计算机研究与发展
Journal of Computer Research and Development
ISSN 1000-1239/ CN 11-1777/ TP
43(2): 329~335, 2006

多线程程序数据竞争的静态检测

吴 萍[1] 陈意云[1] 张 健[2]
[1](中国科学技术大学计算机科学与技术系 合肥 230027)
[2](中国科学院软件研究所计算机科学实验室 北京 100080)
(cynthia@ustc.edu.cn)

Static Data-Race Detection for Multithread Programs

Wu Ping[1], Chen Yiyun[1], and Zhang Jian[2]
[1](*Department of Computer Science and Technology, University of Science & Technology of China, Hefei* 230027)
[2](*Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing* 100080)

Abstract Multithreaded concurrent programs are finding wide application, which brings more detrimental data race errors. Traditional static race detection methods are bothered by false positives caused by conservative analysis of concurrent semantics and alias info. In this paper, a precise and effective analysis framework is proposed. The framework applies precise alias analysis and simulates the happen-before order statically. To improve efficiency, an object-based race checker is proposed and compact equality-class-based alias representation is designed. The framework is implemented in a Java compiler—JTool. Through empirical results, the precision and effectiveness of the proposed algorithm are demonstrated.

Key words concurrent programs; program analysis; data race; alias analysis

摘 要 多线程并发程序的广泛使用带来了更多的数据竞争错误. 传统的数据竞争静态检测由于对并发语义和别名信息的保守分析会导致很多假错误. 因此,提出了一个精确有效的静态检测框架:分析应用了精确的别名分析并静态模拟了访问事件发生序;为提高分析效率,检测算法提出了一个以对象为中心,结合Escape分析缩小检测范围的检测算法并配合设计了压缩的别名等价类表示. 检测框架在一个静态Java编译器JTool上做了实现,对于测试程序取得了很好的分析结果.

关键词 并发程序;程序分析;数据竞争;别名分析
中图法分类号 TP311

1 引 言

多线程并发程序的广泛应用带来了更多数据竞争问题. 多线程程序中的数据竞争是指两个或多个线程访问了同一个内存位置而没有时序限制,并且至少有一个是写操作[1]. 竞争有时是共享数据和通信的方式,但是真正的竞争错误很难排除.

竞争检测方法有静态分析[2~4]和动态分析[5]. 动态分析拥有变量和别名的准确信息,但是监测代码"基本都要花费原有程序开销的3X到30X"[5]. 和动态方法相比,静态检测更加全面,可以及早发现错误. 但由于静态分析的不可判定性[6],例如对并发语义的理解、别名信息的判断,静态检测算法都是不完备的近似算法.

为提高分析精度,减少假错误,本文提出了一个精确有效的数据竞争自动检测框架,分析算法将竞争问题分解为跨线程的控制流分析、以全局对象为

收稿日期: 2004-11-08; 修回日期: 2005-05-23
基金项目: 国家自然科学基金项目(60173049, 60421001); 国家杰出青年科学基金项目(60125207)

中心的访问事件搜集以及时序关系的约束求解. 分析算法应用了精确的上下文敏感和流敏感的别名分析; 静态搜集了交互原语形成的访问事件发生序. 为提高分析效率, 和以前以事件为中心的检测算法不同, 算法提出了以对象为中心的检测, 结合 Escape 分析缩小了检测范围(Escape 分析[7] 鉴别出那些除了创建线程, 不会有其他线程访问的对象).

本文组织如下: 第 2 节引入竞争实例, 结合实例介绍竞争信息表示和检测过程; 第 3 节是算法总体描述; 第 4 节是实验结果和分析; 第 5 节是相关问题讨论; 第 6 节是相关研究; 第 7 节是总结.

2 数据竞争实例和检测过程

本节首先引入一个存在竞争的 Java 程序.

```
public class MainThread{
    static Obj p, q, x;//suppose q, x not alias
    public static void main(String args[]){
①ChildThread T1= new ChildThread(q);
②ChildThread T2= new ChildThread(x);
③T1.start(); T2.start();
  Synchronized(p){
④   m1(q);
⑤   q.f= 0;
  }
⑥T2.join();
⑦x.f= 200;
  }
⑧public static void m1(Obj y){y.f= 100;}}
  class ChildThread extends Thread{
  Obj a, b;
⑨ChildThread(Obj o){b= o;}
  public void run(){
  Synchronized(a){
⑩     MainThread.m1(b);
⑪     m2();
   }}
⑫public void m2( ){
⑬  obj z= new obj(…);
⑭  z.g= ..}}
```

图 1 是例子程序和简单的执行时间图. 例子程序共有 3 个线程 *main*, T_1 和 T_2. 主线程创建并启动子线程 T_1 和 T_2 后调用了 $m_1()$ 并对静态变量 q 的域 f 赋值; 然后在语句⑥处等待线程 T_2 的结束. 最后对静态变量 x 的域 f 赋值. 线程 T_1 和 T_2 执行相同的代码, 它们都试图进入由同步对象 $T_1.a$ 和 $T_2.a$ 保护的同步块, 并调用 $MainThread.m_1$ 和实例方法 m_2. 如果 $T_1.a= T_2.a$, 两个子线程就构成互斥同步; 如果 $T_1.a= T_2.a= MainThread.p$, 主线程同步块就和两个子线程同步块互斥, 这正是别名分析是竞争分析主要组成的原因. 图 1 的执行时间图对应 $T_1.a!= MainThread.p$ 的执行实例: 3 条竖实线标记 3 个线程执行, 线程之间的交互用虚线表示, 竞争对用双箭头表示

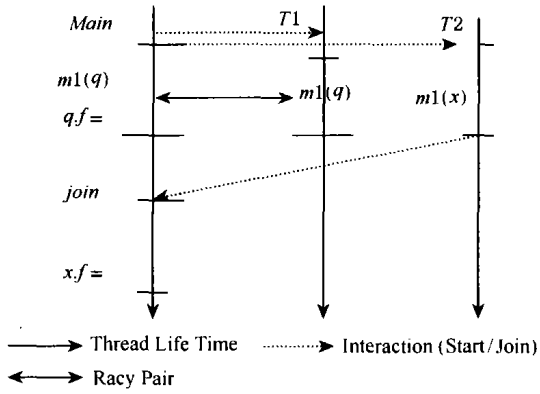


Fig. 1 A racy Java program and execution graph.

图 1 存在数据竞争的 Java 程序和执行图

2.1 跨线程的控制流抽象

跨线程的调用图(inter-thread call graph) 是对过程间、线程间方法调用的抽象. 方法抽象为节点, 调用抽象为从节点之间的有向边. 在 Java 程序中, 主线程的 *main* 函数和其他线程运行体共同构成了 ICG 的入口. 为了使分析过程更加清晰, 同一线程类的不同实体也用不同的线程节点区分. 除了 ICG, 我们还用控制流图(control flow graph) 抽象方法内部的控制流.

图 2 是例子程序对应的 ICG. 线程开始节点表示 *Thread.start()* 调用, 它的目标节点是对应的

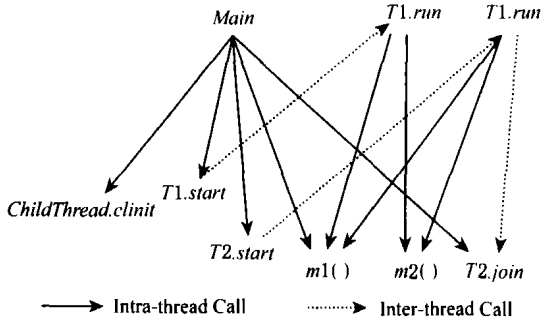


Fig. 2 Inter-thread call graph(ICG).

图 2 跨线程调用图

run( )调用, 节点之间用虚线边相连, 表示一个跨线程的函数调用. 类似的, 主线程中的 $T_2.join()$ 调用也对应着一条从 T_2 节点到它的虚线边. 线程内部的方法调用用实线边表示.

2.2　对象等价类、访问事件和方法概要

以下是用来纪录竞争信息的数据结构:

(1) *AliasSet* ::= 〈*V*, *fieldMap*, *state*, *fieldName*〉

和以事件为中心的竞争分析不同, 我们的分析以对象为中心, *AliasSet* 就是记录对象别名和 Escape 信息的数据结构. 若干别名引用共享一个 *AliasSet* 等价类, 不同 *AliasSet* 形成一个随着程序执行发生变化的列表. 等价类的 union 操作保证每个引用在每个程序点仅对应一个 *AliasSet*, 从而避免了迭代时的不动点操作. *AliasSet* 的元组成员定义如下:

① *V* 为指向当前*AliasSet* 的引用集合;

②*fieldMap* 为从实例域名到对应域值的 *AliasSet* 的映射. 映射只在这个域被访问时展开, 简单类型和 Object 类型的域不对应任何映射;

③*state* 为如果对象可以从一个引用常量访问得到或是通过显式/隐式赋值成为其他线程可访问的状态, 那么 *state* 为 global, 所有从这个节点可达的节点的 *state* 也为 global; 局部对象 *state* 为 local; 函数参数 *state* 为 unkown;

④*fieldname* 为所在对象域名.

(2) *AccessEvent* ::= 〈*location*, *thread*, *AccessType*, *fieldName*, *syncObjSet*〉

global 或 unkown 对象的访问事件定义如下:

①*location* 为按照调用链排列的程序点位置序列, 是访问事件的惟一标识;

② *thread* 为以线程开始位置表示的线程标识;

③*AccessType* 为访问类型, 读或写;

④*fieldname* 为全局对象的域名;

⑤*syncObjSet* 为当前访问拥有的同步对象的 *AliasSet* 集合.

(3) *Method Summary* ::= 〈〈$f_0, f_1, \cdots, f_n$〉, *r*, 〈$g_0, \cdots, g_m$〉, *syncObjCache*〉

Method Summary 是上下文敏感分析用来抽象方法副作用的数据结构. 我们的 *summary* 记录了方法体中所有程序点对参数、返回值、全局对象的访问事件集合, 局部对象访问事件不考虑. *Summary* 滤除了局部对象访问事件搜集, 避免了基于事件的检测算法中对所有对象做盲目的别名分析. *Method Summary* 的元组成员定义如下:

①f_i 为参数的访问事件. 对于实例化方法, f_0 代表 *this* 对象; 对于类方法, f_0 缺省;

② *r* 为返回值的访问事件;

③ g_i 为全局对象的访问事件;

④*syncObjCache* 为记录不同调用上下文中由不同的初始同步对象产生的每个程序点的同步对象集合.

2.3　时序关系的静态抽象

发生序(happen-before) 关系[8]在动态竞争检测中用来比较事件发生先后. 有些线程交互形成的发生序关系其实可以静态确定而且对竞争分析精度有很大影响. 例如简单的 *start*/*join* 原语形成的时序关系就可以静态收集, 并且经常出现主线程生成对象并传递给子线程, 然后主线程不再对它访问, 主线程和子线程中的对象访问并不构成竞争. 所以我们像动态分析一样收集事件发生序关系以提高分析精度. 对于语句 *Si* 和 *Sj*, 定义发生序关系(<)是满足下列条件的最小关系:

①(*Si*.*thread* = *Sj*.*thread*) ∧ (*Si*.*locaion* < *Sj*.*location*) ⇒ *Si* < *Sj*;

②(*Si*.*thread* != *Sj*.*thread*) ∧ (*Si*.*action* = *thread*.*start*) ∧ (*Sj*.*action* = first stmt in run body) ⇒ *Si* < *Sj*;

③(*Si*.*thread* != *Sj*.*thread*) ∧ (*Si*.*action* = *thread*.*join*) ∧ (*Sj*.*action* = last stmt in run body) ⇒ *Sj* < *Si*;

④(*Si* < *Sj*) ∧ (*Sj* < *Sk*) ⇒ *Si* < *Sk*.

为表示访问事件以及它们之间的发生序关系, 我们设计了时序约束图(temporal constraint graph). 图 3 是例子程序的 TCG, 节点代表语句的一次执行, 节点之间的有向边代表执行先后关系. 同一线程中的语句按照调用顺序首尾相连, 不同线程中的语句由于 *start*/*join* 原语也存在先后关系. 图 3 中还用下划线标示了 5 个全局对象访问事件. 通过时序约束图的抽象, 我们把对语句之间发生先后的判断转换成对访问节点在时序图中是否存在一条有向路径的判断:

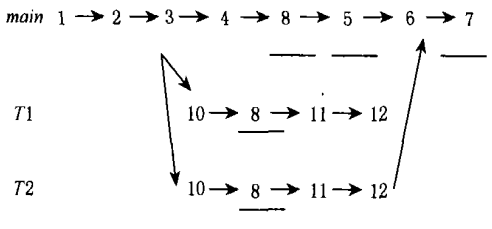


Fig. 3　Temporal constraint graph(TCG).

图 3　时序约束图

2.4 竞争条件定义

基于以上抽象结构可以形式化地定义竞争条件. 全局量 g 访问集合中的两个事件 $AE1$ 和 $AE2$, 如果它们访问了 g 的同一个域, 那么它们构成竞争对当且仅当同时满足以下 4 个条件:

①$\lceil(AE1<AE2)\wedge\lceil(AE2<AE1)$;

②$AE1.thread!=AE2.thread$;

③$AE1.AccessType=WRITE\vee AE2.AccessType=WRITE$;

④$AE1.syncObjSet\wedge AE2.syncObjSet=\varnothing$.

2.5 分析过程介绍

下面结合实例介绍算法的工作过程:

首先, 构造 ICG 和 CFG, 通过保守 *Thread.start* 计数分析程序是否是单线程, 如果是则不存在竞争对. 构造 CFG 的同时还得到了所有事件的时序关系, 例如例子程序形成的 TCG 如图 3 所示.

接下来搜集每个程序点的同步对象, 并记录在当前方法 *Method Summary* 的 *syncObjCache* 中. 我们从每个线程节点开始对 ICG 和 CFG 自顶向下做深度优先遍历, 收集不同调用上下文和不同控制流点的锁集合. 对例子程序分析后得到 $main:\langle\varnothing,\varnothing,\varnothing,\{\{\}\xrightarrow{}1\{\}\ 2\{\}\ 3\{\}\ 4p\ 5p\ 6\{\}\ 7\{\}\}\rangle$, $T1.run:\langle\varnothing,\varnothing,\varnothing,\{\{\}\xrightarrow{}10\{T1.a\}\ 11\{T1.a\}\}\rangle$, $T2.run:\langle\varnothing,\varnothing,\varnothing,\{\{\}\xrightarrow{}10\{T2.a\}\ 11\{T2.a\}\}\rangle$, $m1:\langle\varnothing,\varnothing,\varnothing,\{\{4p\}\xrightarrow{}8p;\{3:10T1.a\}\xrightarrow{}8T1.a;\{3:10T2.a\}\xrightarrow{}8T2.a\}\rangle$, $m2:\langle\varnothing,\varnothing,\varnothing,\{\{3:11T1.a\}\xrightarrow{}13T1.a\ 14T1.a;\{3:11T2.a\}\xrightarrow{}13T2.a\ 14\ T2.a\}\rangle$. 由于还没有做访问事件的信息收集, 摘要元组前 3 项为空. *main* 函数调用仅有一次并且不存在输入锁, 所以它的摘要中 *syncObjCache* 只有一个映射, 即输入的同步对象为空的情况, 这个映射记录了 *main* 函数的 7 条语句各自拥有的同步对象. *T1.run*, *T2.run*, *m1*, *m2* 的 *syncObjCache* 与 *main* 函数类似.

接着为各个线程节点收集全局量访问信息. 首先在 ICG 上自底向上对过程内部语句分析, 收集全局对象访问事件; 如果出现函数调用语句, 再将上下文信息向下传播. 这是个跨过程上下文敏感分析, 被调用函数记录别名信息和访问事件形成摘要, 传播给调用函数后结合调用上下文对摘要做分析. 局部于线程的对象不参与任何竞争, 所以摘要只记录 global 对象的访问事件, 这可以看做是 Escape 分析的一个扩展; 输入/输出参数的状态要带入上下文决定, 所以摘要中也记录下对它们的访问. 上下文不敏感分析将所有调用处信息合并起来传播给所有的被调用函数, 并将所有返回处的信息传播给所有的调用函数. 例如例子程序对 $m1()$ 的 3 次调用, 上下文不敏感分析的结果是 3 次调用都访问了 $q.f$ 和 $x.f$, 这无疑会带来很多的假错误.

过程内部采用了流敏感的别名分析, 每条语句按照转换函 $Out=Gen+(In-Kill)$ 更新 *AliasSet* 列表、记录访问事件. 流敏感分析记录了单个线程控制流顺序带来的影响, 减少了假竞争对. 比如局部对象 l 在逃逸成为全局量之前对它的访问不会形成任何竞争对, 而流不敏感分析会合并所有程序点状态, 引入假全局访问. 流敏感分析也有和并发语义冲突的地方, 例如对静态域 $l1.f$ 的赋值: $l1.f=l2$ 使得局部量 $l2$ 逃逸, 对 $l1.f$ 的再次赋值无法保证在 $l2$ 逃逸期间没有其他线程通过 $l1.f$ 引用 $l2$, 所以这时不做 *kill* 操作.

经过访问信息搜集形成了各个函数摘要: $m1:\langle\{f1:8,,write,f,\{\}\rangle,\varnothing,\varnothing,\ldots\rangle$, $m2:\langle\varnothing,\varnothing,\varnothing,\ldots\rangle$, $main:\langle\varnothing,\varnothing,\langle MainThread.q:\langle 4:8,main,write,f,\{p\}\rangle,\langle 5,main,write,f,\{p\}\rangle;MainThread.x:\langle 7,main,write,f,\{\},\ldots\rangle\rangle\rangle$, $T1.run:\langle\varnothing,\varnothing,\langle MainThread.q:\langle 10:8,T1,write,f,\{T1.a\}\rangle\rangle,\ldots\rangle$, $T2.run:\langle\varnothing,\varnothing,\langle MainThread.x:\langle 10:8,write,f,\{T2.a\}\rangle\rangle,\ldots\rangle$.

最后收集每个线程节点的全局量访问事件, 并根据竞争条件给出竞争对. 对于例子程序, 如果 $T1.a$ 和 p 不指向同一个内存单元, 那么主线程对 $q.f$ 的两次访问和 $T1$ 对 $q.f$ 的访问构成竞争. 主线程对 $x.f$ 的访问和 $T2$ 对 $x.f$ 的访问没有构成竞争, 因为 *join* 原语分割了两个访问事件.

3 算法描述

(1) 首先扫描整个程序构造 ICG, CFG 和 TCG. 分析程序是否是单线程程序, 如果是单线程程序则退出分析.

(2) 对 ICG 和 CFG 自顶向下做深度优先遍历, 搜集每个程序点的锁对象集合并记录在当前方法摘要的 *syncObjCache* 中:

```
foreach threadroot r in ICG.roots
    DFS-traverse-fn(r,{});
DFS-traverse-fn(fn, ls){
```

```
        foreach Map x in syncObjCache
        if(x.entry-lockset == ls) return;
        x = new Map(); x.entry-lockset = ls;
        x.exit-locksets = DFS-traverse-stmts(fn,
        fn.entry, ls, ls);
        fn.summary.syncObjCache.add(x);
        return;
}
DFS-traverse-stmts(fn, s, entry-ls, ls){
        if (s is end-of-path) return;
        if (s is lock) ls.add-lock(s);
        else if (s is unlock) ls.remove-lock(s);
        if s is call DFS-traverse-fn (s.fn, ls);
        if s is normal stmt set.add(s.Location, ls);
        DFS-traverse-stmts(s.next);}
```

(3) 为各个线程节点收集全局量的访问信息:

```
partition ICG into SCC sets
foreach SCC in bottom-up order{
  foreach method m in the SCC {
    if methodcall{
      unify scc or apply callee's summary
    }else{
      analyze statement according to table 1's
      transfer function *
    }}}
```

Table 1　Statements' Transfer Function

表 **1**　语句转换函数

IR stmt	Transfer Function
Move $l = m$ ($l = g, g = m$)	$AS(l).v - l; AS(m).v + 1$
Load $l1 = l2.f$ (f is static/nonstatic field)	$AS(l1).v - l1; AS(l2.f).v + l1$; Record AE in summary
Store $l1.f = l2$ (f is nonstatic)	$AS(l1.f).v - l1.f; AS(l2).v + l1.f$
Store $l1.f = l2$ (f is static)	$AS(l2).v + l1.f; AS(l2).state = global, propagate;$ Record AE in summary
Call $l = l0.op(f_1 \cdots f_k)$	Apply real params into summary
New $l =$ *new cls();*	Add $AS(l)$ to AliasSet List

(4) 按照竞争定义对每对线程节点(r_1, r_2)记录的全局变量 g 的访问事件做出竞争判断.

算法最坏时间/空间复杂度至少是程序大小的指数级. 上下文敏感分析通过区分不同调用路径对被调用函数做分析, 它的最坏时间/空间复杂度是调用链深度的指数级. 循环/递归结构的存在使得流敏感分析的迭代次数在最坏情况下也是语句的指数级.

4　实验结果

我们在 MIT 的一个 Java 编译器 FLEX[9] 上构造了 JTool 分析框架. 实验环境是 P Ⅳ 2.66GHz, 256MB 内存的 Linux 系统, 运行系统是 jdk1.4.2-04. 表 2 列出了我们使用的 6 个测试程序: Strsplit 是 CodeProject 网站[10] 提供的两个单线程程序. Philo, elevator, sor 是文献[3]中用到的 3 个多线程测试程序. Philo 是一个简单的哲学家进餐的实现. Elevator 是一个电梯的实时离散事件模拟器; sor 是一个并行的科学计算程序; MainThread 是例子程序. 表 2 描述了测试程序的属性, 前 3 项分别是应用程序大小、类数目和链接库数目.

Table 2　Benchmark Programs

表 **2**　测试程序属性

Program	Application Size(loc)	Application Class	Lib Class	Method Num	Bytecode Num	User Thread
Strsplit	38	1	12	46	135	1
Multiset	70	2	5	13	75	1
Philo	81	2	129	192	3605	2
Elevator	528	5	142	311	6820	2
Sor	300	7	132	205	4483	3
MainThread	41	3	6	19	79	3

表 3 是实验结果. 由于静态检测是保守分析, 我们用竞争对数目衡量检测精度[2~4], 一般来说精度越大竞争对越少. 从表 3 看到, 对于两个单线程程序, 我们得到了正确的分析结果且用时较少. 由于没有看到其他应用上下文敏感和流敏感分析的算法报道, 惟一知道的和本文精度最接近的算法是 Praun 等人的文献[3], 所以在表 3 中我们和文献[3]的 3 个测试程序进行了比较: 对于 Philo, Elevator, Sor, 我们没有产生任何假错误. 这个结果和文献[3]的实验数据相比较(运行环境是 GNU libgcj version 2.96, P Ⅳ 1.4GHz), 分析时间是 X10 的关系, 但是错误率要小得多, 特别是 Elevator 程序. 由于文献[3]也通过符号执行搜集了事件发生序, 我们认为精度的提高主要来自别名分析. 最后对于存在竞争的例子程序, JTool 也得到了正确的竞争对.

Table 3 Experiments Result
表 3 实验结果

Program	Real Racy Pair	Reference [3]		JTool	
		Racy Pair	Analysis Time(s)	Racy Pair	Analysis Time(s)
Strsplit	0	–	–	0	1.49
MultiSet	0	–	–	0	1.93
Philo	0	2	0.8	0	4.88
Elevator	0	33	1.5	0	20.01
Sor	0	2	018	0	5158
MainThread	2	–	–	2	01954

为了有针对性的测试别名分析、时序分析和 Escape 分析对算法的影响, 我们对测试程序分别不做别名, 只按名称判断是否同一对象(I); 不做时序分析(II); 做别名但不做 Escape 分析进行测试(III)1 表 4 是 I, II, III 的分析结果1 从表 4 中可以看到别名分析对竞争精度的影响是最大的, 而且这种影响会随着程序规模的增大而更加突出1 时序分析只影响了 Sor 的分析结果, 但是我们相信它还是可以找到特定模式下的竞争1 最后, Escape 分析也提高了分析精度, 由于没有基于事件的上下文敏感和流敏感分析数据, 我们没有比较 Escape 分析对算法效率提高的影响1

Table 4 Specific Analysis I, II, III
表 4 针对性分析 I, II, III

Program	Real Racy Pair	Racy Pair		
		I	II	III
Strsplit	0	0	0	0
Multiset	0	0	0	0
Philo	0	5	0	0
Elevator	0	242	0	4
Sor	0	0	6	0
MainThread	2	2	2	2

5 竞争和 Escape 分析及内存一致性模型

从数据流分析的角度, 竞争分析和 Escape 分析都可以看做别名分析的扩展1 通过 Escape 分析可以预先过滤那些不会参与竞争的局部对象访问事件, 我们正是在这个结论的基础上构造了以对象为中心的分析算法1 从分析目标的角度, 这两个问题又是相互对立的1 竞争是错误检测, Escape 分析是程序正确假定上的优化1

和顺序程序不同, 并发程序执行依赖底层内存模型定义它在共享内存环境下的合法输出1 内存系统可以在符合一致性模型的要求下重排读写操作顺序1 竞争问题和内存模型紧密相关1 一方面, 不同的一致性模型下竞争分析结果会不同, 在顺序一致性模型下没有竞争的程序在弱一致性的并发模型上可能出现竞争; 另一方面, 竞争分析结果可以被用来安全的优化特定的内存访问, 没有参与竞争的内存访问可以遵循较为松散的内存模型1 我们的分析建立在顺序一致性模型上, 这也是多数并发分析的假定模型1

6 相关研究

文献[2]是第 1 个没有利用类型分析、对面向对象并发程序做自动分析的静态工具1 它以访问事件为中心, 分别对特定路径和所有路径做 must 和 may 的别名分析得到确定竞争和可能竞争的对象对1 由于应用保守的流不敏感分析, 也没有抽象 startPjoin 原语, 8 个很小的测试程序, 确定竞争数为 0, 可能的竞争数却很多1

文献[4]对 C 语言的竞争按模式匹配检测1 对非常大的系统软件, 它的分析速度很快1 但是它没有别名分析, 只是根据经验对简单分析产生的竞争对按照可能性等级排列, 所以准确度比较低1

以前的检测算法都是以事件为中心并且不会利用对象状态简化别名分析过程, 我们惟一知道的和本文最接近的分析算法是文献[3]1 文献[3]的分析也是在对象层次的, 但在数据结构和算法设计上和我们完全不同1 文献[3]构造了复杂的 OUG(object use graph)抽象不同线程对同一对象的访问事件, 每个全局对象都必须各自对所有方法的 CFG 做符号执行, 删除和该对象无关的操作后才能得到自己的 OUG1 OUG 相当于我们的别名信息和时序约束抽象的综合, 不同 OUG 的遍历收集中会形成很多重复的控制流信息和数据流信息, 但由于应用了流不敏感分析, 文献[3]的开销还是要小于我们的算法1 从第 4 节的实验结果可以看出, 它的分析效率很高, 但精确度低于我们的算法1

7 结 论

我们提出了一种自动精确的数据竞争静态检测方法1 分析过程应用了流敏感、上下文敏感别名分

析, 充分考虑了时序关系对竞争问题的影响1 分析以对象为中心做信息搜集, 通过精简的对象模型并结合 Escape 分析提高了分析效率1 实验结果表明, 算法可以在有效的时间内对中小规模的程序取得精确分析结果, 精确度超过了现有算法1

算法实现的测试程序规模还很有限, 需要进一步对实际应用规模的程序进行测试分析; 最好提供区分竞争错误和良性竞争的机制1

参 考 文 献

1 R1 H1 Netzer, B1 P1 Miller1 What are race conditions? Some issues and formalizations1 ACM Letters on Programming Languages and Systems, 1992, 1(1): 74~ 88

2 J1-D1 Choi, A1 Loginov, V1 Sarkar1 Static datarace analysis for multithreaded object-oriented programs1 IBM Research, Tech1 Rep1: RC22146, 2001

3 C1 Praun, T1 Gross1 Static conflict analysis for multi-threaded object-oriented programs1 In: Proc1 ACM SIGPLAN 2003 Conf1 Programming Language Design and Implementation1 New York: ACM Press, 2003: 115~ 128

4 Dawson Engler, Ken Ashcraft1 RacerX: Effective, static detection of race conditions and deadlocks1 ACM Symposium on Operating Systems Principles. New York: ACM Press, 2003: 237~ 252

5 J1 Choi, K1 Lee, A1 Loginov, *et al*1 Efficient and precise datarace detection for multithreaded object-oriented programs1 In: Prod ACM SIGPLAN 2002 Conf1 Programming Language Design and Implementation1 New York: ACM Press, 2002: 258~ 269

6 W1 Landi1 Undecidability of static analysis1 ACM Letters on Programming Languages and Systems, 1992, 1(4): 323~ 337

7 Erik Ruf1 Effective synchronization removal for Java1 In: Proc. ACM SIGPLAN 2000 Conf1 Programming Language Design and Implementation. New York: ACM Press, 2000. 208~ 218

8 L1 Lamport1 Time, clocks, and the ordering of events in a distributed system1 Communications of the ACM, 1978, 21(7): 558~ 565

9 Martin Rinard. The flex program analysis and compilation system. http://www.flex-compiler.csail.mit.edu, 1999-06-10

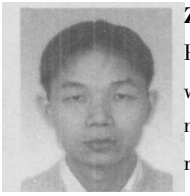
10 http://www1codeproject1com/, 2004

Wu Ping, born in 1978, Ph. D. Received her B1 A. s degree in computer science from the University of Science and Technology of China1 Her main research interests include program analysis, software engineering1

吴萍, 1978 年生, 博士, 主要研究方向为程序分析、软件工程1

Chen Yiyun, born in 1946, Professor and Ph1 D1 supervisor of the University of Science and Technology of China1 His main research interests include formal methods, theory of program language and software architecture1

陈意云, 1946 年生, 教授, 博士生导师, 主要研究方向为形式化方法、程序设计语言理论、软件系统结构1



Zhang Jian, born in 1969, Professor and Ph1 D1 supervisor of the Institute of Software, the Chinese Academy of Sciences1 His main research interests include automatic reasoning, constraint solving and formal methods1

张健, 1969 年生, 研究员, 博士生导师, 主要研究方向为自动推理、约束满足和形式化方法1

Research Background

Traditional static race checkers are bothered by many false positives caused by conservative analysis of concurrent semantics and alias information1 We propose an automatic, precise and effective algorithm for static race checker1 The analysis applies precise alias analysis for concurrent programs, and collects temporal constraints for access events statically1 We propose an object-based race checker compared with the previous event-based algorithm1 The object-based race checker, combined with escape analysis and compact object model, improves the algorithm. s performance effectively1 Experimental result shows our algorithm can get a precise result for medium-small scale benchmark programs with a reasonable time expense, and the precision exceeds current algorithms1 Our work is funded by the National Natural Science Foundation of China (60173049, 60421001) and the National Science Fund for Distinguished Young Scholars of China (60125207)1