doi:10.3772/j.issn.1002-0470.2016.03.002

基于软硬件协同设计的解释器指令分派方法①

傅 杰②* ** *** 靳国杰**** 章隆兵** *** 王 剑** ***

(* 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 为了降低指令分派造成的运行开销以提高解释器的性能,提出了一种采用软硬件协同设计的解释器指令分派方法。其核心思想是在软件层面通过对指令分派表进行优化以消除了代价较高的地址常量加载操作,在硬件层面通过扩展处理器的访存指令进一步实现基于硬件支持的访存加速。软硬件协同设计可以最大限度地降低由指令分派引入的运行时开销,从而提升解释执行的效率。试验结果表明,该方法能够显著提升解释器的性能。对于 SPECjvm98 和 DaCapo 测试集,解释器总体性能提升了11.5%,且单项性能的最大提升幅度高达15.4%。该方法通用性强,实现代价低,适用于现代主流处理器平台上高性能解释器的设计和优化。

关键词 解释器,指令分派,软硬件协同设计,虚拟机,优化

0 引 言

解释器(interpreter)是一种处理计算机语言的系统软件,常用于高级编程语言的实现[1-3]。从 20 世纪 60 年代著名的 Lisp 语言到新兴的 Java、Python 及 JavaScript 等编程语言,均使用解释器构建其执行引擎。由于具有结构简单、易于实现以及运行时资源消耗少等优势[4],解释器被广泛应用于各种实际系统中。解释器和即时(just-in-time, JIT)编译器[5]通常以互补的形式共同组成 Java 等高级语言虚拟机[6-8]的执行引擎。应用程序由解释器开始执行,JIT 编译器仅编译程序中频繁执行的部分。因此,解释执行的速度直接影响系统的启动性能。对于客户端浏览器中的 JavaScript 引擎,解释器的性能在很大程度上决定了系统的响应速度,直接影响用户的使用体验。与此同时,在工业机器人和高档数控机床等领域,控制指令和加工程序的执行也普遍采用解释器来实现。解释器的性能决定了智能化控制的实时性和自动化生产的效率。随着人类社会信息化的不断发展,由解释器运行速度慢而导致的性能和效率等问题将变得越来越普遍和突出。因此,进行解释器的优化以改善其性能,对提高生产效率具有十分重要的意义。

解释执行的基本单位以字节码最为常见。每个字节码都对应一段特定的本地指令序列。字节码的解释执行最终由其所对应的本地指令序列在硬件上的执行来实现。解释执行分为取字节码、指令分派、取操作数和执行等四个步骤。其中,指令分派是指当前字节码执行完毕后,查找并跳转到下一个字节码的本地指令序列入口的过程。指令分派的频繁发

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, 2014ZX01030101),国家自然科学基金(61221062, 61133004, 61173001, 6123009, 61222204, 61432016)和 863 计划(2012AA010901, 2013AA014301)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:2015-11-16)

生会引入大量额外的运行时开销,严重影响解释执行的性能。因此,对指令分派进行优化是提升解释器性能的有效途径。本文研究了解释器指令分派的优化技术,提出了一种采用软硬件协同设计的指令分派方法。实验结果表明,本文方法大幅降低了指令分派时查找本地指令序列入口地址的开销,显著提升了解释器的性能。

1　相关工作

早期的解释器大都采用 Switch 结构来实现[9]。20 世纪 70 年代,Bell[1]研究发现 Switch 结构的解释器在指令分派时存在二次跳转和边界检查等冗余操作,严重影响解释器的性能,从而提出了著名的线索化代码(threaded code)结构的解释器[1],以加速指令分派的过程。线索化代码结构的解释器通过在本地指令序列的末尾独立实现指令分派,消除了 Switch 结构解释器中的冗余操作,从而获得了更高的性能。此后,线索化代码结构成为高性能解释器设计的主流。

然而,线索化代码结构的解释器进行指令分派时依然存在较大开销。为了进一步提升指令分派的速度,文献[2]提出了超级指令(superinstruction)技术。该技术将频繁执行的连续字节码序列等效替换为一条新定义的字节码(称为"超级指令"),以消除被替换的字节码之间的指令分派。随后,Casey 等人[3]的研究表明,由于指令分派中用于实现跳转功能的间接转移指令存在多个跳转目标地址,导致处理器对间接转移猜测的正确率下降,从而降低解释器的性能。针对上述问题,他们提出了指令冗余(instruction redundancy)技术,通过增加若干语义功能相同的冗余的字节码,以减少间接转移指令可能的跳转目标数目,从而提升分支预测的准确性。但是,由于受字节码中操作码域编码长度的限制,实际能够提供的超级指令或冗余字节码的数量十分有限,因而其优化空间较小。

文献[10]设计了一种新型的分支预测器。该方法利用字节码指针来区分不同的转移场景,并在解释器中插入引导指令来提升处理器对间接转移猜测的正确率。然而该方法要求对处理器内部的分支预测器进行定制,硬件实现开销较大。此外,最新的研究结果表明[11],随着分支预测技术的不断发展,在当今主流的处理器上,间接转移猜测已不再是制约解释器性能最主要的因素,因而其优化空间非常有限。

针对上述研究的不足,本文以降低获取本地指令序列入口地址的开销为突破口,对解释器指令分派进行了优化。本文首先对解释器的性能瓶颈进行分析,并在此基础上,提出了一种软硬件协同设计的解释器指令分派方法。最后通过对比实验,分析和验证了本文方法的有效性。

2　解释器性能瓶颈分析

本节以 HotSpot 虚拟机[7]的解释器为代表,介绍了现代高性能解释器设计的关键技术。随后,又以 MIPS 架构的处理器为例,深入分析了解释器的性能瓶颈。

2.1　解释器设计的关键技术

HotSpot 的解释器基于线索化代码结构进行设计,模拟了一个基于栈式体系结构的抽象处理器。它采用了栈顶缓存(stack caching)的优化技术[12],将操作数栈顶的一个元素缓存到一个通用寄存器中,从而减少取操作数时冗余的栈顶访存操作。

栈顶缓存技术使得字节码的本地指令序列存在多个不同的入口。在指令分派阶段,解释器根据下一个字节码和当前的栈顶缓存状态查询指令分派表,以获取本地指令序列的入口地址。表1展示了

表 1　解释器栈顶缓存状态

缓存状态	缓存对象
btos	一个 byte 类型数据
ctos	一个 char 类型数据
stos	一个 short 类型数据
itos	一个 int 类型数据
ltos	一个 long 类型数据
ftos	一个 float 类型数据
dtos	一个 double 类型数据
atos	一个引用类型数据
vtos	无缓存

所有的栈顶缓存状态及其含义。从表 1 中可以看出,由于存在 9 种不同的栈顶缓存状态,故字节码本地指令序列的入口最多可以有 9 个。图 1 展示了指令分派表的基本结构,其中每列存储一个字节码的本地指令序列所有可能的入口地址。例如,图 1 中阴影部分的表项存储编码为96的字节码(iadd,表示整数加法)在栈顶缓存状态为 itos 时的本地指令序列的入口地址。由于指令分派表在解释执行期间被频繁访问,因此合理地设计指令分派表的结构以加快指令分派时的查表操作,是提升解释器性能的关键。

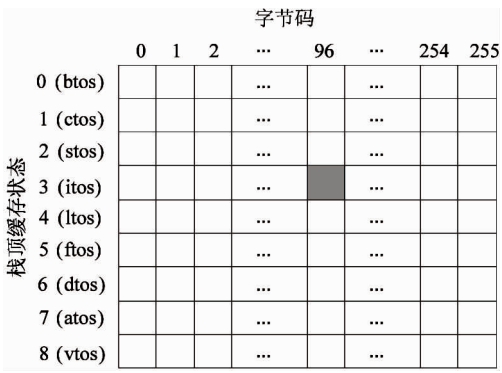

图 1　解释器指令分派表

2.2　指令分派瓶颈分析

解释执行期间,指令分派的开销对解释器的性能有着很大的影响。由于直接统计指令分派的执行时间较为困难且存在较大误差,本文使用指令的数量来间接反映指令分派的开销。图 2 统计了指令分派部分的指令条数占本地指令序列指令数目的比例。统计结果表明,在绝大多数字节码的本地指令序列中,用于指令分派的本地指令数目占本地指令总数的 25% 以上,几何均值达到21.4%,最高可达 68.8%。因此,解释器运行时,指令分派约占总执行开销的 20%,甚至更高。

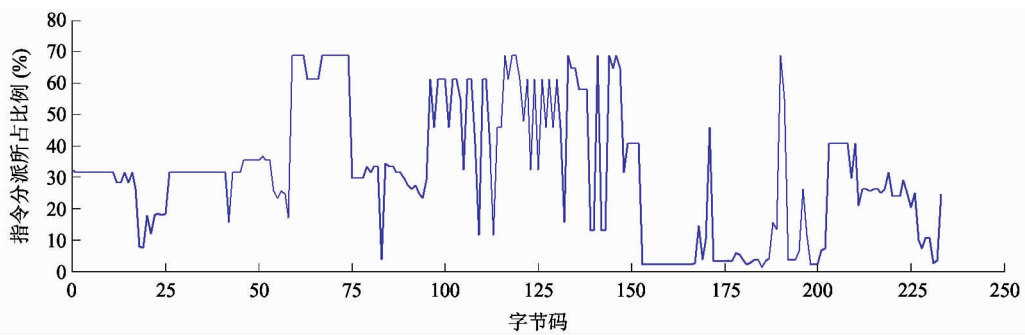

图 2　指令分派在本地指令序列中所占比例

图 3 展示了在 MIPS 架构处理器的 64 位系统中,字节码 iadd 的本地指令序列。为了更清晰地展示本地指令序列的结构和语义,图中省略了对分支指令延迟槽的优化。由图 3 可知,iadd 的本地指令序列共有 18 条机器指令,其中用于指令分派的机器指令多达 11 条,占本地指令序列的61.1%。由于指令分派部分的指令总是全部执行,并且指令分派频繁发生,故指令分派在解释执行期间产生的额外开销很大,成为解释器性能的瓶颈。

指令分派时,解释器首先根据下一个字节码和当前的栈顶缓存状态查询指令分派表,以获取本地指令序列的入口地址(记为 *Entry*),随后跳转到 *Entry* 处继续执行。设下一个待执行的字节码编码为 $x(x \in [0, 255])$,当前栈顶的缓存状态为 $s(s \in [0, 8])$,则指令分派可进一步细分为以下几个步骤:

(1) 加载待查询的指令分派表的行首地址到通用寄存器 *Reg* 中(即图 3 中第 8 ~ 13 条指令);

(2) 计算存储 *Entry* 的表项在待查询的行内的偏移 *offset*(即图 3 中第 14 条指令);

(3) 从指令分派表中地址为 *Reg* + *offset* 的位置加载 *Entry* 的值(即图 3 中第 15 和 16 条指令);

(4) 跳转到 *Entry* 继续执行(即图 3 中第 17 条指令)。

上述 4 个步骤中,前 3 步为获取 *Entry* 的操作,最后一步执行跳转动作。由于获取*Entry*需要9条

```
#   # 取操作数
1   lw      v0, 0x0(sp)   # vtos入口
2   daddiu sp, sp, 0x8
3   lw      v1, 0x0(sp)   # itos 入口
4   daddiu sp, sp, 0x8

#   # 执行规定的语义动作
5   addu    v0, v1, v0    # 相加求和

#   # 取下一个字节码
6   lbu     s1, 0x1(s0)   # 加载下一个字节码
7   daddiu s0, s0, 0x1    # 字节码指针s0前移

#   # 指令分派
8   lui     t3, 0x0
9   ori     t3, t3, 0x55                加载指令分派表
10  dsll    t3, t3, 16                  待查询的行的首
11  ori     t3, t3, 0x56cf              地址到寄存器t3
12  dsll    t3, t3, 16
13  ori     t3, t3, 0xffffc070
14  dsll    t2, s1, 3     # 计算offset于t2
15  daddu  t3, t3, t2     # 计算Entry的地址
16  ld      t3, 0x0(t3)   # 加载Entry的值到t3
17  jr      t3            # 跳转到Entry
18  nop                   # 延迟槽
```

图 3　字节码 iadd 的本地指令序列

机器指令(即图 3 中第 8 ~ 16 条),占了指令分派的大部分执行时间,故获取 *Entry* 的操作构成了指令分派的瓶颈。进一步分析可知,第(1)步中加载待查询的指令分派表的行首地址耗费了 6 条机器指令(即图 3 中第 8 ~ 13 条),是导致获取 *Entry* 值代价过高的主要原因。

3　软硬件协同的指令分派设计

本文针对指令分派时获取本地指令序列的入口地址代价过高的问题,设计并实现了一种软硬件协同的指令分派方法,主要包括软件优化的指令分派表访问技术和基于硬件支持的访存加速策略。

3.1　指令分派表的优化设计

指令分派期间,为了查找本地指令序列的入口地址,需加载待查询的指令分派表的行首地址到寄存器中。由图 3 可知,完成该操作需要 6 条机器指令,实现代价较高。本文设计了一种便于高效访问的指令分派表,其结构如图 4 所示,完全消除了加载操作的开销。本文从以下四个方面对指令分派表进行了优化。

(1)对指令分派表的行列进行了转置,为后文基于硬件支持的指令分派表的访问加速奠定了基

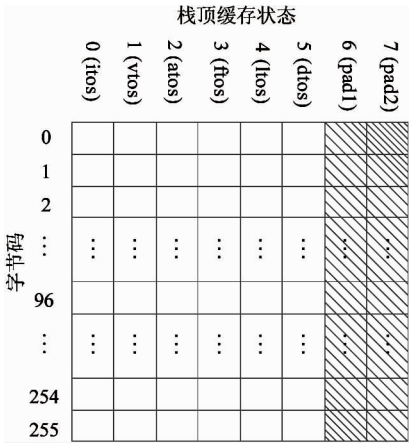

图 4　优化后的指令分派表结构

础。转置前的指令分派表每行有 256 个表项。当加载行中的某个表项时,行内偏移的取值范围较大。对于 64 位的系统,转置前行内偏移的范围是$[0, 2^{11}-8] \subseteq [-2^{11}, 2^{11}-1]$,故要求访存指令中偏移量编码域的长度不少于 12 位。但是由于精简指令集架构的处理器通常采用固定长度的指令编码格式,访存指令中所能提供的编码偏移量的长度非常有限。本文对原有指令分派表进行了转置,可以减少机器指令中偏移量编码所需的长度,使得通过硬件支持的方式加速指令分派表的访问成为可能。

(2)剔除了三类无用表项,减少了解释执行时对系统资源的消耗。指令分派时,需根据当前栈顶的缓存状态确定下一个字节码的本地指令序列的入口地址。由于 HotSpot 解释器将缓存状态为 btos、ctos 和 stos 的入口地址均映射到 itos 的入口地址,因此图 4 中的指令分派表结构在设计时,消除了 btos、ctos 和 stos 相关的表项,以进一步减少解释执行对内存等资源的消耗。

(3)添加了两列填充表项,以减少查表的计算开销。指令分派表在添加填充表项前,需执行一次乘法操作来计算待查表项的行偏移。由于乘法运算的代价较高,在设计指令分派表时,每行的末尾均添加了两个填充表项,如图 4 中阴影部分所示,从而可通过一次快速的左移位操作计算出行偏移。因此,填充表项的引入可将代价较高的乘法运算转换为高效的移位操作,查表时计算代价明显降低。

(4)合理设计指令分派表每列的顺序,提高访存效率。图 5 展示了访问指令分派表时栈顶缓存状

态的频率分布。从图中可知指令分派表的访问集中在对 vtos、itos、atos 和 ftos 缓存状态的查询。因此在设计指令分派表时，将 vtos、itos、atos 和 ftos 这 4 个表项安排在相邻的位置，以增强访存的局部性，提高访存效率。

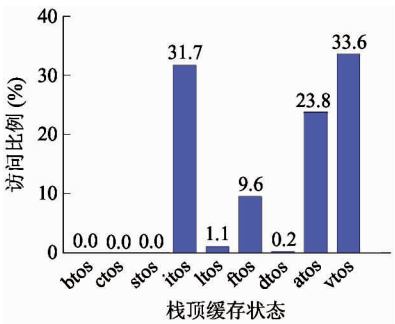

图 5　指令分派表访问时栈顶缓存状态的频率分布

3.2　软件优化的指令分派表访问技术

根据 3.1 节中指令分派表结构的设计，指令分派表的访问技术也做了相应的软件优化。其核心思想是将指令分派表的起始地址缓存到一个可以高速访问的部件中，以消除优化前代价较高的行首地址加载操作。

图 6 具体展示了基于软件优化的指令分派表查找本地指令序列入口地址的方法。本文选择了一个通用寄存器作为指令分派表的高速缓存部件，该寄存器称为指令分派寄存器（记为 *gpr*）。解释器初始化时，将指令分派表的起始地址（即图 6 中标“0”的箭头所指向的位置）保存到指令分派寄存器中。设下一个待执行的字节码编码为 $x(x \in [0, 255])$，

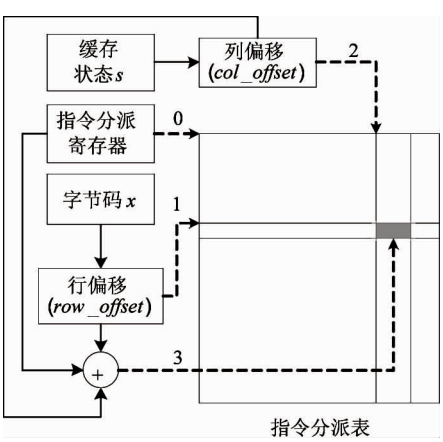

图 6　优化的指令分派表访问方法

当前栈顶的缓存状态为 $s(s \in [0,5])$，待查找的本地指令序列的入口地址为 *Entry*，则获取 *Entry* 的步骤如下：

（1）根据字节码 x 的编码，计算待访问表项的行偏移 *row _ offset*（即图 6 中标“1”的箭头所指向的位置）：

$$row\_offset = 8 \times AddressLength \times x \quad (1)$$

其中 *AddressLength* 表示系统中一个指针所占的字节数目，为一个固定的常量。*row _ offset* 保存在一个通用寄存器中。它的值将在解释执行的过程中实时计算。具体实现时，上述乘法操作可以转换为对 x 的快速左移位操作，需要左移的位数为 $\log_2(8 \times AddressLength)$。

（2）根据当前的栈顶状态 $s(s \in [0, 5])$，计算待访问表项的列偏移 *col _ offset*（即图 6 中标“2”的箭头所指向的位置）：

$$col\_offset = AddressLength \times s \quad (2)$$

由于 $AddressLength \leqslant 8$，故 $col\_offset \in [0, 40]$，为一个较小的立即数。其值在解释器初始化时确定。

（3）从指令分派表中地址为 *gpr* + *row _ offset* + *col _ offset* 的表项（即图 6 中标“3”的箭头所指向的表项）中加载 *Entry* 的值。

图 7 展示了采用图 6 中的方法实现的指令分派方案。行偏移 *row _ offset* 的值在第 1 条左移位指令 dsll 中完成计算，结果存于寄存器 t2 中。在第 2 条加法指令 daddu 中，指令分派寄存器 *gpr* 与 t2 相加得到待查询的指令分派表的行首地址，并将其保存到寄存器 t3 中。随后，第 3 条访存指令 ld 实现从地址为 t3 + *col _ offset* 的指令分派表中加载 *Entry* 的值。最后，通过第4条间接转移指令jr跳转到入口为 *Entry* 的本地指令序列中继续执行，从而完成一次指令分派。

```
#  # 指令分派
1  dsll   t2, s1, 6     # 计算row_offset于t2
2  daddu  t3, gpr, t2   # 计算Entry所在行的首地址
3  ld     t3, col_offset(t3)   # 加载Entry的值到t3
4  jr     t3            # 跳转到Entry
5  nop                  # 延迟槽
```

图 7　纯软件优化的指令分派

对比图3可以看出,优化前用于加载指令分派表待查询的行首地址的6条机器指令已全部消除,指令分派部分的机器指令数目由原来的11条减少为5条。通过对SPECjvm98和DaCapo等测试集的分析知,解释执行期间指令分派的次数可高达数百亿次。因此优化后,可以减少千亿条数量级机器指令的执行,从而大幅度提高指令分派的效率。

3.3　基于硬件支持的访存加速策略

由3.2节中的分析可知,为了获取*Entry*的值,需要访问地址为*gpr* + *row _ offset* + *col _ offset*的内存单元。其中*gpr*为基址寄存器,保存指令分派表的起始地址;*row _ offset*为索引寄存器,保存待查询的行的索引;*col _ offset*为一个较小的立即数,表示目标单元在行内的偏移。由于MIPS等精简指令集的处理器仅支持"基址+偏移"的寻址方式,故上述操作需要两条机器指令(图7中第2、3条)才能完成。

为了进一步加快指令分派的速度,本文对处理器的访存指令进行了扩展,新引入了一种支持"基址+索引+偏移"的寻址方式的访存指令,使得对*Entry*值的加载操作仅由一条指令即可完成,从硬件层面直接加速指令分派时对内存地址*gpr* + *row _ offset* + *col _ offset*的访问速度。其硬件实现原理如图8所示。扩展指令的实现主要涉及对处理器流水线中指令译码器、寄存器重命名单元和访存部件的扩展。由于扩展的访存指令使用了新定义的指令操作码,指令译码器需增加对扩展指令的译码功能。同时,由于扩展指令中编码了基址寄存器、索引寄存器和结果寄存器,故在流水线的寄存器重命名阶段需要对三个寄存器进行重命名,从而将指令中编码的三个寄存器分别映射到物理寄存器堆中相应的物理寄存器上。此外,由于访存的有效地址计算需要对"基址+索引+偏移"进行求和运算,故访存部件中原有的双操作数地址运算单元也需要进行简单的扩展,以支持3操作数的求和运算。最后,根据有效地址完成相应的访存操作,并将访存结果写回结果寄存器中。扩展指令的硬件逻辑简单,实现代价低,很容易在各种处理器中实现。

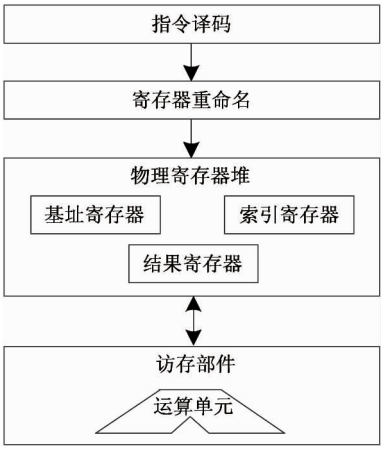

图8　扩展访存指令的硬件实现原理

在MIPS架构中,指令编码长度固定为32位。原有的访存指令(如ld指令等)最大仅支持16位的偏移量。而扩展的访存指令中需要额外使用5位用于对新增的索引寄存器进行编码,故无论扩展的访存指令如何设计,其所支持的偏移量最大不能超过16-5 = 11位。由3.1节中的分析可知,优化前的指令分派表要求访存指令中的偏移量不少于12位,因此,本文对原有指令分派表的行和列进行了转置。转置后,即使对于64位的系统,指令分派表行内偏移的取值范围是$[0, 40] \subseteq [-2^6, 2^6 - 1]$,故偏移量的编码只需要7位,满足了硬件扩展的约束。

基于上述分析,我们总共实现了12条扩展的访存指令,包含8条定点指令和4条浮点指令,寻址方式均为"基址+索引+偏移"的访存模式。上述扩展指令分别为存取字节(gssbx/gslbx)、存取半字(gsshx/gslhx)、存取字(gsswx/gslwx)、存取双字(gssdx/gsldx)、存取单精度浮点数(gsswxc1/gslwxc1)和存取双精度浮点数(gssdxc1/gsldxc1)。这些指令均支持长度为8位的访存偏移量,完全满足优化后指令分派表访问时对7位偏移的需求。

图9展示了添加了硬件支持后,指令分派的实现方案。图中使用了一条扩展访存指令gsldx,用于从内存中加载一个64位的双字。对比图7可以看出,加载*Entry*的操作由图7中的两条机器指令减少为图9中的一条gsldx指令即可完成。由于gsldx与原有64位访存加载指令ld所需的执行周期相同,故扩展指令的引入可以消除一条加法指令dad-

du,从而进一步提高指令分派的效率。

```
#  # 指令分派
1  dsll   t2, s1, 6                  # 计算row_offset
2  gsldx  t3, gpr, t2, col_offset    # 加载Entry的值
3  jr     t3                         # 跳转到Entry
4  nop                               # 延迟槽
```

图 9　软硬件协同的指令分派

4　实验与结果分析

4.1　实验平台

为了验证本文方法的有效性,本研究在 OpenJDK8[13] HotSpot 虚拟机的解释器上,采用业界广泛使用的 SPECjvm98 和 DaCapo-9. 12-bach 测试集进行实验。实验中,SPECjvm98 和 DaCapo 的输入规模分别设为最大值 100 和默认大小。由于 DaCapo 中 *batik* 和 *eclipse* 引用了与 OpenJDK8 不兼容的低版本类库,因此本文将这两个测试项排除在外。

实验中使用的硬件平台为 MIPS 兼容的龙芯 3 号处理器,主频 900MHz,内存 4G,操作系统为 64 位的 CentOS 6.4。本研究进行了以下 3 组对比试验:

(1)传统的实现方案,即图 3 中的方法;

(2)纯软件的优化方案,即图 7 中的方法;

(3)软硬件协同的优化方案,即图 9 中的方法。

为了减小实验误差,每个测试程序均运行 10 次,取其均值作为最终的实验结果,并以传统方案的结果为基准对实验结果进行了归一化处理。

4.2　实验结果与分析

图 10 展示了解释器在三种方案下的总体性能。从图中可以看出,相对于传统方案,采用纯软件的优化方案,解释器总体性能提升幅度已高达8.3%。而增加硬件支持后,解释器总体性能进一步提升约 3.2%,最终达到11.5%的总体性能提升。

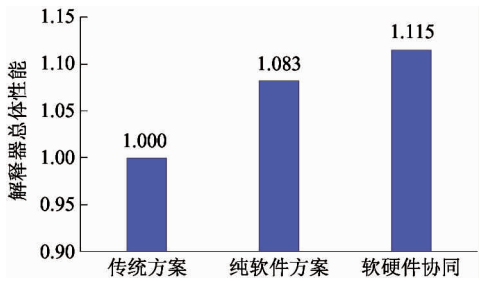

图 10　三种方案下的解释器总体性能

图 11 展示了解释器在三种方案下,SPECjvm98 和 DaCapo 测试集的性能。图中的数据表明,软硬件协同的优化使得绝大多数的测试项目获得 10%以上的性能提升,最高性能提升幅度高达15.4%。此外,从图 11 中还可以看出,使用硬件支持,解释器的性能可在纯软件优化的基础上进一步提升约 2% ~ 5%的性能,故硬件支持是纯软件优化的重要补充。因此,软硬件协同设计的指令分派对解释器性能的提升具有非常好的效果。

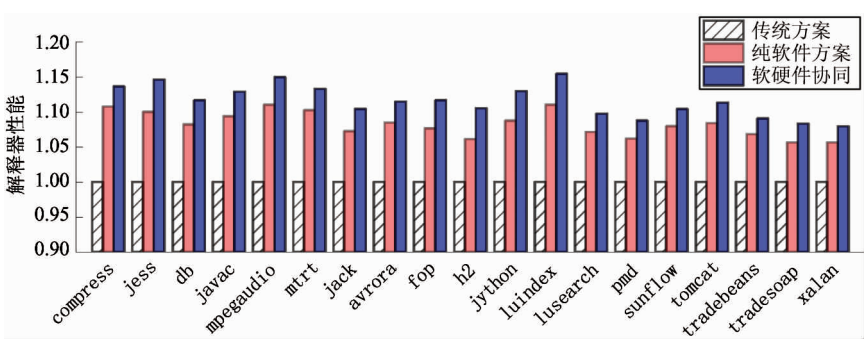

图 11　解释器在 SPECjvm98 和 DaCapo 测试集上的性能

为了进一步揭示解释器性能提升的原因,本文从静态和动态两个方面进行分析。从静态角度看,字节码本地指令序列中,指令分派部分机器指令数目的减少是解释器性能提升的根本原因。图 12 展示了软硬件协同优化前后,指令分派在字节码的本地指令序列中所占的比例。图中的结果表明,优化后指令分派的比例显著下降。对于绝大多数字节码,该比例下降幅度超过了 50%,所有字节码指令

分派部分所占比例的几何均值由原来的21.4%下降为9.9%,下降幅度高达53.7%。此外,指令分派部分机器指令数目的减少,使得所有字节码的本地指令序列中机器指令的总数减少了6.7%,解释执行所需的存储空间进一步降低。指令分派中机器指令数目的大幅减少,从理论上证明了软硬件协同设计对解释器指令分派优化的良好效果,并且预示着动态执行的机器指令数目的大幅减少。

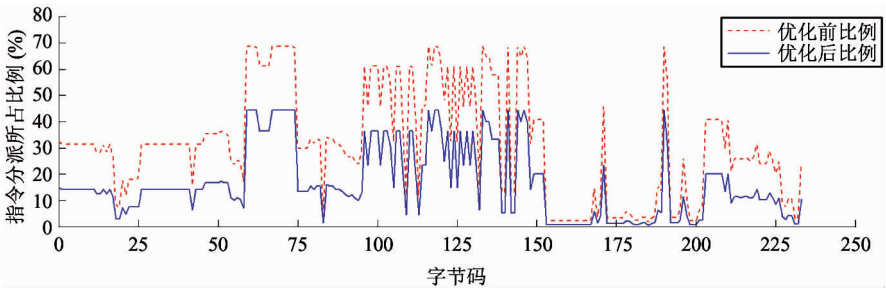

图 12　软硬件协同优化前后指令分派在字节码的本地指令序列中所占比例

从动态角度看,运行时实际执行的机器指令数目的减少是解释器性能提升的直接原因。表2展示了软硬件协同优化后,在 SPECjvm98 和 DaCapo 测试集上,解释器运行时减少执行的机器指令数目。由表 2 的数据知,软硬件协同优化后,解释器减少执行的机器指令数目高达千亿条的数量级,平均下降幅度为优化前的16.67%,最高优化幅度高达22.87%。实验结果与理论预期相符。由于软硬件协同的优化可以大幅度减少指令分派所执行的机器指令数目,故能充分地优化解释器的性能。

表 2　软硬件协同优化后减少执行的机器指令

测试项目	优化前执行的指令数目(千亿条)	优化后减少的指令数目(千亿条)	减少的比例(%)
compress	4.11	0.85	20.68
jess	0.71	0.12	17.03
db	1.46	0.23	15.79
javac	0.80	0.13	16.31
mpegaudio	3.58	0.82	22.87
mtrt	0.91	0.14	14.95
jack	0.53	0.07	13.11
avrora	3.06	0.52	16.91
fop	0.32	0.05	16.30
h2	8.25	1.28	15.48
jython	4.27	0.67	15.77
luindex	0.85	0.17	20.72
lusearch	3.16	0.52	16.42
pmd	0.95	0.15	15.06
sunflow	17.55	3.06	17.42
tomcat	1.35	0.24	17.90
tradebeans	9.57	1.51	15.73
tradesoap	9.42	1.53	16.25
xalan	3.29	0.52	15.77
几何均值	2.22	0.37	16.67

5　结　论

解释器指令分派一直是影响解释器性能最关键的因素。本文对解释器指令分派的开销进行了分析,指出对字节码本地指令序列入口地址的获取构成了指令分派的瓶颈,并且有针对性地提出了一种采用软硬件协同设计的指令分派优化方案。该方案具体包括优化的指令分派表设计、软件优化的指令分派表访问技术和基于硬件支持的访存加速策略。试验结果表明,本文的方案大幅降低了指令分派的开销,有效提升了解释器的性能。本文的方法通用性强,实现代价低,对于现代高性能解释器的设计和优化具有很好的借鉴意义。未来的工作将从软硬件协同设计的角度出发,对高级语言虚拟机的动态编译系统和数组越界检查等运行时机制做进一步的分析和优化。

参考文献

[1] Bell J R. Threaded code. *Communications of the ACM*, 1973,16(6):370-372

[2] Casey K, Gregg D, Ertl M A, et al. Towards superinstructions for java interpreters. In: Proceedings of the 7th International Workshop on Software and Compilers for Embedded Systems, Vienna, Austria, 2003. 329-343

[3] Casey K, Ertl M A, Gregg D. Optimizingindirect branch prediction accuracy in virtual machine interpreters. *ACM Transactions on Programming Languages and Systems*, 2007, 29(6):1-36

[4] Brunthaler S. Virtual-machine abstraction and optimization techniques. *Electronic Notes in Theoretical Computer Science*, 2009, 253(5):3-14

[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] 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] 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

[8] 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

[9] Mccandless J, Gregg D. Compiler techniques to improve dynamic branch prediction for indirect jump and call instructions. *ACM Transactions on Architecture and Code Optimization*, 2012, 8(1):1-24

[10] 黄明凯, 刘先华, 谭明星等. 一种面向解释器的间接转移预测技术. 计算机研究与发展, 2015, 52(1):66-82

[11] Rohou E, Swamy B N, Seznec A. Branch prediction and the performance of interpreters: Don't trust folklore. In: Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization, Washington, DC, USA, 2015. 103-114

[12] Ertl M A. Stack caching for interpreters. In: Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation, New York, USA, 1995. 315-327

[13] Oracle. OpenJDK8. http://openjdk.java.net/projects/jdk8/: Oracle Corporation, 2014

An instruction dispatch approach using hardware-software co-design for interpreters

Fu Jie[*,**,***], Jin Guojie[****], Zhang Longbing[**,***], Wang Jian[**,***]

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

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

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

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

Abstract

To reduce the overhead caused by instruction dispatch to improve the performance of interpreters, an instruction dispatch approach based on hardware and software co-design is proposed. Its main idea is to eliminate the expensive operation of constant address loading by optimizing the instruction dispatch table in the aspect of sofware, and to acceleratethe speed of memory access under the support of hardware by enhancing the processor's instruction set in the aspect of hardware. The hardware-software co-design can minimize the runtime overhead of instruction dispatch, thus improving the performance of interpreters. The experimental results showed that the proposed approach significantly improved the performance of interpreters. For benchmarks of SPECjvm98 and DaCapo, the overall performance of interpreters was improved by 11.5%, and the highest performance boost was up to 15.4%. The approach is highly versatile, easy to implement and can be applied to the design and implementation of high performance interpreters on mainstream processors.

Key words: interpreter, instruction dispatch, hardware and software co-design, virtual machine, optimization