并发 Java 程序同步操作的有效删除*

吴 萍[1+], 陈意云[1], 张 健[2]

[1](中国科学技术大学 计算机科学技术系,安徽 合肥 230027)

[2](中国科学院 软件研究所 计算机科学重点实验室,北京 100080)

Effective Synchronization Removal in Concurrent Java Programs

WU Ping[1+], CHEN Yi-Yun[1], ZHANG Jian[2]

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

[2](Laboratory of Computer Science, Institute of Software, The Chinese Academy of Sciences, Beijing 100080, China)

+ Corresponding author: Phn: +86-10-62562796, E-mail: cynthia@mail.ustc.edu.cn, http://www.iscas.ac.cn

Received 2004-05-18; Accepted 2005-03-11

Wu P, Chen YY, Zhang J. Effective synchronization removal in concurrent Java programs. *Journal of Software*, 2005,16(10):1708−1716. DOI: 10.1360/jos161708

Abstract: Synchronization operations make a huge expense for concurrent Java programs. This paper proposes an effective and precise static analysis algorithm for the redundant synchronization removal. The algorithm consists of two phases-basic analysis and inter-thread temporal analysis. Both phases take the effect of control flow relation and thread control relation into count. This paper also constructs a Java compiler-JTool and implements the algorithm on it. To deterministic single-threaded programs, the removal ratio reaches 100% and to multi-threaded programs, the removal ratio is higher than the existing analysis tools.

Key words: concurrent program; program analysis; escape analysis; synchronization removal

摘 要: 同步操作是并发 Java 程序非常大的一部分开销.在现有程序分析方法的基础上,提出了一种精确而有效的冗余同步操作的静态删除方法.该方法分为基本处理和线程间时序分析两个阶段,充分考虑了控制流结构和线程交互时序对同步删除的影响.构造了一个 Java 编译器 JTool,并在其上实现了同步删除算法.对于确定的单线程程序,同步删除率达到 100%;对于多线程程序,同步删除率高于现有的分析工具.

关键词: 并发程序;程序分析;escape 分析;同步删除

中图法分类号: TP311 文献标识码: A

Java 实现了语言级的并发支持.在应用程序,特别是一些服务器程序中,用户可以很方便地使用 Java 提供的同步结构(synchronized 方法和块)和同步原语对共享数据做安全操作.这种内建的并发支持还体现在类库

* Supported by the National Natural Science Foundation of China under Grant Nos.60173049, 60421001 (国家自然科学基金); the National Science Fund for Distinguished Young Scholars of China under Grant No.60125207 (国家杰出青年科学基金)

作者简介: 吴萍(1978−),女,安徽东至人,博士生,主要研究领域为程序分析,软件工程,形式化方法;陈意云(1946−),男,教授,博士生导师,主要研究领域为程序设计语言理论和实现技术,形式化描述技术,软件安全;张健(1969−),男,博士,研究员,博士生导师,主要研究领域为自动推理,约束求解,形式化方法.

中.Java 提供了存在大量同步操作的 thread-safe 库,比如 Vector 类以及部分 Swing API,程序可以在多线程环境中安全地使用这些库所提供的功能.

粗粒度的保守锁机制容易写出正确的代码,但也带来了大量的同步操作.同步操作是多线程程序一个非常大的开销,对于多处理器系统,它意味着流水线的锁定,所有线程必须等待同步结束.“同步开销通常占总体执行时间的 5%~10%”[1],在对 Marmot[2]的测试中,“5 个中等大小的单线程程序在同步上花费了 26%~60%的时间”.因此,人们越来越意识到同步优化的重要性.

为了减小同步开销,有些方法是直接改良同步原语(primitive)的动态实现,比如 thin lock[3].对于程序分析来说,则是通过静态的自动分析消除不必要的同步操作.确切地说,只要在线程 T 对对象 O 的同步操作期间没有其他线程 T'对 O 做同步,那么 T 中的同步就可以删除.现有的同步优化方法都是这个同步删除条件的某种保守近似.比如,有的方法分析程序是否只有一个线程,如果是单线程程序,则不需要做同步,这种方法的缺点是无法处理多线程程序.Java 应用程序中经常有些单线程程序使用 JDK 中的帮助线程,这样,程序是多线程的,但其实帮助线程并不和主线程共享数据.还有所谓的 Escape 分析*,它通常是鉴别出那些在程序执行中除了创建线程,不会有其他线程访问的对象(thread-local 对象),这些对象是不需要同步的.比较精确的 escape 分析一般开销都很大,而且对单线程或是同步数据结构存储在静态变量中的程序处理结果很差.例如,“现有 escape 分析系统对单线程程序的平均同步移出率低于 55%”[4].

从同步删除条件的定义可以看到:同步对象即使逃离了当前线程并可以被其他线程访问,只要在其他线程中没有被同步,或是同步有时间先后,比如通过 start/join 原语限定了执行顺序,那么原线程中的同步操作也是可以去掉的.所以,我们对保守的 escape 分析做进一步扩展:首先,应用一个基本处理过程删除对那些只被单线程同步的对象的同步操作.我们分析了对象的 escape 信息并跟踪 escape 对象在不同线程中的流动:即使是 escape 对象,若只被单线程同步那么也可以删除对它的同步操作.这个处理过程对单线程、使用帮助线程或是调用 thread-safe 类库的程序分析简单而有效,并且在分析精度上超过了现有的 escape 分析.接下来,对剩下的对象做线程间的时序分析,继续寻找可以消除的同步操作.

本文第 1 节介绍算法中用到的基本数据结构.第 2 节结合程序实例介绍算法.第 3 节是算法的系统描述.第 4 节是实验结果和分析.第 5 节是相关研究.第 6 节是总结.

1 用于程序抽象的数据结构

程序分析通过对程序执行状态的抽象和模拟计算程序点具有的属性,因此它需要一些数据结构抽象控制结构和数据信息.本节介绍算法使用到的主要数据结构.线程间调用图是跨线程的控制流抽象,是应用数据流分析的最高层表示.控制流图反映了过程内部的控制流,它和线程间调用图一起构成了程序分析框架.Alias set 是记录对象信息的基本单位.Method summary 则反映了过程调用对数据信息的修改.

1.1 线程间调用图ICG(inter-thread call graph)和控制流图CFG(control flow graph)

跨线程的调用图是对过程间、线程间方法调用的抽象.它记录了所有方法调用(包括同步方法,join,start 和 init 函数)的位置.程序中的方法抽象为图中的节点,调用关系抽象为从调用方法节点到被调用方法节点的有向边.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多线程程序 ICG 的入口.为了使分析过程更加清晰,即使同一线程类的不同实体也用不同的线程节点区分.除了 ICG,我们还用控制流图抽象方法内部的控制流,它的基本处理单元是语句.ICG 和 CFG 共同构成了数据流分析的框架基础.

图 1 是第 2.2 节中的实例程序对应的 ICG.图中节点显示了所有的方法调用.线程开始(start)节点表示 Thread.start()调用,它的目标节点是对应的 run()调用,它们之间用虚线边相连,表示一个跨线程的函数调用.类似

* Escape 分析的范围很广:有分析逃离当前方法的,应用的优化为对象的栈上分配;有分析逃离当前线程的,应用的优化为同步删除;有分析逃离当前分析区域的,应用的优化为特定区域上的对象分配.我们的分析目标是同步删除,即分析那些逃离了当前线程、可以被其他线程访问的对象,又叫做 escape 对象.

地,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则用实线边表示.

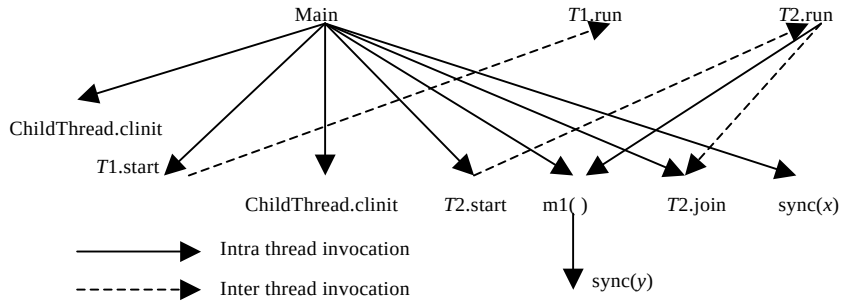

Fig.1 ICG example

图 1 ICG 实例

1.2 Alias set和method summary

1.2.1 *alias set*::=⟨*fieldMap, synchronized, syncThreads, escaped*⟩

Alias set 记录了对象的各种信息,由于描述的是对象值,无须考虑简单类型(primitive type).Alias set 是一个元组的等价类表示,若干别名引用共享一个 alias set,元组成员定义对象的各个属性:

- *fieldMap* 从实例域名到对应域值的 alias set 的映射.映射只在这个域被访问时展开,简单类型或 Object 类型域不对应任何映射,域名$ELT 表示一个数组对象的内容.
- *synchronized* 布尔值,如果该值是同步操作的目标则为真.
- *syncThreads* 只有对于 escaped 为真的对象,才记录可能同步该值的线程实例集合,线程实例用创建位置表示.
- *escaped* 布尔值,如果对象可以从一个引用常量(静态域)访问得到或是由于显式/隐式赋值(参数传递)成为多线程可访问的状态,escaped 取真;所有从 Escape 节点可达的节点传递得到 escaped 也取真.

1.2.2 *Method Summary* ⟨⟨f_0,f_1,…,f_n⟩,r,e⟩

Method summary 是上下文敏感分析用来抽象方法副作用(side-effect)的数据结构,也就是方法的转换函数.Method summary 记录了方法体内每条语句带来的参数、返回值、异常值的同步和别名信息.在每个调用位置带入上下文信息就得到这次调用对各个对象值状态的影响.Method summary 也是一个元组表示,元组成员定义如下:

- f_i 形参对应的 alias set,对于实例化方法,f_0 是 this 对象的 alias set;对于类方法,f_0 缺省.
- r 返回值的 alias set.
- e 异常对象的 alias set.

1.3 Alias set图

为了组织所有别名集合,我们引入了 alias set 图.Alias set 图与 points-to 图类似,只是它以实例对象为中心.它记录了引用池中的每个引用到可能指向的包含有数据流信息的对象值的映射.互为别名的对象指向同一个 alias set.从 alias set 图中可以搜集 alias set 的索引、指向它的别名引用集合等信息.Alias set 图随着程序分析过程不断变化,不同程序点都对应着各自的 alias set 图.由于分支结构的存在,引用可以指向多个对象.

2 分析过程及实例说明

下面我们结合程序实例说明分析算法是如何工作的.

2.1 基本处理阶段

传统的 escape 分析对单线程以及同步数据结构存储在静态变量中的程序处理结果很差,而分析程序是否只有单个线程适用面很窄.在基本处理阶段,我们将它们结合起来,并对传统的 escape 分析作进一步的扩展.在扩

展部分,我们借鉴了文献[4]的方法——不仅分析对象的 escape 信息,而且跟踪对象在不同线程中的流动,并记录对它做同步操作的线程信息.即使是 escape 对象,若只被单线程同步,那么也可以删除对它的同步操作.我们的算法采用了流敏感(flow-sensitive)的数据流分析,考虑到了过程内部语句的执行顺序对数据流信息的影响,所以分析精度超过了文献[4].下面我们结合具体实例对基本处理阶段的算法作一个简单描述.

首先,扫描程序并构造 ICG 和 CFG.统计程序的线程个数,如果是单线程程序则删除所有同步操作.

接下来,利用调用图和控制流图为 escape 对象扩展访问的线程和同步信息.首先在 ICG 上自底向上分别对单个过程作数据流分析,计算每个程序点对象的 alias set.这些信息构成了一个 alias set 图,别名引用则共享一个 alias set.Alias set 随着语句的执行发生变化,对应各条语句的转换函数(transfer function)详见第 3 节算法介绍.

当过程中出现函数调用语句时,信息就开始自底向上传播.我们的跨过程数据流分析是上下文敏感(context-sensitive)分析,使用可重用的多态 method summary 记录每个方法对数据信息的修改,以识别不同的调用上下文.我们把对被调用函数分析得到的 method summary 和全局对象的 alias set 传播给调用函数,然后结合当前方法的上下文信息(参数和全局变量的 alias set),在当前方法中继续作过程内部分析.

下面的程序说明了通过上下文敏感分析,同步删除精度的提高.

```
1 public class Test{
2           static Data data= … ;
3           static Data f1(Data a) {
4                 Data o=new Data();
5                 a=o;
6                 return o;
7           }
8           static void f2(){
9                 Data o_f2=f1(Data o2) ;
10                o_f2.foo(3);
11          }
12          static void f3(){
13                Data o_f3=f1(data) ;
14                o_f3.foo(10);
15          }
16 }
```

函数 *f*2 和 *f*3 都调用了函数 *f*1,*o*2,*o_f*3,*o_f*2 是局部变量,Data.foo()函数体中有对调用对象的同步操作.我们的分析对 *f*2()和 *f*3()分别应用 *f*1()的 summary 信息:将 *f*1()的 method summary⟨⟨*a*0⟩,*a*0,,⟩带入两处调用得到 *o_f*2 是局部变量 *o*2 的别名,*o_f*3 是全局变量 data 的别名,所以对前者的同步是可以删除的.而上下文不敏感分析会将 *f*1 的两处调用信息合并,得到返回值 *o_f*2 和 *o_f*3 都是全局静态变量的别名,所以如果 data 还在另一线程中被同步,那么两处同步操作就都不能删除了.

我们的数据流分析还是流敏感的,考虑到了控制流(顺序/分支/循环)对数据信息的影响.程序被当作语句序列来处理,计算的是每个控制流点的数据信息.

下面的程序是 Ruf 的文献[4]中的一个例子.

```
1 Class SimpleVector{
2       Object[] elements;
3       Static SimpleVector v;
4}
/*Thread T1 */
5static void test2() {                          /*SimpleVector.v: ⟨{},false,{},true⟩*/
6       SimpleVector v2 = new SimpleVector();
7       SimpleVector.⟨init⟩(v2);
8       Object o2 = SimpleVector.elementAt(v2, 0); /*v2:{elements->a9,true,{},false}*/
9       SimpleVector.v = v2;
        /*v,v2:{elements->a9,true,{},true} {elements->a9,true,{T1},true }*/
10}
/* Thread T2 */
11static void test3() {
12      SimpleVector v3 = SimpleVector.v;
```

```
13    Object o3 = SimpleVector.elementAt(v3,0);
      /*v,v3: {elements->a9, true,{T2},true} {elements->a9,true,{T1,T2},true}*/
14}
```

我们通过这个例子说明流敏感分析对同步删除率的提高.其中的注释部分是在不同程序点对象的 alias set,黑体部分是 Erik 得到的不同分析结果.在 Erik 的流不敏感分析中,由于各个程序点的对象数据信息都相同,线程 $T1$ 执行完 SimpleVector.v=$v2$ 后得到 $v2$ 逃逸了,这个信息会传播给前一条语句 Object $o2$= SimpleVector.elementAt($v2$,0)**(就好像交换了语句执行顺序一样),而 elementAt()调用中又有对 $v2$ 的同步操作,这样,$T1$ 分析完得到 $v2$ 逃逸并且有对它的同步操作.由于线程 $T2$ 中也有对 $v2$ 所指全局对象的同步操作,这样,第 8 行和第 13 行的同步操作就都不能删除了.实际上,$T1$ 对 $v2$ 的同步操作发生在逃逸之前,它无法和 $T2$ 中 $v3$ 的同步操作真正并发执行,这两个同步是可以删除的.利用我们的分析就可以得到这样的结果.

流敏感分析对顺序语句分析的关键是计算引用赋值语句带来的等式左右两边对象 alias set 的变化.根据流敏感分析的转换函数 Out=Gen+(In−Kill),除了添加当前赋值($v0$=$v2$)带来的新的 alias set 指向以外,还要消除上次赋值($v0$=$v1$)的影响.对全局变量 g 的赋值是个特殊情况:g=l 使得表达式右边的局部量 l 逃逸,即使对 g 再次赋值也无法保证在 l 逃逸期间没有其他线程引用该变量,所以我们不做 kill 操作而让 l 的状态保持 global,并且 g 始终指向 l.

在程序的分支结构中,我们为分支中引用的对象变量引入 True 和 False 两个新对象.根据数据流等式 $In=\wedge_{i\in pred(bb)}Out_i$,我们在控制流的合并点对各个分支得到的同一引用的 alias set 做 Union 操作,所以一个引用可能指向若干个 alias set 等价类.

下面是控制流合并的一个简单实例.

```
/*Thread T1 SimpleVector.v: <{},false,{},true>*/
1static void test2() {
2     SimpleVector v1 = new SimpleVector();   /*v1: {elements->a9,false,{},false}*/
3     SimpleVector v2 = new SimpleVector();   /*v2: {elements->a10,false,{},false}*/
4     if (…){
5          Object o2 = SimpleVector.elementAt(v1, 0);
6          SimpleVector.v = v1;                    /*v, v1_T: {elements->a9,true,{},true}*/
7          Sync(v2);                               /*v2_T: {elements->a10,true,{},false}*/
8     }else{
9          SimpleVector.v = v2;                    /*v, v2_F: {elements->a9,true,{},true}*/
10         Object o2 = SimpleVector.elementAt(v2,0); /*v,v2_F: {elements->a10,true,{T1},true}*/
11    }
12}                                               /*v,v2_F:{elements->a10,true,{T1},true}*/
/* Thread T2 */
13static void test3() {
14    SimpleVector v3 = SimpleVector.v;
15    Object o3 =SimpleVector.elementAt(v3,0);    /*v3,v,v2_F:{elements->a10,true,{T1,T2},true}*/
16}
```

在上面的实例中,线程 $T1$ 的语句 12 处,$v1$ 指向两个 alias set,分别是 $v1\_F$:{elements->a9,false,{},false}和 $v1\_T$:{elements->a9,true,{},true}.$v2$ 也指向两个 alias set,分别是 $v2\_T$:{elements->a10,true,{},false} 和 $v2\_F$:{elements->a10,true,{$T1$},true}.我们将全局量 SimpleVector.v 在 $T1$ 不同程序点的状态传播到线程 $T2$ 中***.在 $T2$ 中继续分析,发现静态变量又被同步,最终得到 $v3$,v,$v2\_F$:{elements->a10,true,{$T1$,$T2$},true}状态.所以在接下来的自顶向下的传播过程中,会得到 10,15 处的同步操作应保留.而 $T1$ 的 true 分支中的同步可以消除.

当包括主线程在内的所有线程都完成了分析以后,就得到了最终的各个对象的 alias set 属性.Escape 对象的 alias set 在分析过程中逐渐发生变化,它记录了对象被哪些线程同步.比如,调用 thread-safe 库类的程序,如果自身没有使用共享对象,那么这个对象的同步线程集合就是单个线程.Escape 对象只要没有被多个线程同步,对它的同步操作就是可以删除的.所以我们将对象最终的 alias set 重新带入方法中,自顶向下寻找它们的变化对原来

** elementAt()函数体对第 1 个形参做同步操作,它的信息通过 method summary 传递给了 $v2$.

*** 根据第 3.2 节的优化,只需传播 SimpleVector.v 的同步状态:{elements->a10,true,{$T1$},true}.

同步方法或同步块的影响,确定可以消除同步的位置.例如上面实例中从主线程向下传播中得到 false 分支中的同步不能删除,true 分支的同步则是冗余的,可以删除.

2.2 线程间时序分析阶段

经过基本处理阶段,只有那些确实被多个线程同步的 escape 对象才保留了同步操作.但即使是对同一对象有多个同步操作,只要这些同步操作存在时序关系,那么它们依然是可以删除的.为了实现更精确的分析,我们接下来为这些对象搜集访问的先后信息,继续寻找可以删除的同步操作.

图 2 是例子程序和简单的执行时间图.根据 ICG 和 CFG 记录的特定程序范围和锁范围带来的操作的先后顺序对图 2 中的程序分析后,我们得到{*q*,*a*,*a*1}〈,,true,{Main,*T*1},true〉和{*x*,*a*,*a*1}〈,,true,{Main,*T*2},true〉.从右边线程交互的时间轴上可以看到,join 操作分割了 main 和 *T*2 中对 *x*.*f* 的两个同步操作,而对 *x*.*f* 的同步操作恰好仅有两个,也就是说,对这一对象的同步操作完全有序,所以这两个同步操作都可以删除.

```
public class MainThread{
        static Obj q, x; //suppose q, x not alias
        public static void main(String args[]){
                MyThread T1=new ChildThread(q);
                T1.start();
                MyThread T2=new ChildThread(x);
                T2.start();
                m1(q);
                T2.join();
                Synchronized(x){x.f=200};
        }
        public static void m1(Obj y){
                synchronized(y) {
                        y.f=100;
                }
        }
}
//thread T1 and T2
class ChildThread extends Thread{
        Obj a;
        ChildThread(Obj a1){a=a1;}
        Public void run(){
                        MainThread.m1(a);
        }
}
```

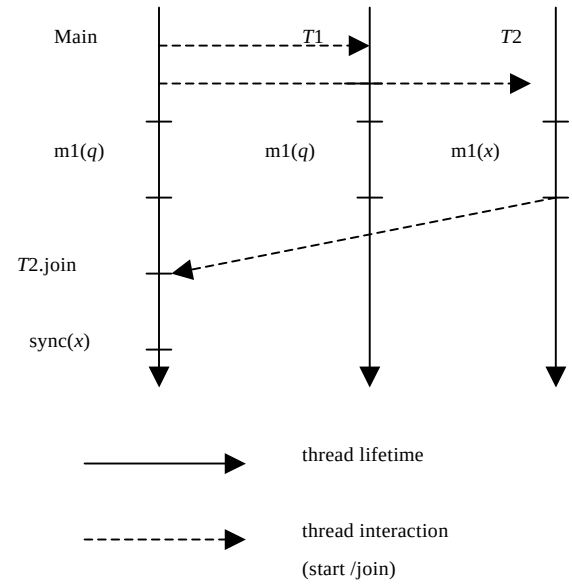

Fig.2 Inter-Thread temporal analysis

图 2 线程间时序分析

3 算法描述

3.1 预处理

首先扫描整个程序****,利用类型信息构造 ICG,并为每个过程建立 CFG.对程序中出现的 Thread.start()调用语句计数,如果个数为 1,删除程序中所有的同步操作.这种计数方法是非常保守的,例如任何分支中的 start 语句都会使得计数加 1;循环体中的 start 语句认为执行超过 1 次.

3.2 信息收集

在信息收集阶段,首先将 ICG 划分成强连通分量 SCC(strongly connected component)的集.然后按照自底向上的顺序对各个 SCC 节点中的每种方法进行 escape 分析:在函数体内部带入全局变量的 alias set 以及每个形

**** 我们的分析对象是 whole-program,不考虑 dynamic class loading 等动态属性.

参变量所赋予的 alias set,对函数体的语句依次遍历并按表 1 定义的转换函数计算各个控制点的 alias set 图,最后将得到的 method summary 以及更新后的全局对象 alias set 反馈给调用函数并自顶向下传播当前上下文信息;如果存在递归,则对调用函数和被调用函数的 method summary 做 unify 操作.虽然这使得递归调用处成为上下文不敏感的,但是这比对 SCC 做迭代直到得到不动点(fix point)要更简单.类似地,我们对流图中的循环结构也是做 unify 操作.

下面是信息收集的高层算法.

```
partition ICG into SCC sets
for each SCC in bottom-up order{
    for each method m in the SCC {
        if methodcall{
            for each call site s in m and each target t of s
                if t and m are in the same SCC
                    unify actuals of s and formals of t
                else
                    get the summary info of t and apply current context top-down      //in previous
                                                    //bottom-up analysis, t must have been analyzed
            }else{
                partition into basicblock's scc
                analyze statement according to table 1's transfer function
        }
    }
}
```

如果是自底向上将一个线程分析完后得到的全局量信息传递给下一个线程,那么由于全局变量的所有执行状态对其他线程都是可见的,所以在向其他线程传播时,全局变量在所有程序点的 alias set 集合都要传播给下一个线程.为了简化,如果当前线程得到的某个全局量 alias set 中没有当前线程的同步操作,那么不用传递这个全局量,在下一个线程中依然使用这个全局对象的初始值.另外,需要说明的是,算法无法分析动态加载的方法和 native 库函数的方法体,我们将传递给这些方法的实参对象的 escaped 属性置为真.

表 1 是字节码语句对应的转换函数.其中 *l*,*v* 代表局部变量,*g* 代表全局变量.表中没有出现多层应用表达式,这是因为 Java 源程序中的多层引用 a.b.cd..在字节码中已经转换成一系列的单层引用.

Table 1 Transfer function for statements

表 **1** 语句转换函数

Statement type	Transfer function
l=*v*	AS(*l*)=AS(*v*); if *l* and *v* are different thread variables, set AS(*l*).escaped=true AS(*v*).escaped=true
l=*g*(*g*=*l*,*g*1=*g*2)	AS(*l*)=AS(*g*); set AS(*l*).escaped=true and propagate to all reachable nodes from *l*
p=*q*.*f* (*f* is static/nonstatic field)	AS(*p*)=AS(*q*).fieldmap(*f*); Set and propagate *p*'s escape state if *f* is static field
p.*f*=*q* (*f* is static/nonstatic field)	AS(*p*).fieldmap(*f*)=AS(*q*); Set and propagate *q*'s escape state if *f* is static field
*l*1=*l*2[…] and *l*1[…]=*l*2	Act like normal load/store, but set fieldmap's name as $ELT
l=new *c*l()	Create new alias set and update AS(*l*)
return *l*	AS(*l*)=*r*
l=*l*0.op(*f*1…*fk*)	Apply real params in method summary
throw *v*	AS(*l*)=*e*
monitorEnter *v* or synchronized method	*v*.synchronized=true or this. synchronized=true
if …else…	Union alias sets in True/False branch

3.3 信息传播

分析完所有线程后得到的 alias set 再自顶向下传播给各个线程,观察当前引用对应的 alias set 元组属性,标记冗余同步操作.如果引用对应多个 alias set,根据同步调用处的标记选择正确的分支对象.

3.4 线程间交互分析

根据 ICG 和 CFG 记录的线程开始位置、join 位置和剩余的同步操作的位置建立剩下的同步操作的先后关

系.为了简化分析,我们只考虑同步线程集合大小为 2 的对象同步操作.这样,只要检查是否有 join 操作分割这两个操作,如果有,它们就可以被消除.

4 实验结果

我们在 MIT 的一个用 Java 编写的 Java 编译器 FLEX[5]上构造了 JTool.JTool 接受字节码作为输入,可以作为一个基本的分析框架对包括 escape 分析在内的数据流问题作上下文敏感和流敏感的分析.为了测试 JTool 中实现的同步删除算法,我们选取了 3 个单线程程序和 3 个多线程程序作为 benchmark.实验环境是一台 P4 2.66GHz,256MB 内存的 Linux 系统,运行系统带有 generic Java 扩展的 jdk1.4.2-04.

MultiSet,StrSplit 和 SortObjects 是 CodeProject 网站上开发者提供的两个单线程程序.BoundedBuffer, EscapeTest 和 Sum 是文献[5]提供的 3 个多线程测试程序.表 2 列出了测试程序的大小、同步操作移出前/后的数目和分析时间(分析时间不包括读进最初的 class 文件).从表中可以看到:单线程程序的同步移出率达到了 100%,多线程程序也达到了很高的移出率.由于算法应用了上下文敏感和流敏感分析,它们的最坏时间/空间复杂度都是程序大小的指数级[11,12].即使不处理控制流图和调用图中的循环/递归结构,算法复杂度也至少和程序大小成线性关系.虽然表 2 的移出分析时间显示算法开销很大,我们仍认为是在合理的分析时间范围内.

Table 2 Experiment results

表 **2** 测试程序分析结果

Spec. name	Method num	Bytecode num	Sync. of number		Analysis time (s)
			Before	After	
Multiset	16	134	3	0	214.216
StrSplit	547	13 861	224	0	1 195.767
SortObjects	585	14 527	234	0	1 483.296
BoundedBuffer	577	14 471	226	154	1 317.320
EscapeTest	587	14 805	236	164	1 335.781
Sum	594	14 739	225	151	1 321.332

5 相关研究

多数同步删除算法[1,6–9]都是基于最基本的 escape 分析,即找出逃离当前线程的对象.它们的区别只在于数据流分析时的精度有所不同.比如,文献[7]应用了跨过程的、流敏感和上下文敏感的数据流分析以及一个简单的 escape 和堆对象的 shape 分析,并且只模拟了一层域的解引用.文献[8]的方法更精确一些,它是上下文敏感和流敏感的,并且模拟了对象任意层次的嵌套.

Erik Ruf 首先提出对传统 escape 对象扩充访问的同步线程信息.他的分析文献[4]跟踪每个线程对每个对象的同步,促使编译器移出对那些被多个线程访问,但只被单线程同步的对象的同步操作.这是一个上下文敏感、流不敏感的过程间分析.我们的基本处理阶段借鉴了文献[4]的分析方法.但是应用了更精确的流分析,而且还进一步考虑了对存在时序关系的冗余同步操作的删除.

Martin Rinard 的 escape 分析[10]是一个上下文敏感、流敏感、跨过程的分析过程,并且考虑到了时序关系.文献[10]的主要贡献在于设计了一种新的并行交互图(PIG)抽象,它保存了线程交互、别名、逃逸信息以及线程访问对象的时序关系.PIG 相当于我们的 summary,alias set 和时序分析的集合.但是这种分析主要应用于基于区域(region-based)的内存管理,如减少对悬空指针的动态检查,而不是用于同步删除的优化.文献[10]的 escape 对象特指已经作过分析的方法和线程无法访问到的对象,分析边界也不是通常意义上的单个方法或单个线程.

6 结 论

本文提出并实现了一种更精确的同步删除方法.在分析的两个阶段都通过时序分析(语句顺序和线程间的顺序)提高了并发程序同步操作的移出率,并最终加大了对 Java 语言多线程程序的优化.它可以应用于现有编译器、虚拟机的优化过程,也可以作为单独的分析工具应用于进一步的分析过程.实验结果表明,用我们的算法可以在有效的时间内对中/小规模的程序取得精确的分析结果.在开发 JTool 分析工具的过程中,我们也认识到,由

于流敏感、上下文敏感分析固有的指数级时间和空间复杂度,这种精确分析很难扩展到大程序分析.如何在精度和分析开销之间取得平衡,设计更加实用的算法是进一步研究的内容.

虽然两个阶段的分析使得我们基本上移除了绝大多数满足删除条件的同步操作,但是分析算法还可以进一步提高.比如 points-to 算法可以采用 path-sensitive 分析产生更加精确的 points-to 集合.我们的两步分析对于由于路径条件(比如,线程 1: *x*++; *F*=TRUE 线程 2: if (*F*==TRUE)*x*−−;)限定而出现的执行顺序依然无法判断.文献[13]中采用的符号执行是可能的一种解决方法,但是对于大规模的程序分析代价可能很大.

References:

[1] Aldrich J, Chambers C, Sirer E, Eggers S. Static analyses for eliminating unnecessary synchronization from Java programs. In: Proc. of the 6th Int'l Symp. on Static Analysis. London: Springer-Verlag, 1999. 19−38.

[2] Fitzgerald R, Knoblock TB, Ruf E, Steensgaard B, Tarditi D. Marmot: An optimizing compiler for java. Software-Practice and Experience, 2000,30(3):199−232.

[3] Bacon DF, Konuru R, Murthy C, Serrano M. Thin locks: Featherweight synchronization for Java. In: Proc. of the Conf. on Programming Language Design and Implementation (PLDI'98). New York: ACM Press, 1998. 258−268.

[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] http://www.flex-compiler.csail.mit.edu

[6] Blanchet B. Escape analysis for object-oriented languages: Application to Java. In: Proc. of the 14th Annual Conf. on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'99). New York: ACM Press, 1999. 20−34.

[7] Bogda J, Hoelzle U. Removing unnecessary synchronization in Java. In: Proc. of the 14th Annual Conf. on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'99). New York: ACM Press, 1999. 35−46.

[8] Choi JD, Gupta M, Serrano M, Sreedhar VC, Midkiff S. Escape analysis for Java. In: Proc. of the 14th Annual Conf. on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'99). New York: ACM Press, 1999. 1−19.

[9] Choi JD, Gupta M, Serrano MJ, Sreedhar VC, Midkiff SP. Stack allocation and synchronization optimizations for Java using escape analysis. In: Proc. of the ACM Sigplan Trans. on Programming Languages and Systems (TOPLAS 2003). New York: ACM Press, 2003,25(6):876−910.

[10] Salcianu A, Rinard M. Pointer and escape analysis for multithreaded programs. In: Proc. of the Symp. Principles and Practice of Parallel Programming (PpoPP 2001). New York: ACM Press, 2001. 12−23.

[11] Wilson RP. Efficient, context-sensitive pointer analysis for C programs [Ph.D. Thesis]. Stanford University, 1997.

[12] Muth R, Debray S. On the complexity of flow-sensitive dataflow analyses. In: Proc. of the Symp. on Principles of Programming Languages (POPL 2000). New York: ACM Press, 2000. 67−80.

[13] von Praun C, Gross TR. Static conflict analysis for multi-threaded object-oriented programs. In: Proc. of the Conf. on Programming Language Design and Implementation (PLDI 2003). New York: ACM Press, 2003. 115−128.