doi:10.3772/j.issn.1002-0470.2015.07.002

基于 NUMA 架构的解释器访存优化设计与实现①

任 彤②* ** ***** 傅 杰* ** *** 靳国杰****

(* 中国科学院大学 北京 100049)

(** 中国科学院计算技术研究所 北京 100190)

(*** 计算机系统结构国家重点实验室 北京 100190)

(**** 龙芯中科技术有限公司 北京 100190)

(***** 中国电子设备系统工程公司研究所 北京 100141)

摘 要 为了提高非一致内存访问(NUMA)架构虚拟机解释器的访存性能,研究了解释器在 NUMA 架构下的访存优化技术,提出了一种 NUMA 架构下的解释器访存优化方案,而且设计并实现了解释器的静态指令分派优化方法和动态指令分派优化方法。根据这一方案虚拟机启动时首先获取 NUMA 节点信息,并在每个 NUMA 节点中自动生成解释器所需的全部数据结构;解释器在运行时,通过静态或动态的指令分派技术来实现其执行线程在 NUMA 节点上访存的局部化。试验结果表明,上述方法能够显著提升解释器在 NUMA 系统中的性能。在 DaCapo 测试集上的总体性能提升了 8%,最高性能提升幅度高达 23%,而且算法实现代价低,适用于绝大多数的 NUMA 服务器系统。

关键词 非一致内存访问(NUMA), 虚拟机, 解释器, 响应速度, 启动性能, 访存优化

0 引 言

随着计算机网络、存储与通信等技术的飞速发展,人类已进入云计算和大数据时代[1]。云计算环境下,高度异构的设备使得人们对程序的可移植性有着更加迫切的需求。人们希望所发布的应用程序能够在各式各样的设备上运行,而无需考虑底层硬件实现的差别。于是,基于虚拟机实现的编程语言(如 Java 和 C#等)越来越受到程序员的青睐。与传统 C/C++等静态编译模式不同,基于虚拟机技术的程序源代码并不直接被编译成机器代码,而是被编译为功能语义等价的平台无关的中间代码表示[2](如 Java 使用的字节码)。程序的运行是通过虚拟机对这种平台无关的中间代码模拟执行来实现的。虚拟机技术使得程序具有"一次编译,到处运行"的高度可移植性。并且,通过虚拟机内部设计的各种检测和监视机制(如对空对象访问和数组越界访问的检查),使得程序的运行更加安全可靠。

大数据时代数据规模呈指数级增长,科学计算和事务处理对计算机系统的性能提出了更为苛刻的要求[3]。为了应对不断增长的计算需求,多处理器计算机系统已经成为主流。对称多处理器[4](symmetric multi-processor, SMP)系统是一种常见的多处理器计算机。在 SMP 系统中所有的处理器共享系统总线,对内存的访问延迟相同。而当处理器数目增大时,SMP 系统对系统总线的竞争冲突增大,系统总线成为制约系统性能的瓶颈。因此,SMP 系统

① 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, 2014ZX01020201),国家自然科学基金(61221062, 61133004, 61173001, 61232009, 61222204, 61432016)和 863 计划(2012AA010901, 2012AA011002, 2013AA014301)资助项目。

② 男,1979 年生,博士生,工程师;研究方向:多核处理器性能分析与优化;联系人,E-mail: rentong_1979@126.com

(收稿日期:2015-02-20)

通常仅支持几个到十来个处理器,其可扩展性较差。为了充分发挥多处理器系统的优势,增强系统的并发度和可扩展性,非一致内存访问(non-uniform memory access, NUMA)系统[4]应运而生。NUMA 系统由于更易于大规模并行,更适合于构建分布式系统,并且还解决了 SMP 系统的可扩展性问题,因而已成为高端服务器的主流设计架构。因此以 NUMA 架构的机器作为服务器的基础硬件,以虚拟机作为服务器应用的基础软件,已成为当前云计算和大数据产业生态的主流[5]。而结合 NUMA 架构的硬件特点和虚拟机的软件特性,对整个服务器应用系统进行分析和优化,则是提升服务器服务质量的关键。虚拟机的执行引擎通常包括解释器和编译器两种类型[6]。相对于编译器,解释器具有结构简单、易于实现和可维护性强等诸多优势,是实现虚拟机执行引擎的首选方案。例如,广泛使用的 Python 语言,其执行引擎仅由解释器来实现。此外,由于编译器的编译行为具有滞后性,解释器往往决定了服务器的响应速度和系统的启动性能。因此结合 NUMA 架构的硬件特性提升虚拟机解释器的性能,对于缩短服务器对用户的响应时间,加快应用系统的启动速度具有十分重要的意义。本文研究了解释器在 NUMA 架构下的优化技术,提出了 NUMA 结构下虚拟机解释器访存优化的思路。试验表明,采用本文提出的设计思路能够增强 NUMA 系统中解释器访存的局部性,从而提高虚拟机的性能。

1 相关工作

图 1 展示了一个典型的 NUMA 架构的计算机系统。该系统中总共有两个 NUMA 节点:节点 0 和节点 1。每个节点由一个 CPU 处理器和一个直接相连的存储器组成。不同节点间的 CPU 通过高速互联总线相连。

从图 1 不难看出,NUMA 系统中存储系统在物理结构上具有分布式的特性。分布的存储系统使得处理器对不同节点中的内存访问具有不同的访存延迟。例如,CPU0 访问节点 1 中内存 1 的延迟将远大于 CPU0 对本节点中内存 0 的访问。这是由于当

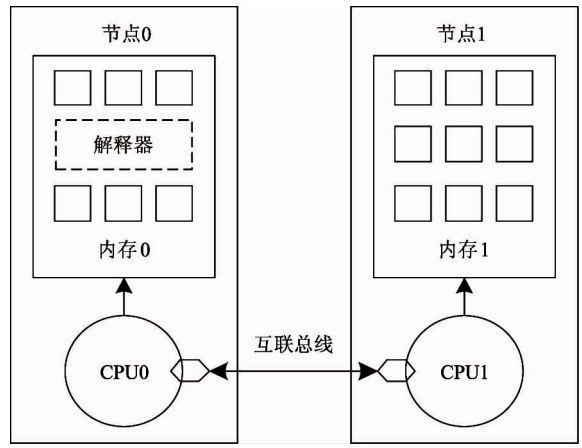

图 1 NUMA 系统示意图

CPU0 访问内存 1 时,内存 1 中的数据需要通过高速互联总线进行传输。数据在高速互联总线上的传输带宽与速率远小于相同节点内处理器对内存的直接访问。因此,可以将 NUMA 系统中的访存划分为两类:本地存储访问和跨节点存储访问。本地存储访问,是指访存目标地址所在节点和发起访存请求的处理器所在节点相同的存储访问;跨节点存储访问,是指访存目标地址所在节点和发起访存请求的处理器所在节点不同的存储访问。

为了增强虚拟机的可扩展性和可维护性,现代高性能虚拟机通常采用在虚拟机启动时自动生成解释器的策略。例如,Oracle OpenJDK 的 HotSpot 虚拟机[7]启动时会随机在内存中生成一个解释器。进一步,对于 NUMA 系统,现有方法会随机在某个节点的内存中生成解释器。如图 1 所示,虚拟机随机在节点 0 的内存中生成了解释器。此时,若解释器线程运行在 CPU1 上,由于 CPU1 在解释执行的过程中需要频繁访问节点 0 中所生成的解释器的数据结构,故导致大量跨节点访存行为,降低了解释器在 NUMA 系统中的性能。因此,由于忽略了 NUMA 硬件架构的特殊性,现有虚拟机中对解释器的设计和实现方式效率低下。

近年来,由于云计算和大数据时代的到来,针对 NUMA 硬件架构的系统优化逐渐成为研究的热点[8,9]。首先,服务器上广泛使用的 Linux 操作系统针对 NUMA 架构的特殊支持日益完善[5]。从 Linux 内核 2.5 版本开始,Linux 操作系统在进程调度器、存储管理器以及应用程序 API 等方面进行了大量的

NUMA 优化工作,至今都还在不断延续和深入发展。而对于 Oracle OpenJDK 的 HotSpot 虚拟机,在 2014 年发布的 JDK8 版本中,研究人员针对 NUMA 结构特性重点对其垃圾收集器进行了卓有成效的优化[9]。此外,对于实际生产环境中的上层应用系统,数据库厂商、网络服务提供方和大型数据中心等也有大量关于如何在 NUMA 架构上提升系统性能的研究和探索。上述大量研究和实践表明,减少 NUMA 系统中跨节点存储访问的数量是提升 NUMA 系统性能的关键。

综上,提升虚拟机解释器性能对优化服务器应用具有重要意义。而在 NUMA 架构下,访存优化是提高虚拟机解释器性能的重要手段,其关键在于减少解释器在 NUMA 系统中的跨节点存储访存。

结合前人对 NUMA 系统的经验,针对现有虚拟机中解释器在 NUMA 架构上的设计缺陷,本文提出了一种 NUMA 结构下虚拟机解释器访存优化方法。本文首先分析了解释器的基本工作原理。然后提出了 NUMA 架构下访存优化的解释器的设计思想,并在此基础上,进一步提出了静态指令分派和动态指令分派等两种实现方案。最后通过对比试验,分析和验证了本文方法的有效性。

2　解释器原理概述

本文以 Oracle JDK8[7]的 HotSpot 虚拟机为例介绍解释器执行的基本过程。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两个即时编译器(client JIT 和 server JIT)[10,11]构成。Java 程序中所有方法均由解释器开始执行,而即时编译器仅编译执行频度较高的方法。

图 2 展示了解释器的执行过程。解释器按照取指令、指令译码、指令分派和执行的顺序对虚拟机指令进行模拟执行[12]。HotSpot 虚拟机采用了基于模板的解释器,图 3 展示了模板解释器的指令分派过程。图 3 中,被解释执行的方法由一系列虚拟机指令构成,例如 iadd(整型加法指令),isub(整型减法指令),fadd(单精度浮点型加法指令),dadd(双精度浮点型加法指令)等。每条虚拟机指令均对应一个指令模板,解释器对某条虚拟机指令进行解释执行时,需要跳转到该虚拟机指令对应的模板,随后执行模板中相应的本地机器指令。当解释器执行完一个模板中的本地机器指令时,需要跳转到下一条虚拟机指令所对应的模板继续执行。这种解释器由一个指令模板跳转到另一个指令模板的过程,称为解释器指令分派[12]。

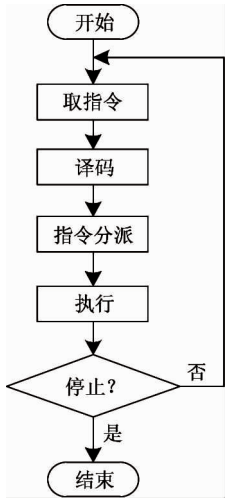

图 2　解释器执行过程

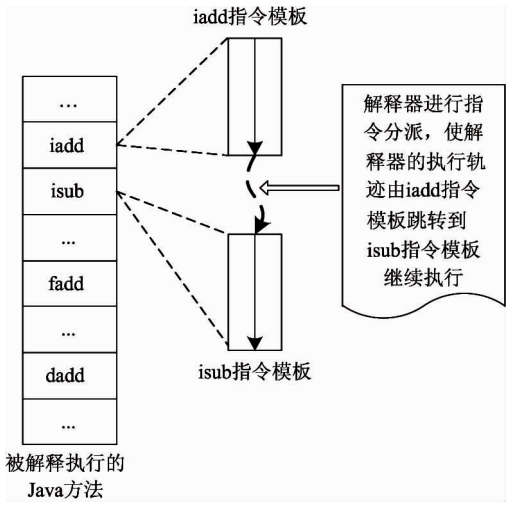

图 3　**HotSpot** 解释器指令分派过程

从存储访问的角度看,解释器指令分派本质上是对解释器模板数据结构的频繁访问过程。由于解释器每执行一个字节码便进行一次解释器指令分派,因而实际系统中由解释器指令分派引起的存储访问数量十分庞大。因此,在 NUMA 系统中减少由

于解释器指令分派导致的跨节点存储访问的数量，是增强解释器在 NUMA 架构上的访存局部性和提升解释器性能的重要途径。

3 NUMA 架构的解释器优化

3.1 NUMA 架构解释器访存优化基本思想

解释器指令分派需要对解释器中的相关数据结构的频繁访问，在 NUMA 架构的计算机中容易导致大量的跨节点存储访问。为了减少甚至避免对解释器数据结构的跨节点访问，一种简单有效的方法是解释器线程在某个 NUMA 节点执行时仅访问该节点中的数据结构。图 4 展示了本文解释器访存优化的基本原理：解释器初始化时，在每个 NUMA 节点上均生成一个完整的解释器；NUMA 节点上的解释器线程执行时仅访问本节点内部的解释器中的数据结构。

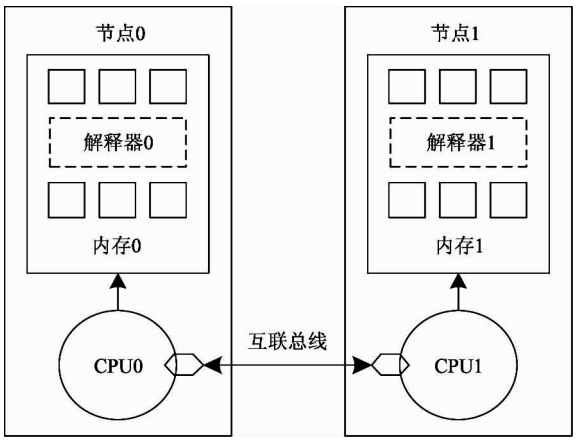

图 4 NUMA 架构下解释器访存优化基本原理

在虚拟机启动时，很容易在不同的 NUMA 节点中各生成一个解释器。如何保证解释器线程在执行时仅访问本节点内部的数据结构，是实现上述优化的关键。为此，本文提出了两种实现方案：(1)静态指令分派的解释器访存优化；(2)动态指令分派的解释器访存优化。

3.2 静态指令分派的解释器访存优化

图 5 展示了静态指令分派的解释器访存优化方法，主要涉及解释器生成、解释器线程启动、静态指令分派和操作系统线程调度等 4 个方面。

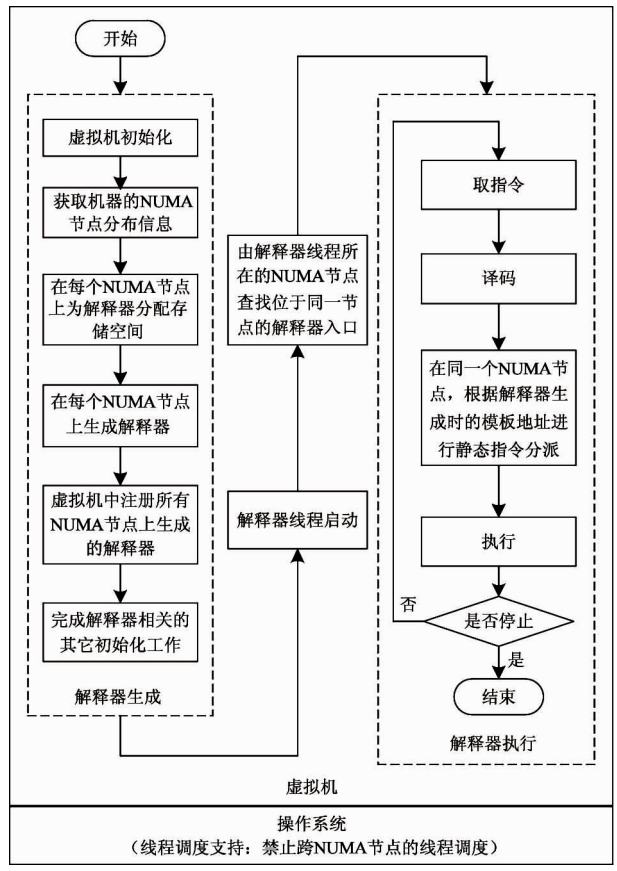

图 5 静态指令分派的解释器访存优化

该方法的关键技术步骤如下：

(1)解释器生成阶段。在每个 NUMA 节点的内存中均生成一套完整的解释器，并记录同一个 NUMA 节点中解释器的所有模板的地址。

(2)解释器线程启动阶段。获取解释器线程所在的 NUMA 节点信息，随后查找该节点的解释器入口地址，并跳转到该入口地址开始解释器执行。

(3)静态指令分派。根据解释器生成时所保存的位于同一个 NUMA 节点中的模板地址进行指令分派。

(4)线程调度。禁止跨 NUMA 节点的线程调度。

该方法在解释器线程启动时获取解释器线程所在的 NUMA 节点，并且在进行解释器指令分派时，仅依赖解释器生成时所保存的位于同一节点中的模板地址信息。显然，该方法的一个前提条件是解释器线程始终在同一的 NUMA 节点上执行。因此，采用本方法需要操作系统线程调度的支持，使得操作系统在进行线程调度时不能出现跨节点的线程迁移

现象。

3.3　动态指令分派的解释器访存优化

图 6 展示了动态指令分派的解释器访存优化方法,主要涉及解释器生成、解释器线程启动、动态指令分派和操作系统线程调度等 4 个方面。

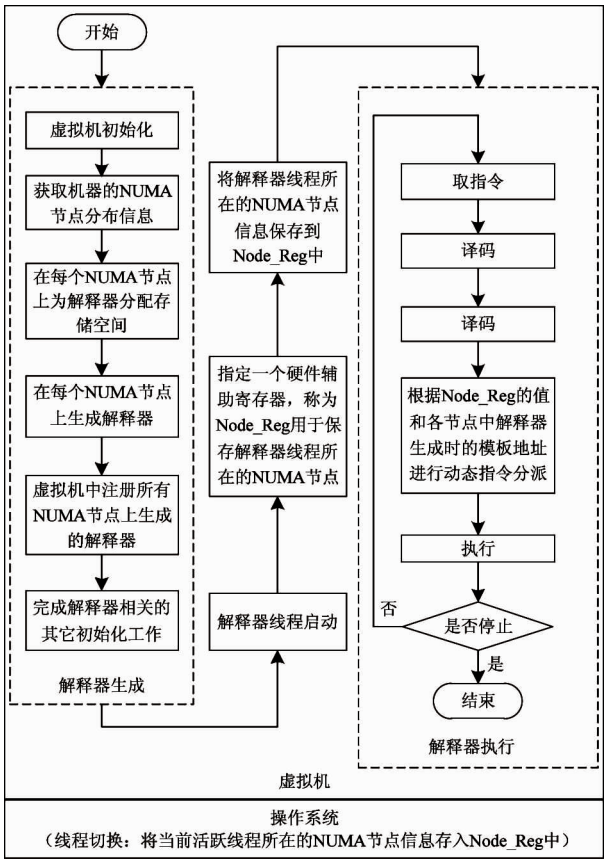

图 **6**　动态指令分派的解释器访存优化

该方法的关键技术步骤如下:

(1)解释器生成阶段。在每个 NUMA 节点的内存中均生成一套完整的解释器,并记录同一个 NUMA 节点中解释器的所有模板的地址。

(2)解释器线程启动阶段。指定一个硬件辅助的寄存器,称为 Node _ Reg;将解释器线程所在的 NUMA 节点信息保存到 Node _ Reg 中。

(3)动态指令分派。根据寄存器 Node _ Reg 的值和各节点中解释器生成时记录的模板地址进行指令分派。

(4)线程调度。线程调度时,将当前活跃线程所在的 NUMA 节点信息存入 Node _ Reg 中。

该方法在一个硬件辅助的寄存器 Node _ Reg 中保存了当前解释器线程所在的 NUMA 节点信息,解释器指令分派时,不仅依赖解释器生成时所记录的模板地址信息,同时还依赖于当前 Node _ Reg 的值。显然,相对于静态指令分派的解释器访存优化,该方法的一大优势是对操作系统的线程调度没有任何附加限制。

4　试验与结果分析

本文使用了 Oracle OpenJDK8 的 HotSpot 虚拟机作为试验平台。HotSpot 虚拟机是产业界和学术界广泛使用的一款高性能 Java 虚拟机。该虚拟机的执行引擎包含解释器和 JIT 编译器,支持纯解释器模式运行、混合模式运行和纯翻译模式运行等三种运行方式。为了更有针对性地测试解释器的优化效果,本研究在测试时启用了“-Xint”参数,指定虚拟机以纯解释器模式运行。

为了验证解释器访存优化的效果,本研究选用了业界权威的 DaCapo-9. 12-bach[13] 测试集进行试验。该测试集中共包含 14 个测试项,其中由于 batik 和 eclipse 引用了与 JDK8 不兼容的低版本类库,因此本文的试验结果将它们排除在外。此外,为了更为全面地展示本文算法的优化效果,试验还对目前广泛使用的大型 Java 软件系统进行了测试,包括 Eclipse、Tomcat、Jetty 和永中 Office 等。其中 Eclipse 为著名的 Java 程序开发平台;Tomcat 和 Jetty 均为网络服务器软件,Tomcat 大量应用于传统的网络服务环境中,而 Jetty 则更多地应用于云计算和分布式网络系统中;永中 Office 为江苏无锡永中软件有限公司用 Java 语言开发的类似于 Microsoft Word 和 WPS 的文字处理软件。

试验中使用了龙芯 3 号处理器[14] 双节点的 NUMA 架构计算机系统,主频 900M Hz,DDR3 内存 8G。操作系统为 Linux,内核版本为 2.6.32。

4.1　试验设置

在进行静态指令分派的解释器访存优化试验时,本研究修改了 Linux 操作系统内核,禁止了线程的跨 NUMA 节点调度。

对于动态指令分派的解释器访存优化,本文使

用了龙芯 3 号处理器中一种称为内容寻址存储器(content addressable memory,CAM)表的硬件支持的结构[15]。CAM 表是使用专用比较电路设计的集成在处理器内部的特殊存储区域。由于其设计简单,实现代价低,现已广泛应用于各种处理器中。CAM 表支持快速比较和查找,其访问速度非常接近于对 CPU 内部通用寄存器的访问。龙芯 3 号处理器中 CAM 表的大小为 64 项,每项由 48 位的查询索引区域和 64 位的查询数据域构成。试验中,使用了 CAM 表的第一项,即 CAM 表索引值为 0 的那一项。此外,本文还对内核线程调度进行了修改,使得在上下文切换时将当前活跃线程所在的 NUMA 节点信息保存到 CAM 表的第一项。对 CAM 表的一次访问仅需一条机器指令即可实现,在龙芯 3 号处理器的高性能流水线中只需一个时钟周期便可完成操作。因此,上下文切换时保存一项 CAM 表内容的开销几乎是可以忽略不计的。

4.2 试验结果

表 1 主要展示了 HotSpot 虚拟机运行 DaCapo 测试集时,解释器由于指令分派而导致的访存数量。从中不难看出,解释器运行时,访存数量十分庞大。

表 1 虚拟机运行时数据

测试项目	访存次数(亿次)	是否多线程
avrora	17.55	否
fop	1.57	否
h2	42.13	是
jython	34.18	否
luindex	1.14	否
lusearch	2.20	是
pmd	0.54	否
sunflow	17.66	是
tomcat	12.69	是
tradebeans	27.88	是
tradesoap	6.39	是
xalan	1.83	是

图 7 展示了解释器优化前后的总体性能。试验结果均以优化前为基准进行了归一化处理。图 7 的试验结果表明,采用静态指令分派的访存优化,解释器总体性能的提升幅度为 7%;而采用动态指令分派的访存优化,总体性能的提升幅度为 8%。由此可见,本文算法对提升 NUMA 架构下解释器的性能具有显著的效果,并且采用动态指令分派的访存优化比采用静态指令分派的访存优化略有优势。

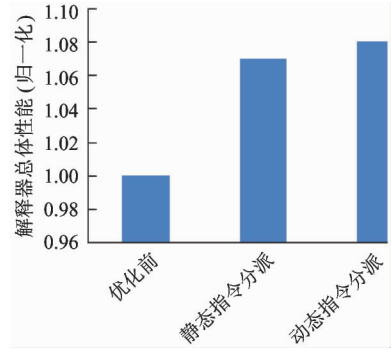

图 7 优化前后解释器总体性能

图 8 展示了解释器在不同优化算法下各个测试项目的性能。从图中可以看出,两种访存优化方法均使得测试集中的所有测试项目获得不同幅度的性能提升效果,而对于 avrora、h2、jython 和 tradebeans 等项目,性能提升幅度超过了 10%,并且最高性能提升幅度高达 23%。此外,从图 8 还可以看出,除 avrora 和 jython 外,动态指令分派的优化效果均优于静态指令分派的优化效果。

表 1 中的度量数据可进一步解释图 8 的试验结果。例如,对于 avrora、h2、jython 和 tradebeans 等测试项目,由于其访存数量非常巨大(超过了 20 亿次),故在试验中获得的性能提升的幅度较大。此外,对于非多线程的测试项目,静态指令分派优化的优化效果和动态指令分派的优化效果大致相当。而对于多线程的测试项目,如 h2、sunflow 和 tradebeans 等,动态指令分派的优化效果明显优于静态指令分派的效果。这是由于采用静态指令分派优化时,将解释器线程的运行限定到了某个单一的 NUMA 节点上,内核进行线程调度的选择范围受限。而动态指令分派对内核线程调度没有额外约束,所有 NUMA 节点均可进行调度。故对于多线程的测试项目,动态指令分派优化的效果通常要优于静态指令分派的效果。

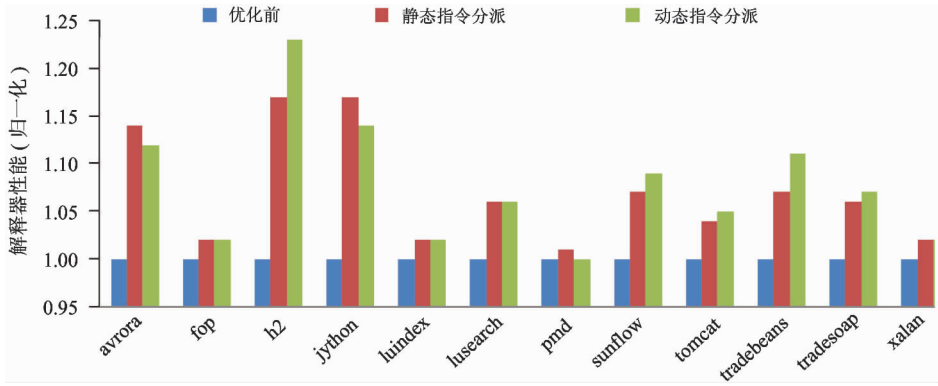

图 **8**　优化前后解释器在 **DaCapo** 测试集上的性能

图 9 展示了本文工作对大型软件启动性能的优化效果。从图中不难看出,本文方法对实际的大型应用软件的启动性能也有明显的提升效果。由于动态指令分派优化能够更加充分地发挥所有 NUMA 节点的性能,故对于实际应用系统而言,动态指令分派的优化效果也优于静态指令分派的优化效果。

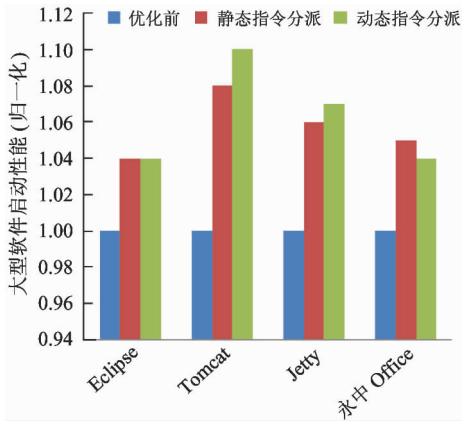

图 **9**　大型软件启动性能测试

栈顶缓存[16](stack caching)是目前和本文最相关的一种解释器访存优化方法。其基本思想是使用通用寄存器将解释器执行堆栈的一个或多个栈顶元素(一般不超过两个)缓存起来,将原来堆栈的栈顶元素访问用寄存器访问替代,从而实现了解释器访存的加速。但是在现代广泛使用的 NUMA 系统中,单纯的解释器栈顶缓存并未考虑解释器访存的局部性问题。本文在解释器栈顶缓存优化的基础上,提出了一种增强解释器在 NUMA 系统中访存局部性的技术,进一步提升了解释器的性能。

5　结　论

软硬件相结合的系统优化,对于提升整个系统的性能,尤其是 NUMA 架构机器上虚拟机的性能,具有十分重要的意义。本文针对 NUMA 架构硬件设计的特殊性,提出了一种 NUMA 架构下解释器的访存优化思路。并在此基础上,设计并实现了两种具体的解释器访存优化实现方法:静态指令分派的访存优化和动态指令分派的访存优化。试验结果表明,两种优化方法均能有效提高 NUMA 架构下解释器的性能。在 DaCapo 的测试中,解释器的总体性能提升了约 8%,最高性能提升幅度高达 23%。同时,由于动态指令分派能充分利用系统中的各个 NUMA 节点,故通常比静态指令分派获得更多的性能提升效果。本文方法实现代价低,并且很容易推广应用于多节点的 NUMA 架构计算机系统,对于软硬件结合的全系统优化具有很好的参考意义。未来的工作将对虚拟机中垃圾收集在 NUMA 架构上的优化作进一步研究。

参考文献

[1] 程学旗, 靳小龙, 王元卓等. 大数据系统和分析技术综述. 软件学报, 2014, 25(9):1889-1908

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

[3] 苏文, 王焕东, 台运方等. 面向云计算的多核处理器存储和网络子系统优化设计. 高技术通讯, 2013, 23(4):360-367

[4] Hennessy J L, Peterson D A. Computer ArchitectureA Quantitative Approach. 4th Edition. Amsterdam/Boston/

Heidelberg/London/New York/Oxford/Paris / San Diego / San Francisco / Singapore / Sydney / Tokyo: Morgan Kaufmann Publishers, 2007. 195-200

[5] Lameter C. NUMA (Non-Uniform Memory Access): An Overview. *Queue*, 2013, 11(7):40-51

[6] 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

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

[8] Liu M, Li T. Optimizing virtual machine consolidation performance on NUMA server architecture for cloud workloads. In: Proceedings of the 41st Annual International Symposium on Computer Architecture, Minnesota, USA, 2014. 325-336

[9] Oracle. Java HotSpot™ Virtual Machine Performance Enhancements. http://docs.oracle.com/javase/7/docs/technotes/guides/vm/performance-enhancements-7.html: Oracle Corporation, 2014

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

[11] 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

[12] Casey K, Ertl M A, Gregg D, et al. Optimizing indirect branch prediction accuracy in virtual machine interpreters. *ACM Transactions on Architecture and Code Optimization*, 2007, 29(6):1-37

[13] 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 Object-oriented programming systems, languages, and applications, New York, USA, 2006. 169-190

[14] 龙芯中科技术有限公司. 龙芯处理器用户手册. http://www.loongson.cn:龙芯中科技术有限公司, 2012

[15] 邱吉. 基于软硬件协同设计的动态翻译系统性能优化技术研究:[博士学位论文]. 北京:中国科学院大学计算技术研究所, 2013. 100-102

[16] Ertl M A. Stack caching for interpreters. In: Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation (PLDI'95), New York, USA, 1995. 315-327

Design and implementation of memory optimization for NUMA based Interpreters

Ren Tong[*,**,*****], Fu Jie[*,**,***], Jin Guojie[****]

([*]University of Chinese Academy of Sciences, Beijing 100049)

([**]Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190)

([***]State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190)

([****]Loongson Technology Corporation Limited, Beijing 100190)

([*****]China Electronic Systems Engineering Corp, Beijing 100141)

Abstract

In order to improve the performance of a virtual machine's interpreter under the non-uniform memory access (NUMA) architecture, a study of memory access optimization was conducted. Then, a scheme for memory optimization of the interpreters under the NUMA architecture was proposed, and based on it, two novel approaches for an interpreter's memory optimization, namely the one using static instruction dispatching and another using dynamic instruction dispatching, were designed and implemented. According to the scheme, the virtual machine collects the information of the NUMA nodes when it starts up, and automatically generates all the data structures the interpreter needs in each NUMA node; when the interpreter is running on a NUMA node, it uses the two mentioned dispatching optimization approaches to realize its local access to the NUMA nodes. The experimental results demonstrated that the proposed scheme can significantly improve the performance of interpreters in NUMA systems. The results of the experiment using the DaCapo showed that the overall performance of the interpreter was improved by 8%, and the highest increment was up to 23%. The proposed memory optimization algorithm is very easyto implement and can be applied to most NUMA systems.

Key words: non-uniform memory access (NUMA), virtual machine, interpreter, response speed, startup performance, memory optimization