计算机研究与发展
Journal of Computer Research and Development
ISSN 1000-1239/CN 11-1777/TP
44(3):536~543, 2007

OpenMP Fortran 程序中死锁的静态检测

王昭飞　　黄　春
(国防科学技术大学计算机学院　长沙　410073)
(wzhaofei1981@163.com)

Static Detection of Deadlocks in OpenMP Fortran Programs

Wang Zhaofei and Huang Chun
(*School of Computer Science*, *National University of Defense Technology*, *Changsha* 410073)

Abstract　The deadlocks related to barriers are one kind of the major factors that cause OpenMP programs to malfunction. Static detection of these hazards can help enhance the correctness of OpenMP programs before they are executed. For convenience of detection, this kind of deadlocks is classified into two categories. By searching and data flow analysis, the first and the second category of deadlocks are detected according to the existence rule and nonuniformity rule respectively. Traditional control flow graph is extended to represent OpenMP programs. For each detected deadlock, backtracking is used to record the related paths in the control flow graph, and static branch prediction is employed to quantify its severity. Based on these ideas, a tool, called C-Checker, to statically detect deadlocks in OpenMP Fortran programs is implemented. Experiments show the C-Checker can effectively detect the deadlocks concerned.

Key words　OpenMP; deadlock; BARRIER; static detection

摘　要　与 BARRIER 相关的死锁是导致 OpenMP 程序失效的重要隐患之一. 对该类隐患的静态检测有助于在 OpenMP 程序运行之前提高其正确性. 为了便于检测,将这种死锁分为两类. 借助搜索与数据流分析分别按照存在性规则和非一致性规则检测第 1 类和第 2 类死锁. 扩展了传统的控制流图以表示 OpenMP 程序. 对于每个检测到的死锁,通过回溯记录控制流图中相关的路径,并利用静态分支预测量化其严重程度. 基于上述思想,实现了一个 OpenMP Fortran 程序中死锁的静态检测工具 C-Checker. 实验表明,该工具能有效地检测 OpenMP 程序中与 BARRIER 相关的死锁.

关键词　OpenMP;死锁;BARRIER;静态检测

中图法分类号　TP311.56

OpenMP 以其良好的灵活性和可移植性成为了当前共享存储并行程序设计的主流[1]. 然而,由于编写并行程序的复杂性,OpenMP 程序容易出错[2]. 程序的正确性保障是 OpenMP 程序设计环境面临的一个重要问题. 与 BARRIER 相关的死锁是导致 OpenMP 程序失效的重要隐患之一. 本文采用静态分析力图在 OpenMP 程序运行之前检测其中的这种死锁,对于提高其正确性很有意义.

程序的表示是静态程序分析的基础. 文献[3]用并发控制流图(concurrent control flow graph, CCFG)表示显式并行的共享存储程序. 该类程序包含 cobegin/coend 和 parloop 两种并行结构及基于事件的同步结构. CCFG 引入的同步边和冲突边能表示线程之间的同步关系和数据访问冲突. Diego 扩展了 CCFG 以表示锁同步和栅栏同步[4].

作为一种静态分析技术,数据流分析考察与变量取值相关的程序属性,能解决到达定值识别等数据流问题[5]. 该技术考察了程序所有的控制流路

收稿日期:2006-04-03;修回日期:2006-09-18
基金项目:国家"八六三"高技术研究发展计划基金项目(2002AA1Z2101,2004AA1Z2210)

径,故可得到保守的分析结果. 文献[4]借助数据流分析寻找显式并行程序中由锁变量确定的互斥区. 该信息可用于检测死锁等锁同步异常.

1 与 BARRIER 相关的死锁及其分类

死锁是指程序中的线程因互相等待而陷入不能继续运行的僵局.OpenMP 中的 BARRIER 指定了一个栅栏同步点以同步当前线程组[1]. 只有当前线程组的所有线程到达某个栅栏同步点后,它们才能继续执行该同步点之后的代码. 若部分线程不能到达该同步点,则线程组的其他线程将在该同步点处永远等待,从而形成死锁. 与 BARRIER 相关的死锁可分为两类:第 1 类是 BARRIER 紧嵌套于一次仅被当前线程组的一个线程执行的区域中;第 2 类是当前线程组的各个线程遇到 BARRIER 的次数不完全相等. 图 1 和图 2 中的 OpenMP 程序分别包含第 1 类和第 2 类死锁.

```
1 program demo1
2    integer omp_get_thread_num
3    isum = 0
4    call omp set_num_threads(4)
5  !$OMP PARALLEL PRIVATE(id)
6    id = omp_get_thread_num()
7  !$OMP CRITICAL
8    call work(id,isum)
9  !$OMP END CRITICAL
10 !$OMP END PARALLEL
11 end program demo1
12 subroutine work(ipart,itotal)
13    itotal = itotal + ipart
14 !$OMP BARRIER
15 end subroutine work
```

Fig. 1 An OpenMP program with the 1st category of deadlock.

图 1 一个包含第 1 类死锁的 OpenMP 程序

```
1 program demo2
2    integer omp_get_thread_num
3    call omp set_num_threads(5)
4  !$OMP PARALLEL PRIVATE(tid)
5    id = omp_get_thread_num()
6    if(id<4) then
7  !$OMP BARRIER
8    else
9       print *,tid
10 end if
11 !$OMP BARRIER
12 !$OMP END PARALLEL
13 end program demo2
```

Fig. 2 An OpenMP program with the 2nd category of deadlock.

图 2 一个包含第 2 类死锁的 OpenMP 程序

若一个区嵌套于另一个区之中且没有并行区嵌套于二者之间,则称前者紧嵌套于后者之中[1]. *demo*1 中第 14 行的 BARRIER 紧嵌套于一次仅由一个线程执行的 CRITICAL 区中. 当前线程组包含 4 个线程且只有执行 CRITICAL 区的线程会遇到该 BARRIER,故 *demo*1 会死锁. 与此类似的情况是:BARRIER 紧嵌套于 MASTER 区,SINGLE 区或由锁变量确定的互斥区中.

5 个线程执行图 2 中的并行区,且编号小于 4 的线程能遇到第 7 行和第 11 行的 BARRIER,而编号为 4 的线程仅遇到第 2 个 BARRIER. 故 *demo*2 会死锁. 当前线程组如何分工以执行 SECTIONS 区和调度类型不是 static 的循环区依赖于实现. 调度类型为 static 的循环区的任务分派在编译时可确定. 但只有知道当前线程组的每个线程在执行所分得的每次迭代中遇到 BARRIER 的数目,才能准确地检测死锁. 这些信息的获取所需代价较高,故保守地认为紧嵌套于 SECTIONS 区或循环区中的 BARRIER 会导致第 2 类死锁.

第 2 类死锁其实包含第 1 类死锁. 之所以把死锁分为两类,是因为针对两类死锁的特征设计不同的检测方法可简化死锁的检测. 只要能确定一次仅由单个线程执行的区域包含 BARRIER,就可保守地认为第 1 类死锁会发生;而第 2 类死锁的检测需要确定当前线程组的每个线程遇到 BARRIER 的所有可能的数目,检测算法较复杂.

2 与 BARRIER 相关的死锁的检测

BARRIER 仅同步当前线程组. 因此,C-Checker 以并行区为单位检测死锁.OpenMP 程序中死锁的静态检测需要解决以下问题:

1) OpenMP 程序的表示. 死锁涉及并行区、BARRIER、CRITICAL、SINGLE 和锁例程等多种 OpenMP API. 合适的 OpenMP 程序表示是死锁检测的基础.

2) 不确定程序行为的分析. 程序行为依赖于分支和循环的控制条件取值,富于变化. 而控制条件的取值可能依赖于程序运行时的外部输入.

3) 有意义的警告信息的生成. 为了辅助用户消除程序中的死锁,错误检测器除了要定位死锁外,还要将其发生的来龙去脉呈现给用户.

2.1 OpenMP 程序的表示

本文在传统的控制流图中引入了并行基本块以表示 OpenMP API,并增加了结构边以便确定指导

命令和控制结构的作用域[5]. 除指导命令对的第 2 条指导命令和 THREADPRIVATE 外,OpenMP Fortran 程序中每个指导命令和加锁及解锁例程由一个并行基本块表示. 结构边包括前向结构边和后向结构边. 前者是由表示指导命令对的第 1 条指导命令的基本块到表示其第 2 条指导命令的基本块的边,以及由控制结构的首基本块到其尾基本块的边. 后者是与前向结构边的端点相同但方向相反的边. 图 3、图 4 和图 5 分别展示了过程 *demo*1,*work* 和 *demo*2 的控制流图. 图中的圆角矩形表示并行基本块,虚线边表示结构边.

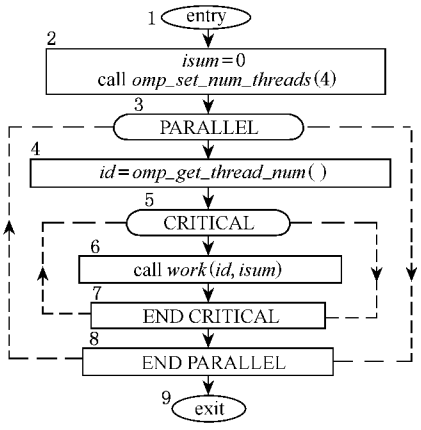

Fig. 3 The control flow graph of procedure *demo*1.

图 3 过程 *demo*1 的控制流图

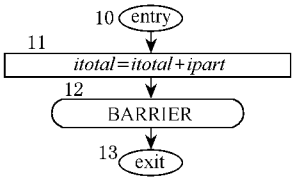

Fig. 4 The control flow graph of procedure *work*.

图 4 过程 *work* 的控制流图

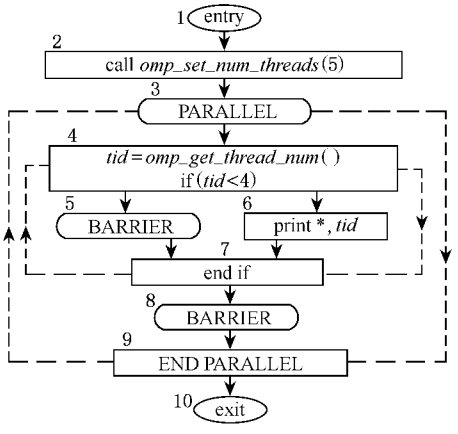

Fig. 5 The control flow graph of procedure *demo*2.

图 5 过程 *demo*2 的控制流图

调用图的边信息中记录了与边尾相接的结点表示的过程对与边头相接的结点表示的过程的所有调用点[5]. 为了便于通过回溯记录跨过程的死锁路径,本文在调用点信息中增加了表示调用点是否活跃的标志 *active* 及调用过程中回溯的终点. 若 C-Checker 正在检查给定的被调过程,则称与该次调用相关的调用点是活跃的. C-Checker 在分析程序时若遇到过程调用,则记录调用过程中回溯的终点并将 *active* 置为真,然后转去分析被调过程,等分析完被调过程后再将该标志置为假. 图 6 给出了程序 *demo*1 的调用图.

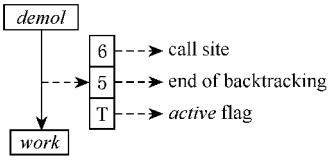

Fig. 6 The call graph of program *demo*1.

图 6 程序 *demo*1 的调用图

2.2 不确定程序行为的分析

程序控制流图中的任意路径在运行时是否被执行是一个不可判定问题[6]. 因此,C-Checker 在不能做出上述判断的情况下保守地认为其中的每条路径都可能被执行,并使用数据流分析来处理不确定程序行为. 这种做法的关键是把待解决的问题转换为已有的数据流问题.

到达定值识别是一类典型的前向数据流问题[5],目的是确定哪些变量定值可以到达给定的程序点. 对某个变量的一次定值可以到达给定的程序点,是指控制流图中至少存在一条从此次定值点到该程序点的路径,且该变量在此路径上没有被重新定值[5]. 数据流问题可由一组数据流方程刻画,其答案可通过求解数据流方程组获得.

2.3 对检测到的死锁的解释

为了便于用户核实并消除检测到的死锁,C-Checker 通过回溯记录控制流图中与死锁相关的路径. C-Checker 在给定过程中的每次回溯记录该过程中的一条路径,并在必要时记录后续回溯的起点. 回溯的起点是导致死锁的 BARRIER 对应的基本块或者在以前回溯时记录的起点. 回溯的终点是与死锁相关的代码区域的首基本块.

C-Checker 在回溯的每一步根据一定的条件选择当前基本块的一个前驱加入路径,并把符合条件但暂时没被选中的前驱记录到队列中. 在一次回溯完成后,C-Checker 根据保存到队列中的基本块和以

往记录的路径多次回溯就能找到用于解释相同死锁的其他路径.若给定过程中回溯的终点是其入口基本块,则表明所找的路径是跨过程的.此时,C-Checker 首先根据调用图找到活跃的调用点,并从中获得调用过程中回溯的终点.然后,C-Checker 从该调用点开始继续回溯就能找到跨过程的路径在调用过程中的部分.

为了便于用户优先排查发生概率较大的死锁,C-Checker 利用静态分支预测估计死锁路径的执行概率以量化死锁的严重程度.Thomas 等人总结了一组用于静态地预测分支转移的启发式规则[7].据此得到量化的静态分支预测规则如表 1 所示.对于不属于该表中所给类别的分支,C-Checker 根据机会均等的原则估计其执行概率.即若某个分支点产生了 n 个分支,则每个分支的执行概率为 $1/n$.C-Checker 在回溯过程中的每一步按上述规则估计由下一次加入路径的基本块转移到当前基本块的概率.C-Checker 假设程序运行时一条路径上各个分支点的转移彼此无关,并通过计算这些分支点的转移概率的乘积得到路径的执行概率.

Table 1 Heuristic Rules for Static Branch Prediction
表 1 静态分支预测的启发式规则

Category		Branch	Probability
branches of loop header		branch leading to loop body	0.8
		branch leading beyond loop	0.2
branches of if construct	equality comparison of real data	true branch	0.1
		false branch	0.9
	integer data no less than 0	true branch	0.8
		false branch	0.2
	equality comparison of pointers	true branch	0.1
		false branch	0.9

2.4 第 1 类死锁的检测

C-Checker 根据存在性规则检测第 1 类死锁.该规则是指只要 BARRIER 紧嵌套于 CRITICAL 区、MASTER 区、SINGLE 区或由锁变量确定的互斥区中,第 1 类死锁就会发生.前 3 种区域的作用域容易由相关的指导命令对确定,但识别由任意锁变量确定的互斥区并非易事.OpenMP 提供了简单锁和嵌套锁.对于已被加锁的简单锁,线程只有在该锁被释放后才能再次对其加锁.而对于已被加锁的嵌套锁,占有该锁的线程在解锁之前可重复地对其加锁.每个嵌套锁有一个嵌套数,每次对其加锁和解锁分别使该嵌套数增 1 和减 1.仅当其嵌套数为 0 时,嵌套锁才处于未加锁状态.C-Checker 借鉴了 Diego 的做法[4],利用到达定值识别检测由锁变量确定的互斥区中的死锁.

2.4.1 前 3 种区域中第 1 类死锁的检测

为了检测上一段提到的前 3 种区域中的第 1 类死锁,C-Checker 顺序地扫描这些区域的控制流图中的基本块,寻找紧嵌套于这些区中的 BARRIER.若被考察的基本块包含过程调用,则 C-Checker 转去扫描被调过程.等分析完被调过程后,C-Checker 再考察该基本块的剩余部分.若找到了符合条件的 BARRIER,则 C-Checker 首先由其对应的基本块开始向该 BARRIER 所在区域的首基本块回溯并记录所经过的路径.然后,Checker 根据所得的路径信息输出警告.

在回溯过程中,若上次加到路径中的基本块是控制结构的尾基本块或指导命令对的第 2 条指导命令对应的基本块,则 C-Checker 根据后向结构边将该控制结构的首基本块或该指导命令对的第 1 条指导命令对应的基本块加入路径.否则,C-Checker 任意选择当前基本块的一个前驱加入路径.这种做法既能保持路径的有效性,又能减少路径的数目和长度.

2.4.2 由锁变量确定的互斥区中死锁的检测

C-Checker 认为与每个锁变量相关的加锁和解锁例程均产生一个对该锁变量的定值,且只有这些例程才产生对该锁变量的定值.设 SL 和 NL 分别表示简单锁变量和嵌套锁变量,BB 表示并行区内不紧嵌套于 SECTIONS 区和循环区及上文给出的前 3 种区域中的 BARRIER 对应的基本块.根据存在性规则,可知以下两个命题成立.

命题 1. BB 可能位于由 SL 确定的互斥区中,当且仅当存在可以到达 BB 入口处的由加锁例程

产生的对 SL 的定值.

命题 2. BB 可能位于由 NL 确定的互斥区中,当且仅当存在可以到达 BB 入口处的对 NL 的定值,且该定值由加锁例程产生或者由不会使 NL 的嵌套数变为 0 的解锁例程产生.

C-Checker 检测由锁变量确定的互斥区中死锁的过程如下:首先,利用到达定值识别确定可以到达并行区中基本块入口处的锁变量定值. 然后,根据命题 1 或命题 2 检测死锁. 在识别可以到达 BB 入口处的由不会使 NL 的嵌套数变为 0 的解锁例程产生的对 NL 的定值时,C-Checker 记录相关的嵌套锁例程调用序列. 若检测到了死锁,则 C-Checker 分以下两种情况记录控制流图中相关的路径:

第 1,若存在可以到达 BB 入口处的由加锁例程产生的锁变量定值,则对于每个这样的定值,C-Checker 由 BB 开始回溯直到产生该定值的加锁例程对应的基本块并记录所经过的路径. 除最后一个加入路径的基本块外,C-Checker 要求该定值可以到达路径中其他每个基本块的入口处和出口处. 第 2,若存在可以到达 BB 入口处的由解锁例程产生的对 NL 的定值且该例程不会使 NL 的嵌套数变为 0,则 C-Checker 由 BB 开始回溯直到产生该定值的解锁例程对应的基本块. 回溯过程中选择加入路径的基本块的思路与情况 1 的做法相同.

在第 1 种情况下,Checker 根据所得的路径输出警告. 在第 2 种情况下,Checker 根据得到的路径和本节提到的嵌套锁例程调用序列输出警告.

2.5 第 2 类死锁的检测

C-Checker 根据非一致性规则检测第 2 类死锁. 该规则是指若当前线程组的各个线程遇到 BARRIER 的次数不完全相等,则第 2 类死锁将发生. 基于第 1 节的分析,本文采用与第 2.4.1 节类似的方法检测 SECTIONS 区和循环区中的第 2 类死锁. 对于其他情形的第 2 类死锁,C-Checker 引入了一个虚变量,并认为每个 BARRIER 产生一个虚变量定值且只有 BARRIER 才产生这样的定值. 每个并行区的结束处隐含了一个 BARRIER[1]. 为了便于分析,C-Checker 假设每个并行区的开始处也隐含一个 BARRIER,并引入了 BARRIER 序列.

定义 1. BARRIER 序列是指满足如下性质的序列:

1) 序列中的每个元素均为 BARRIER;

2) 序列中的第一个和最后一个元素分别是并行区的开始和结束处隐含的 BARRIER;

3) 除最后一个元素外,序列中的其他每个元素产生的虚变量定值可以到达下一个元素对应的基本块的入口处.

C-Checker 为每个 BARRIER 关联了一个同步点数. 该数表明了线程在执行并行区的过程中遇到每个 BARRIER 指定的栅栏同步点的数目,其计算规则如下:若 BARRIER 不位于循环中,则其同步点数为 1. 否则,分情况处理. 若该循环的迭代空间大小在编译时能确定,则该 BARRIER 的同步点数就是其值;否则,令该 BARRIER 的同步点数为 $*$. 由 BARRIER 的同步点数不难定义 BARRIER 序列的同步点数及其计算规则. 若给定的 BARRIER 序列包含同步点数为 $*$ 的 BARRIER,则其同步点数为 $*$. 否则,该 BARRIER 序列的同步点数为其中所有元素的同步点数之和. 根据非一致性规则,可知如下命题成立.

命题 3. 并行区可能包含第 2 类死锁,当且仅当该并行区中 BARRIER 序列的数目大于 1 且各个 BARRIER 序列的同步点数不完全相同或全为 $*$.

若并行区可能包含第 2 类死锁,则 C-Checker 基于线程在执行并行区时经过 BARRIER 序列的概率量化死锁的严重程度. 不妨设 C-Checker 在并行区中一共得到 n 个 BARRIER 序列,且该并行区由 t 个线程执行. 再设第 i 个 BARRIER 序列包含 C_i 个元素,且当前线程组的每个线程经过该序列的概率均为 P_i,$1\leqslant i\leqslant n$. 设每个线程在执行并行区时遇到 d 个 BARRIER 的概率为 Q_d,则 $Q_d=\sum P_k$. 其中,$1\leqslant k\leqslant n$ 且 $C_k=d$. 令 $D=\{d\mid$ 存在 i,使 $d=C_i,1\leqslant i\leqslant n\}$. 设第 2 类死锁发生的概率为 $prob$,则 $prob=1-\sum Q_j^t$,其中的 j 满足 $j\in D$.

C-Checker 检测第 2 类死锁的过程如下. 首先,采用可达虚变量定值识别确定可以到达并行区中基本块入口处的虚变量定值. 然后,由并行区的尾基本块开始向其首基本块回溯并记录该并行区中的 BARRIER 序列. 在回溯过程中,C-Checker 要求下一次加入序列的 BARRIER 产生的虚变量定值可以到达当前 BARRIER 对应的基本块的入口处. 最后,按照命题 3 检测第 2 类死锁. 若检测到第 2 类死锁,则 C-Checker 根据所得的 BARRIER 序列和死锁发生的概率生成警告.

3 静态检测器的实现及评测

CCRG OpenMP 是一个 OpenMP Fortran 并行编译器[8],由预编译器和 Fortran 编译器组成. 预编译器由语法分析器和并行转换器组成. 其中,前者把 OpenMP Fortran 源程序转换为语法树中间表示;

后者根据 OpenMP 的语义重构得到的中间表示，并生成等价的添加了运行时库调用的 Fortran 程序. C-Checker 在并行转换器对 OpenMP 程序的中间表示重构之前检测死锁.

下文给出了用 OpenMP 程序评测 C-Checker 所得的结果. 实验环境如下：Pentium Ⅳ 2.4GHz CPU，512MB 主存，内核为 2.4.21 的 Red Hat Linux，CCRG OpenMP compiler 1.0. 图 7 是一个包含嵌套锁的 OpenMP 程序，图 8 是其控制流图. 把 C-Checker 应用于 *demo*1，*demo*2 和 *demo*3 所得的结果如表 2 所示. 用该工具检查取自 SPEComp2001 的程序所得的结果如表 3 所示[9].

```
1 program demo3
2   call omp_init_nest_lock(nl)
3   read *,i
4  !$OMP PARALLEL
5   call omp_set_nest_lock(nl)
6   if(i.ge.0)then
7     call omp_set_nest_lock(nl)
8     i=-1
9   else
10    i=2
11  end if
12 call omp_unset_nest_lock(nl)
13 !$OMP BARRIER
14  if(i.eq.-1) then
15    call omp_unset_nest_lock(nl)
16 end if
17 !$OMP END PARALLEL
18 call omp_destroy_nest_lock(nl)
19 end program demo3
```

Fig. 7 An OpenMP program using nestable lock routine.

图 7 一个使用嵌套锁例程的 OpenMP 程序

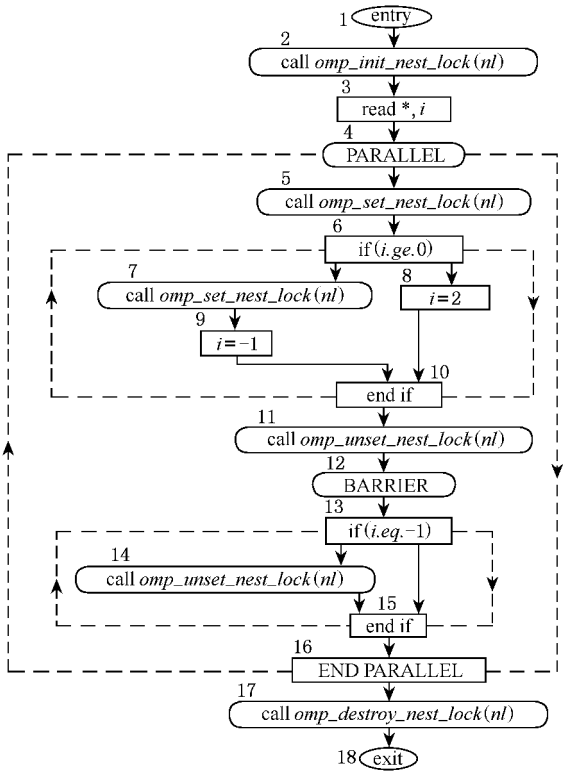

Fig. 8 The control flow graph of program *demo*3.

图 8 *demo*3 的控制流图

Table 2 Results of Applying C-Checker to 3 Examples

表 2 把 C-Checker 应用于 3 个示例程序所得的结果

Program	Barrier Block	Path Related to Deadlock	Execuion Probability
*demo*1	12	5,6,11,2	1.0
*demo*2	5	3,5,8,9 3,8,9	0.9375
*demo*3	12	5,7,11 11,12	0.8

Table 3 Results of Applying C-Checker to SPEComp2001

表 3 把 C-Checker 应用于 SPEComp2001 所得的结果

Program	Size(line)	Deadlock		Program Compilation Time		
		Reported	Real	C-Checker off (s)	C-Checket on (s)	Growth Ratio (%)
swim.f	454	0	0	0.030	0.033	10.0
mgrid.f	552	0	0	0.047	0.053	12.8
gafort.f90	1343	0	0	0.047	0.110	134.0
wupwise.f	2470	0	0	0.080	3.226	4022.5
applu.f	3808	2	0	0.310	0.690	122.6
apsi.f	7666	0	0	0.326	9.433	2793.6
fma.f90	60982	0	0	3.560	109.900	2987.1

C-Checker 报告 *demo*1 中第 14 行的 BARRIER 将导致第 1 类死锁,因为其位于从 CRITICAL 区调用的过程中. C-Checker 还报告 *demo*2 中第 7 行的 BARRIER 可能导致第 2 类死锁. C-Checker 在图 5 中找到了两个 BARRIER 序列,如表 2 中第 3 行第 3 列的单元格所示. 由非一致性规则,C-Checker 认为 *demo*2 可能发生第 2 类死锁. 由表 1 估计执行该并行区的线程经过这两个 BARRIER 序列的概率均为 0.5. C-Checker 再由第 2.5 节的公式估计死锁发生的概率为 0.9375.

C-Checker 报告 *demo*3 中第 13 行的 BARRIER 可能导致死锁. 该 BARRIER 对应于 12 号基本块,且只有第 12 行的解锁例程产生的锁变量定值可以到达其入口处. C-Checker 进一步发现有一个锁例程调用序列表明该例程不会使 *nl* 的嵌套数变为 0,如表 2 中第 4 行第 3 列左边的单元格所示. 根据命题 2,C-Checker 认为第 13 行的 BARRIER 可能位于由锁变量 *nl* 确定的互斥区中,故第 1 类死锁可能发生.

C-Checker 报告 applu.f 中第 3352 行和第 3373 行的 BARRIER 可能导致死锁. 这两个 BARRIER 所在并行区的代码段如图 9 所示. 然而,程序运行时执行该并行区的各个线程遇到 BARRIER 的次数相同. 这意味着检测到的死锁不会发生. C-Checker 在该并行区中共找到了 4 个 BARRIER 序列. 其中,第 1 个序列由该并行区的开始和结束处隐含的 BARRIER 组成,第 2 个和第 3 个序列分别仅比第 1 个序列多第 3352 行和第 3373 行的 BARRIER,第 4 个序列比第 1 个序列多图 9 中的两个 BARRIER. 经过分析发现,线程在执行并行区时仅可能经过上述序列中的第 4 个序列. 对于该并行区中迭代空间的上界为变量的循环,C-Checker 保守地认为其循环体可能一次也不被执行,从而线程可能经过前 3 个 BARRIER 序列. 进而,C-Checker 根据非一致性规则错误地认为 applu.f 可能死锁.

```
!$OMP PARALLEL PRIVATE(npx, npy, …)
  □
do L = 3, npx + npy + nz - 3
  □
!$OMP BARRIER  !line 3352
end do
do L = npx + npy + nz, 2, - 1
  □
!$OMP BARRIER  !line 3373
end do
!$OMP END PARALLEL
```

Fig.9 A code segment of applu.f that contains BARRIERs.

图 9 applu.f 中一个包含 BARRIER 的代码段

wupwise.f,apsi.f 和 fma.f90 是较复杂的程序,故启用死锁检测后它们的编译时间增长显著. 但前两个程序的编译时间仍可接受.

4 讨 论

实验表明,C-Checker 能检测出定制的 OpenMP 程序中的死锁,但还存在不足. 首先,死锁检测的有效性有待进一步评测. 实验检测到的真死锁仅位于小程序中. SPEComp2001 中的程序本身不会死锁,故无法评测 C-Checker 检测真死锁的能力. 其次,保守的分析使 C-Checker 可能产生误报. 最后,死锁检测尚不完备. 目前,C-Checker 还不能检测由并行区调用的过程中的第 2 类死锁和与锁变量相关的第 1 类死锁.

以后拟从 3 个方面改进 C-Checker. 第 1,选取更多较大的含 BARRIER 的真实 OpenMP 程序以评测 C-Checker. 若所选程序本身不会死锁,则要研究程序的错误注入技术. 第 2,在找到与死锁相关的执行路径后,采用符号计算考察该路径能否被执行. 希望通过剔出不被执行的程序路径以降低 C-Checker 的误报率. 符号计算的引入依赖定理证明器或约束求解器的支持. 为此,需要研究如何把它们集成到 C-Checker 中. 第 3,利用过程间的数据流分析实现跨过程的死锁检测. 这样,由并行区调用的过程中的死锁也将能被检测.

5 总 结

本文采用静态分析检测 OpenMP Fortran 程序中与 BARRIER 相关的死锁,并实现了静态检测器 C-Checker 的原型. 该工具设计了适合 OpenMP 程序的死锁检测的程序表示形式. C-Checker 把所关注的死锁分为两类,并借助于搜索和数据流分析分别按照存在性规则和非一致性规则检测第 1 类和第 2 类死锁. C-Checker 通过回溯记录与死锁相关的执行路径,并利用静态分支预测量化死锁的严重程度,从而便于用户核实检测到的死锁及优先排查发生概率较大的死锁.

参 考 文 献

[1] OpenMP Architecture Review Board. OpenMP Fortran application program interface, Version 2.5 [OL]. http://www.openmp.org, 2005-05

[2] P Paul, S Sanjiv. OpenMP support in the Intel thread checker [C]. The 4th Int'l Workshop on OpenMP Application and Tools, Toronto, Canada, 2003

[3] L Jaejin. Compilation techniques for explicitly parallel programs: [Ph D dissertation] [D]. Urbana, Illinois: University of Illinois at Urbana Champaign, 1999

[4] N Diego. Analysis and optimization of explicitly parallel programs: [Ph D dissertation] [D]. Edmonton: University of Alberta, 2000

[5] S Steven. Advanced Compiler Design and Implementation [M]. San Francisco, California: Morgan Kaufmann, 1997

[6] S Micha, P Amir. Two approaches to interprocedural data flow analysis [G]. In: Program Flow Analysis: Theory and Applications. Engiewood Cliffs, New Jersey: Prentice-Hall, 1981. 189-233

[7] B Thomas, L James. Branch prediction for free [J]. ACM SIGPLAN Notices, 1993, 28(6): 300-313

[8] Huang Chun, Yang Xuejun. CCRG OpenMP: Experiments and improvements [C]. The 1st Int'l Workshop on OpenMP, Eugene, Oregon, USA, 2005

[9] A Vishal, D Max, E Rudolf, *et al*. SPEComp: A new benchmark suite for measuring parallel computer performance [C]. The 2nd Int'l Workshop on OpenMP Application and Tools, West Lafayette, IN, USA, 2001

Wang Zhaofei, born in 1981. Received his B. A.'s and M. A.'s degree in computer science and technology, from Hebei University, Hebei, China in 2003, and from the National University of Defense Technology, Hunan, China in 2005 respectively. Since 2006, he has been a Ph. D. candidate at the National University of Defense Technology. His current research interests include static program analysis.

王昭飞，1981 年生，博士研究生，主要研究方向为静态程序分析.

Huang Chun, born in 1973. Associate professor at the National University of Defense Technology since 2003. Her main research interests include advanced compiler, parallel programming environment, and embedded system.

黄　春，1973 年生，副研究员，主要研究方向是先进编译器、并行程序设计环境和嵌入式系统.

Research Background

Although OpenMP is used widely for shared memory parallel programming nowadays, OpenMP programs may fail due to deadlocks, data race and so on. Hence, it is valuable to remove these hazards from OpenMP programs. Static analysis examines program code and tries to find software defects before the programs are executed. We use static analysis to detect the deadlocks related to barriers in OpenMP programs. Program flow analysis has become a commonly used static analysis technique. We extend traditional program flow analysis techniques to detect deadlocks. Sometimes the program information needed by deadlock detection is either unavailable or difficult to obtain at compile time. Consequently, C-Checker may do much conservative analysis and report false alarms. To address this problem, we quantified the severity of the reported deadlocks and recorded the related execution paths. Symbolic evaluation may help eliminate infeasible paths and reduce false deadlocks reported. Our work is supported by the 863 High-Tech Research and Develop Program of China.