略,从编译队列中选择方法进行编译。HotSpot 虚拟机可根据需要创建多个编译线程。当方法 m 的编译完成后,此后对 m 的调用将直接执行翻译后生成的本地代码。研究表明[7],本地代码的执行效率比解释器的执行效率高出 10 倍以上。

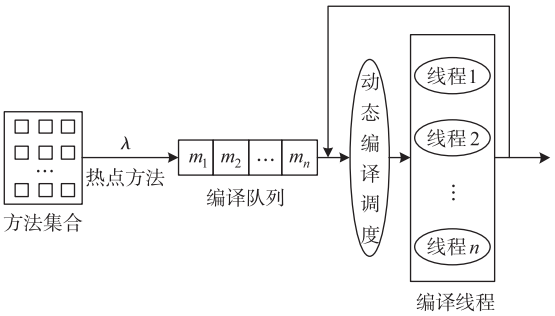

图 1 动态编译基本过程示意图

研究发现[14],在编译阈值一定的情况下,当系统启动或程序行为突变时,虚拟机识别热点方法的速率 λ 将远大于虚拟机编译方法的速率 μ。这将导致编译队列中瞬间积压大量待编译的方法,从而使方法在编译队列中的平均等待时间增大。Jantz 和 Kulkarni 等人[4,5]的研究成果表明,对性能影响较大的方法在编译队列中等待时间太长是虚拟机启动性能低和响应速度慢的重要原因,并进一步指出编译调度优化是提高虚拟机性能的有效途径。

3 编译队列监视下的 Size-Speed 调度算法

3.1 动态编译调度模型

设在 t 时刻,编译队列中共有 n 个等待编译的方法,分别记为 $m_1, m_2, \cdots, m_n$。设方法 m_k($k \in [1, n]$) 对程序性能影响的重要程度为 $h_t(k)$,编译方法 m_k 所需时间为 C_k。显然,方法 m_k 的重要程度 $h_t(k)$ 越高、编译时间 C_k 越短,则编译该方法所带来的收益也越大。记函数 $b_t(k)$ 表示 t 时刻编译方法 m_k 的收益,则 $b_t(k)$ 可表示为

$$b_t(k) = \frac{h_t(k)}{C_k} \tag{1}$$

假定在 t 时刻选择方法 m_k 进行编译。由于编译方法 m_k 需要的时间为 C_k,则编译队列中其余 $n-1$ 个方法被延迟编译的时间为 C_k,这将对系统的性能产生负面影响。对方法 m_i($i \in [1,n], i \neq k$) 而言,其重要程度 $h_t(i)$ 越大,则延迟编译造成的损失也越大。记函数 $l_t(k)$ 表示 t 时刻选择编译方法 m_k 产生的代价,则 $l_t(k)$ 可表示为

$$l_t(k) = C_k \cdot \left[\sum_{i=1}^{k-1} h_t(i) + \sum_{i=k+1}^{n} h_t(i) \right] \tag{2}$$

于是,在选择某个方法进行编译时,需要对其所带来的收益和所产生的代价进行权衡。令调度时对方法 m_k 的优先级计算函数为 $y(k) = \alpha \cdot b_t(k) - (1-\alpha) \cdot l_t(k)$,其中,$\alpha$ 为可调参数,且 $\alpha \in [0,1]$,则 t 时刻应选择编译的最优方法 m_K 的判别准则为

$$K = \arg\max_k \{y(k)\} \tag{3}$$

而

$$\begin{aligned} y(k) &= \alpha \cdot b_t(k) - (1-\alpha) \cdot l_t(k) \\ &= \alpha \cdot \frac{h_t(k)}{C_k} - (1-\alpha) \cdot C_k \cdot \left[\sum_{i=1}^{k-1} h_t(i) + \sum_{i=k+1}^{n} h_t(i) \right] \\ &= \alpha \cdot \frac{h_t(k)}{C_k} - (1-\alpha) \cdot C_k \cdot \left[\sum_{i=1}^{n} h_t(i) - h_t(k) \right] \end{aligned} \tag{4}$$

又令 $H(t) = \sum_{i=1}^{n} h_t(i)$,则 $H(t)$ 在时刻 t 为一常量,故有

$$y(k) = \alpha \cdot \frac{h_t(k)}{C_k} - (1-\alpha) \cdot C_k \cdot [H(t) - h_t(k)] \tag{5}$$

将 $y(k)$ 对 $h_t(k)$ 求偏导,得

$$\frac{\partial y(k)}{\partial h_t(k)} = \alpha \cdot \frac{1}{C_k} + (1-\alpha) \cdot C_k > 0 \quad (6)$$

将 $y(k)$ 对 C_k 求偏导,得

$$\begin{aligned}\frac{\partial y(k)}{\partial C_k} &= -\alpha \cdot \frac{h_t(k)}{C_k^2} - (1-\alpha) \cdot [H(t) \\ &\quad - h_t(k)] \\ &= -\alpha \cdot \frac{h_t(k)}{C_k^2} - (1-\alpha) \cdot [\sum_{i=1}^{k-1} h_t(i) \\ &\quad + \sum_{i=k+1}^{n} h_t(i)] < 0 \end{aligned} \quad (7)$$

故 $y(k)$ 随 $h_t(k)$ 单调递增,且随 C_k 单调递减。于是,虚拟机应优先选择重要程度高、所需编译时间短的方法进行编译。重要程度高的方法优先,可以充分适应程序的动态行为,提高虚拟机的响应速度。而所需编译时间较短的方法优先,可以降低方法在编译队列中的平均等待时间,提高动态编译的吞吐量。

3.2　Size-Speed 调度算法

基于3.1中的调度模型,本文提出了 Size-Speed 调度算法。其主要思想是利用方法的运行速度(speed)来估计方法的重要程度,利用方法的大小(size)来估计编译所需的时间。

方法 m_k 对性能影响的重要程度 $h_t(k)$ 与程序未来的行为密切相关。至今对 $h_t(k)$ 的估计都是一个开放性问题。本文借鉴了文献[6]中提出的方法运行速度的概念。定义方法 m_k 在 t 时刻的运行速度为

$$speed_t(m_k) = \frac{\Delta execute(m_k)}{\Delta t} \quad (8)$$

其中,Δt 为 t 时刻前的一个时间段,$\Delta execute(m_k)$ 为方法 m_k 在 Δt 时间段内被调用的次数和内部循环迭代的次数之和,即

$$\begin{aligned}\Delta execute(m_k) &= \Delta ic\_count(m_k) \\ &\quad + \Delta bc\_count(m_k)\end{aligned} \quad (9)$$

方法运行速度越快,则认为其重要程度越高,因此可令 $h_t(k) = speed_t(m_k)$。

文献[15]的研究表明,方法的编译时间与方法的大小存在线性关系。一般来说,方法越大,所需的编译时间也越长。因此,本文用方法的大小来估计所需的编译时间,即令 $C_k = size(m_k)$。

至此,优先级函数(式(5))可表示为

$$\begin{aligned}y(k) &= \alpha \cdot \frac{speed_t(m_k)}{size(m_k)} - (1-\alpha) \cdot size(m_k) \\ &\quad \cdot H(t) + (1-\alpha) \cdot size(m_k) \cdot speed_t(m_k)\end{aligned} \quad (10)$$

其中 $\alpha \in [0,1]$ 为可调参数,且

$$H(t) = \sum_{i=1}^{n} speed_t(m_i) \quad (11)$$

于是,t 时刻动态编译调度问题可转换为一个优化问题:从编译队列中寻找一个方法,使得优先级函数(式(10))的值最大。求解该优化问题的具体步骤如下:

步骤 1:对编译队列中的每个方法按式(8)计算方法的运行速度。

步骤 2:对编译队列中所有方法按式(11)计算 $H(t)$ 的值。

步骤 3:对编译队列中的每个方法按式(10)计算调度的优先级。

步骤 4:遍历整个编译队列,选择优先级最大的方法进行编译。

从上述算法步骤不难看出,对于长度为 n 的编译队列,该算法的时间复杂度为 $O(n)$。

3.3　编译队列监视下的 Size-Speed 调度

现有的热点识别技术通常采用基于阈值的方法[4]。若 t 时刻方法 m 执行的次数超过编译阈值 T,则判定方法 m 为热点方法,并将方法 m 加入编译队列中等待编译。在等待编译期间,系统并没有对方法 m 是否依旧活跃进行监视。若在时刻 t 之后,方法 m 不再执行(例如,方法 m 总共被执行 $T+1$ 次),那么编译方法 m 不能带来任何性能上的提升,反而增加了系统的编译开销。因此,虚拟机有必要对编译队列中的方法进行监视。当侦测到某个方法

不再活跃时,系统及时将其从编译队列中删除,以减少不必要的编译开销,同时降低重要方法的编译延迟。

方法在时刻 t 的运行速度可以很好地反映该方法的活跃性。因此,对编译队列中方法活跃性的监视可以在编译调度过程中进行。调度时,计算各个方法的运行速度,当发现方法 m 的运行速度小于或等于活跃度阈值 S 时,则判定方法 m 不再活跃,并将其从编译队列中删除。于是,编译队列监视下的 Size-Speed 调度算法步骤如下:

步骤 1:对编译队列中的每个方法按式(8)计算方法的运行速度。

步骤 2:将运行速度小于或等于活跃度阈值 S 的方法从编译队列中删除。

步骤 3:对编译队列中所有方法按式(11)计算 $H(t)$ 的值。

步骤 4:对编译队列中的每个方法按式(10)计算调度的优先级。

步骤 5:遍历整个编译队列,选择优先级最大的方法进行编译。

对于长度为 n 的编译队列,该算法的时间复杂度为 $O(n)$。本文的实验中,活跃度阈值 S 取值为 0。

4 实验结果与分析

为了验证编译队列监视下的 Size-Speed 动态编译调度算法的效果,本文选用业界广泛使用的 DaCapo[16] 测试集进行实验。DaCapo 总共包含 14 个测试项,覆盖了现代虚拟机的典型应用场景。由于最新发布的 DaCapo-9.12-bach 中,batik 和 eclipse 引用了与 JDK8 不兼容的低版本类库,因此本文将它们排除在实验结果之外。实验使用的机器配置为 X86 处理器,主频 1.6GHz,DDR3 内存 2G,操作系统为 64 位的 Ubuntu 12.04。为使实验结果更具普遍适用性,实验中禁用了 HotSpot 虚拟机的分层编译功能。

4.1 实验设置

对于每个测试项,DaCapo 可提供两种运行结果,即首次运行的结果和稳定状态下的运行结果。研究人员通常将首次运行的结果作为虚拟机启动性能和响应速度的度量[4-6],而将稳定状态下的结果作为虚拟机峰值性能的度量。因此,为度量算法对虚拟机启动性能和响应速度的影响,本文的实验数据均取自 DaCapo 中各测试项首次运行的结果。本文进行了以下 5 组对比实验:

(1)原始的 FCFS 调度;

(2)基于 Size 的调度;

(3)基于 Speed 的调度[6];

(4)基于 Size-Speed 的调度;

(5)编译队列监视(Queue-Watch)下的 Size-Speed 调度算法,简称 QWSS 算法。

为了降低实验误差,每组实验均运行 10 次,取其均值作为最终的实验结果,并以 FCFS 算法的结果为基准对实验结果进行了归一化处理。

4.2 实验结果

虚拟机的总体性能通常用测试集内所有项目测试结果的几何均值来衡量[4-6]。图 2 展示了在 QWSS 算法下,虚拟机总体性能随可调参数 α 取值的不同而变化的趋势。从图 2 可以看出,系统性能随 α 的变化较为平缓。当 $\alpha = 0.7$ 时虚拟机具有最高的性能,故后文中的实验均取 $\alpha = 0.7$。

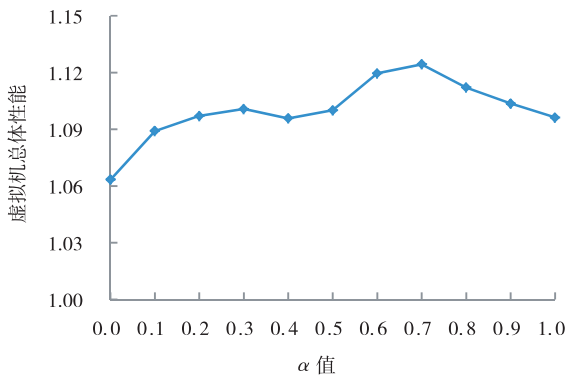

图 2 虚拟机总体性能随 α 的变化趋势

图 3 展示了虚拟机在不同算法下的总体性能。实验结果表明,相对于 FCFS 调度,采用 Size-Speed 调度算法,虚拟机总体性能提升幅度高达 10.7%,明显高于基于 Size 调度的算法(6.6%)和基于 Speed 调度的算法(7.1%)。而 QWSS 算法可在 Size-Speed 调度的基础上进一步提升约 2% 的性能,达到 12.4% 的提升幅度。

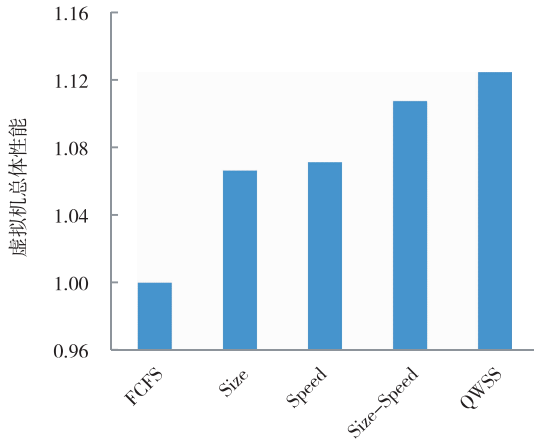

图 3　不同算法下虚拟机的总体性能

图 4 展示了虚拟机在不同调度算法下各个测试项目的实验结果。从图中可以看出,与 FCFS 算法相比,基于 Size 或 Speed 的调度算法均存在一些性能下降的项目,而 Size-Speed 和 QWSS 算法则未出现任何性能下降的项目,并且 QWSS 算法对于 jython(54.3%)、h2(29.1%)、tradebeans(23.8%)、xalan(19.1%)、tradesoap(12.9%)和 tomcat(10.8%)等项目,均有显著的性能提升效果。

此外,对于所有的测试项目,基于 Size-Speed 的调度算法均优于基于 Size 或 Speed 的调度算法,充分体现了 Size-Speed 调度算法的优势。并且编译队列监视可在 Size-Speed 调度的基础上进一步提升虚拟机性能,尤其是在 jython、tomcat 和 xalan 等项目上。这表明了编译队列监视对动态编译系统优化的有效性。

基于 Size 的调度算法只追求提升动态编译的吞吐量,却忽视了程序的动态行为,而基于 Speed 的调度算法仅关注程序的动态行为,却忽略了对系统

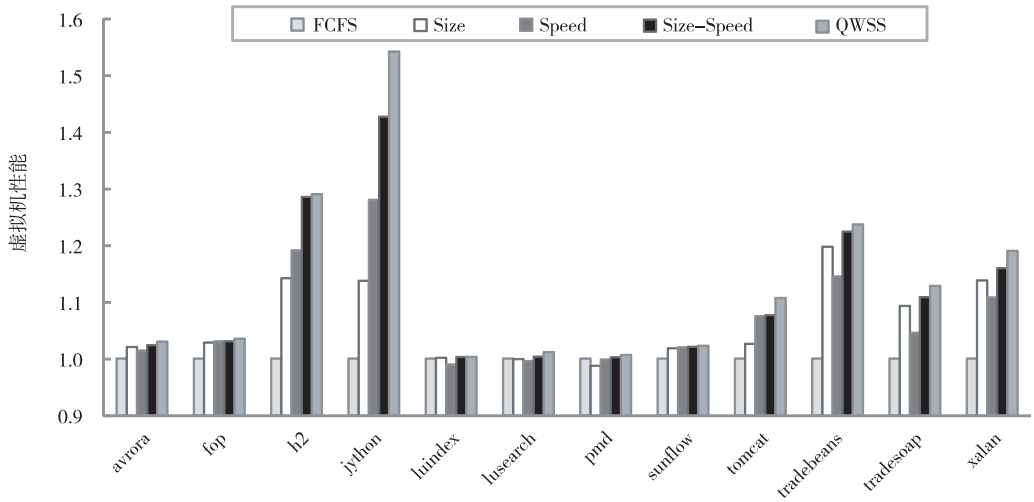

图 4　不同算法下虚拟机在 DaCapo 测试集上的性能

编译吞吐量的优化。故单纯基于 Size 或 Speed 的调度算法均不能充分发挥虚拟机动态编译系统的性能。本文提出的 QWSS 算法既适应了程序的动态行为,又能提高编译的吞吐量,因此能够获得较高的性能提升。

4.3　实验分析

实验中收集了虚拟机在 FCFS 以及 QWSS 两种算法下的运行时数据。重点对比了不同算法下编译队列的平均长度、峰值队列长度和解释执行的时间比例,并且统计了编译队列监视期间撤销的方法数

量及其占编译方法总数的比例,具体数据如表 1 所示。

由于在编译阈值一定的前提下,热点方法在不同算法下进入编译队列的平均速率相同,因此,平均队列长度越短则表明动态编译的吞吐量越高。从表 1 中的数据可以看出,QWSS 算法大幅降低了编译队列的平均队列长度和峰值队列长度。由此可见,QWSS 算法的动态编译吞吐量也获得了提高。

表 1 虚拟机运行时数据

测试项目	平均队列长度		峰值队列长度		解释执行比例		编译队列监视	
	FCFS	QWSS	FCFS	QWSS	FCFS(%)	QWSS(%)	撤销数目	比例(%)
avrora	2.7	1.7	61	39	30.1	29.2	11	3.4
fop	10.9	4.9	45	34	79.9	79.2	4	1.9
h2	5.3	2.6	68	54	58.3	53.3	22	3.4
jython	31.6	9.9	128	101	43.2	38.2	24	5.9
luindex	2.5	1.8	22	20	86.7	86.4	1	1.1
lusearch	58.6	26.2	136	135	86.6	85.9	5	2.4
pmd	152.1	59.5	262	183	96.9	96.2	0	0.0
sunflow	1.7	1.4	26	25	38.4	38.1	1	0.8
tomcat	12.3	5.4	61	54	79.9	76.5	64	9.5
tradebeans	9.4	3.5	85	58	62.5	57.7	29	2.7
tradesoap	29.6	6.0	222	76	60.4	57.2	78	3.6
xalan	45.7	15.1	127	66	77.4	76.6	21	5.1

表 1 中的数据还可进一步解释图 4 的实验结果。例如,对于 sunfow、luindex 和 avrora 等,由于其运行时平均队列长度较短,可供调度的空间有限,故在实验中性能提升的幅度较小。而对于 pmd 和 lusearch 等,虽然运行时平均队列长度较大,但由于其解释执行的时间占主导地位,故编译调度的效果也较小。jython 的平均队列长度足够大,并且解释执行的时间比例仅为 38.2%,故在实验中获得了 54.4% 的性能提升。而对于 h2,虽然其平均队列长度不如 jython 大,但由于程序运行的时间分布非常集中,故也表现出明显的性能提升效果。由于 tomcat、jython 和 xalan 等撤销的编译方法数目占编译方法总数的比例较大,相应减少的编译开销也越多,故编译队列监视的效果较好。

动态编译调度通常需要遍历整个编译队列。为了表明遍历不会对系统产生负面影响,本文以 QWSS 算法为例对虚拟机进行了窗口调度实验。窗口调度是指调度时仅查看队列中固定数目的方法的调度,并称该固定数目为窗口大小。本文选取了用 QWSS 算法时性能提升较为明显的 6 个项目进行实验,得到的测试结果如图 5 所示。图中横坐标表示编译调度的窗口大小,纵坐标表示虚拟机的性能。实验结果表明,随着调度窗口的增大,虚拟机性能呈上升趋势。较大的调度窗口,不仅使得虚拟机可以在更大的范围内选择一个最佳的方法进行编译,还能使虚拟机在编译队列监视的过程中有更多的机会及时删除不再活跃的方法。因此,调度时对整个编译队列进行遍历能够获得更多的性能提升,不会对系统产生负面影响。

为了对 QWSS 算法在运行时引入的额外开销进

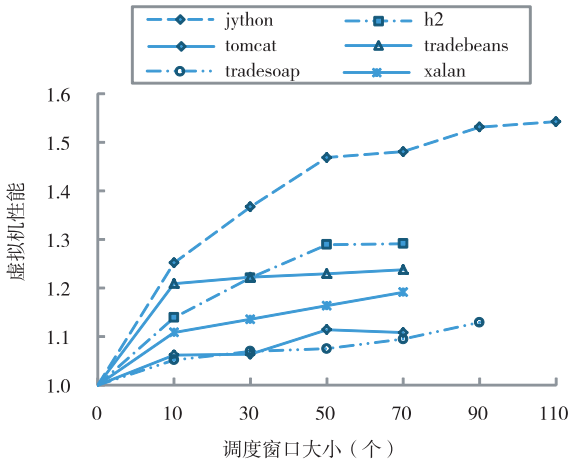

图 5　虚拟机窗口调度性能曲线

行更加细致的量化分析,本文在 DaCapo 测试集上进一步对虚拟机的动态编译时间和编译调度时间进行了统计分析,实验结果如表 2 所示。从表 2 中的数据可以看出,QWSS 算法的编译调度时间仅占虚拟机动态编译时间的 0.02% ~0.07%,并且绝大多数情况下,算法总的调度开销不超过编译开销的 0.05%。因此,用 QWSS 算法时中由优先级计算、编译队列遍历和监视等产生的额外运行时开销十分微小,对实际系统的影响是可以忽略的。

表 2　QWSS 算法运行时开销度量

测试项目	编译时间(μs)	调度时间(μs)	调度/编译(%)
avrora	4137750	2982	0.07
fop	5651982	1881	0.03
h2	24429797	7055	0.03
jython	19448120	5222	0.03
luindex	2292524	621	0.03
lusearch	10543030	4037	0.04
pmd	1443182	483	0.03
sunflow	2795397	1065	0.04
tomcat	27577540	6129	0.02
tradebeans	42145729	10639	0.03
tradesoap	102656638	53329	0.05
xalan	11369550	7090	0.06

5　结　论

本文从理论上给出了一个通用的动态编译调度模型,并在此基础上提出了基于 Size-Speed 的动态编译调度算法。此外,针对目前广泛使用的基于阈值的热点识别技术所存在的缺陷,本文提出了在调度时进行编译队列监视的方案来减少额外的编译开销。实验结果表明,编译队列监视下的 Size-Speed 动态编译调度算法可以有效提高 HotSpot 虚拟机的启动性能和响应速度。在 DaCapo 的测试中,虚拟机总体性能提升了 12.4%,单项性能提升幅度最高可达 54.3%。本文的算法通用性强,实现代价低,很容易推广应用于其它虚拟机系统,对优化虚拟机动态编译系统具有很好的参考意义。未来的工作包括研究如何加快虚拟机对热点方法的识别速度,以及对垃圾回收算法进行优化以提高虚拟机性能等。

参考文献

[1] 林闯,苏文博,孟坤等. 云计算安全:架构、机制与模型评价. 计算机学报,2013,36(9):1765-1784

[2] Lindholm T, Yellin F, Bracha G, et al. The Java™ Virtual Machine Specification. http://docs. oracle. com: Oracle Corporation, 2013

[3] Arnold M, Fink S J, Grove D, et al. A survey of adaptive optimization in virtual machines. *Proceedings of the IEEE*, 2005, 93(2): 449-466

[4] Jantz M R, Kulkarni P A. Exploring single and multilevel JIT compilation policy for modern machines. *ACM Transactions on Architecture and Code Optimization*, 2013, 10(4): 1-29

[5] Kulkarni P A. JIT compilation policy for modern machines. In: Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications, New York, USA, 2011. 773-788

[6] Stadler L, Duboscq G, Mössenböck H, et al. Compilation

queuing and graph caching for dynamic compilers. In: Proceedings of the 6th ACM Workshop on Virtual Machines and Intermediate Languages, New York, USA, 2012. 49-58

[7] Hölzle U, Ungar S. Reconciling responsiveness with performance in pure object-oriented languages. *ACM Transactions on Programming Languages and Systems*, 1996, 18 (4): 355-400

[8] Inoue H, Hayashizaki H, Wu P, et al. Adaptive multi-level compilation in a trace-based Java JIT compiler. In: Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications, New York, USA, 2012. 179-194

[9] Harris T. Controlling run-time compilation. In: Proceedings of the IEEE Workshop on Programming Languages for Real-Time Industrial Applications, Madrid, Spain, 1998. 75-84

[10] Kulkarni P, Arnold M, Hind M. Dynamic compilation: the benefits of early investing. In: Proceedings of the 3rd International Conference on Virtual Execution Environments, New York, USA, 2007. 94-104

[11] Oracle. Java SE Development Kit 8. http://www.oracle.com/: Oracle Corporation, 2014

[12] Kotzmann T, Wimmer C, Mössenböck H, et al. Design of the Java HotSpot™ client compiler for Java 6. *ACM Transactions on Architecture and Code Optimization*, 2008, 5 (1): 1-32

[13] Paleczny M, Vick C, Click C. The Java HotSpot™ server compiler. In: Proceedings of the 2001 Symposium on Java™ Virtual Machine Research and Technology Symposium, California, USA, 2001. 1-1

[14] Nagpurkar P, Krintz C, Hind M, et al. Online Phase Detection Algorithms. In: Proceedings of the International Symposium on Code Generation and Optimization, Washington, USA, 2006. 111-123

[15] Schilling J L. The simplest heuristics may be the best in Java JIT compilers. *ACM SIGPLAN Notices*, 2003, 38(2): 36-46

[16] Blackburn S M, Garner R, Hoffmann C, et al. The DaCapo benchmarks: Java benchmarking development and analysis. In: Proceedings of the 21st Annual ACM SIGPLAN Conference on ObjectOriented Programming SystemsLanguages and Applications, New York, USA, 2006. 169-190

A Size-Speed dynamic compilation scheduling algorithm under compilation queue monitoring

Fu Jie[*,**,***], Liao Bin[*,**,***], Chen Xinke[*,**,***],
Jin Guojie[****], Zhang Longbing[*,**,***], Wang Jian[*,**,***]

([*] University of Chinese Academy of Sciences, Beijing100049)

([**] State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences, Beijing100190)

([***] Institute of Computing Technology, Chinese Academy of Sciences, Beijing100190)

([****] Loongson Technology Corporation Limited, Beijing 100095)

Abstract

To improve the startup performance and the response speed of virtual machines, the optimization of dynamic compilation was studied, and a novel Size-Speed dynamic compilation scheduling algorithm under compilation queue monitoring was proposed. The Size-Speed scheduling algorithm uses both the size and the running speed of a method

to be compiled in a program to compute the priority of its compilation to make the scheduling not only adapt to the dynamic behavior of the program, but also improve the dynamic compilation throughput. Moreover, the algorithm performs the compilation queue monitoring during the scheduling, and reduces the overhead of dynamic compilation by calculating the activity of each method in the compilation queue to remove the inactive methods. The experimental results showed that the proposed algorithm significantly improved the startup performance and response speed of a virtual machine system. For the benchmark of DaCapo, the overall performance of the virtual machine was improved by 12.4%, and the highest performance boost was up to 54.3%. The algorithm is highly versatile, easy to implement and can be applied to most dynamic compilation systems.

Key words: virtual machine, dynamic compilation, compilation scheduling, startup performance, response speed, optimization