doi:10.11887/j.cn.202205010 http://journal.nudt.edu.cn

并行程序中同步瓶颈的检测和优化方法*

张 杨,李柳旭
(河北科技大学 信息科学与工程学院,河北 石家庄 050018)

摘 要:针对并发程序中锁的不当使用可能导致性能瓶颈的问题,提出检测和优化并发程序中同步瓶颈的方法IdeSync。IdeSync使用静态分析方法获取同步方法和同步块,构建静态同步依赖图,采用基于执行路径的动态分析技术进行同步依赖关系分析,构建同步依赖图。为了暴露性能瓶颈,在同步依赖图上通过增加程序工作负载的方式,监测临界区的性能变化,并针对检测到的同步瓶颈给出优化建议。在实验中通过HSQLDB、SPECjbb2005和RxJava等12个大型实际应用程序对IdeSync的有效性进行验证,共检测到72个同步瓶颈,根据优化建议进行优化后程序性能均有所提升。实验表明,IdeSync能够有效地检测和优化同步瓶颈。

关键词:同步瓶颈;并行程序;锁;性能优化;程序分析
中图分类号:TP311 **文献标志码**:A **文章编号**:1001-2486(2022)05-092-10

Detection and optimization approaches for synchronization bottlenecks in parallel programs

ZHANG Yang, LI Liuxu
(School of Information Science and Engineering, Hebei University of Science and Technology, Shijiazhuang 050018, China)

Abstract: Aiming at the problem that improper locks in parallel programs may lead to performance bottlenecks, an approach called IdeSync was proposed to detect and optimize synchronization bottlenecks. IdeSync leveraged the static analysis to obtain the synchronized methods and blocks, and constructed a static synchronization dependency graph. The dynamic analysis technology based on the execution path was used to analyze the synchronization dependency and build the synchronization dependency graph. In order to expose the performance bottleneck, the performance change of the critical section was monitored by increasing the program workload on the synchronization dependency graph, and optimization suggestions were given for the detected synchronization bottleneck. The effectiveness of IdeSync was evaluated with 12 large real-world applications such as HSQLDB, SPECjbb2005 and RxJava, and a total of 72 synchronization bottlenecks were detected. All these bottlenecks were optimized based on IdeSync's suggestion to achieve performance improvements, which shows that IdeSync can effectively detect and optimize synchronization bottlenecks.

Keywords: synchronization bottleneck; parallele programs; lock; performance optimization; program analysis

锁是并行程序中比较通用的同步控制方式之一,主要用于在并发程序中确保数据访问和程序状态的正确性。然而在多核时代锁的使用容易导致锁竞争问题,该问题是由多个线程同时竞争临界区资源引起的,如果一个线程获得了锁,那么其他想获取该锁的线程都将被阻塞,直到持有该锁的线程释放锁。由于锁竞争会使多核处理器中只有一个处理核处于运行状态,其他处理核只能等待,严重降低并行程序性能。目前关于锁竞争的研究有很多,例如,Schörgenhumer等[1]通过JVMTI和字节码检测工具来收集有关Java应用程序中锁竞争的详细信息;Patros等[2]通过修改Java虚拟机记录线程驻留时锁竞争的数据信息;Zheng等[3]通过记录程序执行并且跟踪分析来识别不必要的锁竞争;Feng等[4]在程序运行时动态检测操作系统在Java应用程序上引起的锁竞争。

不恰当的同步控制会进一步加剧锁竞争,导致程序性能下降。为此,Rinard等[5]介绍了自适应复制技术,自动消除在对象上执行原子操作的多线程程序中的同步瓶颈。目前还有一些研究通过性能指标来识别性能瓶颈[6-7]。Free Lunch[8]工具通过计算线程在某个时间间隔内获取锁花费的时间百分比检测由锁导致的性能瓶颈。

* **收稿日期**:2021-12-23
基金项目:国家自然科学基金资助项目(61440012);河北省高等学校科学研究计划重点资助项目(ZD2019093)
作者简介:张杨(1980—),男,河北秦皇岛人,副教授,博士,硕士生导师,E-mail:zhangyang@hebust.edu.cn

很多学者对并行程序性能分析进行了研究。Trahay[9]在运行应用程序时收集数据,设计方法来检测 OpenMP 应用程序中的可伸缩性问题。Selakovic 等[10]通过静态模式匹配性能问题。Kroening 等[11]提出了针对 C/Pthread 的静态死锁分析,识别与死锁分析有关的语句,但静态分析有时存在误报率较高的缺陷。Zhang 等开发的框架 VarCatcher[12]使用并行特征向量来分析结果,通过查看不同的执行模式,在多个运行中查找多个程序执行之间会出现性能差异的地方。QURO[13]是一个查询感知的编译器,它可以基于锁竞争自动在事务代码内对查询进行重新排序,同时保留程序语义以提高性能。Cicirelli[14]用 Max-Plus 函数证明了在局部同步情况下效率存在一个非零下限,相对于全局同步具有一定优势。有一些研究[15-16]利用分散的锁管理器来实现高性能,NetLock[17]是新的集中式锁管理器,在不牺牲策略支持灵活性的情况下实现高性能。WAIT[18]工具为了测量锁的使用情况,对获取锁时被阻塞的线程数进行统计。GAPP[19]工具无须通过检查源码就可识别并行程序中的一系列瓶颈。Tallent 等[7]提出并评估了 3 种锁竞争问题,分析了锁竞争导致的性能损失。然而,目前研究在检测性能瓶颈的同时未进一步提出优化建议。

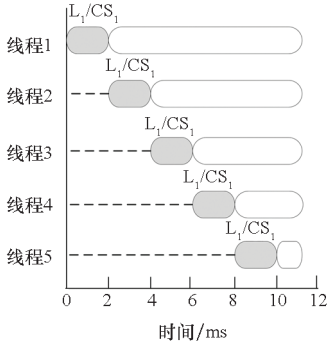
关于针对并发程序的优化,Diniz 等[20]对同一锁上过于频繁的同步进行粗化,以减少锁开销。Zhang 等[21]将内置监视器转换为 StampedLock,通过细化锁的方式减轻锁竞争的影响。Curtsinger 等[22]使用因果分析方法来识别具有优化机会的代码。Yu 等[23]提出了基于跟踪的动态方法来识别一般性能问题,并拆分锁来提升性能。Arbel 等[24]提出了一个用于并发数据结构的代码转换方法,它在不牺牲正确性的情况下提高程序的可伸缩性。

从目前的研究现状来看,锁导致的性能瓶颈检测研究仍存在以下问题:①已有方法采用单一静态分析或动态分析,然而静态分析导致误报率较高,动态分析虽然可以增加准确性,但会导致检测时间显著增加,目前还没有采用动静结合的方法来检测同步瓶颈;②对于锁导致的性能瓶颈认识不清,对于如何检测锁导致的性能缺陷还需要进一步研究;③已有的方法虽然检测了性能瓶颈,但需要对这些性能瓶颈提出进一步优化建议。

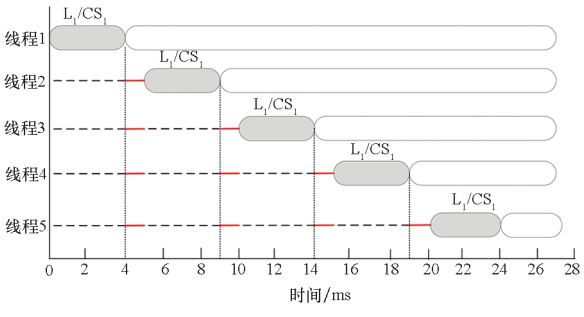
针对目前研究存在的问题,本文提出了同步瓶颈检测和优化方法,该方法针对并行程序中锁相关代码,采用动静结合分析技术分析同步依赖关系,构建同步依赖图。通过增加程序工作负载,监测同步依赖图中所有临界区的变化来检测同步瓶颈。

1　研究动机

在并行程序中,只有一个线程处于运行状态,其他线程处于空闲状态。若临界区中存在耗时且不必要的同步操作,增加程序工作负载后,临界区执行时间增加,加剧锁竞争。这里以 Apache 的 Xalan[25]工具中的 releaseXMLReader( )方法为例说明锁导致的性能瓶颈问题。该方法首先判断阅读管理器是否为空,若不为空则调用 XMLReaderManager 类中的 releaseXMLReader( )方法释放阅读器。该方法的执行情况如图 1 所示,图 1(a)表示该方法增加工作负载之前的执行情况,可以看出线程 1 首先获得锁 L_1 进入临界区 CS_1,当线程 1 释放该锁后线程 2、线程 3、线程 4 和线程 5 依次获取锁 L_1 进入临界区 CS_1,虚线表示等待时间。图 1(b)为增加程序工作负载后该方法的执行情况,可以看出在增加程序工作负载后,在每次临界区执行完之后会有 1 ms 的空闲,这可能是由于该方法中判断阅读管理器是否为空是读操作,使用同步锁导致程序的等待,加剧了其他 4 个线程竞争临界区 CS_1 资源的情况。



(a) 增加负载前
(a) Before increasing the load



(b) 增加负载后
(b) After increasing the load

图 1　releaseXMLReader( )方法的执行
Fig. 1　Execution of the releaseXMLReader( )

2 检测和优化框架

本节首先给出了方法的整体框架,然后对框架中的每个部分进行详细介绍。

2.1 方法框架

检测和优化并发程序中同步瓶颈的方法 IdeSync。该方法首先进行静态同步依赖分析,使用控制流分析生成源程序对应的控制流图,再通过访问者模式分析遍历控制流图,检测源程序中的同步代码块和同步方法。考虑到同一个锁对象可能存在别名现象,使用别名分析找到存在别名的锁对象,增加静态同步依赖关系的准确性。在静态同步依赖关系的基础上结合程序执行路径进行动态同步依赖分析。通过增加程序的工作负载,在程序执行过程中监测同步依赖图中所有临界区的变化,检测同步瓶颈。最后针对发现的同步瓶颈提出优化建议,并通过重构程序进行优化,方法框架如图 2 所示。

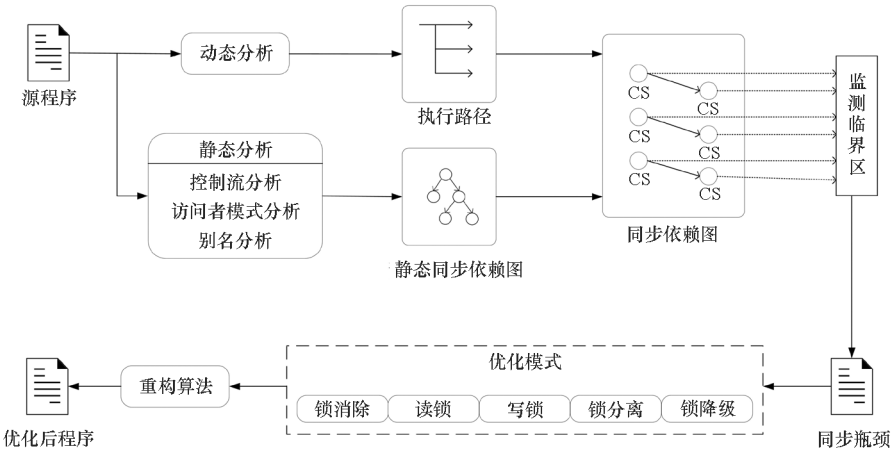


图 2 IdeSync 方法框架

Fig. 2 Framework of IdeSync

2.2 动静结合同步依赖关系分析

本节采用动静结合分析技术分析同步依赖关系,构建同步依赖图,采用动静结合分析技术增加了静态分析同步依赖关系的准确性,同时也降低了动态分析同步依赖关系的分析时间。

2.2.1 静态同步依赖关系分析

在源码的基础上使用控制流分析、访问者模式分析和别名分析静态分析方法。通过控制流分析生成有向控制流图,控制流图上包含程序执行的节点,其中包含监视器的进入和退出指令,同步代码块的开始是监视器的入口指令 MonitorEnter,同步代码块的结束是监视器的出口指令 MonitorExit。然后,使用访问者模式遍历源码,检测被 synchronized 修饰的同步方法,如果该方法是静态同步方法,则监视器对象是当前类的对象,如果该方法是同步方法,则监视器对象是当前类的实例对象。主要分析了以下 4 种同步依赖关系:同步方法之间存在调用关系;同步方法中存在同步块;同步块之间存在嵌套关系;同步块中调用同步方法。

同步依赖图 $T=(P,S)$,P 为结点,S 为边。$P=\{P_1,P_2,\cdots,P_n\}$,P_i 表示源程序中的同步方法或同步块,其中 $1\leqslant i\leqslant n$。$S=\{P_i\rightarrow P_j\}$,表示 P_i 为 P_j 的父结点,P_j 依赖于 P_i。构建静态同步依赖图的算法如算法 1 所示。首先生成临界区对应的中间表示 *IR*,然后通过 *IR* 生成对应的指令集 *Ins*,对每一条指令 i 进行遍历,调用 buildGraph() 方法构建同步依赖图(行 1 ~ 4)。如果指令 i 为方法调用指令,则对该临界区指令集 Ins_1 中的指令 i_1 进行遍历;若指令 i_1 为 MonitorEnter 指令,说明同步方法中存在同步块,则添加结点 P_{M1}、P_{B1} 和边 $S=\{P_{M1}\rightarrow P_{B1}\}$(行 9 ~ 10);若指令 i_1 为方法调用指令,说明同步方法中调用了同步方法,则添加结点 P_{M2}、P_{M3} 和边 $S=\{P_{M2}\rightarrow P_{M3}\}$(行 11 ~ 12);若指令 i 为 MonitorEnter 指令,则对该临界区指令集 Ins_2 中的指令 i_2 进行遍历;若指令 i_2 为方法调用指令,说明同步块中存在同步方法,则添加结点 P_{B4}、P_{M4} 和边 $S=\{P_{B4}\rightarrow P_{M4}\}$(行 18 ~ 19);若指令 i_2 为 MonitorEnter 指令,说明同步块中存在同步块,则添加结点 P_{B5}、P_{B6} 和边 $S=\{P_{B5}\rightarrow P_{B6}\}$(行 20 ~ 21)。被关键字 synchronized 修饰的同步方法和同步块都持

算法 1　构建静态同步依赖图

Alg. 1　Building static synchronization dependency graph

输入:临界区
输出:$T=(P,S)$

```
1.  通过临界区的中间表示 IR 获得临界区的指令集 Ins;
2.  for 遍历指令集 Ins 中每一条指令 i do
3.    buildGraph(i);
4.  end for
5.  char buildGraph(指令 i);
6.  if 指令 i 为同步方法调用指令 then
7.    通过该临界区对应的中间表示 IR 获得临界区的指令集 Ins1
8.    for 遍历指令集 Ins1 中每一条指令 i1 do
9.      if 指令 i1 为 MonitorEnter 指令 then
10.       添加结点 PM1、PB1 和边{PM1→PB1};
11.     else if 指令 i1 为同步方法调用指令 then
12.       添加结点 PM2、PM3 和边{PM2→PM3};
13.     end if
14.   end for
15. else if 指令 i 为 MonitorEnter 指令 then
16.   通过该临界区对应的中间表示 IR 获得临界区的指令集 Ins2
17.   for 遍历指令集 Ins2 中每一条指令 i2 do
18.     if 指令 i2 为同步方法调用指令 then
19.       添加结点 PB4、PM4 和边{PB4→PM4};
20.     else if 指令 i2 为 MonitorEnter 指令 then
21.       添加结点 PB5、PB6 和边{PB5→PB6};
22.     end if
23.   end for
24. end if
```

有相应的锁对象,可能存在锁对象不同但指向同一内存地址,因此需要进行别名分析,增加同步依赖分析的准确性。

以 SPECjbb2005 中 primeWithDummyData( ) 方法为例,部分静态同步依赖情况如图 3 所示,图中每个结点与该结点的子结点存在同步依赖关系,序号 1 ~ 19 代表图中的 19 个结点。从图 3 中可以看出,多个节点之间存在相关的依赖关系,而且节点 2 对节点 7 和 8 都存在依赖关系。

2.2.2　动态同步依赖关系分析

对于给定的一个测试用例,使用动态分析获取程序的执行路径,使用 JProfiler 提供的 API 生成可记录的信息进行分析。首先为执行路径中出现的临界区创建节点,对存在等待关系的临界区添加边,用虚线表示;然后结合静态同步依赖图,判断执行路径中的临界区之间是否存在静态依赖关系,若有则添加边,用实线表示;最后删除执行路径中不存在等待关系和静态依赖关系的节点。

图 4 为 SPECjbb2005 的部分同步依赖图,图中的 CS_2 对应图 3 中的 primeWithDummyData( ) 方法,CS_3 对应图 3 中 loadInitialOrders( ) 方法,由于 CS_2 和 CS_3 存在静态依赖关系,因此添加实线边,线程 1 中的 CS_1 和线程 2 中的 CS_1 之间的虚线表示存在等待关系。

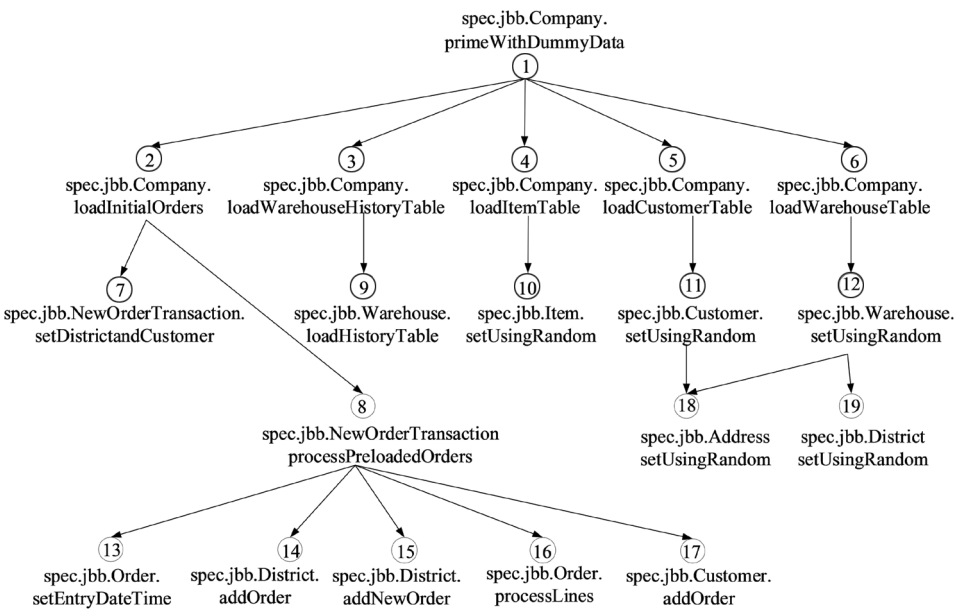


图 3　静态同步依赖图

Fig. 3　Static synchronization dependency graph

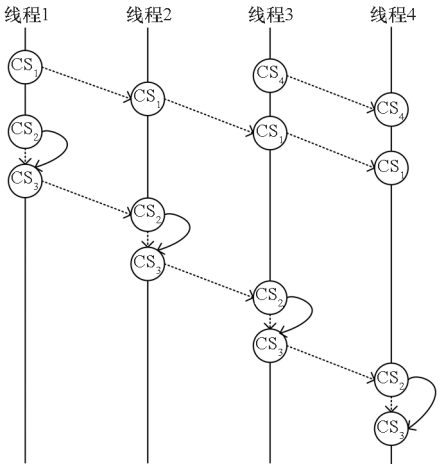


图4　部分同步依赖图

Fig. 4　Partial synchronization dependency graph

2.3　监测临界区

在依赖图的基础上,对临界区的执行进行监控。首先,给定一个测试用例以及测试用例的一组工作负载 $W=\{W_1,W_2,\cdots,W_n\}$,以不同的工作负载执行程序监测临界区在不同工作负载下执行时间、临界区等待时间和 CPU 使用率的变化。需要监测同步依赖图中出现的临界区,并考虑同步依赖图中临界区之间的同步依赖关系来确定同步瓶颈。静态分析可以尽可能发现全部临界区,动态分析需要根据特定的执行路径,会暴露某些执行路径上的执行情况。

判断增加工作负载后程序中的临界区是否满足以下 3 个条件,判定该临界区是否为同步瓶颈。①临界区执行时间明显增加;②临界区的 CPU 使用率稳定;③临界区增加额外等待时间。

随着工作负载的增加,临界区执行时间通常也会增加。当增加工作负载后,如果临界区执行时间明显增加但 CPU 使用率稳定,则该临界区可能存在性能问题。然后,判断该临界区是否增加额外等待时间。将增加额外等待时间定义为:工作负载分别为 W_1 和 W_2 时,因竞争临界区资源增加的等待时间。结合同步等待关系发现同步瓶颈,此外需要根据同步依赖图中临界区之间的静态同步依赖关系进一步确定判定为同步瓶颈。例如,增加工作负载后,若图 4 中在 CS_2 与 CS_3 处增加的额外等待时间相同,则认为 CS_2 增加的额外等待时间是由与其存在静态依赖关系的 CS_3 导致的,因此,判定 CS_2 不是同步瓶颈,CS_3 是同步瓶颈。

2.4　优化

以实现程序最大并行度为原则,对同步瓶颈进行优化。同步锁为互斥锁,读写锁可以提供比互斥锁更好的并行性。首先考虑对不必要的同步进行锁消除来缓解开销,然后对无法进行锁消除的同步瓶颈采用粗粒度的读写锁,即读锁、写锁和锁分解,替换同步锁增加程序的并行度,并在必要时添加细粒度的读写锁来缓解同步瓶颈,即锁降级。因此,使用 5 种同步瓶颈优化模式:锁消除、读锁、写锁、锁分解和锁降级。

1)锁消除:同步依赖图中一条边连接的两个临界区内均只存在读操作。

2)读锁:临界区内只包含读操作。

3)写锁:临界区内只包含写操作。

4)锁分解:临界区内不存在 if 判断语句,但同时存在读操作和写操作。

5)锁降级:临界区内只存在一个 if 语句且 if 语句最后有写操作和至少一个读操作。

负面效应分析用于分析临界区并生成读写字符序列,R 代表读操作,W 代表写操作,如果操作、方法或表达式修改了其本地环境之外的状态,会产生负面效应。遍历指令集中的每条指令,判断临界区内是否有写指令,若有则返回 W,否则返回 R,输出临界区的读写字符序列。根据读写字符序列确定同步瓶颈优化模式并进行重构,具体实现算法如算法 2 所示。首先输入同步依赖图中一条边连接的两个临界区:如果两个临界区为两个不同的临界区,则判断两个临界区内是否只存在读操作,若是,则消除后者的同步锁(行 1 ~ 4);如果两个临界区表示同一个临界区,则调用 refactor( ) 方法分析两个临界区的读写字符序列(行 5 ~ 7),若输入为图 4 线程 1 中的 CS_1 和线程 2 中的 CS_1,则只需分析一次 CS_1 的读写字符序列即可。获取临界区内 if 条件判断语句的数量(行 9 ~ 15),如果临界区内不存在 if 语句,当临界区的读写字符序列为 RW 或 WR 时,将同步锁分解为读锁和写锁(行 17 ~ 18),当临界区的读写字符序列为 R 时,将同步锁替换为读锁(行 19 ~ 20),当临界区的读写字符序列为 W 时,将同步锁替换为写锁(行 21 ~ 22)。如果临界区内存在一个 if 条件判断语句且 if 语句最后有写操作和至少一个读操作时,将进行锁降级(行 24 ~ 25)。如果临界区不满足上述所有条件,则使用写锁(行 26)。

算法 2　重构算法

Alg. 2　Refactoring algorithm

输入:同步依赖图中一条边连接的两个 CS, $<CS_1, CS_2>$

输出:重构结果

```
1.  if CS1 和 CS2 表示两个不同的临界区 then
2.    if(CS1.str == R&& CS2.str == R) then
3.      return 锁消除;
4.    end if
5.  else if CS1 和 CS2 表示同一个临界区 then
6.    refactor(CS1.str);
7.  end if
8.  refactor(str){
9.    通过 CS 对应的中间表示 IR 获取 CS 的指令集 Ins;
10. int j=0;
11. for 遍历指令集 Ins 中每一条指令 i do
12.   if 指令 i 为 if 条件判断指令 then
13.     j ++ ;
14.   end if
15. end for
16. if(j==0) then
17.   if(str == RW || str == WR) then
18.     return 锁分解;
19.   else if (str == R) then
20.     return 读锁;
21.   else if (str == W) then
22.     return 写锁;
23.   end if
24. else if(j == 1&&if 语句最后有写操作和至少一个读操作) then
25.   return 锁降级;
26. else return 写锁;
27. end if
28. }
```

3　实验评估

本节对提出的检测方法 IdeSync 进行评估,首先对实验环境配置和选取的应用程序进行介绍,然后给出实验结果,并对结果进行分析。

3.1　环境配置

所有实验都在配有 2.30 GHz 英特尔酷睿 i5 CPU 和 4 GB RAM 的笔记本电脑上进行,操作系统使用 64 位 Windows10 系统,安装了 JDK 1.8.0_271、Eclipse 4.13.0、WALA 1.5.3[26] 和 JProfiler 11.1.4[27]。

3.2　研究问题

RQ1:使用 IdeSync 方法,是否可以有效发现每个应用程序中的同步瓶颈?

RQ2:检测到的同步瓶颈在 5 个优化模式中的分布情况如何?

RQ3:同步瓶颈优化后的程序性能是否有所提升?

3.3　应用程序

选取的应用程序包括 HSQLDB[28]、Jenkins[29]、JGroups[30]、SPECjbb2005[31]、Xalan[25]、RxJava[32]、Antlr[33]、Fop[34]、Kafka[35]、MINA[36]、Cassandra[37] 和 Jmeter-plugins[38]。HSQLDB 是开源的 Java 数据库,Jenkins 是一个开源的自动化服务器,SPECjbb2005 是 Java 应用服务器测试程序,JGroups 是群组通信工具,Xalan 和 Fop 分别是 Apache 公司的 XSLT 转换处理器和格式化对象处理器,RxJava 是 Netflix 公司的在 Java VM 上使用可观测的序列来组成异步的、基于事件的程序的库,Antlr 是一个解析器生成器,Kafka 是由 Apache 软件基金会开发的开源流处理平台,MINA 是 Apache 公司的网络应用框架,Cassandra 是 Apache 公司的开源分布式 NoSQL 数据库系统,Jmeter-plugins 是 Apache JMeter 工具的一组独立插件。这些应用程序的版本号信息、源代码行数、同步方法和同步块数量见表 1。

表 1　应用程序及其配置

Tab. 1　Application and their configuration

应用程序	版本	源代码行数	同步方法	同步块数量
HSQLDB	2.4.1	175 568	613	71
Jenkins	2.190.2	160 246	227	47
JGroups	4.1.5	122 885	138	41
SPECjbb2005	1.01	12 519	168	22
Xalan	2.7.2	89 149	51	31
Antlr	4.7.2	60 515	3	13
RxJava	2.2.13	99 623	8	20
Fop	2.3	198 555	7	7
Kafka	2.6.0	416 697	1 286	436
MINA	2.1.3	23 482	9	3
Cassandra	3.11.4	431 002	226	13
Jmeter-plugins	1.4.0	27 170	74	19

为了暴露同步瓶颈,针对不同应用程序选择了不同的工作负载。对于 HSQLDB 应用程序,通过增加客户数增加工作负载,分别为 10、20、30、

40;对于 Jenkins 应用程序,通过增加转移作业的数量增加工作负载,分别为 50、100 和 150;对于 SPECjbb2005 应用程序,通过增加线程数增加工作负载,分别为 1、2、3、4、5、6、7、8;对于 JGroups 应用程序,通过增加邮件数量增加工作负载,分别为 1×10^6、2×10^6、3×10^6;对于 Xalan 应用程序,通过增加 XML 文档的数量增加工作负载,分别为 1 000、1 500、2 000;对于 RxJava 应用程序,通过增加测试基准测试的数量增加工作负载,分别为 12、24、36、47;对于 Antlr 应用程序,通过增加测试使用旧的和新的 unicode 流机制对某些 unicode 字形进行 lex 的速度的数量增加工作负载,分别为 400、600、800;对于 Fop 应用程序,通过增加复制测试文件的数量增加工作负载,分别为 100、200、300;对于 Kafka 应用程序,通过增加打印转换文档的数量增加工作负载,分别为 20、40、60;对于 MINA 应用程序,通过向服务器发送消息的数量增加程序的工作负载,分别为 1 000、2 000、4 000;对于 Cassandra 应用程序,通过增加忙碌的工作线程数量增加程序的工作负载,分别为 1、2、3、4、5、6、7、8;对于 Jmeter-plugins 应用程序,通过增加循环的次数增加程序的工作负载,分别为 20、40、60。

3.4 实验结果及分析

3.4.1 RQ1 的评估

为了回答 RQ1,对选取的 12 个应用程序中同步锁数量、同步瓶颈数量以及同步瓶颈所占同步锁的百分比进行了统计,如表 2 所示。从实验数据中可以看出,在 12 个应用程序中共检测到了 72 个同步瓶颈,有 50 个同步瓶颈主要分布在 HSQLDB、SPECjbb2005 和 JGroups 中,SPECjbb2005 中存在的同步瓶颈数量最多,同步瓶颈百分比为 13.2%;在 MINA 中只发现了 1 个同步瓶颈,但由于该程序中的同步锁数量较少,同步瓶颈所占的百分比高达 8.3%,其次是在 SPECjbb2005 和 Antlr 中;在 RxJava 和 Fop 中同步锁数量比较少,因此共发现了 3 个同步瓶颈;虽然 Kafka、Jenkins 和 Cassandra 中的同步锁数量较多,但是在 Kafka 中只存在 6 个同步瓶颈,Cassandra 中只存在 2 个同步瓶颈,Jenkins 中未发现同步瓶颈。

3.4.2 RQ2 的评估

为了回答 RQ2,对满足 5 个优化模式的同步瓶颈进行了统计,如表 3 所示。第 3 ~ 7 列分别为 5 种优化模式对应的同步瓶颈数量。从应用程序角度可以看出,只有 HSQLDB、SPECjbb2005 和 Xalan 中存在锁消除的情况;HSQLDB 和 SPECjbb2005 中同步瓶颈数量较多,这两个应用程序中的同步瓶颈涵盖了 5 种优化模式;JGroups 中同步瓶颈数量相对较多,但没有建议进行锁消除和锁分解的同步瓶颈;在 Xalan 和 Kafka 中均包含 6 个同步瓶颈,Xalan 中没有建议进行锁分解的同步瓶颈,Kafka 中没有建议进行锁消除的同步瓶颈;Fop 和 MINA 中都只有 1 个同步瓶颈且均建议使用读锁;Antlr 中的两个同步瓶颈均建议进行锁降级。从优化模式角度可以看出,检测到的

表 2 同步瓶颈情况

Tab. 2 Synchronization bottleneck situation

应用程序	同步锁	同步瓶颈	同步瓶颈百分比/%
HSQLDB	684	14	2.0
Jenkins	274	0	0
JGroups	179	11	6.1
SPECjbb2005	190	25	13.2
Xalan	82	6	7.3
Antlr	16	2	12.5
RxJava	28	2	7.1
Fop	32	1	3.1
Kafka	1 722	6	0.3
MINA	12	1	8.3
Cassandra	239	2	0.8
Jmeter-plugins	93	2	2.2

表 3 同步瓶颈在优化模式中的分布情况

Tab. 3 Distribution of synchronization bottlenecks in optimized mode

应用程序	同步瓶颈	锁消除	读锁	写锁	锁分解	锁降级
HSQLDB	14	1	4	6	1	2
JGroups	11	0	4	3	0	4
SPECjbb2005	25	2	6	15	1	1
Xalan	6	2	2	1	0	1
Antlr	2	0	0	0	0	2
RxJava	2	0	1	1	0	0
Fop	1	0	1	0	0	0
Kafka	6	0	2	2	1	1
MINA	2	0	1	0	0	0
Cassandra	1	0	1	0	1	0
Jmeter-plugins	2	0	2	0	0	0

72 个同步瓶颈中建议进行锁消除的只有 5 个;有 24 个建议使用读锁,有 4 个建议进行锁分解,有 11 个建议进行锁降级,这 3 种优化在一定程度上都可以增加程序的并行度;建议使用写锁的有 28 个,其中来源于 SPECjbb2005 的有 15 个,在其他程序中建议使用写锁的情况较少。

将 IdeSync 和目前已有的重构工具 ReLocker[39] 进行了对比。ReLocker 是一个可以在不同锁机制之间进行转换的自动重构工具,ReLocker 工具在 2010 年被提出,只支持 JDK1.6 版本。计划将表 1 中所有程序进行两个工具对比,然而由于 ReLocker 版本问题,只找到了 HSQLDB 和 Cassandra 两个程序的适用版本,分别为 1.8.0.10 和 0.4.0,由于版本差异,这两个程序中锁的数量跟表 1 中程序版本的数量略有不同,HSQLDB 和 Cassandra 分别含有 266 个和 57 个同步锁。

表 4 给出了 ReLocker 和 IdeSync 重构结果的对比。在 ReLocker 中,对 HSQLDB 测试程序分别推荐了 31 个读锁和 212 个写锁,然而存在 23 个不能重构的问题;这种不能重构的问题在 Cassandra 中仍然存在,但数量仅有 3 个。相比之下,IdeSync 不仅可以推断读锁和写锁,而且不存在不能重构的问题,它可以实现更为细粒度的锁重构,不仅推荐了一定数量的锁分解和锁降级,而且还推断锁消除的情况,这表明 IdeSync 可以更好地对锁的使用进行优化。从总体来看,IdeSync 性能要优于 ReLocker。

表 4　ReLocker 和 IdeSync 对比

Tab.4　Comparison between ReLocker and IdeSync

测试程序	ReLocker			IdeSync				
	读锁	写锁	不能重构	锁消除	读锁	写锁	锁分解	锁降级
HSQLDB	31	212	23	1	43	193	23	6
Cassandra	4	50	3	2	4	47	1	3

3.4.3　RQ3 的评估

为了回答 RQ3,对不恰当地使用同步锁而限制了程序并行执行的同步瓶颈进行优化,并选取了 HSQLDB、JGroups、SPECjbb2005 和 Antlr 应用程序,对同步瓶颈优化前后的程序性能进行对比。

在 HSQLDB 程序中,同步瓶颈所在的方法分别为 writeCommitStatement( )、initPool( )、getStore( )、prepareStatement( )、connect( )、executeQuery( )、setInt( )、createStatement( )、forceSync( )、writeDeleteStatement( )、writeRowOutToFile( )、newSession( )、getBundleHandle( ) 和 nextTask( )。选取测试程序 JDBCBench,测试优化前后事务数分别为 1×10^5、2×10^5、3×10^5、4×10^5 时,程序每秒处理事务的情况,结果如图 5 所示。从结果可以看出,随着处理事务数的增加,程序并行执行效果越明显,事务率均有不同程度的提高,分别提高了 1.8%、4.8%、6.2% 和 8.4%。

在 JGroups 程序中,同步瓶颈所在的方法分别为 convertViews( )、getThreadName( )、startInfoSender( )、startViewConsistencyChecker( )、start( )、clearViews( )、addInfo( )、installView( )、adjustSuspectedMembers( )、getNewThreadName( ) 和 addChannelListener( )。选取测试程序 RoundTripRpc 和 UnicastTestRpc,测试优化前后两

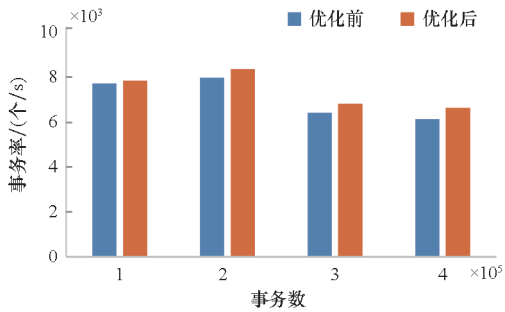


图 5　HSQLDB 的性能对比

Fig.5　Performance comparison in HSQLDB project

个群组间消息的发送与接收能力,结果如图 6 所示。从结果中可以看出,优化后两个程序的请求处理率分别提高了 4.8% 和 4.5%。

在 SPECjbb2005 中,同步瓶颈所在的方法分别为 newOrderIter( )、getOrderPtr( )、removeNewOrder( )、addOrder( ) 和 process( ) 等。程序优化前后的运行结果如图 7 所示。从结果中可以看出,在线程数为 1、2 和 3 时,优化前的堆内存使用比低于优化后的堆内存使用比,随着线程数的增加,当线程数为 4、5、6、7 和 8 时,优化后的堆内存使用比均低于优化前的堆内存使用比,分别减少了 12.7%、36.9%、23.5%、1.1% 和 6.8%。

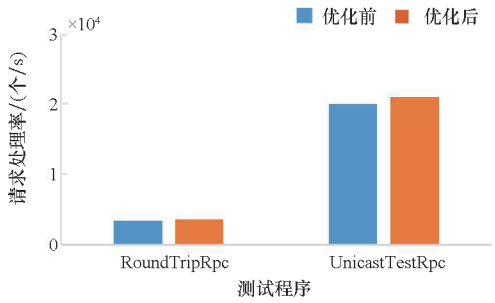


图 6 JGroups 的性能对比

Fig. 6 Performance comparison in JGroups project

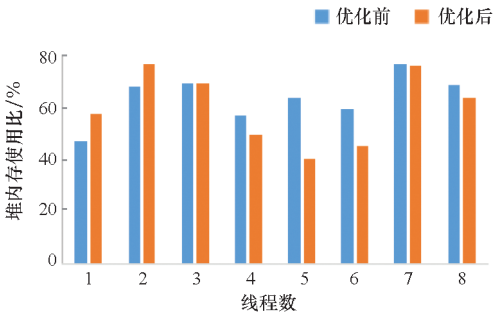


图 7 SPECjbb2005 的性能对比

Fig. 7 Performance comparison in SPECjbb2005 project

在 Antlr 中,同步瓶颈所在的方法分别为 addDFAState( )和 addDFAEdge( )。选取了测试程序 TimeLexerSpeed,测试了优化前后使用旧的 unicode 机制对元素进行测试,结果如图 8 所示。从结果中可以看出,元素数为 400 时优化后花费的时间高于优化前花费的时间,但随着元素数量的增加,当元素数为 600 和 800 时,优化后花费的时间比优化前花费的时间均有所下降,分别缩短了 2.1% 和 2.8% 。

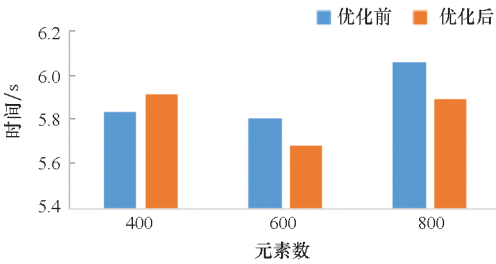


图 8 Antlr 的性能对比

Fig. 8 Performance comparison in Antlr project

通过对 RQ3 的评估,发现在 SPECjbb2005 程序中线程数为 1、2 和 3 时,以及在 Antlr 程序中元素数为 400 时,优化后程序性能未得到提升,这可能是由于优化后读写锁的获取和释放操作带来的开销所导致的,当程序线程任务或负载较小时,优化操作所需的开销高于优化效果。

3.5 有效性威胁

本节对实验过程中威胁有效性的几个因素进行了讨论:

1)由于并发程序执行的不确定性,性能测试的实验数据可能在一定范围内上下浮动。为了确保实验数据的有效性,所有实验结果都是在 10 次运行的基础上取平均值得到的。

2)本实验只选取了 12 个应用程序进行了评估,它们并不能代表所有程序。为了缓解这个有效性威胁,尽量选用涉及数据库、服务器等多个领域的程序。

4 结论

本文提出了检测和优化同步瓶颈的方法 IdeSync。该方法首先使用静态分析方法检测程序源码中的同步方法和同步块,采用动静结合分析技术分析同步依赖关系,构建同步依赖图,增加了静态分析同步依赖关系的准确性,降低了动态分析同步依赖关系的难度。然后增加程序工作负载,监测临界区的变化检测同步瓶颈,最后针对检测到的同步瓶颈,给出优化建议,并在 Eclipse JDT 框架下以插件的形式实现自动优化重构工具。在实验中选取 HSQLDB、SPECjbb2005 和 RxJava 等 12 个大型实际应用程序验证了该方法的有效性。在接下来工作中,将使用更多的应用程序和更大规模的测试集对 IdeSync 方法进行验证。

参考文献(References)

[1] SCHÖRGENHUMER A, HOFER P, GNEDT D, et al. Efficient sampling-based lock contention profiling for Java[C]// Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering, 2017: 331 – 334.

[2] PATROS P, AUBANEL E, BREMNER D, et al. A Java util concurrent park contention tool[C]//Proceedings of the 6th International Workshop on Programming Models and Applications for Multicores and Manycores, 2015: 106 – 111.

[3] ZHENG L, LIAO X F, HE B S, et al. On performance debugging of unnecessary lock contentions on multicore processors: a replay-based approach [C]//Proceedings of IEEE/ACM International Symposium on Code Generation and Optimization (CGO), 2015: 56 – 67.

[4] XIAN F, SRISA-AN W, JIANG H. Contention-aware scheduler: unlocking execution parallelism in multithreaded Java programs[C]//Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented Programming Systems Languages and Applications, 2008: 163 – 179.

[5] RINARD M C, DINIZ P C. Eliminating synchronization bottlenecks using adaptive replication[J]. ACM Transactions on Programming Languages and Systems, 2003, 25(3): 316 – 359.

[6] JOAO J A, SULEMAN M A, MUTLU O, et al. Bottleneck identification and scheduling in multithreaded

applications[J]. ACM SIGARCH Computer Architecture News, 2012, 40(1): 223 – 234.
[7] TALLENT N R, MELLOR-CRUMMEY J M, PORTERFIELD A. Analyzing lock contention in multithreaded applications[C]// Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 2010: 269 – 279.
[8] DAVID F, THOMAS G, LAWALL J, et al. Continuously measuring critical section pressure with the free-lunch profiler[J]. ACM SIGPLAN Notices, 2014, 49(10): 291 – 307.
[9] TRAHAY F. Contribution to automatic performance analysis of parallel applications[D]. Paris: Institut Polytechnique de Paris, 2021.
[10] SELAKOVIC M, PRADEL M. Performance issues and optimizations in JavaScript: an empirical study [C]// Proceedings of IEEE/ACM 38th International Conference on Software Engineering (ICSE), 2016: 61 – 72.
[11] KROENING D, POETZL D, SCHRAMMEL P, et al. Sound static deadlock analysis for C/Pthreads[C]//Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE), 2016: 379 – 390.
[12] ZHANG W H, JI X F, SONG B, et al. VarCatcher: a framework for tackling performance variability of parallel workloads on multi-core[J]. IEEE Transactions on Parallel and Distributed Systems, 2017, 28(4): 1215 – 1228.
[13] YAN C, CHEUNG A. Leveraging lock contention to improve OLTP application performance[J]. Proceedings of the VLDB Endowment, 2016, 9(5): 444 – 455.
[14] CICIRELLI F, GIORDANO A, MASTROIANNI C. Analysis of global and local synchronization in parallel computing[J]. IEEE Transactions on Parallel and Distributed Systems, 2021, 32(5): 988 – 1000.
[15] WEI X D, SHI J X, CHEN Y Z, et al. Fast in-memory transaction processing using RDMA and HTM [C]// Proceedings of the 25th Symposium on Operating Systems Principles, 2015: 87 – 104.
[16] YOON D Y, CHOWDHURY M, MOZAFARI B. Distributed lock management with RDMA: decentralization without starvation [C]//Proceedings of the 2018 International Conference on Management of Data, 2018: 1571 – 1586.
[17] YU Z L, ZHANG Y W, BRAVERMAN V, et al. NetLock: fast, centralized lock management using programmable switches[C]// Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication, 2020: 126 – 138.
[18] ALTMAN E, ARNOLD M, FINK S, et al. Performance analysis of idle programs [C]//Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications, 2010: 739 – 753.
[19] NAIR R, FIELD T. GAPP: a fast profiler for detecting serialization bottlenecks in parallel linux applications[C]// Proceedings of the ACM/SPEC International Conference on Performance Engineering, 2020: 257 – 264.
[20] DINIZ P C, RINARD M C. Lock coarsening: eliminating lock overhead in automatically parallelized object-based programs[J]. Journal of Parallel and Distributed Computing, 1998, 49(2): 218 – 244.
[21] ZHANG Y, SHAO S, LIU H, et al. Refactoring Java programs for customizable locks based on bytecode transformation[J]. IEEE Access, 2019, 7: 66292 – 66303.
[22] CURTSINGER C, BERGER E D. Coz: finding code that counts with causal profiling[C]//Proceedings of the 25th Symposium on Operating Systems Principles, 2015: 184 – 197.
[23] YU T T, PRADEL M. Pinpointing and repairing performance bottlenecks in concurrent programs[J]. Empirical Software Engineering, 2018, 23(5): 3034 – 3071.
[24] ARBEL M, GOLAN-GUETA G, HILLEL E, et al. Towards automatic lock removal for scalable synchronization[C]// Proceedings of International Symposium on Distributed Computing, 2015: 170 – 184.
[25] Apache. Xalan [CP/OL]. (2014 – 04 – 11) [2021 – 08 – 10]. http://xalan.apache.org/xalan-j/.
[26] Anon. WALA [CP/OL]. (2019 – 8 – 28) [2021 – 08 – 10]. http://wala.sourceforge.net/wiki/index.php/Main_Page.
[27] Anon. JProfiler [CP/OL]. (2021 – 6 – 24) [2021 – 08 – 10]. http://www.ej-technologies.com/.
[28] Anon. HSQLDB [CP/OL]. (2021 – 10 – 21) [2021 – 11 – 08]. http://hsqldb.org/.
[29] Anon. Jenkins [CP/OL]. (2019 – 01 – 04) [2021 – 11 – 08]. https://jenkins.io/zh/.
[30] Anon. JGroups [CP/OL]. (2020 – 06 – 09) [2021 – 11 – 08]. http://www.jgroups.org/.
[31] Anon. SPECjbb2005 [CP/OL]. (2013 – 10 – 21) [2021 – 11 – 08]. https://www.spec.org/jbb2005/.
[32] Anon. ReactiveX [CP/OL]. (2019 – 01 – 24) [2021 – 11 – 08]. http://reactivex.io/.
[33] Anon. ANTLR [CP/OL]. (2021 – 03 – 11) [2021 – 11 – 08]. https://www.antlr.org/.
[34] Apache. Fop [CP/OL]. (2018 – 05 – 24) [2021 – 11 – 08]. https://xmlgraphics.apache.org/fop/.
[35] Apache. Kafka [CP/OL]. (2020 – 08 – 03) [2021 – 11 – 08]. http://kafka.apache.org/.
[36] Apache. MINA [CP/OL]. (2020 – 07 – 03) [2021 – 11 – 08]. http://mina.apache.org/.
[37] Apache. Cassandra [CP/OL]. (2019 – 02 – 11) [2021 – 11 – 08]. https://cassandra.apache.org/.
[38] Apache. Jmeter-plugins [CP/OL]. (2016 – 04 – 05)[2021 – 11 – 08]. http://jmeter-plugins.org/.
[39] SCHAFER M, SRIDHARAN M, DOLBY J, et al. Refactoring Java programs for flexible locking [C]// Proceedings of the 33rd International Conference on Software Engineering (ICSE), 2011: 71 – 80.