Java 虚拟机中动态内联策略的改进

周 晶 王 雷 刘志成

(北京航空航天大学 计算机学院, 北京 100083)

摘 要: 在 Java 虚拟机中,传统的基于计数的热点探测方法往往不能得到方法间的调用关系,因此,采用这种机制的动态编译器会丧失一些优化机会,比如方法内联. 提出了基于在线反馈信息的动态内联策略. 同已有的内联算法相比,该方法引入了 BC _ Map,它由基线编译器对方法进行第一次编译时构建. BC _ Map 可以确定对方法 M 进行调用的字节码在其调用者中的索引. 通过构建 Guarded _ Rec 记录,编译器能够准确判断出虚方法的接收者对象的类型,进而确定方法间的调用关系. 使用该信息指导的保护内联优化可以避免静态判断虚方法的接收者对象类型的缺陷,得到更加准确的动态信息,使应用程序的性能得到提升.

关 键 词: Java 程序设计语言; 编译程序; 优化

中图分类号: TP 301

文献标识码: A **文 章 编 号**: 1001-5965(2006)03-0352-05

Dynamic inlining scheme improvement in JVM

Zhou Jing Wang Lei Liu Zhicheng

(School of Computer Science and Technology, Beijing University of Aeronautics and Astronautics, Beijing 100083, China)

Abstract: In the Java virtual machine(JVM), traditional counter-based hotspot-detecting method usually can not get the caller-callee relationship, therefore, the dynamic compiler which applied this scheme would lose some optimization opportunities, such as method inlining. A dynamic inlining scheme based on online feedback information was proposed. Compared with the algorithms presented, the new method introduced BC _ Map, it was constructed by the baseline compiler when a method was compiled for the first time. BC _ Map can identify the index of the bytecode that invoked method M in its caller. With constructing Guarded _ Rec record, the compiler can get the type of virtual method's receiver object correctly, and then identify the relationship between caller and callee. The guarded inlining guided by this information can avoid identifying the type of virtual method's receiver object statically, get the dynamic information more precisely, and then it can increase applications' performance.

Key words: Java programming language; program compilers; optimization

Java 语言的易移植性,安全性,动态性[1]等各种特性使得 Java 的应用越来越广,但同时也给高性能的 Java 虚拟机的设计和实现带来了一定的挑战. 动态编译技术是虚拟机中的核心技术,它极大地提高了虚拟机的性能,而整个应用程序的性能,在很大程度上取决于动态编译技术的优劣.

方法内联(method inlining)是一种被广泛使用的编译优化方法,该方法的主要思想是用被调用方法的方法体来替代对该方法的调用,从而降低方法调用时的开销[2]. 除了这些好处,方法内联还可以带来一些新的优化机会[3],从而使生成代码的质量得到进一步地提高. 方法内联的缺点是它会使代码的体积产生膨胀,因此在对方法做内联优化时一定要避免代码膨胀过度.

在 Java 语言中,如果方法没有被声明为 static,final 以及 special,那么对这些方法的调用都将

收稿日期: 2005-03-25

基金项目: 英特尔中国研究中心(ICRC)基金资助项目

作者简介: 周 晶(1981-),男,北京人,硕士生, anderszhj@163.com.

采用虚方法调用机制,即这些方法的接收者对象的类型要在运行时才能确定. 因此,在对虚方法做内联优化时要使用保护内联(guarded inlining)策略. 保护内联通常有方法比较(method test)和类比较(class test)2 种方法[4].

本文在总结了目前国际上已有的几种内联算法的基础上,指出某类算法的缺陷,并针对此种缺陷进行改进,提出了基于在线反馈信息的动态内联算法.

1　内联优化算法及其缺陷

虚方法的内联优化分为 2 步:①检测虚方法的执行频度,判断哪些方法经常被执行,这一技术叫做热点探测(hotspot detection)[5]. 多数情况下,只考虑对热点方法进行内联优化. ②判断这些热点方法的接收者对象的类型,然后使用保护内联策略对其进行优化.

1.1　热点方法的探测机制

目前国际上广泛采用的热点方法探测机制主要有 3 种:

1) 基于采样的(sample-based)探测方法,即周期性的扫描堆栈栈顶,如果在若干次扫描中发现栈顶总是同一个方法,则认为该方法是热点方法. 这种方法的优点是通过对运行时堆栈的多层展开可以确定方法间的调用关系. IBM T. J. Watson 研究中心开发的 JikesRVM[6] 就是采用这种策略来对热点方法进行探测的.

2) 基于计数的(counter-based)探测方法,即在方法的入口点和循环返回边处插入计数器以确定方法和循环的执行次数. 使用这种技术的系统主要有 Self-93[7], HotSpot JVM[5] 和 ORP(Open Runtime Platform)[8].

3) 基于采样与基于计数相结合的探测方法,采用这种策略的系统主要有 IBM DK for Java[9].

1.2　已有的热点探测机制的缺陷

在对虚方法进行内联优化时,上述第 2)种策略可以精确地判断出方法的执行次数,但是该策略的缺点是无法确定方法间的调用关系. 由于这一缺陷,很多采用此种策略的动态编译器在确定方法的接收者对象的类型时,通常比较保守,因此并不能准确地判断出接收者对象的类型,在确定真正的热点方法以及进行保护内联操作时,都需要该类型信息,所以这种策略会丧失一些优化机会.

1.3　实验平台

研究工作是在英特尔微处理器实验室研发的一款 Java 虚拟机——ORP 上进行的. ORP 是一个源代码公开的平台,它有 2 个即时编译器(Just in Time Compiler)O1 和 O3,没有解释器,其中 O1 是基线编译器,用来快速地生成只经过简单优化的可执行代码,O3 是优化编译器,用来对热点方法进行优化编译. 它采用的热点探测机制是基于计数的,因此存在前面所讲的丧失优化机会的问题,例如下面这个实例.

实例 1　有 A,B,C 3 个类,A 类是 B 类和 C 类的父类,A 类中的方法 foo( )被 B 类和 C 类覆盖(override),对于下面的语句:

```
A x=(A)A.factory(i);
x.foo();
public static A factory(int i){
    if(i<40 000) return new C();
    else return new B();
}
```

在确定热点方法时,ORP 将检查 A.foo( )的 profile 记录,以确定 A.foo( )是否为热点方法,此情况下显然不是,因此编译器将不考虑对 x.foo( )进行内联优化. 由于未能准确地判断出方法的接收者对象的类型,从而使编译器没有准确地确定热点方法,进而丧失了对真正热点方法的一系列优化机会.

对于 Java 中的虚方法调用,同一个调用点可能调用具有继承关系的几个类的同名方法,如本节中的 x.foo( )就可能调用 A 或 B 或 C 的 foo( )方法,因此在做方法内联时要区分对待,有所保护,即保护内联.

保护内联的主要思想是,首先判断运行时虚方法的接收者对象是否为某种特定类型,如果是则用该类型的虚方法体替代对该方法的调用,否则按照缺省的方式对方法进行调用. 在对特定类型进行判断时,ORP 采用类比较方法,因此这种判断就转化为对接收者对象的虚方法表的预测上,本文称这一虚方法表是目标虚方法表. 由于这种预测也是根据接收者对象的静态类型进行的,所以也存在丧失优化机会的可能.

2　基于反馈信息的动态内联算法

前面介绍了基于计数的热点探测机制所存在的缺陷,以及该缺陷对保护内联产生的影响. 本小结将介绍基于在线反馈信息的动态内联算法,该算法目前已在 ORP 上实现.

2.1　映射关系 **BC _ Map** 的结构

由于一个方法对同一个虚方法的调用可能不

止一处,而每一处的调用情况也可能各不相同,所以在确定方法间的调用关系时,编译器需要知道方法调用点的精确位置. 为了满足这一条件,本文的算法用字节码索引 bc _ index 来标记这一位置,并定义了一组映射关系 BC _ Map.

BC _ Map 根据方法 M 的返回地址和其调用者(caller)得到对方法 M 进行调用的字节码在其调用者中的索引,BC _ Map 中每一项的结构如图 1 所示.

BC _ Map 中的每一项由以下 2 部分组成:

1) 每一项有且仅有一个 Method 域,该域表示这一项所对应的方法.

2) 每一项包含一到多个三元组<*I*,*B*,*E*>,其中:

① *I* 表示字节码索引(bc _ index),也就是每条字节码相对于方法开始位置的偏移量.

② *B* 表示每条字节码所对应的二进制代码序列的开始地址(begin _ addr).

③ *E* 表示每条字节码所对应的二进制代码序列的结束地址(end _ addr).

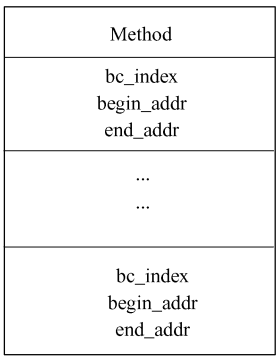


图 1 BC _ Map 中的一项

2.2 构建映射关系 BC _ Map

由于只有虚方法才存在动态绑定的问题,因此本文的算法只考虑虚方法的情况,所以在构建 BC _ Map 时,只对字节码指令 invokevirtual 和 invokeinterface 构建三元组<*I*,*B*,*E*>. 事实证明,采用这种机制构建的 BC _ Map 不仅压缩了 BC _ Map 的大小,同时还提高了对 BC _ Map 的检索效率.

BC _ Map 是由基线编译器对方法进行第 1 次编译时构建的. 使用 BC _ Map 对方法间调用关系进行确定的过程如下:

```
1) unwindStack();
//展开运行时堆栈
p=getCaller(m);
//得到方法 m 的调用者 p
IP=getReturnAddr(m);
//得到返回地址
2) i=0;
while(BC _ Map is not NULL)
{
  entry[i]=getNextEntry(BC _ Map);
  if (entry[i]. Method==p)
  {
    if((IP>entry[i]. begin _ addr)
    &&(IP<=entry[i]. end _ addr))
    return entry[i]. bc _ index;
  }
  i++;
}
```

其中:

① IP 代表方法 *m* 在其调用者 *p* 中的返回地址;

② entry[*i*]代表 BC _ Map 中的第 *i* 项,每一项的结构如图 1 所示.

算法的核心思想是扫描 BC _ Map 中的每一项,比较每一项的 Method 域与当前方法的调用者 *p* 是否一致. 如果存在一致项,继续查看返回地址 IP 落在该项的哪一个<begin _ addr,end _ addr>区间. 如果存在这样一个区间,那么该区间所对应的 bc _ index 就是对方法 *m* 进行调用的字节码在 *p* 中的索引,即方法 *p* 在字节码 bc _ index 处调用方法 *m*.

上述过程是由被调方法(callee)完成的,而内联优化是调用方法对被调用方法的内联过程,在这个过程中,调用方法需要确定在该方法的某一调用点(call-site)处的被调方法.

为了满足上述条件,算法为每个方法 *m* 定义了一个记录 Guarded _ Rec,该记录由若干对二元组<callee ,bc _ index>组成,该二元组表示方法 *m* 在 bc _ index 处调用虚方法 callee.

有了上述信息,编译器就可以检查正确的 profile 记录并准确预测目标虚方法表.

2.3 改进后的动态内联算法

下面是基于在线反馈信息的动态内联算法.

```
1) for(每一个方法 m)
constructBC _ Map(m);
//为每一个方法 m 构建 BC _ Map
2) if(! isVirtualMethod(m))
recompileAt1000(m);
```

```
else
if(executed900Times(m)){
scanBC _ Map(m);
<p,b>=getCaller _ Bc _ index(m);
addToGuarded _ Rec(<m, b>);
recompile(m);
}
3) if(! isOverridden(h))
inspectProfRec(h);
//检查方法 h 的 profile 记录
else
bc _ index _ h=getBc _ index(h);
for(方法 h 的调用者 caller 的 Guarded _ Rec 中的每一项<callee _ i,bc _ index _ i>){
if(bc _ index _ i==bc _ index _ h){
inspectProfRec(callee _ i);
break;
}
}
4) if(! isOverridden(p))
buildIR _ CFG(p);
//为方法 p 构建中间表示和控制流图
else
bc _ index _ p=getBc _ index(p);
for(方法 p 的调用者 caller 的 Guarded _ Rec 中的每一项<callee _ i, bc _ index _ i>){
if(bc _ index _ i==bc _ index _ p){
buildIR _ CFG(callee _ i);
break;
}
}
5) if(! isOverridden(q))
checkVtable(q);
//预测方法 q 的虚方法表为目标虚方法表
else
bc _ index _ q=getBc _ index(q);
for(方法 q 的调用者 caller 的 Guarded _ Rec 中的每一项<callee _ i,bc _ index _ i>){
if(bc _ index _ i==bc _ index _ q){
checkVtable(callee _ i);
break;
}
}
```

第 2)步的含义是,对于一个方法 m,如果它不是虚方法,那么当它执行 1 000 次时引发对该方法的重编译. 否则,当该方法执行到第 900 次时,检索 BC _ Map 以得到方法 m 的调用者 p 和对方法 m 进行调用的字节码在 p 中的字节码索引 b. 下一步将方法 m 和已得到的 b 分别添加到 p 的 Guarded _ Rec 记录的二元组<callee,bc _ index>中,然后引发对方法 m 的重编译.

方法执行次数 1 000 和 900 均为经验值,是以大量的实验为基础的. 如果一个方法是虚方法,则要保证在该方法的调用者对此方法进行内联优化时,Guarded _ Rec 记录中已有对应的条目,因此要在虚方法执行到 900 次(<1 000 次)时检索 BC _ Map,并向其调用者的 Guarded _ Rec 记录中添加相应的二元组条目<callee ,bc _ index>.

第 3)步检查 profile 记录并确定热点方法. 同已有算法不同的是,该步骤首先判断方法 h 是否被覆盖,如果没有被覆盖,则直接检查方法 h 的 profile 记录. 否则,获得对方法 h 进行调用的字节码在 h 的调用者 caller 中的索引 bc _ index _ h,并与方法 caller 的 Guarded _ Rec 记录中全部二元组<callee _ i,bc _ index _ i>的 bc _ index _ i 比较,如果存在一致的 bc _ index _ i,则检查与之对应的方法 callee _ i 的 profile 记录.

第 4)步用来为被内联的方法创建中间表示(IR)和控制流图(CFG),第 5)步用来预测目标虚方法表. 这两步都包含与第 3)步相同的比较过程,只是比较完成后所采取的具体操作不同. 第 3),4),5)三步中的 bc _ index _ h,bc _ index _ p,bc _ index _ q分别代表对方法 h,p,q 进行调用的字节码在各自的调用者 caller 中的索引.

为了进一步说明算法,下面用 1.3 节中的实例 1 来对算法进行说明. 如果有一个方法 W 调用了 x. foo( ),在这种情况下,该 foo( )方法只有在运行时才能知道究竟是 A,B 还是 C 的相应方法. 由于 x 的静态声明类型为 A,但运行时的实际对象类型为 B 和 C,因此,原有的内联策略将不考虑对 x. foo( )进行内联优化.

采用动态内联策略后,编译器可在方法 W 的 Guarded _ Rec 中保存与 foo( )方法相对应的具体信息,如<B. foo( ),bc _ index _ b>或<C. foo( ),bc _ index _ c>,其中 B. foo( )和 C. foo( )分别是 B 类和 C 类的 foo( )方法,bc _ index _ b 和 bc _ index _ c 分别是对 2 个 foo( )方法进行调用的字节码在 W 中的索引,实例 1 首先执行的是 C. foo( ).

有了这些信息,在检查 profile 记录时,编译器首先判断方法 x. foo( )是否被覆盖,显然被覆盖

了,所以检查方法 W 的 Guarded _ Rec 记录,由于前面已经建立了二元组<C. foo( ),bc _ index _c>,因此可以确定该处的 foo( )方法为 C. foo( ),从而去检查 C. foo( )的 profile 记录. 建立中间表示以及比较虚方法表的过程与此过程类似,同样可以准确得到所需信息. 可见基于在线反馈信息的动态内联策略可以弥补已有内联策略的某些缺陷.

3 实验与性能评测

为了验证动态内联算法的性能,在 ORP 上实现了该算法. 实验的硬件平台是有 2 颗 2.40 GHz 的 Intel® Pentium®4 Xeon™CPU,主存为 1 G MB 的轻负载工作站,运行 Linux 操作系统,内核版本是 2.4.20-8. 用来评测算法性能的测试程序包括 SPEC JVM98 中的几个评测标准,以及计算各种不同种类三角形面积的_ area _. 测试结果参见表 1.

表 1 运行评测标准的结果

SpecJVM98 内容	优化前/s	优化后/s	性能提升/%
_201 _ compress	5.727	5.497	4.18
_202 _ jess	4.709	4.772	-1.34
_209 _ db	19.107	18.353	4.11
_213 _ javac	14.333	13.922	2.95
_222 _ mpegaudio	4.118	4.051	1.65
_228 _ jack	6.351	5.878	8.05
_ area _	10.032	9.409	6.62

通过表 1 中的数据,可以看出在采用了动态内联策略后,应用程序的性能得到了一定程度的提升. 其中_ area _计算了 3 种三角形的面积,如果只利用静态信息,那么将丧失优化机会,在使用了动态信息后,就可以准确判断当前计算的三角形种类,于是性能得到了较大的提高. 从图 2 可以看出多数评测标准的优化效果还是比较明显的,但是_ 202 _ jess 的性能在应用动态策略后有了 1.34% 的下降,主要原因是在这个评测标准中,虚方法的接收者对象在多数情况下是父类的对象,所以动态内联策略的优势体现不出来,但是搜集所需的动态信息却增加了系统的运行时开销,因此会使整体性能有所下降,但是下降的幅度仍在合理的范围内. 这个例子同时也说明了动态内联策略所带来的运行时开销是比较低的. 图 2 给出了动态内联策略相对于已有内联策略的加速比.

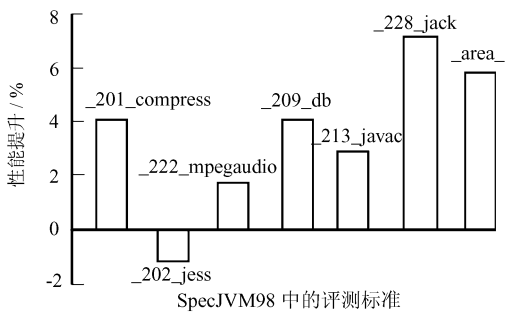


图 2 动态内联策略相对于已有内联策略的加速比

4 结 论

基于在线反馈信息的动态内联算法弥补了已有内联策略的某些缺陷,同时也解决了基于计数的热点探测机制中存在的问题.

在 Java 语言的源程序中,虚方法的调用是很常见的,尤其是一些大型的应用系统,因此准确地判断虚方法的接收者对象的类型就显得非常重要. 本文提出的动态内联算法很好地实现了这一功能,使应用程序的性能得到了提升,对于其他的应用程序,运行时增加的开销也不足 1.5%.

参考文献 (**References**)

[1] Gosling J, Joy B, Steele G. The Java language specification [M]. MA:Addison Wesley, 1996

[2] Ayers A, Gottlieb R, Schooler R. Aggressive inlining[A]. Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation [C]. New York: ACM Press, 1997.134 ~ 145

[3] Cierniak M, Lueh G Y, Stichnoth J M. Practicing JUDO: Java under dynamic optimizations[A]. ACM SIGPLAN Notices[C]. New York:ACM Press, 2000.13 ~ 26

[4] Detlef D, Agesen O. Inlining of virtual methods[A]. Proceedings of the 13th European Conference on Object-Oriented Programming [C]. London:Springer-Verlag, 1999.258 ~ 278

[5] Sun Cop. The Java HotSpot virtual machine, v1.4.1[EB/OL]. http://Java. sun. com/products/hotspot/docs/whitepaper/Java _Hotspot _v1.4.1/Java _ HSpot _ WP _ v1.4.1 _ 1002 _ 1. html,2002-09/2005-03

[6] Burke M G, Choi J D, Fink S, *et al*. The Jalapeño dynamic optimization compiler for Java[A]. Proceedings ACM 1999 Java Grande Conference[C]. New York:ACM Press, 1999

[7] Holzle U, Ungar D. Reconciling responsiveness with performance in pure object-oriented languages [J]. ACM Transactions on Programming Languages and Systems, 1996,18:355 ~ 400

[8] Cierniak M, Eng M, Glew N, *et al*. The open runtime platform: a flexible high-performance managed runtime environment [J]. Intel Technology Journal, 2003, 7(1):5 ~ 18

[9] Suganuma T, Ogasawara T, Takeuchi M, *et al*. Overview of the IBM Java just-in-time compiler [J]. IBM Systems Journal, 2000, 39(1):175 ~ 193