第 33 卷 第 7 期 计 算 机 工 程 2007 年 4 月
Vol.33 No.7 Computer Engineering April 2007
·软件技术与数据库· 文章编号：1000—3428(2007)07—0084—03 文献标识码：A 中图分类号：TP311.52

基于龙芯 2 号的 Java 虚拟机的移植与优化

刘 超
（中国科学院计算技术研究所，北京 100086）

摘 要：Java 语言作为一种跨平台的编程语言在企业应用开发、桌面应用开发及嵌入式开发上获得了广泛的应用。为了在龙芯上运行 Java 程序，将 Sun HotSpot Java 虚拟机移植到了 Linux/龙芯 2 上，该文描述了移植过程中的主要工作、遇到的问题及解决的方法和优化工作。
关键词：Java；Java 虚拟机；Linux；龙芯 2 号；JIT

Transplantation and Optimization of Java Virtual Machine Based on GodSon2

LIU Chao
(Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100086)

【Abstract】 As a cross-platform programming language, Java has received extensive application in enterprise development, desktop development and embedded software development. To run Java programs on Linux/GodSon2, this paper ports Sun HotSpot Java virtual machine. It describes experiences in porting HotSpot JVM to Linux/GodSon2 and some optimization work.
【Key words】 Java; Java virtual machine; Linux; GodSon2; JIT

不同于传统编程语言，Java程序被编译成与平台无关的字节码，由在特定平台上实现的Java虚拟机(JVM)[1]来运行。

龙芯使用了 MIPS 指令集的高性能通用 CPU，龙芯的一个重要应用方向是企业应用。目前在企业应用上，Java/J2EE 独占鳌头。很多大型软件如 mozilla、openoffice 也依赖于 Java 虚拟机。但是在工作之前尚无支持 MIPS 的完整的 JVM。为龙芯开发一个高效的 JVM 对龙芯的发展具有十分重要意义。

目前有很多 Java 虚拟机项目，如 Kaffe 和 Jikes RVM 等，由于 Java 类库、性能等原因，本文选择移植 Sun HotSpot JVM。目前 HotSpot JVM 支持 IA32、IA64、AMD64 和 SPARC 4 种平台，移植主要根据 IA32 版本。

1 HotSpot JVM 概述

HotSpot JVM 有客户版和服务器版 2 个版本，差别主要在于 JIT 编译器。服务器版 JIT 采用了一些高级的编译技术，适合运行服务器程序。客户版实现了一个简单快速的 JIT 编译器。目前移植的是客户版，以后的工作包括移植服务器版。

1.1 垃圾收集器和内存模式

Java中垃圾收集器负责释放内存，解决了传统编程语言中明确释放内存带来的内存泄漏等问题。HotSpot实现了精确的分代式垃圾收集器，所有Java对象包括自反数据[1]都用oopDesc表示，虚拟机中直接使用oopDesc指针而不是句柄来访问对象，这要求垃圾收集后改写指针。HotSpot将堆分为 3 块，分别是垃圾收集堆、代码堆和自由堆。前 2 个堆在虚拟机启动时分别被初始化。垃圾收集堆就是由垃圾收集器管理的堆，所有oopDesc都是从这个堆中分配的。代码堆存放虚拟机运行过程中通过汇编器产生的代码，它以CodeBlob为单位管理这些代码。自由堆即堆中的剩余部分，由new/delete和malloc/free管理。

1.2 线程与同步

每个Java对象关联一个监视器[1]，程序可以在其上执行加锁、解锁等同步语义，这是一笔非常可观的资源开销。同步还带来了运行时的开销，研究表明Java程序中超过 70%的同步实际上是不必要的。只有很少一部分对象会被用于同步，大多数实现采用了延迟创建监视器的策略。文献[3]基于大多数加锁是目标对象没有被加锁的情况，提出了瘦锁和胖锁的概念，加锁一个没有被加锁的对象仅是简单地把线程ID写入瘦锁中，如果发现已被加锁就需要生成胖锁。HotSpot采用了类似于文献[3]的方案，不同的是HotSpot中oopDesc的第 1 个字都被用来实现锁，称为markOopDesc，减少了对象的头部大小。HotSpot中每个Java线程对应于一个操作系统线程，这决定了同步语义必须使用操作系统的机制来实现。对于服务器应用来说，这种实现可能成为一个瓶颈。

1.3 字节码的执行方式

字节码执行方式主要有解释和JIT(即时编译)两种方式。有两种实现解释器的方式：switch语句和线形解释。线形解释器中每个字节码对应一段解释代码，每执行完一字节码都以下一字节码为索引，取得其解释代码地址，并跳转到那里运行。JIT以方法为单位将Java程序编译成本地代码，其执行速度大约是解释执行的 10~15 倍。由于程序局部性原理认为决定程序性能的关键在于小部分代码，因此JIT执行所有方法并不合适。这就产生了混合执行方式的虚拟机，即根据一定的启发因素决定是否JIT执行某方法，这能达到 10%~15%的性能提升[2]。

基金项目：国家“973”计划基金资助项目(2005CB321600)；国家自然科学基金杰出青年基金资助项目“计算机系统结构研究”(60325205)
作者简介：刘 超(1982－)，男，硕士生，主研方向：计算机系统结构
收稿日期：2006-03-22 **E-mail：**yjl_net@ict.ac.cn

HotSpot采用了混合执行方式。在虚拟机启动时产生一个线形解释器，图 1 显示了其结构。解释器维护一个表达式栈顶状态[1]，每对字节码和状态的组合都有一段解释代码，解释执行字节码的同时改变状态。一些经常出现的字节码组合被合并成一条快速字节码，以减少对堆栈的操作和跳转的次数。

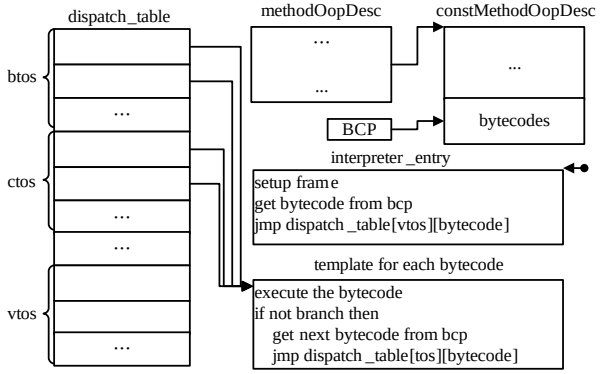

图 1 HotSpot 解释器结构

图 2 给出了 JIT 编译器的结构。编译由类 Compiler Broker 发起，当一个方法的调用次数和方法内执行的回跳次数之和达到一个值时(称为溢出)就会调用它产生一个 CompilerTask。这有两种情景：(1)方法调用时溢出；(2)回跳溢出，此时需要堆栈替换(On Stack Replacement, OSR)。编译线程(默认一个)从 CompileQueue 中取出 CompilerTask 编译。

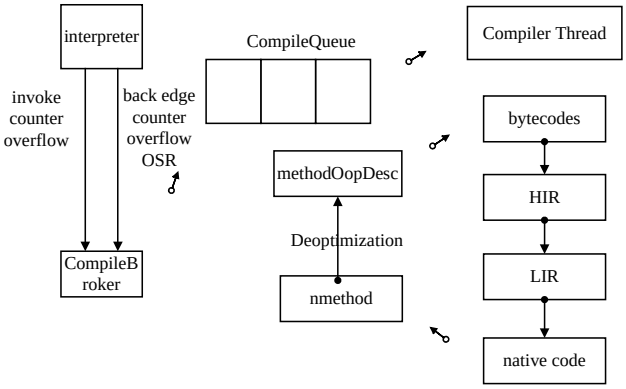

图 2 HotSpot JIT 编译器结构

编译的过程包括从字节码到 2 级中间表示 HIR 和 LIR，再从 LIR 到本地代码。内联是在 HIR 层完成的，寄存器分配是在 LIR 层完成的。在产生 HIR 和 LIR 之后各有一次优化过程，仅实现了非常有限的优化如空指针检查优化、代码重排优化、窥孔优化等。编译完成后需要代码拷贝和重定位。

2 HotSpot 在龙芯上的实现

本部分将描述在移植 HotSpot 到龙芯过程中所做的主要工作，遇到的问题和解决的方法。

2.1 MIPS 汇编器的实现

HotSpot 中所有本地代码(包括解释器和 JIT 编译器编译出的代码)都是用一个汇编器产生的。移植的首要工作是实现一个汇编器。这包括一个基本的汇编器和两个高级汇编器。基本汇编器实现指令的产生并提供标签(label)、转移指令延迟槽、调用 C 函数、原子比较交换操作和内存分配等功能。2 个高级汇编器分别针对解释器和 JIT 编译器，为它们提供特殊支持，如 profile、对常量池的操作、空指针检查、同步支持等。为了解决延迟槽问题，在汇编器中加入一个 delayed 状态，转移指令会设置该状态，必须明确清除它才能产生下一条指令。

2.2 栈帧与堆栈对齐

在 HotSpot 虚拟机中存在 3 种栈帧，分别为解释栈帧、JIT 栈帧和 C/C++本地栈帧。前二者是 Java 栈帧，需要由我们来定义。这有 2 个基本要求：(1)在发生 Java 异常、垃圾收集等情况下，虚拟机能够快速遍历所有活跃的栈帧；(2)本地栈帧必须 8B 对齐，这是对 MIPS ABI 的要求。图 3 显示了本文定义的解释栈帧和 JIT 栈帧示意图。

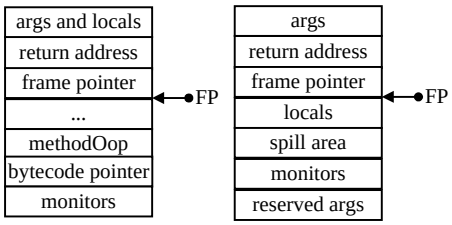

图 3 解释栈帧和 JIT 栈帧

关于本地堆栈 8B 对齐问题，解决方法是将堆栈指针保存在线程数据结构中，然后对齐堆栈。从本地栈帧返回时，再从线程数据结构中恢复堆栈指针。

2.3 JIT 编译器的实现

JIT 编译器需要在指令中嵌入 32 位立即数，MIPS 指令无法做到这点。本文使用 2 条各包含 16 位立即数的指令来模拟 32 位的立即数。这个解决方案产生了另外一个问题，垃圾收集器总是认为目标处是一个 32 位立即数(oopeDesc 指针)，并直接修改之。将这些指针都放在代码后的一块指针区中，垃圾收集器修改指针区。MIPS 指令大多是三操作数指令，这对 JIT 编译器产生了很大的影响。首先，源寄存器的值不会被破坏，减少了寄存器分配和寄存器拷贝。其次，原先需要两条或多条指令实现的功能，使用 1 条指令就可完成。三操作数指令的 1 个典型应用是分支代码的生成。HotSpot 将分支分为比较和跳转 2 个步骤，这 2 步在 LIR 上就已分开。MIPS 中分支指令可以通过一条指令完成，本文改写了 LIR 的框架，将比较和跳转 2 个 LIR 合并。

由于 HotSpot 实现了精确的垃圾收集，因此 JIT 编译器必须告诉垃圾收集器栈桢中哪些位置是对象指针，这个信息称为 gc map。不能为每个代码位置都记录一个 gc map；JIT 编译器仅在 gc point 处产生 gc map。为了让垃圾收集器得到这些信息，必须确保垃圾收集器在垃圾收集时得到的地址和 gc map 的地址一一对应，具体过程涉及到了 MIPS 指令细节，不再详述。

3 性能优化

在移植工作完成之后，本文做了初步的优化工作，下面将详细描述这些优化及其结果。

3.1 寄存器优化

Java 字节码执行过程中(无论是解释还是 JIT)需要调用 C/C++代码完成一些复杂的功能，这个过程中需要保存和恢复寄存器的值。原来的做法是将寄存器的值保存在堆栈中，本文通过分配 callee-saved 寄存器避免了堆栈操作。因为寄存器非常少，所以 IA32 版 JIT 编译器需要频繁地分配临时寄存器，将 3 个寄存器定义为临时寄存器，减少了寄存器分配的次数。关于寄存器的另一个优化是为 Java 的线程对象分配一个专用的寄存器。前文提到每个线程有一个关联的线程数据

结构，在 HotSpot 中对其访问非常频繁。最初的实现用堆栈指针的高 20 位来索引一个数组以得到该指针，这需要 6 条指令，包括一条内存访问指令，维护这个数组也有开销。为线程对象分配一个专用寄存器几乎是零开销的。

3.2 数组越界检查优化

Java要求每次数组访问之前都进行数组越界检查[1]。原先的实现中是在数组访问之前判断是否越界，如越界则抛出Java异常。由于数组越界极少，因此这些检查大多是多余的，基于这一点本文优化了不发生异常的情况。MIPS中有一条tge指令，表示大于等于则发生异常，这正好满足了我们的要求。如果数组访问越界，tge指令使得CPU向操作系统发出异常，改动Linux内核将该异常转换为本文定义的信号SIGTGE，并发送给应用程序(即HotSpot)。在虚拟机中处理这个信号，将它转换为Java数组越界异常并抛出。这样，只需要一条指令就可完成数组越界检查。

3.3 基本块之间优化

JIT 编译器的优化是基于基本块的，基本块之间不能传递优化信息。实现全局优化是不现实的。本文发现很多基本块只有一个前驱，称为单前驱基本块。这种情况下传递优化信息几乎是没有代价的。该优化的关键是如何发现单前驱基本块。本文设计了一个算法，只需简单修改原先的分析基本块的代码就可发现单前驱基本块。伪代码如下：

```
foreach bytecode in method
    if 当前位置已标志为基本块的开始 && 前一条字节码不是分支指令 then
        当前位置的基本块不是单前驱基本块
    if 是条件分支类指令 then
        标志分支目标和下一条指令为基本块的开始
    if 是向后的分支指令 then
        那么目标处的基本块不是单前驱基本块
    else if 是跳转类指令
            标志目标处为基本块的开始
        if 是向后的跳转 then
            那么目标处的基本块不是单前驱基本块
(如果试图标志一个已经标志过的位置，那么该位置处的基本块不是单前驱基本块)
end
```

3.4 优化结果

表 1 给出了上述优化在 SPEC JVM98 基准测试程序 s100 数据集上的结果。

表 1 SPEC JVM98 s100 优化结果

	compress	jess	db	javac	mpegaudio	mtrt	jack
优化前	8.45	14.8	6.06	4.73	18.9	22.4	15.2
寄存器优化	8.79	15.7	6.49	4.81	19.5	23.6	15.9
数组越界优化	8.81	15.6	6.56	4.82	21.2	23.1	15.9
基本块优化	8.86	15.7	6.60	4.83	19.6	23.7	16.0
总效果	9.23	16.7	7.33	5.02	22.6	24.8	16.9

4 总结

本文描述了移植 HotSpot JVM 到 Linux/龙芯 2 过程中的经验和已做的初步优化工作。移植工作主要包括汇编器、解释器和 JIT 编译器的实现，解决了如堆栈对齐、分支指令、原子比较交换操作实现、在指令中嵌入 32 位数据、JIT 编译器对垃圾收集的支持等问题。移植完成后做了初步的优化，包括寄存器分配、数组越界检查优化以及基本块之间的优化等。从总体上说，目前的客户版 JIT 编译器生成的代码性能较差，以后的工作包括优化 JIT 编译器。为了高效地运行服务器程序，还需要移植服务器版虚拟机。

参考文献

1 Lindholm T, Yellin F. The Java Virtual Machine Specification[M]. California, USA: Addison-Wesley, 1999.

2 Radhakrishnan R, Vijaykrishnan N, John L K, et al. Architectural Issues in Java Runtime Systems[C]//Proceedings of the International Symposium on High Performance Computer Architecture. 2000: 387-398.

3 Bacon D F. Konuru R, Murthy C, et al. Thin Locks: Featherweight Synchronization for Java[C]//Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation. 1998: 258-268.

4 Jones R, Lins R. Garbage Collection: Algorithms for Automatic Dynamic Memory Management[M]. John Wiley&Sons Limited Liability Company, 1996.

5 Kim K S, Hsu Y. Memory System Behavior of Java Programs: Methodology and Analysis[C]//Proc. of the ACM SIGMETRICS. 2002.

（上接第 83 页）

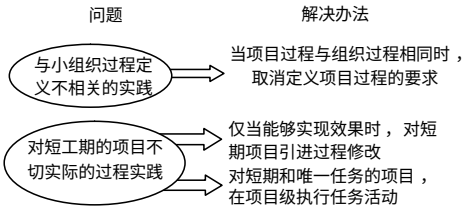

图 7 过程问题的解决办法

4 小结

本文探讨了面向小型软件组织和项目的 CMM 裁剪研究，目的是为小组织和小项目建立一个更为适用的 CMM 裁剪模型。由于裁剪的基本原则是维持软件能力成熟度模型的结构和要求，并未从根本上改变模型本身；同时，适合于小组织、小项目的剪裁实践在许多情况下也同样适用于其它组织和项目，因此该剪裁模型将不仅适用于小组织、小项目，也适用于其它更大范围的组织和项目。

参考文献

1 卡耐基梅隆大学软件工程研究所. 能力成熟度模型(CMM): 软件过程改进指南[M]. 刘孟仁, 译. 北京: 电子工业出版社, 2001-07.

2 GJB5000-2003 军用软件能力成熟度模型[S]. 总装基础局, 2003.

3 Dynamic CMM for Small Organizations Implementation Aspects[R]. Terttu. Orci., 2000.

4 Capability Maturity Model for Small Organizations Level 2 Version 1.0[R]. Terttu. Orci., 2000-09-10.

5 Paulk M C. Using the Software CMM with Good Judgment[Z]. 1998.

6 Paulk M C. Using the Software CMM in Small Organizations[Z]. 1998.