第43卷 第8期 西 安 交 通 大 学 学 报 Vol. 43 №8
2009年8月 JOURNAL OF XI'AN JIAOTONG UNIVERSITY Aug. 2009

Java 程序数据竞争的增量式检测

张昱[1,2]，郝允允[1,2]
（1. 中国科学技术大学计算机科学技术系，230027，合肥；2. 安徽省计算与通信软件重点实验室，230027，合肥）

摘要：针对静态竞争检测无额外运行开销但精度低，而动态竞争检测精度高但因插桩有额外运行开销，在实际 Java 虚拟机的即时编译器中以竞争检测遍形式设计实现了一种精确有效的增量式竞争检测算法. 结合锁集和发生序关系，依次对即时编译的每个方法进行一次方法内分析，收集独立于上下文的方法摘要，并以方法摘要为基础自下而上进行上下文敏感的跨线程方法间分析，增量计算并及时输出潜在的竞争信息. 实验表明，算法对应用程序无插桩且不受程序规模限制，具有与 O'Callahan 等人的动态竞争检测算法类似的精度，检测时间仅占总编译时间的 2%～4%.

关键词：增量式检测；数据竞争；程序分析；锁集；发生序关系
中图分类号：TP311；TP314 **文献标志码**：A **文章编号**：0253-987X(2009)08-0022-06

Incremental Detection of Data Race for Java Programs

ZHANG Yu[1,2], HAO Yunyun[1,2]
(1. Department of Computer Science and Technology, University of Science and Technology of China, Hefei 230027, China; 2. Anhui Province Key Laboratory of Software in Computing and Communication, Hefei 230027, China)

Abstract: Static race detection techniques consume no extra run-time cost but have lower precision, while dynamic ones have higher precision but consume extra run-time cost due to instrumentation. A new precise and efficient algorithm on incrementally detecting potential data races in Java programs is presented, which is implemented as a race detection pass in the just-in-time (JIT) compiler of the Java virtual machine. The algorithm combines lockset-based and happens-before-relation-based detection. Then the algorithm does an intra-method analysis on each method compiled by JIT in turn, and collects summaries independent of the context. The context-sensitive inter-thread analysis is proposed based on the method summaries, to compute incremental race information. The resulting information is output in time. Experimental results show that the algorithm has no instrumentation cost and unlimited program scale, and that the algorithm has the similar precision as O'Callahan, et al's algorithm on dynamic race detection, and consumes only 2%-4% of the total compilation time.

Keywords: incremental detection; data race; program analysis; lockset; happens-before-relation

在多线程程序中，当 2 个线程在无时序约束的情况下访问同一个内存位置并且至少有一个是写访问时，就会导致数据竞争[1]. 数据竞争使多线程程序的执行具有不确定性，难于调试和理解. 所以，研发精确、高效的自动数据竞争检测工具很有价值.

数据竞争检测技术分动态[2-3]和静态[4-6]：动态技术是在程序中通过插桩代码跟踪程序的执行情况，并对执行踪迹进行在线或事后分析来识别潜在竞争；静态技术是通过静态程序分析来识别潜在竞争. 这 2 类技术均使用锁集和/或发生序(happens-

收稿日期：2009-01-29. 作者简介：张昱(1972—)，女，博士，副教授. 基金项目：国家自然科学基金"软件安全性的验证和编译"资助项目(60673126)；Intel 公司研究基金"新并行编程语言特性与范型的设计及可编程性研究"资助项目.

before)关系[7]帮助分析. 采用动态技术能获得变量和别名的准确信息，准确捕获共享内存访问的发生序和/或锁定信息，几乎不报告假竞争(即误报)，但依赖于程序输入，只分析与程序执行路径有关的代码，会漏报一些真正的竞争且开销很大[2]. 静态技术不受程序规模限制，但因静态分析的不可判定性[8]，往往采用不完备的近似算法，所以会报告很多假竞争.

针对动态和静态技术各自的优缺点，结合锁集和发生序关系，本文在 Java 虚拟机(JVM)的即时编译器(JIT)中以竞争检测遍形式设计实现了一种精确、有效的增量式竞争检测算法.

1　数据竞争检测框架

Java 语言中线程对象的 start/join 方法用于启动线程和等待线程执行结束. 线程可通过 synchronized 方法/语句对 Java 对象对应的监视器加锁或解锁. Java 对象由一组简单类型或引用类型的域组成，能被多个线程并发访问的有类的静态域和对象域.

JVM 通过虚拟机核心控制 Java 字节码的载入，启动 JIT 编译待执行方法的字节码得到本地码并管理运行之. Harmony 中的 JIT 采用了流水线编译优化虚拟机核心传来的方法. 流水线将方法的字节码先后翻译成与平台无关的高级中间表示(HIR)和与平台相关的低级中间表示(LIR)，最后发射为本地码. 在 HIR 和 LIR 上会执行一系列优化遍，为了节省空间，JIT 只保存当前编译方法的 HIR 和 LIR.

本文的竞争检测以 HIR 上的遍形式来实现，它不仅需要方法内分析，还需要上下文敏感的跨线程方法间分析. 受制于流水线，竞争检测遍不能得到除当前流经流水线的方法之外的其他方法的 HIR，解决该问题的一种办法是在需要时自行启动流水线编译指定方法，但需频繁启动流水线，开销较大. 当分析大型程序时，递归启动流水线易引起栈溢出，若限制启动层次又会使分析不精确.

综上，本文的办法是，竞争检测遍按方法流经流水线的顺序进行增量检测，并依次对流经流水线的各方法进行一次方法内分析，收集、保存独立于上下文的摘要. 当某方法的摘要完整时(即该方法无调用点或各调用点的摘要均已结合到该方法摘要中)，将该方法摘要自下而上结合到调用者摘要中并增量计算和输出潜在的竞争信息. 一个方法的摘要包括共享内存访问事件信息(包含访问时持有的锁)、start/join 引起的线程发生序关系、尚未分析的调用点信息等. 这种检测可避免频繁启动流水线.

2　数据竞争实例和检测过程

图 1 是一个含数据竞争的 Java 程序示例. 例中主方法 main 创建一个 Example 类实例 e，并调用其实例方法 mb，接着执行一段由锁 e. b 保护的代码块以对 e. a. f 赋值，最后调用实例方法 mc. 方法 mb 创建并启动线程 t_1，然后调用方法 md；方法 mc 创建并启动线程 t_2. t_1 和 t_2 执行相同的代码，见图 1 第 32～第36 行.

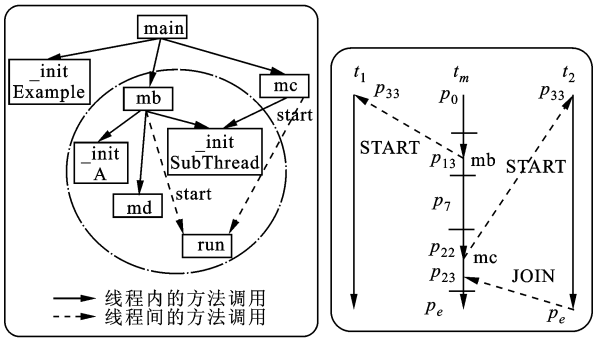
```
1:public class Example{
2:A a,b,c;...
3:public static void
  main(String args[ ]){
4:   Example e=new Example( );
5:   e.mb( );
6:   synchronized(e.b){
7:      e.a.f=0;
8:   }
9:   e.mc( );
10: }
11: public void mb( ){
12:     SubThread t1=new
        SubThread(a,b);
13:     t1.START( );
14:     A aa=new A( );
15:     md(aa);
16: }
17: public void md(A a1){
18:    a1.f=0;
19: }
20: public void mc( ){
21:     SubThread t2=new
        SubThread(a,c);
22:     t2.start( )
23:     t3.join( );
24:     a.f=100;
25: }
26: }//end class Example

27: class SubThread{
28:   A oa, ob;
29:   SubThread(A_oa, A_ob){
30:      oa=_oa;ob=_ob;
31:   }
32:   public void run( ){
33:      synchronized(ob){
34:         oa.f=50;
35:      }
36:   }
37: }
```

图 1　一个含有数据竞争的多线程程序示例

2.1　多线程调用图

示例程序对应的多线程调用图和线程时序图如图 2 所示. 多线程调用图(MCG)是对线程内和线程间方法调用的抽象，记为 G，其中节点对应程序中的方法，实线弧、虚线弧分别表示线程内和线程间的方法调用. 图 2a 为图 1 中以 main 方法为入口的 MCG，记为 G_{main}，点划线所圈部分是以 mb 为入口的 MCG，记为 G_{mb}. 类 Example、A 和 SubThread 的实例初始化方法_init 在创建类实例时调用. 由于线程的 start 方法实际执行的是线程的 run 方法，故图中线程间方法调用弧均指向 run.

竞争检测遍动态地构建、维护应用程序的 MCG，它为已分析的方法 m 维护调用者集合 $S_{\text{caller}}(m)$ 和被调用者集合 $S_{\text{callee}}(m)$ 来保存当前的 MCG. 当对 m 及其调用者 c 进行方法间分析后，就将 c 从 $S_{\text{caller}}(m)$ 中删除，将 m 从 $S_{\text{callee}}(c)$ 中删除. 这种动态维护方法不仅能降低空间开销，而且适用于



(a)多线程调用图 (b)线程时序图

图 2 示例程序对应的多线程调用图和线程时序图

Java 这种因动态类加载等特性导致的“开放世界”环境.

2.2 指向图、域引用关系以及逃逸状态

在方法内分析时,以对象为中心构造方法内的对象引用与其引用的对象间的指向图、对象间的域引用关系,来传播计算对象的逃逸状态. 对象可抽象为

$$n::=\langle \mathrm{nid},\nu,\xi\rangle$$

式中:nid 是对象标识;ν 是引用该对象的变量集合(即别名集合);ξ 是逃逸状态,取满足如下格的 3 种值之一

$$E_{\mathrm{N}}\prec E_{\mathrm{A}}\prec E_{\mathrm{G}}$$

其中:E_{N} 表示未逃逸出当前方法(未逃逸);E_{A}表示通过参数/返回值逃逸出当前方法但未逃逸出线程(参数逃逸);E_{G} 表示逃逸出线程(全局逃逸).

对象间的域引用关系集合 $\mathscr{F}$ 由下面的元组组成

$$\langle n_{\mathrm{from}},\ f,\ n_{\mathrm{to}}\rangle$$

其表示对象 n_{from} 中的域 f 引用对象 n_{to}.

显然,只有全局逃逸的对象才可能发生竞争,而参数逃逸的对象需要结合方法间分析才能进一步确定其是否全局逃逸,这 2 类是竞争检测的关注点.

2.3 访问事件

对全局或参数逃逸的对象 n,在方法内分析时需记录对其域的如下访问事件到集合 $\mathscr{X}(n)$ 中

$$a::=\langle f,\ t,\ p,\ \mathscr{L},\ \omega\rangle$$

式中:f 表示访问的域;t 表示当前线程($\varnothing$表示未知);p 表示访问所在的程序点(2 条指令之间的位置),是访问事件的惟一标识;$\mathscr{L}$ 表示访问时持有的锁集;$\omega\in\{\mathrm{R},\ \mathrm{W}\}$表示访问类型,R 为读,W 为写.

2.4 时序约束

动态竞争检测利用事件的发生序关系来减少假竞争数,在静态检测中引入静态确定的发生序关系有助于改善分析精度[4]. 本文将线程的 start/join 方法调用引起的线程时序约束纳入竞争检测之中,线程时序约束集合 $\mathscr{H}$ 由下面的元组组成

$$\beta::=\langle t_1,\ p_1,\ t_2,\ p_2,\eta\rangle$$

其中时序类型 $\eta\in\{\mathrm{START},\mathrm{JOIN}\}$,表示线程 t_1 执行到位置 p_1 之前的访问先于线程 t_2 执行到位置 p_2 之后的访问.

图 2b 是图 1 对应的线程时序图,其中有主线程 t_{m} 以及分别在方法 mb 和 mc 中启动的线程 t_1 和 t_2. p_i 表示语句 i 对应的程序位置,如 p_{13}、p_{22}分别是启动线程 t_1、t_2 的位置,p_{23}是执行线程 t_2 时执行实例方法 join 的位置,p_{33}是线程 t_1、t_2 的执行入口. 线程执行的入口和出口在未知时分别记为 p_0 和 p_{e}. 3 个线程之间存在 3 条时序约束:$\langle t_{\mathrm{m}},\ p_{13},\ t_1,\ p_{33},\ \mathrm{START}\rangle$;$\langle t_{\mathrm{m}},\ p_{22},\ t_2,\ p_{33},\ \mathrm{START}\rangle$;$\langle t_2,\ p_{\mathrm{e}},\ t_{\mathrm{m}},\ p_{23},\ \mathrm{JOIN}\rangle$.

定义 1 如果 2 个访问事件 α_1 和 α_2 满足以下条件之一,称 α_1 在 α_2 之前发生,记为 $\alpha_1\rightarrow\alpha_2$:①$(\alpha_1[t]=\alpha_2[t])\wedge(\alpha_1[p]<\alpha_2[p])\Rightarrow\alpha_1\rightarrow\alpha_2$,其中 $\alpha_1[t]$表示取元组 α_1 中的属性 t;②假设 $\alpha_1[t]=t_1$、$\alpha_2[t]=t_2$,如果存在时序约束$\langle t_1,\ p_1,\ t_2,\ p_2,\ \mathrm{START}\rangle$且 $\alpha_1[p]<p_1$,$p_2\leqslant\alpha_2[p]$,或者存在$\langle t_1,\ p_1,\ t_2,\ p_2,\ \mathrm{JOIN}\rangle$且 $\alpha_1[p]\leqslant p_1$,$p_2<\alpha_2[p]$,则 $\alpha_1\rightarrow\alpha_2$;③$(\alpha_1\rightarrow\alpha_2)\wedge(\alpha_2\rightarrow\alpha_3)\Rightarrow(\alpha_1\rightarrow\alpha_3)$.

2.5 调用点

当分析某一方法时,若 m 的被调用者尚未分析,则需要记录 m 中尚未分析的调用点信息,调用点信息集合 $\mathscr{I}$ 由下面的元组组成

$$l::=\langle t,\ m,\ p,\ \mathscr{A},\ r,\ \mathscr{L}\rangle$$

式中:m、t 分别为被调用者(方法)及其线程;$\mathscr{A}$ 为实参对象集;r 为返回值接收者.

2.6 方法摘要

m 的摘要(MS)可抽象为

$$\mathscr{M}_m::=\langle\mathscr{N},\mathscr{F},\mathscr{X},\mathscr{I},\mathscr{H},\tau\rangle$$

式中:$\mathscr{N}$、$\mathscr{X}$和 $\mathscr{F}$记录全局和参数逃逸的对象集及其访问事件以及对象间的域引用关系;$\mathscr{I}$记录尚未结合的调用点信息;$\mathscr{H}$、τ 分别记录 G_m 中已创建的线程时序约束和在其中启动且已分析出的线程数. 线程的 run 方法摘要中的 τ 初始为 1,其他方法的 τ 初始为 0.

若 G_m 中各调用弧均已分析,则 $\mathscr{M}_m[\mathscr{I}]$为空并称 $\mathscr{M}_m$ 是完整的. 在本文算法中,只有 $\mathscr{M}_m$ 是完整的,才将其结合到调用者的摘要中并计算竞争信息,才能保证每条调用弧仅分析一次. 当分析某一调用弧时,被调用者的摘要不变,而调用者的摘要被更新,同时该调用弧将从当前的 MCG 中删除.

2.7　数据竞争条件

定义 2　当对象 n 的域 f 的 2 个访问事件 α_1 和 α_2 同时满足：①$\alpha_1[t] \neq \alpha_2[t]$；②$(\alpha_1[\omega]=\mathrm{W}) \vee (\alpha_2[\omega]=\mathrm{W})$；③$\alpha_1[\mathscr{L}] \cap \alpha_2[\mathscr{L}]=\varnothing$；④$\neg((\alpha_1 \rightarrow \alpha_2) \vee (\alpha_2 \rightarrow \alpha_1))$时，称对象 n 的域 f 的访问存在竞争，记作 isConflict(α_1, α_2).

由定义 2 知，仅当对象 n 的域 f 被 2 个不同线程访问时才可能发生竞争. 当结合了 2 个方法的摘要时，若 τ 均为 0，说明 2 个方法中的访问事件均为线程内访问，不构成竞争，无需计算竞争；否则，这些访问事件可能属于不同线程的访问，需要计算竞争. 利用方法摘要中的 τ 属性，能避免在每次方法间分析时都计算竞争.

2.8　增量式竞争检测过程

下面是结合图 1 的增量式检测过程. 经编译后所得摘要如表 1 所示，实施方法间摘要结合后的一些方法摘要如表 2 所示.

假设流水线编译方法的顺序为 main、$\_init_{Example}$、mb、$\_init_{SubThread}$、run、$\_init_{A}$、md、mc. 当编译 main 时，

表 1　经编译后所得摘要

元素	$\mathscr{M}_{main}$
$\mathscr{N}$	$\langle o_e,\{v_e\}, E_A\rangle$, $\langle \varphi_{e.b},\{v_{e.b}\}, E_A\rangle$, $\langle \varphi_{e.a},\{v_{e.a}\},E_A\rangle$
$\mathscr{F}$	$\langle o_e, f_b, \varphi_{e.b}\rangle$, $\langle o_e, f_a, \varphi_{e.a}\rangle$
$\mathscr{X}$	$\mathscr{X}(\varphi_{e.a})=\{\langle f_f, t_m, p_7, \{\varphi_{e.b}\}, W\rangle\}$
$\mathscr{I}$	$\langle t_m, m_{\_init_{Example}}, p_4, \{\cdots\}, o_e, \varnothing\rangle$, $\langle t_m, m_{mb}, p_5, \{o_e\}, \varnothing, \varnothing\rangle$, $\langle t_m, m_{mc}, p_9, \{o_e\}, \varnothing, \varnothing\rangle$
$\mathscr{H}$	$\varnothing$
τ	0

元素	$\mathscr{M}_{mb}$
$\mathscr{N}$	$\langle o_{t_1}, \{v_{t_1}\}, E_G\rangle$, $\langle o_{aa}, \{v_{aa}\}, E_A\rangle$, $\langle \varphi_{this}, \{v_{this}\}, E_A\rangle$, $\langle \varphi_a, \{v_a\}, E_A\rangle$, $\langle \varphi_b, \{v_b\}, E_A\rangle$
$\mathscr{F}$	$\langle \varphi_{this}, f_a, \varphi_a\rangle$, $\langle \varphi_{this}, f_b, \varphi_b\rangle$
$\mathscr{X}$	$\varnothing$
$\mathscr{I}$	$\langle \varnothing, m_{\_init_{SubThread}}, p_{12}, \{\varphi_a, \varphi_b\}, o_{t_1}, \varnothing\rangle$, $\langle t_1, m_{run}, p_{13}, \{o_{t_1}\}, \varnothing, \varnothing\rangle$, $\langle \varnothing, m_{\_init_A}, p_{14}, \{\cdots\}, o_{aa}, \varnothing\rangle$, $\langle \varnothing, m_{md}, p_{15}, \{\varphi_{this}, o_{aa}\}, \varnothing, \varnothing\rangle$
$\mathscr{H}$	$\{\langle \varnothing, p_{13}, t_1, p_0(p_{33}), \mathrm{START}\rangle\}$
τ	0

元素	$\mathscr{M}_{run}$
$\mathscr{N}$	$\langle \varphi_{this}, \{v_{this}\}, E_G\rangle$, $\langle \varphi_{ob}, \{v_{ob}\}, E_G\rangle$, $\langle \varphi_{oa}, \{v_{oa}\}, E_G\rangle$
$\mathscr{F}$	$\langle \varphi_{this}, f_{oa}, \varphi_{oa}\rangle$, $\langle \varphi_{this}, f_{ob}, \varphi_{ob}\rangle$
$\mathscr{X}$	$\mathscr{X}(\varphi_{oa})=\{\langle f_f, \varnothing, p_{34}, \{\varphi_{ob}\}, W\rangle\}$
$\mathscr{I}$	$\varnothing$
$\mathscr{H}$	$\varnothing$
τ	1

注：经编译方法 main、mb、run 得到的摘要分别为 $\mathscr{M}_{main}$、$\mathscr{M}_{mb}$、$\mathscr{M}_{run}$；$v_{变量名}$、$f_{域名}$、$m_{方法名}$分别表示程序中的变量、域和方法的标识.

表 2　实施方法间摘要结合后的一些方法摘要

元素	$\mathscr{M}_{mb}^{(run)}$
$\mathscr{N}$	$\langle o_{t_1}, \{v_{t_1}\}, E_G\rangle$, $\langle o_{aa}, \{v_{aa}\}, E_A\rangle$, $\langle \varphi_{this}, \{v_{this}\}, E_A\rangle$, $\langle \varphi_a, \{v_a, v_{t_1.oa}\}, E_G\rangle$, $\langle \varphi_b, \{v_b, v_{t_1.ob}\}, E_G\rangle$
$\mathscr{F}$	$\langle \varphi_{this}, f_a, \varphi_a\rangle$, $\langle \varphi_{this}, f_b, \varphi_b\rangle$, $\langle o_{t_1}, f_{ob}, \varphi_b\rangle$, $\langle o_{t_1}, f_{oa}, \varphi_a\rangle$
$\mathscr{X}$	$\mathscr{X}(\varphi_a)=\{\langle f_f, t_1, p_{13}(p_{34}), \{\varphi_b\}, W\rangle\}$
$\mathscr{I}$	$\langle \varnothing, m_{\_init_A}, p_{14}, \{\cdots\}, o_{aa}, \varnothing\rangle$, $\langle \varnothing, m_{md}, p_{15}, \{\varphi_{this}, o_{aa}\}, \varnothing, \varnothing\rangle$
$\mathscr{H}$	$\{\langle \varnothing, p_{13}, t_1, p_0(p_{33}), \mathrm{START}\rangle\}$
τ	1

元素	$\mathscr{M}_{main}^{(mb)}$
$\mathscr{N}$	$\langle o_e, \{v_e\}, E_A\rangle$, $\langle \varphi_{e.b}, \{v_{e.b}\}, E_G\rangle$, $\langle \varphi_{e.a}, \{v_{e.a}\}, E_G\rangle$, $\langle o_{t_1}, \{v_{t_1}\}, E_G\rangle$
$\mathscr{F}$	$\langle o_e, f_b, \varphi_{e.b}\rangle$, $\langle o_e, f_a, \varphi_{e.a}\rangle$, $\langle o_{t_1}, f_{ob}, \varphi_{e.b}\rangle$, $\langle o_{t_1}, f_{oa}, \varphi_{e.a}\rangle$
$\mathscr{X}$	$\mathscr{X}(\varphi_{e.a})=\{\langle f_f, t_m, p_7, \{\varphi_{e.b}\}, W\rangle, \langle f_f, t_1, p_5(p_{34}), \{\varphi_{e.b}\}, W\rangle\}$
$\mathscr{I}$	$\langle t_m, m_{mc}, p_9, \{o_e\}, \varnothing, \varnothing\rangle$
$\mathscr{H}$	$\langle t_m, p_5(p_{13}), t_1, p_0(p_{33}), \mathrm{START}\rangle$
τ	1

元素	$\mathscr{M}_{main}^{(mc)}$
$\mathscr{N}$	$\langle o_e, \{v_e\}, E_A\rangle$, $\langle \varphi_{e.b}, \{v_{e.b}\}, E_G\rangle$, $\langle \varphi_{e.a}, \{v_{e.a}\}, E_G\rangle$, $\langle \varphi_{e.c}, \{v_{e.c}\}, E_G\rangle$, $\langle o_{t_1}, \{v_{t_1}\}, E_G\rangle$, $\langle o_{t_2}, \{v_{t_2}\}, E_G\rangle$
$\mathscr{F}$	$\langle o_e, f_b, \varphi_{e.b}\rangle$, $\langle o_e, f_a, \varphi_{e.a}\rangle$, $\langle o_{t_1}, f_{ob}, \varphi_{e.b}\rangle$, $\langle o_{t_1}, f_{oa}, \varphi_{e.a}\rangle$, $\langle o_{t_2}, f_{ob}, \varphi_{e.c}\rangle$, $\langle o_{t_2}, f_{oa}, \varphi_{e.a}\rangle$
$\mathscr{X}$	$\mathscr{X}(\varphi_{e.a})=\{\langle f_f, t_m, p_7, \{\varphi_{e.b}\}, W\rangle, \langle f_f, t_1, p_5(p_{34}), \{\varphi_{e.b}\}, W\rangle, \langle f_f, t_m, p_9(p_{24}), \varnothing, W\rangle, \langle f_f, t_2, p_9(p_{34}), \{\varphi_{e.c}\}, W\rangle\}$
$\mathscr{I}$	$\varnothing$
$\mathscr{H}$	$\langle t_m, p_5(p_{13}), t_1, p_0(p_{33}), \mathrm{START}\rangle$, $\langle t_m, p_9(p_{22}), t_2, p_0(p_{33}), \mathrm{START}\rangle$, $\langle t_2, p_e(p_{34}), t_m, p_9(p_{23}), \mathrm{JOIN}\rangle$
τ	2

注：$\mathscr{M}_{mb}$结合 $\mathscr{M}_{run}$得到的摘要为 $\mathscr{M}_{mb}^{(run)}$；$\mathscr{M}_{main}$结合 $\mathscr{M}_{mb}$、$\mathscr{M}_{mc}$得到的摘要分别为 $\mathscr{M}_{main}^{(mb)}$、$\mathscr{M}_{main}^{(mc)}$.

检测遍对其方法内分析得到 $\mathscr{M}_{main}$（见表 1），其中 $\mathscr{N}$ 包含图 1 中的 e、e. b 和 e. a 引用的对象 o_e 以及非本方法创建的对象 $\varphi_{e.b}$、$\varphi_{e.a}$. 由于 e 作为参数传给 mb 和 mc，故 o_e、$\varphi_{e.b}$ 和 $\varphi_{e.a}$ 均属参数逃逸. $\mathscr{X}$ 包含主线程 t_m

对 $\varphi_{e,a}$ 的域的写访问，$\mathcal{I}$ 包含 p_4、p_5 和 p_9 3 处尚未分析的调用点信息. 由于 main 方法中无 start/join 调用，故 $\mathcal{H}=\varnothing$ 且 $\tau=0$.

当程序执行到 p_5 时，启动编译 mb 得到 $\mathcal{M}_{mb}$（见表 1）. 因变量 t_1 引用线程对象，故 o_{t_1} 全局逃逸. mb 的 4 个调用点中只有 run 方法由线程 t_1 执行，其余均由执行 mb 的线程（$\varnothing$ 表示线程未知）来执行. p_{13} 处的 start 调用使 $\mathcal{H}$ 包含一时序约束，即当前线程运行到 p_{13} 先于对 t_1 的执行.

当程序执行到 p_{13} 时，启动编译 run 得到 $\mathcal{M}_{run}$（见表 1），线程对象 φ_{this} 及其域所引用的对象均全局逃逸. run 方法无调用点，故将 $\mathcal{M}_{run}$ 结合到调用者 mb 的摘要中，得到新的 $\mathcal{M}_{mb}$（见表 2）.

$\mathcal{M}_m$ 结合调用者摘要 $\mathcal{M}_c$ 的具体过程如下.

(1) $\mathcal{N}$ 和 $\mathcal{F}$ 的映射：包括将形参映射到实参（如 $\mathcal{M}_{run}$ 的 φ_{this} 映射到 $\mathcal{M}_{mb}$ 的 o_{t_1}），m 的返回值映射到 c 的返回值接收者，m 中的静态域节点上传到 $\mathcal{M}_c$. 因域引用派生了节点映射（如 $\mathcal{M}_{run}$ 的 φ_{ob} 映射到 $\mathcal{M}_{mb}$ 的 φ_b），故 $\mathcal{M}_m[\mathcal{F}]$ 经节点映射变换上传到 $\mathcal{M}_c[\mathcal{F}]$. 节点映射时需计算逃逸状态，只有参数/全局逃逸的节点才可上传到 $\mathcal{M}_c$.

(2) $\mathcal{X}$ 和 $\mathcal{I}$ 映射：将 $\mathcal{M}_m$ 中的相关元组经节点映射、线程映射和位置变换后上传到 $\mathcal{M}_c$. 线程映射是将调用点的线程标识作为执行 m 的线程标识；位置变换是将 m 中的位置包装成 c 中的位置. 如 $\mathcal{M}_{run}$ 的 $\mathcal{X}(\varphi_{oa})=\{\langle f_f,\varnothing,p_{34},\{\varphi_{ob}\},W\rangle\}$ 映射为 $\mathcal{M}_{mb}$ 的 $\mathcal{X}(\varphi_a)=\{\langle f_f,t_1,p_{13}(p_{34}),\{\varphi_b\},W\rangle\}$，$p_{13}(p_{34})$ 中 p_{13} 为 mb 中的位置，p_{34} 则是实际位置.

(3) $\mathcal{M}_c[\tau]+=\mathcal{M}_m[\tau]$.

(4) 从 $\mathcal{M}_c[\mathcal{I}]$ 中去掉对 m 的调用信息.

当执行到 p_{15} 时，编译 md 得到 $\mathcal{M}_{md}$ 并结合到 $\mathcal{M}_{mb}$，此时测出 $\mathcal{M}_{mb}$ 中的 o_{aa} 向下逃逸到 md 时未逃逸出线程且未逃逸到 mb 的调用者，故 o_{aa} 是线程局部的，可从 $\mathcal{M}_{mb}[\mathcal{N}]$ 中去除. $\mathcal{M}_{md}$ 结合到 $\mathcal{M}_{mb}$ 后，$\mathcal{M}_{mb}$ 在 $\mathcal{M}_{mb}^{(run)}$（见表 2）基础上从 $\mathcal{N}$ 中去掉 o_{aa} 且 $\mathcal{I}=\varnothing$，随后结合到调用者 main 的摘要中，得到 $\mathcal{M}_{main}^{(mb)}$（见表 2）. 由于 $\mathcal{M}_{main}^{(mb)}[\tau]=1$ 且 $\mathcal{X}(\varphi_{e,a})$ 含 2 条对域 f 的写访问，故需检测是否存在竞争；又由于 2 个访问均受 $\varphi_{e,b}$ 保护，故无竞争.

当执行到 p_9 时，编译 mc 并将所得 $\mathcal{M}_{mc}$ 结合到 $\mathcal{M}_{main}$，得到 $\mathcal{M}_{main}^{(mc)}$（见表 2）. 其中，$\mathcal{M}_{main}^{(mc)}[\tau]=2$ 且 $\mathcal{X}(\varphi_{e,a})$ 含4 条对域 f 的写访问，线程 t_m 在 $p_9(p_{24})$ 的写访问与 t_2 在 $p_9(p_{34})$ 的写访问存在时序关系，而与 t_1 在 $p_5(p_{34})$ 的写访问无时序关系，又因 t_m 在 $p_9(p_{24})$ 的写访问与 t_1 在 $p_5(p_{34})$ 的写访问无公共锁集，故存在竞争. 此外，t_1 在 $p_5(p_{34})$ 的写访问与线程 t_2 在 $p_9(p_{34})$ 的写访问无时序关系且没有公共的锁集，故这 2 个访问也存在竞争.

3 算法描述及特殊情况处理

当编译 m 时，竞争检测遍的工作流程如下.

(1) 若无 $\mathcal{M}_m$，则方法内分析 m 形成 $\mathcal{M}_m$. 指向图的构造参见文献[9]，相关语句的处理如图 3 所示.

1) $l_1.f=l_2$：设l_1指向节点o_1且$o_1[\xi]!=E_N$，则增加对域f的写访问事件到$\mathcal{M}_m[\mathcal{X}(o_1)]$.
2) $l_1=l_2.f$：设l_2指向节点o_2且$o_2[\xi]!=E_N$，则增加对域f的读访问事件到$\mathcal{M}_m[\mathcal{X}(o_2)]$.
3) t=newThread()：创建线程标识及线程节点.
4) t.start()：在MCG中增加线程间调用弧，在$\mathcal{M}_m[\mathcal{H}]$中增加START型线程时序约束.
5) t.join()：在$\mathcal{M}_m[\mathcal{H}]$中增加JOIN型线程时序约束.
6) $l_r=\mathrm{op}(l_1,\cdots,l_n)$或$l_r=l_1.\mathrm{op}(l_2,\cdots,l_n)$或$\mathrm{op}(l_1,\cdots,l_n)$或$l_1.\mathrm{op}(l_2,\cdots,l_n)$
如果$\mathcal{M}_{op}[\mathcal{H}]$不存在或不完整，在$\mathcal{M}_m[\mathcal{F}]$中记录该调用点信息，并在MCG中增加线程内调用弧；否则调用combineMS(op, m).

图 3 方法内分析中相关语句的处理

(2) 以 m 为起点自下而上进行方法间分析，如图 4 所示，其中 combineMS(m, c)完成摘要结合和竞争计算（见 2.8 节）.

```
InterMethodAnalysis (G, m){
输入：调用图G、方法m
输出：更新m的调用链上的方法摘要，输出潜在竞争信息
1: if (isIntegrity(m)) { //m的摘要是完整的
       //从G中依次移除m的每个调用者c
2:    while ((c=popCaller (G , m))!=NULL){
3:       combineMS(m , c); //将M_m结合到M_c，计算潜在的竞争
4:       InterMethodAnalysia(G , c); //递归处理c及其调用者
5:    } //end while
6: }// end if
```

图 4 以 m 为起点自下而上的方法间分析

本文对如下情况做特殊处理.

(1)循环：对每个循环体只分析一遍，这不仅可以提高分析效率，而且对分析的精度几乎没有影响.

(2)分支：检测所有可能的执行路径将得到更全面的竞争信息. 本文的方法内分析是部分流敏感的，但方法间分析依赖于程序输入，若分支含方法调用，则仅分析其中一分支的调用弧.

(3)递归调用：因调用环内的方法摘要无法结合到上层调用者摘要中，故本文在方法间分析结束后深度遍历 MCG，若检测出环，则断开其中一条调用弧，继续对环内方法进行方法间分析.

(4)数组：对数组采取保守分析，即把整个数组当作一个对象，把所有数组元素视为数组中的一个域. 这种做法可能会误报竞争，这也是本文误报的主要来源.

4　实验结果与分析

本文在 Harmony 上实现竞争检测遍，实验平台为 Windows XP，CPU 为 AMD Athlon™ 64×2 双核处理器，主频为 3800＋2.0 GHz，内存为 896 MB，测试程序如表 3 所示.

表 3　实验中使用的基准程序

基准程序	代码行数	线程数	含义
tsp	706	3	Traveling Salesman Problem Solver
rayTracer	1 906	4	Java Grande：3D ray tracer
moldyn	1 338	4	Java Grande：molecular dynamics simulation
Monte Carlo	3 605	4	Java Grande：Monte Carlo simulation
SPECjbb	17 495	8	SPECjbb 2005

表 4 列出加入时序约束前后的竞争检测结果，其中分析时间栏为检测总时间，竞争对象数栏为检测到的竞争访问对象数(输出信息可具体到域). 由表 4 可知，增加时序约束对检测时间影响小并能有效减少误报数，如 SPECjbb 在无时序约束时有 6 个竞争错误(均是误报)，而引入时序约束后误报为 0.

表 4　加入时序约束前后的竞争检测结果

基准程序	无时序约束分析		有时序约束分析		
	分析时间/ms	竞争对象数	分析时间/ms	竞争对象数	真正的竞争对象数
tsp	8	4	8	0	0
rayTracer	50	2	29	2	1
moldyn	38	7	46	2	0
Mmonte Carlo	6	3	24	0	0
SPECjbb	7 644	6	7 527	0	0

竞争检测遍对应用程序无插桩，仅占编译时间，不占运行时间. 表 5 列出了竞争检测时间和总编译时间，从中可见检测时间仅为总编译时间的 2%～4%.

表 5　竞争检测时间

基准程序	总编译时间/ms	竞争检测时间/ms	检测时间编译时间之比/%
tsp	319.871	8.364 22	2.6
rayTracer	853.180	29.115 70	3.4
moldyn	1 111.300	46.067 40	4.1
Monte Carlo	1 153.080	24.668 20	2.1
SPECjbb	182 767.000	7 527.360 00	4.1

O'Callahan 等人提出结合锁集和时序约束的动态竞争检测算法[3]，有精度高的优点. 本文在即时编译时结合锁集和时序约束检测，具有静态与动态结合的特点. 表 6 列出本文与文献[3]的实验结果，可见本文算法在精度上与文献[3]相当. 对于 rayTracer 和 moldyn，本文的假竞争数比文献[3]多，但其余的均优之. 本文的分析无运行开销，文献[3]因插桩而影响性能(2.2x～2.9x)，这说明本文方法在保证效率的同时，具有和动态方法相近的精度.

表 6　本文与文献[3]的实验结果对比

基准程序	本文分析			文献[3]分析		
	检测时间/ms	竞争对象数	*B-N-F*	检测时间/ms	竞争对象数	*B-N-F*
tsp	8	0	0	16 870	3	1-0-2
rayTracer	29	2	1-0-1	189 750	1	1-0-0
moldyn	46	2	0-0-2	43 940	0	0
Monte Carlo	24	0	0	5 390	1	0-1-0
SPECjbb	7 527	0	0	—	2	0-1-1

注：B、N 和 F 分别表示真正的竞争数、不影响程序正确性的竞争数、假竞争数.

5　结束语

本文提出了一种基于即时编译的、结合锁集和时序约束的增量式竞争检测算法，并在实际 JVM 上得以实现. 实验表明，本文算法在精度上与文献[3]相当，漏报率和误报率极低且无运行开销，检测时间仅占总编译时间的 2%～4%. 输出的竞争信息可精确到指令位置，便于快速定位错误，可用性强. 由于本文算法仅分析应用程序运行时调用的方法，并且方法内分析是全路径的，故检测部分地依赖于程序输入，可能会漏报竞争，但通过不同的程序输入进行多次检测可减少甚至消除漏报. 此外，算法只分析采用监视器实现的同步情况而未处理其他同步情况，并且对数组采取保守分析，导致算法仍可能误报，这将在下一阶段继续改进.

参考文献：

[1] NETZER R H，MILLER B P. What are race conditions? some issues and formalizations [J]. ACM Letters on Pro-

gramming Languages and Systems, 1992,1(1):74-88.

[2] CHRISTIAENS M, BROSSCHERE K. TRaDe: a topological approach to on-the-fly race detection in Java programs [C]// Proc of 1st Java Virtual Machine Research and Technology Symposium. Berkeley, CA, USA: USENIX Association, 2001:105-116.

[3] O'CALLAHAN R, CHOI J D. Hybrid dynamic data race detection [C]// Proc of the ACM SIGPLAN Symp on Principles and Practice of Parallel Programming. New York, USA: ACM, 2003: 167-178.

[4] HENZINGER T A, JHALA R, MAJUMDAR R. Race checking by context inference [C]// Proc of the ACM SIGPLAN Conf on Programming Language Design and Implementation. New York, USA: ACM, 2004: 1-13.

[5] 吴萍, 陈意云, 张健. 多线程程序数据竞争的静态检测[J]. 计算机研究与发展, 2006, 43(2): 329-335.
WU Ping, CHEN Yiyun, ZHANG Jian. Static data-race detection for multithread programs [J]. Journal of Computer Research and Development, 2006, 43(2): 329-9335.

[6] NAIK M, AIKEN A, WHALEY J. Effective static race detection for java [C]// Proc of the ACM SIGPLAN Conf on Programming Language Design and Implementation. New York, USA: ACM, 2006: 308-319.

[7] LAMPORT L. Time, clocks, and the ordering of events in a distributed system [J]. Communications of the ACM, 1978, 21(7): 558-565.

[8] LANDI W. Undecidability of static analysis [J]. ACM Letters on Programming Languages and Systems, 1992,1(4): 323-337.

[9] SALCIANU A, RINARD M. Pointer and escape analysis for multithreaded programs [C]// Proc of 8th ACM SIGPLAN Symp on Principles and Practices of Parallel Programming. New York, USA: ACM, 2001: 12-23.

(编辑 苗凌)