ISSN 1000-9825, CODEN RUXUEW
Journal of Software, Vol.18, No.10, October 2007, pp.2393−2402
DOI: 10.1360/jos182393

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel/Fax: +86-10-62562563

基于持续监控的动态内联优化*

王 雷+, 周 晶, 金茂忠

(北京航空航天大学 计算机学院,北京 100083)

Dynamic Inlining Based on Continuous Monitoring

WANG Lei+, ZHOU Jing, JIN Mao-Zhong

(School of Computer, BeiHang University, Beijing 100083, China)

+ Corresponding author: Phn: +86-10-82317621, Fax: +86-10-82328213, E-mail: wanglei@buaa.edu.cn

Wang L, Zhou J, Jin MZ. Dynamic inlining based on continuous monitoring. *Journal of Software*, 2007,18(10): 2393−2402. http://www.jos.org.cn/1000-9825/18/2393.htm

Abstract: Dynamic compilation is an effective optimization, but the overhead is too heavy, the accuracy of information is not enough, and the redundant code is growing rapidly in the current information collection and continuous monitoring. In this paper, the dynamic compiler is designed based on online feedback and continuous monitoring in Intel ORP (open runtime platform). To solve these problems in information collection, the program instrumentation technique is improved. The continuous monitoring for the type of virtual method's receiver object is implemented. Based on the information from online feedback and continuous monitor, the dynamic inlining optimization is invoked by the compiler. The dynamic unloading algorithm that releases the redundant code in the dynamic optimization is introduced. The results of SpecJVM98 and Java Grande Forum Benchmark show that the performance of JVM is improved on average and the system load is reduced by the dynamic unloading algorithm.
Key words: dynamic compilation; inlining; continuous optimization

摘 要: 动态编译技术是非常有效的一项优化技术,但是,当前的信息采集与持续监控技术面临运行开销过大、信息精度不够以及代码过渡膨胀等问题.以 Intel ORP(open runtime platform)作为基础平台,设计了基于在线反馈与持续监控的动态编译系统;根据当前信息采集技术存在的一些问题改进了代码插装机制;实现了对虚方法接收者对象的类型持续监控;编译系统根据在线采集和持续监控所获得的信息指导内联优化;针对持续监控过程中产生大量无用代码的问题,提出了已编译代码动态卸载方法.SpecJVM98 和 Java Grande Forum Benchmark 等测试基准的运行结果表明,被测程序的平均性能得到了提高.同时,代码动态卸载算法也有效地减轻了系统的运行时负载.
关键词: 动态编译;内联优化;持续优化
中图法分类号: TP314 **文献标识码**: A

动态编译技术在20世纪60年代就已经出现,被应用在Lisp,Fortran,Smalltalk,Self,BASIC,C等语言中[1],但并没有得到广泛的重视.直到Internet的高速发展和Java语言的出现,动态编译技术才得到了长足的发展,并成为当

* Supported by the Beijing Natural Science Foundation of China under Grant No.4023012 (北京市自然科学基金); the Intel China Research Center (Intel中国研究中心)

Received 2006-08-07; Accepted 2006-11-06

前研究热点.为了提高Java程序性能,首先出现了即时(just in time,简称JIT)编译技术[2],其主要思想是在程序开始执行时将字节码文件编译成本地机器码,这在很大程度上提高了程序的执行速度.此后,出现了选择性优化(selective optimization)技术和自适应优化(adaptive optimization)技术.选择性优化的主要特点是在程序执行过程中动态选择一些热点(hot spots)方法,对其实施高级的优化策略,Hotspot JVM[3],IBM Java JIT compiler[4],JUDO[5]等系统采用了这种优化方式.自适应优化[6]是一种混合执行的机制,实际上是将即时编译器和解释器结合起来,有选择地编译和优化频繁执行的方法.自适应优化基于这样一个经验:一般程序中,80%的程序执行时间集中在 20%的代码段上,对这 20%的代码段进行编译、优化将能取得较好的效果.目前,动态编译优化是Java虚拟机中非常有效的一项优化技术,同时也是一个非常有挑战性的领域.由于动态编译发生在程序的执行过程中,所以,编译系统必须在编译开销和优化所能带来的性能提升上进行仔细地均衡.但是,很好地做到这一点往往是很困难的.当前,信息采集与持续监控技术面临运行开销过大、信息精度不够以及代码过渡膨胀等问题.本文以Intel ORP(open runtime platform)[7]作为基础平台,设计了基于在线反馈与持续监控的动态编译系统;分析了当前信息采集(profiling)技术存在的一些问题,改进了代码插装机制;实现了对虚方法接收者对象(receiver object)的类型持续监控;将在线采集信息和持续监控获得的信息,用于指导内联优化;提出了已编译代码(JITed code)动态卸载方法.最后的测试基准表明,本文的工作提高了系统的整体性能,一定程度地减轻了系统负荷.

1 动态编译系统框架

我们所设计的动态编译系统框架如图 1 所示,该系统是一个闭路反馈系统,能够根据在线采集的反馈信息和持续监控的结果选择优化方法,进行动态编译.

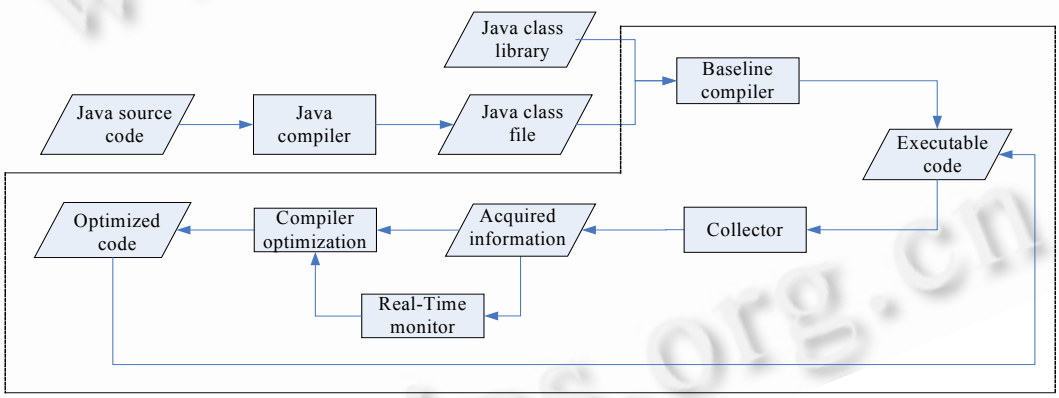


Fig.1 The framework of dynamic compiler

图 1 动态编译系统框架

在该系统中,Java 程序的整个运行流程概述如下:

首先,Java 程序经过 Java 编译器的编译后生成相应的类文件,这里的 Java 语言编译器可以是 Sun JDK(Java development kit)的 Javac 编译器、GNU 的 GCJ(GNU compiler for Java)编译器或者其他任何可以将 Java 的源文件编译成字节码文件的编译器.编译好的类文件与 Java 类库一起作为输入,进入 Java 动态运行环境.图 1 中虚线框部分是基于在线反馈与持续监控的动态编译系统,由基线编译器、优化编译器、信息采集器和实时监控器组成.

该系统的输入是编译好的类文件与 Java 的类库.当第一次执行某个程序时,首先使用基线编译器生成可执行代码.然后,通过信息采集器在程序运行时采集一些动态信息,例如方法和循环的执行频度、Cache 命中率、分支命中率等等.采集到的反馈信息可以被实时监控器和优化编译器使用,优化编译器通过对这些信息的分析,制定有效的优化策略,并对代码进行重新编译.实时监控器用来对程序行为进行持续监控,其目的是发现程序的行为是否发生变化.如果存在某种变化,我们称该程序产生了“状态迁移(phase shift)”.这时,已有的优化策略有可

能失效,实时监控器需要引发优化编译器重新对代码进行优化.静态编译的优化策略是固定的,而动态编译技术可以根据反馈信息灵活地在运行过程中针对某些关键代码进行深度优化.另外,系统可以根据实际情况逐步优化程序,直到最终生成高质量的代码,同时避免了编译时间和系统资源的浪费.我们使用 Intel ORP 的 O1 编译器作为基线编译器,O3 编译器作为优化编译器.

在信息采集器设计过程中,我们分析了传统的基于代码插装技术中的缺点,改进了信息采集技术,可以更加准确地判断虚方法的接收者对象类型;在实时监控器中,实现了在程序运行时对同一调用点处的虚方法接收者对象类型的持续监控;在虚方法的接收者对象类型发生变化时,可以准确地检测到这种迁移,并激活优化编译器,实现了动态内联优化.

2 信息采集技术的改进

动态编译技术中的一项关键技术就是信息采集技术,目前广泛采用的信息采集技术主要有:基于采样的(sample-based)方法[8−10]和代码插装(instrumentation)的方法[3,5,11].基于采样的方法通过周期性地扫描堆栈栈顶,如果在若干次扫描中发现栈顶总是同一种方法,则认为该方法是热点方法.JikesRVM[4]在探测热点方法时采用的是基于采样的机制,所以展开一层堆栈就可以得到栈顶方法的调用者,因此,对虚方法接收者对象的具体类型可以非常准确地进行判断.但是,采样方法收集来的信息往往不够详细、精确,例如,它一般不能收集基本块执行的频度信息,不足以用来指导某些类型的优化.代码插装的方法通过向正在运行的程序中插装一些代码来获取所需信息,可以精确地判断出方法的执行次数.采用这种技术的系统主要有HotSpot JVM[3]和ORP[7].但是,当前的代码插装技术无法确定方法间的调用关系.正是由于这一缺陷,采用该方法的编译系统不能准确地判断出虚方法接收者对象的类型.因此,在确定真正的热点方法以及进行保护内联(guarded inlining)操作时,通常采用比较保守的策略,而这会导致一些优化机会的丧失.

在本文的信息采集器中,我们采用了代码插装技术,在每种方法的入口点和循环返回处插入计数器,并为每种方法建立信息采集(profile)记录,统计执行次数.由于一种方法对同一种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.为了确定运行时方法间的动态调用关系,我们构建了两个映射 *BC_Map* 和 *Guarded_Rec*.*BC_Map* 是对虚方法进行调用的字节码与其相应机器指令的映射,*BC_Map* 根据方法 *m* 的返回地址及其调用者(caller),得到对方法 *m* 进行调用的字节码在其调用者中的索引.为了准确地描述调用关系,本文使用字节码索引 *bc_index* 来标记这一位置.*BC_Map* 定义如下:

定义 1. 若程序P中有n种方法m_i,则$BC_Map=\{e_{m_i} | m_i \in P, 1<i<n\}$,*BC_Map*中的每一项$e_{m_i}$由以下两部分组成:

1) 有且仅有一个 *Method* 域,该域表示这一项所对应的方法;
2) 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*).

由于只有虚方法才存在动态绑定的问题,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*⟩.采用这种方式构建 *BC_Map*,不仅压缩了 *BC_Map* 的大小,同时提高了对 *BC_Map* 的检索效率.我们修改了基线编译器(ORP O1 编译器),让它对方法进行第一次编译时构建 *BC_Map*.

使用 *BC_Map* 对方法间调用关系进行确定的算法如下:

(1) 展开运行时堆栈,得到当前运行方法 *m* 的调用者 *p*,以及方法 *m* 在其调用者 *p* 中的返回地址 IP;
(2) 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.如果存在一致的项,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*.如果不存在,返回 Null.

上述算法可以得到当前正在运行方法 *m* 在其调用者 *p* 中的字节码索引 *bc_index*.但是,该过程是由被调方法(callee)完成,而内联优化是调用方法对被调用方法的内联过程.因此,我们需要为每种方法建立一个 *Guarded_Rec* 结构,记录在方法调用点(call-site)处的被调方法.*Guarded_Rec* 定义如下:

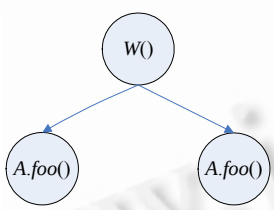
定义 2. 若i为方法m的一个调用点(call site),C为方法m的调用点集合,则$Guarded_Rec=\{r_i|i\in C\}$,其中,$r_i$为二元组$\langle callee_i,bc_index_i\rangle$,$callee_i$代表方法$m$中的被调用者(callee),$bc_index_i$代表调用点所对应的字节码索引.例如,$\langle callee_0,bc_index_0\rangle$表示方法$m$在$bc_index_0$处调用方法$callee_0$.

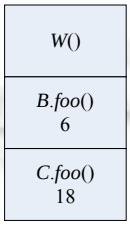
对于 Java 中的虚方法调用,同一个调用点可能调用具有继承关系的几个类的同名方法,虚拟机需要查找虚方法表来确定具体要执行的方法.ORP 采用"类比较"方法,即用当前对象的虚方法表地址和被内联虚方法所属类型的虚方法表地址进行对比:如果两者不同,那么按照缺省的方式去执行;否则,执行被内联的代码.本文称被内联虚方法所属类型的虚方法表是"目标虚方法表".由于 ORP 根据接收者对象的静态类型构造目标虚方法表,因而丧失了一些优化机会.

本文通过构造 *BC_Map* 和 *Guarded_Rec* 得到运行时方法间的动态调用关系,准确地确定虚方法接受者对象的类型,从而更加精确地检查虚方法的执行频度,建立目标虚方法表,提高内联优化的效率.下面用一个实例进行简单说明.

实例 1:有 *A*,*B*,*C* 三个类,*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).对于图 2 中的语句,在确定热点方法时,如果不知道程序的动态调用关系,只能按照静态调用关系(如图 3 所示)检查 *A.foo*()的 profile 记录,以确定 *A.foo*()是否为热点方法.此种情况下,*A.foo*()显然不是,因此,编译器将不考虑对 *x.foo*()进行内联优化.而我们通过 *BC_Map* 可以得到方法 *W* 的 *Guarded_Rec*,如图 4 所示.该结构表明,方法 *W* 在 *bc_index* 为 6 的第 1 个调用点处调用了 *B.foo*(),在 *bc_index* 为 18 的第 2 个调用点处调用了 *C.foo*().有了这些信息,编译器能够构建动态程序调用图,如图 5 所示.利用这些信息,对于调用点 1,我们首先判断方法 *x.foo*()是否被覆盖.实例 1 中 *x.foo*()被覆盖了,所以,检查方法 *W*()的 *Guarded_Rec* 记录可以确定该处的 *foo*()方法为 *B.foo*(),从而去检查 *B.foo*()的信息采集记录.同理,对于调用点 2,编译器知道该处的调用方法是 *C.foo*().因此,编译器通过得到准确的方法动态调用关系,进行更有效的优化.文献[12]给出了优化算法的详细描述.但是,如果在我们完成优化之后发生了"状态迁移",那么这种优化方法将会失效.因此,下面我们将详细介绍基于持续监控的动态内联优化.

```
Void W(){
    A x=A.factory(i);
    …
    x.foo();//B.foo()
    …
    x.foo();//C.foo()
}
```





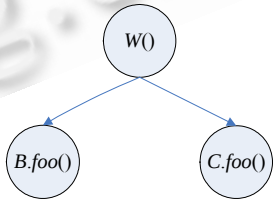


Fig.2 Example 1 Fig.3 Static call graph Fig.4 *Guarded_Rec* of method *W* Fig.5 Dynamic call graph

图 2 实例 1 图 3 静态调用图 图 4 方法 *W* 的 *Guarded_Rec* 图 5 动态调用图

3 程序持续监控的研究

持续监控中的一个重要概念就是"状态迁移",持续监控的主要目的也是发现状态迁移.对于不同的优化类型,"状态迁移"有着不同的含义.例如:对于分支预测,状态迁移是指经常执行的路径由一个分支迁移到另外一个分支;对于代码布局,状态迁移可以定义为经常执行的代码块由*A*,*B*,*C*变成了*A*,*D*,*E*等等.Kister[13]周期性地采集一些向量,计算向量的相似性.如果这个相似性结果超过了某一个阈值,那么表明正在运行的程序发生了状态迁移.Nagpurkar[14]将系统分为两个模块:第 1 个模块通过比较当前窗口(current window)和后续窗口(trailing window)中元素的相似性来确定程序运行阶段的相似性,并将结果作为第 2 个模块的输入;第 2 个模块对这些输

入进行分析,并判断是否发生状态迁移.

目前,持续监控是一个很具挑战性的研究领域,对于某些优化,例如方法内联和代码布局等,如果编译器能够检测到状态迁移并采取相应的优化策略,将会对程序的性能有很大提高.但是,由于持续监控的开销很大,因此,选择哪些因素进行监控成为一个关键问题.表 1 给出了 SpecJVM98 几个测试用例中有关虚方法调用的统计数据,其中,第 2 列的 call sites 是指测试基准中调用点(call site)的数目,第 3 列的 virtual(%)是指虚方法调用在所有调用点中的比例,第 4 列的 calls 是指所有调用点处方法的执行次数,第 5 列是指虚方法执行次数所占的百分比.根据这些数据不难看出,在 Java 语言中,虚方法的使用非常频繁,这里也存在大量的优化机会.因此,本文将重点集中在对虚方法接收者对象类型的监控上.

Table 1 The ratio of virtual method calls in SpecJVM98

表 1 SpecJVM98 中的虚方法调用比例

Benchmarks	Call sites	Virtual (%)	Calls	Virtual (%)
_201compress	2 953	61.9	18 163 854	86.7
_202_jess	4 771	65.6	6 188 662	94.0
_209_db	3 261	62.9	3 202 215	72.8
_228_jack	5 117	63.2	5 643 344	65.0

3.1 虚方法接收者对象类型的持续监控

虚方法区别于一般方法的特点就是虚方法的接收者对象的类型要在运行时才能确定,而且同一调用点处的虚方法调用情况会随着程序运行而发生变化.在构建 *BC_Map* 时,我们保证对于每一个调用点,该调用点处的方法都是最近被调用的方法.但是,由于状态迁移后的虚方法是由优化编译器生成的可执行代码,而 *BC_Map* 是由基线编译器所构建,所以,优化后的虚方法返回地址不在 *BC_Map* 的地址范围内.下面,我们以实例 1 为例进行说明,方法 *W* 被编译后的代码如图 6 所示,其中,图 6(a)为基线编译器生成的代码,图 6(b)为优化编译器生成的代码.

```
W(){
        A x = A.factory(i);
        x.foo();   // B.foo()
        . . .
}
```

(a)

```
W(){
        A x = A.factory(i);
        mov eax, DWORD PTR [eax]   // get vtable
        cmp eax, 0bc3508h
        jnz _default_invocation
        content of B.foo()
        _default_invocation:
        x.foo();
        . . .
}
```

(b)

Fig.6 The result of *W*() after compilation

图 6 *W*()的编译结果

为了更加清楚地说明问题,我们只给出了 *W*()中第 1 个调用点处的信息,用粗体字标明.由于 *BC_Map* 是由基线编译器构建的,所以当优化编译器根据 *B.foo*()的返回地址去检索 *BC_Map* 时,它会根据 *B.foo*()的调用点(图 6(a)中的粗体字)得到 *B.foo*()在 *W*()中的字节码索引.假设 *W*()被优化编译器编译后,调用点 1 发生了状态迁移,即该处的被调用方法由 *B.foo*()变成了 *C.foo*(),这时,调用点 1 的位置就由图 6(a)中的位置变成了图 6(b)中的位置(粗体字处).由于两个编译器代码的地址范围不同,所以无法通过检索 *BC_Map* 得到 *C.foo*()在 *W*()中的字节码索引.

根据以上分析,当虚方法的接收者对象类型发生变化时,已有的 *BC_Map* 已经不能满足要求,因此,本文构建了新的映射 O3_*BC_Map*.O3_*BC_Map* 维护了一个"*real_caller*",用来保存相关方法的真正调用者.除了需要维护 *real_caller*,O3_*BC_Map* 还包含一个 *root* 项,该项保存被优化的起始方法,这是由于当发生状态迁移时,编译器需要重新编译最外层的起始方法.

定义 3. 若程序P中有n种方法m_i,则O3_*BC_Map*=$\{e'_{m_i} | m_i \in P, 1<i<n\}$,其中,每一项$e'_{m_i}$由以下两部分组成:

1) 有且仅有一个 *root* 域,该域表示优化后的最外层起始方法;
2) 包含一到多个四元组$\langle C,I,B,E\rangle$,其中:
   ① C 表示方法的真正调用者(*real_caller*),该域表示某些被优化方法的真正调用者;
   ② 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;
   ③ 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*);
   ④ 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*).

除了上面这些信息,O3_*BC_Map* 还包括 min_*addr*,max_*addr*,*index_num* 这 3 项信息,它们分别表示 O3_*BC_Map* 中所有项的 *begin_addr* 和 *end_addr* 的最小值与最大值,以及 *bc_index* 的数目.这 3 项信息主要是为了提高检索效率.

构建 O3_*BC_Map* 的过程如下:

1) 记录 *root* 域为当前被编译的方法,*index_num* 清零;
2) 为当前方法中的字节码 *bc_i* 构建四元组$\langle C,I,B,E\rangle$:
   ① 首先要判断当前的字节码 *bc_i* 是否为 *invokevirtual* 和 *invokeinterface*,如果是,那么记录下该字节码的真正调用者,并将该项内容保存进 *real_caller*[*index_num*].同时,将该字节码在 *real_caller* 中的字节码索引添加进 *bc_index*[*index_num*],并将该字节码所对应机器指令的起始地址 *begin* 和结束地址 *end* 分别添加进与 *bc_index*[*index_num*]对应的 *begin_addr* 和 *end_addr* 域.
   ② 如果该字节码是第 1 条字节码,那么将该字节码对应的 *begin_addr* 添加进 min_*addr*,如果该字节码是最后一条字节码,那么将该字节码对应的 *end_addr* 添加进 max_*addr*.
   ③ 将 *index_num* 加 1.
3) 为当前方法中的每条字节码依次执行"2)"中的操作,直到该方法的最后一条字节码.

由于只有虚方法才存在动态绑定的问题,因此,与构建 *BC_Map* 中的策略一样,构建 O3_*BC_Map* 时也只考虑虚方法的情况.我们修改了优化编译器(ORP O3 编译器),让它对方法进行优化编译时构建 O3_*BC_Map*.

3.2 基于持续监控的动态内联优化

本文持续监控的主要目标是对虚方法接收者对象的类型进行监控.实时监控器检测"状态迁移条件"是否满足;如果满足,表明存在状态迁移.状态迁移条件定义如下.

状态迁移条件:假设方法 m 在其调用者 p 中的返回地址 IP,方法 m 的真正调用者为 *real_caller*,m 在 *real_caller* 中的字节码索引为 *bc_index*,*real_caller* 的保护内联记录为 *Guarded_Rec*,e 为 O3_*BC_Map* 中的一项.若(IP$\in$[*e.begin_addr*,*e.end_addr*])$\wedge$($\langle m$,*bc_index*$\rangle\in$*Guarded_Rec*)为真,则 *real_caller* 在 *bc_index* 处发生了状态迁移.

实时监控器发现程序中存在状态迁移,并且当方法的执行频度超过一定阈值时,引发优化编译器,进行重新编译优化.具体算法如图 7 所示.

算法共分成 4 步:

第 1)步展开运行时堆栈得到当前运行方法 m 的调用者 p,以及方法 m 在调用者 p 中的返回地址 IP;

第 2)步先将标志位"phase_shift"置为 false,表明没有发生状态迁移,接下来开始检索 *BC_Map*:如果能够找到与 IP 相对应的项,那么说明对方法 m 的调用发生在基线编译器生成的代码中,因此直接进行第 4)步的操作;否则,表明方法 m 的调用者已经优化过,因此进行第 3)步的操作,开始检索 O3_*BC_Map*;

第 3)步是算法的核心部分.在这一步,首先要扫描 O3_*BC_Map*,比较返回地址 IP 是否落在某个 O3_*BC_Map* 的[min_*addr*,max_*addr*]中.如果找到这样一个 O3_*BC_Map*,根据 IP 在 O3_*BC_Map* 的项中找出方法 m 的真正调用者 *real_caller* 以及 m 在 *real_caller* 中的字节码索引 *bc_index*.然后判断状态迁移条件是否满足:如果不满足,那么将方法 m 和它对应的字节码索引添加进 *real_caller* 的 *Guarded_Rec*;如果满足,检查 *root* 方法的执行频度;如果 *root* 执行的次数大于阈值α,表明 *root* 是一种热点方法,因此需要对 *root* 进行重新优化,将标志位

“*phase_shift*”设置为 true;

第 4)步:如果 *phase_shift* 为真,那么对 *root* 进行重新编译;否则,对方法 *m* 进行编译.

```
1) unwindStack();   //Unwind stack in runtime
   p=getCaller(m);   //Get the caller p of the method m
   IP=getReturnAddr(m);   //Get the return address
2) phase_shift=false;
   Scan_BC_Map();
   If (IP in BC_Map) Then 4);
      Else 3);
3) int i=0;
   While (!(IP in [O3_bcmap[i].min_addr, O3_bcmap[i].max_addr]) && (i<O3_BC_MAP_NUM))
       i++;
   //Search O3_BC_Map
   If (IP in [O3_bcmap[i].min_addr, O3_bcmap[i].max_addr])
    For (int j=0; j<O3_bcmap[i].index_num; j++)
    If ((IP in [O3_bcmap[i].begin_addr[j], O3_bcmap[i].end_addr[j]])
                        real_caller=O3_bcmap[i].caller_handle[j];
                        num=(real_caller)→callee_num;
                        int k=0;
                        While ((k<=num) &&
                        ((real_caller)→guarded_rec.bc_index[k]!=O3_bcmap[i].bc_index[j]))
                              k++;
                        If ((real_caller)→guarded_rec.bc_index[k]==O3_bcmap[i].bc_index[j])
                              (real_caller)→guarded_rec.callee_handle[k]=m;
                              root_mh=O3_bcmap[i].root;
                              root_prof=O1_method_get_profile_info(root_mh);
                              If ((root_prof→m_entry)>α)
                                    phase_shift=true;
                        Else
                              (real_caller)→guarded_rec.callee_handle[num]=m;
                              (real_caller)→guarded_rec.bc_index[num]=bcmap[i].bc_index[j];
                              (real_caller)→callee_num++;
                        break;
   4) If (phase_shift)
              method_recompile(root_mh,O3_jit);
   Else
              method_recompile(m,O3_jit);
```

Fig.7 The algorithm of dynamic inlining

图 7 动态内联优化算法

这里需要强调的一点就是,不一定发生了状态迁移就对方法进行重新编译,这时还要进一步分析,只有当方法*root*的执行频度到达一定程度之后,才重新编译.本文运行Java Grande Forum Benchmark[15]测试基准对0,200,400,600,800 等阈值进行了测试.实验结果见表 2.

Table 2 The result of different thresholds

表 2 频度阈值的实验结果

Java grande forum benchmark suit	0	200	400	600	800
JGFSearchBenchSizeA (s)	19.202	19.344	18.951	19.305	19.076
JGFRayFracerBenchSizeA (s)	10.675	10.692	10.693	10.686	10.714
JGFSeriesBenchSizeA (s)	36.382	36.781	37.289	36.802	36.685

表 2 中的实验结果表明,不同阈值对测试基准程序的性能产生了不同的影响,在不同测试基准程序之间的优化效果也不同,不存在一个对所有测试基准都是最优的阈值.通常情况下,产生状态迁移的虚方法执行次数会高出 *root* 很多,因此对 *root* 执行频度的限制不应过高,本文将阈值设为相对较低的 400.

3.3 代码动态卸载算法

程序持续监控的主要目的是发现状态迁移,并引发编译器进行重编译.编译后每次执行的都是新代码,系统中会出现很多无用的旧代码.如果这些旧代码得不到及时的处理,就会给系统带来负担.因此,持续监控系统有必要提供一种对已编译代码进行动态卸载的机制,用来在运行时对旧代码进行动态回收.除了持续监控系统,具有多个编译器的虚拟机系统也都存在类似的问题.

由于 ORP 虚拟机为每种方法维护了一个 *JIT_info* 数据结构保存与编译器相关的信息,其中包括编译器标识、异常表地址以及指向可执行代码的指针等信息.因此,我们在算法中设置了两个代:*jit_infos_first* 和 *jit_infos_second*,其中,*jit_infos_first* 用来存放最新的 *JIT_info*,大小是 6;*jit_infos_second* 存放旧的 *JIT_info*,大小是 10.整个算法分为两步,具体流程如下:

```
1) jit_info_O3=get_O3_JIT_info();
   jit_info_O1=get_O1_JIT_info();
   jit_info_O1→mark_release();
   jit_info_O3→mark_release();
   addto_jit_infos_first(jit_info_O1);
   addto_jit_infos_first(jit_info_O3);
2) If (jit_infos_first_is_full())
       trace_stack_mark_unrelease();
           For (each jit_infos_first_i in jit_infos_first)
               If (jit_infos_first_i→is_release())
                   release_jit_info(jit_infos_first_i);
       Else
                   jit_infos_first_i→mark_release();
           addto_jit_infos_second(jit_infos_first_i);
                   If (jit_infos_second_is_full())
                       trace_stack_mark_unrelease();
                       For (jit_infos_second_j in jit_infos_second)
                           If (jit_infos_second_j→is_release())
                               release_jit_info(jit_infos_second_j);
                       rearrange_jit_info(jit_infos_second);
                   If (second_num>=0.8*second_capacity)
                       reallocate(second_capacity*2);
               jit_infos_first_empty();
```

上述流程第 2)步是算法的核心部分,用来释放冗余的代码.首先,编译器要判断第一个代是否已经满了:如果未满则不进行释放;否则开始释放.释放时,首先要对运行时的堆栈进行扫描,在扫描的过程中,将堆栈上的方法标记为“不可释放”,由于堆栈上的方法正在被使用,因此不能释放.这样,第一个代 *jit_infos_first* 中不在堆栈上的方法将可以被释放.

4 实验与性能分析

我们采用SpecJVM98[16]和Java Grande Forum Benchmark(JGF)[15]作为测试基准,对本文的优化算法进行了评测.SpecJVM98 是由标准性能评测公司(Standard Performance Evaluation Corporation)开发的测试基准,包括多个测试程序,用来在各个方面对Java虚拟机的性能进行评测.Java Grande Forum Benchmark是由Java Grande Forum发布的一组测试基准,主要用来测试对网络带宽、处理能力以及内存都有较高要求的大规模Java程序,每一个测试程序都对应两到三种不同的大小(size).实验硬件平台为:CPU为AMD Athlon™ XP 2500+,主存为 256 MB,运行Linux操作系统,内核版本是 2.4.20-8.

4.1 基于持续监控的内联优化实验结果

上述实验环境下,在阈值设置为 400 时,对基于持续监控的动态内联优化的实验结果见表 3.表 3 中第 1 列前 4 行为SpecJVM98 中的几个测试用例,后 6 行为Java Grande Forum Benchmark测试基准,其中包括SizeA和

SizeB两种大小.第 2 列为在原来ORP上的运行结果.ORP系统采用深度优先的内联优化算法[7],在确定虚方法接受者对象的类型时,使用了保守的静态推测机制.第 3 列是采用持续监控优化方法后的结果.第 4 列为性能提升的百分比.通过表 3 我们可以看出,持续监控策略对SpecJVM98 测试基准的性能有一定提升,但_202_jess有明显的下降;Java Grande Forum Benchmark测试基准的整体性能有很大提升,但JGFSOR性能略微有所下降.主要原因是这两个测试基准中的状态迁移不很频繁.而持续监控的开销由信息采集、在线分析、确定优化策略和引发重编译组成,因此运行开销比较大,如果优化的效果不能弥补这种开销,就会造成性能下降.因此,持续监控优化主要适用于迁移率较高的、规模较大的计算密集型程序.

Table 3 Experimental results

表 3 实验结果

Benchmark	ORP/s	Continuous monitoring (s)	Result (%)
_201_compress	10.324	9.931	3.96
_202_jess	9.156	9.460	–3.21
_209_db	28.43	27.961	1.68
_228_jack	9.531	9.457	0.78
JGFSearchBenchSizeA	19.526	18.951	3.03
JGFSearchBenchSizeB	86.513	84.782	2.04
JGFRayFracerBenchSizeA	12.696	10.693	18.73
JGFRayFracerBenchSizeB	140.518	114.93	22.26
JGFSeriesBenchSizeA	38.501	37.289	3.25
JGFSeriesBenchSizeB	385.973	374.388	3.09
JGFSORBenchSizeA	4.162	4.17	–0.19
JGFSORBenchSizeB	9.291	9.319	–0.30

4.2 已编译代码动态卸载实验结果

为了减轻系统的负载,本文提出了一种对已编译代码的动态卸载算法,在程序的运行过程中对系统中不再使用的代码和数据信息进行卸载.表 4 为运行 SpecJVM98 以及 Java Grande Forum Benchmark 测试基准的结果.表 4 的结果表明,动态卸载算法释放了系统中的无用代码,在一定程度上减轻了系统的负载.

Table 4 The dynamic unloading code

表 4 动态卸载代码

Benchmark	Unloading code size (byte)
_201_compress	213 787
_202_jess	3 880 739
_209_db	558 154
_228_jack	15 794 270
JGFSearchBenchSizeA	39 669
JGFRayFracerBenchSizeA	64 072
JGFSeriesBenchSizeA	2 233
JGFSORBenchSizeA	2 630

5 结束语

本文分析了当前信息采集技术所存在的问题,通过改进代码插装机制可以准确地得到方法间的动态调用关系,同时实现了对虚方法接收者对象类型的持续监控.该算法可以及时检测到程序中的状态迁移,并引发优化编译器进行重新优化.基于 ORP 平台,设计了动态编译系统的框架,并实现了基于持续监控的动态内联优化.针对持续监控系统产生大量无用代码的问题,提出一种动态代码卸载算法,该算法不仅可以解决持续监控系统中的问题,而且还可以应用到其他具有多个编译器的虚拟机系统中.SpecJVM98,Java Grande Forum Benchmark 等测试基准的运行结果表明,本文的优化算法使程序的平均性能得到了提高,同时,代码动态卸载算法也有效地降低了系统的运行时负载.本文目前仅将持续监控策略用于方法内联优化,以后可以进一步扩展,用于其他类型的优化.另外,本文采用了比较简单的内联决定策略,因此这方面还可以进一步优化.

References:

[1] Bill V. Inside the Java Virtual Machine. 2nd ed., New York: McGraw-Hill Osborne Media, 2000.

[2] Suganuma T, Yasue T, Kawahito M. A dynamic optimization framework for a Java just-in-time compiler. ACM SIGPLAN Notices, 2001,36(11):180−195.

[3] Sun Microsystems. The Java hotspot virtual machine. http://java.sun.com/products/hotsport/ docs/whitepaper

[4] Arnold M, Fink S, Grove D, Hind M, Sweeney P. Adaptive optimization in the Jalapeño JVM. ACM SIGPLAN Notices, 2000, 35(10):47−65.

[5] Cierniak M, Lueh GY, Stichnoth JM. Practicing JUDO: Java under dynamic optimizations. ACM SIGPLAN Notices, 2000,35(5): 13−26.

[6] Steve M. The Java HotSpot (tm) perfomance engine: An in-depth look. 1999. http://java.sun.com/developer/technicalArticles/ Networking/HotSpot/

[7] Open runtime platform (ORP) from Intel corporations. http://orp.sourceforge.net

[8] Arnold M, Fink S, Grove D, Hind M, Sweeney P. A survey of adaptive optimization in virtual machines. Proc. of the IEEE, 2005,93(2):449−466.

[9] Hölzle U, Ungar D. Reconciling responsiveness with performance in pure object-oriented languages. ACM Trans. on Programming Languages and Systems, 1996,18(4):355−400.

[10] Hazelwood K, Grove D. Adaptive online context-sensitive inlining. In: Proc. of the IEEE Computer Society, Int'l Symp. on Code Generation and Optimization. San Francisco: IEEE Computer Society, 2003. 253−264.

[11] Paleczny M, Vic C, Click C. The Java HotSpot server compiler. In: Proc. of the Java Virtual Machine Research and Technology Symp. Monterey: USENIX Association, 2001. 1−12.

[12] Zhou J, Wang L, Liu ZC. Dynamic inlining scheme improvement in JVM. Journal of Beijing University of Aeronautics and Astronautics, 2006,32(3):352−356 (in Chinese with English abstract).

[13] Kistler T, Franz M. Continuous program optimization: A case study. ACM Trans. on Programming Languages and Systems, 2003, 25(4):500−548.

[14] Nagpurkar P, Hind M, Krintz C, Sweeney P, Rajan V. Online phase detection algorithms. In: Proc. of the 4th Annual Int'l Symp. on Code Generation and Optimization (CGO). New York: IEEE Computer Society, 2006. 111−123.

[15] Java grande forum. http://www.epcc.ed.ac.uk/javagrande/

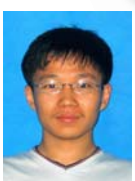
[16] SPEC JVM98 benchmarks. http://www.specbench.org/osg/jvm98

附中文参考文献:

[12] 周晶,王雷,刘志成.Java 虚拟机中动态内联策略的改进.北京航空航天大学学报,2006,32(3):352−356.

王雷(1969－),男,黑龙江佳木斯人,博士,副教授,主要研究领域为编译技术,操作系统,软件工程.

金茂忠(1941－),男,教授,博士生导师,主要研究领域为编译技术,软件工程.



周晶(1981－),男,硕士生,主要研究领域为编译技术,操作系统.