软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2013,24(10):2289−2299 [doi: 10.3724/SP.J.1001.2013.04372]

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

高可扩展性的 MHP 分析算法*

印 乐, 黄 磊

(计算机体系结构国家重点实验室(中国科学院 计算技术研究所),北京 100190)

通讯作者: 印乐, E-mail: yinle@ict.ac.cn, http://www.ict.ac.cn

摘 要: 并行发生(may happen in parallel,简称 MHP)分析计算并行程序中哪些语句可以并行执行,它是并行分析技术的重要组成部分.提出一种针对 Java 程序的新颖的 MHP 分析算法.与已有算法相比,新算法抛弃了"子线程只会被父线程等待同步"的假设,以非耦合的方式分别处理 start 同步和 join 同步;新算法的处理逻辑虽然更加简单,但却更加完备;在计算控制信息时,新算法不必像已有算法那样通过内联构造全局的控制流图,显著地提高了算法的扩展性.新的 MHP 算法被用来过滤静态数据竞争检测中虚假的数据竞争.在 14 个 Java 测试程序上的实验结果表明,新的 MHP 算法计算控制信息的开销远远小于已有算法.

关键词: MHP;可扩展性;数据竞争;静态分析

中图法分类号: TP301 **文献标识码**: A

中文引用格式: 印乐,黄磊.高可扩展性的 MHP 分析算法.软件学报,2013,24(10):2289−2299. http://www.jos.org.cn/1000-9825/4372.htm

英文引用格式: Yin L, Huang L. Highly scalable MHP analysis algorithm. Ruan Jian Xue Bao/Journal of Software, 2013,24(10): 2289−2299 (in Chinese). http://www.jos.org.cn/1000-9825/4372.htm

Highly Scalable MHP Analysis Algorithm

YIN Le, HUANG Lei

(State Key Laboratory of Computer Architecture (Institute of Computing Technology, The Chinese Academy of Sciences), Beijing 100190, China)

Corresponding author: YIN Le, E-mail: yinle@ict.ac.cn, http://www.ict.ac.cn

Abstract: May happen in parallel (MHP) analysis decides whether a pair of statements in a parallel program can be executed concurrently; it plays an important part in parallel analyses. This paper proposes a novel may-happen-in-parallel analysis algorithm for Java programs. Compared with the existing MHP algorithm, the new alogrithm discards the unnecessary assumption that a child thread can only be join-synchronized by its parent thread, and processes start-synchronization and join-synchronization independently in a decoupled way. This makes the processing logic of the algorithm more concise and more complete than that of the existing algorithm. When computing dominator information, the new algorithm has better scalability for it does not need to construct the global flow graph for the program by inlining, which is needed by the existing algorithms. The new MHP analysis algorithm is used to sift false warnings reported by a static data race detection tool. The experimental results on 14 Java programs show that the time of computing dominator information of the new MHP analysis is much shorter than that of the existing algorithm.

Key words: MHP; scalable; data race; static analysis

可并行发生(may happen in parallel,简称 MHP)分析计算并行程序中哪些语句可以并行执行.MHP 是一种重要的并行分析,它可以增加数据流分析并验证分析的准确性[1−3].

* 基金项目: 国家自然科学基金(61202055, 60921002); 国家重点基础研究发展计划(973)(2011CB302504); 国家高技术研究发展计划(863)(2012AA010902)

收稿时间: 2012-01-04; 修改时间: 2012-04-11; 定稿时间: 2013-01-07

MHP 的另一个重要应用就是数据竞争检测.如果来自不同线程两个冲突的访存操作不能并行运行,那么它们就不会构成数据竞争.有些数据竞争检测[4,5]把注意力放在关键区同步的分析(锁集合算法)上而忽视了由于 *start*()和 *join*()方法引起的线程创建和终止相关的序同步问题.文献[6]为待分析程序计算时序约束图(temporal constraint graph,简称 TCC),用 TCC 来描述 Java 程序中的 *start*()和 *join*()方法的调用所限定的不同线程操作之间的先后顺序.要确定待分析程序中任意两条语句(文献[6]称其为事件)之间的先后顺序,其计算的时间复杂度是 $\Theta(N_s^2)$,其中,N_s 是程序中所有语句的条数.文献[6]使用较小的测试程序集(它所使用的 5 个测试程序中最大的程序也只有 528 行代码).对小程序而言,计算 TCG 的开销完全是可以承受的.

有一系列的工作专门致力于研究MHP算法.早期的研究工作针对Ada语言进行并行区域分析.文献[7]提出了 B4 分析算法.文献[8]将 B4 分析应用到 Ada 语言的同步模型,并将其扩展成过程间分析.文献[9]提出一种迭代的方式来计算在 Ada 语言中无法并行执行的程序语句.这种方式能获得的结果比文献[7,8]的结果要准确,算法的复杂度是 N_s 的五次方.文献[10]提出一种在 Ada 语言同步模型之上的数据流分析算法,在最坏情况下,算法的复杂度为 N_s 的六次方,但是实际运行时复杂度可能是 N_s 的三次方甚至更低.文献[11,12]是针对 X10 这样的高层并行语言提供的 async/finish/atomic 同步机制进行 MHP 分析.X10 语言的同步机制不涉及线程对象和过程间的别名分析,因此非常简单.而且 X10 作为一种实验性的语言,目前并没有太多的商业应用选择 X10 来编写.

Java 是目前最流行的多线程语言之一.不论在工业界还是学术界,有大量的 Java 语言编写的应用程序.已有的针对 Java 语言的 MHP 分析可以分为两类:

- 第 1 类方法仍然使用数据流分析来计算 MHP 信息,属于这类方法的研究工作有文献[13,14].文献[13,14]将 Java 程序中所有的方法都内联到方法 *main*,获得并行执行图(parallel execution graph,简称 PEG).通过在 PEG 上解数据流方程来计算程序中每个节点的 MHP 信息.内联使得文献[13,14]只能处理规模较小的 Java 程序,无法处理真正的大规模的实际应用.
- 另一类是 Barik 提出的基于线程创建树(thread creation tree,简称 TCT)的 MHP 分析算法[15].Barik 的算法根据线程之间的创建关系构造 TCT,并通过内联获得线程内控制流图(intra-thread control flow graph,简称 ICFG),然后在 TCT 和 ICFG 上计算 MHP 信息.

已知的针对 Java 的 MHP 算法都不具有良好的扩展性,原因是:

- PEG 和 ICFG 是全局的流图,它们或者通过内联程序中的所有方法来获得,或者通过内联一个线程访问的所有方法来获得;内联会造成代码急剧膨胀,这种做法对于大的程序来说是不可接受的.
- 解数据流方程的时间复杂度为 $\Theta(n^3)$,计算流图中节点的控制信息的时间复杂度为 $\Theta(n^2)$,这里,n 是流图中节点(基本块)的个数.不论采取哪种方法,当 n 的值增加时,算法的复杂度都会急剧升高.

我们提出一种针对 Java 语言的新的 MHP 分析算法.据我们所知,Barik 的算法是目前已发表的针对 Java 语言的最快的 MHP 算法.实验结果表明,Barik 算法的执行速度是数据流 MHP 算法的 1.77 倍[15].与 Barik 的算法相比,我们提出的新算法有如下优点:

- 新算法抛弃了 Barik 算法中子线程只会被父线程等待结束这一假设,以非耦合(decoupled)的方式处理 Java 中的 ***start*** 同步和 ***join*** 同步.新算法比 Barik 算法在逻辑上要简单,而且能处理更多的 ***join*** 同步,因此也更加完备.
- 新算法具有很好的扩展性.与 Barik 算法一样,新算法的主要开销是计算数据流图中的控制和后控制信息.但与 Barik 算法在 Java 程序的全局数据流图上计算控制和后控制信息不同,新算法只在程序的每个方法内计算这些信息.算法分析和实验结果表明,随着程序规模的扩大,Barik 算法的开销急剧增加,达到不可忽略的地步,而新算法仍然保持极小的开销,具有很好的扩展性.

本文第 1 节以一个简单程序作为例子,描述 MHP 分析依赖的静态线程和 3 种同步的概念,并显示 MHP 分析的结果.第 2 节采取自顶向下的方式详细介绍 MHP 算法.第 3 节给出 MHP 算法在 14 个测试程序上应用的效果.第 4 节将本文的 MHP 算法与已知的 Java 上最好的 Barik 算法进行比较.第 5 节对全文进行总结.

1 问题描述

本节首先描述 MHP 分析所依赖的静态线程的概念,然后介绍静态线程之间存在的 *start* 同步、*join* 同步和 *join-then-start* 同步,最后给出 MHP 算法分析结果的实际例子.

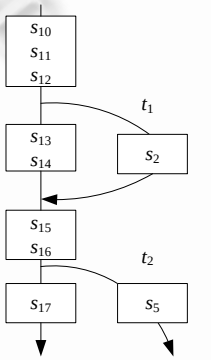
Java 程序运行时首先启动一个初始线程,称为主线程.主线程从程序的入口 *main* 方法开始执行.在执行的过程中,主线程可以派生出新的线程,后者可以派生出更多的线程.这些线程是程序执行时动态产生的.

本文提出的 MHP 算法是一种静态分析算法,它将程序划分为多个静态线程.一个静态线程可以对应多个动态线程.对于分别属于不同静态线程的两条语句 s_1 和 s_2,MHP 分析算法报告 s_1 与 s_2 是否可以并行执行,即在程序执行时,位于两个动态线程的语句 s_1 和 s_2 是否可以并行地执行.

静态线程的划分由两条规则决定:*main* 方法及其所调用的其他方法(*Thread*::*start* 方法除外)构成了静态主线程;对某个线程对象调用 *Thread*::*start* 方法意味着一个新的静态线程,它由该线程对象的 *run* 方法和 *run* 方法所调用的其他方法(*Thread*::*start* 方法除外)构成.静态线程的划分和 MHP 分析的结果可以由图 1 的例子说明.在本文接下来的叙述中,如果没有特别说明,“线程”都表示静态线程.

```
1  class Task1 extends Thread {
2    public void run() {Main.cnt++;}
3  }
4  class Task2 extends Thread {
5    public void run() {Main.cnt--;}
6  }
7  public class Main {
8    static int cnt;
9    public static void main(...) {
10     cnt++;
11     Task1 t1=new Task1();
12     t1.start();                    // t1
13     cnt++;
14     t1.join();
15     Task2 t2=new Task2();
16     t2.start();                    // t2
17     cnt++;
18   }
19 }
```

(a)



(b)

$mhp(s_{10},s_2)$ → false
$mhp(s_{15},s_2)$ → false
$mhp(s_5,s_2)$ → false
$mhp(s_{13},s_2)$ → true

(c)

Fig.1 Static threads in the program and the results of MHP analysis

图 1 程序中的静态线程和 MHP 分析结果

图 1(a)是源程序,图 1(b)是程序中静态线程的示意图.s_i 表示源程序第 i 行语句.程序中主线程由 *main* 方法中 8 条语句 s_{10}~s_{17} 构成.主线程创建线程对象 t_1,并调用 t_1.*start*()启动了线程 t_1,后者含有一条语句 s_2;主线程调用 t_1.*join*(),等待线程 t_1 执行结束.主线程还派生线程 t_2,它含有语句 s_5.程序中 3 个线程之间的派生/等待结束(*start*/*join*)关系图形化地表示在图 1(b)中.

图 1(c)是部分 MHP 分析结果:

- 主线程的语句 s_{10} 执行之后才执行语句 s_{12} 并派生线程 t_1,然后,t_1 才执行语句 s_2.因此,s_{10} 必定先于 s_2 执行.我们说 s_{10} 与 s_2 之间存在 ***start*** 同步.也就是说,同一个线程内的语句 s_{10} 和 s_{12} 的顺序,决定了语句 s_{10} 与 s_2 之间的顺序,s_{10} 和 s_2 属于不同的线程.
- 主线程的语句 s_{14} 执行完毕返回时意味着 t_1 已经结束,主线程接着执行语句 s_{15}.因此,语句 s_{15} 必定后于 s_2 执行.我们说 s_{15} 与 s_2 之间存在 ***join*** 同步.也就是说,同一个线程内的语句 s_{14} 与 s_{15} 之间的顺序,决定了语句 s_2 与 s_{15} 之间的顺序,s_2 与 s_{15} 属于不同的线程.

- 主线程先执行语句 s_{14} 等待线程 t_1 结束,然后执行语句 s_{16} 派生线程 t_2.因此,s_2 必定先于 s_5 执行.我们说 s_2 与 s_5 之间存在 ***join-then-start*** 同步.也就是说,同一线程内存的语句 s_{14} 和 s_{16} 决定了语句 s_2 与 s_5 之间的先后顺序,s_2 与 s_5 属于不同的线程.

语句 s_{13} 与 s_2 之间不存在任何同步,因此它们可以并行执行.

2 MHP 算法

在介绍了静态线程概念和 3 种同步之后,本节详细介绍我们提出的新的 MHP 算法.首先给出集合以及集合上关系的定义,然后自顶向下地介绍 MHP 算法.

2.1 线程之间的关系

这一节我们定义若干集合以及这些集合上的关系,其目的是获得关系 ***TstartT***,***TJoinT*** 和 ***TTTJS***,它们分别对应 ***start*** 同步、***join*** 同步和 ***join-then-start*** 同步.

一个 Java 程序 ***P*** 中的所有线程构成一个集合 ***T***.程序 ***P*** 中所有对线程对象的 *start* 方法的调用语句构成集合 ***S***.程序 ***P*** 中所有对线程对象的 *join* 方法的调用语句构成集合 ***J***.

TST(t_1,s,t_2)是定义在集合 ***T*** 和集合 ***S*** 上的关系,它表示线程 t_1 在语句 s 上调用 *start* 方法,启动了线程 t_2 的执行.***TJT***(t_1,j,t_2)是定义在集合 ***T*** 和集合 ***J*** 上的关系,它表示线程 t_1 在语句 j 上调用 *join* 方法,t_1 等待线程 t_2 执行完毕(如图 2 所示).

T=$\{t_1,t_2,\ldots,t_i,\ldots,t_N\}$,$t_i$是程序中的一个线程.

S=$\{s_1,s_2,\ldots,s_i,\ldots,s_M\}$,$s_i$是程序中一条*start*()方法的调用语句.

J=$\{j_1,j_2,\ldots,j_i,\ldots j_Q\}$,$j_i$是程序中一条*join*()方法的调用语句.

TST=$\{(t_1,s,t_2)|t_1,t_2\in \boldsymbol{T}\wedge s\in \boldsymbol{S}\wedge t_1$在$s$语句上调用$t_2.start()\}$.

TJT=$\{(t_1,j,t_2)|t_1,t_2\in \boldsymbol{T}\wedge j\in \boldsymbol{J}\wedge t_1$在$j$语句上调用$t_2.join()\}$.

Fig.2

图 2

在图 2 的基础上,根据图 3 中的规则,计算得到 3 个新的关系 ***TstartT***,***TJoinT*** 和 ***TTTJS***.

$$\boldsymbol{TStartT}(t_1,s,t_2)=\boldsymbol{TST}(t_1,s,t_2) \quad (1)$$

$$\boldsymbol{TStartT}(t_1,s,t_2)=\boldsymbol{TStartT}(t_1,s,t_2)\wedge\boldsymbol{TStartT}(t_2,s',t_3) \quad (2)$$

$$\boldsymbol{TJoinT}(t_1,s,t_2)=\boldsymbol{TJT}(t_1,s,t_2) \quad (3)$$

$$\boldsymbol{TJoinT}(t_1,s,t_2)=\boldsymbol{TJoinT}(t_1,s,t_2)\wedge\boldsymbol{TJoinT}(t_2,s',t_3) \quad (4)$$

$$\boldsymbol{TTTJS}(t_1,t_2,t_3,j,s)=\boldsymbol{TJoinT}(t_3,j,t_1)\wedge\boldsymbol{TStartT}(t_3,s,t_2)\wedge t_1!=t_2 \quad (5)$$

Fig.3

图 3

2.2 算法思路

对于两条语句 s_1 与 s_2,假设 s_1 属于线程 t_1,s_2 属于线程 t_2,我们的 MHP 算法并不直接判断 s_1 与 s_2 是否可以并行执行,而是试图分析它们之间是否存在确定的先后顺序.如果 s_1 与 s_2 之间存在先后顺序,则它们不能并行,反之可以并行.判断 s_1 与 s_2 是否存在先后顺序的算法实现在方法 *happen_before* 中,下面的流程图(如图 4 所示)描述了方法 *happen_before* 的处理逻辑.

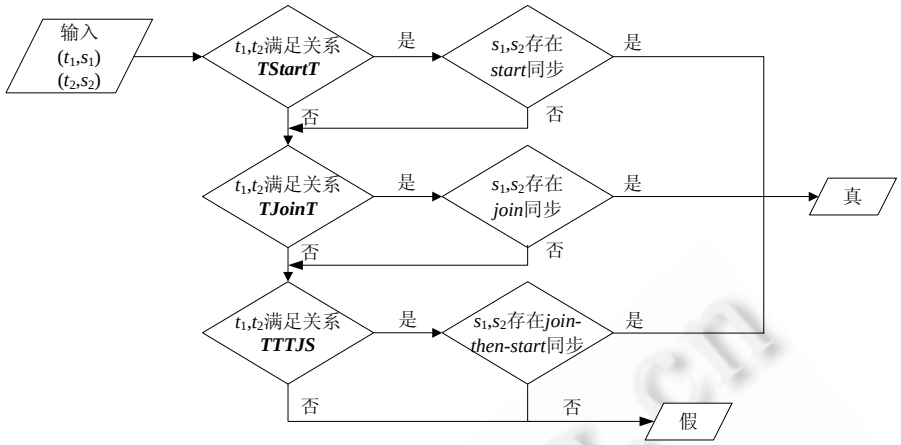


Fig.4 Flow chart of the method *happen_before*

图 4 方法 *happen_before* 的流程图

2.3 MHP算法伪码

MHP 分析算法在方法 *MHP*()中得以实现.对于程序中的两条语句 s_1 和 s_2,*MHP*(s_1,s_2)返回真表示它们可以并行执行.方法 *MHP*()调用了方法 *happen_before*()来判断两条语句的执行是否存在先后顺序.图 4 已经描述了方法 *happen_before*()的处理流程.方法 *happen_before*()调用其他方法来实现其处理逻辑.其中,方法 *Thread*(*s*)返回语句 *s* 所属的线程.我们使用方法 *TStartT*(),*TJoinT*()和 *TTTJS*()分别表示关系 ***TstartT***,***TJoinT*** 和 ***TTTJS***.方法的返回值为真时,表示关系得到满足,否则,不满足.方法 *po_before*(*s*,*s*′)用来判断语句 *s* 是否先于语句 *s*′,其中,*s* 和 *s*′属于同一个线程(如图 5 所示).方法 *po_before*()的实现细节将在下一节给出详细介绍.

```
1  MHP(s1,s2)                          12    t1=Thread(s1);
2  {                                   13    t2=Thread(s2);
3     if (happen_before(s1,s2))        14    if (TStartT(t1,s,t2))
4        return false;                 15       return po_before(s1,s);
5     if (happen_before(s2,s1))        16    else if (TJoinT(t2,j,t1))
6        return false;                 17       return po_before(j,s2);
7                                      18    else if (TTTJS(t1,t2,_,j,s))
8     return true;                     19       return po_before(j,s);
9  }                                   20
10 happen_before(s1,s2)                21    return false;
11 {                                   22 }
```

Fig.5 MHP algorithm pseudocode

图 5 MHP 算法伪码

2.4 语句之间的程序序

方法 *happen_before* 的实现用到了方法 *po_before*,后者判断同一个线程内语句之间的程序序.程序序(program order,简称 po)是程序文本指定的同一个线程内语句之间的先后顺序;$s \xrightarrow{po} s'$ 表示 *s* 在程序序上先于 *s*′.例如,图 1 的示例程序主线程有 8 条语句 s_{10}~s_{17},我们有 $s_i \xrightarrow{po} s_j$,当且仅当 $10 \leqslant i < j \leqslant 17$.

当语句属于同一个方法时,我们使用方法控制流图上的控制和后控制信息来判断语句间的程序序:如果 *s* 和 *s*′属于同一个方法,*s* 控制 *s*′或者 *s*′后控制 *s*,则有 $s \xrightarrow{po} s'$.当语句 *s* 和 *s*′属于不同的方法时,需要将它们映射成同一个方法中的两条语句,然后比较这两条语句的程序序.图 6 显示了这样的一个例子.

图 6 的程序只有一个线程.语句 s_3 和 s_4 分别属于方法 *foo* 和方法 *bar*.s_3 与 s_4 之间的程序序是这样决定的:语句对(s_3,s_4)被映射成语句对(s_6,s_7);语句 s_6 和 s_7 位于方法 *main* 中,且在方法 *main* 的控制流图上,s_6 控制 s_7.即有

$s_6 \xrightarrow{po} s_7$,因此有 $s_3 \xrightarrow{po} s_4$.显然,方法 *main* 首先调用方法 *foo*(语句 s_6),执行语句 s_3 之后,方法 *foo* 返回;然后调用方法 *bar*(语句 s_7),执行语句 s_4.因此,s_3 在程序序上先于 s_4.

```
1  public class Main {
2     static int cnt;
3     static void foo() {cnt++;}
4     static void bar() {cnt−−;}
5     public static void main() {
6        foo();
7        bar();
8     }
9  }
```

Map: $(s_3,s_4) \longrightarrow (s_6,s_7)$

$s_3 \xrightarrow{po} s_4$

$s_6 \xrightarrow{po} s_7$

Fig.6 How to get the program order among two statements in different methods

图 6 如何计算属于不同方法的两个语句之间的程序序

图 7 给出判断语句程序序和进行语句对映射的算法伪码.

```
1   po_before(a,b)
2   {
3       (a′,b′)=Map(a,b);
4       if (dominate(a′,b′)||
5               post_dominate(b′,a′))
6           return true;
7
8       return false;
9   }
10  Map(a,b) {
11      p1=a.path;
12      p2=b.path;
13      c=last_common(p1,p2);
14      a=next(c,p1);
15      b=next(c,p2);
16      return (a′,b′);
17  }
```

Fig.7 Program order and statement pair mapping

图 7 程序序和语句对映射

方法 *Map* 将语句对(*a*,*b*)映射成语句对(*a*′,*b*′).当 *a* 和 *b* 属于同一个方法时,(*a*′,*b*′)就是(*a*,*b*).我们仍然以图 6 的程序为例说明 *Map* 算法.*a.path* 表示从方法 *main* 到语句 *a* 的路径,它是方法和调用语句构成的一个序列.对图 6 所示的程序 s_3.*path* 的值为"*main*.s_6.*foo*",它表示方法 *main* 含有语句 s_6,而 s_6 调用方法 *foo*;类似地,s_4.*path* 的值为"*main*.s_7.*bar*".方法 *last_common* 返回两条路径中共同前缀的最后一个元素,因此,*last_common*(s_3.*path*,s_4.*path*)=*main*.方法 *next*(*c*,*p*)返回路径 *p* 中元素 *c* 的后继元素,因此,*next*(*main*,*main*.s_6.*foo*)返回值 s_6,*next*(*main*,*main*.s_7.*bar*)返回值 s_7.最后,语句对(s_3,s_4)被映射成(s_6,s_7).

2.5 其他同步结构

除了线程派生和结束引起的同步之外,在多线程程序中还会使用其他类型的同步.例如,栅障同步让先执行到栅障的线程等待,直到所有的线程都执行到栅障为止.栅障之前的区域与栅障之后的区域是不能并行的. Java 语言并不提供栅障同步的结构,栅障只能由程序员自己实现.有多种方法可以实现栅障同步[16],因此,它的实现因没有固定的形式而难以分析.据我们所知,目前尚未有发表的工作使用静态分析来检测栅障同步.

Java 多线程程序中另一种常用的同步是 *wait*,*notify* 和 *notifyall* 方法.在 C/C++语言中,与之对应的同步结构是 POSIX 线程库提供的 *pthread_cond_wait* 和 *pthread_cond_signal* 函数.不论是 Java 还是 C/C++,这种同步结构的使用都必须与锁的使用相配合.以 Java 语言为例:一个线程首先获得一个锁,当它调用 *wait* 方法时会释放锁并睡眠;当该线程以后被唤醒时,需要重新获得之前释放的锁才能继续往下执行.在锁的保护下,多个调用 *wait* 方法在同一个对象上等待的线程之间才不会出现竞争条件(race condition).一个线程对 *notify* 或者 *notifyall* 的调用不会释放任何它已经获得的锁.正如文献[17]所指出的那样,从数据竞争检测的角度来看,*wait*-*notify*/*notifyall* 同步可以视为锁同步.

本文提出的 MHP 和 Barik 算法都只对线程派生和结束所引起的同步进行分析,对栅障同步和锁同步都不

作处理.

3 实验结果

MHP 算法被实现在一个静态的程序分析工具 Chord 中[17].Chord 使用一系列的分析,包括别名分析、方法调用图分析、线程逃逸分析和锁同步分析,来逐步求精可能发生数据竞争的冲突访存对的集合.Chord 采用上下文敏感的分析,我们在进行实验时,对 Java 中的实例方法采用 *k*-object 敏感的上下文敏感[18,19],参数 *k* 设定为 3,而对 Java 中的静态方法采用复制上下文敏感(context-copy sensitivity)[17].

静态分析一般会报告大量虚假的警告.对 Chord 来说,虚假警告有两个主要的来源:

一方面,别名分析对数组访问不能精确地进行分析,对一个数组的所有访问都被认为是互相别名的.

另一方面,Chord 没有进行 MHP 分析.

对于第 1 个原因,目前静态分析找不到更好的方法来解决,因此数组引起的虚假警告是不可避免的.对于第 2 个原因,在应用了我们的 MHP 算法之后,大量的虚假警告被过滤掉了.

我们选择了 14 个 Java 程序对 MHP 算法进行测试.测试的前 11 个程序来自 Java Grande 基准测试程序集[20]:

- barrier,forkjoin,sync 这 3 个程序包含了一些简单的同步操作;
- crypt,lufact,series,sor,sparseMatmult 这 5 个程序是一些常见应用的核心算法;
- moldyn,montecarlo,raytracer 是完整的应用程序;
- tsp 是解决推销员旅行问题(traveling salesman problem)的算法程序;
- mtrt 来自 SPECJVM98 基准测试程序集[21],它也是一个 ray tracer 程序;
- Jbb 是 SPEC JBB2000 商业基准测试程序[22].

在这些程序上检测数据竞争的结果见表 1.

Table 1 Results of data race detection

表 1 数据竞争检测结果

Program	Size	#Real races	#Races (MHP)	#Races (original)
barrier	199	1	4	35
forkjoin	146	0	0	0
sync	209	2	4	21
crypt	655	0	72	442
lufact	1 041	1	45	137
Series	384	0	6	28
Sor	291	2	96	160
Matmult	285	0	2	23
Moldyn	846	1	219	435
Montecarlo	3 128	10	10	109
Rattracer	1 431	1	159	427
Tsp	706	11	19	223
Mtrt	3 751	2	17	50
jbb	30 486	46	46	211
Total	52 211	77	699	2 301

表 1 的第 2 栏“Size”是每个测试程序的代码行数.第 3 栏“#Real races”是检测到的真正的数据竞争的个数.第 4 栏“#Races (MHP)”是应用了 MHP 分析之后,报告的总的数据竞争的个数.第 5 栏“#Races (orignial)”是没有使用 MHP 分析时报告的总的数据竞争的个数.

总共检测到 77 个真实的数据竞争.当应用 MHP 算法时,报告了 699 个数据竞争,除去 77 个真实的数据竞争外,其他 622 个警告都是虚假的信息.当不应用 MHP 算法时,报告的数据竞争的个数是 2 301.

可以看到,使用 MHP 可以过滤掉 1 602 个虚假的警告信息,以 2 301 为分母,有 69%的虚假警告被过滤掉,MHP 算法有效地提高了 Chord 检测数据竞争的精度**.

** 本文使用的 14 个 Java 程序都是父线程 join 同步子线程.因此,对这 14 个 Java 程序应用 Barik 算法和本文提出的 MHP 算法将具有同样的分析结果.

4 新算法与 Barik 算法的比较

目前,针对 Java 多线程程序的最好的 MHP 分析是 Barik 提出的基于 TCT 的分析算法.我们将新算法与 Barik 算法进行比较,从以下两方面说明新算法的创新性.

4.1 算法设计逻辑比较

Barik 算法以线程创建树 TCT 为核心进行分析.TCT 中的节点是线程,一个节点的孩子节点是其派生的子线程.TCT 用来帮助描述线程之间的 ***start*** 同步.TCT 上的每个节点具有布尔类型的 *must-join* 属性,以此来描述线程之间的 ***join*** 同步.当一个节点的 *must-join* 属性为真时,表示该节点被其父线程调用 *join* 方法.因此,Barik 算法将 *start* 同步与 *join* 同步分析紧密结合在一起,两者的关系是紧耦合(tightly coupled)的.紧耦合的设计使得 Barik 算法存在两个缺陷:

(1) Barik 算法在处理 ***start*** 同步和 ***join*** 同步时需要考虑 *start* 与 *join* 的各种组合,处理逻辑较为复杂;

(2) Barik 算法假设子线程只会被父线程调用方法 *join* 同步的情况,因此无法处理任意两个线程之间的 *join* 同步.

Barik 算法定义了两层并行性:线程级并行和节点级并行.两条语句如果满足这两层并行性,则它们可以并行执行.Barik 算法的处理逻辑见表 2.

Table 2 Barik algorithm's processing logic

表 2 Barik 算法的处理逻辑

线程级并行	(1) t_p *join* t_1, t_p *join* t_2; (2) t_p *join* t_1, t_p *not-join* t_2; (3) t_p *not-join* t_1, t_p *not-join* t_2;
节点级并行	(4) t_1 *join* t_2; (5) t_1 *not-join* t_2;

t_p 是 TCT 上 t_1 与 t_2 共同的最近的祖先节点.在已知线程 t_i 派生了线程 t_j 的条件下,t_i *join* t_j 表示 t_i 与 t_j 之间存在 ***join*** 同步;t_i *not-join* t_j 表示两者之间没有 ***join*** 同步.

与 Barik 算法不同,我们的 MHP 算法以一种非耦合(decoupled)的方式来处理 ***start*** 同步和 ***join*** 同步.新算法使用关系 ***TStartT*** 描述 ***start*** 同步,使用关系 ***TJoinT*** 来描述 ***join*** 同步,将 ***join*** 同步的处理与 ***start*** 同步的处理分离开来.新算法的处理逻辑见表 3.

Table 3 New MHP algorithm's processing logic

表 3 新算法的处理逻辑

(1) t_1 与 t_2 满足 ***TStartT*** 关系;
(2) t_1 与 t_2 满足 ***TJoinT*** 关系;
(3) t_1 与 t_2 满足 ***TTTJS*** 关系;

与 Barik 算法不同,新算法通过排除串行性来判断语句 s_1 与 s_2 是否可以并行:如果在表 3 的 3 种情况下语句 s_1 与 s_2 之间都不存在同步,那么它们就可以并行执行.显然,新算法的逻辑更加简单.

TCT 节点的 *must-join* 属性只能表示子线程被父线程 *join* 同步的情况.这种情况非常普遍,在这种情况下,Barik 算法和新算法分析的准确性是相同的.但新算法没有将 ***join*** 同步限制在特定的情况下,它可以处理任意两个线程之间的 ***join*** 同步,因此,新的 MHP 算法比 Barik 算法更完备(sound).虽然子线程被父线程 *join* 同步的情况非常普遍,但新算法是在保持分析的准确性和获得简洁性的同时获得更好的完备性的.

4.2 算法可扩展性比较

Barik 算法需要将线程中执行的所有方法内联到线程初始执行的入口方法中,这种做法本身就不具有很好的扩展性.一般而言,内联作为一种编译优化只会将内联的对象限定在一些较小的方法中.即使如此,仍然可能造成程序的代码膨胀——设想一种方法 M 在程序中被多处调用,属于频繁使用的方法,内联这样的方法会导致

代码急剧膨胀.而 Barik 算法所采取的内联策略是不加区分地内联所有方法,因此对于大的程序,其内联的结果是不可接受的.

抛开内联带来的代码膨胀的问题,Barik 算法在内联完毕之后的入口方法的控制流图上计算控制和后控制信息.在一个有 n 个基本块的控制流图上,计算控制信息的时间复杂度是 $\Theta(n^2)$[23].假设程序中有 k 个线程,每个线程有若干方法,第 i 种方法含有的基本块的个数是 n_i.Barik 算法将线程调用的所有方法内联到线程的入口方法,入口方法的基本块的个数为 $\sum n_i$,在入口方法的控制流图上计算控制信息的时间复杂度为 $\Theta((\sum n_i)^2)$,计算所有线程的控制信息的时间复杂度是 $\Theta(k(\sum n_i)^2)$. 新算法只需分别计算每种方法的控制信息,其时间复杂度是 $\Theta(k\sum n_i^2)$. 测试程序中,$\sum n_i^2$ 的值远远小于 $(\sum n_i)^2$,即新算法计算控制信息的时间复杂度远远小于 Barik 算法计算同样信息的时间复杂度.我们按照 Barik 算法的做法来计算控制信息,将其时间开销与我们计算控制信息的开销进行比较,见表 4.

Table 4 Time of computing dominator and post-dominator information

表 4 计算控制和后控制信息的时间

Program	#Threads	#Methods	n_i (average)	$k\sum n_i^2$	$k(\sum n_i)^2$	#Time 1	#Time 2	#Time 3
barrier	3	26	7.2	2 397	21 237	1ms	14ms	1m25s
forkjoin	2	22	6.1	1 238	16 666	2ms	11ms	1m29s
sync	3	25	6.4	1 794	18 786	1ms	12ms	1m31s
crypt	3	32	6.5	2 037	35 921	2ms	61ms	1m42s
lufact	2	38	9.3	5 295	71 965	3ms	118ms	1m35s
series	2	29	5.6	1 255	19 445	1ms	17ms	1m33s
sor	2	27	6.7	2 352	22 472	1ms	18ms	1m39s
matmult	2	26	6.8	2 532	28 642	5ms	11ms	1m36s
moldyn	2	43	8.7	19 188	74 938	9ms	53ms	1m12s
montecarlo	2	210	4.5	5 337	450 305	5ms	254ms	1m34s
raytracer	2	126	4.6	4 144	164 810	6ms	165ms	1m12s
tsp	2	21	14.5	7 331	50 125	2ms	33ms	1m21s
mtrt	2	2 914	4.0	178 428	136 380 584	42ms	3m47s512ms	11m22s
jbb	2	2 435	7.1	264 487	195 337 012	34ms	14m04s943ms	33m49s

表 4 的第 2 栏“#Threads”表示每个程序中线程的个数.第 3 栏“#Methods”是方法的个数.第 4 栏“n_i (average)”表示每种方法平均具有的基本块的个数;大部分方法含有的基本块都小于 10.第 5 栏和第 6 栏分别给出 $k\sum n_i^2$ 和 $k(\sum n_i)^2$ 的值,前者是我们的 MHP 算法计算控制信息的复杂度边界,后者是 Barik 算法的计算控制信息的复杂度边界.这两栏的数据显示,理论上,对计算控制信息的复杂度而言,我们算法的方式远远小于 Barik 算法的方式.

最后 3 栏是测试的时间值,其单位分别为分(m)、秒(s)和毫秒(ms).“#Time 1”是计算每种方法的控制信息的时间,累加得到的总和;这是新算法所采取的方式.“#Time 2”是使用 Barik 方法计算的内联了其他所有方法之后的入口方法上的控制信息所花费的时间.“#Time 3”是检测工具测试整个程序所花费的总时间,MHP 分析采用的是新算法.

“Time 1”和“Time 2”分别是新 MHP 算法和 Barik 算法计算控制信息各自所需的时间.正如理论所预测的那样,对所有的测试程序,后者的值都明显大于前者.对前 12 个程序而言,计算控制信息的时间是毫秒级别的,与检测整个程序的总时间相比微不足道,可以忽略.但是对于 mtrt 和 jbb 两个大程序,实验显示,我们的算法有更好的扩展性,计算控制信息的时间虽然明显增加,但仍然控制在毫秒级别.而 Barik 算法计算控制信息的时间急剧增加,达到了不可忽视的程度:计算 mtrt 的控制信息有 3m47s,计算 jbb 的控制信息时间长达 14m04s.表 4 的实验结果表明,我们提出的新算法比 Barik 算法具有更好的可扩展性.

5 结束语

本文实现了一个针对 Java 语言的新的 MHP 分析算法,与目前 Java 上已知最好的 Barik 算法相比,新的 MHP 算法采取非耦合的方式处理 ***start*** 同步和 ***join*** 同步,设计逻辑更简单,也更完备.同时,新算法不需要在全局的控制

流图计算控制和后控制信息,具有良好的扩展性.在 14 个实际的 Java 应用程序上实验统计得到的数据表明,新的 MHP 算法的开销远远低于已有的 MHP 算法开销.

References:

[1] Krinke J. Static slicing of threaded programs. In: Proc. of the ACM SIGPLAN/SIGSOFT Workshop on Program Analysis for Software Tools and Engineering. New York: ACM Press, 1998. 35−42. [doi: 10.1145/277631.277638]

[2] Masticola S, Ryder B. A model of Ada programs for static deadlock detection in polynomial time. In: Proc. of the Workshop on Parallel and Distributed Debugging. New York: ACM Press, 1991. 97−107. [doi: 10.1145/122759.122768]

[3] Naumovich G, Avrunin GS, Clarke LA. Data flow analysis for checking properties of concurrent Java programs. In: Proc. of the 21st Int'l Conf. on Software Engineering. New York: ACM Press, 1999. 399−410. [doi: 10.1145/302405.302663]

[4] Zhang LB, Zhang FX, Wu SG, Chen YY. A lockset-based dynamic data race detection approach. Chinese Journal of Computers, 2003,26(10):1217−1223 (in Chinese with English abstract).

[5] Fu H, Cai M, Dong JX, Jin X, Gong Y. Enhanced data race detection approach based on lockset algorithm. Journal of Zhejiang University (Engineering Science), 2009,43(2):328−333 (in Chinese with English abstract).

[6] Wu P, Chen YY, Zhang J. Static data-race detection for multithread programs. Journal of Computer Research and Development, 2006,43(2):329−335 (in Chinese with English abstract). [doi: 10.1360/crad20060221]

[7] Callahan D, Subhlok J. Static analysis of low-level synchronization. In: Proc. of the ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging. New York: ACM Press, 1988. 100−111. [doi: 10.1145/68210.69225]

[8] Duesterwald E, Soffa ML. Concurrency analysis in the presence of procedures using a data flow framework. In: Proc. of the 4th ACM SIGSOFT Workshop on Software Testing, Analysis, and Verification. New York: ACM Press, 1991. 36–48. [doi: 10.1145/120807.120811]

[9] Masticola S, Ryder B. Non-Concurrency analysis. In: Proc. of the 4th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming. New York: ACM Press, 1993. 129−138. [doi: 10.1145/155332.155346]

[10] Naumovich G, Avrunin GS. A conservative data flow algorithm for detecting all pairs of statements that may happen in parallel. In: Proc. of the 6th ACM SIGSOFT Symp. on the Foundations of Software Engineering. New York: ACM Press, 1998. 24–34. [doi: 10.1145/288195.288213]

[11] Agarwal S, Barik R, Sarkar V, Shyamasundar R. May-Happen-in-Parallel analysis of X10 programs. In: Proc. of the 12th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming. New York: ACM Press, 2007. 183–193. [doi: 10.1145/1229428.1229471]

[12] Lee J, Palsberg J. Featherweight X10: A core calculus for async-finish parallelism. In: Proc. of the 15th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming. New York: ACM Press, 2010. 25−36. [doi: 10.1145/1693453.1693459]

[13] Naumovich G, Avrunin GS, Clarke LA. An efficient algorithm for computing MHP information for concurrent Java programs. In: Proc. of the 7th European Software Engineering Conf. Held Jointly with the 7th ACM SIGSOFT Int'l Symp. on Foundations of Software Engineering. London: Springer-Verlag, 1999. 338−354. [doi: 10.1145/318774.319252]

[14] Li L, Verbrugge C. A practical MHP information analysis for concurrent Java programs. In: Proc. of the 17th Int'l Workshop on Languages and Compilers for Parallel Computing (LCPC 2004). Berlin: Springer-Verlag, 2004. 194–208. [doi: 10.1007/11532378_15]

[15] Barik R. Efficient computation of may-happen-in-parallel information for concurrent Java programs. In: Proc. of the 18th Int'l Workshop on Languages and Compilers for Parallel Computing. Berlin: Springer-Verlag, 2005. 152−169. [doi: 10.1007/978-3-540-69330-7_11]

[16] Lea D. Concurrent Programming in Java. 2nd ed., New York: Addison-Wesley, 1999. 305−308.

[17] Naik M, Aiken A, Whaley J. Effective static race detection for Java. In: Proc. of the 2006 ACM SIGPLAN Conf. on Programming Language Design and Implementation. New York: ACM Press, 2006. 308−319. [doi: 10.1145/1133981.1134018]

[18] Milanova A, Rountev A, Ryder B. Parameterized object sensitivity for points-to and side-effect analyses for Java. In: Proc. of the Int'l Symp. on Software Testing and Analysis. New York: ACM Press, 2002. 1−11. [doi: 10.1145/566172.566174]

[19] Milanova A, Rountev A, Ryder B. Parameterized object sensitivity for points-to analysis for Java. ACM Trans. on Software Engineering Methodology, 2005,14(1):1−41. [doi: 10.1145/1044834.1044835]

[20] The Java grande forum benchmark. http://www.epcc.ed.ac.uk/research/java-grande/

[21] SPEC JVM98 benchmarks. http://wwww.spec.org/jvm98

[22] SPEC2000 Java business benchmark. http://www.spec.org/osg/jbb2000/

[23] Aho AV, Sethi R, Ullman JD. Compilers: Principles, Techniques, and Tools. New York: Addison Wesley, 1986. 671−672.

附中文参考文献:

[4] 章隆兵,张福新,吴少刚,陈意云.基于锁集合的动态数据竞争检测方法.计算机学报,2003,26(10):1217−1223.

[5] 富浩,蔡铭,董金祥,金星,龚宜.基于锁集合算法的增强型数据竞争检测方法.浙江大学学报(工学版),2009,43(2):328−333.

[6] 吴萍,陈意云,张健.多线程程序数据竞争的静态检测.计算机研究与发展,2006,43(2):329−335. [doi: 10.1360/crad20060221]

印乐(1977－),男,湖南长沙人,博士,主要研究领域为程序分析.

E-mail: yinle@ict.ac.cn

黄磊(1980－),女,博士,主要研究领域为编译优化.

E-mail: leihuang@ict.ac.cn