即时编译器辅助的垃圾收集中的插桩算法研究

张 昱[1,2]，袁丽娜[1,2]

[1]（1 中国科学技术大学 计算机科学技术系，安徽 合肥 230027）

[2]（2 中国科学院计算机系统结构重点实验室，北京 100080）

E-mail ：yuzhang@ustc.edu.cn

摘 要：即时编译器辅助的垃圾收集技术结合显式和自动内存管理的优点，在编译阶段由即时编译器分析应用程序并在其中插桩显式释放内存的指令，以便垃圾收集器及时回收死亡对象所占用的内存空间，从而减轻垃圾收集器的负担。提出一种应用于该项技术的插桩算法，它基于控制流中的支配关系并提供不同的插桩策略，保证插桩的正确性和灵活性；它能够主动获得域引用从而释放对象及其域引用的内存空间。实验表明基于该插桩算法的垃圾收集器能够回收大量的内存空间，提高 Java 程序的执行效率。

关键词：插桩；即时编译器；垃圾收集器；内存管理

中图分类号：TP **文献标识码：**A **文章编号：**1000-1220（2008）02--

Study on Instrumentation Algorithm for Just-in-time Compiler Assisted Garbage Collection

ZHANG Yu[1,2]，YUAN Li-na[1,2]

[1] *(Department of Computer Science & Technology, University of Science & Technology of China, Hefei 230027, China)*

[2] *(Key Laboratory of Computer System & Architecture, Institute of Computing Technology, Chinese Academy of Sciences, Beijing, 100080)*

Abstract: Just-in-time compiler assisted garbage collection (JIT-assisted GC) combines the advantages of explicit memory management and automatic memory management. Under the framework of JIT-assisted GC, JIT compiler analyses the applications to find dead objects and their dead points, then instruments the explicit "free" instructions in compilation time, accordingly garbage collector could reclaim the spaces owned by those dead objects in time and relieves its load on automatic garbage collection. A new algorithm on instrumenting "free" instructions in JIT-assisted-GC is proposed in this paper. It bases on the domination relations in the control flow and various instrumenting strategies are presented to ensure the validity and flexibility of instrumentation. Moreover, the object fields which refer to dead objects are detected, and instructions for loading those field references are produced and instrumented automatically. Experimental results show that the proposed algorithm for JIT-assisted-GC could reclaim plentiful memory spaces and improve the performance of Java applications efficiently.

Key words: instrument；Just-in-time；garbage collection；memory management

1 引 言

Java 语言采用垃圾收集器(Garbage Collector，GC)进行内存管理。垃圾收集采用自动的内存管理技术在堆上管理对象的分配与回收。当堆空间不足时，GC 就会自动启动垃圾收集来识别并回收死亡对象所占用的内存空间。这种内存管理方式减轻了程序员的编程负担，降低了程序中与内存相关的错误的发生几率，然而它也为整个系统运行带来了额外的时空开销，是影响 Java 虚拟机性能的重要因素之一。

为了提高 GC 管理内存的性能，一种解决方法是将一部分对象在堆以外的内存空间中分配，如在运行栈上分配[1-2]，或者采用基于区域的内存分配方式[3-4]。由于 GC 在进行对象回收时，有大量的时空开销消耗在识别堆中哪些对象是死亡的，一些研究工作[5-8]提出了即时编译器(Just-in-time

收稿日期：2008-- 基金项目：中国科学院计算机系统结构重点实验室开放课题；Intel 公司研究基金资助项目；国家自然科学基金资助项目(60673126) 作者简介：**张昱**，女，1972.8，博士，副教授，研究方向为程序设计语言理论与实现技术，特别是并行语言设计、编译，并行程序分析和验证等。中国计算机学会（CCF）会员，会员号 E200009601M；**袁丽娜**，女，1985.2，硕士研究生，研究方向为程序分析技术。

Compiler，JIT)辅助的内存管理技术，做法是由 JIT 分析应用程序以识别确定其中的死亡对象及位置，并在程序中合适的位置安插显式释放内存的指令来主动通知 GC 回收内存；GC 接到显式内存释放请求后，不仅可以立即回收内存空间还可以采取某些策略对这些空间进行有效重用，从而减轻了 GC 自动回收内存的负担。

现有的关于即时编译器辅助的垃圾收集的研究工作主要集中在程序分析技术[5-8]和支持对象显式回收的垃圾收集器[9]上。文献[5]结合了流不敏感的指针分析和流敏感的活跃变量分析，能够分析出程序中的短生命期对象，但无法识别那些被对象域所引用的对象的生命期。文献[7]根据过程内的别名分析和全局约束系统，分析出程序中具有唯一引用(变量或对象的域)的对象的生命期，通过为对象所属的类添加析构器的方式来释放整个对象空间。笔者所在的课题组也进行了这方面的研究：[8]提出一种基于逃逸分析的、过程间的对象生命期分析方法，不仅能分析对象是否全局共享，而且能分析非全局共享对象的生命期在线程中的哪个方法内结束；[9]提出一种基于即时编译器辅助的并行垃圾收集器的实现，它能够支持显式的对象回收操作，且可以有效地重用这些对象空间。

本文重点研究如何根据程序分析识别出的对象死亡位置，为死亡对象确定一个合适的插桩位置并安插显式内存释放指令。插桩主要涉及到以下三个问题：一是如何确定正确的插桩位置，因为某些情况下直接在对象死亡位置安插显式的内存释放指令会引起编译时或运行时的错误。例如在分支中创建的对象若其生命期在分支外结束，此时直接在死亡位置安插显式内存释放指令会导致运行时错误；二是如何释放那些在方法 *M* 外创建，在 *M* 内死亡，但在 *M* 中没有显式引用的对象空间，如某些通过 *M* 中的调用点处的实参和返回值接收者的域所返回的对象；三是如何更好地利用程序分析结果，使得分析出的死亡对象的内存空间能够尽可能多地被释放。针对这些问题，本文提出一种基于控制流中支配关系的插桩算法，它为不同类型的死亡对象提供不同插桩策略，确保了插桩的正确性和灵活性，且能够主动生成指令获得对象的域引用，从而释放对象及其域引用的对象内存空间。实验结果表明基于该插桩算法的垃圾收集器能够回收大量的内存空间，提高垃圾收集器性能和 Java 程序的执行效率。

2 即时编译器辅助的垃圾收集框架

我们在开源的 Java SE 平台 Apache Harmony[11]上实现了即时编译器辅助的垃圾收集。整个框架如图 1 所示，涉及到即时编译器(JIT)、垃圾收集器(GC)和虚拟机核心(VMCore)这三个模块。

JIT 采用流水线机制来编译优化 Java 字节码。流水线依次载入待编译的方法的字节码，首先将字节码翻译成平台无关的高级中间表示(High-level Intermediate Representation，HIR)，然后由代码选择器将 HIR 转换为平台相关的低级中间表示(Low-level Intermediate Representation，LIR)，最后发射为可执行的本地代码。在 HIR 和 LIR 上分别执行一系列的优化遍，本文描述的插桩算法和对象生命期分析一起作为一个优化遍在 HIR 上被实现。

JIT 侧(见图 1 中 JIT 模块里的阴影部分)的主要工作是：1)在 HIR 和 LIR 中增加支持对象显式释放内存的指令。2)在 HIR 上添加一个优化遍。该优化遍执行对象生命期分析和显式的内存释放指令的插桩。3)修改代码选择器和代码发射器，使之支持扩展的 HIR 到扩展的 LIR 的变换并能够将显式的内存释放指令发射为运行时的指令调用。本文将介绍对象生命期分析并详细阐述插桩算法。

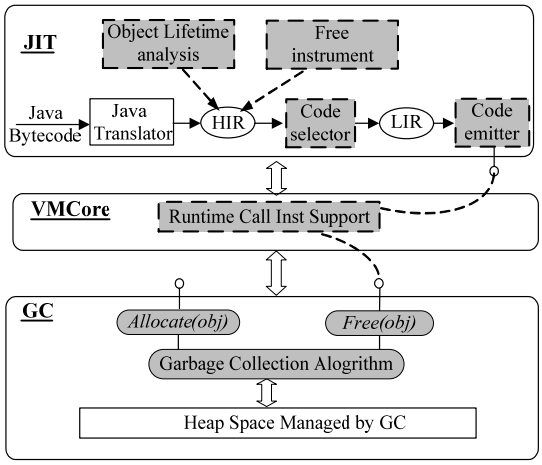

图 1 即时编译器辅助的垃圾收集器的框架

Fig. 1 Framework of JIT-assisted-GC

VMCore 侧(图 1 的 VMCore 模块里的阴影部分)负责协调 JIT 和 GC 这两个模块的交互。需要在其中添加新的运行时支持，将显式释放指令生成的指令调用映射到 GC 中提供的对象显式回收接口 free()。

GC 侧(见图 1 中 GC 模块里的阴影部分)的主要工作是提供对对象显式回收接口 free()的实现，改进对象分配接口 allocate()的实现以支持对显式回收空间的及时重用，它的实现详见文献[9]。

3 对象生命期分析

在对被编译的方法插入显式内存释放指令之前，必须知道在该方法内哪些对象是死亡的，这就依赖于对象生命期分析。本文基于的对象生命期分析在文献[8]的基础上进行了进一步的改进。与[8]相比，在过程内分析中引入了活跃变量分析，因此对于那些非全局对象，不仅能够分析其生命期在线程的哪个方法内结束，还能分析其生命期在方法的哪个基本块尾部结束；在过程间信息传播时不仅结合了形参与实参，返回值与返回值接收者之间的信息，还进一步对它们的域所引用的对象信息进行结合，因此能够识别出更多的死亡对象，特别是域引用的对象的生命期。本文所使用的对象生命期分析是一个过程间的、上下文敏感的分析，在此只简单介绍，将另文详细阐述。

对于一个方法 *M*，下面的数据流方程描述了在 *M* 中哪些对象是可能死亡的：

Create(*M*) = {O_c | Object O_c is created in *M* }；

In(*M*) = {O_{in} | Object O_{in} is passed into *M* through the call sites in *M* }；

Out(*M*) = {O_{out} | Object O_{out} is passed into the callers of *M*

through the return value or the formal parameter of *M*}；

MayDead(*M*) = Create(*M*) + In(*M*) – Out(*M*)；

其中 Create(*M*)表示在方法 *M* 中创建的对象集合；In(*M*)表示在 *M* 的被调用者中创建，在方法 *M* 中的调用点处通过实参和返回值接收者传递到 *M* 中的对象集合；Out(*M*)表示通过 *M* 的形参和返回值传递到 *M* 的调用者中的对象集合。MayDead(*M*)表示可能在 *M* 中死亡的对象集合。

下面通过一个例子来说明对象生命期分析是如何识别死亡对象的。图 2 是 Jolden[10]中基准程序 BH 的代码片段(为便于阅读，这里使用 Java 源代码来示意，本文的分析工作是在对应的 HIR 指令上展开的)。方法 subdivp 中有 3 处调用点，分别位于第 14~16 行。其中第 14 行是一个对象分配点也是隐含的调用点，它创建了一个类型为 MathVector 的对象，通过自动调用类 MathVector 的构造器来初始化新创建的对象，最后将该新创建的对象赋值给变量 *dr*，记该新创建的对象记为 O_{dr}。在对象初始化中，构造器会创建一个 double 类型的数组对象并将其赋值给对象 O_{dr} 的 *data* 域，记该数组对象为 $O_{dr.data}$。方法 subtraticon 只做简单的数学计算，没有改变实参和返回值。方法 dotProduct 也只做简单的数学计算，它的返回值是基本数据类型而非对象类型。由上可知，In(subdivp) = {$O_{dr.data}$ }，Create(subdivp) = {O_{dr}}。方法 subdivp 只返回基本数据类型且没有对形参的域进行赋值，所以 Out(subdivp) = $\varnothing$。因此 MayDead(subdivp) = { O_{dr}，$O_{dr.data}$ }，表示在方法 subdivp 中有两个对象是可能死亡的，而这两个对象不是全局对象，即不会被 subdivp 所在线程之外的线程所访问，其生命期在 subdivp 中结束，因此可以在它们死亡之后将其内存空间显式释放。由于方法 subdivp 中没有显式地出现对对象 $O_{dr.data}$ 的引用，我们的编译器会主动生成指令获得域引用对象的引用，从而释放对象 $O_{dr.data}$ 的内存空间，如 17~19 行所示，具体的插桩过程将在下一节描述。

```
1: class MathVector implements Cloneable{
2:    public final static int NDIM = 3;
3:    private double data[];
4:    MathVector(){
5:       data = new double[NDIM];
6:       for (int i=0; i < NDIM; i++)
7:             data[i] = 0.0;
8:    }
9: }
10: final class Cell extends Node{
11:    MathVector pos;
12:    ……
13:    final boolean subdivp(double dsq, HG hg){
14:       MathVector dr = new MathVector();
15:       dr.subtraction(pos, hg.pos0);
16:       double drsq = dr.dotProduct();
17:       double[] v = dr.data;
18:       free(v);
19:       free(dr);
20:       return (drsq < dsq);
21:    }
22:}
```

图 2 Jolden[10]中基准程序 BH 的代码片段，斜体字是编译器插桩的

Fig. 2 Code fragment of BH, one of the Jolden[10] benchmarks. The code in italics is inserted by our compiler.

对于一个方法 *M*，它的对象生命期分析结果分为两部分：<*DeadInfo*，*MethodSummary*>，一是死亡对象信息 *DeadInfo*，二是方法概要 *MethodSummary*。*DeadInfo* 记录了哪些对象是死亡的以及对象的死亡位置，插桩算法将根据这些死亡位置来确定对象的插桩位置。由于我们在对象生命期分析中引入了活跃变量分析，因此分析确定的对象死亡位置是精确到基本块的，即不仅知道哪些对象在 *M* 中死亡，还能够知道这些对象在哪个基本块尾部死亡。如图 2 所示，可以判断出 MayDead(subdivp)中的两个对象在第 16 行指令所在的基本块尾部死亡。*MethodSummary* 记录了 *M* 的形参和返回值信息以及它们的域所引用的对象信息，应用于过程间的信息结合，在每个调用点处均将被调用者的方法概要传播到调用者中，因此分析是上下文敏感的且能够判断出更多的死亡对象。

需要说明的是当判断出一个对象 O_1 在某个基本块尾部死亡之后，总是先去检查它的域所引用的对象 O_2 是否死亡，保证能够识别出被域引用的对象的生命期。对于图 2 中的方法 subdivp，MayDead(subdivp) = { O_{dr}，$O_{dr.data}$ }，假设先判断 $O_{dr.data}$，因为它被 O_{dr} 的 *data* 域所引用，所以生命期未结束。在判断出 O_{dr} 生命期结束之后，检查它的 *data* 域所引用的对象 $O_{dr.data}$，由于 $O_{dr.data}$ 只被 O_{dr} 引用，所以 $O_{dr.data}$ 在 O_{dr} 生命期结束之后死亡。

4　插桩

插桩包括以下两步：一是插桩信息的收集，即为死亡对象选择合适的引用，然后根据对象生命期分析确定的死亡位置确定出合适的插桩位置，并为死亡对象选取合适的插桩策略，在 4.1 至 4.3 节进行论述；二是根据插桩算法进行真正的插桩，在 4.4 节进行论述。插桩是局部于一个方法的，对其他方法不会产生影响。

4.1　相关概念

我们假定程序基于静态单赋值(Static Single Assignment，SSA)格式，SSA 格式可以使得分析具有流敏感性。我们还假定所有形如 *x.y.f* 这类多级的变量表示方式将转化为 *t* = *x*. *y* 和 *t.f*。

定义 1. 对于方法 *M* 的控制流图中的两个基本块结点 A 和 B，如果到达结点 B 的控制流在到达 B 之前必经过结点 A，称结点 A 支配结点 B，记作 Dominate(A, B) = true。

必须保证对象的分配点所在基本块结点支配显式释放对象指令插桩所在的结点，即在释放对象空间时必须保证无论程序沿着哪条路径执行时均会先创建对象，否则程序执行时会发生错误。

定义 2. 对于方法 *M*，*M* 中死亡对象的插桩信息是一个二元组集合，即 *InstrumentInfo* = {<*ref*, *loc*>}。其中 *ref* 描述某死亡对象 *O* 的引用，它是一个二元组即 *ref* = <*var*, *fielddesc*>，若 *fielddesc* 为 NULL，说明变量 *var* 是 *O* 的引用；若 *fielddesc* 为某个域 *f*，说明 *var.f* 是 *O* 的引用，即此时的死亡对象 *O* 被变量 *var* 所引用的对象的 *f* 域所引用。*loc* 描述插桩位置，在此代表控制流图中的结点。

对于那些在方法 *M* 中创建并在 *M* 中死亡的对象，在 *M* 中肯定会显式出现引用它的变量。那些在 *M* 外创建但在 *M*

中死亡的对象，在 M 中则不一定会显式出现引用它的变量，如第 3 节例子中的对象 $O_{dr.data}$。此时需要借助其它变量来获得这些对象的引用，这些对象的生命期在 M 中结束说明它们一定曾被 M 中出现的变量的域所引用，因此可以用这些变量和域描述一起间接地表示这些对象的引用。

一个死亡对象 O 的引用分为以下 2 类：

1) 对象分配点处的引用：例如 a = new A()，a 为该分配点所创建对象的引用。特别地，对于 $v = v_0.\text{foo}(v_1, v_2, \ldots, v_n)$；若 foo 方法返回一个新创建的对象 O_1，此时 v 就为对象 O_1 的引用；若 foo 方法通过 v 或 v_i(i= 0,1,…,n)的某个域 f 返回一个新创建的对象 O_f，此时 $v.f$ 或 $v_i.f$(i= 0,1,…,n)就为对象 O_f 的引用。例如图 2 中方法 subdivp 的第 14 行是一个隐含的调用点，它不仅返回一个对象 O_{dr} 赋给 dr，还通过 O_{dr} 的域 $data$ 返回一个数组对象 $O_{dr.data}$，$O_{dr.data}$ 的引用为 $dr.data$。我们称该类引用是第I类引用，每个对象只有一个该类引用，记为引用 $ref_{\text{I}}(O)$。

2) 由于 phi 语句、赋值语句或域赋值带来的引用。例如 a=phi(a_1, a_2)，则 a 为 a_1 或 a_2 所引用的对象的引用；$a = b$，则 a 为 b 所引用的对象的引用；$a.f = b$，则 $a.f$ 为 b 所引用的对象的引用。我们称该类引用是第 II 类引用，每个对象的第 II 类引用的个数不定，一般大于等于 1，记为引用集合 $Ref_{\text{II}}(O)$。

死亡对象 O 的插桩位置可以从以下 2 种结点中选取：

1) return 语句所在的结点：在此简称 return 结点。在程序不发生异常情况下，肯定有一个 return 结点会被执行到，所以这是一个可供选择的位置。我们称该种插桩位置为第 I 类插桩位置，该类结点是一个集合，个数至少为 1，与对象无关，记为位置集合 $Loc_{\text{I}}(O)$。

2) 对象生命期分析得到的对象死亡位置：因为我们分析的粒度是精确到基本块的，所以这个位置是可以获得的。我们称该种插桩位置为第 II 类插桩位置，每个死亡对象只有一个第 II 类插桩位置，记为位置 $loc_{\text{II}}(O)$。

4.2 插桩信息收集

对于一个死亡对象 O，用 $ref_{\text{I}}(O) = <var_{\text{I}}, fielddesc_{\text{I}}>$和 $Ref_{\text{II}}(O) = \{<var_{\text{II}}, fielddesc_{\text{II}}>\}$分别表示对象 O 的第 I 类引用和第 II 类引用集合，$Loc_{\text{I}}(O)$和 $loc_{\text{II}}(O)$分别表示对象 O 的第 I 类插桩位置集合和第 II 类插桩位置，$Node(var)$表示变量 var 第一次出现时所在的基本块结点，称作 var 所在的结点。

插桩信息收集算法 Collector 依次扫描死亡对象信息中的每个死亡对象 O，当对象 O 满足某个插桩类型的条件时，将相应的引用和插桩位置记录到插桩信息 $InstrumentInfo$ 中。该算法总是优先地为死亡对象选择第 I 类引用和第 I 类插桩位置，插桩信息收集时的优先级为：插桩类型 1>插桩类型 2> 插桩类型 3> 插桩类型 4。

插桩信息收集算法 Collector：根据方法 M 的对象生命期分析结果，为方法 M 中的死亡对象收集插桩信息。

输入：方法 M 中的死亡对象信息 $DeadInfo$。

输出：方法 M 中死亡对象的插桩信息 $InstrumentInfo$。

```
1:  for (each dead object O in DeadInfo) {
2:      bool isRecord = false;
3:      obtain ref_I(O) = <var_I, fielddesc_I> from DeadInfo
4:      for(each loc_I  in Loc_I(O) ){ //插桩类型 1
5:          if(Dominate( Node(var_I), loc_I) ){
6:              record < ref_I(O), loc_I > to InstrumentInfo;
7:              isRecord = true;
8:          }
9:      }
10:     if (isRecord)    continue;
11:     if(Dominate( Node(var_I), loc_II(O) ){ //插桩类型 2
12:         record < ref_I(O), loc_II(O)> to InstrumentInfo
13:         isRecord = true;
14:     }
15:     if (isRecord)    continue;
16:     obtain Ref_II(O) from DeadInfo and select the last
        element ref_II(O)= <var_II, fielddesc_II> from Ref_II(O)
17:     for(each loc_I in Loc_I(O)){ //插桩类型 3
18:         if (Dominate( Node(var_II), loc_I) ){
19:             record < ref_II(O), loc_I > to InstrumentInfo
20:             isRecord = true;
21:         }
22:     }
23:     if (isRecord)    continue;
24:     if(Dominate( Node(var_II), loc_II(O)) ){//插桩类型 4
25:         record < ref_II(O), loc_II(O)> to InstrumentInfo
26:     }
27: } //end for
```

4.3 例子

下面通过一个例子说明考虑结点的控制流支配关系的必要性和各种插桩类型的适用情况。

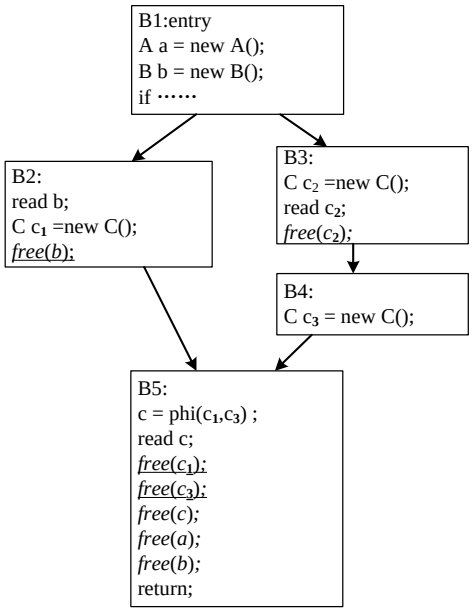

图 3 方法 M 的控制流图

Fig. 3 Control flow graph of method M

如图 3 所示，在基本块结点 B2 和 B4 中分别创建了一个 C 类型的对象，它们均在基本块 B5 中死亡。Dominate(B2, B5) = Dominate(B4,B5) = false，如果不考虑支配关系，就会在 B5 中安插两条带有下划线的显式释放指令，而程序在执

行时只会经过其中一条分支，在执行“*free*(c_1);*free*(c_3);”时肯定会发生错误，因为此时释放的是一个并没有创建的对象空间，c_1 或 c_3 是无效的对象引用。

一个死亡对象肯定有第 I 类引用，但当对象的第 I 类引用所在的结点不能支配 return 结点和对象死亡点时，此时需要使用第 II 类引用。由于 B5 中的 phi 语句，所以 *c* 是 c_1 和 c_3 这两个变量引用的对象的第 II 类引用，因此可以在 return 结点即基本块 B5 中安插 *free*(*c*)。而 Dominate(B1,B5) = true，所以基本块 B1 中创建的 A 类型和 B 类型的对象均可在基本块 B5 中释放。

我们优先地将 *free* 指令安插到 return 结点，这样可以保证 *free* 指令尽可能多地被执行到。基本块 B1 中创建了一个引用为 *b* 的 B 类型对象，在 B2 尾部死亡，若在 B2 中安插显式释放指令，当程序实际执行时走 B1-B3-B4 路径时，此时 B2 中的 *free* 指令就不会被执行到，而只要程序不发生异常情况，B5 中的 *free* 指令一定会被执行到。对于作用域是某个语句块中的对象，如分支或循环，一般情况下均不能在 return 结点中安插 *free* 指令，此时对象的死亡点是一个很好的选择，如 B3 中创建的 C 类型的对象，特别的当 B3 处在循环中时，在对象死亡点插桩会带来很大的收益。

4.4 插桩算法

当收集好插桩信息后，此时可以进行插桩，插桩算法 Instrumentor 首先依次扫描每条插桩信息，然后创建相应的显式内存释放指令 *freeInst*，最后根据插桩位置在方法的控制流图中安插新创建的指令。

特别地，如果一条插桩信息<*ref,loc*>中的 *ref.fielddesc* 不为空，表明此时要释放的是域引用的对象，首先需要新建一条取域地址指令获得域地址 *fieldaddr*，然后新建一条加载指令获得域引用 *y*，最后创建一条显式内存释放指令用于释放变量 *y* 所引用的对象空间，如 Instrumentor 算法的 4~6 行所示。插桩位置 *loc* 是精确到某个基本块的，*loc* 的最后一条指令假设为 *lastInst*。如果 *lastInst* 不是分支跳转指令，分为两种情况：如果 *lastInst* 是 return 指令，则在 *lastInst* 之前安插，如 12~18 行所示；否则在 *lastInst* 之后安插显式内存释放指令，如 19~25 行所示。但当 *lastInst* 为分支跳转指令时，此时在 *lastInst* 各个跳转目标指令之前插桩，如 27~35 行所示。

插桩算法 Instrumentor：根据方法 *M* 的插桩信息，为 *M* 安插显式的内存释放指令。

输入：方法 *M* 的 HIR；在 *M* 中死亡的对象的插桩信息 *InstrumentInfo*。

输出：方法 *M* 修改后的 HIR。

```
1:  for (<ref, loc>∈ InstrumentInfo) {
2:     x = ref.var ;
3:     if ( ref.fielddesc != NULL) {
4:        //创建取域地址指令，fieldaddr 为域地址
          ldFieldAddrInst = makeLdFieldAddr(x, fielddesc,
                                    fieldaddr);
5:        ldIndInst = makeLdInd(fieldaddr, y);
6:        freeInst = makeFreeInst(y);
7:     }
8:     else
9:        freeInst = makeFreeInst(x);
10:    lastInst = getLastInst(loc);
11:    if (! isBranchJumpInst(lastInst) ){
12:       if(isReturnInst(lastInst){
13:          if (ref.fielddesc != NULL){
14:             insertBefore(ldFieldAddrInst, lastInst);
15:             insertBefore(ldIndInst, lastInst);
16:          }
17:          insertBefore(freeInst, lastInst);
18:       }
19:       else{
20:          if (ref. fielddesc != NULL){
21:             insertAfter(ldFieldAddrInst, lastInst);
22:             insertAfter(ldIndInst, lastInst);
23:          }
24:          insertAfter(freeInst, lastInst);
25:       } // end if(isReturnInst(lastInst)
26:    }
27:    else {
28:       for ( targetInst_i ∈ jumpTarget(lastInst) ){
29:          if (ref.fielddesc != NULL){
30:             insertBefore(ldFieldAddrInst, targetInst_i);
31:             insertBefore(ldIndInst, targetInst_i);
32:          }
33:          insertBefore(freeInst, targetInst_i);
34:       } //end for
35:    }
36: } //end for
```

5 实验结果与分析

笔者在开源的 Java SE 平台 Apache Harmony 上，用 C++ 实现了 JIT-assisted-GC 的对象生命期分析和插桩算法，并做了相关的测试。实验平台的操作系统是 Windows XP，CPU 为 AMDX2，主频为 2.0G，内存为 896M。测试用例为 Jolden[10] 中的基准程序。

5.1 正确性验证

正确性验证主要检验对象生命期分析的正确性和插桩的正确性。体现在是否提前释放还没有死亡的对象空间或是在错误的位置释放死亡的对象空间。为了进行该项测试，修改了后端垃圾收集器的 free()接口，将回收的内存空间标记为不可访问和不可重用的。这样如果错误地释放了某个对象空间，则再次访问该对象时会发生异常。另外在流水线上对 LIR 有多项检测，如检验操作数在方法入口处是否活跃，辅助地测试了插桩的正确性，例如图 3 中若在 B5 中插桩那两条带有下划线的 *free* 指令，则操作数 c_1 和 c_3 在程序入口处就是活跃的，说明代码肯定有错误。实验结果显示程序没有发生编译时错误、运行时异常或错误，说明所设计和实现的算法能正确地释放死亡对象的内存空间。

5.2 效果测试

在虚拟机默认的堆大小(256MB)下，测试了

JIT-assisted-GC 显式回收对象空间大小和各种插桩类型的收益，结果如表 1 和表 2 所示。

表 1 插桩算法回收的内存空间大小

Table 1 Memory reclaimed by instrument algorithm

基准程序	分配对象数目/空间大小	free(x)/free(x.f)插桩数目	显式释放对象数目/空间大小
BH	2701783/67MB	31/15	2438018/60MB
TSP	1599242/51MB	9/2	1048578/28MB
Power	809979/24MB	11/3	760004/23MB
Health	3388012/60MB	16/0	600009/14MB

从表 1 可以看出，4 个基准程序的大部分内存空间都可以被显式回收。最少的 Health 释放了 23%的对象空间，最多的 Power 分配了 24MB 的内存，显式释放了 23MB 的内存空间。表 1 的第 3 列表示插桩的显式内存回收指令数，那些在循环中或者是递归调用方法中插桩的显式释放指令带来的收益最多，例如 Health 回收的 14MB 空间中将近 14MB 都来自于其中，这些地方插桩的显式释放指令回收的空间也最容易被重用。

表 2 各种插桩类型的收益

Table 2 Incomes of each instrument type

基准程序	插桩 free 数/显式释放的对象数目/空间大小 插桩类型 1	插桩类型 2	插桩类型 3	插桩类型 4
BH	30/2428013/60MB	16/10005/117KB	0/0/0B	0/0/0B
TSP	5/1048577/28MB	6/1/12B	0/0/0B	0/0/0B
Power	14/760004/23MB	0/0/0B	0/0/0B	0/0/0B
Health	2/500005/13MB	12/3/36B	1/1/20B	1/100000/1MB

表 2 所示的是各种插桩类型下的收益。插桩类型 1 安插的显式内存回收指令只要在程序不发生异常的情况下，总会被执行到，所以这种插桩类型的收益远远大于其它 3 种。插桩类型 3 的对象很少，因为如果一个对象的第 I 类引用所在的结点不能够支配 return 结点，则它的第 II 类引用所在的结点也很难支配 return 结点。插桩类型 2 和插桩类型 4 针对的均是作用域在某个语句块的对象，如在一个分支中创建且在该分支中死亡的对象。

5.3 性能改进

为比较原有的 GC 和 JIT-assisted-GC 在性能上的差异，我们考核在不同堆大小情况下运行四个基准程序时 GC 的停顿时间，结果见图 4 至 7 所示，其中横轴表示 GC 的堆大小，纵轴表示 GC 的停顿时间。

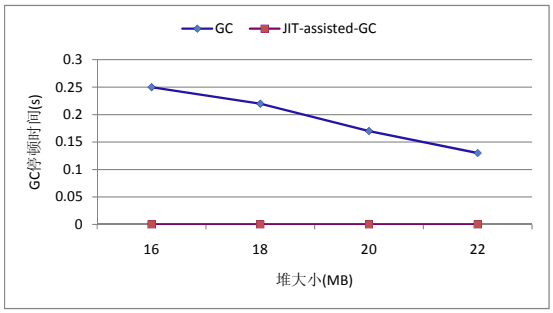

图 4 BH 的性能对比

Fig. 4 Performance comparison of BH

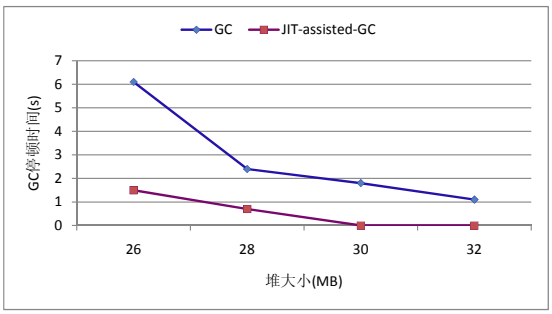

图 5 TSP 的性能对比

Fig. 5 Performance comparison of TSP

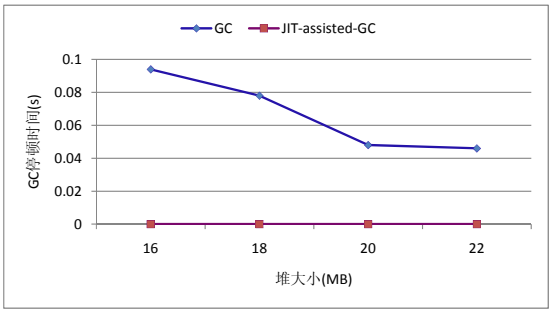

图 6 Power 的性能对比

Fig. 6 Performance comparison of Power

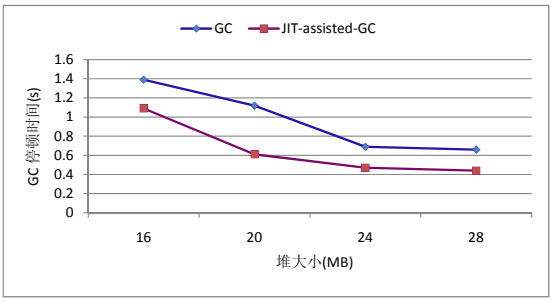

图 7 Health 的性能对比

Fig. 7 Performance comparison of Health

从图 4 至 7 可以看出，对于这四个基准程序，JIT-assisted-GC 在停顿时间上都少于原有的 GC，提高了应用程序的执行效率。随着堆的大小变大，性能提升的幅度会逐渐变小，这是因为可以显式回收的对象数目固定，而当堆变大时，同样大小的重用空间带来的性能提升相对减少，如图 5 和 7 所示。而在图 4 和图 6 中，即使将堆大小设置为最小的 16MB，JIT-assisted-GC 的停顿时间都为 0，这是因为此时使用 JIT-assisted-GC 恰好不会发生垃圾收集过程。

6 结束语

提出一种应用于即时编译器辅助的垃圾收集中的插桩算法，该算法首先考虑了控制流中的支配关系，保证了插桩的显式释放内存的指令的正确性；其次在描述对象引用时引入域描述且在插桩时能够主动生成指令获得对象的域引用，从而释放对象域所引用的对象内存空间，达到对整个对象空间的释放；最后为不同对象提供不同的插桩类型，保证可以高效地利用对象生命期分析信息，使得分析出的死亡对象空

间尽可能多的被释放。实验结果表明应用该插桩算法的垃圾收集器能够回收大量内存空间，提高内存空间的利用率和垃圾收集器的执行效率。

References:

[1] David Gay, Bjarne Steensgaard. Fast escape analysis and stack allocation for object-based programs[C]. In :International Conference on Compiler Construction. Berlin: 2000, 82-93.

[2]J.D.Choi, M.Gupta, M.J.Serrano et al. Stack allocation and synchronization optimizations for Java using escape analysis[C]. ACM Transactions on Programming Languages and Systems (TOPLAS), 2003, Volume25(Issue6): 876-910

[3] J.Bogda,U.Holzle．Removing unnecessary synchronization in Java[C].In:Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages and applications. Denver:1999,35-46.

[4] G.Salagnac， S.Yovine， D.Garbervetsky．Fast Escape Analysis for Region-based Memory Management[C]. Electronic Notes in Theoretical Computer Science, 2005, 99-110.

[5] Samuel Z. Guyer, Kathryn S. McKinley, Daniel Frampton. Free-Me: a static analysis for automatic individual object reclamation[C]. In Proceedings of ACM SIGPLAN conference on Programming Language Design and Implementation, Ottawa:2006, {41}: 364-375.

[6] Sigmund Cherem, Radu Rugina.Compile-Time Deallocation of Individual Objects[C].In Proceedings of the International Symposium on Memory Management, Ottawa, Canada, June 2006, 138 – 149.

[7] Sigmund Cherem, Radu Rugina. Uniqueness inference for compile-time object deallocation[C]. Montreal, Quebec, Canada: Proceedings of the 6th International Symposium on Memory Management, 2007, 117-128.

[8] Liu Yu-yu, Zhang Yu. An Object Lifetime Analysis Method Based on Escape Analysis[C]. China National Computer Conference, CNCC 2007. Suzhou.

[9] Wu Ting-peng. Zhang Yu, Liu Yu-yu. Parallel Garbage Collector Based on Just in Time Compiler Assistance[J]. Computer Engineering.

[10]Brendon Cahoon, Kathryn S. McKinley. Jolden Benchmarks. ftp://ftp.cs.umass.edu/pub/osl/benchmarks/jolden.tar.gz. 2001.

[11] Apache Software Foundation. Harmony - Open Source Java SE implementation. http://harmony.apache.org/index.html. 2005.

附中文参考文献：

[8] 刘玉宇，张昱。一种基于逃逸分析的对象生命期分析方法。中国计算机大会 2007，苏州。

[9] 吴廷鹏，张昱，刘玉宇。基于即时编译器辅助的并行垃圾收集器。计算机工程，已录用。