面向Java锁机制的字节码自动重构框架

张 杨 张冬雯 仇 晶
(河北科技大学信息科学与工程学院 石家庄 050000)

摘 要 Java语言提供了同步锁、可重入锁和读写锁等几种锁机制,在并行程序设计中不同的数据结构使用这几种锁机制时获得的性能通常是不同的。为了在不同的锁机制之间进行自动转换,进而帮助程序员了解程序的性能,提出了一种面向Java锁机制的字节码自动重构框架,并基于该框架实现了字节码重构工具Lock2Lock。Lock2Lock在Quad中间表示的基础上对字节码进行静态分析,并对分析的结果进行一致性验证,通过Javassist完成字节码的重构。使用红黑树、消费者生产者程序以及SPECjbb2005 3个测试程序对Lock2Lock重构工具进行了测试,结果表明,Lock2Lock可以成功地实现从同步锁到可重入锁或读写锁的重构。

关键词 Java锁,软件重构,程序分析,字节码转换
中图法分类号 TP311 **文献标识码** A **DOI** 10.11896/j.issn.1002-137X.2015.11.017

Automated Refactoring Framework for Java Locks

ZHANG Yang ZHANG Dong-wen QIU Jing
(School of Information Science and Engineering, Hebei University of Science and Technology, Shijiazhuang 050000, China)

Abstract Java locks, such as synchronized, ReentrantLock, and ReadWriteLock, often obtain different performances when applied on different data structures. To learn about which lock will obtain the best performance, there is a strong need to transform from one lock to another automatically. This paper presented an automated refactoring framework for the transformation of Java locks. The framework extensively relies on static program analysis to perform the refactoring. The evaluation was performed using three benchmarks, such as red-black tree, producer-consumer problem and SPECjbb2005. The successful refactoring results are observed and the time of the refactoring is acceptable.

Keywords Java lock, Software refactoring, Program analysis, Bytecode transformation

1 引言

随着多核/众核处理器的普及,并行程序设计将成为未来软件开发技术的主流。在并行程序设计中,一方面要保证获得高性能,另一方面要确保数据访问的正确性。同步机制是保证程序状态和数据访问正确性的必备措施,被认为是可以和软件的业务逻辑相分离的一种典型的横切属性[1]。目前常用的同步控制方式包括锁、无锁算法[2](Lock-Free Algorithm,LFA)和软件事务性内存[3](Software Transactional Memory,STM)等,其中,LFA和STM被认为是多核时代避免锁竞争问题的解决方案,但是LFA相对复杂且难于掌握,STM则开销较大并且对I/O操作和线程间通信的支持不足。虽然锁的使用会导致锁竞争问题,但是锁已经被程序员广泛接受,在未来一段时间内将继续存在和使用。

Java语言提供了同步锁、可重入锁和读写锁等几种锁机制,不同的数据结构应用不同的锁机制获得的性能通常是不同的(见第2节)。为了了解程序使用哪一种锁可以获得更好的性能,程序员往往需要在不同的锁机制之间进行尝试(例如将程序中使用的同步锁全部替换为读写锁)。传统的做法是对程序进行手动改写,这种做法在程序规模较小时可以适用,但对于某些大规模的程序,由于使用锁的地方较多并且锁对象使用不一,这种手动的改写工作量巨大,而且容易出错。因此迫切需要提供一种在不同的锁机制之间自动转换的方法和工具,以帮助程序员了解程序使用哪一种锁时性能最优。

软件重构是一种在不改变软件系统外部行为的前提下通过改变软件内部结构来提高软件质量的活动。将软件重构和并行程序设计技术相结合,有利于并行程序的设计和优化,主要表现为:(1)借助并行编程语言或并行库可以将原有的串行程序重构为并行程序,在不改变软件外部行为的情况下,通过提高软件内部的并发程度来提高程序的性能;(2)对并行程序内部影响程序性能的因素(如同步)进行重构,以进一步优化程序的性能;(3)通过对并行程序内部结构进行重构,可以增强并行程序的可维护性和可理解性。

本文将软件重构的方法应用于并行程序中的锁机制上,来实现从一种锁机制到另一种锁机制的自动重构。然而,面向Java锁机制的自动重构并不是一件容易的事,这主要因为(1)锁的使用通常具有全局性,在这种情况下局部分析很难保证重构的一致性;(2)在重构过程中,不仅要对加锁和解锁操

到稿日期:2014-11-15 返修日期:2015-01-20 本文受国家自然科学基金项目(61440012),河北省高等学校青年拔尖人才计划项目(BJ2014023),河北省自然科学基金项目(F2012208016)资助。

张 杨(1980-),男,博士,讲师,CCF会员,主要研究方向为软件并行化重构、并行程序设计,E-mail:zhangyang@hebust.edu.cn;张冬雯(1963-),女,博士,教授,主要研究方向为软件重构;仇 晶(1983-),女,博士,副教授,主要研究方向为软件工程、自然语言处理。

作进行重构，而且与锁相关的一些线程通信操作（如 wait、notify 和 notifyAll 等）也需要同时进行重构；(3)不同锁机制对锁对象定义和操作的方式不同，例如，同步锁依赖于隐式的内置监视器对象，因此同步锁不需要显式定义锁，而可重入锁和读写锁不仅需要显式地定义锁对象，而且读写锁还需要考虑如何使用读锁和写锁。

针对上述问题，本文提出一种面向 Java 锁机制的字节码自动重构框架，该框架在中间表示的基础上通过字节码分析、验证和转换技术，可以将程序中同步锁自动重构为可重入锁或读写锁。通过红黑树、生产者消费者程序和 SPECjbb2005 3 个测试程序对框架的有效性进行实验，实验表明，基于该框架实现的重构工具 Lock2Lock 可以成功地对这些程序进行重构。

2 背景及问题的提出

2.1 Java 锁机制

同步锁是一种互斥锁，以 synchronized 关键字作为修饰符，它的使用形式简单，易于理解和使用，但同步锁依赖于隐藏在对象后的内置监视器，很不直观。可重入锁和读写锁是从 JDK1.5 版本开始引入的锁机制，它和同步锁具有相同的基本行为和语义[6]，但增加了许多功能，如在尝试获取锁时可中断、测试锁是否正在被持有、非阻塞加锁操作和获取锁的顺序等。读写锁除了提供了可重入锁的一些特性外，还把锁分为读锁和写锁。在没有写锁控制的情况下，多个线程可以同时获取读锁。相对于其它两种锁，读写锁允许更大程度的并发。在选择使用读写锁时需要考虑读的频率、读写操作的持续时间以及正在读写的线程数等因素。有关 3 种锁机制更详细的介绍请参见 Java API 规范[6]。

2.2 问题的提出

本节对 Java 语言中 3 种线程不安全的数据结构 ArrayList、HashMap 和 TreeSet 分别使用同步锁、可重入锁和读写锁进行同步控制，并通过实验给出了程序使用不同的锁机制时的性能比较结果。图 1 给出了对 ArrayList 进行读写操作时使用不同锁机制的代码，其它几种结构的代码与其类似。

```
public class SyncTest {
    private List<Integer> myList;
    public SyncTest(List<Integer> myList) {
        this.myList=myList;
    }
    public synchronized Object get(int index) {
        return myList.get(index);
    }
    public synchronized boolean insert(int v) {
        return myList.add((Integer)v);
    }
}
```

(a)同步锁

```
public class ReenTest{
    private List<Integer> myList;
    private Lock lock=new ReentrantLock();
    public ReenTest(List<Integer> myList){
        this.myList=myList;
    }
    public Object get(int index){
        lock.lock();
        try{ return myList.get(index);
        }finally{lock.unlock();}
    }
    public boolean insert(int v) {
        lock.lock();
        try{ return myList.add((Integer)v);
        }finally{lock.unlock();}
    }
}
```

(b)可重入锁

```
public class RewrTest{
    private List<Integer> myList;
    private ReadWriteLock lock=new ReentrantReadWriteLock();
    public ReenTest(List<Integer> myList){
        this.myList=myList;
    }
    public Object get(int index){
        lock.readLock().lock();
        try{ return myList.get(index);
        }finally{lock.readLock().unlock();}
    }
    public boolean insert(int v) {
        lock.writeLock().lock();
        try{ return myList.add((Integer)v);
        }finally{lock.writeLock().unlock();}
    }
}
```

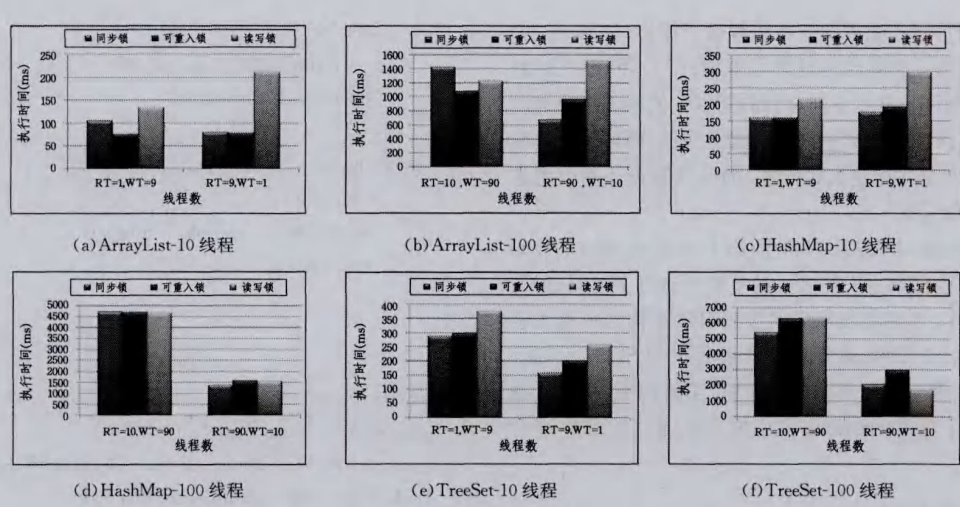
(c)读写锁

图 1 ArrayList 分别使用 3 种锁机制的代码示例

对每种数据结构使用 3 种锁的性能分别进行了测试。在实验中，分别选择 10 线程和 100 线程来执行读写操作，同时选取线程总数的 10%作为读线程或写线程的数量（图 2 中 RT 表示读线程数，WT 表示写线程数），每个线程将执行 50000 次的读操作或写操作，所有实验数据是在执行 10 次的基础上取平均值后得到的。

图 2 给出了 3 种锁机制的性能比较。图 2(a)是 ArrayList在 10 线程配置下的执行情况，使用可重入锁的执行时间最少，而使用读写锁的执行时间最多；而对于图 2(b)，当读线程数为 10、写线程数为 90 时，使用同步锁的执行时间最多；当读线程数为 90 而写线程数为 10 时，使用读写锁的执行时间最多。对于 HashMap，当线程总数为 10 时，读写锁的执行时间最多，如图 2(c)所示；当线程总数为 100 时，各种锁的执行时间相差不大，如图 2(d)所示；对于 TreeSet，在 10 线程时使用读写锁的执行时间最多，使用同步锁的执行时间最少，如图 2(e)所示；当读线程数为 90、写线程数为 10 时，读写锁的执行时间最少，如图 2(f)所示。

从实验结果可以看出一种锁机制的性能并不是绝对优于其它锁机制，虽然可以通过分析该数据结构的读写操作机制来预判读操作或写操作所花费的时间比重，但是不同线程数对于执行时间有很大影响。由此可见，使用不同锁时决定程序性能的因素较多，单纯依靠经验判断很难抉择，因此迫切需要提供一种方法及工具实现从一种锁机制到另一种锁机制的重构，进而帮助程序员了解程序使用哪一种锁性能较好。



(a) ArrayList-10 线程　(b) ArrayList-100 线程　(c) HashMap-10 线程

(d) HashMap-100 线程　(e) TreeSet-10 线程　(f) TreeSet-100 线程

图 2　3 种锁机制的性能比较

3　面向 Java 锁机制的重构

3.1　重构框架

本文提出一种面向 Java 锁机制的自动重构框架，如图 3 所示。

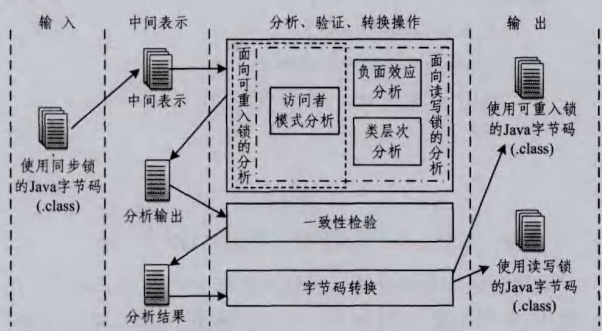


图 3　Java 锁重构框架

该框架以使用同步锁的 Java 字节码文件作为输入，输出使用可重入锁或读写锁的字节码文件。中间过程描述如下：

(1)为了便于程序分析，将字节码中间表示形式转换为其它中间表示形式。

(2)为了实现自动重构，采用程序静态分析技术。在实现面向可重入锁重构时，采用访问者模式分析，对程序中使用锁的代码进行遍历；在面向可重入锁重构的基础上，采用负面效应(Side-effect)分析实现面向读写锁重构的分析工作，该分析主要用于决定使用读锁还是写锁，分析完毕后产生分析输出。

(3)为了防止中间表示形式和字节码表示形式出现不一致的情况，对分析输出进行一致性检验，产生分析结果，分析结果将作为程序转换的依据。

(4)将分析结果和字节码文件作为输入，经过字节码转换，得到输出。

依据该重构框架，本文设计了一个重构工具 Lock2Lock(下载网址 http://lock2lock.googlecode.com)。Lock2Lock 采用 Quad 作为中间表示形式，分别使用 Joeq 编译器[4] 和 Javassist[5] 作为字节码的分析和转换框架。

3.2　Quad 中间表示

Quad 是 Joeq 编译器[4] 中使用的中间表示形式，它使用寄存器作为数据的存储结构，Quad 指令使用控制流的方式进行组织。Quad 除了可以保留程序中属性访问和虚方法调用的相关信息外，还可以保存程序的本地变量和临时变量信息。相对于 Java 字节码的栈结构，Quad 的寄存器结构更利于程序的分析[4]。同步块中的加锁和解锁操作在字节码表示形式中使用 monitorenter 和 monitorexit 表示，而在 Quad 中间表示中使用 MONITORENTER 和 MONITOREXIT 后跟寄存器名的形式来表示。

3.3　程序静态分析

3.3.1　面向可重入锁重构的静态分析

软件重构的对象即是程序分析的对象，从同步锁到可重入锁的重构通常需要解决以下几个问题：

(1)对程序进行遍历，找到所有使用同步锁的代码，以便在重构时将同步锁中的 synchronized 关键字替换为可重入锁的 lock 和 unlock 操作。

(2)建立同步锁对象和可重入锁对象之间的对应关系，即将程序中同步锁的内置监视器对象等价替换为可重入锁的对象。依据此对应关系，显式定义可重入锁对象。

(3)如果存在线程通信操作，则将其进行相应替换，例如，使用同步锁时需要用 wait、notify 和 notifyAll 方法，而使用可重入锁或读写锁时需要使用 await、signal 和 signalAll 方法。此外，线程通信操作对象也要同步替换。

对于问题(1)，因为锁的使用常常具有全局性，即在一个类中使用的锁对象在另一个类中也可以使用，所以需要对整个程序进行遍历分析。本文使用基于访问者模式的程序分析方法对字节码中同步锁的表示形式进行遍历，考虑同步锁有同步方法和同步块两种表现形式，所以分别进行遍历。对于同步方法，主要判断对象方法的修饰符中是否存在 synchronized 关键字；而对于同步块，主要对 Quad 表示中 MONITORENTER 和 MONITOREXIT 指令进行遍历。

解决问题(2)的难点在于同步锁通常使用内置监视器而不需要显式定义锁对象，而可重入锁通常需要显式地定义锁对象，并在锁对象上完成 lock 和 unlock 操作。为了建立同步锁内置监视器对象和可重入锁对象之间的对应关系，分别对同步方法和同步块两种表现形式对同步锁的内置监视器对象进行分析。对于同步方法，将当前类对象(对于实例方法)或当前类(对于静态方法)作为内置监视器对象；对于同步块，Lock2Lock 根据同步块的不同使用方式来识别内置监视器对象。

解决问题(3)的方法是在临界区中对 wait\notify\notify-

All 方法的调用进行遍历，发现在哪些对象上对这些方法进行调用。

面向可重入锁分析将得到锁使用的相关信息，如锁属性名、锁对象名以及线程通信相关的操作等，这些信息将作为一致性检查和字节码转换的依据。

3.3.2 面向读写锁重构的静态分析

除了上面的分析外，由于读写锁分为读锁和写锁，因此转换时需要判断使用读锁还是写锁。本文采用负面效应分析来推断使用读锁还是写锁，如图 4 所示。该分析算法对临界区中的代码进行判断，进而推断使用读锁还是写锁。负面效应分析算法在检查方法调用指令时(行 7)，可能有些方法是接口或抽象类的方法，使用类层次分析找到该抽象方法的具体实现。

```
1. public boolean hasSideEffect(Quad q){
2.     if(q 正在写静态属性或实例属性) return true;
3.     if(q 对数组进行写操作) return true;
4.     if(q 是堆内存写指令) return true;
5.     if(q 是方法调用指令){
6.         if(调用 wait\notify\notifyAll) return true;
7.         for(方法调用中出现的指令 newQuad)
8.             return hasSideEffect(newQuad);
9.     }
10.    return false;
11. }
```

图 4 负面效应分析算法

3.3.3 一致性验证

字节码和 Quad 中间表示之间不存在一对一的映射关系，因此对同步机制的字节码和 Quad 中间表示之间的对应关系进行了分析，发现字节码和 Quad 中间表示中加锁指令在文件中出现的顺序并不总是保持一致，这将给重构造成影响，主要因为程序静态分析是在 Quad 中间表示上完成的，而重构转换将在字节码基础上完成，因此有必要对软件分析结果进行一致性校验。解决方法是对程序分析阶段已经遍历过的每一个 MONITOENTER 指令检查其出现的次序，确保与字节码中锁的出现顺序一致。Joeq 编译器对于每一个 Quad 中间表示提供了 getBCI 方法，通过该方法可以得到每个 MONITORENTER 指令的字节码偏移量，偏移量小意味着出现较早，按照偏移量对分析结果进行调整可以确保分析结果与字节码中锁出现的次序一致。

3.4 字节码重构

使用 Javassist 字节码转换框架完成字节码重构，重构工作包括：

(1)向类中增加属性。锁属性名已经表明锁需要在哪个类中定义，只需要找到该类，在类中对其进行定义即可。该操作是通过继承 Javassist 中的类 javassist. expr. ExprEditor 并重写其 edit 方法来实现的。

(2)将同步锁中的 synchronized 关键字替换为可重入锁或读写锁的 lock 和 unlock 操作。由于同步块和同步方法的使用方式不同，因此分别进行重构。对于同步块，将 monitorenter 和 monitorexit 替换为锁的加锁和解锁操作；对于同步方法，去掉了 synchronized 修饰符，并在方法之前和之后加入加锁和解锁操作，并加入异常捕获语句，以确保异常发生时解锁操作总是会被执行。

(3)如果临界区存在线程通信操作，同样需要进行替换。

4 实验

4.1 实验配置

实验使用的机器配备了 Intel Xeon E5506 四核处理器，主频为 2.13GHz，内存 12GB。在软件上，使用 64 位 Linux 操作系统(内核版本 2.6.38)，JDK 版本为 1.7。

使用红黑树(RBTree)[7]、生产者消费者问题(PC problem)[8]和 SPECjbb2005[9]3 个测试程序验证 Lock2Lock 的重构功能。表 1 给出了这 3 个测试程序中同步方法、同步块以及线程通信操作的数量。

表 1 测试程序及其配置

测试程序	描述	同步方法数	同步块数	线程通信操作数
RBTree	红黑树	5	0	0
PC problem	生产者消费者问题	0	2	4
SPECjbb2005	SPEC 企业级应用基准测试程序	165	22	8

4.2 实验结果

(1)RBTree

RBTree 是一种自平衡二叉查找树。RBTree 是对 STM 测试时经常使用的一个基准测试程序，最初在 dstm2[7] 中发布，该程序包含了一系列事务性读写操作。为了测试 Lock2Lock，对该程序进行了修改，把该程序中事务性读写操作的部分改为使用同步锁进行同步控制，改写后的程序包含 5 个同步方法。

Lock2Lock 可以成功地重构该程序。Lock2Lock 进行面向可重入锁和面向读写锁的重构时间分别为 629ms 和 650ms，其中程序分析时间占总时间的绝大部分，大概是转换时间的 4 倍。由于都是同步方法，重构过程中没有出现 Quad 中间表示和字节码表示不一致的情况，即顺利通过了一致性检验。

(2)生产者消费者问题

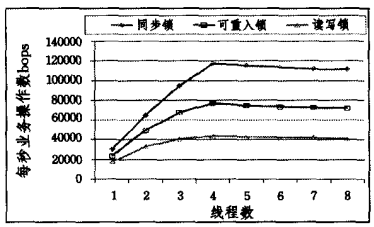
PC problem 是用于模拟线程间同步和通信的一个经典应用。生产者和消费者共享一定数量的缓冲池，生产者将生产的数据放入缓冲池中，消费者从缓冲池中取出数据进行消费，当缓冲池满后生产者不能生产而只能等待，在缓冲池空后消费者不能消费只能等待。文献[8]使用面向方面技术分离同步关注点实现了 PC problem 程序，这里使用了该程序的面向对象版本(http://code. google. com/p/lock2lock/ downloads/list)，该测试程序包含 2 个同步块和 4 个线程通信操作。

Lock2Lock 可以成功重构该程序，在重构过程中，首先对字节码进行分析，分别对每个类进行遍历，产生了分析的输出，一致性检验部分根据分析输出检测到不一致的情况，并进行了调整；最后，通过 Javassist 转换成功。面向可重入锁重构和面向读写锁重构的时间分别为 708ms 和 746ms。

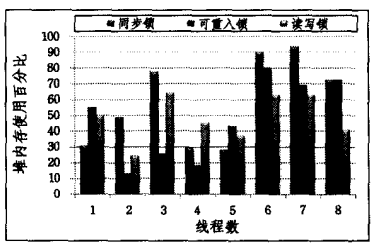
(3)SPECjbb2005

SPECjbb2005 是一个用于测试 Java 企业级应用的基准测试程序[9]，该测试程序包含 165 个同步方法、22 个同步块和 8 个线程通信操作。Lock2Lock 可以成功地对 SPECjbb2005测试程序进行重构，面向可重入锁重构和面向读写锁重构的时间分别为 3610ms 和 3790ms。图 5 给出了重构后 SPECjbb2005 程序的运行结果，其中使用同步锁的 SPECjbb2005 程序是由 SPEC 公司发布的未施加任何修改的程序，使用可重入锁和读写锁的程序是 Lock2Lock 重构后的

程序。图5(a)给出了在不同线程执行情况下每秒的业务操作数bops,从中可以看出,使用同步锁的SPECjbb2005程序的bops最多,使用读写锁的bops最少,这种现象不是由Lock2Lock重构引起,而是由该程序使用不同的锁引起的。图5(b)给出了在不同线程的执行情况下堆内存使用的百分比,可以看出,不同锁在不同线程数的情况下堆内存使用的百分比是不同的。因此,在具体应用中,可以使用Lock2Lock将SPECjbb2005中的同步锁重构为可重入或读写锁,进而测试使用哪一种锁时堆内存的使用情况最优。



(a)每秒业务操作数bops



(b)堆内存使用百分比

图5 SPECjbb2005性能测试

5 相关工作及讨论

随着多核处理器的普及和众核处理器设计技术的不断发展,软件并行化重构将越来越得到人们的重视。国外一些著名大学和研究机构正在从事软件并行化重构方面的研究,Danny等[10]通过使用Java并行库来重构串行程序使之变为可重入的并行程序,通过将int类型转换为AtomicInteger和将HashMap转换为ConcurrentHashMap使数据访问是线程安全的。Franklin等[11]将Java程序中的内部类重构为lambda表达式,并且使用lambda表达式对关于集合迭代的循环进行了并行化重构。Frank领导的程序分析和转换研究小组设计了一个软件重构工具Relocker[12],Relocker提出了锁转换的算法,可以实现锁重构。Lock2Lock与Relocker的不同之处在于Lock2Lock主要关注字节码变换,使用Joeq和Javassit作为字节码分析和转换的工具,而Relocker则基于WALA编译器实现源对源的转换,二者的关注点不同,采用的技术也不同。英国St. Andrews大学的Hammond教授领导的研究小组提出了使用重构方法来产生并行程序的工具ParaPhrase[13],它通过一组形式化的模式重构规则增强并行程序设计的可编程性。FlexSync[14]是一个面向方面的同步库,支持多种同步控制方式(如锁、原子块操作和STM)的定制,FlexSync的不足之处在于对可重入锁的支持不足。Lock2Lock可以很好地处理从同步锁到可重入锁的重构,因此可以被认为是FlexSync的一个很好的补充。此外,FlexSync由于使用了STM,因此对于线程间通信操作还不能很好地处理,但Lock2Lock没有这方面的限制,可以很好地进行线程间通信操作的转换。AutoLocker[15]是一个锁的推测工具,可以实现对象访问前加锁和解锁操作。Lock2Lock与这两种锁的实现关注点是不同的。

在国内,许多大学和研究所很早就开始进行软件重构的研究工作,但软件并行化重构方面的研究还有待深入。文献[16]对面向并发的程序重构技术进行了研究,探讨了并行化重构技术和并行化软件内部重构的优化方法等。南京大学吕建教授等对并发对象的同步模型进行研究,提出一种并发面向对象广谱规约语言的同步模型-卫式路径结构,该结构可以支持同步代码的复用[17]。钱巨等[18]针对并发程序中同步粒度过粗的问题,通过将一个粗粒度的锁分解为细粒度的锁来逐步演化同步结构,提出了一种面向Java的自动锁分解重构方法,以此来提高程序的并行性。

结束语 本文将软件重构的方法应用于并行程序内部的锁机制上,针对Java锁机制中的同步锁、可重入锁和读写锁,提出了一种自动重构框架,该框架在中间表示的基础上通过字节码分析、验证和转换技术,可以实现同步锁到可重入锁或读写锁的转换。在红黑树、生产者消费者程序和SPECjbb2005 3个测试程序上进行了重构实验,结果表明Lock2Lock可以成功地对这几个程序进行重构,该工具可以减少程序员的工作量,并且可以帮助程序员了解程序使用某一种锁的性能。

参考文献

[1] Kiczales G, Lamping J, Mendhekar A, et al. Aspect-oriented programming[C]// Proceedings of European Conference on Object-Oriented Programming. Jyvaskyla, Finland, 1997: 241-268

[2] Barnes G. A method for implementing lock-free shared-data structures [C]// Proceedings of the fifth annual ACM symposium on Parallel algorithms and architectures. Velen, Germany, 1993: 261-270

[3] Shavit N, Touitou D. Software transactional memory[J]. Distributed Computing, 1997, 10(2): 99-116

[4] Whaley J. Joeq: A virtual machine and compiler infrastructure [J]. Science of Computer Programming, 2005, 57(3): 339-356

[5] Chiba S. Javassist—a reflection-based programming wizard for Java[C]// Proceedings of OOPSLA'98 Workshop on Reflective Programming in C++ and Java. Denver, Colorado, 1998: 92-115

[6] Oracle. Java API Specification [EB/OL]. (2011-11-23). http://download.oracle.com/javase/6/docs/api/java/util/concurrent/locks/package-summary.html

[7] Herlihy M, Luchangco V, Moir M. A flexible framework for implementing software transactional memory [J]. ACM Sigplan Notices, 2006, 11(2): 253-262

[8] Zhang Y, Zhang J, Zhang D. Implementing and testing Producer-consumer problem using aspect-oriented programming [C]// Proceedings of Fifth International Conference on Information Assurance and Security. Xi'an, China, 2009: 749-752

[9] Adamson A, Dagastine D, Sarne S. SPECjbb2005--A year in the life of a benchmark[C]// Proceedings of 2007 SPEC Benchmark Workshop. Austin, USA, 2007: 151-160

[10] Dig D, Marrero J, Ernst M D. Refactoring sequential Java code for concurrency via concurrent libraries [C]// Proceedings of IEEE 31st International Conference on Software Engineering. Vancouver, Canada, 2009: 397-407

[11] Franklin L, Gyori A, Lahoda J, et al. LAMBDAFICATOR: from imperative to functional programming through automated refac-

toring[C]//Proceedings of the 2013 International Conference on Software Engineering. San Francisco, USA, 2013:1287-1290

[12] Schafer M, Sridharan M, Dolby J, et al. Refactoring java programs for flexible locking[C]//Proceedings of 33rd International Conference on Software Engineering (ICSE). Hawaii, USA, 2011:71-80

[13] Brown C, Hammond K, Danelutto M, et al. Paraphrasing: Generating Parallel Programs Using Refactoring[J]. Formal Methods for Components and Objects, 2013, 10(8):237-256

[14] Zhang C. FlexSync: An aspect-oriented approach to Java synchronization[C]//Proceedings of the 31st International Conference on Software Engineering. Vancouver, Canada, 2009: 375-385

[15] McCloskey B, Zhou F, Gay D, et al. Autolocker: synchronization inference for atomic sections[J]. ACM SIGPLAN Notices, 2006, 23(2):346-358

[16] 赵思奇. 面向并发的程序重构技术研究[D]. 南京:东南大学, 2010
Zhao Si-qi. A novel approach of concurrency-oriented software refactoring[D]. Nanjing: Southeast University, 2010

[17] 吕建,杨大军,廖宇,等. 一种并发面向对象同步模型研究[J]. 软件学报, 2002, 13(1):71-80
Lv Jian, Yang Da-jun, Liao Yu, et al. Research on a Concurrent Object-Oriented Synchronization Model[J]. Journal of Software, 2002, 13(1):71-80

[18] 陶彬贤,张磊,钱巨. Java 程序自动锁分解重构[J]. 计算机科学与探索, 2013, 7(5):451-459
Tao Xian-bin, Zhang Lei, Qian Ju. Automated Split Lock Refactoring for Java Programs[J]. Journal of Frontiers of Computer Science and Technology, 2013, 7(5):451-459

(上接第 67 页)

最后,手动关闭工作节点的进程来引入节点失效的方式,以此来测试容错性能。BitDew-MapReduce 和 Tree-MapReduce 中设置的节点故障超时时间均为 30s。最开始采用了 40 个工作节点,每隔 10s 产生一个节点失效,失效的节点数分别为 5、10、15、20。在节点失效的情况下,作业完成时间的比较结果如图 6 所示。从图 6 中可看出,在 20 个节点失效的情况下,采用 Tree-MapReduce 比采用 Hadoop 时间开销减少了约 17.9%。

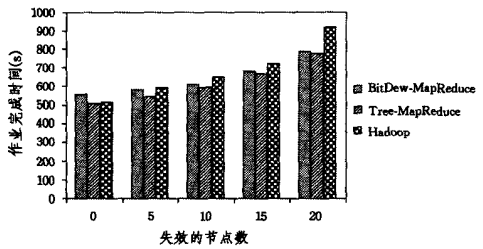


图 6 随着失效节点数增加的作业完成时间开销

结束语 本文提出一种基于树型结构的新型 MapReduce 并行模型。该模型以 P2P 的形式将大规模的桌面 PC 节点组织起来,模型的底层采用了 P2P-MPI 框架,采用基于消息传递的模式来实现 MapReduce 应用层。在 MapReduce 应用层的实现中,在 Map 阶段采用广播的形式来分发数据块,在 Reduce 阶段建立反向二叉树来实现有效的结果合并和化简。这种基于树型结构的 MapReduce 模型充分利用了 MPI 和 MapReduce 各自的优点,具有较好的可扩展性。在 P2P-MPI 框架的支撑上,由于克服了 MPI 的缺点,使得所提出的 MapReduce 模型具有较好的容错性能。性能比较的结果表明,基于树型结构的 MapReduce 并行模型在容错性能方面具有较优的性能,且系统简单,易于应用开发。该模型适合于利用 Internet 或 Intranet 环境下不可靠的桌面 PC 资源进行海量科学数据分析。

参 考 文 献

[1] Dean J, Ghemawat S. MapReduce: Simplified Data Processing on Large Clusters[J]. Communications of the ACM, 2008, 51(1): 107-113

[2] Anderson D P. BOINC: A System for Public-Resource Computing and Storage[C]//Proc. of the 5th International Workshop on Grid Computing (GRID 2004). 2004:4-10

[3] Cappello F, Djilali S, Fedak G, et al. Computing on Large-scale Distributed Systems: XtremWeb Architecture, Programming Models, Security, Tests and Convergence with Grid[J]. Future Generation Computer Systems, 2005, 21(3):417-437

[4] Litzkow M J, Livny M, Mutka M W. Condor-A Hunter of Idle Workstations[C]//Proc. of the 8th International Conference on Distributed Computing Systems (ICDCS 1988). 1988:104-111

[5] Lin H, Ma X, Feng W. Reliable MapReduce Computing on Opportunistic Resources[J]. Cluster Computing, 2012, 15(2):145-161

[6] Marozzo F, Talia D, Trunfio P. P2P-Mapreduce: parallel data processing in dynamic cloud environments[J]. Journal of Computer and System Sciences, 2012, 78(5):1382-1402

[7] Costa F, Silva J N, Veiga L, et al. Large-scale volunteer computing over the Internet[J]. Journal of Internet Services and Applications, 2012, 3(3):329- 346

[8] Tang B, Moca M, Chevalier S, et al. Towards mapreduce for desktop grid computing[C]//Proc. of the 5th International Conference on P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC 2010). 2010:193-200

[9] Lu L, Jin H, Shi X, et al. Assessing mapreduce for Internet computing: a comparison of Hadoop and BitDew-MapReduce[C]//Proc. of the 13th ACM/IEEE International Conference on Grid Computing (GRID 2012). 2012:76-84

[10] Genaud S, Rattanapoka C. P2P-MPI: A Peer-to-Peer Framework for Robust Execution of Message Passing Parallel Programs on Grids[J]. Journal of Grid Computing, 2009, 5(1):27-42

[11] Genaud S, Rattanapoka C. A Peer-to-Peer Framework for Message Passing Parallel Programs[M]//Xhafa F, eds. Parallel Programming, Models and Applications in Grid and P2P Systems, Advances in Parallel Computing. IOS Press, 2009:118-147

[12] Carpenter B, Getov V, Judd G, et al. MPJ: MPI-like message passing for Java[J]. Concurrency-Practice and Experience (CONCURRENCY), 2000, 12(11):1019-1038

[13] Fedak G, He H, Cappello F. BitDew: A Data Management and Distribution Service with Multi-protocol File Transfer and Metadata Abstraction[J]. Journal of Network and Computer Applications, 2009, 32(5):961-975