

基于逃逸分析的循环中栈式分配优化研究

王 雷 徐 星

(北京航空航天大学计算机学院 北京 100083)

摘 要 栈式内存分配可以有效地提高 Java 程序的执行效率,但是在循环中,对象的栈式分配比率和栈空间的大小变成了一对很难协调的矛盾.文中实现了一种控制流非敏感(flow-insensitive)的、过程间(inter-procedural)的、上下文相关(context-sensitive)的逃逸分析(escape analysis)方法.在此基础上,提出以循环为基本单位的分配策略,引入了对象栈和区域栈帧等概念;通过对循环的分析,实现了基于逃逸分析的栈式分配. SPECjvm98 测试基准表明,在可控栈空间大小的条件下,该算法的栈式分配比率达到 8.3%~25%(平均 15.18%).

关键词 JVM; 程序分析; 逃逸分析; 栈式分配; 循环分析

中图法分类号 TP316

The Study of Stack Allocation in Loop Based on Escape Analysis

WANG Lei XU Xing

(School of Computer, Beihang University, Beijing 100083)

Abstract The technique of stack allocation improves the efficiency of Java program, but the trade-off between the ratio of allocated objects and the size of stack is difficult. In this paper, a flow-insensitive, inter-procedural, and context-sensitive escape analysis is implemented. The allocation policy based on loop as a basic unit is presented, and the concepts of object stack and stack region are proposed. The authors implement the stack allocation based on escape analysis through the loop analysis. The result of SPECjvm98 shows that 8.3% to 25% (with an average of 15.18%) of all objects could be stack allocated in the new algorithm with the controlled size of stack.

Keywords JVM; program analysis; escape analysis; stack allocation; loop analysis

1 引 言

Java 语言作为现在主流的面向对象程序设计语言,已经得到了工业界的支持和市场的认可.随着网络技术的发展,Java 已经推广到了各种平台,但是在一些资源受限制的环境中性能成为影响其应用的主要因素.

Java 程序以字节码(byte-code)^[1]形式在不同平台的虚拟机上执行.在虚拟机中,Java 程序所需内存可以被分配在堆或者栈内,内存的分配与释放

对整个系统性能有很大影响.栈式分配^[2]与堆分配相比,有 3 个很重要的优点:(1)栈式分配比堆分配减少了同步操作;(2)栈式分配减轻了垃圾收集器的负担;(3)栈空间始终是 Cache 热点,对栈的操作速度明显快于对堆的操作.本文希望通过静态的逃逸分析(escape analysis),在不改变程序语义的前提下进行对象的栈式分配,提高程序执行效率.

逃逸分析^[3]是一种静态分析方法,用以确定数据的生存时间是否超过它的静态域的存在时间.如果对象 O 没有逃逸出方法 M,那么 O 必然也没有逃逸出创建它的线程 T;这样 O 就可以存储在 M 方法

的栈帧中,而不必存储在堆空间中。

在循环中进行对象分配时,对象的栈式分配比率(在栈中分配的对象字节数占总分配字节数的比率)和栈空间的大小变成了一对很难协调的矛盾。如果对循环中的对象进行栈式分配,必然加大空间开销。当栈空间达到一定数值时,将引起 Cache 命中率严重下降,随之系统性能严重下降。最严重的情况是空间开销可能达到不可控的程度,耗尽内存导致系统崩溃。因此,很多论文中对循环中生成的对象不作栈式分配,但是分配循环中对象,就会减少逃逸分析的收益,降低栈式分配比率。Blanchet^[4]和 Gay^[5]通过对循环的分析,分配了循环中可重用的定长对象。但是这些方法不仅增加了分析的开销,而且无法解决变长对象的栈式分配问题。

本文以文献[6]的逃逸分析框架为基础,实现了一种控制流非敏感的、过程间的、上下文相关的全局数据流分析方法。在此基础上,提出了以循环为基本单位的栈式分配方法,引入了对象栈、区域栈帧等概念;实现了基于循环的数据流分析方法,采用深度优先的生成树算法,定位循环的入口点、迭代点和出口点;通过插入栈式分配代码,在 ORP 平台上实现了基于逃逸分析的栈式分配。本文的算法与文献[4-5]区别在于,本文不但可以栈式分配循环中的变长对象进行,而且较好地控制了栈帧空间的增长。

本文第 2 节探讨循环中栈式分配的空间复杂度问题;第 3 节介绍逃逸分析框架、方法描述的构建、方法内和方法间分析算法;第 4 节描述对象栈、区域栈帧等概念以及循环分析和栈式分配的实现;第 5 节对本文的算法进行性能评测和比较;第 6、7 节介绍相关工作并总结本文工作。

2 循环中空间复杂度的探讨

下面是一个非常简单的绘图函数,该函数从 0 到 h 循环,每次循环会在图像中画一行像素:

```
void drawImage (Scene scene, int w, int h, PNGFile
                pngFile) {
    for (int y=0; y<h; y++) {
        Row row=new Row(w);
        row.draw(scene,y);
        pngFile.outputRow(row);
    }
}
```

如果采用 GC (Garbage Collection) 管理内存,所有的 *row* 对象分配在堆中。由于只有最近分配的

对象是可以由根对象集合可达的,所以一旦开始垃圾收集,大部分对象都可以被回收。在效果上,drawImage 函数虽然分配了 $O(hw)$ 的内存,但是它实际的空间复杂度只有 $O(w)$ 。

在栈式分配的情况下,我们发现所有的 *row* 对象都没有逃逸出方法 drawImage,所以每个 *row* 对象都会被分配在方法栈帧内。这种情况下,在方法返回之前,方法栈帧中的空间复杂度为 $O(hw)$ 。

如果采用比较好的栈式分配算法^[5],可以发现每次进入循环时,上次循环使用的 *row* 对象已经无用了,重用这段内存。但是,这种方法提高了分析的复杂度,而且无法处理在循环中分配变长对象的情况。本文采用了以循环为基本单位进行栈式分配的方法,建立以区域栈帧为基础的对象栈,比较妥善的解决了这个问题。

3 逃逸分析框架

本文的系统框架由 3 个模块组成,分别是:基于字节码的逃逸分析框架、逃逸分析相关的优化策略和基于逃逸分析的栈式分配。图 1 描述了逃逸分析的工作流程。

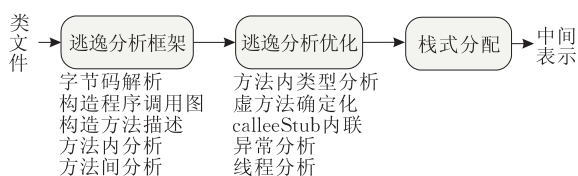


图 1 逃逸分析工作流程

在图 1 中,虚拟机在执行 Java 方法之前,首先启动逃逸分析基本框架模块,该模块解析方法的字节码,构造程序调用图,执行全局的数据流分析,然后应用逃逸分析优化策略和栈式分配优化,把字节码格式类文件翻译成虚拟机内部使用的中间表示形式 (Intermediate Representation, IR),最后,虚拟机的即时编译器再把 IR 优化编译成本地代码。

本文的数据流分析基于程序调用图 $G = \langle P, V \rangle$, 其中 P 是结点,代表方法, V 是边,代表方法调用。构造程序调用图的时候,对于动态绑定点,采用如下的保守规则。

规则 1. 如果在某个动态绑定点调用虚方法、抽象类的方法或者是接口类的方法,则在这个调用点假设调用所有可能的虚方法、抽象类方法和接口类方法的实现。

如图 2 所示,左边的代码在对象 *o* 上调用 *bar*

虚方法,而且 MyClass 和它的子类 MyClass1 和 MyClass2 都实现了 bar 方法,所以根据规则 1,在

bar 虚方法的调用点,可能调用 3 个方法,即这 3 个类中 bar 方法的实现.

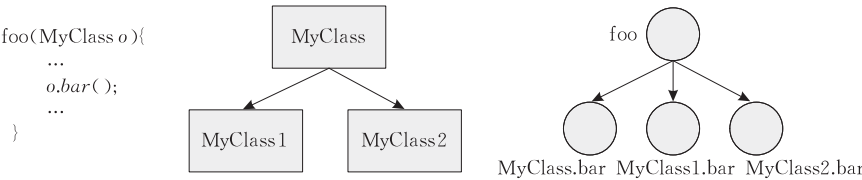


图 2 程序调用图示例 1

为了使程序调用图是一个单根的结构,需要构造一个伪 root 结点,并且把 main 方法、类型初始化方法<clinit>和线程起始点 run 方法直接作为这个伪结点的调用方法,如图 3 所示. 逃逸分析首先在解析 Java 字节码的过程中,构造程序调用图,并且为程序调用图中的每个方法构造方法描述. 然后,在程序调用图中划分出递归调用,即程序调用图中的强连通结点(Strongly Connected Components, SCC). 最后,执行方法间分析和方法内分析.

(varNo)、变量创建点(bytecodeIdx)和变量类型(classHandle),其中变量号是对象变量或者引用变量的唯一标识,变量创建点记录创建这个变量的字节码的位置. calleeStub 记录方法间的调用关系,该数据结构由 3 部分组成,分别是参数表(argMap)、调用点(callSite)和方法描述(methodSummary). calleeStub 数据结构中的参数表表示调用关系的实参,调用点是方法调用发生的字节码位置,方法描述是指被调用方法的方法描述.

构建方法描述过程中,采用如下原则:

- (1) 如果等价类中某个引用或者对象是类的静态域,或者是实现 Runnable 接口的类对象,则标记这个等价类的逃逸状态为 GlobalEscape.
- (2) 如果等价类中某个引用或者对象是方法的形参、返回值或者异常,则标记这个等价类的逃逸状态为 ArgEscape.

3.2 方法内分析

方法内分析的主要目的是保证方法内的逃逸状态稳定. 基本思想:如果某个等价类 A 可以由等价类 B 的 fieldMap 关系可达(reachable),而且 B 的逃逸状态是 GlobalEscape,那么 A 的逃逸状态也被标记为 GlobalEscape.

3.3 方法间分析

方法间分析算法分为 3 步:

- 1. 创建程序调用图,并划分成 SCC.
- 2. 自底向上遍历程序调用图,由 callee 向 caller 传播方法上下文(method_context),然后对 caller 执行方法内分析使其逃逸状态稳定.
- 3. 自顶向下遍历程序调用图,并且由 caller 向 callee 传播逃逸状态.

4 循环中的栈式分配

在逃逸分析基础上,为了以循环为单位进行栈式分配,本文提出了对象栈和区域栈帧等概念,下面将在循环分析的基础上,实现栈式分配优化.

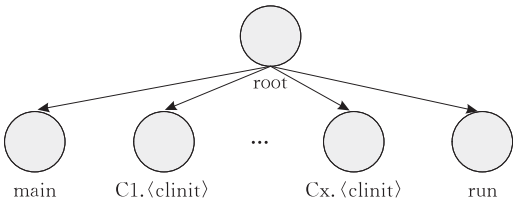


图 3 程序调用图示例 2

3.1 构建方法描述

方法描述(methodSummary)需要记录对象的创建、对象之间的引用关系、方法上下文(Method-Context,即方法的参数、返回值和异常)、子方法调用. 本文使用 calleeStub 数据结构表示方法间的调用关系. 方法描述由 4 部分组成,分别是参数表(argMap)、异常(exception)、等价类(equivalent-Class)和 calleeStub. 参数表包括方法的形参和返回值,并且每个形参、返回值和异常均是变量(varItem). 等价类由 4 部分组成,分别是 fieldMap(域表)、参数向量(argVector)、逃逸状态(escape-Status)和别名引用(即所有可能的别名变量). fieldMap 记录该等价类的域的信息,其中每个域都是引用变量. 参数向量说明该等价类是否表示方法的形参、返回值或者异常. 逃逸状态说明该等价类所包含别名引用的逃逸状态,它的状态值可以是 NoEscape、ArgEscape 或者 GlobalEscape. 逃逸状态之间的大小关系为 NoEscape < ArgEscape < GlobalEscape. 变量(varItem)是指方法中的对象变量和引用变量,它由 3 部分组成,分别是变量号

4.1 对象栈

面向对象语言的实现大量采用了机器栈,所谓机器栈,就是由栈指针寄存器维护的一段内存,一般用来执行子程序跳转、返回等操作;另外,在异常中断处理或者由信号引发的中断处理中,也常用栈指针寄存器维护的栈空间.在 Java 虚拟机中,面向 C 语言的本地方法调用(JNI)就是通过机器栈来实现.

为了提高效率,Java 中的方法调用栈应该尽量通过机器栈来实现.一般来说,方法调用栈的空间在逻辑地址上总是连续的.在本文中,我们希望控制在栈帧中分配对象的数量(避免可能出现的栈空间无限增长);另外,需要在栈式分配中处理变长对象,这意味着栈帧内会变得不连续.这两种特性在机器栈实现是不合适的,因此本文引入了“对象栈”的概念.

定义 1. 对象栈为栈式分配的对象单独设置一块栈空间,我们把它和由栈指针寄存器维护的方法调用栈分开来.

在对象栈的实现过程中,为了确定对象栈的增长是否超过了限制,我们在对象栈的下端放置一个

不可写的页面.每当对象栈增长超过了限制,就会触发信号通知对象栈已满.这种方式的速度非常快,而开销很小.

4.2 区域栈帧及其操作

本文将对象栈分为若干区域,我们称之为区域栈帧,我们为区域栈帧定义了以下几种操作:

- (a) 创建一个新区域栈帧;
- (b) 在当前区域栈帧分配对象或清空对象;
- (c) 释放一个区域栈帧(其中的对象也随之失效).

本文把循环作为栈式分配的基本单元,在循环入口点创建一个区域栈帧;如果循环中创建的对象没有逃逸出它所在的方法,那么将该对象分配在区域栈帧中;在一遍循环结束以后,清空区域栈帧中的对象;在退出循环时候释放当前的区域栈帧.这样,方法调用栈和对象栈之间构成一个比较松散的耦合关系.图 4(a)显示了一段程序和 8 个我们所关注的执行点.图 4(b)显示了 1~8 个点处,方法调用栈和对象栈的状态变化.

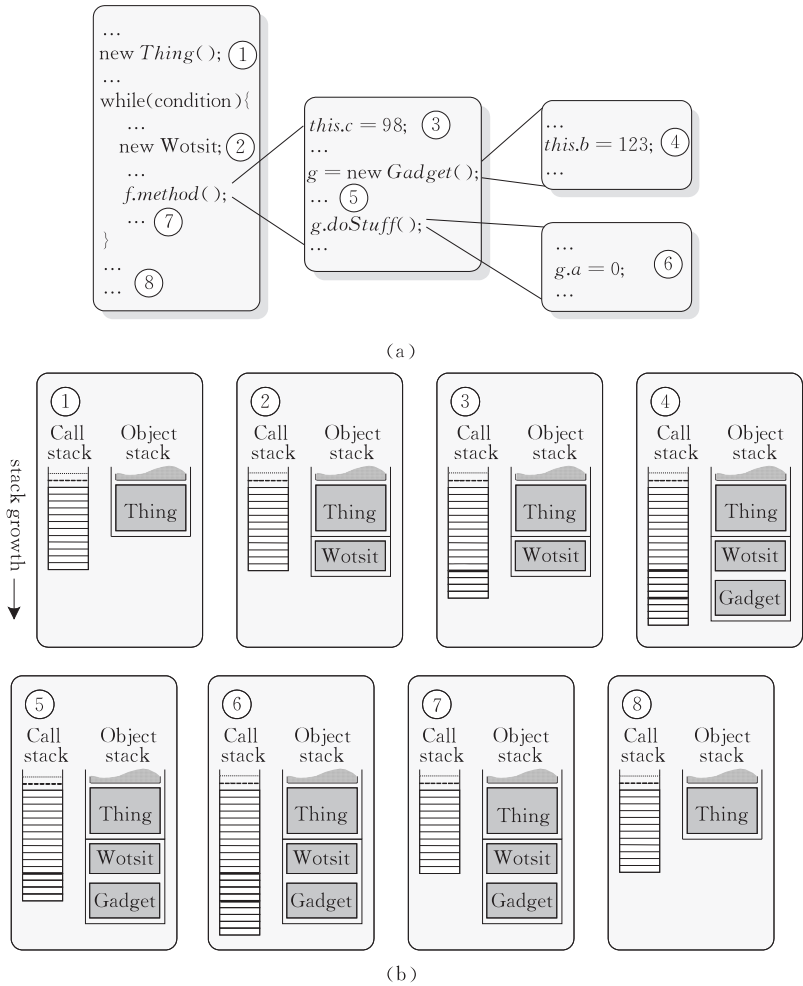


图 4 对象栈和方法调用栈

4.3 循环分析

为了实现基于循环的区域栈帧分配和释放,首先需要为 Java 方法的字节码构建程序流图 FG , $FG=\langle N,E\rangle$,其中 N 为指令节点的集合, E 为边的集合. 本文通过回边来定位流图中循环的位置. 如果从流图入口点到某个节点 B 的每一条路径都包含节点 A ,那么称流图中的节点 A 是节点 B 的必经节点. 如果边的头结点是尾结点的必经结点,那么称这条边是程序流图中的一条回边.

Java 语言不提供 goto 跳转,因此所有方法的流图都是可归约流图. 可归约流图有如下性质^[7]:

(1) E 可以划分成不相交的前向边集合 EF 和后向边集合 EB ,使得 $\langle N,EF\rangle$ 形成一个从入口节点可到达其中每个节点的 DAG(无环有向图),并且 EB 中边的都是回边;

(2) 在此流图中的所有循环都是由回边所刻画的自然循环.

这样,我们可以通过找到流图中的回边,来探测方法中的循环. 为了找到回边,本文使用深度优先排序方式,从初始节点开始搜索整个图,构造出一颗 DFS 生成树. 根据 DFS 树,流图中的边可以分为 3 类:

(1) 树中有从节点 m 到 m 的祖先的边,称这类边为后退边(区别于回边).

(2) 树中由从节点 m 到 m 后代的边,称为前进边,DFS 树中所有的边都是前进边.

(3) 如果 m 和 n 互相不是祖先,称为是交叉边.

根据文献[7]中的证明:如果流图是可归约的,那么后退边正好就是流图中的回边;反之,对于任意的可归约流图,每个回边都是后退边. 值得注意的是,在流图中有以下结论: $\langle m,n\rangle$ 是一条后退边,当且仅当 $dfn[m]\geq dfn[n]$ (在 dfn 数组中保存了节点的深度信息). 根据上述结论,我们能够找到流图中所有的回边.

给定一条回边 $\langle n,d\rangle$,该边所代表的自然循环为: d 加上所有不经过 d 而能到达 n 的节点集合,其中 d 是该循环的入口点. 这样我们可以找出流图中所有的循环体. 找到循环体后,根据哪些指令节点在循环之内,哪些指令节点在循环之外,可以得到 3 类与循环相关的有向边:

- (a) 由循环外指向入口点:进入循环的有向边;
- (b) 由循环内指向入口点:开始一次新循环的有向边;

(c) 由循环内指向循环外:这是离开循环的有向边(这种边不止一条,因为需要考虑 Java 的异常处理);

本文对这 3 类有向边进行特殊处理,在循环入口点插入代码,初始化一个新的区域栈帧;每次循环重新开始时,插入循环体代码,清空上次循环中分配的所有对象;在最终离开循环时,插入出口代码,释放区域栈帧. 图 5 描述了循环分析算法,算法中一些符号的含义如表 1 所示.

```
//根据字节码生成控制流图
FG := GenerateFG( $m_b$ )
//创建 DFS 树  $T$  和节点的深度信息  $dfn[n]$ 
search(FG,  $T$ ,  $dfn$ );
//构建回边集合
for each  $e \in E$  do
    if  $dfn[e_h] > dfn[e_t]$  then //  $e$  是一条回边
        insert( $e$ ,  $E_b$ ); //将  $e$  加入回边集合  $E_b$ 
//根据回边找到循环体,并插入相关代码
for each  $e \in E_b$  do
    //通过回边  $e$  找到循环中的所有节点
    loop := findloop( $e$ , FG);
    for each  $e \in FG$  do
        //  $e$  为由循环外指向入口点的边
        if  $e_h \notin loop$  and  $e_t = loop$  入口点 then
            insert(Enter loop); //插入入口代码
        //  $e$  为由循环内指向入口点的边
        if  $e_h \in loop$  and  $e_t = loop$  入口点 then
            insert(Iterate loop); //插入循环体代码
        //  $e$  由循环内指向循环外的边
        if  $e_h \in loop$  and  $e_t \notin loop$  then
            insert(Leave loop); //插入出口代码
```

图 5 循环分析算法

表 1 循环分析算法的符号

m_b	方法 m 的字节码
$FG=\langle N,E\rangle$	流图, N 为指令节点的集合, E 为边的集合
$T=\langle N,E\rangle$	DFS 生成树, N 为节点的集合, E 为边的集合
$dfn[i]$	DFS 树中节点 i 的深度信息
$e=\langle e_h,e_t\rangle$	图或者树中的边, e_h 为头节点, e_t 为尾节点
E_b	回边集合
$loop$	回边所确定的循环体

图 6(a)是一段循环代码和该代码的程序流图,每条指令为一个节点,指令与其后继指令之间通过一条有向边连接,所有的分支、条件转移等也都用有向边标明. 图 6(b)中,阴影部分是循环分析算法在流图中发现的循环体. 该循环的入口点是语句“ $i<len$;”. 黑色的方框是对 3 类有向边插入的相关代码. 在图 6(b)中同时显示了我们对于 Java 异常的处理. 当异常发生时,处理方法与循环出口点相同,释放区域栈帧.


```
for(i = 0; i < len; i++){
    Thing t = new Thing(i);
    t.dostuff();
}
```

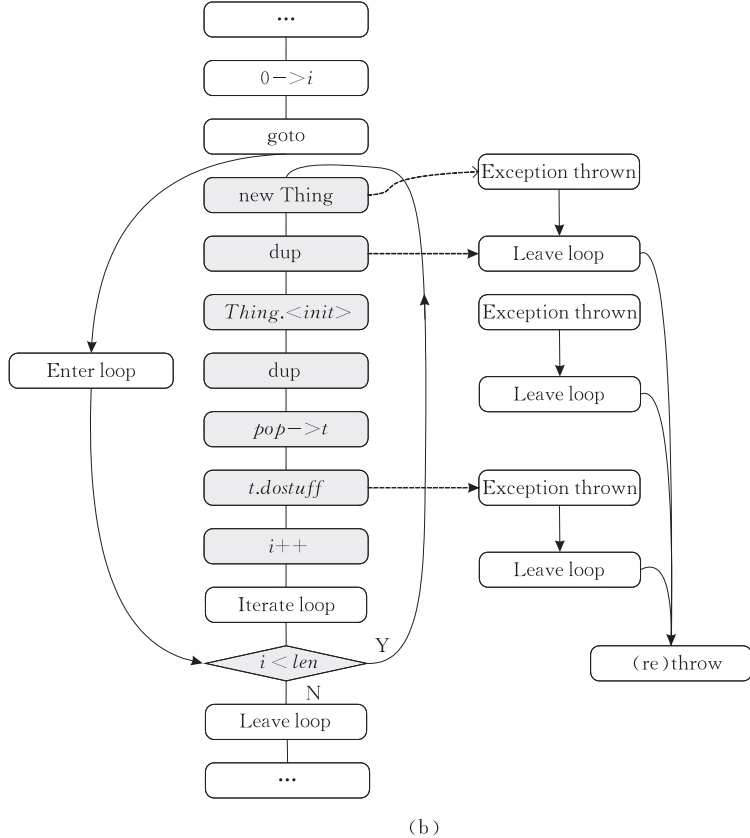
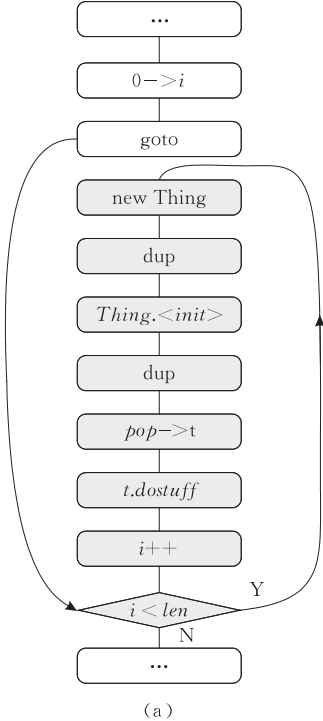


图 6 循环分析前后对比

5 性能评测

本文在 Intel 的 ORP 平台 (Open Runtime Platform)^①上,实现了基于逃逸分析的栈式分配优化.系统运行的硬件平台是双处理器的 Dual Pentium4 Xeon 2400MHz,内存是 512MB.

为了便于比较,我们选择了 SPECjvm98 中被普遍使用的 4 个例子,分别为 jess、db、javac 以及 jack.表 2 为测试用例的基本情况.

表 2 4 个测试用例的基本情况统计

测试用例	内存分配总量/B	分配的对象数量	方法调用的数量
Jess	3969853	101592	5867568
db	3783533	113273	1216964
javac	7602866	214734	2982147
jack	20003402	694988	7119896

5.1 栈式分配对象的比例

本文算法的运行结果为:jess、db、javac 和 jack 的栈式分配比率分别为 8.3%、8.4%、19%和 25%,表 3 是与 Blanchet^[4]、Gay^[5]以及 Qian^[8]进行的比较.

表 3 栈式分配比率结果比较

测试用例	本文/%	Blanchet/%	Gay/%	Qian/%
jess	8.3	21	5.3	6.50
db	8.4			0.74
javac	19.0	13		8.80
jack	25.0	20		22.87

从表 3 中可以看出本文基于循环的栈式分配方法显示出一定的优势. Blanchet 在循环处理中采用生存期分析的方法,试图重用循环中栈式分配的对象,但是它无法处理循环中的变长对象. 4 个测试用例中,除 jess 以外,我们的结果在 javac 和 jack 中分别比 Blanchet 方法的栈式分配比率提高了 8%以及 7%. 另外,需要指出的是,Blanchet 方法属于全数据流分析,无法单独对某个方法或者对象进行分析,也无法在处理动态加载的对象. 与 Qian 方法相比,我们的平均栈式分配比率超出 5.43%.

5.2 栈空间

在提高栈式分配对象比例的同时,我们把栈空间的大小也控制在了可接受的范围之内.

在非保守情况下,Blanchet 方法在 javac 中最多

① <http://orp.sourceforge.net/>

可以栈式分配 43% 的对象,但是付出的代价是消耗的栈空间比采用保守方法时增加了 10 倍,大大超出了一级 Cache 的容量,这将直接导致 Cache 命中率下降,使栈式分配变得得不偿失;而本文栈空间内存占用量的峰值如表 4 所示,最坏情况下栈空间峰值占用量为 66KB,大多数时间内占用的栈空间小于 CPU 的一级缓存大小,不会明显降低 Cache 命中率.

表 4 区域栈帧内存占用量的峰值

测试用例	分配比率/%	区域栈的峰值/KB
jess	8.3	15
db	8.4	16
javac	19.0	27
jack	25.0	66

5.3 运行时间

针对 4 个测试用例,分别运行了 20 次,为了避免虚拟机初始化的影响,我们取后 15 次的平均值(见表 5). 与原来的 ORP 系统相比,除了 javac 的运行时间有所增加之外,其它测试用例都有不同程度的下降. 本文对循环内对象的栈式分配,主要增加了编译的时间开销. 由于插入循环入口、出口等代码,对运行时的性能也有一定影响. 栈式分配优化对性能的改善小于它带来的开销时,程序运行的性能会有所下降,例如 javac 程序.

表 5 测试用例运行时间比较

测试用例	本文/s	原 ORP/s
jess	3.73	3.82
db	8.66	8.72
javac	8.17	8.14
jack	5.48	5.54

6 相关工作

Ruggieri^[9] 针对一个非面向对象,但具有垃圾回收机制的语言进行了讨论,提出了一个逃逸分析的基本框架,首先提出了对堆中内存进行栈式分配的思想. Blanchet^[4] 提出了基于类型系统的静态逃逸分析方法,并用于栈式分配. Blanchet 通过生存期分析方法确定对象的状态,来解决循环情况下空间复杂度较高的问题. Blanchet 分两种情况测试了他的算法,第 1 种保守情况,除非循环中分配的对象可以通过生存期分析方法被重用,否则不分配所有循环中的对象,这种情况的空间大小基本处于可控范围;第 2 种非保守情况下,不论生存期分析是否奏

效,都在栈中分配所有非逃逸的对象,这种情况下,空间大小就很可能失控. 然而,即使在较保守的第 1 种情况下,空间大小还是会在递归调用中变得不可控. Gay^[5] 在 Marmot 中实现了一个相对保守但简易高效的逃逸分析算法,该算法的时间复杂度线性于指令条数. 在循环中栈式分配对象的处理方面, Gay 通过循环分析判断被分配的对象之间是否可以重用内存. 如果对象不重用内存,就不进行栈式分配. 对象数组以及地址被其它对象引用的对象,也不会被分配在栈中. 另外,对于可以栈式分配的对象, Gay 把对象直接拆分为一组局部变量存入栈帧,降低了创建对象的开销. Choi^[10] 在他的论文中描述了一个基于连接图(connection graph)的逃逸分析算法. 该方法更加精确,对于地址被其它对象引用的情况,不会像 Gay 方法一样,简单将其标识为逃逸,而是通过连接图进行更加细致的分析. 另外, Choi 通过对 java 类型系统的分析,将对象中的引用属性和其它属性进行了区分. 但 Choi 的实现中,没有考虑空间复杂度的问题,对所有可栈式分配的对象都进行了分配. 同时 Choi 也将逃逸分析的结果用于指导冗余同步消除. Baker^[11] 采用了比较乐观的栈式分配策略,他的基本思想是,不论对象是否可以栈式分配,都分配到栈中;当发现已经分配在栈中的对象无法被栈式分配的时候,才把它重新放置到堆中来管理. Baker 设定了一种向下增长的栈结构,把堆结构放置在栈的上方,提出了读闸(read barrier)的概念. 但读闸为程序带来了较大的负担, Berger^[12] 指出,实现读闸可以使程序的开销增加了 50%. Qian^[8] 基本思想与 Baker^[11] 类似,首先将所有的对象都分配到栈上,然后通过写闸(write barrier)捕获已逃逸的对象. Qian 的方法不需要进行逃逸分析、程序员标注或者特别的类型系统,因此易于与已有的 JVM 集成,但是分析精度损失比较严重.

7 结束语

本文研究了 Java 逃逸分析技术在栈式分配方面的应用,引入了对象栈、区域栈帧等的概念,将方法调用栈和对象分配栈区分开来,把循环作为区域栈帧建立和释放的基本单位. 在逃逸分析的基础上,我们通过对循环的分析,在 ORP 平台上实现了栈式分配优化.

测试结果显示,本文的方法平均栈式分配了 15.18% 的内存对象,较前人方法有一定提高;同时,

将运行过程中栈的峰值内存控制在可接受的范围内.

参 考 文 献

[1] Tim L, Frank Y. The Java Virtual Machine Specification. Second Edition. Berkeley: Addison Wesley, 1999

[2] Appel A W. Garbage collection can be faster than stack allocation. Information Processing Letters, 1987, 25(4): 275-279

[3] Blanchet B. Escape analysis for java™: Theory and practice. ACM Transactions on Programming Languages and System, 2003, 25(6): 713-775

[4] Blanchet B. Escape analysis for object oriented languages. application to java//Proceedings of the 14th Annual Conference on Object-Oriented Programming Systems, Languages and Applications. Denver, CO, 1999: 20-34

[5] Gay D, Steensgaard B. Fast escape analysis and stack allocation for object-based programs//Proceedings of the International Conference on Compiler Construction (CC'2000). Berlin, Germany, 2000: 82-93

[6] Wang L, Sun X. Escape analysis for synchronization remov-

al//Proceedings of the 21st Annual ACM Symposium on Applied Computing. Dijon, France, 2006: 1419-1423

[7] Nielson F, Nielson H R, Hankin C. Principles of Program Analysis. New York: Springer-Verlag, 1999

[8] Qian F, Hendren L. An adaptive, region-based allocator for Java//Proceedings of the International Symposium on Memory Management (ISMM). Berlin, Germany, 2002: 127-138

[9] Ruggieri C, Murtagh T P. Lifetime analysis of dynamically allocated objects//Proceedings of the 15th ACM Symposium on Principles of Programming Languages (POPL'88). San Diego, California, USA, 1988: 285-293

[10] Choi J D, Gupta M, Sreedhar V C, Midkiff S. Escape analysis for java//Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications. Denver, CO, 1999: 1-19

[11] Baker H G. CONS should not CONS its arguments, or a lazy alloc is a smart alloc. ACM SIGPLAN Notices, 1992, 27(3): 24-34

[12] Berger E, Zorn B, McKinley K. Reconsidering custom memory allocation//Proceedings of the 2002 ACM SIGPLAN Conference on Object-Oriented Programming: Systems, Languages, and Applications. Seattle, Washington, 2002: 1-12



WANG Lei, born in 1969, Ph. D. , associate professor. His research interests include JVM, compiler and operating system.

XU Xing, born in 1982, master. His research interests include JVM and compiler.

Background

This work is partially supported by Beijing Natural Science Foundation, Intel China Research Center and Scientific Research Foundation for Returned Scholars. The main concerned problem in this paper is the stack allocation of object in loop. In JVM, the memory needed for a Java program is allocated in the heap or stack. Compared with heap allocation, stack allocation has three important advantages which include: (1) it reduces some synchronization operations; (2) it lightens some burden of the garbage collector; (3) since the stack space is a hot spot of Cache, stack operations are much faster than ones in heap. However, when objects are allocated in loop, the conflict between the ratio of stack allocation of objects and size of the stack space is irreconcilable. Blanchet and Gray realize the allocation of fixed-length ob-

jects that can be reused in loop, but these methods do not solve the problem of stack allocation of variable objects.

In this paper, the allocation policy based on loop as a basic unit is presented, and the concepts of object stack and stack region are proposed. The authors implement the stack allocation base on escape analysis through the loop analysis. The result of SPECjvm98 shows that 8.3% to 25% (with an average of 15.18%) of all objects could be stack allocated in the new algorithm with the controlled size of stack. Before this paper's work, the group implemented a flow-insensitive, inter-procedural, and context-sensitive escape analysis, which has been applied to remove unnecessary synchronization in Java. For the authors' benchmarks, 7.3% to 99.9% of synchronization operations are eliminated.