软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2021,32(12):3710−3727 [doi: 10.13328/j.cnki.jos.006132]

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

基于下推自动机的细粒度锁自动重构方法*

张 杨, 邵 帅, 张冬雯

(河北科技大学 信息科学与工程学院,河北 石家庄 050018)

通讯作者: 张杨, E-mail: zhangyang@hebust.edu.cn

摘 要: 针对粗粒度锁会严重影响并发程序的可伸缩性问题,提出一种面向细粒度锁的自动重构方法.该方法借助访问者模式分析、别名分析、负面效应分析等多种程序分析技术获取临界区代码的读写模式,然后使用下推自动机构建不同锁模式的识别方法,根据识别结果进行代码重构.与以往锁重构方法的不同之处在于,该方法考虑了锁降级模式,使重构适用性更广.基于此方法,在 Eclipse JDT 框架下,以插件的形式实现了自动重构工具 FLock.在实验中,从重构个数、改变的代码行数、重构时间、准确性和重构后程序性能等方面对 FLock 进行了评估,并与已有的重构工具 Relocker 和 CLOCK 进行了对比.对 HSQLDB,Jenkins 和 Cassandra 等 11 个大型实际应用程序的重构结果表明: FLock 共重构了 1 757 个内置监视器对象,每个程序重构平均用时 17.5s.该重构工具可以有效地实现粗粒度锁到细粒度锁的转换,与手动重构相比,有效提升了细粒度锁的重构效率.

关键词: 细粒度锁;读写锁;重构;下推自动机;程序分析

中图法分类号: TP311

中文引用格式: 张杨,邵帅,张冬雯.基于下推自动机的细粒度锁自动重构方法.软件学报,2021,32(12):3710−3727. http://www.jos.org.cn/1000-9825/6132.htm

英文引用格式: Zhang Y, Shao S, Zhang DW. Automated refactoring approach for fine-grained lock based on pushdown automaton. Ruan Jian Xue Bao/Journal of Software, 2021,32(12):3710−3727 (in Chinese). http://www.jos.org.cn/1000-9825/6132.htm

Automated Refactoring Approach for Fine-grained Lock Based on Pushdown Automaton

ZHANG Yang, SHAO Shuai, ZHANG Dong-Wen

(School of Information Science and Engineering, Hebei University of Science and Technology, Shijiazhuang 050018, China)

Abstract: As coarse-grained locks have a negative impact on the scalability of concurrent programs, this study proposes an automatic refactoring approach to convert a coarse-grained lock into a fine-grained one. Several static analyses, such as visitor pattern analysis, alias analysis, and side-effect analysis are employed in this approach. The read and write pattern of a critical section is inferred by side effect analysis, and then a push down automaton is proposed to identify the read and write pattern. Finally, refactoring is conducted based on these results. An automatic tool FLock is implemented as the Eclipse plug-in. The proposed approach is evaluated by eleven open-source projects including HSQLDB, Jenkins, and Cassandra, by presenting results such as the number of refactored locks, changed lines of code, refactoring time, accuracy, program performance after refactoring. FLock is also compared with the existing tools Relocker and CLOCK. The experimental results show that a total of 1757 built-in monitors are refactored and each refactoring takes an average of 17.5 seconds. The experiments reveal that the proposed tool can help developers convert coarse-grained locks into fine-grained locks effectively.

Key words: fine-grained lock; read-write lock; refactoring; pushdown automaton; program analysis

* 基金项目: 国家自然科学基金(61440012); 河北省自然科学基金(18960106D); 河北省高等学校科学研究计划(ZD2019093); 河北省研究生创新资助项目(CXZZSS2020094)

Foundation item: National Natural Science Foundation of China (61440012); National Natural Science Foundation of Hebei Province (18960106D); Scientific Research Foundation of Hebei Educational Department (ZD2019093); Innovation Foundation Project of Hebei Province (CXZZSS2020094)

收稿时间: 2020-03-04; 修改时间: 2020-05-10, 2020-06-04; 采用时间: 2020-08-13

锁是并发程序中用于保护程序状态和数据访问正确性的必备措施,然而锁竞争问题是目前多核/众核时代影响并发程序性能的主要问题之一.锁竞争是指当临界区被一个互斥锁保护时,如果一个线程获得了该锁,那么其他请求访问该临界区的线程都将被阻塞,直到持有该锁的线程释放该锁.锁竞争问题的存在,会严重降低程序并行度,损害程序的可伸缩性,降低多核/众核处理器的执行效率.

不恰当的并发控制方式通常会进一步加剧锁竞争,程序开发人员有时习惯于使用粗粒度锁,例如在方法的修饰符中加入 synchronized 关键字,使整个方法处于锁的保护之中.这种加锁方式虽然可以降低程序开发的复杂性,然而由于粗粒度锁控制的临界区代码较长,导致其他想获取该锁的线程等待时间增加,往往会加剧锁竞争.许多开发人员尝试使用细粒度锁,相比于粗粒度锁,细粒度锁只对一小部分代码进行加锁,可以有效减少锁的持有时间和线程等待时间,减少锁竞争问题的影响.

与粗粒度锁比较,使用细粒度锁并非一件容易的事.要想实现细粒度的加锁方式,通常可以采用升级锁、降级锁、优化读锁等加锁方式,也可以采用将粗粒度读写锁分解为细粒度读写锁的方式.程序开发人员不仅需要对代码模式进行分析以确定使用何种细粒度锁的加锁方式,而且还需要在同一种方式的不同实现机制之间进行选择,例如在 JDK 中,读写锁和邮戳锁分别提供了降级锁,这两种锁提供的降级锁的实现方法截然不同,程序开发人员需要在两种锁机制之间进行选择,增加了细粒度锁的使用难度.在传统的方法中,为了使用细粒度锁,开发人员通常需要手工地对并发程序中使用粗粒度锁的代码进行重构.然而这种重构方式既费时费力,还可能会给代码引入新的错误,因此迫切需要对面向细粒度锁的重构方法进行研究.

目前,针对锁重构的研究有很多:Tao 等人[1]提出了针对 Java 程序根据类属性域划分锁保护域的自动锁分解重构方法;Yu 等人[2]在进行优化同步瓶颈的研究中提出了一种锁分解方式;Emmi 等人[3]提出了一种自动锁分配技术,推断获取锁的位置并确保加锁的正确性,避免发生死锁;Kawachiya 等人[4]提出一种锁保留算法,该算法允许锁被线程保留;Schafer 等人[5]针对重入锁及读写锁提出了一种自动重构算法,并实现了重构工具 Relocker; Zhang 等人提出了面向可定制锁[6]和邮戳锁[7]的自动重构方法;Arbel 等人[8]提出了并发数据结构的代码转换,他们的转换采用基于锁的代码,并用乐观同步替换其中的一些加锁代码以减少争用;Bavarsad[9]针对软件事务性内存,提出了一种读写锁分配技术来克服全局时钟的开销.在工业界,集成在 IntelliJ IDEA 上的重构插件 LockSmith[10]和基于 Eclipse JDT 的并发重构插件[11]都可以实现锁重构.从目前国内外的研究现状来看,许多学者已经认识到锁竞争问题以及并发控制方式在程序设计中的重要性,并对锁粒度问题以及锁机制相关的问题进行了研究.但大部分研究是对锁进行消除和对同步锁进行分解,对细粒度锁的重构方法还有待深入研究.

要想实现面向细粒度锁的自动重构,需要解决以下几个问题:(1) 代码分析时需要对临界区中的每一条语句进行读写操作分析,而不能像 Relocker[5]工具那样将整个临界区代码作为一个整体进行分析;(2) 在获取读写模式分析后,需要研究如何对读写操作模式进行识别,进而推断出使用哪一种细粒度锁;(3) 需要研究如何构建由粗粒度锁到细粒度锁的重构代码.

为了解决上述问题,本文提出基于下推自动机的细粒度锁自动重构方法,通过访问者模式分析、别名分析、锁集分析、负面效应分析等程序分析方法获取临界区代码的读写模式,然后使用下推自动机构建不同锁模式的识别方法,根据识别结果进行代码重构.基于此方法,在 Eclipse JDT 框架下,以插件的形式实现了一个自动重构工具 FLock.在实验中,从重构个数、改变的代码行数、重构时间、准确性和重构后程序性能等方面对 FLock 进行了评估,并与已有重构工具 Relocker[5]和 CLOCK[7]进行了对比,对 HSQLDB,Jenkins 和 Cassandra 等 11 个大型实际应用程序的重构结果表明:FLock 共重构了 1 757 个内置监视器对象,每个程序重构平均用时 17.5s.该重构工具可以有效实现粗粒度锁到细粒度锁的转换,与手动重构相比,有效提升了细粒度锁的重构效率.

本文的主要贡献有 3 个方面.

1) 提出了一种面向细粒度锁的自动重构方法;
2) 以 Eclipse 插件的形式实现了自动重构工具 FLock,可以实现源码级别的重构,帮助开发者完成从粗粒度锁到细粒度读写锁的自动转换;
3) 使用 11 个大型实际应用程序对 FLock 进行了评估,并与现有工具 Relocker 和 CLOCK 进行了对比.

本文第 1 节介绍本文的研究动机.第 2 节介绍面向细粒度锁的自动重构方法.第 3 节展示自动重构工具 FLock 的使用界面.在第 4 节给出对本文所提出的方法和工具的实验评估.第 5 节对相关工作进行介绍.第 6 节是本文的总结.

1 研究动机

读写锁是 JDK 1.5 版本中引入的一种锁机制,它包含一对相互关联的锁:读锁和写锁.写锁是一个排它锁,只能由一个线程持有;读锁是一个共享锁,在没有线程持有写锁的情况下,读锁可以由多个线程同时持有,读锁允许在访问共享数据时有更大的并发性.Pinto 等人[12]通过研究 SourceForge 上的 2 227 个含有并发结构的 Java 工程发现:Java 并发包还没有得到良好的应用,只有大约 23%有并发编程结构的 Java 工程在使用.

为了说明细粒度锁重构的必要性,我们在代码结构上进行了说明.图 1 展示了 *processCached*(·)方法的 3 种实现方式,该程序段选自读写锁的 Java API 文档[13],是一种典型的缓存处理操作.方法 *processCached*(·)模拟了对数据库及缓存的操作,首先判断数据是否存在于缓存中:如果存在,则直接从缓存中读取数据;如果不存在,则从数据库中把数据写入缓存.在图 1(a)中,该方法使用 synchronized 进行同步控制,整个方法都处于该锁的保护下,是一种粗粒度的保护.图 1(b)为使用 Relocker[5]进行重构后的代码,由于该方法中包含对缓存的写入操作,按照 Relocker 的锁推断策略,将使用写锁对其进行重构.然而我们发现:写操作的执行仅仅发生在 if 语句的条件成立时,如果条件不成立,写锁根本不会执行,只需要执行读操作.如果把全部代码使用写锁进行保护,可能会降低程序的性能.如果使用读锁,将允许多个线程同时读,可以提高程序的并发性.图 1(c)为改进后的代码,是一种细粒度的加锁方式.该方式首先获取读锁并判断 cacheValid(第 3 行、第 4 行):如果 if 条件不成立,则直接读取并释放读锁(第 16 行~第 20 行);如果成立,则释放读锁获得写锁(第 5 行、第 6 行).为了保证程序状态的一致性,这里需要重新对状态进行判断(第 8 行),因为其他线程可能已经对缓存进行了修改.当从数据库中写入缓存之后,获得读锁再释放写锁,完成锁降级操作(第 9 行~第 14 行).从图 1(c)可以看到:该方式将一直使用读锁,直到写入的时候再加写锁.

```
1 public Synchronized void processCached(·){
2   if (!cacheValid){
3     ... //write to cache
4     }
5     ... //read data
6 }
```

(a) Method *processCached*(·) based on built-in monitor

```
1 ReentrantReadWriteLock rwl=new ReentrantReadWriteLock(·);
2 public void processCached(·){
3 rwl.WriteLock(·).Lock(·);
4   try {
5     if (!cacheValid){
6       ... //write to cache
7     }
8     ... //read rata
9   } finally {
10     rwl.writeLock(·).unLock(·);
11   }
12 }
```

(b) Method *processCached*(·) based on *ReentrantReadWriteLock*

```
1  ReentrantReadWriteLock rwl=new ReentrantReadWriteLock(·);
2  public void processCached(·){
3  rwl.readLock(·).Lock(·);
4    if (!cacheValid){
5      rwl.readLock(·).unLock(·);
6      rwl.writeLock(·).Lock(·);
7      try {
8        if (!cacheValid){
9          ... //write to cache
10         rwl.readLock(·).Lock(·);
11       }
12     } finally {
13       rwl.writeLock(·).unLock(·);
14     }
15   }
16   try {
17     ... //read rata
18   } finally {
19     rwl.writeLock(·).unLock(·);
20   }
21 }
```

(c) Method *processCached*(·) based on lock downgrading

Fig.1 Three implementations of the method *processCached*(·)

图 1 *processCached*(·)方法的 3 种实现方式

从上面的例子可以看出:使用锁降级实现了对临界区的细粒度保护,加锁方式更为合理,并且可以在一定程度上减少了锁竞争.由于读锁是共享锁,允许多个线程同时访问,在不发生写入时可以增加程序的并发性.

2 面向细粒度读写锁的重构

本节首先给出了重构的整体框架,之后对程序分析方法、基于下推自动机的锁模式推断以及重构算法进行了介绍.

2.1 重构框架

在重构过程中,首先通过访问者模式对源码对应的抽象语法树进行遍历,主要用到的程序静态分析方法包括锁集分析、别名分析和负面效应分析:锁集分析用来对监视器对象进行收集,并存储监视器对象和锁对象的对应关系;别名分析用来解决锁集中的别名问题;负面效应分析用来分析临界区是否产生负面效应,并生成对应的临界区读写模式序列.下推自动机对临界区读写模式序列进行识别,进而进行锁推断,在读锁、写锁、锁降级、锁分解这 4 种加锁模式中选择相应的加锁模式并进行重构.在重构之后,需要对重构前后的一致性进行检测,以保证重构的正确性.面向细粒度锁的重构框架如图 2 所示.

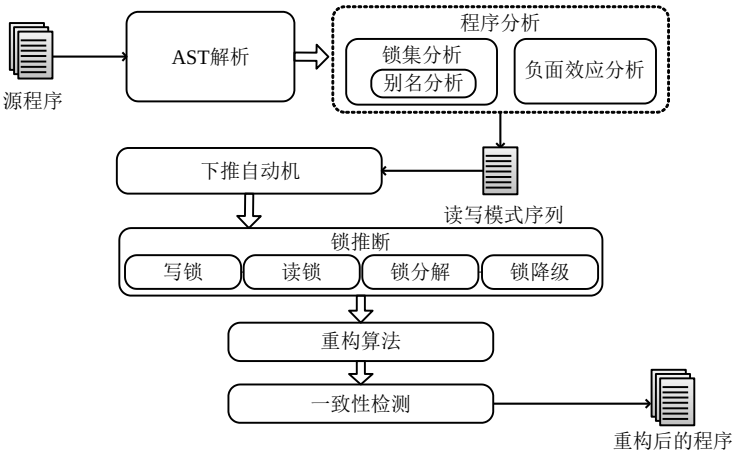


Fig.2 Refactoring framework

图 2 重构框架图

2.2 AST解析

在重构框架中,首先对源程序进行解析,生成一个抽象语法树(abstract syntax tree,简称 AST).在具体的解析过程中,首先通过 Eclipse JDT[14]获取用户在进行重构操作时所选择的对象,然后将用户选择的对象存放到 ICompilationUnit 对象中,最后将 ICompilationUnit 对象对应的元素解析成 AST.AST 节点的类型很多,每个节点表示一个源程序中的一个语法结构,包括类型、表达式、语句、声明等.使用访问者模式分析对 AST 上的同步方法节点以及同步块语句进行遍历,找到同步方法和同步块.在具体的实现过程中,通过继承 EclipseJDT 中提供的抽象类 ASTVisitor 实现了一个子类作为具体访问者,用于具体类型节点的遍历.

2.3 程序分析方法

2.3.1 锁集分析

在进行重构之前,首先通过访问者模式遍历程序中所有的方法,并对监视器对象进行收集.在重构过程中,还需要对监视器对象是否产生别名进行判断,进行别名分析.别名是指监视器对象名称不同,但是同时指向相同的内存地址.对于监视器对象不同的临界区,需要使用不同的锁对象进行重构;对监视器对象相同的临界区,则使用相同的锁对象进行重构.使用一个键-值对集合对监视器对象和读写锁锁对象的映射关系进行存储,监视器对象作为键-值对集合的键,监视器对象对应的读写锁对象为键-值对集合的值.

监视器集合定义为 $MonitorSet=\{m_1,m_2,\ldots,m_n\}$,其中,$m_i$ 为监视器对象,$i\in\{1,2,\ldots,n\}$.监视器 m_i 的指向集定义为 $PoniterSet_i$,如果对于任意的 m_i 和 $m_j,i\neq j$,存在 $PoniterSet_i\cap PoniterSet_j\neq\varnothing$,则认为 m_i 和 m_j 互为别名,并把 m_i 和

m_j作为别名对$\langle m_i,m_j\rangle$存储在可能别名集*AliasSet*中.在进行重构之前,首先通过别名集*AliasSet*构建锁集*LockSet*,对别名对中的监视器对象对应的锁对象进行实例化,并存入锁集*LockSet*中.别名对中两个监视器对象应对应相同的锁对象,例如别名对$\langle m_i,m_j\rangle$在 *LockSet* 中表示为两个键值对组合$\langle m_i{:}lock_k\rangle$,$\langle m_j{:}lock_k\rangle$,其中,$lock_k$ 为锁对象;而没有产生别名问题的监视器对象在重构过程中对锁对象进行实例化,并存入锁集 *LockSet* 中.

2.3.2 负面效应分析

负面效应是指对非本地环境的修改[15],例如一个操作、方法或表达式在执行过程中修改了内存单元,则程序将产生负面效应.由于本文提出的重构方法不仅要重构读锁和写锁,而且还要进行细粒度的锁重构,因此负面效应分析需要对整个临界区进行分析,并生成一个临界区读写模式序列来表示临界区的读写操作.在生成临界区对应的读写模式序列时,使用字符 *r* 表示临界区的一个读操作,使用字符 *w* 表示一个写操作,使用字符 *c* 表示一个 if 条件判断语句的开始且条件判断为读操作,使用字符 *e* 表示一个 if 判断语句的结束.

本文的负面效应分析通过 WALA[16]的中间表示 IR 对方法中的指令进行遍历分析,判断指令是否修改内存单元.在对方法调用指令进行分析时,为了保证重构工具的执行效率,将方法调用进入的层数限制为 5 层.具体分析算法如算法 1 所示.

1) 首先获得临界区所对应的指令集,调用 *sideEffectAnalysis* 方法对每一条指令进行遍历分析(第 1 行~第 4 行);
2) 在 *sideEffectAnalysis* 方法中,首先对方法调用进入的层数限制进行判断(第 6 行),如果大于限制层数 5 层,为了保证临界区安全,将其认定为一个写操作,返回字符 *w*(第 23 行);
3) 之后,对每条指令进行分析,如果指令修改了静态字段,或修改了实例字段,或修改了堆内存,则当前指令产生了负面效应,返回字符 *w*(第 7 行、第 8 行);
4) 如果指令是方法调用指令,首先对调用层数的计数器加 1(第 11 行),之后,递归调用 *sideEffectAnalysis* 方法对被调用方法里的指令进行分析:若被调用方法包含写指令,则当前方法调用指令产生了负面效应,返回字符 *w*(第 11 行~第 15 行);如果当前被调用方法没有产生负面效应,则返回字符 *r*(第 16 行);
5) 如果是 if 判断语句且判断语句为读操作,返回字符 *c*;如果指令是 if 条件判断结束指令,则返回字符 *e*(第 17 行~第 20 行);
6) 其他指令均返回字符 *r*(第 21 行).

算法 1. 负面效应分析算法.

输入:目标临界区 *c*;

输出:*c* 对应的临界区读写模式序列 *str*.

```
1.  通过目标临界区 c 对应的中间表示 IR 获临界区 c 的指令集 Ins;
2.  for 遍历指令集 Ins 中每一条指令 i do
3.    str.append(sideEffectAnalysis(i,0));
4.  end for
5.  char sideEeffectAnalysis (指令 i,int limit):
6.    if limit<5 then
7.      if 指令修改静态字段||指令修改实例字段||指令修改堆内存 then
8.        return w;
9.      else if 指令为方法调用指令 then
10.       limit++;
11.       for 被调用方法中的每一条指令 k do
12.         if sideEeffectAnalysis(k,limit)==w then
13.           return w;
14.         end if
```

```
15.          end for
16.          return r;
17.       else if 指令为 if 条件判断指令&&判断语句为读操作 then
18.          return c;
19.       else if 指令为 if 条件判断结束指令 then
20.          return e;
21.       else return r;
22.       end if
23.    else return w;
24.    end if
```

2.4 锁推断

在负面效应分析中,使用字符 c 表示一个 if 语句的开始,e 表示一个 if 语句的结束.由于判断语句每一个开始对应着一个结束,负面效应分析会产生类似 c^nAe^n(A 代表其他操作)的字符序列,其中,n>0.在匹配过程中,需要记录 c 和 e 的对应关系,因为 n 的值是不确定的,导致状态的个数也是不确定的,所以我们使用下推自动机对负面效应分析产生的临界区读写模式序列进行模式匹配,以确定程序重构后的加锁模式.

定义 1. 用于推断细粒度锁模式的下推自动机 $M_{fg}=(Q,\Sigma,\Gamma,\delta,q_0,Z_0,F)$是一个七元组模型,其中,

- $Q=\{q_0,q_1,\ldots,q_7\}$是一个有穷状态集;
- $\Sigma=\{r,w,c,e\}$是输入字母表,其中,r 代表一个读操作,w 代表一个写操作,c 代表一个 if 条件判断语句的开始且条件判断为读操作,e 为一个 if 条件判断语句的结束标志;
- $\Gamma=\{Z_0,C,R,W,D,A,B,V\}$是有限的堆栈字母表;
- $\delta=\delta(q,x,X)$是转移函数,一般使用规则$\langle q,x,X,q',T\rangle$代表转移函数,其中:$q$ 为状态符号;x 为输入符号;X 和 T 为栈符号,表示当下推自动机处于状态 q、当前输入字符为 x 且栈顶符号为 X 时,则下推自动机的状态变为 q',并用符号串 T 代替栈顶符号 X.符号串 T 有 3 种形式,分别为 X',$X'X$,ρ,其中,X'表示用字符 X'替换栈顶元素,$X'X$ 表示将 X'压入栈中,ρ表示弹出当前栈顶元素;
- q_0 为初始状态;
- Z_0 为初始栈顶符号;
- $F=\{q_1,q_3,q_4,\ldots,q_7\}$为终态集.

下推自动机 M_{fg} 的状态转换图如图 3 所示,其中,规则$\langle q,x,X,q',T\rangle$在图中表示为$\langle x,X/T\rangle$,状态 q 到状态 q'的转移用直线箭头表示.

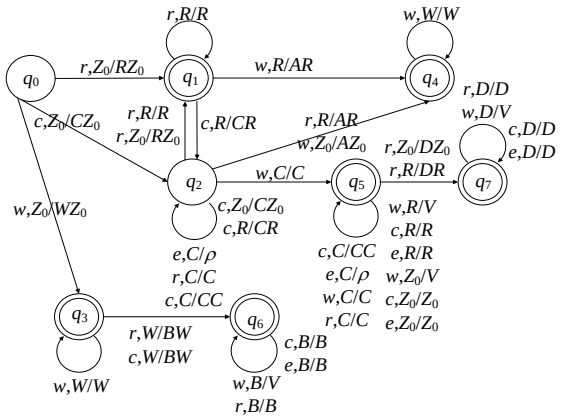


Fig.3 State transition diagram of pushdown automaton M_{fg}

图 3 下推自动机 M_{fg} 状态转换图

例如:图 3 中,当下推自动机处于状态 q_0、当前输入字符为 r 且栈顶符号为 Z_0 时,则下推自动机由状态 q_0 转移到 q_1,并把字符 R 压入栈中.

在锁模式定义之前,为了简化表示终态所识别的符号集,首先定义符号串集合 $S_{rc}=\{r,c\}^+$为字符 r 和 c 组成的正闭包,$S_{re}=\{r,e\}^+$为字符 r 和 e 组成的正闭包,$S_{wc}=\{r,w,c\}^+$为字符 r,w 和 c 组成的正闭包,$S_{we}=\{r,w,e\}^+$为字符 r,w 和 e 组成的正闭包,$S=\{S_{wc}S_{we}\}$,其中,字符 c 和字符 e 数量相等.

定义 2(写锁模式的定义). 一个临界区被推断为写锁,当且仅当读写模式序列被终态 q_3 所接受.

终态 q_3 所识别的序列为 $S_{w1}=\{w\}^+$,表示临界区只包含写操作,将使用写锁进行同步保护.

定义 3(读锁模式的定义). 一个临界区被推断为读锁,当且仅当读写模式序列被终态 q_1 所接受.

终态 q_1 所接受的序列集为 $S_r=\{S_{rc}S_{re}\}^+$,其中,字符 c 和字符 e 的数量相同.序列集 S_r 不包含字符 w,表示临界区没有产生负面效应,所以使用读锁.

定义 4(锁降级模式的定义). 一个临界区被推断为锁降级,当且仅当读写模式序列被终态 q_7 所接受.

终态 q_7 所接受的序列集为 $S_d=\{r^*cS^+er^+\}$,表示临界区首先包含读操作或没有,之后是一个 if 判断语句块,其中,判断语句为读操作,语句块内包含其他操作但至少有一个写操作,if 块之后包含至少一个读操作且只包含读操作.

定义 5(锁分解模式的定义). 一个临界区被推断为锁分解,当且仅当读写模式序列被终态 q_5,q_4,q_6 所接受.

终态 q_5 识别的序列集为 $S_{s1}=\{r^*cS^+e\}$,表示临界区首先包含读操作或没有,之后是一个 if 判断语句块,判断语句为读操作,语句块内包含其他操作但至少有一个写操作;终态 q_4 和 q_6 识别的序列表示读写操作分离的临界区,终态 q_4 所识别的序列集为 $S_{s2}=\{S_rS_{w1}\}$,代表临界区前半部分为读操作后半部分为写操作,终态 q_6 所识别的序列集为 $S_{s3}=\{S_{w1}S_r\}$,代表临界区前半部分为写操作后半部分为读操作.

定义 6(下推自动机停机的定义). 当输入读写模式序列为空,或栈顶符号为 V 时,下推自动机停止对读写模式序列的识别.

当读写模式序列为空时,表示临界区没有操作,则对临界区使用读锁;当栈顶符号为 V 时,所识别的序列集为 $S_{w2}=\{X_S(S_{w1}\cup S_r\cup S_d\cup S_{s1}\cup S_{s2}\cup S_{s3})\}$,表示读写模式序列不符合细粒度锁规则,临界区将使用写锁进行同步保护.

2.5 重构算法

本节给出了重构算法的设计,首先对相关代码建立 AST;之后遍历 AST 的所有方法节点,找到同步方法和包含同步块的方法节点;最后,根据负面效应分析得到的读写串对应的锁模式进行重构.重构算法如算法 2 所示.具体流程如下.

1) 首先获取当前的监视器对象(第 1 行),并判断锁集 *LockSet* 中是否存在监视器对象所对应的锁对象(第 2 行):如果存在,则从锁集 *LockSet* 中获得监视器对象对应的锁对象(第 3 行);否则生成一个新的锁对象,并把监视器对象与锁对象的对应关系存入锁集 *LockSet* 中(第 5 行、第 6 行);
2) 通过负面效应分析获得 c 对应的临界区读写模式序列 *str*(第 8 行);
3) 如果 c 为同步方法或同步块,则移除 synchronized 锁,并通过下推自动机识别临界区读写模式序列 *str*,根据识别结果进行重构(第 9 行~第 13 行);
4) 最后对重构前临界区 c 和重构后的 *c_r* 进行一致性检测(第 17 行):如果符合一致性检测规则,返回重构结果 *c_r*(第 15 行);否则,将使用写锁对临界区 c 进行重构(第 16 行~第 19 行),其中,Consistency 方法基于一致性检测规则进行定义(第 3 节给出).

算法 2. 重构算法.

输入:临界区 c;

输出:临界区 c 的重构结果.

1. 获取到临界区 c 对应的监视器对象 M;
2. **if** $m\in LockSet$ **then**
3. *lockfiled*←*LockSet*(m);

4. **else**
5. *lockfiled*←**new** *ReentrantReadWriteLock*(·);
6. *LockSet*←⟨*m*:*lockfiled*⟩;
7. **end if**
8. *str*←*getEffectAnalysis*(*c*);
9. **if** *c* 为同步方法||*c* 为同步块 **then**
10. 移除 synchronized 相关锁;
11. *Results*←*Pushdown*(*str*);
12. 根据识别结果 Results 将 *c* 重构为 *c_r*;
13. **end if**
14. **if** *Consistency*(*c*,*c_r*) **then**
15. **return** *c_r*;
16. **else**
17. 使用写锁将 *c* 重构为 *c_w*;
18. **return** *c_w*;
19. **end if**

3 一致性检测

FlexSync[17]是一个可以通过施加不同标记在不同同步机制之间进行重构转换的工具,FlexSync 中定义了一致性检查规则,以此来检查在不同同步机制之间转换的一致性.为了尽可能地保证重构的正确性,本节参照 FlexSync 的一致性检测规则,给出了 FLock 重构前后的一致性检测规则.在给出规则之前,我们首先对相关的内容进行了定义.

定义 7(临界区集合). 一个应用程序 P 中所有临界区的集合定义为 $C=\{c_1,c_2,\ldots,c_n\}$.对于$\forall c_i\in C(1\leqslant i\leqslant n)$,$c_i=\{c_{i1},c_{i2},\ldots,c_{ik}\}$表示 c_i 被划分为 k 个细粒度的临界区.

由于 P 中的代码是有限的,因此 C 是一个有限集合,对于$\forall c_i\in C(1\leqslant i\leqslant n)$,$c_i$ 表示某一个临界区.c_i 中可以包含若干读写操作 $op_{i1},op_{i2},\ldots,op_{ir}$,表示为$\{op_{i1},op_{i2},\ldots,op_{ir}\}\subseteq c_i$.

C_{before} 和 C_{after} 分别表示重构前和重构后的临界区集合,由于 FLock 在重构时需要分割临界区,因此$|C_{before}|<|C_{after}|$,其中,$|C|$表示集合 C 元素个数.

定义 8(重构前锁集合). 一个应用程序 P 中所有用于保护临界区的锁集合定义为集合 $S=\{s_1,s_2,\ldots,s_m\}$.

由于 C 是有限集合,所以 S 也是一个有限集合.由于一个锁可以保护多个临界区,故 $m\leqslant n$.对于$\forall s_e\in S$,s_e 表示既包含加锁操作又包含解锁操作.

定义 9(锁保护). 对于$\forall c_i\in C(1\leqslant i\leqslant n)$,$\exists s_e\in S(1\leqslant e\leqslant m)$,如果临界区 c_i 处于锁 s_e 的保护中,则定义保护关系为 $s_e\odot\{c_i\}$.进一步,一个锁可以保护多个临界区,定义为 $s_e\odot C_k$,其中,C_k 是 C_{before} 的子集,即 $C_k\subseteq C_{before}$.

定义 10(重构后锁集合). 一个应用程序 P 中重构之后所有的锁集合定义为集合 $L=\{l_1,l_2,\ldots,l_t\}$.对于$\forall l_a\in L(1\leqslant a\leqslant t)$,$l_a\odot C_v$,$C_v$ 是 C_{after} 的子集,即 $C_v\subseteq C_{after}$.

定义 11(锁状态原子性保持). 对于$\forall l_a\in L$,如果 l_a 在没有释放锁的情况下完成锁状态切换,则说明锁状态原子性没有被破坏,定义其为$\triangleright l_a$,否则定义为$\neg l_a$.

定义 11 说明了锁状态的原子性问题,举例来说:读写锁在由写锁切换为读锁时,可以不用释放该锁,在该锁的内部完成锁状态的切换.

定义 12(重构前后锁的对应关系). 重构前,对于$\forall c_k\in C(1\leqslant k\leqslant n)$,$\exists s_e\in S(1\leqslant e\leqslant m)$,$s_e\odot C_k$;重构后,对于$\forall c_v\in C(1\leqslant v\leqslant n)$,$\exists l_a\in L(1\leqslant a\leqslant t)$,$l_a\odot C_v$.如果 $C_v\subseteq C_k$,则 s_e 和 l_a 之间存在一一对应关系,记为 $s_e\equiv l_a$.

定义 13(Happens-before 关系). 对于$\forall c_i\in C,\{op_{i1},op_{i2},\ldots,op_{ir}\}\subseteq c_i,\exists u,v(1\leqslant u\leqslant r,1\leqslant v\leqslant r)$,如果 op_{iu} 先发生于

op_{iv},则定义 Happens-before 关系为 $op_{iu}\geqslant op_{iv}$.依据 Happens-before 关系的传递性,如果 $op_{iu}\geqslant op_{iv}$ 并且 $op_{iv}\geqslant op_{iw}$,则 $op_{iu}\geqslant op_{iw}$.

Happens-before 关系的定义是建立在 Java 内存模型基础上的读写操作发生序关系,是 Java 语言内存一致性的重要准则.该关系建立的方式之一是通过程序中的同步关系,解锁操作之前的操作先发生于解锁之后获取该锁的操作.

基于如上的定义,下面给出了重构的一致性检测规则.

规则 1. 对于重构前,$\forall c_i\in C(1\leqslant i\leqslant n),\exists s_e\in S(1\leqslant e\leqslant m)$,使得 $s_e\odot\{c_i\}$;重构后,$c_i=\{c_{i1},c_{i2},\ldots,c_{ik}\}$,对于 $\forall c_{ij}(1\leqslant i\leqslant n,1\leqslant j\leqslant k),\exists l_a\in L(1\leqslant a\leqslant t)$,使得 $l_a\odot\{c_{ij}\}$.

规则 1 说明了重构之前处于锁保护的临界区,在重构之后仍处于锁的保护中.

规则 2. 对于重构前,$\forall c_i\in C(1\leqslant i\leqslant n),\exists s_e\in S(1\leqslant e\leqslant m)$,使得 $s_e\odot\{c_i\}$;重构后,$c_i=\{c_{i1},c_{i2},\ldots,c_{ik}\}$,对于 $\forall c_{ij}(1\leqslant i\leqslant n,1\leqslant j\leqslant k),\exists l_a\in L(1\leqslant a\leqslant t)$,如果有 $l_a\odot\{c_{ij}\}$,则 $s_e\equiv l_a$.

规则 2 说明:如果重构前临界区 c_i 在 s_e 的保护中,即使重构后 c_i 被分割为多个临界区,保护这些临界区的锁 l_a 和 s_e 之间存在一一对应的关系.如果所有的锁都可以被重构,那么重构前后的锁集合应满足 $|S|=|L|$.需要说明的是:对于 FLock 重构前后锁的一一对应关系,通常表现在同步锁的监视器对象和读写锁对象的一一对应关系上.

规则 1 和规则 2 从重构前后代码结构上进行了一致性说明,保证了代码结构的完整性.

由于 FLock 在进行面向细粒度锁重构时主要采用了降级锁和锁分解两种方式,下面主要针对这两种方式进行规则定义.

规则 3. 重构前,对于 $\forall c_i\in C(1\leqslant i\leqslant n),\exists s_e\in S(1\leqslant e\leqslant m)$,使得 $s_e\odot\{c_i\}$;重构后,$c_i=\{c_{i1},c_{i2},\ldots,c_{ik}\}$,如果 $\exists l_a\in L(1\leqslant a\leqslant t)$,使得 $s_e\equiv l_a$,对于 $\forall c_{ij}(1\leqslant i\leqslant n,1\leqslant j\leqslant k)$,有 $(l_a\odot\{c_{ij}\})\wedge(\sim l_a)$成立,则锁降级重构前后可以保证一致性.

规则 3 主要针对降级锁的重构进行了约束.虽然锁降级过程中会由写锁降级为读锁,但该过程是在锁的内部进行转换的,期间并没有释放锁,可以保证锁的原子性,所以锁降级重构前后是一致的.

规则 4. 重构前,对于 $\forall c_i\in C(1\leqslant i\leqslant n),\exists s_e\in S(1\leqslant e\leqslant m)$,使得 $s_e\odot\{c_i\}$;重构后,$c_i=\{c_{i1},c_{i2},\ldots,c_{ik}\}$,如果 $\exists l_a\in L(1\leqslant a\leqslant t)$,使得 $s_e\equiv l_a$,且对于 $\forall c_{ij}(1\leqslant i\leqslant n,1\leqslant j\leqslant k)$,有 $(l_a\odot\{c_{ij}\})\wedge(\sim l_a)$.当 $\{op_{i1},op_{i2},\ldots,op_{ir}\}\subseteq c_i$ 时,对于 $\forall op_{ip},op_{iq}(1\leqslant p\leqslant r,1\leqslant q\leqslant r)$,如果在 C_{before} 中 $op_{ip}\geqslant op_{iq}$,则在 C_{after} 中仍有 $op_{ip}\geqslant op_{iq}$.

规则 5. 重构前,对于 $\forall c_i\in C(1\leqslant i\leqslant n),\{op_{i1},op_{i2},\ldots,op_{ir}\}\subseteq c_i,\exists s_e\in S(1\leqslant e\leqslant m)$,使得 $s_e\odot\{c_i\}$;重构后,$c_i=\{c_{i1},c_{i2},\ldots,c_{ik}\}$,如果 $\exists l_a\in L(1\leqslant a\leqslant t)$,有 $s_e\equiv l_a$,且对于 $\forall c_{ix},c_{iy}(1\leqslant i\leqslant n,1\leqslant x\leqslant k,1\leqslant y\leqslant k,x\neq y)$,有 $(l_a\odot\{c_{ix},c_{iy}\})\wedge(\sim l_a)$.对于 $\forall op_{ip},op_{iq}(1\leqslant p\leqslant r,1\leqslant q\leqslant r)$,如果 $\{op_{ip}\}\subseteq c_{ix},\{op_{iq}\}\subseteq c_{iy},op_{ip}$ 和 op_{iq} 访问同一内存位置并且 op_{ip} 或 op_{iq} 是写操作,则不能进行锁分解.

规则 4 和规则 5 共同保证了锁分解重构的一致性.规则 4 对锁分解的重构进行了约束,该规则通过检查临界区读写语句的 Happens-before 关系,确保该关系在重构前后没有改变.在 FLock 工具的重构对象中,重构前后代码都涉及同步关系,可以在同步关系的基础上建立 Happens-before 关系,进而进行规则判定.规则 5 从保持原有临界区的原子性角度对锁分解进行了约束,确保原有临界区的原子性没有被破坏.如果存在 op_{ip} 和 op_{iq} 访问同一内存位置,有可能因为线程交互而改变原有操作语义,在这种情况下,FLock 只推断一个写锁而不进行锁分解.

4 重构工具实现

FLock 是在 Eclipse JDT 框架下设计并以 Eclipse 插件的形式实现的,对 Eclipse 中的基础类 Refactoring 和 UserInputWizardPage 等进行扩展,实现相关的重构逻辑设计和可视化的重构工具界面设计.FLock 重构界面截图如图 4 所示,展示了重构前后的代码之间的对比,其中,左侧为重构前的代码,右侧为重构后代码的预览.

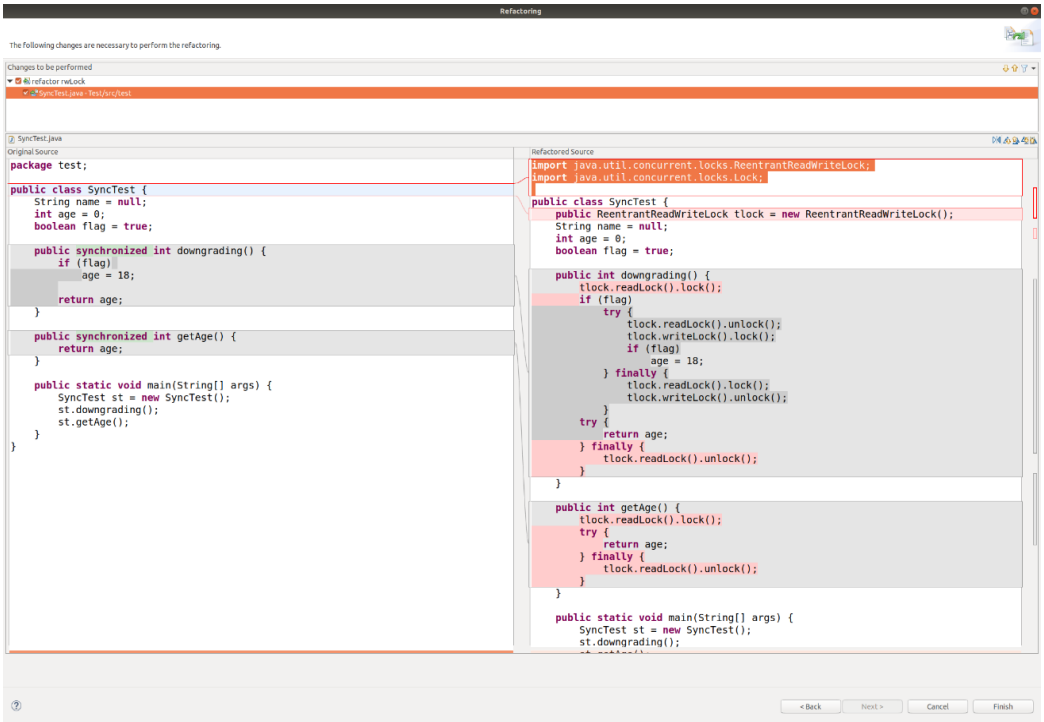


Fig.4 Screenshot of the FLock

图 4 FLock 重构工具界面

5 实验评估

本节对所提出的方法和工具进行了实验评估.首先对实验配置和测试程序进行介绍,然后从重构数目、改变的代码行数和时间等方面给出了实验结果,并对结果进行了分析[18].

5.1 实验配置

所有实验都是在 HP Z240 工作站上完成的,该工作站搭载 Intel Core i7-7700 处理器,该处理器主频为 3.6GHz,有 4 个处理核,均支持超线程技术,可以支持 8 个线程同时运行,内存为 8GB.软件上,操作系统使用 Ubuntu 16.04,使用 Eclipse 4.12.0 作为重构工具的开发平台,使用 JDK 1.8.0_221 和程序分析工具 WALA 1.52.

5.2 测试程序

本文选取了 11 个实际应用程序来验证我们重构工具的有效性和适用性,这些应用程序包括 HSQLDB[19],Jenkins[20],Cassandra[21],SPECjbb2005[22],JGroups[23],Xalan[24],Fop[25],RxJava[26],Freedomotic[27],Antlr[28],MINA[29].其中,HSQLDB 是一个开源的 Java 数据库,Cassandra 是 Apache 公司的开源分布式 NoSQL 数据库系统,Jenkins 是一个开源的自动化服务器,JGroups 是群组通信工具,SPECjbb 2005 是 Java 应用服务器测试程序,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,Freedomotic 是一个开源的物联网框架,Antlr 是一个解析器生成器,MINA 是 Apache 公司的网络应用框架.这些程序的版本号信息、包含同步方法和同步块的个数以及源码行数见表 1.

Table 1 Benchmarks and their configuration
表 1 测试程序及其配置

名称	版本	同步块	同步方法	代码行数
HSQLDB	2.4.1	71	613	175 568
Jenkins	2.190.2	47	227	160 246
Cassandra	3.11.4	13	226	431 022
SPECjbb2005	1.01	22	168	12 519
JGroups	4.1.5	41	138	122 885
Xalan	2.7.2	31	51	89 149
Fop	2.3	7	25	198 555
RxJava	2.2.13	20	8	99 623
Freedomotic	5.6.0	6	15	56 211
Antlr	4.7.2	13	3	60 515
MINA	2.1.3	3	9	23 482

5.3 实验结果及分析

在实验中,使用 FLock 对 11 个测试程序进行自动重构,对重构个数、代码行数、重构后程序的准确性进行了验证,并与重构工具 Relocker 和 CLOCK 进行了对比.

5.3.1 锁重构个数

我们首先对 FLock 重构后的不同类型锁个数进行了汇总,重构结果见表 2.

Table 2 Refactoring results by FLock
表 2 FLock 重构结果

测试程序	重构前		重构后						重构
	代码行数	内置监视器	代码行数	锁降级	锁分解	读锁	写锁	违背一致性规则	时间(s)
HSQLDB	175 568	684	179 324	6	39	109	530	41	18
Jenkins	160 246	274	162 008	3	14	19	238	23	16
Cassandra	431 022	239	432 442	2	24	39	174	26	73
SPECjbb2005	12 519	190	13 301	1	2	58	129	17	24
JGroups	122 885	179	124 126	5	33	28	113	9	7
Xalan	89 149	82	89 386	2	5	19	56	12	19
Fop	198 555	32	198 790	2	0	9	21	4	15
RxJava	99 623	28	99 794	0	1	8	19	2	8
Freedomotic	56 211	21	56 485	2	1	2	16	3	5
Antlr	60 515	16	60 574	2	5	1	8	2	5
MINA	23 482	12	23 549	0	3	1	8	0	2
总计	1 429 775	1 757	1 439 779	25	127	293	1 312	139	192

从实验结果可以看出:在 11 个程序中,共有 391 个粗粒度锁重构为细粒度锁.内置监视器对象转换为锁降级模式的个数有 25 个,其中,HSQLDB 中最多,包含 6 个,集中分布在 org.hsqldb 包里的 Session 类和 Table 类中,这两个类分别是用来执行 session 和保存数据库表的数据结构和方法;在 RxJava 和 MINA 中,由于原程序中包含的内置监视器对象较少,在重构之后没有监视器对象转换为锁降级模式.内置监视器对象转换为锁分解模式的个数为 127 个,在 HSQLDB,Cassandra 和 JGroups 测试程序中重构后转换为锁分解模式的个数较多;测试程序 Fop 中包含 32 个内置监视器对象,重构后没有内置监视器转换为锁分解模式.在重构为读锁方面,有 293 个内置监视器对象转换为读锁,在测试程序 HSQLDB 和 Cassandra 中,读锁的占比相对较高;在程序 Antlr 和 MINA 中,使用锁分解模式比使用读锁的个数多.

在重构工具 FLock 中,加入了对临界区的一致性规则检测,表 2“违背一致性规则”列展示了不满足一致性检测的临界区数目.对于所有的测试程序,除了 MINA 测试程序外,其余 10 个程序均存在违背一致性检测规则的情况发生,共有 139 个,其中,HSQLDB 中最多,有 41 个.对于这些不满足一致性规则的临界区,我们没有对他们进行锁分解,而是采用写锁作为临界区的保护模式.

5.3.2 重构前后改变的代码行数

我们对重构前后改变的代码行数进行了统计,这些代码行数的改变在一定程度上反映了程序员在手动重构时所需要花费的工作量.我们使用 SLOCCount(https://dwheeler.com/sloccount/)工具对代码行数进行统计,该

工具是由 David A.Wheeler 开发的用于精确统计代码行数的工具,不仅适用于多个操作系统平台,而且适合于多种语言.

11 个测试程序共包含 1 429 775 行代码,重构后代码行数为 1 439 779 行,增加了 10 004 行代码.由于重构前使用的是同步锁,重构后使用的读写锁需要显示的加锁和解锁,并且需要使用 try-finally 语句块,所以重构后会增加代码行数.对于 HSQLDB 测试程序,由于其包含的同步锁数目最多,重构前后改变的代码行数也最多,达到 3 756 行;对于 Jenkins,Cassandra,SPECjbb2005 和 JGroups 测试程序,由于它们包含的同步锁数目在 100 到 300 之间,代码改变的行数也较多,分别增加了 1 762 行、1 420 行、782 行和 1 241 行;对于 RxJava,Freedomotic,Antlr 和 MINA 测试程序,由于包含的同步锁较少,代码行数的改变也较少,分别有 171 行、274 行、59 行和 67 行.

这些代码行数在一定程度上体现了程序开发人员重构程序时的工作量,如果使用手动重构的方式,开发人员需要首先在程序中找到同步锁出现的位置,然后进行修改.例如:在手动重构 HSQLDB 测试程序时,需要在 17 万行的代码中寻找 684 个同步锁并进行重构,最终结果会增加 3 756 行代码,这种手动重构耗时较长并且容易给程序引入新的错误,对于该测试程序,我们发现同步锁大部分分布在 org.hsqldb 包中,分布相对比较集中;在 FOP 中,32 个内置监视器对象在重构后只增加 235 行代码,但 32 个监视器对象分布在 2 034 个 java 文件中,分布非常稀疏,遍历过程十分耗时.在使用我们开发的自动重构工具重构 HSQLDB 和 FOP 时,分别仅需要 18s 和 15s 即可完成,可以大大节省程序员的工作量.

5.3.3 重构时间

11 个测试程序重构的总耗时为 192s,每个程序平均耗时 17.5s,见表 2.HSQLDB 中监视器对象较多,有 684 个监视器对象,重构耗时为 18s;程序 Cassandra 规模相对较大,源码行数为 431 022 行,重构耗时为 73s;JGroups 和 Xalan 重构耗时分别为 24s 和 19s;SPECjbb2005,RxJava,Freedomotic,Antlr 和 MINA 的程序规模相对较小,重构耗时约为 2s~8s.通过对这些程序的重构时间进行分析,我们发现:重构工具在重构时的时间消耗主要是用于程序的静态分析上,程序越大,静态分析时间越长,造成总的重构时间也越长.对于 FOP 测试程序中,源码行数为 198 555 行,虽然其与 HSQLDB 程序的代码规模相当,但其中包含同步锁个数非常少,分布相对比较稀疏,通过手动的方式进行重构会花费大量的时间在搜索代码上;而 FLock 可以自动地完成重构,用时也仅仅为 15s,大大减少重构耗时,有效提高开发效率.

5.3.4 重构的正确性

为了对重构后测试程序的正确性进行验证,我们主要从以下 3 个方面进行了验证,包括验证推断锁的类型是否正确、加锁和解锁的位置是否正确、锁结构的使用是否正确.由于目前没有相关的自动验证工具,我们主要采用了手工检查的方式.

在验证推断锁类型的准确性方面,我们手动地检查每一个测试程序中每一个临界区的代码,我们发现:推断出来的每一个锁都符合我们的推断规则,都是正确的类型.此外,我们发现:在 Cassandra 测试程序中,部分代码在重构前就使用了读写锁.我们将这些读写锁重构为同步锁,然后使用 Flock 再对其进行重构,结果发现:Flock 推断的锁类型和原有的锁类型基本相同,只有一处存在不同之处.该处发生在 CompactionStrategyManager 类的 maybeReload 方法中,该方法原本使用写锁,但是 Flock 将其推断为锁分解的方式,经过手工检查发现:使用锁分解的方式并未改变程序原逻辑,是正确的细粒度加锁方式.

在判断加锁和解锁位置以及锁的使用方面,我们主要检查了:(1) 每种锁的加锁操作是否对应一个解锁操作;(2) 解锁操作是否被放入 finally 语句块中;(3) 细粒度锁的结构是否正确.经过检查,我们并未发现相关错误.我们也通过运行程序的方式检查了程序的正确性,发现这些程序在执行过程中并没有报错,都可以正确运行.

5.3.5 重构前后性能对比

本节选取 HSQLDB,Jenkins,Cassandra,JGroups 和 SPECjbb2005 进行性能测试.之所以选择这 5 个程序,是因为它们在发布的版本中都提供了相应的基准测试程序.

HSQLDB 提供了 JDBCBench 测试程序,该程序的测试结果如图 5 所示,其中:横坐标为处理事务数,分别给出了事务数为 100k,200k,300k 和 400k 的情况;纵坐标为事务处理率.从图中可以看出:该测试程序在处理 100k

个事务时,程序重构后使用细粒度读写锁的事务率比重构前使用同步锁的事务率略有提升,但不明显;在处理事务量达到 200k,300k 和 400k 时,重构后比重构前的程序在事务率上的提升大约有 15%,19%和 22%,提升较为明显.总的来说,HSQLDB 使用细粒度的读写锁在事务率方面重构后比重构前平均提升了 18%,说明使用细粒度锁有助于提升 HSQLDB 的性能,而我们设计的工具可以帮助更快地转换为细粒度锁.

对于 Jenkins,我们选取了其中的单元测试程序 UpdateCenterTest,FunctionsTest 和 FilePathTest 进行了测试.图 6 给出了 3 个单元测试的执行时间,可以看出:单元测试 UpdateCenterTest 在重构前后程序的执行时间基本没有变化,而 FunctionsTest 在重构后执行时间缩短了 17.3%,FilePathTest 在重构后执行时间缩短了 20.9%.

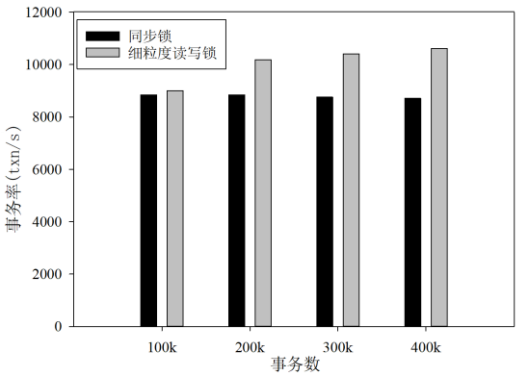


Fig.5 Performance comparison of the HSQLDB benchmark before and after refactoring

图 5 HSQLDB 重构前后的性能比较

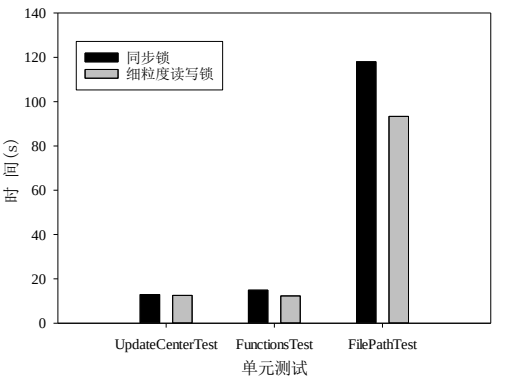


Fig.6 Performance comparison of the Jenkins benchmark before and after refactoring

图 6 Jenkins 重构前后的性能比较

对于 JGroups,我们选取了测试程序 RoundTrip,RoundTripRpc,UnicastTest 和 UnicastTestRpc 进行测试,这 4 个测试程序都是测试两个群组之间消息的发送与接收能力.实验结果如图 7 所示,其中,横坐标为测试程序,纵坐标为请求处理率.从结果中可以看出:4 个测试在重构后请求处理率都有所提升,分别提升了 15.5%,7%,5%和 9%.

对于测试程序 Cassandra,我们选取了其中的单元测试程序 BatchTests,SimpleQueryTest 和 ManyRowsTest 进行性能测试,图 8 给出了 3 个单元测试程序重构前后的执行时间.对于单元测试程序 BatchTests,Cassandra 重构后程序的执行时间与重构前程序的执行时间基本相同;对于 SimpleQueryTest 和 ManyRowsTest 这两个单元测试程序,执行时间都有不同程度的下降,都较重构前的执行时间大约缩短了 7%.

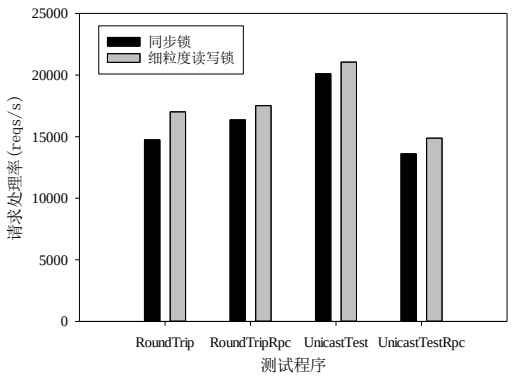


Fig.7 Performance comparison of the Jgroups benchmark before and after refactoring

图 7 JGroups 重构前后的性能比较

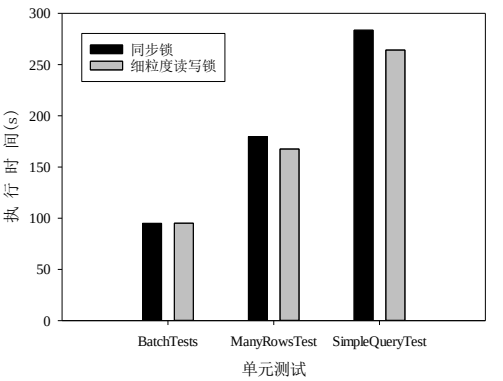
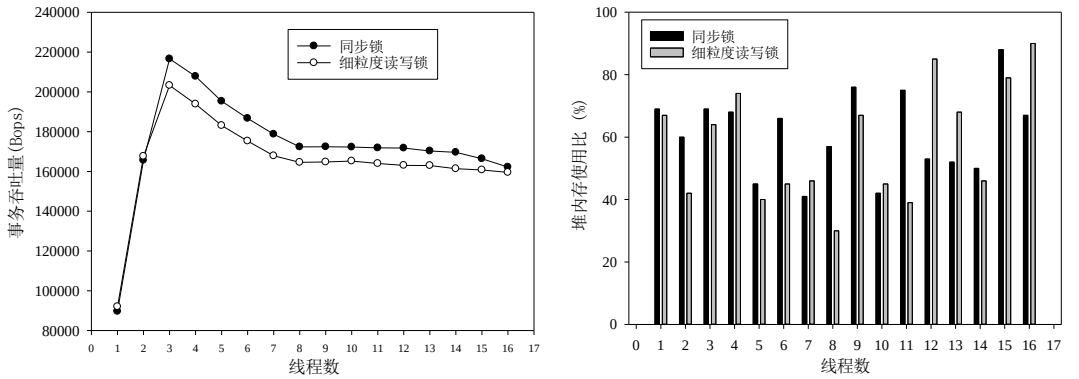


Fig.8 Performance comparison of the Cassandra benchmark before and after refactoring

图 8 Cassandra 重构前后的性能比较

图 9 给出了 SPECjbb 2005 运行结果,其中,图 9(a)给出了吞吐量随线程数变化的情况,图 9(b)给出了堆内存

使用的百分比随着线程数变化的情况.从图 9(a)可以看出:当吞吐量达到峰值之后,使用同步锁的吞吐量要比使用细粒度读写锁的吞吐量稍微高一些.这也说明并不是所有情况下,使用细粒度读写锁性能都比使用同步锁要好.图 9(b)给出了在不同线程执行情况下使用的堆内存占总内存的百分比,在线程数为 8 时,程序重构前的堆内存使用占比高于程序重构后;而在线程数为 12 时,程序重构后的堆内存使用占比高于程序重构前.这说明在不同锁模式和不同线程数下堆内存的使用情况各有不同,而我们设计的重构工具为开发人员提供一种快速的细粒度锁和粗粒度锁重构的方式,开发人员可以通过测试来找到程序堆内存使用最合适的情况.



(a) 事务吞吐量 (b) 堆内存使用比

Fig.9 Performance comparison of the SPECjbb2005 benchmark before and after refactoring

图 9 SPECjbb2005 重构前后的性能比较

5.3.6 工具对比

我们将 FLock 和已有的重构工具 Relocker[5]进行了对比.Relocker 是 Schafer 等人设计实现的一个自动重构工具,可以把同步锁重构为可重入锁或读写锁.该工具是一种粗粒度锁的推断模式,不能实现细粒度的加锁模式.由于 Relocker 工具开发较早,不支持高版本的 JDK,因此在该实验中使用的 JDK 版本为 1.6.测试程序选择了 HSQLDB 和 Cassandra,版本分别为 1.8.0.10 和 0.4.0(比表 1 的版本低),这两个程序中包含的同步锁的数量也较之前实验选取的版本有所不同:HSQLDB 总共包含 266 个同步锁,Cassandra 总共包含 57 个同步锁.我们也尝试使用 Relocker 对其他测试程序进行重构,但由于 Relocker 开发较早,均不支持对这些版本进行重构.

表 3 给出了 Relocker 和 FLock 重构结果的对比,可以看出:Relocker 只使用读锁或写锁进行同步保护,而且还存在不能重构的问题;FLock 不仅可以推断读锁和写锁,而且可以实现更为细粒度的锁重构.在读锁的推断上,FLock 比 Relocker 推断出了更多的读锁,经人工验证,使用 FLock 进行重构后的读锁使用正确.Relocker 的推断结果不准确的原因可能与分析深度有关,这里的分析深度使用的是 Relocker 的默认值.在重构时间上,Relocker 重构 HSQLDB 耗时 7s,FLock 耗时 6s;Cassandra 程序规模较大,Relocker 重构耗时 50s,FLock 重构耗时 42s.总体来看,FLock 在重构效率上有所提升.

Table 3 Comparison of Relocker and FLock

表 3 Relocker 和 FLock 重构对比

名称	Relocker				FLock				
	读锁	写锁	不能重构	时间(s)	锁降级	锁分解	读锁	写锁	时间(s)
HSQLDB 1.8.0.10	31	212	23	7	8	23	45	190	6
Cassandra 0.4.0	4	50	3	50	3	1	6	47	42

我们还将 FLock 与重构工具 CLOCK[7]进行了对比,CLOCK 是一个面向邮戳锁的自动重构工具.这里选取了本文实验中监视器对象相对较多的 7 个测试程序进行重构对比,实验结果见表 4.由于邮戳锁是不可重入锁,所以 CLOCK 对部分临界区不能执行重构,其中,HSQLDB 和 SPECjbb 2005 包含较多不能重构的锁,分别为 61 和 75 个;由于 CLOCK 在锁推断上和 FLock 不同,所以在锁降级重构数量上也有所差异.两个工具在重构效率上,CLOCK 在重构 HSQLDB,Cassandra 和 Fop 时比 FLock 慢 1s~2s,其他程序在重构时间上基本一致.

Table 4 Comparison of CLOCK and FLock

表 4 CLOCK 和 FLock 重构对比

名称	CLOCK							FLock				
	读锁	写锁	乐观读锁	升级锁	降级锁	不能重构	时间(s)	读锁	写锁	锁降级	锁分解	时间(s)
HSQLDB	66	494	30	23	10	61	19	109	530	6	39	18
Jenkins	5	200	11	13	0	45	16	19	238	3	14	16
Cassandra	13	158	10	18	3	37	75	39	174	2	24	73
SPECjbb 2005	14	65	36	0	0	75	7	58	129	1	2	7
JGroups	10	102	3	13	3	48	24	28	113	5	33	24
Xalan	3	43	2	8	0	26	19	19	56	2	5	19
Fop	6	15	0	8	1	2	16	9	21	2	0	15

关于重构后程序性能方面,我们对 HSQLDB 和 SPECjbb2005 这两个程序使用不同重构工具重构后,使用 3 种锁的性能进行了对比.HSQLDB 的实验结果如图 10 所示,结果表明,粗粒度的读写锁和邮戳锁在事务率上要低于使用细粒度读写锁的程序.SPECjbb2005 的实验结果如图 11 所示,由于 JDK 版本原因,Relocker 不能对 SPECjbb2005 进行重构,这里只对比了 CLOCK 和 FLock 的重构结果.可以看出:程序在使用细粒度读写锁时,相较于使用粗粒度读写锁和邮戳锁的程序,事务率有明显提升.

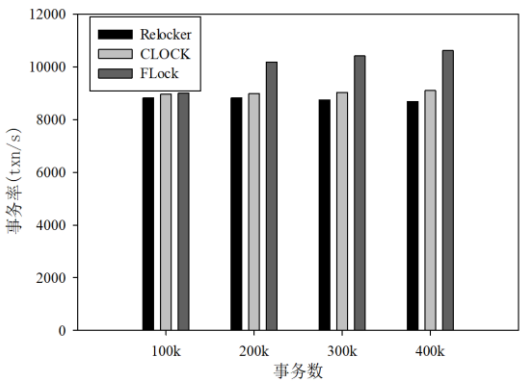


Fig.10 Performance comparison of the HSQLDB benchmark after refactoring

图 10 HSQLDB 重构后的性能比较

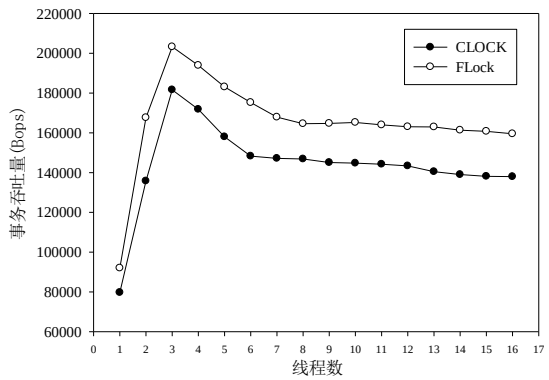


Fig.11 Performance comparison of the SPECjbb2005 benchmark after refactoring

图 11 SPECjbb2005 重构后的性能比较

5.3.7 有效性威胁

实验中有几个可能威胁有效性的因素.

1) 由于并发程序执行的不确定性,不排除性能测试实验结果可能存在细微的偏差.为了避免误差,我们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) 在实验中,我们采用手工检查的方式对重构后程序的正确性进行验证.手动的检查方式存在着一些不足之处,可能会出现人为验证不准确等问题.为了减少不准确情况的发生概率,我们选取了两组学生分别进行两次检查的方式,尽可能避免出现问题;
3) 本实验只选取了 11 个应用程序对 FLock 进行了评估,它们并不能代表所有程序,不能完全证明 FLock 在所有的应用程序中可以成功完成重构.但是我们选取的应用程序涉及数据库、群组通信、物联网等多个领域,具有很好的代表性.未来的工作中,我们也将使用更多应用程序对 FLock 进行测试.

6 相关工作

本文实现了面向细粒度读写锁的自动重构工具,我们主要关注的相关工作有两个方面:面向锁的优化和自动化的重构工具.

6.1 面向锁的优化

Emmi 等人[3]提出了一种自动锁分配技术,采用带有原子性规范的注解对程序进行标记,自动推断程序中应该获取锁的位置.Kawachiya 等人[4]提出一种锁保留算法,该算法允许锁被线程保留,当一个线程尝试获得锁操作时,如果线程保留了该锁,它就不用执行获取锁的原子操作;否则,线程使用传统的方法获得锁. Halpert 等人[30]提出了基于组件的锁分配技术,用于分析数据依赖性,自动将具有可调粒度的锁对象分配给临界区.Tamiya Onodera 等人[31]设计了一种基于锁保留的自旋锁,并使用这种自旋锁替换传统的自旋锁.Arbel 等人[8]提出了使用乐观同步替换程序中的一些加锁代码,以减少争用,可以在不牺牲正确性的情况下提高其可扩展性.Bavarsad[9]针对软件事务性内存,提出了两种优化技术来克服全局时钟的开销:第一种技术是读写锁定分配,它不利用任何中央数据结构来保持事务的一致性,仅当事务成功提交时,此方法才能提高软件事务性内存的性能;第二种优化技术是一种动态选择基线方案的自适应技术,用来减小读写锁定分配的成本.Gudka[32]提出了一种针对 Java 的锁定推断方法,使用锁定推断来实现原子块,该方法可以全面分析 Java 库方法.以上工作的研究目的与我们相似,通过锁分配、锁保留、原子块等技术减少临界区竞争;而我们使用锁降级、锁分解实现对临界区的细粒度保护方式,从而减少临界区竞争.

Hofer 等人[33]提出了一种通过跟踪 Java 虚拟机中的锁定事件来分析 Java 应用程序中的锁争用的方法,该方法不仅检测线程在锁上被阻塞情况,还检测是哪个其他线程通过持有该锁来阻止它,并记录它们的调用链.Tallent[34]提出了 3 种量化锁争用的方法,可以在低开销的前提下,有效提供锁争用检测精度.Inoue[35]提出了一个基于 Java 虚拟机,使用硬件性能计数器来检测应用程序获取锁的位置以及阻塞位置的方法.以上研究是用于揭示锁争用的原因以及识别锁性能瓶颈的分析工具,我们的研究提出的是一个解决锁争用问题的重构工具.

6.2 自动化的重构工具

Dig 等人提出了一个软件并行化重构工具 CONCURRENCER[36],该工具可以将串行的 Java 代码进行并行化重构,以及面向循环并行化重构的重构工具 Relooper[37].Wloka 等人[38]提出了一个用于把串行程序转变为可重入的程序的自动重构工具 Reentrancer,从而使程序更易并行化实现.Brown 等人[39]提出了一个用于生成并行程序的重构工具 ParaPhrase,从而增加并行程序的可编程性.以上研究以各种形式实现了重构工具,并很好地实现了工具的自动化,但都是面向并发程序的自动重构工具,而我们的工具是面向锁的自动重构.

McCloskey[40]提出了悲观的原子块,在不牺牲性能或兼容性的情况下,保留了类似事务性内存中乐观原子块的许多优点,并实现了自动重构工具 Autolocker. Schäfer 与 IBM T.J. Watson 研究中心合作设计了一种面向 Java 显示锁的重构工具 Relocker[5],它能将同步锁重构为可重入锁,以及将可重入锁重构为读写锁.与 Relocker 相比,我们的重构方法引入了锁降级、锁分解模式,使重构的适用性更广.Zhang 等人[6]提出了一种用于在字节码级别锁重构方法,由于在字节码层实现,可视性较差一些;我们的重构工作直接在源代码层实现,转换过程更直观.Zhang 等人[7]提出了一个面向邮戳锁的自动重构工具 CLOCK,该重构工具支持优化读锁和锁升级等操作,但是受限于邮戳锁的非可重入性,适用范围有限.我们的研究面向细粒度锁的重构则不受该限制.

Tao 等人[1]提出了针对 Java 程序、根据类属性域划分锁保护域的自动锁分解重构方法,并以 Eclipse 插件形式实现了自动重构工具.Yu 等人[2]在进行优化同步瓶颈的研究中提出了一种锁分解方法,该方法重新构造锁的依赖关系,使用细粒度的锁来保护不相交的共享变量集,并实现了工具 SyncProf. Greenhouse 等人[41]根据类的功能将对象的状态划分为多个子区域,然后通过不同的锁来保护每个分区锁分解方法.在工业界,得到广泛应用的重构工具包括集成在 IntelliJ IDEA 上的重构插件 LockSmith[10]以及一个基于 Eclipse JDT 的并发重构插件[11],这两个插件都可以实现锁分解、锁合并等重构.以上研究均是通过不同锁对象对类中的方法实现锁分解,我们的研究是通过读写锁的分离在方法内实现锁分解.

7 总 结

本文提出了一种面向细粒度锁的自动重构方法,该方法结合借助访问者模式分析、别名分析、负面效应分

析等多种程序分析技术对读写模式进行分析,并设计了基于下推自动机的锁模式识别方法.以 Eclipse 插件的形式实现了自动重构工具 FLock,在 HSQLDB,Jenkins 和 Cassandra 等 11 个开源项目中使用该重构工具,对本文提出的方法进行了验证.实验中总计重构了 1 757 个监视器对象,每个程序重构平均用时 17.5s.与手动重构相比,可以有效提升重构效率.实验结果表明,该重构工具可以有效地实现粗粒度锁到细粒度锁的转换.我们下一步的工作包括:1) 探索更多的细粒度锁的重构模式,尝试发现更多的可以使用细粒度锁的场景;2) 使用更多实际应用程序对 FLock 进行测试;3) 完善重构工具 FLock,通过比较使用粗粒度锁和细粒度锁的程序性能,在重构算法中加入性能权衡,帮助程序员在粗粒度锁和细粒度锁之间做出选择,进而决定是否进行重构.

References:

[1] Tao B, Qian J. Refactoring Java concurrent programs based on synchronization requirement analysis. In: Proc. of the 2014 IEEE Int'l Conf. on Software Maintenance and Evolution. IEEE, 2014. 361−370. [doi: 10.1109/ICSME.2014.58]

[2] Yu T, Pradel M. SyncProf: Detecting, localizing, and optimizing synchronization bottlenecks. In: Zeller A, ed. Proc. of the 25th Int'l Symp. on Software Testing and Analysis. ACM, 2016. 389−400. [doi: 10.1145/2931037.2931070]

[3] Emmi M, Fischer JS, Jhala R, *et al*. Lock allocation. In: Hofmann M, ed. Proc. of the ACM SIGPLAN Notices. ACM, 2007. 291−296. [doi: 10.1145/1190216.1190260]

[4] Kawachiya K, Koseki A, Onodera T. Lock reservation: Java locks can mostly do without atomic operations. In: Ibrahim M, ed. Proc. of the ACM SIGPLAN Notices. ACM, 2002. 130−141. [doi: 10.1145/582419.582433]

[5] Schafer M, Sridharan M, Dolby J, *et al*. Refactoring Java programs for flexible locking. In: Taylor RN, ed. Proc. of the 2011 33rd Int'l Conf. on Software Engineering (ICSE). IEEE, 2011. 71−80. [doi: 10.1145/1985793.1985804]

[6] Zhang Y, Shao S, Liu H, *et al*. Refactoring Java programs for customizable locks based on bytecode transformation. IEEE Access, 2019,7:66292−66303. [doi: 10.1109/ACCESS.2019.2919203]

[7] Zhang Y, Dong S, Zhang X, *et al*. Automated refactoring for stampedlock. IEEE Access, 2019,7:104900−104911. [doi: 10.1109/ACCESS.2019.2931953]

[8] Arbel M, Golan-Gueta G, Hillel E, *et al*. Towards automatic lock removal for scalable synchronization. In: Moses Y, ed. Proc. of the Int'l Symp. on Distributed Computing. Springer-Verlag, 2015. 170−184. [doi:10.1007/978-3-662-48653-5_12]

[9] Bavarsad AG, Atoofian E. Read-write lock allocation in software transactional memory. In: Proc. of the 2013 42nd Int'l Conf. on Parallel Processing. IEEE, 2013. 680−687. [doi:10.1109/ICPP.2013.81]

[10] Locksmith.https://intellij-support.jetbrains.com/hc/en-us/community/posts/206761105%E2%80%93Ann-LockSmith-concurrency-oriented-refactorings-for-IntelliJ-IDEA

[11] Concurrency-related refactorings for JDT. https://wiki.eclipse.org/Concurrency-related_refactorings_for_JDT

[12] Pinto G, Torres W, Fernandes B, *et al*. A large-scale study on the usage of Java's concurrent programming constructs. Journal of Systems and Software, 2015,106:59−81. [doi: 10.1016/j.jss.2015.04.064]

[13] https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/locks/ReentrantReadWriteLock.html

[14] Eclipse JDT. https://www.eclipse.org/jdt/

[15] Ghezzi C, Jazayeri M. Programming Language Concepts. 3rd ed., New York: John Wiley & Sons, 2008. 214−220.

[16] WALA. http://wala.sourceforge.net/wiki/index.php/Main_Page

[17] Zhang C. FlexSync: An aspect-oriented approach to Java synchronization. In: Proc. of the 2009 IEEE 31st Int'l Conf. on Software Engineering. IEEE, 2009. 375−385. [doi: 10.1109/ICSE.2009.5070537]

[18] Shao S. An automated refactoring approach for fine-grained lock [MS. Thesis]. Shijiazhuang: Hebei University of Science and Technology. 2020 (in Chinese with English abstract).

[19] HSQLDB. http://hsqldb.org/

[20] Jenkins. https://jenkins.io/zh

[21] Cassandra. https://cassandra.apache.org/

[22] SPECjbb2005.https://www.spec.org/jbb2005/

[23] JGroups. http://www.jgroups.org/

[24] Xalan. http://xalan.apache.org/xalan-j/

[25] Fop. https://xmlgraphics.apache.org/fop/

[26] RxJava. http://reactivex.io/

[27] Freedomotic. https://www.freedomotic-iot.com

[28] Antlr. https://www.antlr.org/

[29] MINA. http://mina.apache.org/

[30] Halpert RL, Pickett CJF, Verbrugge C. Component-Based lock allocation. In: Proc. of the 16th Int'l Conf. on Parallel Architecture and Compilation Techniques (PACT 2007). IEEE, 2007. 353−364. [doi: 10.1109/PACT.2007.23]

[31] Onodera T, Kawachiya K, Koseki A. Lock reservation for Java reconsidered. In: Odersky M, ed. Proc. of the European Conf. on Object-Oriented Programming. Springer-Verlag, 2004. 559−583. [doi: 10.1007/978-3-540-24851-4_26]

[32] Gudka K, Harris T, Eisenbach S. Lock inference in the presence of large libraries. In: Noble J, ed. Proc. of the European Conf. on Object-Oriented Programming. Springer-Verlag, 2012. 308−332. [doi: 10.1007/978-3-642-31057-7_15]

[33] Hofer P, Gnedt D, Schörgenhumer A, *et al*. Efficient tracing and versatile analysis of lock contention in Java applications on the virtual machine level. In: Avritzer A, ed. Proc. of the 7th ACM/SPEC on Int'l Conf. on Performance Engineering. ACM, 2016. 263−274. [doi: 10.1145/2851553.2851559]

[34] Tallent NR, Mellor-Crummey JM, Porterfield A. Analyzing lock contention in multithreaded applications. In: Govindarajan R, ed. Proc. of the 15th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming. ACM, 2010. 269−280. [doi: 10.1145/1693453.1693489]

[35] Inoue H, Nakatani T. How a Java VM can get more from a hardware performance monitor. In: Arora S, ed. Proc. of the 24th ACM SIGPLAN Conf. on Object Oriented Programming Systems Languages and Applications. ACM, 2009. 137−154. [doi: 10.1145/1640089.1640100]

[36] Dig D, Marrero J, Ernst MD. Refactoring sequential Java code for concurrency via concurrent libraries. In: Proc. of the 2009 IEEE 31st Int'l Conf. on Software Engineering. IEEE, 2009. 397−407. [doi: 10.1109/ICSE.2009.5070539]

[37] Dig D, Tarce M, Radoi C, *et al*. Relooper: Refactoring for loop parallelism in Java. In: Arora S, ed. Proc. of the 24th ACM SIGPLAN Conf. Companion on Object Oriented Programming Systems Languages and Applications. ACM, 2009. 793−794. [doi: 10.1145/1639950.1640018]

[38] Wloka J, Sridharan M, Tip F. Refactoring for reentrancy. In: Van Vliet H, ed. Proc. of the 7th Joint Meeting of the European Software Engineering Conf. and the ACM SIGSOFT Symp. on the Foundations of Software Engineering. ACM, 2009. 173−182. [doi: 10.1145/1595696.1595723]

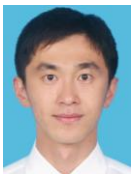
[39] Brown C, Hammond K, Danelutto M, *et al*. Paraphrasing: Generating parallel programs using refactoring. In: Beckert B, ed. Proc. of the Int'l Symp. on Formal Methods for Components and Objects. Springer-Verlag, 2011. 237−256. [doi: 10.1007/978-3-642-35887-6_13]

[40] McCloskey B, Zhou F, Gay D, *et al*. Autolocker: Synchronization inference for atomic sections. In: Morrisett JG, ed. Proc. of the Conf. Record of the 33rd ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages. ACM, 2006. 346−358. [doi: 10.1145/1111037.1111068]

[41] Greenhouse A, Halloran TJ, Scherlis WL. Observations on the assured evolution of concurrent Java programs. Science of Computer Programming, 2005,58(3):384−411. [doi: 10.1016/j.scico.2005.03.002]

附中文参考文献:

[18] 邵帅.面向细粒度读写锁的自动重构方法研究[硕士学位论文].石家庄:河北科技大学,2020.



张杨(1980－),男,博士,副教授,CCF 高级会员,主要研究领域为软件重构,并发软件分析.

邵帅(1996－),男,硕士生,主要研究领域为并发软件分析,智能化软件.

张冬雯(1964－),女,博士,教授,CCF 专业会员,主要研究领域为软件重构,并发软件分析.