ISSN 1000-9825, CODEN RUXUEW
Journal of Software, Vol.19, No.3, March 2008, pp.522−532
DOI: 10.3724/SP.J.1001.2008.00522

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

在开放世界中实现逃逸分析*

史晓华[1+]，吴甘沙[2]，金茂忠[1]，LUEH Guei-Yuan[2]，刘 超[1]，王 雷[1]

[1](北京航空航天大学 软件工程研究所,北京 100083)

[2](Programming System Laboratory, Microprocessor Technology Labs, Intel, Santa Clara, CA, USA)

Escape Analysis: Embrace the Open World

SHI Xiao-Hua[1+], WU Gan-Sha[2], JIN Mao-Zhong[1], LUEH Guei-Yuan[2], LIU Chao[1], WANG Lei[1]

[1](Software Engineering Institute, Beijing University of Aeronautics & Astronautics, Beijing 100083, China)

[2](Programming System Laboratory, Microprocessor Technology Labs, Intel, Santa Clara, CA, USA)

+ Corresponding author: Phn: +86-10-82338487, E-mail: xhshi@buaa.edu.cn, http://www.buaa.edu.cn

Shi XH, Wu GS, Jin MZ, Lueh GY, Liu C, Wang L. Escape analysis: Embrace the open world. *Journal of Software*, 2008,19(3):522−532. http://www.jos.org.cn/1000-9825/19/522.htm

Abstract: Prior implementations of escape analysis (EA) make a closed-world/whole-program assumption: All methods that can possibly be executed during the program execution are known statically. The Java programming language has several language features, e.g., dynamic class loading, native invocation and reflection, which break the closed-world assumption. These open-world features raise a major concern on the practicability and applicability of the prior approaches to Java applications that use or rely on the features. This paper proposes and evaluates a new escape analysis framework which handles the Java open-world features. The framework also provides a mechanism that controls the analysis complexity to reduce the runtime overhead with acceptable precision. The result shows that the EA framework, which is implemented in Intel's Open Runtime Platform on X86, eliminates 70% and 94% of dynamic synchronized operations and improves the runtime performance 15.77% and 31.28%, for SPECjbb2000 and db of SPECjvm98 respectively.

Key words: escape analysis; just-in-time compiler; Java virtual machine

摘 要: 逃逸分析(escape analysis)是一种可以有效减少 Java 程序中同步负载和内存堆分配压力的跨函数全局数据流分析算法.此前绝大多数逃逸分析的实现都基于一个所谓"封闭世界(closed world)"的前提:所有可能被执行的方法在做逃逸分析前都已经得知,并且,程序的实际运行不会改变它们之间的调用关系.但当真实的 Java 程序运行时,这样的假设并不成立.Java 程序拥有的许多特性,例如动态类加载、调用本地函数以及反射程序调用等等,都将打破所谓"封闭世界"的约定.这样的真实运行环境被称为"开放世界".在开放世界中,实现逃逸分析将面临许多重要的问题,例如,能否正确、全面地捕捉动态载入的类和方法,并分析它们与原有程序的关系;逃逸分析算法的复杂性是否能够得以控制,以保证即时编译器的重新分析时间不会过长,等等.提出一个新的逃逸分析架构,它可以有效地处理上述开放世界所面临的问题.该分析架构将增量分析 Java 程序,动态捕获新载入和调用的类及方法,同时,在复杂性和精度之间进行权衡,正确、有效地降低程序的运行负载.该分析架构已经在 Intel 的开放式 Java 虚拟机系统 ORP

* Supported by the National Natural Science Foundation of China under Grant No.60573084 (国家自然科学基金)
Received 2006-10-29; Accepted 2007-03-06

中实现,经过实际测试,可以有效地消除一些主要基准测试程序,如 SPECjbb2000 和 SPECjvm98 的 db 中 70%~94%的同步操作,大幅度地提高 15%~31%的程序的运行速度.

关键词: 逃逸分析;即时编译器;Java 虚拟机

中图法分类号: TP301 **文献标识码:** A

Java 提供了语言级的并发程序设计手段,例如同步的方法和程序块(synchronized method and block)[1],编程者可以方便地使用这些手段对线程和共享数据进行有效控制.在现代多处理器系统中,如对称多处理器 SMP 架构上,此类同步运算的最终实现通常依靠上锁和解锁指令集来完成.以 X86 Xeon 体系结构为例,对某一内存区域的上锁操作即便不会造成多个线程对同一信号量的争夺,也会导致相关的缓存区域被刷新;两者都会造成程序运行时较大的开销.逃逸分析是针对这一问题而提出的一种跨函数(inter-procedure)全局数据流分析算法.它通过分析每个对象及其域可能被访问的范围,判断出:1) 一个对象是否仅被创建它的线程访问,从而可以合法地忽略作用在这个对象之上的同步操作;2) 一个对象是否仅被创建它的方法以及被该方法调用的子方法访问,从而可以合法地将此对象分配在程序运行栈,而不是内存堆上.这样可以有效地减少内存收集模块(garbage collector)的负载,提高程序的运行效率.

此前,许多与逃逸分析有关的工作[2−6]以及其他类似的跨函数数据流分析优化[7−9]对被分析的 Java 程序都有一个前提约束条件,那就是被分析程序需要符合“封闭世界”的要求,即所有相关的程序信息在做逃逸分析之前都可以获知,并且这些信息在 Java 程序运行时仍是全面和正确的.同时,由于此前的逃逸分析都是在一个静态编译器中实现,不需要对 Java 程序进行动态分析,因此,对算法复杂性也不存在特殊要求.但是,真实的 Java 程序在运行时,通常都会动态载入并调用一些类和方法以及本地函数,而一旦这样的情况发生,此前关于封闭世界的假设前提便失去意义,逃逸分析结果的正确性也就无法保证,其后果将导致整个程序运行出错.

近年来,已有一些相关的研究工作开始讨论开放世界(与封闭世界相反)的问题.Vivien 等人[10]提出了一种为了将对象分配在堆栈上而对程序进行增量分析的算法.该算法假设所有被分析的 Java 方法都与调用它们的方法无关,并且将调用者与被调用者之间的关系进行最大保守约定.根据我们的实践经验,这样的约定过于保守,将导致绝大部分的优化无法进行.Bogda 等人[11]讨论了几种在程序运行时进行逃逸分析的可能的解决方案,但并没有讨论如何有效、全面地获取打破封闭世界的各种可能性,以及传统逃逸分析算法如果需要支持动态重新分析,需要如何进行改进.Sreedhar[5]与 Arnald[12]提出的方案是,一旦发生破坏封闭世界约定的情况,所有的优化将被自动取消,从而保证程序运行的正确性.但是,根据我们的实践经验,大部分程序在初始阶段便会动态载入一些 Java 类并执行其中的方法,按照上述约定,几乎无法在实际程序中得到逃逸分析优化的好处.Pechtchanski 等人[13]试图采用一种较为激进的手段,在程序开始时假设所有的对象均未逃逸出创建它们的线程,同时消除所有的同步操作;随着程序的运行,根据具体的运行情况,动态地将被认定为逃逸对象的同步操作加以恢复.这样的解决手段会导致即时编译器高频率地重新分析所有的程序,不断对同步操作进行调整,同时带来极大的编译负载.但是,文中并没有讨论这样做之后,Java 程序运行时会增加多少时间和空间的开销.

无论是封闭世界还是开放世界,作为一种跨函数全局数据流分析算法,逃逸分析都将面临时间和空间复杂性的挑战.上下文无关[6]的解决方案可以极大地减少算法所需的编译时间和空间,但是其付出的代价是分析精度的急剧下降.上下文相关[3,4]的解决方案可以较好地保证分析精度,但其所需编译时间和空间都远远高于上下文无关算法.在 Java 虚拟机中,当发生诸如动态类载入等情况,需要对程序进行重新分析时,即时编译器必须在复杂性和分析精度之间作出适当的平衡.

1 开放世界的特点及引发的问题

Java 程序开始运行后,发生以下 3 种情况之一,都会破坏封闭世界的假设前提.

1.1 动态类载入

动态类载入作为一种重要的 Java 语言特性,可以通过延迟的类加载来提高系统的安装效率,也可以通过类

的动态重新载入降低软件和设计的复杂性.但是对于逃逸分析而言,动态载入的类在程序运行前未知,新的类和方法的引入将会破坏此前算法所作的分析结论,从而可能使得此前的优化不再正确,并导致整个程序运行出错.但是,如果简单地把所有可能被动态载入的类的相关对象都认定为将逃逸出当前线程,尽管相对简化了问题,却会使逃逸分析的有效性和精度大为降低.

1.2 调用本地函数

所谓本地函数,是指在Java类库和程序中,部分方法是采用本地语言,如C,C++或者汇编语言等编写的.这样做的目的或者是为了提高 Java 程序的运行效率,或者是为了实现一些与系统和硬件有关的接口.在这些本地函数中,可以修改某些对象的域,甚至可以调用其他 Java 方法.如果逃逸分析不能全面了解本地函数的内容,作出的优化判断很可能是不正确的.但是,Java 虚拟机和及时编译器无法保证可以在程序运行前得到所有本地函数的相关信息.

1.3 反射(reflection)

反射机制可以被应用程序用来动态调用一些编译时不可预见的类和方法.如何动态捕获这样的动态调用关系,是逃逸分析能否在真实的 Java 运行环境中得以实现和应用的关键之一.

2 逃逸分析架构和算法

2.1 架构总述

图1展示了我们在ORP(open runtime platform)[14]中实现的逃逸分析架构.其中主要包括基于字节码的逃逸分析算法模块(BC-based analysis)、封闭世界的动态违例检测模块(violation detection)、重新分析模块(re-analysis)以及优化补偿模块(deopt/comp)等.Java 虚拟机从字节码而不是中间表达式(IR)中提取逃逸分析所需的各种信息.其优点在于,逃逸分析算法的实现可以独立于即时编译器,并且可以排除其他优化造成的干扰.

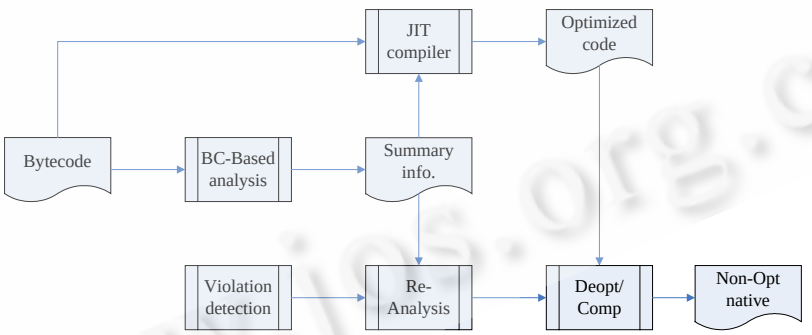


Fig.1 Overview of escape analysis framework

图 1 逃逸分析架构

本文的重点将集中讨论逃逸分析算法模块、封闭世界的动态违例检测模块和重新分析模块等,其他模块的原理和工作情况将另文阐述.

2.2 建立方法摘要

下文将结合表 1 解释逃逸分析算法的符号和公式定义.逃逸分析算法的核心数据结构是一个调用关系图 $G=\langle N,E\rangle$,其中,N 为节点集,E 为有向边集.每个节点代表一种方法.有向边 $m_i \rightarrow m_j$ 表示方法 m_i 可能调用 m_j.$E(G)$ 表示 G 中的所有边,$N(G)$表示 G 中所有节点.我们使用动态边或方法(dynamic edge or method,简称 DEM)来表示封闭世界被破坏时,新引入的节点或者有向边.

Domains and variables

A	Alias sets
C	Contexts
$v,v_i \in V$	Local variables
$v_{ret} \in V$	Return variable
$v_{excp} \in V$	Exception variable
$ref,arrayref \in V$	Field variable
$V_n \subset V$	Variables created by new-family ops
$n \in V_n$	Variable created by new-family ops
$a_0,\ldots,a_n \in A$	Alias sets for callee's actual args
$a_r \in A$	Alias set for callee's return value
$p \in M$	Statically resolved callee
$m \in M$	Methods
$\$bc \in C$	Bytecode index of call or allocation site

States

AS: $V \rightarrow A$	Lookup alias set
$newVar$: $C \rightarrow V$	Create new variable
T: $M \times A \rightarrow M$	Resolve callee's targets

Operations

$\eta(a)$:	$\{n \mid n \in V_n \wedge AS(n) \in reachable(a)\}$
$set(S,st)$	Set escape status flag: $S=(S<st)?\ st: S$

Table 1 Rules of building method summaries

表 1 建立逃逸分析中方法摘要的规则

Bytecode	Rules
Aload *i*	*stack.push*(*AS*(v_i))
Astore *i*	*unify*(*AS*(v_i),*stack.pop*())
Aaload	*stack.push*(*AS*(*arrayref*).*fieldMap*(**$ELT**))
Aastore	*unify*(*AS*(*arrayref*).*fieldMap*(**$ELT**),*stack.pop*())
Areturn	*unify*(*AS*(v_{ret}),*stack.pop*()) $\forall v \in \eta(v_{ret})$: *set*(*AS*(*v*).$S_{up}$,*thread*)
Getstatic	*stack.push*(*AS*(**$GLOBAL**))
Putstatic	*unify*(*AS*(**$GLOBAL**),*stack.pop*())
Getfield	*stack.push*(*AS*(*ref*).*fieldMap*(*fld idx*))
Putfield	*unify*(*AS*(*ref*).*fieldMap*(*fld idx*),*stack.pop*())
Invokevirtual Invokeinterface Invokespecial Invokestatic	*p*=*method_handle*/**p*($v_0,\ldots,v_n$)*/ a_r=(*p.return_type*()==*ref*)? *AS*(*newVar*(**$bc**)): nil a_n=*stack.pop*()...a_0=*stack.pop*() $\forall m \in T(p,a_0)$: *callSites.add*(**$bc**,*m*,⟨$a_0,\ldots,a_n,a_r$,*AS*($v_{excp}$)⟩) if (*p.return_type*()!=*void*) *stack.push*(*AS*(a_r))
New Newarray Anewarray Multianewarray	*n*=*newVar*(**$bc**) *stack.push*(*AS*(*n*)) if (*n.instanceof*(*Thread*)) *set*(*AS*(*n*).S_{up},*global*)
Athrow	*unify*(*AS*(v_{excp}),*stack.pop*()) *set*(*AS*(v_{excp}).S_{up},*thread*)
Others	No action

每种方法的分析摘要 MS 采用一个四元组⟨*aliasSet*,*aliasContext*,*callSites*,*statusArray*⟩表示.其中:

- *aliasSet* 是方法 *m* 中所有等价变量组的集合.方法中的任何变量都将属于 *V* 中某一个等价变量组.每一组等价变量由一个四元组⟨*V*,*fieldMap*,S_{up},S_{down}⟩表示.*V* 是等价变量集合,其中包含的所有变量都属于同一等价类(equivalent class).关于等价变量,一个典型的例子是,如果出现了这样的语句 *a*=*b*,变量 *a* 和变量 *b* 就属于同一个等价类.定义一个特殊变量$*GLOBAL*,代表程序中所有的全局量和静态变量,它们共同的特点是可以被任何线程访问.如果 *V* 中出现了一个$*GLOBAL*,将认为其中所有变量都逃逸出创建线程.*fieldMap* 将所有可能的对象域与其他 *aliasSet* 关联起来.定义一个特殊的域名$*ELT*,对应于对象数组的成员.表 1 中 *unify* 的操作类似于文献[13]中的定义,该操作用来合并两个不同的 *aliasSet*.每个

aliasSet 具有两个逃逸状态位 S_{up} 和 S_{down}.当跨函数分析进行到不同的阶段,例如自底向上阶段时,将使用 S_{up} 状态位进行逃逸状态的分析和传导;当进行到自顶向下的阶段时,将使用 S_{down} 状态位进行分析和传导.在两个阶段转换时,S_{down} 将被赋予 S_{up} 的值.这与此前典型的逃逸分析算法[4]不同,是专门为了支持动态重新分析,避免重新分析时重复此前的工作而设计的;

S_{up} 和 S_{down} 都具有 3 个不同的逃逸状态,分别是 *stack*,*thread* 和 *global*.定义它们之间的大小关系为 *stack*<*thread*<*global*.任何 *aliasSet* 的逃逸分析状态,都只能从小到大进行变化;

- *aliasContext* 的定义为⟨f_0,…,f_n⟩,*ret*,*excp*⟩,它表示方法 *m* 的上下文.其中,f_0~f_n 表示 *m* 的参数,*ret* 表示返回变量,*excp* 表示异常对象.
- *callSites* 表示方法 *m* 中所有被调用的子方法的集合,它由一个三元组⟨*callSite*,*callee*,*siteContext*⟩表示.其中,*callSite* 是该子方法被调用的字节码下标,*callee* 是该子方法的句柄,*siteContext* 属于 *aliasContext*.对于那些 virtual 或 interface 方法调用,由于存在多个可能被调用的子方法,所有这些子方法都需要建立一个属于 *callSites* 的单元.
- *statusArray* 表示完成逃逸分析后,对应于每个有效的字节码下标建立的逃逸分析状态数组.逃逸分析完成后,即时编译器将根据该数组的内容,决定是否将某一方法的同步操作取消,或者将某个对象分配在函数堆栈,而不是内存堆上.一般而言,所有被调用点的逃逸状态小于 *global* 的同步操作都可以取消同步,所有被调用点的逃逸状态为 *stack* 的 *new* 操作,例如 *new*,*anewarray*,*newarray* 或者 *multianewarray* 都可以在堆栈上分配空间.

表 1 中的符号 *AS*(*a*)表示变量 *a* 所属的等价变量组.*reachable*(*a*)表示通过 *AS*(*a*)中的等价变量的 *fieldMap* 关系,可以访问到的所有 *aliasSet*,包括 *AS*(*a*)本身在内.*mContext*(*m*)表示方法 *m* 的 *aliasContext*.

当变量 *a* 为形参或者返回变量,并且 *reachable*(*a*)包含了上述 *new* 操作得到的对象时,*AS*(*a*)的 S_{up} 将被赋予 *thread* 状态.其他情况下,S_{up} 的缺省值为 *stack*.

2.3 跨函数分析

调用关系图 *G* 在进行跨函数分析前便构建完毕.在 *main*,*clinit* 和线程的 *run* 方法之上,是一个伪根节点,并且 *G* 被分割成不同的强连通分支节点(strongly connected components,简称 SCC).图 2 中的跨函数分析包括两个阶段:自底向上阶段和自顶向下阶段.在前一个阶段中,S_{up} 被用来记录从被调用函数传导上来的逃逸状态;在后一个阶段,S_{down} 被用来记录从调用函数传导下去的逃逸状态.在自底向上阶段,每当一种方法收集到所有被它调用的方法传导上来的逃逸状态和对象域关系时,将进行一次方法内(intra-procedure)分析;在自顶向下阶段,每当一种方法收集到所有可能调用它的方法传导下来的逃逸状态时,同样也需要进行一次方法内分析.不同的是,前者使用 S_{up} 进行状态归纳,后者使用 S_{down} 进行状态归纳.*G* 中所有节点完成自底向上的分析后,将转入自顶向下分析,此时,所有等价变量集合的 S_{up} 值首先将被赋给 S_{down}.需要注意的是,所有逃逸状态的改变都将使用表 1 中的 *Set*(*S*,*st*)操作,即所有逃逸状态只能单向改变,不可逆转.

2.3.1 自底向上传播阶段

一个被调用方法 *m* 只能通过参数传递、返回值以及异常对象来影响它的调用者.该阶段中,系统将根据实参和形参、异常变量以及返回值之间的对应关系,一次性地把 $F=\{reachable(v)|v\in mContext(m)\}$中每个 *aliasSet* 的状态和 *fieldMap* 传导(propogate)给 *m* 所有的调用者.

这里定义传导操作 propogate(as_1,as_2),其中,as_1 和 as_2 为 *aliasSet* 中的等价变量组.这样的传导需要完成两件主要工作:1)递归地将 as_2 中所有 *fieldMap* 关系传导给 as_1;2) 在自底向上传播阶段,它将把 as_2 的 S_{up} 状态传导给 as_1;在自顶向下传播阶段,它将把 as_2 的 S_{down} 状态传导给 as_1.

对于一个强连通分支节点中的所有方法,逃逸算法将循环传导所有的参数状态,直到这些状态不再改变.

2.3.2 自顶向下传播阶段

该阶段开始时,所有 *aliasSet* 中,S_{down} 的值将等于 S_{up},然后自顶向下地通过方法调用关系、实参和形参、异常变量以及返回值之间的对应关系,将逃逸状态向下传导.与自底向上传导类似,对同属于一个 *SCC* 节点的方

法,算法将循环传导所有的参数状态,直到这些状态不再改变.

```
intraprocedure_analysis(method,esf) {
    do {  changed=false;
        for (each as in AS(mContext(method)).aliasSet){
            for (each field in as.fieldMap){
                if (esf(field)<esf(as)){changed=true;esf(field)=esf(as);}}}
    } while (changed);
 }
propagate_from_callee(m,cs) {/*in the same SCC node: use context-insensitive solution*/
    if (in same scc(cs.callee,m)){/*merge two method summaries*/
        merge(m,cs.callee);
    } else {/*or else: use context-sensitive solution*/
        for (each ref in mContext(cs.callee)){/*Recursively propagate S_up & fieldMap*/
            actual_arg=corresponding actual arg in cs.siteContext;
            propagate(AS(actual_arg),AS(ref));
        }}
}
propagate_from_caller(cs) {
    for (each ref in mContext(cs.callee)) {/*Recursively propagate S_down only*/
        actual_arg=corresponding actual argument in cs.siteContext;
        propagate(AS(ref),AS(actual_arg));}
 }
interprocedure_analysis() {
    for (each method in bottom-up order){/*Step 1*/
        for (each cs in AS(mContext(method)).callSites)/*Propagate S_up and fieldMap from callee to caller*/
            propagate_from_callee(method,cs.callee);
        intraprocedure_analysis(method,S_up);}
    for (each method in top-down order){/*Step2*/
        for (each as in AS(mContext(method)).aliasSet)/*Initialize S_down*/set(S_down(as),S_up(as));
        intraprocedure_analysis(method,S_down);
        for (each cs in AS(mContext(method)).callSites)      propagate_from_caller(cs);}
 }
```

Fig.2 Escape analysis algorithm

图 2 逃逸分析算法

3 封闭世界的 DEM 检测及重新分析

根据前文有关DEM的定义可知,每当它出现便意味着所谓封闭世界的假设被打破.本文介绍的逃逸分析架构将提供动态捕获 DEM 发生的机制.对于 Java 程序而言,下面 3 种情况都有可能产生 DEM.

3.1 动态类载入导致的DEM

当动态类载入发生时,两类方法可能导致 DEM:1) 新载入的 *init* 和 *clinit* 方法;2) 新载入的虚方法.对于新载入类的初始化方法 *clinit*,将在调用关系图中增加一个新的 *clinit* 节点,以及 *root*→*clinit* 的边.Java 虚拟机将在使用该类中的方法或者变量前调用 *clinit*.逃逸分析需要重新分析该 *clinit* 以及所有此方法可能直接或间接调用的其他方法.

本地函数 *Class*::*newInstance* 和 *Construtor*::*newInstance* 将被用来创建新载入类的对象,而该类对象在创建过程中,还将调用对应的 *init* 初始化方法.由于上述两种 *newInstance* 本地函数在 Java 程序运行前无法得知哪种具体的 *init* 方法将被调用,因此,我们必须在本地函数调用 Java 方法的接口代码中增加一种动态捕捉的机制,来获取相关信息,如图 3 所示.

图 3 中的动态捕捉模块将采用 Hash 表记录这样的三元组信息⟨*caller*,*callSite*,*clssHandle*⟩,其中,*caller* 指的是 *newInstance* 的调用者,*callSite* 指的是 *newInstance* 被调用的位置,*clssHandle* 指的是被调用的 *init* 方法所属的类句柄.该 Hash 表在整个 Java 程序运行过程中都将有效,如果表中已经存在同样的三元组,表示不会引起新的

DEM 发生,否则将引发逃逸分析的重新运行.

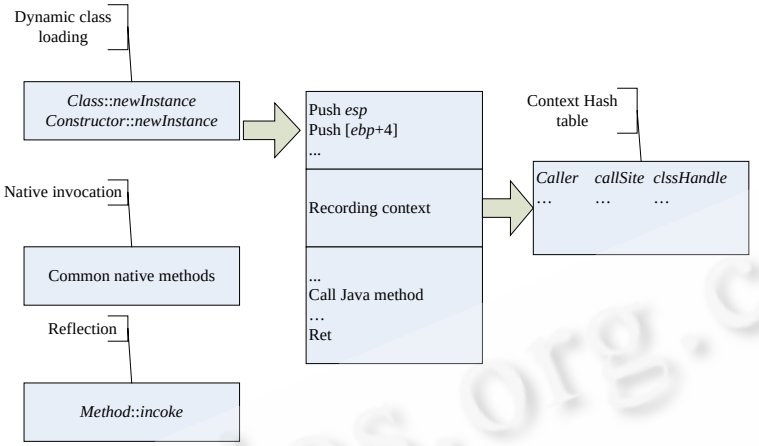


Fig.3 Dynamically catching DEM

图 3 动态捕获 DEM 的发生

对于所有新载入的虚方法,逃逸分析架构将在 G 中增加新的节点,并将所有可能调用该虚方法的节点指向该新节点.这是一种保守但是保证正确的处理手段.例如,当虚方法 *NewClass*::*foo* 被载入后,该方法如果重载了基类的 *foo* 方法,我们将在 *foo* 出现的所有 *callSites* 中增加 *NewClass*::*foo* 的调用信息,以保证逃逸分析的正确性.

3.2 调用本地函数导致的DEM

即便保守地将所有传递到本地函数的参数的逃逸状态设定为 *global*,仍然无法保证分析的正确性,这是因为本地函数仍然可以通过某些接口调用 Java 方法,将不可预知的逃逸状态传递出去.因此,我们需要在本地函数调用 Java 方法的接口中增加图 3 中类似的动态捕捉机制,一旦发生未经处理的从本地函数到 Java 方法的调用,便认为新的 DEM 发生,然后激活逃逸分析的重新分析模块,保证算法的正确性.

3.3 反射导致的DEM

反射函数 *Class*::*getConstructor*,*Class*::*getMethod* 以及 *Constructor*::*newInstance* 和 *Method*::*invoke* 组合在一起,可以在 Java 程序中调用反射对象的任意不可预知的方法.逃逸分析架构无法将所有可能被调用的方法都连接到调用关系图 G 中,这样会极大地增加分析的复杂性,并降低分析精度.

由于所有反射方法的调用都将通过 *Method*::*invoke* 来完成,根据图 3 中的处理手段,同样可以在 *invoke* 本地函数中增加类似机制,捕捉未经处理的 DEM.此外,反射函数 *Class*::*getField* 和 *Field*::*set* 可以用来访问修改反射对象的某个域,该域的名称和内容可能在 Java 程序运行时才能获知,因此,我们可以保守地将所有域对象的属性 S_{up} 都缺省设置为 *global*,从而保证算法的正确性.

3.4 运行过程中进行的重新分析

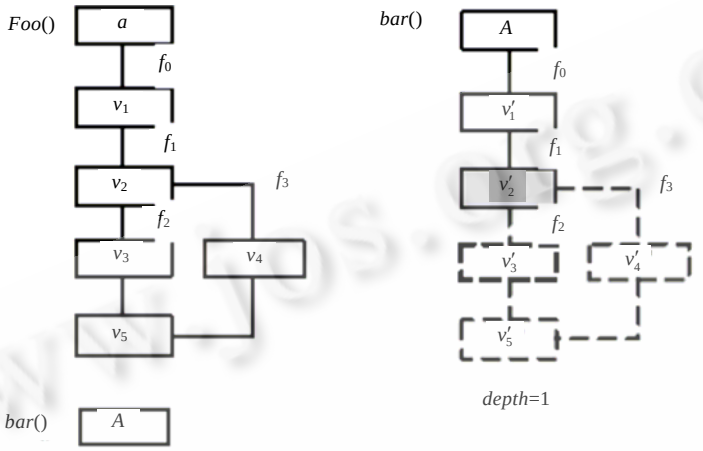
图 1 中的重新分析模块在 DEM 发生时将重新划分 *SCC*,从 DEM 引入的新的边和节点开始,重新运行逃逸分析算法.由于本文介绍的算法采用两个逃逸状态位 S_{up} 和 S_{down},重新分析可以从新载入的方法开始,而不必从 G 的所有叶节点开始.重新分析结束后,即时编译器将根据分析结果取消或恢复同步操作,保证 Java 程序运行的正确性.

4 复杂性控制

逃逸分析在调用方法与被调用方法之间复制 *aliasSet* 和 *fieldMap* 时,将引入较大的时间与空间复杂性,这样的操作占据了逃逸分析算法消耗的大部分时间和内存.本节介绍一种逃逸分析的近似实现,可以用于在即时编

译器重新分析整个程序时控制复杂性与精度,以便在运行时达到某种平衡.

该近似实现通过在复制 *fieldMap* 过程中控制被复制 *aliasSet* 的深度来保证算法在某一约定的复杂性之下可以完成分析.例如在图 4 中,方法 *bar* 调用了方法 *foo*,并且,变量 *A* 和 *a* 分别是 *foo* 的实参和形参.在逃逸分析进行到自底向上传导阶段时,变量 *a* 通过对象域可以访问到的所有 *aliasSet* 的逃逸状态,即变量 *a* 通过 *fieldMap* 可以递归访问的 *aliasSet* 的逃逸状态都必须复制传导到 *bar* 中变量 *A* 所指向的 *aliasSet* 中去.如果不加控制,*A* 所在的 *aliasSet* 及其 *fieldMap* 可以访问的等价变量组将呈现如图 4(b)所示的形态,包括所有虚线所指的范围.



(a) Before propagation (b) After propagation

Fig.4 Replication with constraints on field depth

图 4 复制时采用深度控制

该近似实现的做法是,设定 *depth* 为复制 *fieldMap* 时的控制深度,从变量 *A* 算起,最多有 *depth*+1 层的对象域将被保留在 *bar* 的方法摘要当中.如图 4 所示,当 *depth*=1 时,被复制的 *aliasSet* 将被限制到 v'_2 为止,其他的 *aliasSet*,例如 v'_3, v'_4 等,以及 *fieldMap* 关系 f_2 和 f_3 将被忽略.为了保证逃逸分析的正确性,v'_2 的逃逸分析状态将被设为 *global*.

通过这样的约束,可以将方法间的复制操作的复杂性控制在 $O(n^{depth+1})$以内,其中 *n* 为方法中 *aliasSet* 的数目.这样的设定尽管牺牲了部分精度,但是可以让即时编译器更好地控制重新分析时间,不至于出现 Java 程序通过逃逸分析提高的运行性能被过长的重新分析时间抵消.

采用不同的性能基准测试程序评估本文提出的逃逸分析近似实现后,可以得到一些有意义的数据.图 5~图 7 展示了该近似实现的运行情况.程序运行测试时的有关硬件环境和软件参数与下一节“性能评测”中一致.

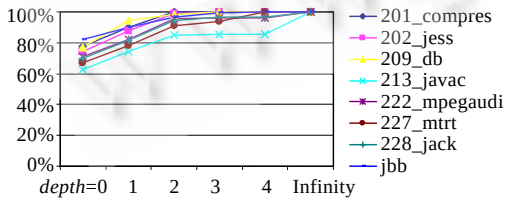


Fig.5 Memory requirement by approx EA

图 5 近似实现消耗的内存空间

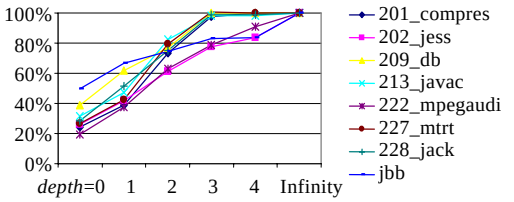


Fig.6 Analysis time by approx EA

图 6 近似实现消耗的分析时间

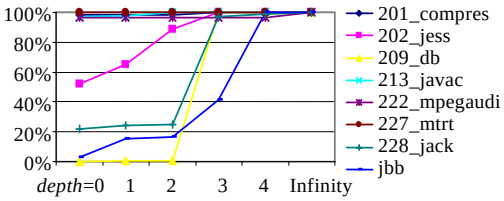


Fig.7 Synchronization removed by approx EA

图 7 近似实现消除的同步操作

图 5~图 7 的比较基线是没有经过复杂性控制的逃逸分析算法,即 *depth*=∞.图 5 显示,当 *depth*=0,即只有一层 *fieldMap* 被复制时,近似实现消耗的内存为原来的 61%~82%.图 6 显示,当 *depth*=0 时,近似实现消耗的编译时间仅为原来的 5%~45%;并且当 *depth*=3 时,仍然可以节省大约 20%的编译时间.对比图 7 给出的数据统计可以发现,当 *depth*=3 时,绝大部分的测试程序都可以达到全面分析的类似精度.也就是说,通过控制方法间复制 *aliasSet* 和 *fieldMap* 的深度,可以有效地降低逃逸分析算法的时间和空间复杂性.

5 性能评测

本文提出的逃逸分析架构和算法已经在 Intel 的开放式虚拟机平台(open runtime platform,简称 ORP)[14]中实现.由于在实践中发现将对象分配在堆栈上对于 Java 程序的整体性能影响并不明显,因此,在本节中所做的性能评测都是针对削减同步操作的优化.ORP 运行时采用的操作系统为 Windows2000 服务器版本,CPU 为 Intel 的 Dual Pentium4 Xeon 2.0G,内存为 2GB,Java 类库采用的是 GNU 的 Classpath 0.04[16].基准测试程序选用的是 SPECjvm98[17]程序包和 SPECjbb2000[18].在运行 SPECjvm98 时堆大小为 128M,运行 SPECjbb2000 时堆大小为 768M.

表 2 中给出了 8 个不同程序的评测数据,其中,前 7 个均为 SPECjvm98 中的测试程序,最后一个为 SEPCjbb2000.这些测试程序中,db 和 SPECjbb2000 中均包含大量的同步操作.逃逸分析算法在重新分析时采用了 *depth*=3 的约束.

Table 2 Performance enhancement by EA

表 2 逃逸分析的优化性能测试

Program	Base (s/throughput)	Mem (M byte)	Init. Ana. (s)	RA time (s)	RA	Imprv (%)	Removed sync. (%)
Compress	6.64	42	0.32	0.01	1	0.66	18.41
Jess	5.14	46	0.49	0.15	13	1.6	0.68
Db	14.53	42	0.20	0.02	2	31.28	93.93
Javac	13.58	53	0.39	0.31	2	4.68	41.98
Mpeg	4.61	42	0.39	0.13	1	1.06	12.40
Mtrt	3.13	42	0.29	0.03	3	1.16	99.10
Jack	8.36	44	0.33	0.28	1	2.24	17.62
SPECjbb	25182	53	0.19	0	0	15.77	70.64

表 2 中, Base 列给出了所有 SPECjvm98 程序在没有经过优化之前的运行时间以及 SPECjbb2000 在优化之前的 throughput 处理能力.Mem 列给出了所有程序在进行逃逸分析时多余消耗的内存大小.“Init. Ana.”给出了在 Java 程序实际运行之前,虚拟机用来进行初始逃逸分析的时间.“RA Time”表明在 Java 程序运行过程中重新进行逃逸分析消耗的时间.“RA #”表示在运行过程中实际发生的重新分析次数.需要注意的是,该项数据为 0 并不意味着没有动态类载入等现象发生,而是没有上文介绍的 DEM 发生.“Imprv %”表示 Java 程序运行性能的提高,其中已将即时编译器产生的负载考虑进去.“Removed sync.%”表示程序性能的提高,对于 SPECjvm98 的测试程序而言,指的是运行时间的缩短;对于 SEPCjbb2000 而言,指的是 throughput 数据的提高.

从表 2 中可以看出,拥有较多同步操作的 SPECjbb2000 和 db 程序的性能分别提高了 15.77%和 31.28%,被削减的同步操作可以高达 70.64%和 93.93%.这也表明了逃逸分析在并行 Java 程序优化中的重要地位.对于其他

程序,由于它们要么是单线程的,例如 compress,jess,javac 和 jack;要么即便是多线程的,同步操作也被基本优化掉,但是由于本身的同步负载并不大,所以对性能的提高也就不明显,例如 mtrt.

总体来说,逃逸分析能够比较高效地消除不必要的同步操作造成的程序负载,比较明显地提高 Java 程序的运行效率.

6 结束语

在 Java 虚拟机中实现逃逸分析首先要保证分析的正确性,然后还需要考虑算法的效率和精度.由于真实的 Java 程序在运行时可能随时发生动态类载入等现象,因此,如何能够正确、全面地收集和分析这些情况,是逃逸分析能够实用化的前提.同时,由于进行逃逸分析时占用的系统资源,包括时间和空间,都将被计算为 Java 程序运行资源的一部分,如何在保证分析精度的前提下减少分析的复杂性也是实现逃逸分析时需要考虑的一个重要方面.

本文提出了一个新的逃逸分析架构,它可以全面、有效地捕捉 DEM 的发生,同时,在算法上充分支持运行时的重新分析;并且在此基础上引入一个逃逸分析的近似实现,可以在保证分析精度的同时有效地控制和降低算法的复杂性.本文较为详细地描述和分析了这样一个架构的运行机制和原理,并采用比较权威的基准测试程序对实现在 ORP 中的该架构进行了充分的测试和评估.

References:

[1] Gosling J, Joy B, Steele G, Bracha G. The Java Language Specifications. 2nd ed., 2000. http://java.sun.com/docs/books/jls/second_edition/html/j.title.doc.html

[2] Blanchet B. Escape analysis for object oriented languages: Application to Java. In: Proc. of the Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'99). New York: ACM Press, 1999. 20–34.

[3] Choi J, Gupta M, Serrano M, Sreedhar V, Midki S. Escape analysis for Java. In: Proc. of the Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'99). New York: ACM Press, 1999. 1–19.

[4] Ruf E. Effective synchronization removal for Java. In: Proc. of the Conf. on Programming Language Design and Implementation (PLDI 2000). New York: ACM Press, 2000. 208−218.

[5] Sreedhar VC, Burke M, Choi JD. A framework for interprocedural optimization in the presence of dynamic class loading. In: Proc. of the Conf. on Programming Language Design and Implementation (PLDI 2000). New York: ACM Press, 2000. 196–207.

[6] Gay D, Steensgaard B. Fast escape analysis and stack allocation for object-based programs. In: Proc. of the 9th Int'l Conf. on Compiler Construction (CC 2000). 2000. 82–93.

[7] Ghemawat S, Randall KH, Scales DJ. Field analysis: Getting useful and low-cost inter-procedural information. In: Proc. of the PLDI 2000. New York: ACM Press, 2000. 334–344.

[8] Ji ZY, Cheng H. Java compiler technology and Java performance. Journal of Software, 2000,11(2):173−178 (in Chinese with English abstract).

[9] Zhong HT, Shu JW, Wen DC, Zheng WM. A communication optimization algorithm based on data-flow analysis of region graph. Journal of Software, 2003,14(2):175−182 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/14/175.htm

[10] Vivien F, Rinard M. Incrementalized pointer and escape analysis. In: Proc. of the Conf. on Programming Language Design and Implementation (PLDI 2001). Snowbird: ACM Press, 2001. 35–46.

[11] Bogda J, Singh A. Can a shape analysis work at runtime? In: Proc. of the Java Virtual Machine Research and Technology Symposium. Monterey: USENIX Association, 2001. 2–2. http://portal.acm.org/citation.cfm?id=1267847.1267849&coll= &dl=

[12] Arnold M, Ryder BG. Thin guards: A simple and effective technique for reducing the penalty of dynamic class loading. In: Proc. of the ECOOP 2002. Málaga: Springer-Verlag, 2002. 498–524.

[13] Pechtchanski I, Sarkar V. Dynamic optimistic interprocedural analysis: A framework and an application. In: Proc. of the Conf. on Object Oriented Programming Systems Languages and Applications (OOPSLA 2001). New York: ACM Press, 2001. 195–210.

[14] Open runtime platform (ORP) from Intel corporation. 2004. http://orp.sourceforge.net

[15] Aho AV, Sethi R, Ullman JD. Compilers: Principles, Techniques, and Tools. Boston: Addison Wesley Reading, 1986. 105–108.

[16] GNU classpath. 2007. http://www.classpath.org

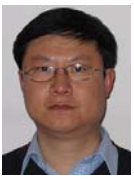
[17] SPECJVM98 benchmarks. 2004. http://www.specbench.org/osg/jvm98/

[18] SPECjbb2000 (Java business benchmark). 2006. http://www.specbench.org/osg/jbb2000/

附中文参考文献:

[8] 冀振燕,程虎.Java 编译程序技术与 Java 性能.软件学报,2000,11(2):173−178.

[9] 钟洪涛,舒继武,温冬婵,郑纬民.基于区域图数据流分析的通信优化算法.软件学报,2003,14(2):175−182. http://www.jos.org.cn/1000-9825/14/175.htm



史晓华(1973－),男,江苏宜兴人,博士,原 Intel 微处理器研究院 Senior Researcher,主要研究领域为并行计算,编译系统,运行时系统,微处理器体系结构.

吴甘沙(1975－),男,Intel 微处理器研究院 Senior Researcher,主要研究领域为微处理器体系结构,运行时系统.

金茂忠(1941－),男,教授,博士生导师,主要研究领域为编译系统,面向对象软件工程,软件测试方法.

LUEH Guei-Yuan(1963－),男,博士,Intel 微处理器研究院 Principal Researcher,主要研究领域为编译器,微处理器体系结构,运行时系统.

刘超(1958－),男,博士,教授,博士生导师,主要研究领域为面向对象软件工程,软件测试方法.

王雷(1969－),男,博士,副教授,主要研究领域为操作系统,微处理器体系结构.