Java Packed Object Synchronization

by

Bing Yang

Master of Computer Science, Beihang University, 2010

A DISSERTATION SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

Doctor of Philosophy(PhD)

In the Graduate Academic Unit of Computer Science

Supervisor(s):	Kenneth B. Kent, PhD, Faculty of Computer Science
	Eric Aubanel, PhD, Faculty of Computer Science
Examining Board:	Gerhard Dueck, PhD, Faculty of Computer Science, Chair
	Wei Song, PhD, Faculty of Computer Science
	Mary Kaye, Faculty of Engineering
External Examiner:	Doug Lea, PhD, Computer Science Dept.,
	State University of New York at Oswego

This dissertation is accepted

Dean of Graduate Studies

THE UNIVERSITY OF NEW BRUNSWICK

May, 2017

©Bing Yang, 2017

Abstract

Concurrency and synchronization are fundamental issues critical for the design of applications that allow multiple threads to access the shared resources concurrently. The study of synchronization has grown in urgency with the proliferation of multi-core processors in recent years.

In this thesis, we proposed thread-safe and efficient approaches to synchronization based on the packed object data model, which were introduced in IBM's J9 Virtual Machine as an experimental enhancement in order to help users gain greater control over the object layout. This object data model organizes data in a multi-tier manner (like the nested C structs), in which the object's data is nested in its enclosing PackedObject instead of being pointed to by an object reference as in the traditional Java object data model. The data overlapping between different tiers of PackedObject poses new challenges for synchronization, which is out of the scope of the traditional Java synchronization mechanisms. In order to address the new synchronization issues and situations raised by the packed object data model, we proposed a few different approaches to synchronization of multi-tier PackedObjects

in multi-granularity locking. *PackedSync* is the first proposed approach for PackedObject synchronization that is based on a monitor without relying on the lock-word field. This approach gains better performance than the Java built-in synchronization approach in the coarse-grained locking. However it works only with two-tier PackedObjects and supports only the single-granularity locking. In order to address the multi-granularity locking on multi-tier PackedObjects, we proposed *MultitierPackedSync* based on an optimized concurrent linked-list that postpones the physical deletion of a node so as to reduce the overhead of unnecessary repetition of deletion and insertion operations. This approach improves the overall performance considerably comparing with *PackedSync*, the Java built-in synchronization and the *Lazy_MultitierPackedSync* that is based on the lazy-list algorithm without our optimization. In order to further improve the performance of multi-tier data synchronization, we implemented a concurrent multiway tree in the left-child right-sibling tree representation that allows multiple threads to in parallel manipulate the tree structure. We then proposed a multi-granularity locking scheme (Tree-based MultitierSync) based on the concurrent multiway tree, which represents the containing and ordering relationship among the data in different tiers of PackedObjects. Comparing to the List-based MultitierSync (MultitierPackedSync), the Tree-based MultitierSync significantly improves the overall performance in different workloads, and the higher workload, the better performance gains can be achieved according to our benchmarks.

Dedication

I dedicate this dissertation to my deceased father: Qihong Yang. His words of inspiration and encouragement in pursuit of excellence, still linger on.

Acknowledgements

It is a great pleasure to work with my supervisor Professor Kenneth, B. Kent and Professor Eric Aubanel. I consider myself lucky to be able to work under their dedicated supervision and guidance in pursuit of my PhD degree. I would not have been able to accomplish this dissertation and the related research work without their continuous support, encouragement and patience.

Besides my supervisors, I would like to thank the rest of my thesis committee: Prof. Gerhard W. Dueck and Prof. Andre Hinkenjann, for their insightful comments and questions. In addition, I would like to thank Stephen MacKay for his valuable feedback on my papers and thesis. My sincere thanks also go to Aaron Graham and other fellow researchers at the Centre for Advanced Studies-Atlantic (CASA) lab. They have been very helpful and supportive in completing my research work.

Furthermore, I would express my gratitude to Tobi Agila, Angela Lin, Daniel Heidinga and Ronald Servant from IBM Ottawa for their support in the research project. Last but not least, I would like to thank my parents: Qihong

Yang and Guoqun Tao for giving birth to me at the first place and supporting me spiritually throughout my PhD study.

Bing Yang

Table of Contents

Abstract **ii**

Dedication **iv**

Acknowledgments **v**

Table of Contents **x**

List of Tables **xi**

1 Introduction **1**

List of Figures **1**

2 Background **8**

2.1 Java and Packed Object Data Model 9

2.1.1 Packed Object Data Types and Layout 10

2.1.2 PackedObject Categories 14

2.1.3 PackedObject Header 15

2.2 Synchronization Mechanisms 16

2.2.1 Synchronization Mechanisms in Java 17
2.2.2 Synchronization Mechanisms in Other Languages . . . 21
2.3 Multi-granularity Locking 23
2.4 Concurrent Data Structures 25
2.4.1 Linked Lists 26
2.4.2 Trees 27

3 Monitor-based Synchronization for PackedObjects 32
3.1 Problem statement 33
3.2 Synchronization Scenarios 36
3.3 Technical Approach 43
3.3.1 Monitor Association 44
3.3.2 Packed Regions 47
3.3.3 Synchronization Interfaces 51
3.4 Evaluation and Analysis 58
3.5 Space Analysis 65

4 Multi-tier Data Synchronization based on A Concurrent Linked-list 67
4.1 Problem Statement 68
4.2 Technical Approach 72
4.2.1 Lazy Concurrent Linked-list Algorithm 72
4.2.2 Optimized Concurrent Linked-list Approach 75
4.2.2.1 Auxiliary Data Structures 75

4.2.2.2 Synchronization Interfaces 78
4.2.3 Algorithm Analysis 90
4.3 Evaluation and Analysis 94
4.4 Space Analysis 102

5 Multi-tier Data Synchronization based on A Concurrent Multiway Tree 104

5.1 Concurrent Multiway Tree Algorithm 106
5.1.1 A Multiway Tree 107
5.1.2 Algorithm 109
5.1.2.1 The *Search* and *SearchFrom* Methods 111
5.1.2.2 The Insert Method 114
5.1.2.3 The Delete Method 117
5.1.3 Algorithm Analysis 121
5.1.4 Correctness 123
5.1.4.1 Dead-lock freedom 123
5.1.4.2 Linearizability 124
5.1.5 Evaluation and Analysis 125
5.2 Multi-Granularity Locking Scheme based on the Concurrent Multiway Tree 131
5.2.1 Technical Approach 132
5.2.1.1 Auxiliary Data Structures 132
5.2.1.2 Synchronization Interfaces 136

5.2.2 Algorithm Analysis 149
5.2.3 Evaluation and Analysis 157
5.2.4 Space Analysis 164

6 Conclusion **166**

6.1 Future Directions 170

Bibliography **181**

Vita

List of Tables

3.1 Comparison of *PackedSync* and Synchronized Approach in Fine-grained Locking with Different Array Sizes (in microseconds) 64

List of Figures

1.1 A typical 3-tier PackedObject data structure 3

2.1 The Relationship between Java Heap and Native Memory . . . 9
2.2 Traditional Java Object Data Model 11
2.3 Packed Object Data Model 11
2.4 Memory Layout of Standard Java Object Line 14
2.5 Memory Layout of Java PackedObject Line 14
2.6 On-heap PackedObject, on-heap derived PackedObject and off-heap PackedObject 15
2.7 The Structure of a PackedObject Header 16

3.1 Multiple objects referring to the same underlying data 34
3.2 Memory layouts of a Packed array (on the left) and a standard Java array 35
3.3 Scenario 1 of On-heap PackedObjects 37
3.4 Scenario 2 of On-heap PackedObjects 39
3.5 Scenario 3 of On-heap PackedObjects 40
3.6 Scenario 1 of Off-heap PackedObjects 42

3.7 Memory Layout of a PackedInt Array 45
3.8 On-Heap Packed Region and Off-heap Packed Region 48
3.9 Irregular overlapping between different native Packed Regions 51
3.10 The Invoking Flow of VM Internal Functions PackedLock/-PackedUnlock 52
3.11 Flowchart of Function getPackedContainerInfo() 53
3.12 Flowchart of PackedLock() 54
3.13 Flowchart of PackedUnlock() 57
3.14 Time Overhead Comparison for Coarse-grained Locking 60
3.15 Throughput comparison for Coarse-grained Locking 61
3.16 Time Overhead Comparison for Fine-grained Locking 61
3.17 Throughput comparison for Fine-grained Locking 62

4.1 Memory layout of PackedObject A of Type ThreeTierClass . . 71
4.2 Insertion and Deletion of Entries of a Lazy-list 73
4.3 An on-heap three-tier PackedObject A and the associated auxiliary data structures 78
4.4 Five cases during traversal of the MonitorList 81
4.5 Locking Scenario of Case 3 83
4.6 Locking Scenario of Case 4 85
4.7 Locking Scenario of Case 5 87
4.8 Unlocking Scenario of Case 3 89
4.9 Time Expense Comparison in Coarse-grained Locking 96

4.10 Time Expense Comparison in Fine-grained Locking 97
4.11 Time expense of *MultitierPackedSync* with different thresholds for *PDCount* 98
4.12 Time expense comparison in multi-granularity locking 100
4.13 Time expense of *MultitierPackedSync* with different array size 101

5.1 Multi-tier data structure of 3-tier PackedObject A 105
5.2 A multiway tree represented in a left-child right-sibling tree . . 108
5.3 A Typical Left-Child Right-Sibling Tree 111
5.4 The *SearchFrom*() Method 113
5.5 The Search() Method 113
5.6 The Insert() Method 115
5.7 Simple Insertion 116
5.8 An Example of Complex Insertion 117
5.9 Complex Insertion 118
5.10 The Delete() Method 120
5.11 Deletion Scenario 122
5.12 Small workload in the search intensive mode 129
5.13 Large workload in the search intensive mode 129
5.14 Small workload in the update intensive mode 130
5.15 Large workload in the update intensive mode 130
5.16 A LCRS tree representation for the *PackedMonitorInfo* of the three-tier PackedObject 134

5.17 Flowchart of the Traversal of PMTree 138
5.18 Flowchart of PackedLock after the simple insertion 140
5.19 Flowchart of PackedLock in case 4 142
5.20 Flowchart of function TraverseAndSetLockState() 143
5.21 Flowchart of PackedLock after the complex insertion 145
5.22 Flowchart of PackedUnlock in Case 4 147
5.23 Flowchart of function TraverseAndClearLockState() 148
5.24 Static Tree Scenario 150
5.25 An example for case 1 of the growing tree scenario 151
5.26 An example for Case 2 of the growing tree scenario 153
5.27 An inconsistent locking State caused by wrong traversal order 156
5.28 Time expense comparison in a small workload (nTask =array-Size*nTier) 159
5.29 Throughput comparison in a small workload (nTask =array-Size*nTier) 159
5.30 Time expense comparison in a large workload (nTask =array-Size*nTier*10) 160
5.31 Throughput comparison in a large workload (nTask =array-Size*nTier*10) 160
5.32 The time expense comparison in different workload as the working thread number is 64 162
5.33 Throughput comparison in different workload as the working thread number is 64 162

Chapter 1

Introduction

The development of multi-core processors in recent years raises new challenges in the development of efficient approaches to allow concurrent threads to access shared resources [25]. With the growing prevalence of multi-core multi-processor systems, parallel programming and concurrency in applications are becoming increasingly important. Concurrency and synchronization are fundamental issues critical for the design of applications that allow multiple threads to access the shared resources concurrently. Concurrent access to a data structure shared among several threads must be synchronized in order to avoid interference between conflicting operations. Such concurrent data structures are widely used in the concurrent applications that fully exploit today's multiprocessor and multicore systems.

Java is a general-purpose, concurrent, object-oriented programming language [23] with language-level support for concurrent programming. Due to its

simplicity and platform independence, Java has been widely used in many applications.

Java applications run in an execution environment called the Java Virtual Machine (JVM) [44]. The JVM has a heap (the Java heap) shared among all JVM threads. The heap is the runtime data area from which memory for all class instances and arrays is allocated. The JVM itself is a native program written in a language (such as C) that consumes native resources, including native memory [39]. Java provides the built-in synchronization mechanism for communicating among threads based on monitor constructs. Each Java object is associated with a monitor and only one thread at a time may hold a lock on the monitor [44]. In addition to the Java built-in synchronization mechanism, Java also provides high-level concurrent features in the java.util.concurrent packages. However, Java synchronization mechanisms are based on the traditional Java object data model which features non-adjacent placement of object instances in the Java heap. The non-adjacent placement of associated Java object instances could slow down object data serialization and garbage collection (GC) [40]. The traditional Java object data model also wastes a great deal of memory on object headers and requires redundant data copying and marshalling when dealing with native data outside the Java heap [35].

In order to avoid the limitation of the traditional Java object data model, IBM's J9 Virtual Machine introduces the packed object data model as an experimental enhancement, which helps Java users gain greater control over

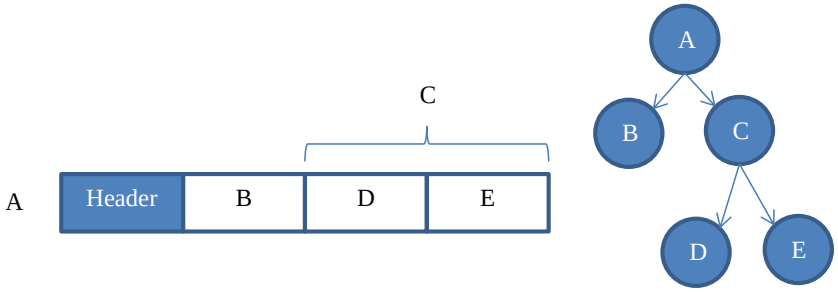


Figure 1.1: A typical 3-tier PackedObject data structure

the object layout in memory and avoid unnecessary data copying and marshalling. The packed object data model organizes object data in a multi-tier manner (like nested C structs) in which the object's data is nested in its enclosing PackedObject instead of being pointed to by an object reference, as in the traditional Java object data model. In the remainder of the thesis, we refer to the multi-level nested data structures as multi-tier data structures. Figure 1.1 shows a typical multi-tier PackedObject which contains a one-tier PackedObject *B* and a two-tier PackedObject *C* that contains two one-tier PackedObjects *D* and *E*. The binary tree on the right of Figure 1.1 shows the data containing relationship among the different PackedObjects.

Due to the multi-tier architecture of the packed object data model, the data overlapping between different tiers of PackedObject poses a new challenge for synchronization, which is out of the scope of the traditional Java synchronization approaches. In the packed object data model, the highest tier PackedObject is the container PackedObject, which is also the coarsest grained data. The lower tier PackedObject data is the finer grained data contained within

the upper tier PackedObject. In a concurrent context, multiple threads can access the data of different granularities in different tiers by referring to the derived PackedObject headers. Due to the data overlapping between different granularities in different tiers of the packedObject, multiple threads could refer to the same underlying data and update the data at the same time. In order to keep the data integrity of multi-tier PackedObjects, a new efficient synchronization approach supporting multi-granularity locking is necessary. Multi-granularity locking is a concept originated from database management systems [49] that organizes the collection of data in a hierarchical architecture. Multi-tier data structures in the packed object data model introduce new challenges, since multiple threads could arbitrarily access data in any granularity, and the hierarchical architecture representing the relationship of the data in different granularities can be dynamically changed. This is different from the database management system which usually has a fixed hierarchical data structure.

This thesis investigates the packed object data model and the traditional Java synchronization approaches and then first proposes a monitor-based synchronization approach. Unlike the traditional Java synchronization that relies on a lock-word field for synchronization, the monitor-based synchronization uses a linked-list to keep the associated monitors for the PackedObjects without relying on the lock-word field. Threads acquire the monitor by hashing the packed data or traversing the associated monitor list. However, this approach only works with two-tier PackedObjects in single-granularity locking.

In order to solve multi-granularity locking on the multi-tier PackedObjects, the thesis investigates the state-of-the-art concurrent linked-list algorithm and proposes another approach based on an optimized concurrent linked-list. This approach organizes the monitor information for the derived PackedObjects in a sorted linked-list shared among the threads. This approach allows multiple threads to concurrently manipulate the shared linked-list and acquire the target monitor for locking or unlocking operations. The linked-list is among the most fundamental data structures and has been used in many applications. Unlike tree structures, the linked-list grows linearly. As the linked-list of monitor information grows, the threads have to traverse more nodes to reach the target monitor information and acquire it for synchronization. Thus the overhead is increased. In order to further improve the performance of multi-tier data synchronization, we propose another approach based on a concurrent multiway tree which allows threads to traverse relatively fewer nodes to acquire their target monitor information. In order to implement the tree-based approach, we propose a concurrent multiway tree algorithm that applies the concurrent linked-list algorithm to the multiway tree in a left-child right-sibling tree representation [57] and deals with more complicated scenarios than linked-lists. Based on the concurrent multiway tree algorithm, we develop a tree-based approach to multi-tier data synchronization which provides a more efficient way for multi-granularity locking on the multi-tier PackedObjects.

In summary, this thesis makes the following contributions:

- Proposes a monitor-based synchronization approach for two-tier PackedObjects without relying on the lock-word as the traditional Java synchronization does. This work was published at CASCON 2015 [60].

- Develops a synchronization framework that allows synchronization on the native data between both the Java side and the native side.

- Proposes a multi-tier data synchronization approach based on an optimized concurrent linked list. This work was published at PPPJ 2016 [59].

- Proposes a concurrent multiway tree algorithm that allows multiple threads to concurrently manipulate on the left-child right-sibling tree representation. This work was published at CASCON 2016 [58].

- Proposes a more efficient multi-granularity locking scheme based on the concurrent multiway tree algorithm for synchronization of multi-tier PackedObjects.

The remainder of this thesis is structured as follows. Chapter 2 presents the background to our synchronization approaches for the multi-tier PackedObjects. Chapter 3 describes the first proposed monitor-based synchronization approach for two-tier PackedObjects and evaluation with analysis of the algorithms. Chapter 4 illustrates the multi-tier data synchronization approach based on an optimized concurrent linked-list and then analyzes and evaluates the algorithms. Chapter 5 presents both the concurrent multi-way tree algo-

rithm and the new efficient multi-tier data synchronization approach based on the concurrent multiway tree algorithm. The evaluation and analysis for both approaches are present in the chapter. We finally conclude in Chapter 6 and discuss our future work.

Chapter 2

Background

In this Chapter, we introduce the background of the packed object data model which poses new challenges for synchronization and concurrent programming, and the existing synchronization mechanisms in Java and other languages and the multi-granularity locking which is a significant mechanism for synchronization on multi-tier PackedObjects. In addition, we also introduce two fundamental concurrent data structures, linked lists and trees, and the existing approaches to build those concurrent data structures. We address the synchronization issues on the packed object data model by developing optimized approaches based on the fundamental concurrent data structures.

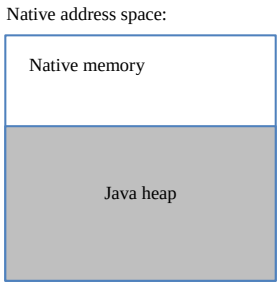


Figure 2.1: The Relationship between Java Heap and Native Memory

2.1 Java and Packed Object Data Model

Java is a general-purpose, concurrent, object-oriented programming language [23], which has gained popularity because of its simplicity and platform independence. Java applications run in an execution environment, called the Java Virtual Machine (JVM) [44]. The JVM has a heap (the Java heap) that is shared among all JVM threads. The heap is the runtime data area from which memory for all class instances and arrays is allocated. The JVM itself is a native program written in a language (such as C) that consumes native resources, including native memory [39]. As distinguished from the Java heap that a Java application uses, native memory is the memory available to the runtime process. Every virtualized resource including the Java heap and Java threads must be stored in the native memory, along with the data used by the virtual machine as it runs. Figure 2.1 shows the relationship between a Java heap and the native memory.

Packed Object support is an experimental enhancement [38] introduced in the

IBM J9 Virtual Machine as a way to help Java users gain greater control over the object layout in memory. This capability enables greater flexibility when dealing with non-Java memory structures; for example, serializing objects or when performing inter-language communication between Java code and other languages or environments.

The object data model is a significant part of the implementation of an object-oriented language and it determines how an object is represented in memory. In the traditional Java object data model, there are two types of data. One is the primitive type, which is a type predefined by the Java language and named by a reserved keyword. Another is the reference type, which is a pointer to the object or a special null reference to no object [44].

2.1.1 Packed Object Data Types and Layout

In the packed object data model, primitive data occupies the minimum amount of space making it more compact for PackedObjects containing primitive data fields. This model is in contrast to a standard Java object where all the primitive data fields are stored in either 4-byte (for byte, short, int, float, char and boolean data types) or 8-byte format (for long and double data types). This is wasteful, especially when there are large numbers of primitive data fields in a Java object. The packed object data model helps reduce the overall size of some patterns of Java objects and improve the access efficiency.

For example, if there is a class *ExampleClass* containing three primitive data



Figure 2.2: Traditional Java Object Data Model

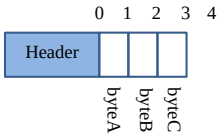


Figure 2.3: Packed Object Data Model

fields of byte data type, the PackedObject *ExampleClass* is defined as below (the annotation @Packed indicates the class is packed):

```
@Packed
public final class ExampleClass extends PackedObject
{
        public byte byteA;
        public byte byteB;
        public byte byteC;
}
```

In the standard Java object model, the memory layout of this class is shown as Figure 2.2:

As a packed type, however, the object looks like this:

When this type is packed, it occupies only three bytes that are required to represent the data. As with other objects, the JVM adds an object header, and a certain amount of padding to maintain object alignment, to any stand-

alone instance of *ExampleClass*.

As for the reference data types in the packed object data model, the data of the object pointed to by the reference field is not located in a separate object as for the traditional Java object model, but embedded in its enclosing object. The headers of the nested PackedObjects are generally omitted. Only the top level PackedObject, which is not contained by any other object, has an object header attached with its data fields upon the creation of the object. The object headers for the nested PackedObjects are created temporarily for reference upon access.

Consider a class that represents a line segment. The line segment is composed of two points, which indicate the coordinates of the two end points of the line segment. The PackedObjects *Point* and *Line* are defined as below:

```
//Point.java:
@Packed
public final class Point extends PackedObject
{
        public int x;
        public int y;
}
//Line.java:
@ImportPacked{"Point"}
@Packed
public final class Line extends PackedObject
{
        public Point  begin;
        public Point  end;
}
```

If *Line* was a standard Java type, the class would consist of two object references, which refer to two separate *Point* objects. Figure 2.4 displays the memory layout of *Line* object.

However, as a packed type, the representation of the PackedObject *Line* is far more compact. There is only one object instance for the PackedObject Line, which occupies only one consecutive space. These two fields are all of packed types, so their data is directly embedded into the PackedObject Line. The memory layout of the PackedObject *Line* is shown in Figure 2.5.

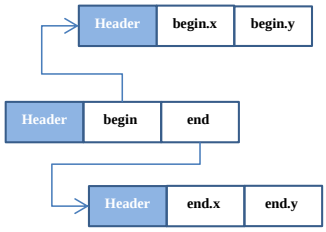


Figure 2.4: Memory Layout of Standard Java Object Line

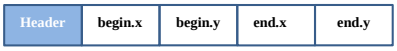


Figure 2.5: Memory Layout of Java PackedObject Line

2.1.2 PackedObject Categories

PackedObjects can represent data on the Java heap as well as data outside the Java heap [38]. Therefore, PackedObjects can be categorized into on-heap PackedObjects and off-heap PackedObjects. An off-heap PackedObject is composed of an object header containing a reference pointing to the data field located in native memory. An on-heap PackedObject's object header and data fields are both located on the Java heap. The nested PackedObject's data field is located in its enclosing PackedObject with the object header omitted. Whenever users access the nested object, a temporary object header is created for reference. Those nested objects are called on-heap derived PackedObjects. Comparatively the enclosing PackedObject on the Java heap is called on-heap non-derived PackedObject (or on-heap PackedObject). The three categories of PackedObjects are shown in Figure 2.6.

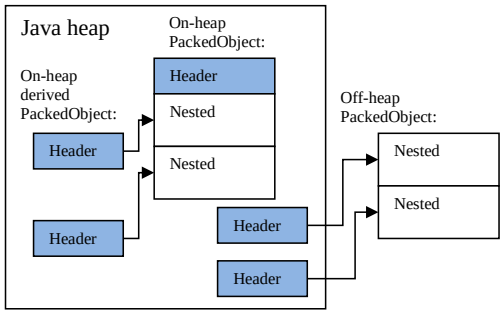


Figure 2.6: On-heap PackedObject, on-heap derived PackedObject and off-heap PackedObject

2.1.3 PackedObject Header

An object in an object-oriented language consists of the object data fields along with a header. The object header fields and their layout depend on the programming language, the virtual machine, etc. In the packed object data model, an object header is composed of three fields: *ClassPtr*, *Target* and *Offset*. As shown in Figure 2.7, *ClassPtr* is a pointer to a structure that contains the class type information of the PackedObject. Target and Offset determine the start address of the PackedObject data. For on-heap PackedObjects, Target is a pointer to the start address of the on-heap non-derived PackedObject's header, which is followed by the data fields; Offset indicates the distance from target to the data field of the PackedObject referenced by the PackedObject header; for off-heap PackedObjects, the Target field of the PackedObject header is always null; the Offset is the actual address of the data in native memory.

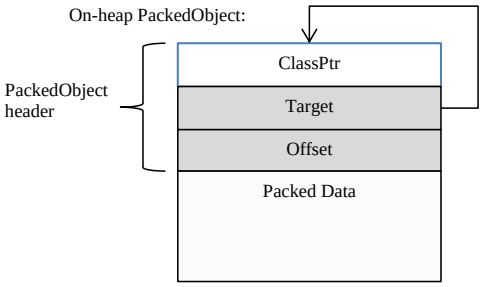


Figure 2.7: The Structure of a PackedObject Header

2.2 Synchronization Mechanisms

Synchronization is the mechanism that controls the order in which tasks execute. This can be done through cooperation or competition [10]. Competitive synchronization is a mechanism that controls access to shared resources between multiple tasks. Competitive synchronization is required when multiple tasks require access to some resource that cannot be accessed simultaneously. For cooperative synchronization, a task must wait for another task to complete some activities before it can continue its execution. A proper synchronization mechanism enforces exclusiveness and ordered access on the shared resources by different tasks.

Semaphores, monitors and message passing are the three main synchronization methods that provide mutual exclusive access to a shared resource. Message passing is often used in distributed systems, which is out of the scope of this thesis. We will focus on synchronization methods (semaphores and monitors) for non-distributed systems.

The semaphore was invented by Dijkstra in 1962 [18]. A semaphore is a data structure consisting of a counter and a queue for storing task descriptors. It is a simple mechanism used to provide limited access to a data structure by placing guards around the code that accesses the structure [10]. Semaphores have only two operations, wait and signal (originally called P and V respectively by Dijkstra). The wait operation blocks a process until the semaphore is open and the signal operation allows another process to enter. The value of the counter represents the number of resources. Semaphores are called binary semaphores if the value of their counters is restricted to 0 and 1, while semaphores that allow an arbitrary resource count are called counting semaphores.

Monitors were first proposed by Brinch Hansen [28] and later refined by Hoare [33]. A monitor is a programming language construct that controls access to shared data. It is a high-level synchronization construct that encapsulates shared data, procedures that operate on the shared data and synchronization between concurrent procedure invocations. A monitor guarantees mutual exclusive access by allowing only one monitor procedure to be executed at a given time. Calls to monitor procedures are implicitly queued if the monitor is busy at the time of the call.

2.2.1 Synchronization Mechanisms in Java

Java is one of many programming languages that have built-in support for concurrent programming. In Java, a unique monitor is associated with ev-

ery object that has a synchronized method in order to accommodate multi-threaded programs. Java synchronization mechanisms complete complicated synchronization operations with the support of the monitor constructs. Much research has been done in the monitor-based synchronization in Java. In order to build an effective synchronization mechanism, time and space efficiency are usually the primary concerns of most research in this field. Krall and Probst [42] proposed a monitor-based synchronization mechanism in their CACAO JIT compiler. The approach associates an object with a monitor via a hash function and thus saves per-object cost and reduces the space overhead for each object. However, most other monitor-based synchronization mechanisms put more emphasis on time efficiency over space efficiency. David Dice [17] proposed a relaxed-lock protocol that uses a lock-word field in the object header to support Java monitor semantics. Apart from that, it also uses a VM internal structure *monRec* to keep track of a queue of monitor information associated with different threads. The low-order bits of the lock-word field are used to distinguish different locking states as well. Bacon et al [4] defined a thin-lock which reserves only 24 bits of the header of each object via compression techniques for the use of synchronization. The 24 bits are used to represent either a thin-lock which is not subject to contention and not in support of wait-notify-notifyAll operations or a reference to a fat lock which is a heavy-weight structure composed of multiple words. The thin-lock was built upon the assumption that in most cases threads are only locking the object that is not locked by other threads. If contention

occurs, the thread needs to enter a spin-lock loop and keeps waiting until the other thread releases the lock. Onodera and Kawachiya [52] later proposed a tasuki-lock, based upon the thin-lock. The tasuki-lock simplified the thin-lock by extending the lock field to a full word instead of 24 bits and it used an extra bit from the other word in the object header to solve the flat lock contention. The extra bit represents two different locking modes: a flat lock mode and an inflated lock mode. In the case of contention, the flat lock is inflated to a heavy-weight monitor. If there are no threads waiting on the monitor, the inflated lock will be deflated to a flat lock. Compared with a thin-lock, the tasuki-lock eliminated busy-waiting while inflating the monitor and implemented a transition mode for deflation. Agesen et al [2] proposed a more space-efficient meta-lock that uses only 2 bits in the object header to represent locking states. However, the other 30 bits in the lock field will be used as a reference to a lockRecord if contention occurs. The lockRecord contains a queue field for linking the lock records of all threads that synchronize on the given object, and a free-list field for linking lock records when they are not in use. Yang et al [61] proposed a light-weight monitor which embedded a 32-bit lock-word in each object and managed the lock queue and wait set through a hash table in case of lock contention and conditional synchronization. Kawachiya et al [41] presented a new algorithm called lock reservation, which exploits the thread locality of Java locks. The key idea is to allow locks to be reserved for threads. When a thread attempts to acquire a lock, it can do so without any atomic operation if the lock is

reserved for the thread, thus the locking overhead is reduced. Dombrowski presented a dynamic monitor allocation approach [19], in which the JVM is specified to not assign a lock-word to each object upon instantiation and let it dynamically grow a lock-word for the object to be locked by appending the lock-word at the end of the object when the object is moved during garbage collection [40].

Apart from the Java built-in support for synchronization, Java extended the usual intrinsic locks by introducing the JUC locks as a part of the java.util.concurrent package [37] since Java 1.5. JUC locks provide more extensive locking operations than can be obtained using synchronized methods and blocks. They have quite different properties (e.g. ReadWriteLock, ReentrantLock) and allow more flexible structuring. The use of synchronized methods or statements provides access to an implicit monitor associated with every object, but forces all lock acquisition and release to occur in a block-structured way, while the JUC lock mechanism allows a lock to be acquired and released in different scopes and allows multiple locks to be acquired and released in any order. JUC locks also provide additional functionality over the monitor-based synchronization by providing a non-blocking attempt to acquire a lock (*tryLock()* can be set to wait a certain period of time for the lock), and an attempt to acquire the lock that can be interrupted(*lockInterruptibly()*). JUC lock also provides fairness by allowing the longest waiting thread to get the lock first.

2.2.2 Synchronization Mechanisms in Other Languages

Most advanced programming languages have support for multithreaded programming at the library or language level. c# is one of the advanced programming languages that have language level support for multithreading. The keyword *lock* is used in c# for synchronization, as opposed to the keyword *synchronized* used in Java. However, the keyword *lock* has the same semantics as *synchronized* except that c# does not turn a whole method into a critical section, instead the explicit *lock(obj)* construct must be used [45]. In addition, the c# keyword *lock* is just a shorthand for using the synchronization features defined by the Monitor class, which is defined in the System.Threading namespace [51]. The *Monitor* class defines several methods that control or manage synchronization, such as *Monitor.Enter()* and *Monitor.Exit()*. *TryEnter()* is another method provided by class Monitor that has the same semantics as the method *TryLock()* of JUC locks. In addition, the class Monitor also defines methods: *Wait()*, *Pulse()* and *PulseAll()* for inter-thread communication. Besides the lock feature and class *Monitor*, c# defines other synchronization objects that offer more functionality for synchronization, such as *Mutex* and *Semaphore*. Both of them can be created and known system-wide, therefore *Mutex* and *Semaphore* objects can be used for not only process-local synchronization, but also inter-process synchronization. Compared to the *lock* keyword, the *Mutex* construct in c# is a heavyweight implementation with a high overhead, as it is designed to work across multiple processes. Acquiring and releasing an uncontended *Mutex*

takes a few microseconds about 50 times slower than a lock [3].

Unlike Java and c#, C++ did not contain built-in support for multi-threaded applications until the C++11 standard, which provides complete built-in support for threads, mutual exclusion, condition variables, and futures [36]. Before C++11 standards, C++ relies on libraries and the operating system to provide such functionality [9]. On Unix-like operating systems, POSIX threads are widely used to provide multithreaded programming support. The Pthread library provides mutex constructs as a means of implementing thread synchronization. The Pthread mutex construct is available in the Linux Kernel from version2.6.x and above. It is implemented by using Fast Userlevel locking (*Futex*) [34]. A futex consists of a shared variable in user space indicating the status of the lock and an associated waiting queue in kernel space. In the uncontended case, acquiring or releasing a futex involves only atomic operations on its lock status word in user space. In the contended case, a system call into the kernel is required.

Ada95 is another programming language that supports synchronization. It is a revision of the Ada language [45, 6, 12] which was originally developed around 1980 for use in US defense systems. Ada has an integrated notion of processes called tasks. In the Ada language, the data shared among tasks has to be implemented by server tasks that other tasks have to communicate with in order to read or write the common data. A monitor-like construct in Ada can be accessed only through a number of operations. The operations are executed with mutual exclusion. The operations are divided into entries

and procedures/functions. Each entry has an associated waiting queue and a Boolean guard. An entry is open when the associated guard is true. A call of an entry will first await the critical region being free and then reserve it. Then the guard of the entry is checked and if not true, the calling tasks are put on the waiting queue of the entry (releasing the region).

2.3 Multi-granularity Locking

A lock is a synchronization mechanism that enforces mutual exclusive access to a shared resource in a concurrent programming environment. Semaphores and monitors are the two most common synchronization primitives that support both competition and cooperation synchronization.

Misuse of locks can result in deadlock or livelock problems. Lock operations usually involve overhead in terms of the CPU time for initializing and destroying locks and the memory space for allocation of locks. Lock contention occurs when one process or thread attempts to acquire a lock held by another process or thread. High lock contention causes the degradation of overall performance. Granularity is an important property of locks that measures the amount of data the lock is protecting. A coarse granularity of locks requires fewer locking operations and thus has less locking overhead but results in higher probability of lock contention. Comparatively, a fine granularity of locks reduces the possibility of lock contention but results in higher lock overhead. Therefore, different granularity of locks involves a tradeoff

between the concurrency and locking overhead [16].

Multi-granularity locking on Relational Databases (RDBs) as well as Object-Oriented Databases (OODBs) is a well-established research area [43]. A database is an organized collection of data, which is usually organized hierarchically into fields, records and files [21]. A field is the minimum unit of data that can be locked. A record consists of a set of fields; and a file is a collection of records. Granularity is defined as the size of lockable objects. Multiple granularity locking is basically an approach to exploit the hierarchical nature of the containing relationship among different objects in the database. The lock manager in Database Management Systems (DBMS) does not detect the conflict between locks at different granularities. Multiple granularity locking helps to cope with this problem. In DBMS, transactions set ordinary locks (such as a shared lock (S lock) or an exclusive lock (X lock)) at a fine granularity and intention locks (which are composed of an intention shared lock (IS lock) and an intention exclusive lock (IX lock)) at a coarse granularity [26]. The coarse granularity locks warn other transactions locking at the coarse granularity about potential conflicts with fine-grained locks.

The packed object data model organizes data in a multi-tier manner that is similar to the hierarchical structure of the general DBMS. However, the multi-granularity locking protocol used in DBMS is based on a fixed hierarchical structure which is different from the dynamically growing hierarchical tree that represents the relationship of different tiers of data in the packed

object data model. The dynamically growing tree poses a new challenge for multi-granularity locking of multi-tier data. In this thesis, we study the packed object data model and use it as a user case for the multi-tier data structures and develop a new effective multi-granularity locking scheme.

2.4 Concurrent Data Structures

Concurrency and synchronization are fundamental issues critical for the design of applications that allow concurrent threads to access shared resources. Concurrent access to a data structure shared among several threads must be synchronized in order to avoid interference between conflicting operations. Such concurrent data structures are massively used in the concurrent applications that fully exploit today's multiprocessor and multicore systems.

Concurrent data structures are far more difficult to design than sequential ones because threads executing concurrently may interleave their steps in many ways, each with a different and potentially unexpected outcome [47]. In this section, we concentrate on the implementation of concurrent data structures that are used to implement sets. Consider implementations of concurrent search structures supporting insert, delete and search operations. If these operations deal only with a key value, then the resulting data structure is a set.

2.4.1 Linked Lists

Linked lists are the fundamental building blocks for concurrent data structures and as the basis for many types of search and dictionary data types [30]. Apart from globally locking the linked-list to prevent concurrent manipulation, the most popular approach to concurrent lock-based linked lists is hand-over-hand locking (also called lock coupling) [5]. This is the earliest lock-based concurrent linked-list approach invented by Bayer and Schkolnick. In this approach, each node has an associated lock. A thread traversing the linked list releases a node's lock only after acquiring the lock of the next node in the list, thus preventing overtaking, which may cause unnoticed removal of a node. This approach gains better concurrency than the global lock by the reduced lock granularity but successive locks make it undesirable and block other threads from proceeding during traversal of the list. Valois [56] suggested the first effective non-blocking implementation of concurrent linked lists based on the Compare-And-Swap (CAS) synchronization primitive. It uses a special auxiliary node in between each regular node to prevent undesirable phenomena while insertion and deletion are interleaving. Harris [29] proposed another efficient lock-free approach which uses a special deleted bit that is accessed automatically with node pointers in order to signify that the node has been logically deleted. M. Michael [46] later presented another lock-free list based on Harris' linked-list algorithms and the algorithm is used in Java Concurrency Package of JDK 1.6.0. Herlihy and Shavit proposed a lock-based approach optimistic-list [31] that reduced the locking overhead

by allowing searching the list without acquiring locks until the target node is found. Verification of the nodes is needed before executing insertion or deletion operations. Heller et al [30] proposed the lazy-list approach, which further improved the optimistic-list approach by assimilating the idea from Harris' approach [29]. Instead of stealing a bit from the reference, the lazy-list approach uses a marked field to indicate logical deletion and marks the node before physically deleting it. In our thesis, we optimize the lazy-list algorithm and implement a multi-tier data synchronization approach based on the optimized concurrent linked list. In addition, our lock-based concurrent multiway tree also uses the idea of a lazy-list but deals with more complicated scenarios for a multiway tree structure, which is described in more detail (see Figure 5.2) in Chapter 5.

2.4.2 Trees

A tree is another fundamental building block for concurrent data structures. It is a data structure represented as a rooted tree directed toward its leaves [57]. A tree can be implemented in a way that each node has pointers to each child of the node plus its data field. However, since the number of children per node can vary greatly and is not known in advance, it might be infeasible to directly link all children in one data structure. One of the effective ways to represent a general tree structure is to keep the children of each node in a linked list of tree nodes. Each node has a pointer to its left-most child and a pointer to its right sibling. This is called the left-child

right-sibling representation [57] of a general tree (also called multi-way tree) on which our multi-granularity locking scheme for multi-tier data is based. This representation of multiway trees will be described in more detail in Chapter 5.

If any node of the tree has no more than two children, the tree is called a binary tree. The binary tree is among the most frequently used data structure in computer applications. Much of the research on tree structures has been focused on the binary search tree. Binary search trees can be classified as either internal or external. Internal trees store a key-value association at every node, while external trees only store values in leaf nodes. A concurrent implementation of any search tree can be achieved by protecting it using a single exclusive lock [47]. Concurrency can be improved by using a reader-writer lock to allow all read-only (search) operations to execute concurrently with each other while holding the lock in shared mode, while update (insert or delete) operations exclude all other operations by acquiring the lock in exclusive mode. A fine-grained locking strategy can further improve the concurrency by using one lock per node rather than a single lock for the entire tree. Bronson et al. [7] proposed a lock-based concurrent relaxed balanced AVL tree algorithm based on optimistic techniques adapted from transactional memory and using hand-over-hand locking [5] for the validation process while searching and navigating the binary tree. An AVL tree [1] is a self-balancing binary search tree in which the heights of the left and right child branches of a node differ by no more than one. If an insertion or

deletion on the tree causes this balance condition to be violated, then one or more rotations are performed to restore the AVL invariant. The process of restoring the tree invariant becomes a bottleneck for concurrent balanced tree implementations because rotation may require locks to guarantee the atomicity of their change and no other mutations affect the balance condition of any node. Drachsler et al. [20] proposed a lock-based internal binary search tree in which each node maintains pointers to its logical predecessor and successor nodes (based on the key order), in addition to maintaining pointers to its left and right children in the tree. Modify operations (insertion and deletion) update this logical information each time they add or remove keys from the tree. Thus the search operations can traverse the logical chain of predecessor and a successor pointer to find the key if it has been moved due to the possible rotation for re-balancing. Natarajan and Mittal [50] present a lock-free algorithm for the binary search tree by using CAS (compare-and-swap) and BTS (bit-test-and-set) atomic instructions. This algorithm tries to reduce the contention by marking edges instead of nodes for deletion and allowing multiple keys to be deleted at once. Ramachandran and Mittal [53] proposed a lock-based algorithm for concurrent manipulation of an unbalanced binary search tree which uses an internal representation of the binary search tree and operates also on edges instead of nodes for minimizing the contention window of a write operation. Brown et al. [8] proposed a general technique for concurrent search trees by using multi-word generalizations of the standard LL (load-linked), SC (store-conditional) and VL

(validate) primitives. An LL instruction reads the contents of a memory location. An SC instruction updates the contents of a memory location with a new value provided the location has not been modified since an LL instruction was last performed on it. A VL instruction returns true if a memory location has not been modified since an LL instruction was last performed on it. Crain et al. gave a different implementation using lock-based STM (Software Transactional Memory) [14] and locks [15], in which all deletions are done by marking the node containing the key. Physical removal of nodes and rotations are performed by a separate thread.

Compared with concurrent binary search trees, there is very little research on multiway search trees that are based on a general tree data structure. A multiway tree is a tree in which a node can have more than two children. If the nodes of a multiway tree hold 1 to m-1 distinct keys, the multiway tree is called a m-way search tree. Spiegel and Reynolds [55] presented a lock-free implementation of a multiway search tree which is a redesign of the skip tree. The lock-free skip tree has relaxed structural properties that allow atomic operations to modify the tree without invalidating the consistency of the data structure by supporting lock-free add and remove operations and wait-free contains operations.

The B-tree is another generalized binary search tree, in which any node can have more than two children, and the sizes of nodes are variable [13]. The keys and data of a B-tree can be stored in both internal and leaf nodes, whereas in a B+ tree [48] (which is a variation of B-tree), the data and keys are only

stored in the leaf nodes. Graefe proposed three designs of hierarchical locking that suit B-tree indices [24], in which he explored techniques permitting introduction and removal of levels in a lock hierarchy on-demand and without disrupting transaction or query processing. Chen and Wang proposed two algorithms to support fine-grained range locks for transactional virtual memory [11]. In one of the two algorithms, a few doubly linked lists are used to detect range lock conflicts among different transactions. This algorithm could incur large overhead when the number of range locks is large. Another more efficient algorithm uses a B+ tree and parent tree structures to help locate the conflicting range locks. It improves the performance by checking only the related ranges instead of all ranges.

In this thesis, our approaches to multi-tier data synchronization are also built upon linked-lists and multiway trees. We optimize a lock-based concurrent linked-list algorithm and apply it to PackedObject synchronization, and we also define a new ordered set based on the left-child right-sibling representation of the multiway tree, on which we build a multi-granularity locking scheme for multi-tier PackedObject synchronization.

Chapter 3

Monitor-based Synchronization for PackedObjects

PackedObject is an experimental enhancement introduced in IBM's J9 virtual machine. PackedObject support helps Java users gain great control over the object layout in memory, reduce the memory footprint and avoid unnecessary data copying and marshalling. However, unlike the traditional Java object model, the packed object data model removes unnecessary fields in the object header including the lock-word field used in the traditional Java synchronization mechanisms. In addition, PackedObjects can be used to represent data outside the Java heap thus can be manipulated by both Java and native code. This creates a need to build new efficient synchronization approaches for the packed object data model. In this chapter, we proposed a monitor-based synchronization approach for the packed object data model

without relying on the lock-word field and address the distinctive synchronization issues with shared memory and data structures that are introduced with the packed object data model.

3.1 Problem statement

Due to the significant change of the object data model, the traditional Java built-in synchronization mechanism does not apply to PackedObject synchronization which has to deal with new and more complicated situations introduced by the packed object data model. As described in Section 2.1, in the packed object data model, object headers for nested PackedObjects are created upon access. Multiple accesses to the same nested field of a PackedObject can cause the creation of multiple PackedObject headers referring to the same packed data. This feature of the packed object data model makes it impossible to use the traditional Java built-in synchronization mechanism for PackedObject synchronization. According to the traditional Java built-in synchronization mechanism, each object is associated with a monitor [44]. Multiple accesses to the same PackedObject result in the creation of multiple derived PackedObject headers which are observed as different class instances in the Java object data model, therefore, multiple threads accessing the same packed data through different object headers can end up with entering different monitors and fail to synchronize on the same packed data with the traditional Java built-in synchronization mechanism.

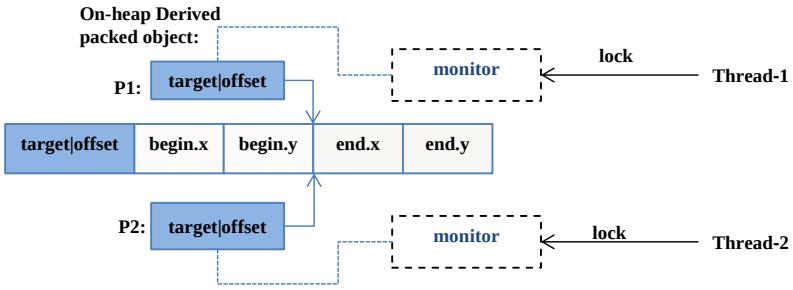


Figure 3.1: Multiple objects referring to the same underlying data

Take the class *Line* defined in Section 2.1 as an example. The line segment is composed of two points which indicate the coordinates of the two end points of the line. If two threads attempt to access *Line*.begin, two distinct PackedObject headers will be created for references of each thread. Figure 3.1 shows the layout of the PackedObject *Line*. Based on the standard Java object data model, these two packedObject headers are two distinct object instances that are associated with two distinct monitors. However, since they are referring to the same piece of data on the same PackedObject *Line*, the standard Java synchronization would fail to keep the consistency of the same data.

In addition, due to the modification of the object data model, synchronization on PackedObjects could have different semantics from the standard Java synchronization in the case of locking on a container PackedObject and its nested fields. Take an array of objects for example, the memory layouts of a packed array and a standard Java array are shown in Figure 3.2.

For the traditional Java built-in synchronization mechanism, locking on the

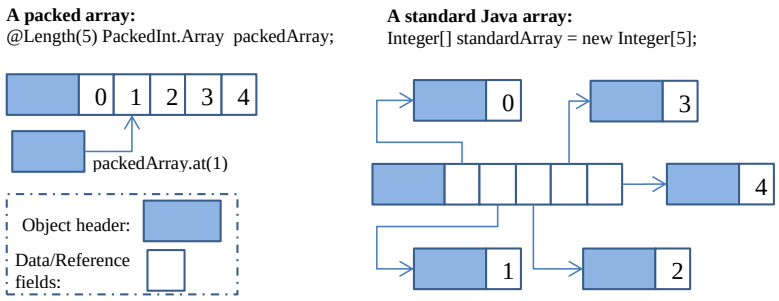


Figure 3.2: Memory layouts of a Packed array (on the left) and a standard Java array

standardArray has no impact on locking on any element of the array, since different elements of the *standardArray* are different objects associated with different monitors. While for PackedObject synchronization, locking on the *packedArray* should be able to prohibit other threads from locking on any element of the *packedArray* because the data of the element resides on the container PackedObject *packedArray*. Likewise, locking on the elements of the *packedArray* should also prevent other threads from locking on the entire *packedArray* because part of the *packedArray* data is currently locked by a thread. This is another issue that is faced by the packed object data model and has never been addressed in Java synchronization mechanisms. For the packed object data model, locking at different granularities (i.e. locking on the container PackedObject and the nested PackedObjects) can incur conflicts. However, for the traditional Java object model, the coarse-level object and the fine-level objects pointed to by references within the coarse-level object are independent from each other. Locking on standard Java objects

in different granularities does not incur conflicts. This chapter proposes a monitor-based synchronization approach to solve the locking conflicts between the container and nested PackedObjects.

As multiple derived PackedObject headers could refer to the same data, it makes no sense to keep the lock-word field in the PackedObject header. Otherwise, lock-words in different PackedObject headers keeping the same copy of the unique lock-word for the particular data (such as *Line.begin* in Figure 3.1) would require extra synchronization. Therefore, in the packed object data model, the lock-word field is removed from the PackedObject header. In this Chapter, we present a lock-word free approach for PackedObject synchronization and tackle the synchronization issues of the packed object data model that have never been addressed by other Java synchronization mechanisms.

3.2 Synchronization Scenarios

Before presenting the monitor-based approach for PackedObject synchronization, we go through all the possible scenarios that might happen and the possible challenges faced with the synchronization of PackedObjects. As illustrated in Chapter 2, PackedObjects are categorized as: on-heap PackedObjects, on-heap derived PackedObjects and off-heap PackedObjects according to the packed object data model. Packed arrays represent collections of PackedObjects which could be located either on-heap or off-heap. Since op-

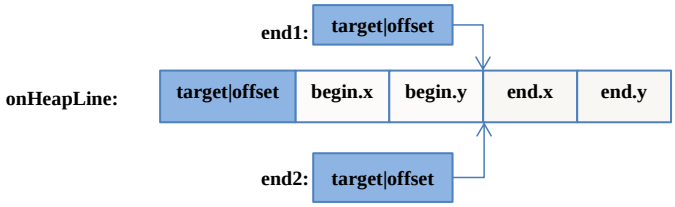


Figure 3.3: Scenario 1 of On-heap PackedObjects

erations on PackedObjects could also apply to packed arrays we will take PackedObjects as an example and go through different scenarios of on-heap and off-heap PackedObjects respectively.

Take the PackedObject Line defined in Section 2.1 as an example. Assume packedLock and packedUnlock are the names of synchronization interfaces for locking and unlocking operations on PackedObjects. These two methods are defined as native methods of the class PackedObject.

On-heap PackedObjects:

Scenario 1:

```
Line onHeapLine = PackedObject.newPackedObject(Line.
    class);
onHeapLine.initialize(...);
//Thread-1:
Point end1= onHeapLine.end;
end1.packedlock();
end1.x++;
end1.packedUnlock();
//Thread-2:
Point end2= onHeapLine.end;
end2.packedlock();
end2.x++;
end2.packedUnlock();
```

As shown above, two different threads are locking on the same nested Packed-Object onHeapLine.end. Their operations should proceed in sequential order. For example, if onHeapLine.end.x's initial value is 0, after these two threads finish their operations, onHeapBox.end.x's value should equal 2.

Scenario 2:

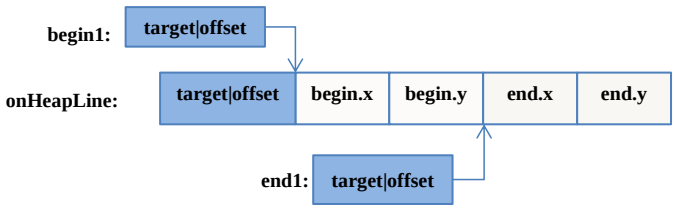


Figure 3.4: Scenario 2 of On-heap PackedObjects

```
Line onHeapLine = PackedObject.newPackedObject(Line.
    class);
onHeapLine.initialize(...);
//Thread-1:
Point begin1= onHeapLine.begin;
begin1.packedlock();
begin1.x++;
begin1.packedUnlock();
//Thread-2:
Point end1= onHeapLine.end;
end1.packedlock();
end1.x++;
end1.packedUnlock();
```

As shown above, multiple threads are locking on different nested PackedObjects *begin1* and *end1*. Since they are not sharing the same segment of data, their operations should be able to proceed in parallel. In other words, because these two threads are updating different data fields of the PackedObject on-

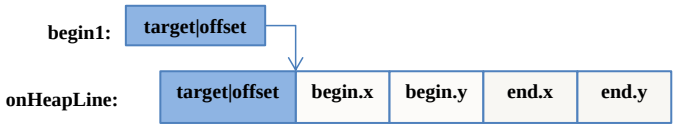


Figure 3.5: Scenario 3 of On-heap PackedObjects

HeapLine, their operations could be executed concurrently without having impact on each other.

Scenario 3:

```
Line onHeapLine = PackedObject.newPackedObject(Line.
    class);
onHeapLine.initialize(...);
//Thread-1:
Point begin1= onHeapLine.begin;
begin1.packedlock();
begin1.x++;
begin1.packedUnlock();
//Thread-2:
onHeapLine.packedLock();
onHeapLine.end.x++;
onHeapLine.packedUnlock();
```

In this scenario, thread-1 is locking on the nested object onHeapLine.begin, while thread-2 is locking on the container Packed-Object onHeapLine. In

this case, the data areas that these two threads are trying to update overlap. Therefore, once thread-1 locks on *onHeapLine*.begin, the nested field origin of *onHeapLine* is exclusively owned by thread-1, while thread-2 trying to lock on the container *onHeapLine* should not be able to execute its update on the container PackedObject until thread-1 finishes its update on the nested field and releases the lock. Likewise, if thread-2 locks on the container PackedObject *onHeapLine* first, it exclusively owns the container PackedObject. Thread-1 trying to update the nested field of *onHeapLine* should not be able to execute its update until after thread-2 finishes its update on the container PackedObject and releases the lock.

Off-heap PackedObjects:

All the synchronization scenarios for on-heap PackedObjects addressed above can also be applied to off-heap PackedObjects. We will consider a different scenario for off-heap PackedObjects as below:
Since the native code may arbitrarily cast the off-heap PackedObject to different packed objects or arrays or wrap multiple adjacent off-heap PackedObjects together as one PackedObject, all these operations and different orders of locking operations may lead to different situations.

Scenario 1:

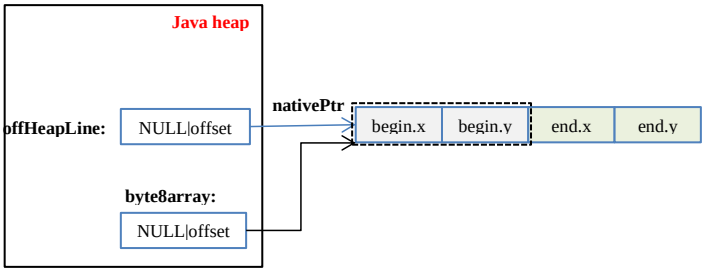


Figure 3.6: Scenario 1 of Off-heap PackedObjects

```
//some struct in C memory, sizeof()=16
long nativePtr= ...;
Line offHeapLine = PackedObject.newNativePackedObject(
    Line.class,nativePtr);
//cast 8 bytes of the native data into a PackedByte
    array with size of 8.
PackedByte.Array  byte8array = PackedArray.
    newNativeArray(PackedByte.class, 8, nativePtr);
//Thread-1:
offHeapLine.packedLock();
offHeapLine.begin.x++;
offHeapLine.begin.y++;
offHeapLine.packedUnlock();
//Thread-2:
byte8array.packedLock();
byte8array.at(index).copyFrom(value);
byte8array.packedUnlock();
```

In this scenario, a partition of native PackedObject *offHeapLine* is cast to an 8 element PackedByte array byte8array, which is now a nested PackedObject derived from *offHeapLine*. Two threads locking on *offHeapLine* and byte8array enter the same situation as addressed in Scenario 3 for on-heap PackedObjects, in which two different threads are locking on the container PackedObject and a nested PackedObject respectively. When thread-1 locks on *offHeapLine*, the off-heap container PackedObject *offHeapLine* is exclusively owned by thread-1, therefore, thread-2 tries to lock on the nested off-heap packed array byte8array whose data is contained by *offHeapLine* is not able to execute its update until after thread-1 finishes its update on the container PackedObject and releases the lock. Likewise, if thread-2 locks on the nested off-heap packed array byte8array first, it exclusively owns the nested packed array. Thread-1 tries to update the container PackedObject *offHeapLine* but should not be able to execute its update until after thread-2 finishes its update on the nested field and releases the lock.

3.3 Technical Approach

With the aim of coping with the different scenarios and situations faced by the packed object data model, we develop a synchronization approach that is lock-word free and capable of addressing the issues and limitations for PackedObjects as addressed in Section 3.1. This approach is monitor-based but without relying on the Lock-word field and synchronized keyword that

traditional Java built-in synchronization mechanisms are based upon. The synchronization of PackedObjects will comply with the scenarios addressed in Section 3.2.

3.3.1 Monitor Association

Java threads synchronize via monitors. Each java object has a monitor associated with it, and only one thread at a time may hold the lock associated with a monitor [44]. In order to synchronize on PackedObjects, there should be an approach to associate each synchronized PackedObject with the correct monitor. While Java objects can be uniquely identified by the object header address (which is the object reference value), PackedObjects cannot be identified in the same way because different PackedObject references could potentially refer to the same underlying data (see Figure 3.1). Thus PackedObjects differ from each other by their actual data addresses instead of the object references. In our approach we utilize the PackedObject's hash code that is calculated based upon the actual address of the packed data to associate a PackedObject with a monitor, allowing PackedObjects to be mapped to the monitor hash table via hashing their packed data addresses.

However, we should not always associate the PackedObject with its monitor by hashing the address of the packed data. In some cases different PackedObjects may share the same monitor even though they are referring to different addresses of packed data. Take a packed array for example. Figure 3.7 shows the memory layout of a packed array. There are two threads executing the

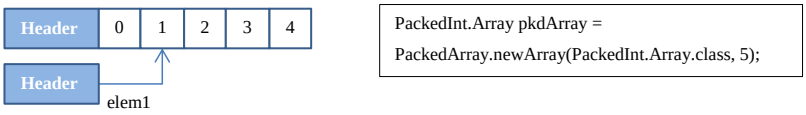


Figure 3.7: Memory Layout of a PackedInt Array

following two pseudo code segments concurrently:

```
//Thread-1:
pkdArray.packedLock();
pkdArray.update();//update pkdArray
pkdArray.packedUnlock();
//Thread-2:
PackedInt elem1 = pkdArray.at(1);
elem1.packedLock();
elem1.update();//update the element elem1 of pkdArray
elem1.packedUnlock();
```

This is actually the scenario 3 of on-heap PackedObjects described in Section 3.2. In this case, PackedObject elem1 and pkdArray are two different objects with different packed data addresses, but elem1's data field is contained in pkdArray. Thus associating these two relating PackedObjects with two distinct monitors by hashing their packed data addresses will end up with synchronization failure since two threads could update the same underlying data at the same time.

However, this situation only happens while synchronizing at different granu-

larities (i.e. the container and nested PackedObjects) at the same time. In order to solve the contention between the container and nested PackedObjects, we use a structure *ContainerInfo* to keep track of the monitor information associated with the container PackedObject. Each container PackedObject will have a *ContainerInfo* associated with it. The structure *ContainerInfo* keeps track of the pointer to the container PackedObject, the number of monitors that are currently associated with the container PackedObject and a linked list for keeping the monitors associated with the container. The definition of structure *ContainerInfo* is as:

```
typedef struct ContainerInfo{
        UDATA container;
        UDATA lockedMonitorCount;
        thread_monitor_t mutex;
        MonInfo* monitorList;
        ObjectMonitor * containerMonitor;
}ContainerInfo;
```

The *ContainerInfo* associated with each container PackedObject is shared by all the PackedObjects that derive from the container PackedObject. For Scenario 2 of on-heap PackedObjects described in Section 3.2, where two threads lock on different nested PackedObjects of the same container, two monitors will be associated separately with these different nested PackedObjects and then added to the monitorList of the *ContainerInfo*.

The mutex is defined in the structure ContainerInfo for mutual exclusion in updating the ContainerInfo. The field containerMonitor in ContainerInfo is used to solve the contention between locking on the container and nested PackedObjects. As addressed above, if two threads lock on the container and its nested PackedObject respectively, they should share the same monitor. If the thread locks on the container PackedObject first, it will first associate the container PackedObject with a monitor by hashing the packed data address of the container. Apart from adding the monitor to monitorList in ContainerInfo, the monitor will also be assigned to containerMonitor. The incoming thread locking on the nested field will check if containerMonitor is null or not. If the containerMonitor is null, the thread will associate the nested object with a monitor by hashing the address of the nested PackedObject and then add the monitor into the monitorList. Otherwise the thread will acquire the containerMonitor for the nested PackedObject.

3.3.2 Packed Regions

In order to solve the synchronization contention between the container and its nested Packedbjects, we name the packed data area of the container PackedObject as a Packed Region. Figure 3.8 shows the Packed Region for on-heap PackedObject and off-heap PackedObject.

Whenever a thread is trying to lock on a PackedObject, it needs to first identify the container of that PackedObject and find the corresponding containerInfo for the container PackedObject. For on-heap PackedObjects, each

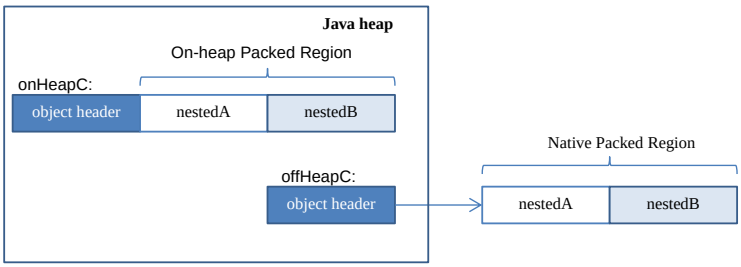


Figure 3.8: On-Heap Packed Region and Off-heap Packed Region

Packed Region will be associated with a structure packedRegionInfo which contains the containerInfo and the hash code of the container PackedObject. According to the structure of the PackedObject header, for on-heap PackedObjects, we can always acquire an on-heap PackedObject's container PackedObject by accessing the target field of the header. Thus we use a hash-table PackedRegionTable to map each on-heap container PackedObject with its PackedRegionInfo. Whenever an on-heap PackedObject is created, there will be an entry of PackedRegionInfo added into the hash-table Packed-RegionTable. The structure Packed-RegionInfo is defined as follows.

```
typedef struct  PackedRegionInfo{
        ContainerInfo containerInfo;
        U_32 hash;
} PackedRegionInfo;
```

However, for off-heap PackedObjects, because native code could arbitrarily wrap multiple adjacent off-heap PackedObjects together as one, the container

PackedObject for off-heap PackedObjects could be changed. The way to determine whether a native PackedObject is contained by another native PackedObject is only based upon their native addresses and the lengths of packed data. Therefore, we use a sorted linked-list *NativePackedRegionList* to store the *NativePackedRegionInfo* for each native container PackedObject. The structure *NativePackedRegionInfo* is defined as follows.

```
typedef struct  NativePackedRegionInfo{
        ContainerInfo containerInfo;
        UDATA nativeAddr;
        UDATA packedDataLen;
        struct NativePackedRegionInfo * next;
} NativePackedRegionInfo;
```

Whenever a native PackedObject is created, a node (represented by *nativePRI*) of type *NativePackedRegionInfo* will be initialized with the containerInfo, the native address and the packed data length of the container PackedObject. Then the function *findContainerUpdateList (&NativePackedRegionList, nativePRI)* will be invoked to add the *nativePRI* to the linked-list *NativePackedRegionList* or update the linked-list based on different situations. Since the native methods can arbitrarily split the native packed data into different packedObjects or wrap consecutive native packed data in to one PackedObject, the containerInfo for a native packedObject can be changed. *findContainerUpdateList()* will execute one of the three operations based on

different situations:

1. If a node N is found in the nativePackedRegionList containing the Packed Region of *nativePRI*, *N* is the container for the newly created native PackedObject. findContainerUpdateList returns the node N.

2. If the Packed Region of *nativePRI* is not overlapped with any other node in the list, insert the *nativePRI* into the nativePackedRegionList and keep the list being sorted in the ascending order of the NativePacked-RegionInfo.nativeAddr, and return *nativePRI*.

3. If the Packed Region of *nativePRI* is found containing the nodes in the list, insert *nativePRI* into the nativePackedRegionList and delete the nodes contained by *nativePRI* and return *nativePRI*. The monitorLists linked in the containerInfo of the deleted nodes will be merged into the monitorList of *nativePRI*.

Since the wrapping operations are arbitrary, it may happen that two Packed Regions are partly overlapped. Figure 3.9 shows the irregular relationship between two Packed Regions of *nativePRI_1* and *nativePRI_2* and the node in the nativePackedRegionList. In order to keep the simplicity of our algorithm, we considered that *nativePRI_1* contains the native packed Region whose start address is greater than the packed region of *nativePRI_1*, *nativePRI_2* is contained by the native Packed Region whose start address is less than the packed region of *nativePRI_2*.

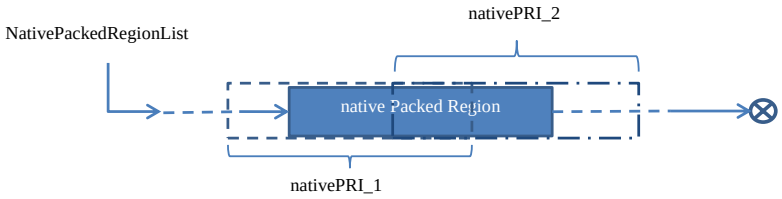


Figure 3.9: Irregular overlapping between different native Packed Regions

3.3.3 Synchronization Interfaces

With the VM structures described above, we can define and implement the synchronization interfaces for PackedObjects. The basic synchronization operations locking and unlocking are represented by packedLock and packedUnlock. These operations are implemented as VM internal functions. Therefore, they could be invoked from either the Java side or the native side. packedLock and packedUnlock are defined in class PackedObject as native methods which call VM internal functions *PackedLock/PackedUnlock* to execute locking/unlocking operations on the PackedObject. Native code can also execute locking/unlocking operations on a particular PackedObject via JNI interfaces that invoke the VM internal functions *PackedLock/PackedUnlock*. Figure 3.10 shows the invoking relationship between synchronization interfaces and underlying VM functions.

The main task of function *PackedLock(PO)* is to acquire the corresponding monitor associated with PO and enter the monitor, while *PackedUnlock(PO)* will find the corresponding monitor for PO and exit it. As discussed in Section 3.3.1, a PackedObject could be associated with a monitor by either

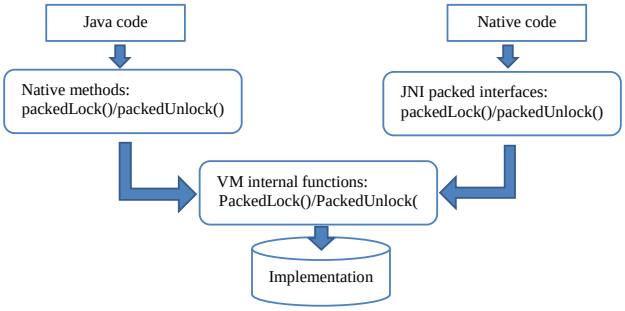


Figure 3.10: The Invoking Flow of VM Internal Functions PackedLock/-PackedUnlock

hashing the address of packed data or sharing the same monitor with its container, depending on whether the container of PO is locked or not and how many monitors associated with the container are currently being locked. This information could be acquired by accessing the containerInfo of PO. Function *getPackedContainerInfo(PO)* is used to fetch the corresponding containerInfo for PO. If PO is an on-heap PackedObject, the containerInfo can be acquired by hashing the target PO and get the entry to *PackedRegionTable* for the container PackedObject. Otherwise, PO is an off-heap PackedObject, and the containerInfo will be acquired by looking up the *NativePackedRegionList* to find the *NativePackedRegionInfo* for the container PackedObject. Figure 3.11 shows the flowchart of function *getPackedContainerInfo()*.

Once the containerInfo for PO is acquired, PackedLock() will check container-Info->lockedMonitorCount, which represents the number of monitors associated with the container that are currently being locked. The flowchart of PackedLock() is shown in Figure 3.12. Three cases are to be addressed based

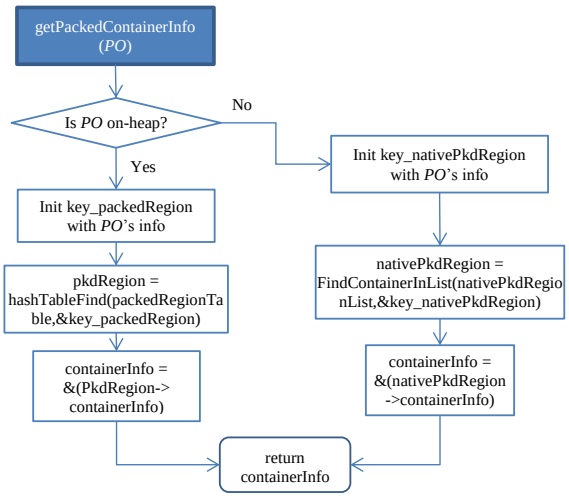


Figure 3.11: Flowchart of Function getPackedContainerInfo()

on the value of containerInfo->lockedMonitorCount:

1. containerInfo->lockedMonitorCount equals 0 which means there is no monitor associated with the container. Call *HashTableFindOrAdd(PO, CInfo)* to hash the PO and associate it with a monitor in monitorTable, and then add the monitor to *containerInfo->monitorList*. If the PO is a container PackedObject, assign the monitor to *containerInfo->containerMonitor*. At the end, update containerInfo->lockedMonitorCount to 1.

2. *containerInfo->lockedMonitorCount* equals 1 which means one monitor associated with the container is being locked. PackedLock() first checks if *containerInfo->containerMonitor* is null or not. If *containerInfo ->containerMonitor* is not null, which means the monitor associated with the container PackedObject is locked, acquire the container-

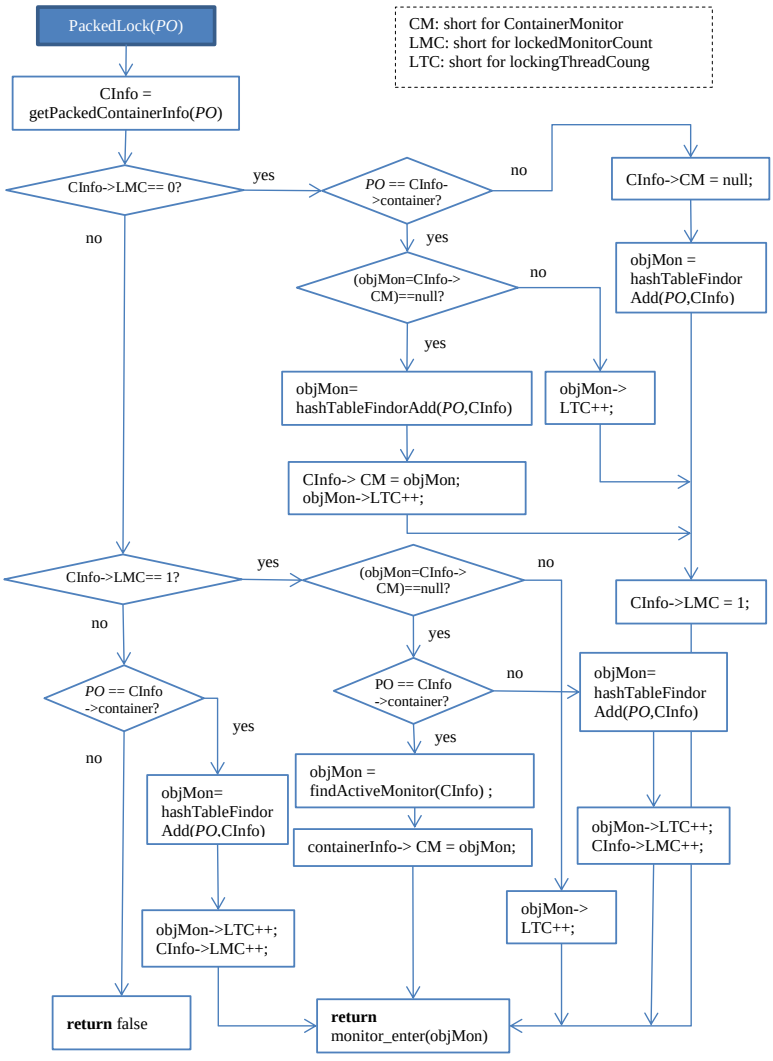


Figure 3.12: Flowchart of PackedLock()

Monitor. Otherwise, *containerInfo->containerMonitor* is null, which means the only one monitor being locked is associated with one of the nested objects instead of the container PackedObject. In this case, if PO is the container, it will simply fetch the monitor from *containerInfo->monitorList* by calling *findActiveMonitor(CInfo)* and assign the monitor to *containerInfo->containerMonitor*. If PO is not the container but a nested PackedObject, PackedLock() will call hashTableFindOrAdd(PO,CInfo) to acquire the monitor associated with PO by hashing its packed data address. If this monitor is newly created, it will be added into *containerInfo->monitorList*.

3. *containerInfo->lockedMonitorCount* is greater than 1, which means there are more than 1 locked monitors associated with different nested PackedObjects of the container. In this case, if PO is the container, which means PackedLock() is trying to lock on the container whose nested fields are currently locked by more than 1 thread, then synchronization on PO fails, and PackedLock() returns false. Otherwise *PackedLock()* calls *hashTableFindOrAdd(PO, CInfo)* to associate PO with a monitor by hashing its packed data address and add the monitor into *containerInfo->monitorList*. *containerInfo->lockedMonitorCount* will be updated accordingly afterwards.

Likewise, the main task of PackedUnlock(PO) is to acquire the corresponding monitor associated with PO and exit the monitor. Figure 3.13 shows the

flowchart of PackedUnlock(). After the containerInfo for PO is acquired, PackedUnlock() checks containerInfo->lockedMonitorCount and deals with the three situations as follows:

1. containerInfo->lockedMonitorCount equals 0, which means no monitor associated with the container of PO is being locked. This situation will lead to an illegal monitor state exception being reported and PackedUnlock() returns false.

2. containerInfo->lockedMonitorCount equals 1. PackedUnlock() will check if containerInfo->containerMonitor is null or not. If it is not null, which means PO's container is currently being locked, then PackedUnlock() will acquire the monitor of the container PackedObject by accessing containerInfo->containerMonitor. Otherwise, containerInfo->containerMonitor is null, which means the locked monitor is currently associated with a nested object, thus PackedLock() call findActiveMonitor() acquire the monitor for PO by searching containerInfo->monitorList. containerInfo->lockedMonitorCount is set to 0 when all threads exit the monitor.

3. containerInfo->lockedMonitorCount is greater than 1, which means there are more than 1 monitors associated with the container of PO currently being locked. In this case, PackedUnlock() will call HashTableFind() to acquire the monitor for PO by looking up the monitorTable, and then exit the corresponding monitor. Once all threads have released

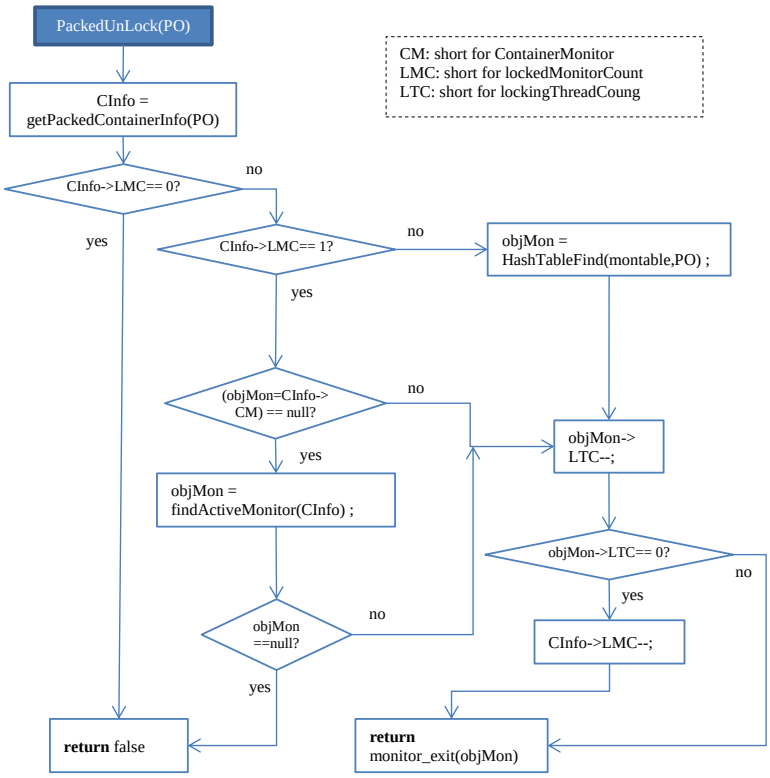


Figure 3.13: Flowchart of PackedUnlock()

the monitor for PO, containerInfo->lockedMonitorCount will be decremented accordingly.

In both of the procedures described above, containerInfo is exclusively accessed and updated. A mutex is defined in the structure ContainerInfo for mutual exclusion in updating its content.

PackedLock and PackedUnlock provide mutual exclusion access to packed object data. With this implementation completed, we can extend synchro-

nization for the packed object data model to support cooperative synchronization by *PackedWait*, *PackedNotify* and *PackedNotifyAll* which works the same way as *Wait()*,*Notify()* and *NotifyAll()* respectively as defined in the Java class Object. The only difference is that packed interfaces acquire the monitor for wait/notify by accessing the containerInfo of the PackedObject. Methods *PackedWait/PackedNotify/PackedNotifyAll* use the same approach to acquire the associated monitor as in *PackedLock/PackedUnlock*, so the nested monitor can be locked out by the containerMonitor. In other words, when the containerMonitor is being locked, the threads locking the nested PackedObect will acquire the containerMonitor instead of the nested monitor. The rest of the thesis will focus on providing more effective approaches for supporting mutual exclusion access to the packed object model and deal with more complicated situations.

3.4 Evaluation and Analysis

This approach is designed to solve the distinctive synchronization situations faced by the packed object data model, such as allowing threads to share the same data from either the Java side or the native side. Standard Java benchmark suites do not apply to the scenario for packed object synchronization. Therefore, we developed our own micro-benchmark programs to evaluate the performance of the approach and analyze its possible bottlenecks. Our micro-benchmark ran on a 1.8GHz 16 Core/32 Thread Xeon, 4xE7520,

Nehalem-based machine with 64GB of RAM, running Linux CentOS.

In order to fairly compare our approach with the Java built-in synchronization used in IBM's J9 VM, in the benchmark programs we avoid the implication of JIT by letting multiple threads lock on an array of objects from the native side with our packed synchronization approach (*PackedSync* for short) and the Java built-in synchronization (Synchronized for short) respectively. For the packed object synchronization mode, the array of objects is defined as a PackedInt array and the synchronization interfaces packedLock()/packedUnlock() are invoked from the native code; while for the standard synchronization mode, the array is a standard array of an object type which contains an int field. The built-in Java native interfaces [54] for Java synchronization MonitorEnter()/MonitorExit() are used for synchronization from the native side. Each thread will randomly go through the entire array and increase the value of each element by 1. Our approach is validated through the benchmark programs which yield the same results under the same experimental setting after the threads finish their tasks.

Our micro-benchmark tests two different types of synchronization scenarios respectively. One is coarse-grained locking by which each thread locks on the entire array and iterates each element of the array and updates its value; another is fine-grained locking by which each thread locks on each element before updating its value and each thread randomly iterates the entire array. In addition, the micro-benchmark evaluates the performance of the *PackedSync* approach by comparing it with the Java built-in synchronization

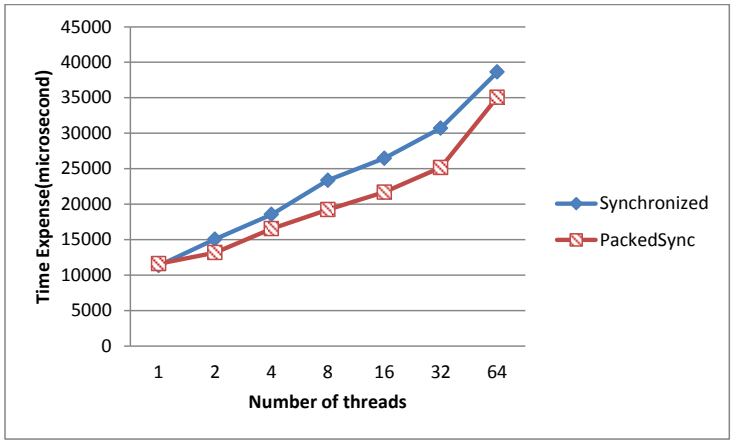


Figure 3.14: Time Overhead Comparison for Coarse-grained Locking

in terms of time efficiency and throughput (the number of operations completed per millisecond). In our benchmark experiments, we calculated the average results across 100 runs. The standard error of mean ranged from 1% to 5% of the average results which is small enough that, therefore, we did not include error bars in the graphs.

With the array size set to 128, we compare the performance of the *PackedSync* and Synchronized approach with a different number of locking threads varying from 1 to 64. Figure 3.14 and 3.15 shows in coarse-grained locking the time expense and throughput comparison between the *PackedSync* approach and the Java built-in synchronization approach. Figure 3.16 and 3.17 shows in experimental results a comparison for fine-grained locking.

As shown in Figure 3.14, *PackedSync* performs much better than the Java built-in synchronization in coarse-grained locking and the time overhead

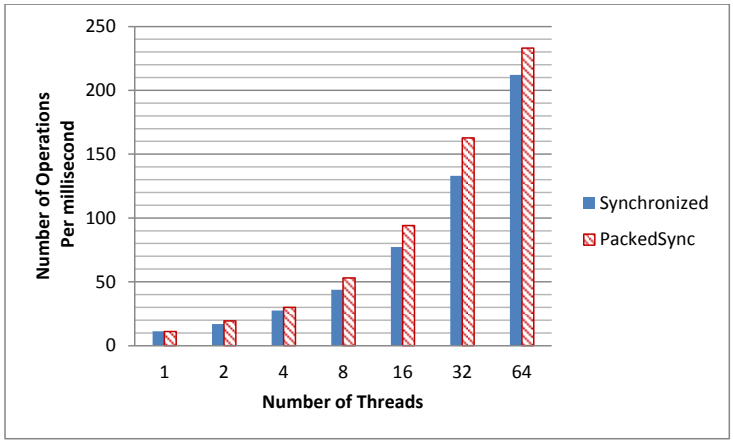


Figure 3.15: Throughput comparison for Coarse-grained Locking

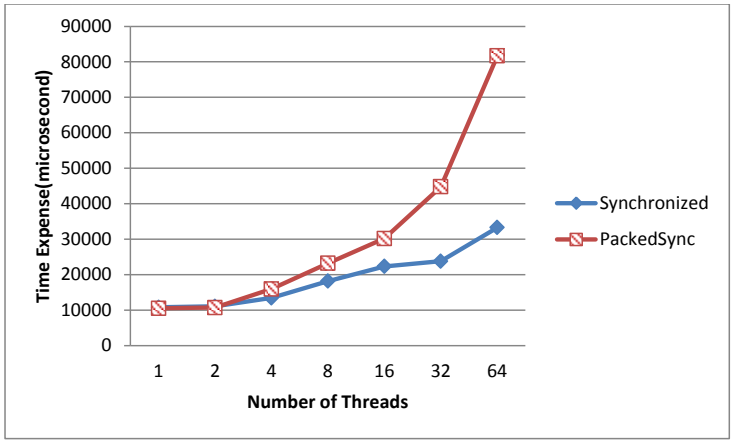


Figure 3.16: Time Overhead Comparison for Fine-grained Locking

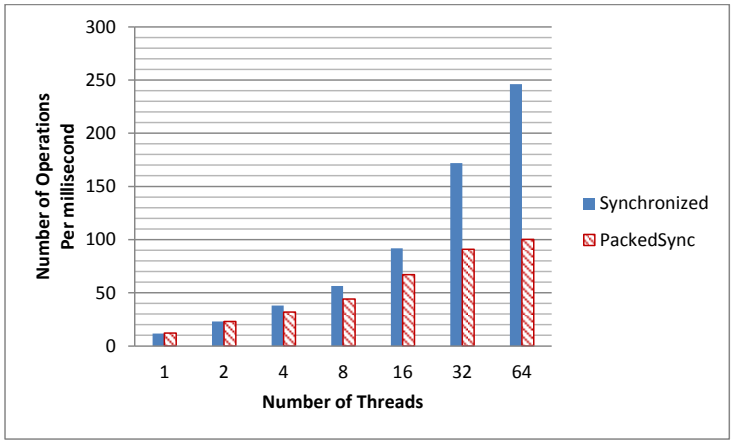


Figure 3.17: Throughput comparison for Fine-grained Locking

dropped about 18% with the *PackedSync* approach. Figure3.15 also shows that the number of locking operations per millisecond for the *PackedSync* is larger than that of the Java built-in synchronization. However, the performance of the *PackedSync* is relatively worse than Java built-in synchronization in the case of fine-grained locking. Their performance is similar when the number of locking threads is small. However, with the increase of the number of locking threads, the time overhead of *PackedSync* increases dramatically. Figure 3.17 shows the throughput of *PackedSync* increases rather slowly in comparison with Java built-in synchronization with the increase of the number of working threads.

As seen from the figures shown above, the scalability of *PackedSync* in fine-grained locking scenarios is relatively poor. The number of locking operations has a significant impact on the overall performance of the *PackedSync* ap-

proach. The reason is because in the *PackedSync* approach, the containerInfo is shared among all threads locking on the container PackedObjects, every locking thread needs to lock on the containerInfo->mutex in order to keep the consistency of the containerInfo throughout the process of PackedObject synchronization. In fine-grained locking, the number of locking threads increases the contention rate on that lock. Therefore the increase of contention on the containerInfo->mutex decreases the PackedSync's performance greatly. In the experiment, since the array size and the number of threads determine the number of locking operations for fine-grained locking, the number of locking operations for fine-grained locking is 128 times the locking operations for coarse-grained locking at the same number of locking threads.

In order to see the impact of the number of locking operations on the performance, we tested fine-grained locking with the same benchmark by varying the number of locking threads at different array sizes from 2 to 32, and compare *PackedSync* to Java built-in synchronization. As described above, in the benchmark a number of threads randomly go through all the elements of the array and lock on each element before updating its value. For a given size of array, each thread completes the same amount of update operations which is the same as the number of locking operations. The experimental results are shown in Table 3.1 which displays the time expense in microseconds for *PackedSync* and Java built-in synchronization with different array sizes and number of threads.

As shown in Table 3.1, we can see that *PackedSync* performs slightly better

Table 3.1: Comparison of *PackedSync* and Synchronized Approach in Fine-grained Locking with Different Array Sizes (in microseconds)

Array Size	Sync Approaches	Number of Threads				
		2	4	8	16	32
2	Synchronized	10626	16089	17592	20931	23503
	PackedSync	10096	12964	16920	19182	24420
4	Synchronized	10433	13665	17460	22213	23963
	PackedSync	7913	12393	14335	21516	24569
8	Synchronized	9455	14759	17530	21365	26972
	PackedSync	10644	15052	17466	18553	25567
16	Synchronized	11170	15427	18186	20500	24453
	PackedSync	10469	15146	18064	24146	27958
32	Synchronized	13594	15682	17994	20409	25555
	PackedSync	13182	16716	19542	24538	31582

than the Java built-in synchronization when the array size is 8 or less. When the array size becomes greater than 8, the *PackedSync* performance is getting worse than the Java built-in synchronization, and the gap between them is getting larger with the increases in locking threads and the array size. However, the array size is not the primary impact on the performance of fine-grained locking. Given an array size, the total number of locking operations for each thread is the same. The more threads there are, the longer time it will take to finish the task. The reason is because the number of working threads is linearly proportional to the total amount of locking operations which determines the level of contention that can occur at a given time on containerInfo->mutex.

Overall, this approach provides a solution for synchronization of the packed object data model which deals with distinctive situations that traditional

Java synchronization mechanisms have not addressed. The experimental results show that our approach can reduce the time overhead by 18% in comparison with the traditional Java synchronization approach in the coarse-grained locking scenario. However, in the fine-grained locking scenario, our approach shows relatively poorer scalability than Java built-in synchronization.

3.5 Space Analysis

Packed object data model was introduced in IBM's J9 VM to reduce the memory footprint and help users to gain greater control over the object layout. However, in order to synchronize on the packedObjects, it is unavoidable to have extra space for auxiliary structures for synchronization purposes. In the *PackedSync* approach, we built a linked-list monitorList to keep track of the monitor information associated with the PackedObject being synchronized. The size of the monitorList is dynamically changed throughout the process of synchronization. In addition, we also need other auxiliary structures, such as containerInfo, PackedRegionInfo and nativePackedRegionInfo, to keep track of the related information with container PackedObjects in order to support synchronization on both on-heap and off-heap PackedObjects. The auxiliary memory footprint of PackedSync is:

On-heap PackedObjects:

*(sizeof(PackedRegionInfo) + sizeof(ContainerInfo)) * NumOfContainers*

*+ sizeof(MonInfo) * NumOfNodes;*

Off-heap PackedObjects:

*(sizeof(NativePackedRegionInfo) + sizeof(ContainerInfo)) * NumOfContainers + sizeof(MonInfo) * NumOfNodes;*

where NumOfContainers represents the number of container packedObjects being synchronized, NumOfNodes represents how many nodes are in the list of MonInfo. NumOfNodes increases during the runtime as the lock/unlock operations are taking place. NumOfNodes will increase up to the number of distinct nested PackedObjects being synchronized.

Chapter 4

Multi-tier Data Synchronization based on A Concurrent Linked-list

The packed object data model organizes object data in a multi-tier manner (like C structs) in which the object's data is nested in its enclosing PackedObject instead of being pointed to by an object reference as in the traditional Java object data model. We refer to these multi-level nested data structures as multi-tier data structures. In this chapter, we present a multi-tier data synchronization approach, *MultitierPackedSync*, based on a concurrent linked-list which optimizes the state-of-the-art linked-list algorithm (lazy-list [30]) and applies it to the multi-tier PackedObject synchronization. Unlike the PackedSync approach in Chapter 3 which deals with two-tier PackedOb-

ject synchronization in single granularity locking, this approach allows multiple threads to lock on a multi-tier PackedObject in multiple granularities and maintain consistency. In this thesis, we chose to optimize the lazy-list algorithm instead of the Harris-Michael style lock-free list algorithms [29, 46]. The main reason is that the lock-free approaches rely on hardware that supports primitive synchronization instructions, such as compare-and-swap, or test-and-set [22], and lock-free algorithms usually require intricate algorithm design, which make the application hard to debug [27]. Comparatively, lock-based approaches match with our monitor-based synchronization approaches to multi-tier PackedObjects and they are simplified and easier to reason about. In addition to the MultitierPackedSync we proposed in this chapter, the concurrent multi-way tree introduced in Chapter 5 is also based on the lazy-list algorithm, which enhances the lazy-list algorithm and applies it to a concurrent multi-way tree structure.

4.1 Problem Statement

Although the packed object data model can bring a great deal of benefit for Java programming, such as reducing the memory footprint, avoiding unnecessary data marshaling and copying between the Java heap and native memory, there are also challenges to be addressed. Since the packed data can be nested within its enclosing PackedObjects, multiple threads accessing different PackedObjects deriving from the same PackedObject could refer to

the same underlying data. Therefore, accesses to the data fields at different tiers of the PackedObject from different threads can cause conflicting operations on the overlapped packed data. Synchronization approaches on multi-tier PackedObjects should be able to deal with the conflict of locking on overlapping data regions. This is out of the scope of the traditional Java synchronization model. For example, consider a 3-tier PackedObject A of class type *ThreeTierClass* containing a 1-tier PackedObject field B and a 2-tier PackedObject field C. The *TwoTierClass* contains two 1-tier PackedObject fields D and E. The three class types *OneTierClass*, *TwoTierClass* and *ThreeTierClass* are defined as:

```
//OneTierClass.java
@Packed public final class OneTierClass extends
   PackedObject
{
        public int content;
}
//TwoTierClass.java
@ImportPacked{ OneTierClass }
@Packed public final class TwoTierClass extends
   PackedObject
{
        public OneTierClass D;
        public OneTierClass E;
}
//ThreeTierClass.java
@ImportPacked{ OneTierClass , TwoTierClass }
@Packed public final class ThreeTierClass extends
   PackedObject
{
        public OneTierClass B;
        public TwoTierClass C;
}
```

The memory layout of PackedObject A of type ThreeTierClass is shown in

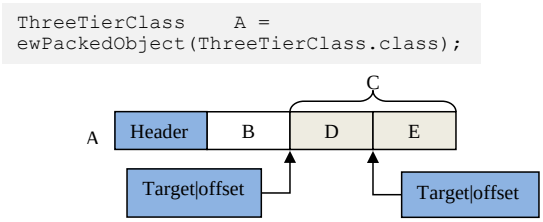


Figure 4.1: Memory layout of PackedObject A of Type ThreeTierClass

Figure 4.1.

As described in Section 2.1.3, target and offset in the PackedObject header determine the start address of the packed data. Different threads can access packed data in any tier concurrently by referring to the derived PackedObject header. For example, as shown in Figure 4.1, if two threads are respectively attempting to access $A.C$ and $A.C.E$, their operations on the overlapped packed data can cause inconsistency of the packed data without multi-tier data synchronization. Our approach should be able to detect the overlapping data in different tiers and ensure mutually exclusive access to the overlapping PackedObjects from the conflicting threads.

As Java threads synchronize through monitors, and each object has a monitor associated with it, detecting the overlapping PackedObjects from different tiers and maintaining the monitor association for those PackedObjects are the major challenges for this approach.

4.2 Technical Approach

In our approach, a concurrent linked-list is utilized to keep track of the monitor information associated with the PackedObjects that share the same Packed Region (e.g. PackedObjects $A.C$ and $A.C.E$ in Figure 4.1). The list of monitor information has to be shared and concurrently accessed by multiple threads that are locking on those PackedObjects. A lock-based scheme is most widely used due to its simplicity of implementation and many lock-based concurrent linked-list algorithms have been developed [31]. We studied one of the state-of-the-art lock-based concurrent linked-list approaches, and then optimized and applied it to our approach for multi-tier data synchronization.

4.2.1 Lazy Concurrent Linked-list Algorithm

The lazy concurrent linked-list (lazy-list for short) algorithm [30] is an improvement on the optimistic-list algorithm [31]. In the lazy-list algorithm, the entries are removed from the list in a lazy manner marking the entries as logically removed at first and then physically unlinking them from the list (physical removal).

There are three basic operations on the linked-list: traversal, insertion and deletion, where both insertion and deletion require the traversal operation. The list is kept sorted by the keys of the entries. During traversal, since no update is needed, no additional synchronization is required. Figure 4.2

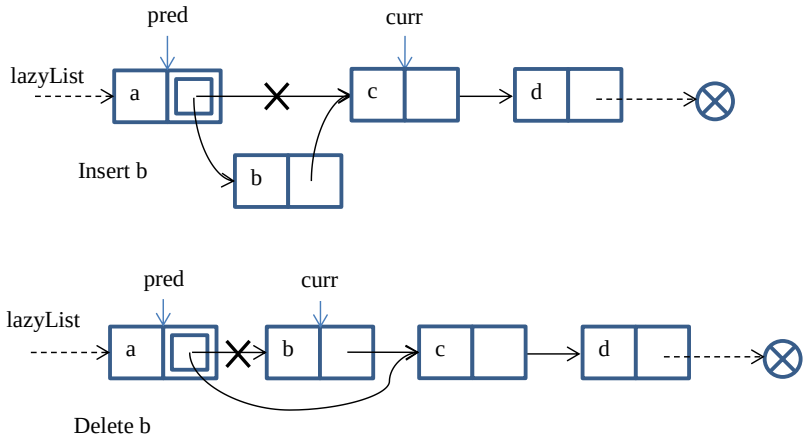


Figure 4.2: Insertion and Deletion of Entries of a Lazy-list

shows the insertion and deletion of entries of a lazy-list.

For insertion, the method traverses down the list searching for the place to add a key while keeping the list sorted. The working thread maintains a local pointer, *curr*, pointing to the current entry that is being traversed, and a pointer, *pred*, pointing to the predecessor entry of the current entry. When the key of *curr* is found higher than the key being added, and *pred*'s key is lower than the key being added, the method locks *pred* and *curr*. Because there is a gap between the unsynchronized traversal and the lock acquisition, it calls validate() to make sure that the correct entries are locked. The function validate() will check if both *pred* and *curr* are still reachable in the list by looking at the marked field indicating logical deletion, and check if that *pred*'s next pointer is still pointing to *curr*. If validate() returns failure, the method needs to start over from the beginning of the list. Otherwise, it

carries out the insertion operation.

For deletion, if the method attempts to remove the entry with key k, it scans through the list without acquiring locks, and traverses both marked and unmarked entries. As with insertion, pointers *curr* and *pred* are used for traversal. When *curr* is set to the first entry with a key greater than or equal to k, the traversal stops. If the key of *curr* equals k, the target entry is found, and then the method locks *pred* and *curr* and invokes validate() to make sure the correct entries are being locked. If validate() returns success, deletion of k is carried out. Otherwise, start over from the beginning of the list. If the target entry is absent, the method unlocks the entries and returns false. The target entry is removed from the list in two steps: first its marked field is set to true which indicates the entry has been logically deleted from the list; second the predecessor entry's next field is redirected to point to the successor entry, which physically removes the target entry from the list. The physical deletion is immediately executed after logical deletion.

Immediate physical deletion in the lazy-list algorithm is not desirable for multi-tier data synchronization since the entries could be frequently accessed in a concurrent context. In addition, validation failure will cause the operation to start over from the beginning of the list. This failure could happen frequently in a concurrent context and increase the overall overhead. We will optimize the lazy-list algorithm and adapt it to the approach of synchronization for multi-granularity locking on multi-tier PackedObjects.

4.2.2 Optimized Concurrent Linked-list Approach

Java has language level support for multi-threaded programing. The monitor construct is the critical built-in mechanism in Java that supports synchronization among multiple threads. Our approach is also based on the monitor constructs for multi-tier PackedObject synchronization. In our approach, we maintain a linked-list of monitors and the related information for multi-tier PackedObject synchronization by optimizing the state-of-the-art lazy-list algorithm and allowing multiple threads to concurrently operate on the monitor information in the list.

4.2.2.1 Auxiliary Data Structures

Due to the significant changes in the packed object data model, our approach to synchronize multi-tier PackedObjects should be able to detect the overlapping between PackedObjects in different tiers. Multiple threads locking on the PackedObjects that share the same Packed Region at any tier should be able to check if the packed data regions they are attempting to lock have already been locked by other threads. In our approach, we use a linked-list to keep track of the monitors and the information related to the data regions being locked. The entries in the list are of type *PackedMonitorInfo* defined as:

```
typedef struct PackedMonitorInfo{
j9thread_monitor_t monitor;
    UDATA offset;
    UDATA packedLen;
    UDATA lockingThreadCount;
    I_8   marked;
    I_16          PDCount;//Physical deletion count
    j9thread_monitor_t mutex;
    struct PackedMonitorInfo *next;
}PackedMonitorInfo;
```

The monitor field is a pointer to the monitor associated with the PackedObject being locked. The offset and packedLen represent the offset and the length of the packed object data and identify the data region guarded by the associated monitor. The linked-list of *PackedMonitorInfo* is sorted by the offset, in ascending order. The lockingThreadCount field indicates how many threads are currently locking on this packed data region (identified by offset and packedLen) through the monitor. The marked field is the same as in the lazy-list algorithm described in Section 4.2.1 for indicating logical deletion of the entry (0 represents the initial value, 1 represents logical deletion). Instead of carrying out physical deletion immediately as the lazy-list algorithm does, we add the *PDCount* field to delay the physical deletion operation in order to reduce unnecessary repetition of deletion and insertion for the entries. *PDCount* is initialized as 0. It is triggered to start counting when

the entry is marked as logically deleted. *PDCount* increments whenever the logically deleted entry is re-visited during the traversal of the list. When *PDCount* reaches a predetermined threshold, physical deletion is triggered to unlink the entry from the list. *PDCount* is set to -1 when the entry of *PackedMonitorInfo* is physically unlinked from the list.

The Packed Region defined in Section 3.3.2 is composed of multiple tier data of PackedObjects which can be accessed by multiple threads from different tiers. In order to allow different threads to detect possible conflicting access to multiple tiers of a container PackedObject, we re-defined the structure ContainerInfo to keep the linked-list of *PackedMonitorInfo* and the related information with the Packed Region. Therefore, each Packed Region is associated with a ContainerInfo, on which multiple threads share the linked-list of PackedMonitorInfo. The structure ContainerInfo for multi-tier PackedObject synchronization is defined as:

```
typedef struct ContainerInfo{
        UDATA container;
        PackedMonitorInfo * monitorList;
        j9thread_monitor_t listMutex;
}ContainerInfo;
```

The container field is a pointer to the container PackedObject containing all the other nested PackedObjects. A thread attempting to lock on a data region of the multi-tier PackedObject will have to acquire the ContainerInfo

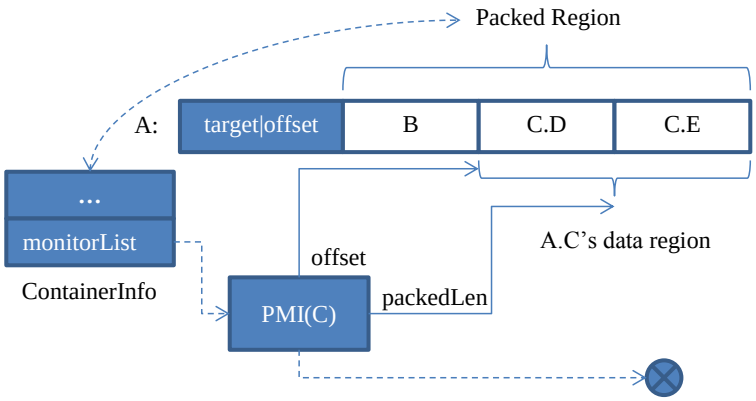


Figure 4.3: An on-heap three-tier PackedObject A and the associated auxiliary data structures

associated with the Packed Region first. Figure 4.3 shows the relationship between an on-heap three-tier PackedObject A (as defined in Section 4.1) and the auxiliary data structures (PMI(C) represents the *PackedMonitorInfo* of the two-tier PackedObject C derived from A).

4.2.2.2 Synchronization Interfaces

With the structures described in Section 4.2.2.1, we can define and implement the synchronization interfaces for PackedObjects. The basic locking and unlocking operations for synchronization are represented by PackedLock and packedUnlock which provide mutual exclusion access to the multi-tier PackedObjects.

A multi-tier PackedObject consists of data regions which can be located at different tiers. One data region can contain another (e.g. $A.C$ contains $A.C.E$ in Figure 4.2). Java synchronizes through monitors. In our algo-

rithm, each data region to be synchronized has a *PackedMonitorInfo* (PMI for short) associated with it. The data region of a PackedObject to be synchronized is identified by the offset and packedLen fields of the associated PMI and guarded by the monitor of the associated PMI. The main task of PackedLock(PackedObject PO) is to traverse the monitorList from the associated ContainerInfo and acquire the corresponding monitor associated with PO and call j9thread_monitor_enter() to enter the monitor, while PackedUnlock(PackedObject PO) traverses the monitorList and looks for the corresponding monitor for PO and exits it by calling j9thread_monitor_exit().

Before describing our algorithm, a few concepts to be used in our description need to be clarified as follows:

Data region of PMI: The data region of a PackedObject is identified by the offset and packedLen fields of PMI and guarded by PMI->monitor. For example, PackedObject A.C's data region in Figure 4.3 is also called the data region of PMI(C), which represents the *PackedMonitorInfo* associated with the PackedObject C.

Containing-region and sub-region: If a data region A contains another data region B, we say A is B's containing-region, and B is A's sub-region. Take the PackedObject A and A.C in Figure 4.3 as an example, PackedObject A's data region is the containing region of A.C's data region, which in turn is the sub-region of A's data region.

Containing-PMI: If the containing-region A is guarded by PMI(A)->monitor, PMI(A) is called the containing-PMI for B, while B is A's sub-region.

Sub-PMI: If the sub-region B is guarded by PMI(B)->monitor, PMI(B) is called the sub-PMI for A, while A is B's containing-region.

Our algorithm starts with acquiring the ContainerInfo associated with the PackedObject to be locked and then traversing the ContainerInfo->monitorList to search for the corresponding monitors for locking/unlocking operations. There are five cases to be addressed during the traversal of the monitorList as shown in Figure 4.4. PMI(PO) represents the PMI for the PackedObject PO to be synchronized. The offset and packedLen of a PMI identify the data region of a PackedObject being guarded by the PMI. PMI(curr) represents the *curr.* PMI being traversed. PMI(pred) represents the predecessor PMI of the *curr.* PMI being traversed. Those five cases are:

- Case 1: The object PO's data region is located separately before the data region guarded by the *curr.* PMI (i.e. PMI(curr)) being traversed.
- Case 2: The object PO's data region is located after the data region guarded by PMI(curr).
- Case 3: The object PO's data region is contained by the data region guarded by PMI(curr).
- Case 4: The object PO's data region contains the data region guarded by PMI(curr).
- Case 5: The object PO's data region equals the data region guarded by PMI(curr).

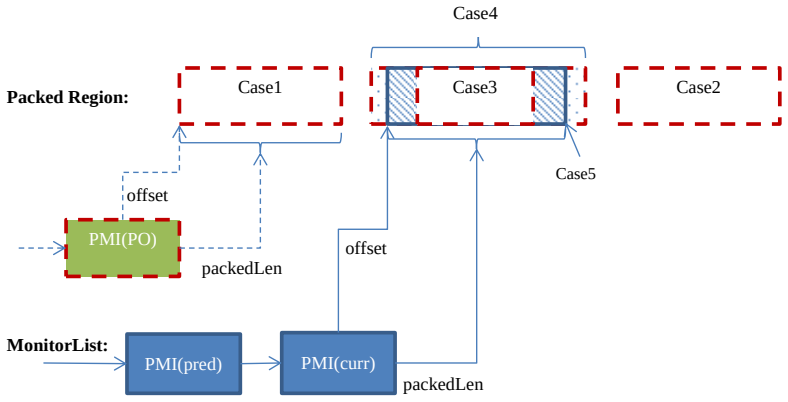


Figure 4.4: Five cases during traversal of the MonitorList

PackedLock:

Case 1: In this case, the method locks PMI(curr) and then PMI(pred). Then it calls validate() to make sure PMI(pred) is still reachable in the monitorList and PMI(pred)->next points to PMI(curr). If validate() returns failure, the method starts over to traverse from the beginning of the monitorList. Otherwise, it creates PMI(PO) and inserts it between PMI(pred) and PMI(curr). In order to verify that no containing-PMI for PO is inserted in between the insertion operation for PMI(PO), the method traverses from the beginning of the monitorList searching for the containing-PMI for PO that is currently being locked. If the PMI is found, it increases PMI->lockingThreadCount by 1 and enters this PMI->monitor. Otherwise, the method carries out the insertion operation for PMI(PO).

Case 2: In this case, the method traverses to the next PMI: PMI(curr)->next

and does the case evaluation again.

Case 3: PMI(curr) is found containing the data region of PO. PMI(curr) is the containing-PMI for PO. In this case, the method locks PMI(curr), and checks if PMI(curr)->lockingThreadCount > 0, which means the containing PMI(curr) is being locked by other threads. If so, the method increases PMI(curr)->lockingThreadCount by 1 and unlocks PMI(curr). Then it traverses down and checks the locking states of sub-PMIs of PMI(curr). Let PMI(sub) represent the sub-PMI of PMI(curr) that is being traversed. If the PMI(sub)->lockingThreadCount > 0, which means that a sub-region of PMI(curr) is being locked, then the method calls j9thread_monitor_wait() to wait until all the threads release PMI(sub) (when PMI(sub)-> lockingThreadCount == 0), and then continues to check the next sub-PMI of PMI(curr). Once all the lockingThreadCounts of sub-PMIs of PMI(curr) are 0, the method acquires PMI(curr)->monitor and enters it.

If PMI(curr)->lockingThreadCount == 0, which means no thread is locking on the containing PMI(curr), then the method traverses down to the next PMI: PMI(curr)->next.

Take the *ThreeTierClass A* defined in Section 4.1 as an example. Figure 4.5 shows the locking scenario of Case 3, in which a thread is attempting to lock on D which is a sub-region of C. It traverses the associated monitorList shown in 4.5. When the traversal reaches PMI(C) which is pointed to by the pointer *curr*, the Case 3 occurs. In this case, the thread could either continue to traverse down the list by moving pointer *curr* to the next node or acquire the

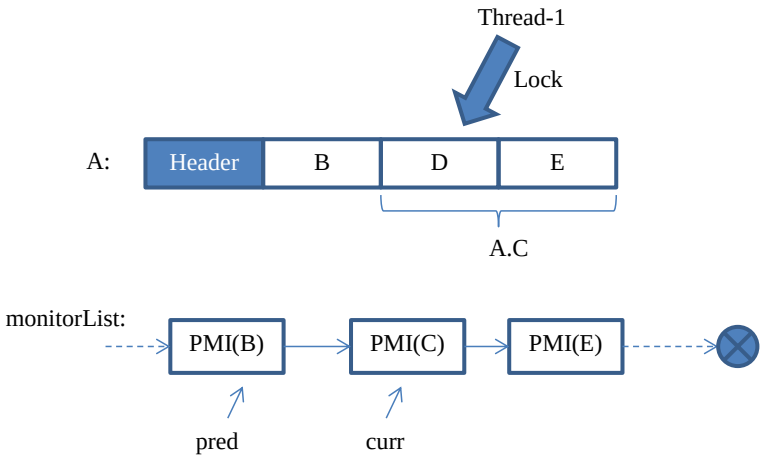


Figure 4.5: Locking Scenario of Case 3

monitor for D's containing-region which is PMI(C)->monitor. This depends on whether the containing-region C is being locked or not. Thus Thread-1 checks PMI(C)->lockingThreadCount to decide what operation to take. If PMI(C)->lockingThreadCount == 0, which means no thread is trying to lock on C, the thread continues traversal to the next PMI (which is PMI(E) in the example of Figure 4.5). Otherwise, the value of *lockingThreadCount* indicates the number of other threads attempting to lock on C. In this condition, the thread is supposed to acquire the PMI(C)->monitor for locking. Before acquiring the monitor, the thread has to traverse PMI(C)'s sub-PMI on the list and make sure no other thread is locking on C's sub-region. If so, the threads wait until all locks on the sub-regions of C are released before acquiring the PMI(C)->monitor.

Case 4: In this case, PMI(curr)'s data region is a sub-region of object PO

to be locked. The method traverses down from PMI(curr) to search for all the sub-PMIs whose data regions are contained by object PO, and set their marked fields to -1, which indicates the PMI should be deleted immediately when their *lockingThreadCount* equals 0. Then the method creates a PMI for object PO and inserts the PMI between PMI(pred) and PMI(curr). In order to verify that there is no containing-PMI for PO being inserted in between the insertion operation of PMI(PO), before the creation of PMI(PO), the method traverses from the beginning of the monitorList and searches for the containing-PMI for PO that is currently being locked. If the PMI is found, the method increases PMI->lockingThreadCount by 1 and enters the containing-PMI's monitor. Otherwise, the method creates PMI(PO) and carries out the insertion operation.

Figure 4.6 shows an example of the locking scenario of case 4, in which a thread is attempting to lock on C. The thread is traversing the monitorList as shown in Figure 4.6. When the traversal reaches PMI(E) which is the sub-PMI of the data region (of C) to be locked, then Case 4 occurs. In this case, PMI(C) is created and supposed to be inserted before PMI(E). Before the insertion, the thread should make sure no containing-PMI for C is inserted and being locked. Therefore, the thread traverses from the beginning of the monitorList and searches for the containing-PMI for C. If such a PMI is found and is being locked, it acquires the containing-PMI's monitor instead of creating PMI for C. Otherwise, the insertion operation is executed.

Case 5: In this case, PMI(curr) is found as the exact *PackedMonitorInfo* for

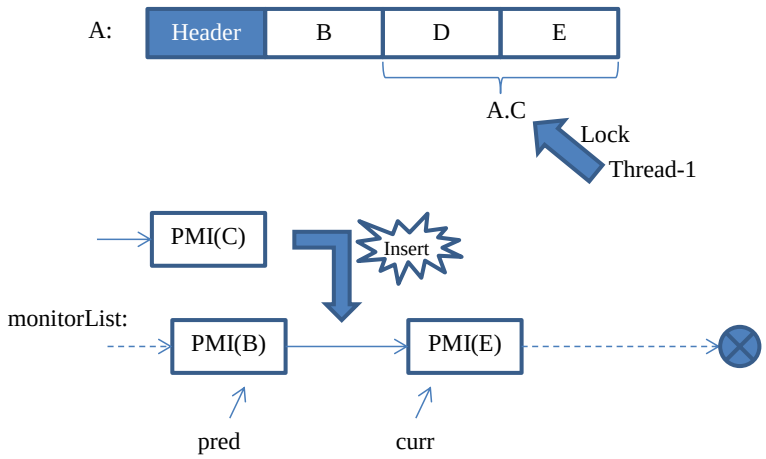


Figure 4.6: Locking Scenario of Case 4

PO. The method checks if PMI(curr)->lockingThreadCount > 0. If so, the method increases lockingThreadCount by 1, and then traverses down and checks the locking states of sub-regions of PMI(curr). If any of the sub-regions of PMI(curr) is being locked, the method calls j9thread_monitor_wait to wait until after all threads release the sub-PMIs of PMI(curr) and then acquires PMI(curr)->monitor and enters it. If PMI(curr)->lockingThreadCount == 0, the method traverses from the beginning of the monitorList and searches for the containing-PMI for PO that is currently being locked. If the PMI is found, the method increases PMI->lockingThreadCount by 1 and enters PMI->monitor. Otherwise, the method increases PMI(curr)->lockingThreadCount by 1 and sets marked and *PDCount* to 0, which means the PMI(curr) is turning back from being logically deleted to being linked back to the monitorList. Finally the method acquires PMI(curr)->monitor

and enters it. Figure 4.7 shows an example of the locking scenario of Case 5, in which a thread is attempting to lock on D and the PMI(D) is in the monitorList. When the pointer *curr* reaches the PMI(D), then Case 5 occurs. In this case, the exact PMI for the object to be locked is found in the monitorList. Therefore, the thread is supposed to acquire the exact PMI's monitor for the locking operation. Before acquiring PMI(D)->monitor, the thread check PMI(D)->lockingThreadCount which indicates the number of threads attempting to lock on D. If the lockingThreadCount is 0, no other thread is attempting to lock on D and D is logically deleted. In this condition, the thread should make sure no containing-PMI for D is inserted and being locked. Therefore, the thread traverses from the beginning of the monitorList and searches for the containing-PMI for D. If such a PMI is found and is being locked, it acquires PMI(D)'s monitor instead of the containing-PMI's monitor. Otherwise, the thread increases PMI(D)->lockingThreadCount by 1, and sets PMI(D)'s fields marked and *PDCount* to 0 indicating the PMI(D) is turning back from logically delete to being linked back to the monitorList. If PMI(D)->lockingThreadCount > 0, this indicates that other threads are also attempting to lock on D. The thread then increases PMI(D)->lockingThreadCount and traverses to next PMI and ensures no thread is attempting to lock on D's sub-region. If so, the thread waits until all the locks on D's sub-region are released and then acquires PMI(D)->monitor.

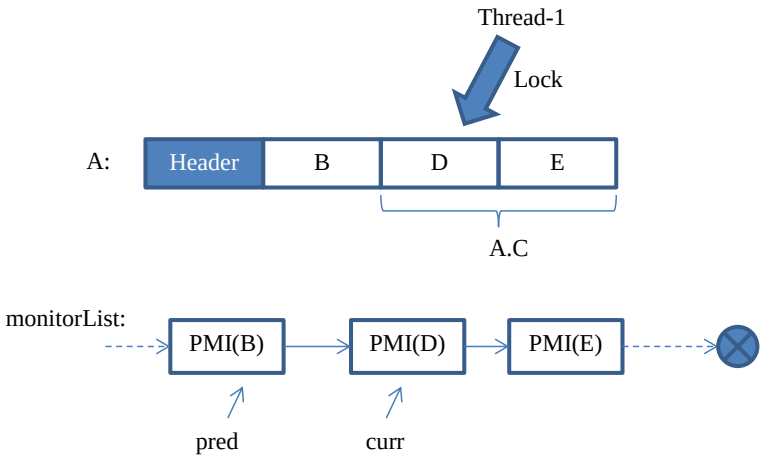


Figure 4.7: Locking Scenario of Case 5

PackedUnlock:

Case 1: This case means the target PMI for the object PO is not found, therefore the method throws a MONITOR_NOT_FOUND exception.

Case 2: This case occurs during the traversal of the monitorList. The method continues traversal to the next PMI: PMI(curr)->next.

Case 3: This case means PMI(curr) is found as the containing-PMI for the object PO. The method traverses down the list to search for the sub-PMIs of PMI(curr). If there is a sub-PMI whose data region contains or equals PO's data region and is being locked, this sub-PMI is the target PMI for unlocking PO. If there is more than one sub-PMI whose data region contains PO's data region, find the sub-PMI whose data region is the best fit for PO as the target PMI (let PMI(target) represent it). Then the method decreases PMI(target)->lockingThreadCount by 1 and acquires this PMI(target)->monitor and

exits it. If none of the sub-PMIs of PMI(curr) meets the requirement, PMI(curr) is the target PMI for the unlocking operation. The method then decreases PMI(curr)->lockingThreadCount and acquires PMI(curr)->monitor to exit it. If PMI(curr)->lockingThreadCount == 0 after decrement, the method sets marked to 1 to indicate that PMI(curr) is logically deleted, and then calls j9thread_monitor_notify_all to notify the threads waiting on the condition that PMI->lockingThreadCount == 0.

Figure 4.8 shows an example of the unlocking scenario of Case 3, in which a thread is attempting to unlock *D*. The thread traverses the monitorList as shown in Figure 4.8. When the pointer *curr* reaches the PMI(C) which is the containing-PMI for the data region (of *D*) to be locked, the Case 3 occurs. In this case, the thread is supposed to release the monitor from the PMI whose data region is the best fit for *D* (if PMI(D) is in the monitorList, PMI(D) is the best-fit PMI) and is being locked (PMI->lockingThreadCount > 0). Therefore, the thread traverses down the monitorList and searches for the target PMI. Once the target PMI is found, the thread decreases the PMI(target)'s *lockingThreadCount* by 1 and releases its monitor. Meanwhile, if the *lockingThreadCount* equals 0 after the decrement, the thread sets marked to 1 to indicate the target PMI is logically deleted, and then notifies other threads waiting on the condition that PMI->lockingThreadCount == 0.

Case 4: This case means PMI(curr) is found as a sub-PMI for the object PO. It is not possible for a thread to lock on a containing-region and

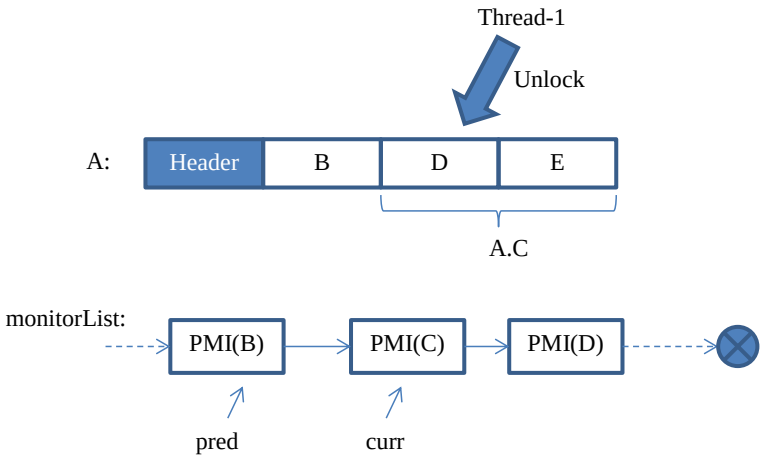


Figure 4.8: Unlocking Scenario of Case 3

then unlock from its sub-region, therefore the method throws a MONITOR_NOT_FOUND exception.

Case 5: This case means PMI(curr) is found as the exact *PackedMonitorInfo* for the object PO. Then the method decreases PMI(curr)->lockingThreadCount by 1 and exits PMI(curr)->monitor. If PMI(curr)->lockingThreadCount == 0, the method sets marked field to 1 to indicate that the PMI is logically deleted, and then calls j9thread_monitor_notify_all to notify the threads waiting on the condition that PMI->lockingThreadCount == 0.

In our algorithm, we assume the physically deleted PMI is not recycled immediately. In a concurrent context, some threads may traverse to the physically deleted node. However they can always detect if a node is physically deleted by checking if PMI->PDCount equals -1 and traverse back to the monitorList by following the physically deleted node's next pointer. Therefore, during the

traversal of the monitorList in PackedUnlock, the method locks each PMI being visited and checks if it is physically deleted (PMI->PDCount == -1). If so, the method releases the lock for the PMI and traverses to the next PMI. If the PMI is not physically deleted, the method checks if the PMI->marked == -1 and PMI->lockingThreadCount == 0. If so, the method carries out physical deletion for this PMI by setting PMI->PDCount to -1 and unlinking the PMI from the monitorList. If the PMI->marked == 1, which means the PMI is logically deleted, then the method increases *PDCount* by 1. If *PDCount* reaches a predetermined threshold after increment, the method carries out physical deletion for this PMI. Otherwise, it releases the lock for the PMI and continues to traverse the next PMI.

4.2.3 Algorithm Analysis

In this section, we analyze how our algorithm can work properly to keep the list sorted and show our major improvements over the state-of-the-art lazy-list algorithm. Meanwhile, we briefly prove the correctness of our algorithm by showing that our algorithm is dead-lock free.

As described in Section 4.2.2, the synchronization operations concurrently traverse through a linked-list ContainerInfo->monitorList and locate the corresponding monitor for the data accessed from different tiers of a PackedObject. The linked-list is sorted in ascending order of PackedMonitorInfo->offset in our algorithm. In order to keep the list sorted, any two adjacent PMIs PMI(pred) and PMI(curr) in the list must satisfy the following condi-

tions:

1. PMI(pred)->offset must be less than or equal to PMI (curr)->offset;
2. If PMI(pred)->offset is equal to PMI(curr)->offset, PMI(pred)->packedLen must be greater than PMI(curr)->packedLen;

Our algorithm satisfies both conditions. In Case 1, PackedLock traverses down the list and finds object PO's data region is located before PMI(curr)'s data region. Therefore PMI(PO)->offset < PMI(curr)->offset. Once PMI(PO) is successfully inserted before PMI(curr), condition 1 is satisfied. In Case 3, PMI(curr) is found as the containing-PMI for PO. As a containing-PMI for PO, PMI(curr) must satisfy that PMI(curr)->offset <= PMI(PO)->offset and PMI(curr)->packedLen >= PMI(PO)->packedLen. Therefore, both conditions can be satisfied if the PMI(PO) is successfully inserted after PMI(curr). In Case 4, the data region of PMI(curr) is found as the sub-region of PO, which means PMI(PO)->packedLen > PMI(curr)->packedLen and PMI(PO)-> offset <= PMI(curr)->offset. The containing PMI(PO) can be then inserted into the list and the conditions continue to be satisfied.

Our concurrent linked-list algorithm is based on the lazy-list algorithm described in Section 4.2.1. We optimized the lazy-list mainly in two aspects: First, the physical deletion is not immediately carried out after logical deletion as in the lazy-list algorithm, but postponed for a certain period of time. Because the deleted PMI is likely to be re-accessed by other threads in a concurrent environment, frequent physical deletion could cause unnecessary

repetition of insertion operations and then increase the overall overhead. Therefore, we use the *PDCount* field as defined in *PackedMonitorInfo* as a way to postpone the physical deletion. Once the PMI is marked for logical deletion, *PDCount* is triggered to start counting. It increments whenever the PMI is visited in PackedUnlock. We set a number as the threshold for the physical deletion count. Before the *PDCount* reaches the threshold, if the logical deleted PMI is being accessed for locking in PackedLock, the PMI will return to normal state by setting *PDCount* to 0 and clearing the marked field. Once the *PDCount* reaches the predetermined threshold, physical deletion is triggered in PackedUnlock and the PMI is physically unlinked from the list.

Second, the locking order is reversed in our algorithm which locks the PMI(curr) before PMI(pred) instead of the contrary order as in the lazy-list algorithm. Our algorithm locks each PMI visited respectively in PackedUnlock and updates the *PDCount* and triggers physical deletion if the *PDCount* reaches the threshold. Physical deletion requires locking on the *curr*. PMI to be deleted and its predecessor PMI. Once the PMI(curr) is found that it meets the condition of physical deletion, PMI(curr)'s lock is already being held by the working thread. Therefore, our algorithm only needs to acquire an extra lock on PMI(pred) and validate if both PMI(pred) and PMI(curr) are reachable in the list and PMI(pred)->next == PMI(curr) before carrying out the physical deletion. If validation failed, due to the reversed locking order (which locks PMI(curr) before PMI(pred)), our algorithm only needs

to unlock PMI(pred) and start from the beginning of the list to search for the valid predecessor PMI to re-lock it and then carry out the physical deletion. Comparatively, the lazy-list algorithm locks PMI(pred) before PMI(curr). If validation returns failure, it has to unlock both locks and start over from the beginning of the list, which increases the locking overhead.

In addition our algorithm is dead-lock free. During the traversal in PackedLock, no PMI is being locked; while during the traversal in PackedUnlock, only one PMI is locked at a time. Both traversal operations can proceed concurrently without interfering with each other. For the insertion operation, locks on PMI(curr) and PMI(pred) are both required. The locking order is kept the same as the deletion operation which locks PMI(curr) before PMI(pred), thus dead-lock can be avoided in our algorithm.

Our approach supports multi-granularity locking on multi-tier data structures, which allows multiple threads to concurrently access the overlapped data regions in different tiers and keep the integrity of the multi-tier data. Threads locking on the data of the higher tier have to wait until the other threads release the locks on the enclosed data regions at the lower tiers. Threads locking on the data of the lower tier notify the threads waiting on them to release the locks.

4.3 Evaluation and Analysis

We evaluated our approaches to multi-tier data synchronization on a 1.8GHz 16 Core/32 Thread Xeon, 4xE7520, Nehalem-based machine with 64 GB of RAM, running Linux CentOS. The algorithms were implemented in the IBM J9 Virtual Machine, and the synchronization interfaces PackedLock and PackedUnlock for mutual exclusion operations were provided. We measured the total execution time in our benchmark with four different algorithms:

1. ***MultitierPackedSync*** The multi-tier data synchronization approach (presented in Section 4.2.2) based on an optimized concurrent linked-list algorithm.

2. ***Lazy_MultitierPackedSync*** The multi-tier data synchronization approach based on the lazy-list algorithm without our optimization described in Section 4.2.1.

3. ***PackedSync*** The two-tier data synchronization approach presented in Chapter 3.

4. ***Synchronized*** The Java built-in synchronization approach.

In our benchmark experiments, we calculated the average results over 100 runs. The standard error of mean ranged from 1% to 5% of the average results which is small enough that, therefore, we did not include error bars in the graphs. Our evaluation is composed of two parts. The first part

evaluates single-granularity locking. In Chapter 3, we developed a two-tier packed object synchronization approach, *PackedSync*, for single-granularity locking. In this part of the evaluation, we continued to use the previous benchmark, in which multiple threads lock on a PackedInt array in either a coarse-grained or a fine-grained manner and access each element of the array in random order and update the data of each element respectively. Our approaches are validated through the benchmark programs which yield the same results under the same experimental setting after the threads finish their tasks.

In order to show the performance gain of our multi-tier data synchronization approach (*MultitierPackedSync*) which is an optimization based on the lazy-list algorithm, we also implemented the *Lazy_MultitierPackedSync* which is based on the lazy-list algorithm without our optimization. In this part of evaluation, we ran the two-tier packedArray benchmark with our *MultitierPackedSync* against *Lazy_MultitierPackedSync*, *PackedSync* and the Java built-in synchronization (*Synchronized*, for short) and compared their performance in terms of time expense. The predetermined threshold for *PDCount* in the *MultitierPackedSync* approach is set to 100. We will discuss the impact of the threshold later.

In the benchmark, we set the array size to 128 and vary the number of threads from 1 to 128. Figure 4.9 and 4.10 shows the performance of the four approaches in coarse-grained locking and fine-grained locking respectively. It should be noted that the Y-axes are different in the two figures. In the

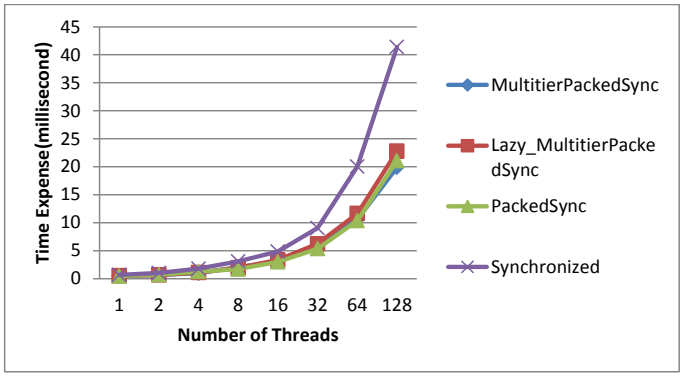


Figure 4.9: Time Expense Comparison in Coarse-grained Locking

coarse-grained locking, each thread locks on the entire array and updates each element of the array; in the fine-grained locking, each thread randomly accesses each element of the array and locks on the element before updating its value.

As seen from Figure 4.9, *MultitierPackedSync* performs almost the same as our previous two-tier approach *PackedSync* in coarse-grained locking. *Lazy_MultitierPackedSync* performs slightly worse than the *PackedSync* and *MultitierPackedSync* when the number of threads goes up. In coarse-grained locking, all three approaches largely overtake the Java built-in synchronization and as the number of threads increases, the gap between them is increased considerably. However, in fine-grained locking (Figure 4.10), our approaches perform worse than the Java built-in synchronization. The *Lazy_MultitierPackedSync* performs prominently worse because of the overhead results from the frequent deletion and insertion and the related lock-

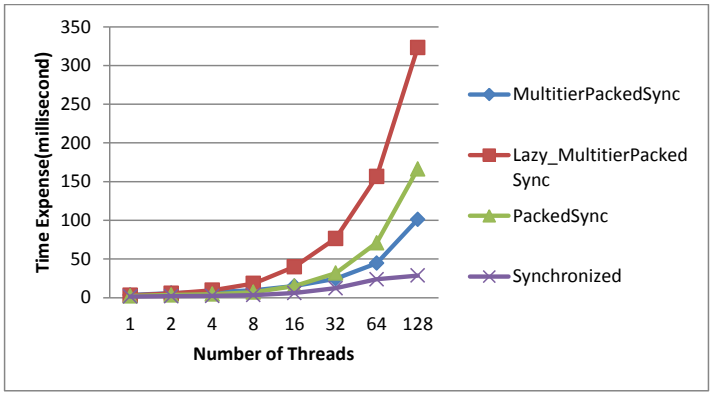


Figure 4.10: Time Expense Comparison in Fine-grained Locking

ing/unlocking operations. The time overhead of both *MultitierPackedSync* and PackedSync also increased dramatically as the number of threads grows in fine-grained locking. Noticeably the *MultitierPackedSync* has a positive outcome compared with PackedSync and *Lazy_MultitierPackedSync* with a large number of working threads. The overall performance is improved substantially in fine-grained locking with the *MultitierPackedSync*. Noteworthy, the performance of all four approaches is very close when the number of threads is small.

As discussed in Section 4.2.3, the postponement of physical deletion in *MultitierPackedSync* reduces the overhead of unnecessary repetition of deletion and insertion to improve performance. The selection of the threshold of *PDCount* influences how long the PMI will be kept in the list after being logically deleted. However, keeping logically deleted entries on the list could have an impact on the length of the monitorList to be traversed. The overhead of

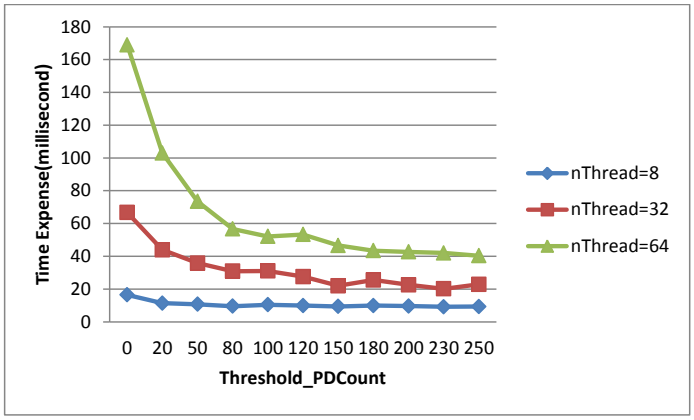


Figure 4.11: Time expense of *MultitierPackedSync* with different thresholds for *PDCount*

increased traversal might counteract the performance gained by reducing the insertion and deletion overhead. In order to examine the impact of the threshold for *PDCount* on the overall performance of *MultitierPackedSync*, we ran the same benchmark and compared *MultitierPackedSync* in the fine-grained locking scenario with 8, 32 and 64 threads, and varied the threshold from 0 to 250 (*PDCount* is defined as a 16 bit signed integer). The experimental result is shown in Figure 4.11.

As we can see from Figure 4.11, the time expense is reduced dramatically as the threshold increases. When the threshold is set to 0, our algorithm will not physically remove the entries that are marked as logically removed until the entries are re-visited during the traversal in PackedUnlock. The threshold 0 causes frequent deletion and insertion for entries and intensely increases the overhead. As seen from Figure 4.11, when the number of threads is 64 and the

threshold is set to 0, the time expense of MultierPackedSync is almost four times as much as the time expense of MultierPackedSync with a threshold of 250. When the number of threads is relatively low, the threshold has less impact on the overall time expense. The time expense for 8 threads appears to be unchanged as the threshold varies. When the number of threads is getting larger, the time expense starts to fluctuate as the threshold changes. As we can see, when the threshold is small, the time expense is getting larger as the number of threads increases. This also indicates that the overhead of frequent insertion and deletion of PMI outweighs the performance gain by traversing fewer monitorList entries to find the target PMI. When the threshold continues to grow, the overall time expense becomes stabilized.

Another part of our evaluation is for multi-granularity locking, for which *MultitierPackedSync* is mainly designed. In this part of the evaluation, we examine the multi-granularity locking with *MultitierPackedSync*. In order to show the performance improvement of *MultitierPackedSync* over the lazy-list algorithm, we compared *MultitierPackedSync* with *Lazy_MultitierPackedSync* in a multi-granularity locking scenario. In the benchmark, we let multiple threads access each elements of a three-tier PackedObject (which is of class type *ThreeTierClass* as defined in Section 4.1) array in random order and update the data of the different tiers respectively. The array size is set to 128 as well. Figure 4.12 shows the time expense for *MultitierPackedSync* and *Lazy_MultitierPackedSync* in multi-granularity locking with different numbers of threads varying from 3 to 192, which are multiples of

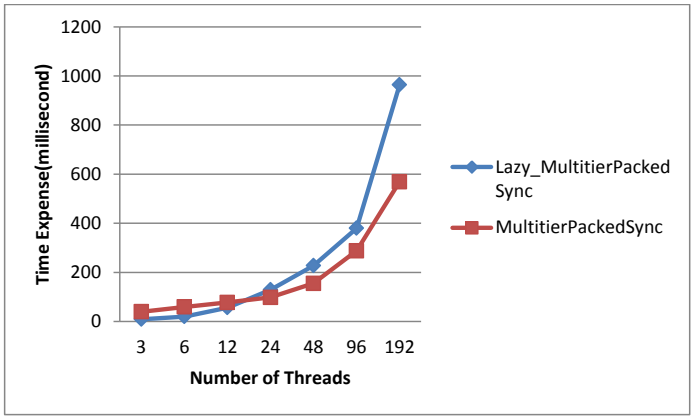


Figure 4.12: Time expense comparison in multi-granularity locking

the tier number. In order to see the impact of array size on the performance of *MultitierPackedSync*, we varied the array size from 2 to 32 and ran the benchmark with *MultitierPackedSync*. The results are shown in Figure 4.13. As seen from Figure 4.12, *Lazy_MultitierPackedSync* performs slightly better than *MultitierPackedSync* in multi-granularity locking when the number of threads is small (less than 12). This is because the traversal overhead of *MultitierPackedSync* counteracts the performance gain by reducing the deletion and insertion overhead, which is low when there are less working threads. As the number of threads goes up, the benefit of our optimization stands out, and the time expense drops considerably in *MultitierPackedSync*. Figure 4.13 compares MulitierPackedSync with different array sizes. As is shown in the figure, the time expense grows dramatically when the array size and the number of threads increase.

Overall, we evaluated our multi-tier data synchronization approach *Multitier-*

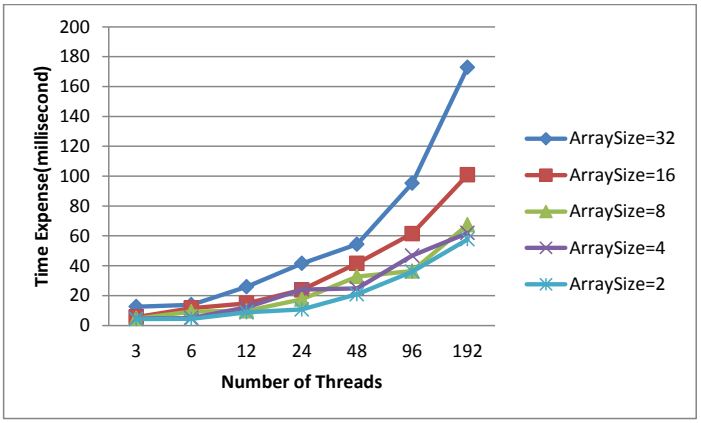


Figure 4.13: Time expense of *MultitierPackedSync* with different array size

PackedSync in single-granularity by comparing it with our previous two-tier PackedObject synchronization approach *PackedSync*, the *Lazy_MultitierPackedSync* that is implemented with the lazy-list algorithm without our optimization and the Java built-in synchronization. The experimental results show that *MultitierPackedSync* shows great improvement over the PackedSync and *Lazy_MultitierPackedSync* in fine-grained locking scenarios and it outperforms Java built-in synchronization in coarse-grained locking. In addition, we ran the multi-granularity locking benchmark with *MultitierPackedSync* and *Lazy_MultitierPackedSync*, The results show that the overall performance is improved significantly with the optimized *MultitierPackedSync* compared to *Lazy_MultitierPackedSync*.

4.4 Space Analysis

Like *PackedSync*, *MultitierPackedSync* approach also needs auxiliary structures for synchronization on multi-tier PackedObjects. However, *MultitierPackedSync* allows threads to delete the *PackedMonitorInfo* after certain period of time by using a threshold *PDCount* to postpone the deletion process. This approach reduces the overhead of frequent deletion and insertion and meanwhile maintains a relatively low space overhead throughout the process of synchronization on multi-tier PackedObjects. In comparison to *PackedSync*, *MultitierPackedSync* is more space efficient even though it still requires certain amount of space for multi-tier data synchronization, which is dynamically changed during the synchronization process.

The auxiliary memory footprint of MultitierPackedSync is:

On-heap PackedObjects:

*(sizeof(PackedRegionInfo)+sizeof(ContainerInfo)) * NumOfContainers + sizeof(PackedMonitorInfo) * NumOfNodes;*

Off-heap PackedObjects:

*(sizeof(NativePackedRegionInfo) + sizeof(ContainerInfo)) * NumOfContainers + sizeof(PackedMonitorInfo) * NumOfNodes;*

where NumOfContainers represents the number of container PackedObjects, and NumOfNodes represent the number of nodes in the linked-List. NumOfNodes varies during the runtime. As the lock/unlock operations are taking place, our algorithm, MultitierPackedSync, unlinks the logically deleted

nodes that are not re-used after being visited in PackedLock for a predetermined threshold. Therefore, NumOfNodes will not keep growing as in PackedSync, which makes MultitierPackedSync more space-efficient.

Chapter 5

Multi-tier Data Synchronization based on A Concurrent Multiway Tree

The multi-tier data structure is the main feature of the packed object data model, in which the packed data is nested in other packed objects instead of being pointed to by an object reference as the traditional Java objects. The multi-tier data structures pose new challenges for synchronization on packed objects. In multi-tier PackedObjects, the packed data can be categorized into different tiers. Data from different tiers can overlap with each other, and one PackedObject data may contain another. Synchronization strategies for the multi-tier data should be able to detect the overlapping data and make sure the data is mutual exclusively accessed and maintain its consistency. Take

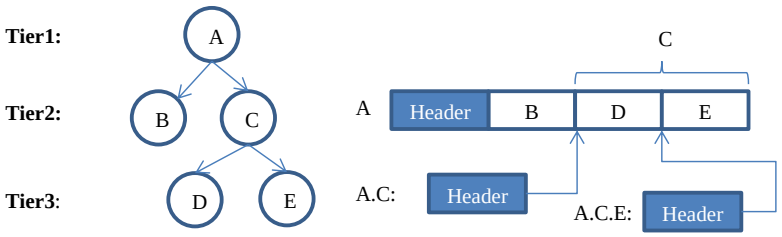


Figure 5.1: Multi-tier data structure of 3-tier PackedObject A

the *ThreeTierClass* defined in Section 4.1 as an example, Figure 5.1 shows the multi-tier data structure for an object A of class type *ThreeTierClass*. As seen from Figure 5.1, the 3-tier PackedObject A contains B and C, while C contains D and E. We use arrows to represent the containing relationship between the PackedObjects, the 3-tier PackedObject A and its derived PackedObjects form a tree structure. In this chapter, we propose a new approach for multi-tier data synchronization based on a multiway tree structure, which represents the containing and ordering relationship among the data in different tiers of a PackedObject.

In Chapter 3, we present an approach *PackedSync* for two-tier PackedObject synchronization which deals with only single-granularity locking on two-tier PackedObjects. When it comes to multi-granularity locking on multiple tier (more than two tiers) PackedObjects, a new approach is necessary. In Chapter 4 we develop a multi-tier data synchronization approach *Multitier-PackedSync* based on an optimized concurrent linked-list. The linked-list is a linearly growing data structure. As the linked-list of *PackedMonitorInfo* grows, the threads have to traverse more nodes to reach the target mon-

itor information and acquire it for synchronization. Thus the overhead is increased. In order to further improve the performance of multi-tier data synchronization, we propose another approach based on a concurrent multiway tree which allows threads to traverse relatively fewer nodes to acquire their target monitor information. Meanwhile the new approach has to deal with more complicated scenarios than linked-lists. Before introducing the tree-based approach for multi-tier data synchronization, we present the concurrent multiway tree algorithm first.

5.1 Concurrent Multiway Tree Algorithm

Unlike linear linked lists, the tree structure organizes data in a hierarchical manner. A general tree (also called multiway tree) can be implemented in a way that each node has pointers to each child of the node plus its data field. However, since the number of children per node can vary greatly and is not known in advance, it might be infeasible to directly link all children in one data structure. One of the effective ways to represent a general tree structure is to keep the children of each node in a linked list of tree nodes. Each node has a pointer to its left-child and a pointer to its right sibling. This is called the left-child right-sibling tree representation [37]. In this section, we first provide the definition and relevant restrictions for the concurrent multiway tree-based ordered data set which represent the containing and ordering relationship among different data ranges. Then we present our concurrent

multiway tree algorithm and analyze and evaluate it.

5.1.1 A Multiway Tree

A multiway tree is a general form of a tree structure. Each node of a multiway tree can have more than two children and the number of the children can vary greatly and is not known in advance. In order to construct a multiway tree efficiently, we use a left-child right-sibling tree to represent a multiway tree structure. In the left-child right-sibling tree, each node points to its left-most child and the right sibling. Meanwhile each node also maintains a pointer to its parent node. Figure 5.2 displays a multiway tree structure and the left-child right-sibling representation of the tree. Parent pointers and null pointers are not drawn in the figure.

The left-child right-sibling tree can be used to store a data set in which different keys can have either containing or ordering relationships. In this section, we define that each node of the left-child right-sibling tree has a key composed of a pair of positive integer values (begin and end) that define a range. For any two keys *k1* and *k2* in a set stored in the multiway tree, they must satisfy the following conditions:

1. k1.begin < k1.end; k2.begin < k2.end;

2. k1 != k2 if k1.begin != k2.begin or k1.end != k2.end;

3. k1 < k2 if k1.end <= k2.begin (e.g. (5,10) < (10,15));

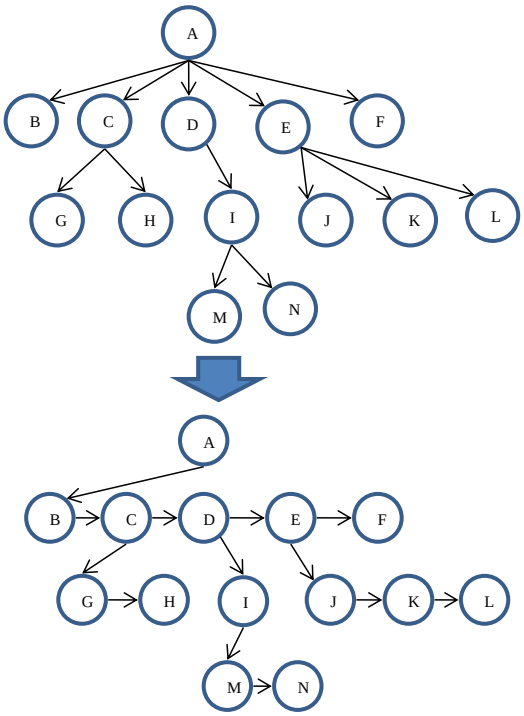


Figure 5.2: A multiway tree represented in a left-child right-sibling tree

4. k1 ∩ k2 == k1 or k2 or NULL (e.g. (5,15) and (10,20) will never exist in the same multiway tree);

5. k1 ⊂ k2 if k1 ∩ k2 == k1 && k1 != k2 (e.g. (5,10) ⊂ (5,20));

We define these restrictions to ensure the relationship between keys can be either distinct or one contains another, which will be further applied to our tree-based approach for multi-tier data synchronization in Section 5.2.

5.1.2 Algorithm

To build a concurrent left-child right-sibling tree (LCRS tree for short) that supports concurrent manipulation, we need to define a data structure (named LCRS_Tree) to keep the root of the tree and a mutex for mutual exclusive access to the structure. The structure LCRS_Tree is defined as:

```
typedef struct LCRS_Tree{
        struct Node* root;
        mutex_t  mutex;
}LCRS_Tree;
```

The node structure of the left-child right-sibling tree is defined as:

```
typedef struct Node{
        struct Key{int begin, int end}  key;
        struct Node* childList;
        struct Node* next;
        struct Node* parent;
        int marked;
        mutex_t  mutex;
}Node;
```

As seen from the data structure, each node of the LCRS tree has a pointer childList pointing to the left-most child and a pointer next pointing to its right sibling node. The pointer parent stores the address of the node's parent. In the concurrent circumstance, we may need to traverse up to the higher level of the tree through the parent pointer in order to locate the target node. The marked field has the same meaning as in the lazy-list algorithm for indicating whether the node is logically deleted (marked == 1) or not (marked == 0). The mutex is used for fine-grained locking. In the LCRS tree, the keys of all sibling nodes are sorted in ascending order. A parent node's key contains all the keys in its childList. A typical LCRS tree with keys is shown in Figure 5.3.

In this Section, we construct a concurrent LCRS tree that supports Search, Insert and Delete operations. A concurrent LCRS-Tree allows multiple threads to execute all three operations concurrently and maintain the consistency.

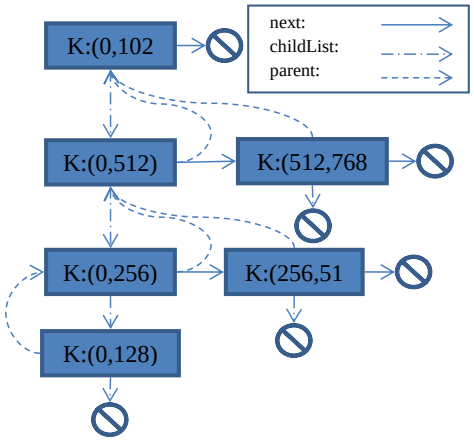


Figure 5.3: A Typical Left-Child Right-Sibling Tree

5.1.2.1 The *Search* and *SearchFrom* Methods

We use a helper function *SearchFrom* to traverse the LCRS tree and return the location information of the target node for the use of three basic operations Search, Insert and Delete. *SearchFrom* is wait-free since it does not need to acquire the locks while traversing the tree. Wait-free search is an important property of our concurrent multiway tree algorithm inherited from the lazy-list algorithm as described in Section 4.2.1. In our algorithm, we need to deal with more complicated scenarios than the concurrent linked list. Figure 5.4 shows the pseudocode of the function *SearchFrom*. SeachFrom starts traversal from the node pointed to by *curr*; *pred* points to the predecessor of *curr*; *isLeftChild* indicates if *curr* is pred's left-most child or right sibling; and *insertParent* indicates that *curr*->key $\subset$ k, and k is sup-

posed to be inserted as the parent of *curr* after *SearchFrom* returns. During the traversal of the LCRS tree, there are 6 cases to be considered.

1. **curr* == NULL. The target node with key k is not found, *SearchFrom* returns NULL.

2. k > (*curr)->key. The key of the target node is greater than (*curr)->key, continue traversing to the next node to look for the target node.

3. k < (*curr)->key. The key of the target node is less than (*curr)->key, so we can guarantee that it will not be found, by virtue of the tree being sorted. Therefore *SearchFrom* returns NULL.

4. k == (*curr)->key. The target node with key k is found in the tree. *SearchFrom* returns the target node pointed to by **curr*.

5. k ⊂ (*curr)->key. The key of the target node is contained by (*curr)->key. *SearchFrom* sets isLeftChild to true and traverses to the childList of **curr* and continues searching for the target node.

6. (*curr)->key ⊂ k. (*curr)->key is contained by the key of the target node. The target node is not found in the tree, and *SearchFrom* returns NULL and sets insertParent to true to indicate that the target node is supposed to be the parent node of **curr*.

The function *Search* is straightforward. It calls the function *SearchFrom* to look for the target node with key k from the root of a tree. Once *SearchFrom*

```
Node* SearchFrom(Key k, Node** pred, Node** curr, boolean
*isLeftChild, boolean* insertParent)
1  *isLeftChild=*insertParent = false;
2  while(*curr!=NULL) do
3    if(k > (*curr)->key ) then//#2
4     *pred = *curr;
5    *curr = (*curr)->next;
6    *isLeftChild = false;
7    else
8      if(k<(*curr)->key) then return NULL;//#3
9      else
10       if(k ⊆ (*curr)-> key) then
11         if(k == (*curr)->key ) then return *curr;//#4
12         else//#5
13           *pred = *curr;
14           Node* child=(*curr)-> childList;
15           *curr = child;
16           *isLeftChild = true;
17           continue;
18       if((*curr)->key ⊂ k) then//#6
19         *insertParent = true;
20         return NULL;
21 return NULL;//#1
```

Figure 5.4: The *SearchFrom*() Method

returns, it checks if *curr* is NULL or not. If *curr* is not NULL and *curr* is not marked as logically deleted, the function Search returns true. Otherwise, the function *Search* returns false. Figure 5.5 shows the pseudo code of the function Search.

```
boolean Search(Key k, LCRS_Tree* tree)
1   Node * pred = NULL;
2   Node * curr = tree->root;
3   boolean isLeftChild;
4   boolean insertParent;
5   Node *target =
SearchFrom(k,&pred,&curr,&isLeftChild,&insertParent);
6   if(target!=NULL&&!target->marked)then return true;
7   else    return false;
```

Figure 5.5: The Search() Method

5.1.2.2 The Insert Method

An insert operation starts by calling the helper function *SearchFrom* to search for the target node with key k from the root of the tree. If the target node is found in the tree and is not marked, the function *Insert* returns false. Otherwise, a new node with key k is supposed to be created and inserted between nodes pointed to by *pred* and *curr* which are returned by *SearchFrom*. Before starting the insertion operation after *pred* and *curr* are returned, the function Insert locks on *pred* and ensures *pred* is not NULL and is unmarked. If *pred* == NULL, *curr* is supposed to be the first node linked by the root of the tree. Then it locks on the structure that holds the link to the root of the tree. If *pred* is not NULL and *pred* is marked, validation of *pred* failed. The function *Insert* needs to release the lock on *pred* and start over to traverse the tree from its root. Otherwise, *pred* is a valid pointer. The successor node pointed to by *curr* is assigned as pred->next or pred->childList based on the value of isLeftChild returned by *SearchFrom*. Our algorithm ensures that marked nodes are unlinked from the tree after the delete operation. Since *pred* is unmarked and locked, which means any other insert or delete operations must have finished or haven't started yet, thus *curr* must be also valid. Then the insert operation is carried out. Figure 5.6 shows the pseudo code of the function *Insert*.

There are two different scenarios of insertion operations for the LCRS tree. One is the simple scenario when the value of insertParent returned by *SearchFrom* is false. In this scenario, we only need to create a target node for the

```
boolean Insert(Key k, LCRS_Tree* tree )
1  Node * pred = NULL;
2  Node * curr = tree->root;
3  boolean isLeftChild, insertParent;
4  StartOver:
5  if((SearchFrom(k,tree,&pred,&curr,&isLeftChild,&insertParent)
   |   != NULL&&!curr->marked) then
6  |  return false;
7  if(pred==NULL) then
8  |  mutex_lock(tree->mutex);
9  |  curr = tree->root;
10 else
11 |  mutex_lock(pred->mutex);
12 |  if(pred->marked) then//in case that pred is deleted from tree
13 |  |  mutex_unlock(pred->mutex);
14 |  |  pred = NULL;
15 |  |  curr = tree->root;
16 |  |  goto StartOver;
17 |  if(isLeftChild == true) then curr = pred->childList;
18 |  else  curr = pred->next;
19 assert(!curr->marked);
20 if(insertParent!=true)then
21-44   Simple Scenario //see detail in Figure 5.7
45 else
46-76   Complex Scenario //see detail in Figure 5.9
```

Figure 5.6: The Insert() Method

key k and insert it between *pred* and *curr* without having to update the target node's childList, since the target node is inserted as the sibling node of *pred* and *curr*. Figure 5.7 shows the pseudo code of the simple insertion operation. Before carrying out the simple insertion, the function *Insert* needs to verify that *curr* satisfies the condition of being the sibling node of the target node (see lines 21-32 in Figure 5.7) in case the *curr* might be changed due to the interleaving of concurrent updating. Lines 21-27 show if *curr* is NULL, the target is to be inserted as the last sibling node of *pred*. In this case, if the key k is found greater than the key of pred->parent, verification failed. The key k should be located in the sibling node of pred->parent. So it traverses up to pred->parent and continues to search for the target by

```
//Simple Scenario: insertParent == false
21   if(curr==NULL) then
22     | if(pred!=NULL && pred->parent!=NULL && k >
       |   |  pred->parent->key) then
23     |   | Node* parent = pred->parent;
24     |   | mutex_unlock(pred->mutex);
25     |   | pred = parent;
26     |   | curr = parent->next;
27     |   | goto StartOver;
28   else
29     | if(!(k < curr->key)) then
30     |   | if(pred==NULL) then mutex_unlock(tree->mutex);
31     |   | else mutex_unlock(pred->mutex);
32     |   | goto StartOver;
33   Node *target = createNode(k);
34   if(pred!=NULL) then
35     | if(isLeftChild == true) then target-> parent = pred;
36     | else     target->parent = pred->parent;
37   target->next = curr;
38   if(pred==NULL) then
39     | tree->root = target;
40     | mutex_unlock(tree->mutex);
41   else
42     | if(isLeftChild==true) then pred-> childList = target;
43     | else     pred->next = target;
44     | mutex_unlock(pred->mutex);
```

Figure 5.7: Simple Insertion

calling *SearchFrom*. Lines 28-32 check if the key to be inserted between *pred* and *curr* is still valid in case other nodes are inserted between *pred* and *curr* after the thread obtains the lock of *pred*. If verification fails, the function *Insert* continues to search for the target node from *curr* by calling *SearchFrom*. Once verification succeeds, the simple insertion is carried out.

When the *curr*'s key is found contained by key k (see lines 18-20 in Figure 5.4) in *SearchFrom*, the target node is supposed to be inserted as the parent node of *curr*, and insertParent is set to true. Then insert operation enters the complex scenario. Figure 5.8 shows an example of the complex insertion scenario. The pseudo code of the complex insertion is shown in Figure 5.9. Likewise, before carrying out the complex insertion, the function has to verify

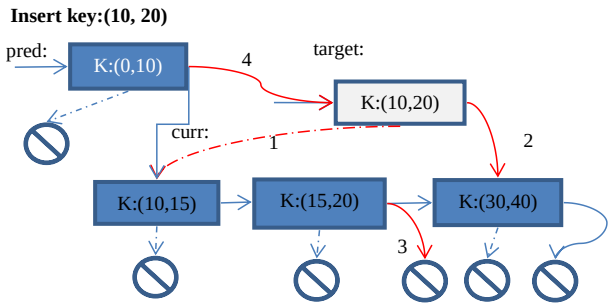


Figure 5.8: An Example of Complex Insertion

that *curr* is not NULL and *curr*'s key is still contained by key k. Otherwise, continue to search for the target node from *curr* by calling *SearchFrom*. Once verification succeeds, the complex insertion is to be executed. In the complex insertion, four pointers have to be updated as noted in Figure 5.8 and Figure 5.9. First, the function links the target node's childList to *curr*. Second, the function needs to traverse down from *curr* to the last node contained by key k by using a hand-over-hand locking strategy (see lines 54-62 in Figure 5.9) and update their parent to target. Once the last child node (referred to as lastChid) is found, it links target->next to lastChild->next. Thirdly the function updates lastChild->next to NULL and finally links *pred* as the predecessor of the target node. The locks are released once all the operations are completed.

5.1.2.3 The Delete Method

A delete operation also starts by calling the helper function *SearchFrom* to look for the target node with key k for deletion. If *SearchFrom* returns

```
//Complex Scenario :insertParent == true
46   if(curr==NULL||!( curr->key ⊂ k)) then
49   |  if(pred!=NULL) then mutex_unlock(pred ->mutex);
50   |  else     mutex_unlock(tree->mutex);
51   |  goto StartOver;
52   Node* target = createNode(k);
53   target->childList = curr;//op1
54   Node* lastChild = curr;
55   mutex_lock(lastChild->mutex);
56   while(lastChild->next !=NULL && lastChild->next->key ⊂ k) do
57   |   lastChild->parent = target;
58   |   Node* next = lastChild->next;
59   |   mutex_lock(next->mutex);
60   |   mutex_unlock(lastChild->mutex);
61   |   lastChild = next;
62   lastChild->parent = target;
63   target->next = lastChild->next;//op2
64   lastChild->next = NULL;//op3
65   mutex_unlock(lastChild->mutex);
66   if(pred==NULL) then
67   |   tree->root = target;
68   |   mutex_unlock(tree->mutex);
69   else
70   |   if(isLeftChild == true) then
71   |   |   target->parent = pred;
72   |   |   pred->child = target;//op4
73   |   else
74   |   |   target->parent = pred->parent;
75   |   |   pred->next = target;//op4
76   |   |  mutex_unlock(pred->mutex);
```

Figure 5.9: Complex Insertion

NULL or a marked node, which means the target node with key k is not found in the tree or found as being marked, then the function *Delete* returns false. Otherwise, the node pointed to by *curr* returned by *SearchFrom* is to be deleted. Before starting the delete operation, the function *Delete* first locks on the predecessor node of *curr* and ensures *pred* is not NULL and is unmarked. If *pred* is marked, the function has to start over to traverse from the root of the tree and search for the target node. If *pred* is NULL, the target node to be deleted is supposed to be the first node pointed to by the root of the tree. In this case, the function *Delete* only needs to acquire a lock on the structure that holds the link to the root of the tree. Once this validation succeeds, the delete operation is to be executed. Figure 5.10 shows the pseudo code of the function *Delete*.

In the delete operation, the function *Delete* needs to update the predecessor node's next pointer or childList pointer based on the value of isLeftChild returned by *SearchFrom* when *pred* is not NULL. If *isLeftChild* is true, which indicates the target node to be deleted is the left-most child of pred. Otherwise, the target node is *pred*'s sibling. If *pred* is NULL, the target node to be deleted is the first node pointed to by the root of the tree, so the function *Delete* needs to update the structure LCRS_tree which holds the link to the root of the tree. In case that the tree structure might be interfered by the possible interleaving of other concurrent operations, the function *Delete* also needs to verify that *pred* is still the predecessor node of *curr* (see lines 16-29 in Figure 5.10). If verification failed, the function *Delete* needs to start over

```
boolean Delete(Key k, LCRS_Tree* tree )
1 Node *pred = NULL;
2 Node *curr = tree->root;
3 boolean isLeftChild,insertParent;
4 StartOver:
5    if(SearchFrom(k,tree,&pred,&curr,&isLeftChild,&insertParent)
        ==NULL||curr->marked) then
6       return false;
7    if(pred==NULL) then mutex_lock(tree-> mutex);
8    else
9    |       mutex_lock(pred->mutex);
10   |       if(pred->marked) then
11   |       |   mutex_unlock(pred->mutex);
12   |       |   pred = NULL;
13   |       |   curr = tree->root;
14   |       |   goto StartOver;
15   Node** toBeUpdated;
16   if(pred == NULL) then    toBeUpdated = &tree->root;
17   else
18   |       if(isLeftChild==true) then
19   |             toBeUpdated = & pred->childList;
20   |       else   toBeUpdated = & pred->next;
21   if(*toBeUpdated != curr) then
22   |  if(pred==NULL) then
23   |  |    curr = tree->root;
24   |  |    mutex_unlock(tree->mutex);
25   |  else
26   |  |    if(isLeftChild==true) then curr = pred->childList;
27   |  |    else  curr = pred->next;
28   |  |    mutex_unlock(pred->mutex);
29   |  goto StartOver;
30   assert(!curr->marked);
31   mutex_lock(curr->mutex);
32   curr->marked = 1;
33   if(curr->childList ==NULL) then *toBeUpdated = curr->next;
34   else
35   |   Node* lastChild = curr->childList;
36   |   mutex_lock(lastChild->mutex);
37   |   while(lastChild->next !=NULL) do
38   |   |   lastChild->parent = curr->parent;
39   |   |   Node* next = lastChild->next;
40   |   |   mutex_lock(next->mutex);
41   |   |   mutex_unlock(lastChild->mutex);
42   |   |   lastChild = next;
43   |   lastChild->parent = curr->parent;
44   |   lastChild->next = curr->next;//op1
45   |   mutex_unlock(lastChild->mutex);
46   |   *toBeUpdated = curr->childList;//op2
47   mutex_unlock(curr->mutex);
48   if(pred == NULL) then mutex_unlock(tree->mutex);
49   else   mutex_unlock(pred->mutex);
50   return true;
```

Figure 5.10: The Delete() Method

to continue traversing from *curr* to look for the target node. Once verification succeeds, delete operation is actually started. It first locks on the target node to be deleted and then set its marked field to 1 indicating that the node is logically deleted. If the target node has no children, the deletion is straightforward and can be done by simply assigning curr->next to *pred*'s next or childList pointer if *pred* is not NULL, or to the root of the tree otherwise. If the target node to be deleted has children, the delete operation needs to not only delete the target node from the tree, but also upgrade the target node's childList to the same level as their former parent (which is the target node) by using a hand-over-hand locking strategy to traverse the childList to find the last child node in the list, meanwhile to update their parent to the target node's parent (see lines 35-43 in Figure 5.10). Figure 5.11 shows the latter scenario of the delete operation. Once the last child node lastChild is found, the function *Delete* updates lastChild->next to curr->next and links *pred* as the predecessor of the left-most child of the target node. These two operations are as noted in both Figure 5.10 (see lines 44 and 46) and 5.11. The locks are released once all the operations are completed.

5.1.3 Algorithm Analysis

In this chapter we present a method that allows multiple threads to concurrently update a multiway tree represented in a left-child right-sibling tree. In the left-child right-sibling tree representation, each node contains a key composed of a pair of positive values representing a range. The sibling nodes of

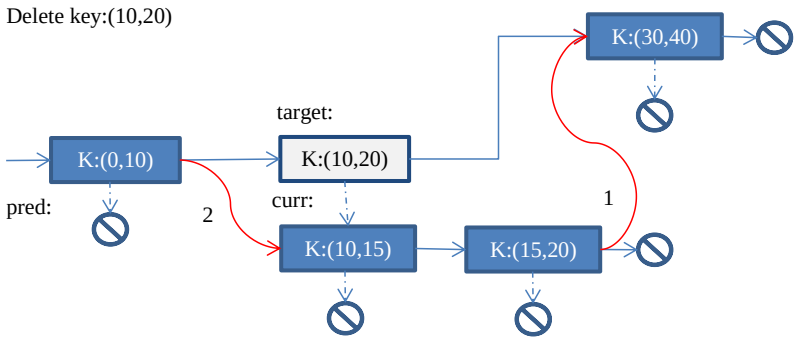


Figure 5.11: Deletion Scenario

the left-child right-sibling tree are organized as a sorted linked list; while the key of the parent node contains all the sibling nodes linked by its childList. Our algorithm is lock-based using the lazy locking mechanism, which was introduced in the lazy-list algorithm for concurrent linked lists. Our algorithm enhances the lazy locking mechanism by applying it to the concurrent multiway tree algorithm, which deals with more complicated scenarios than linked lists. In addition, we also optimize the lazy locking mechanism for our concurrent multiway tree algorithm in the following aspects:

In the insertion scenarios, our algorithm need only acquire one lock on the predecessor node to ensure mutual exclusive access to the tree structure, because once the predecessor node is locked and unmarked, our algorithm ensures that the successor node must be unmarked and connected with the predecessor node. Thus our algorithm can execute the insertion without having to lock on the successor node and call validation for both nodes as in the lazy-list algorithm. The hand-over-hand lock strategy is used in both

complex insertion and deletion scenarios in order to make sure the consistency during the traversal of the childList and keep the simplicity of the algorithm. The validation process is necessary for our algorithm, because our algorithm keeps the wait-free property for search operations inherited from the lazy-list algorithm, which allows searching without acquiring locks. However, instead of acquiring locks on the predecessor node and the successor node and then calling validation as in the lazy-list algorithm, our algorithm simplifies the validation process by locking the predecessor node individually and checking if it is unmarked and allowing threads to traverse a few nodes to locate the target node when validation failed, while the lazy-list algorithm always requires traversing from the beginning of the linked list when validation failed. This can introduce high overhead especially when there is high contention rate with a large number of working threads.

5.1.4 Correctness

In this section we discuss the correctness of the concurrent multiway tree algorithm. To prove the algorithm is correct, we show that the concurrent multiway tree algorithm is dead-lock free, and it generates only linearizable executions.

5.1.4.1 Dead-lock freedom

This algorithm ensures dead-lock freedom by acquiring locks in a certain order. In the algorithm only the Insert and Delete methods require locking

to ensure the consistency of tree structure during the concurrent updating operations. In the simple insertion scenario, there is only one lock at a time being held by the thread of insertion. In the complex insertion and deletion scenarios, more than one lock can be acquired. The algorithm ensures that locks on the predecessor nodes are always acquired before locks on the successor nodes. As described in Section 5.1.1, the predecessor node (p) and successor node(s) must satisfy one of the following two conditions:

1. p->key $<$ s->key
2. s->key $\subset$ p->key

This locking order is mandatory for all the insertion and deletion operations acquiring multiple locks in the concurrent multiway tree algorithm. Therefore dead-lock can be avoided.

5.1.4.2 Linearizability

Linearizability is the standard correctness condition for concurrent shared data structures. It requires that each operation can be assigned a unique linearization point during its execution where the operation appears to take effect atomically [32].

To demonstrate our concurrent multiway tree algorithm is linearizable, it is sufficient to define a linearization point for each operation and then show that said operation produces equivalent results to a sequential execution in which the operation appears to occur instantly at the linearization point.

The linearization point for a failed Search(tree, k) operation occurs at the point when the node with key k is marked by a successful Delete(tree, k) operation, or at the point when *curr* is found to be NULL or *curr*->key is greater than k or contained by k. k will not be found in further search from this point on because *curr*->next->key must be greater than *curr*->key and *curr*->childList->key must be contained by *curr*->key. The linearization point for a successful Search(tree, k) operation occurs at the point when *curr*->key is found equivalent to k and *curr*->marked is not checked.

The linearization point for a unsuccessful Insert(tree, k) is the moment at which an unmarked node with the same key is found in the tree. For a successful Insert(tree, k) operation, the linearization point is the moment at which the pred->childList or pred->next is set depending on whether the target node is to be inserted as pred's child or sibling node.

For a successful Delete(tree, k) operation, the linearization point is the moment when the target node is marked (line 29 in Figure 5.10). For a unsuccessful Delete(tree, k) operation, the linearization point is the point at which it finds a marked node with the same key or the *curr* is found to be NULL. The linearization point of unsuccessful Delete(tree, k) operation is the same as the unsuccessful Search(tree, k).

5.1.5 Evaluation and Analysis

We evaluated our concurrent multiway tree algorithms on a 1.8GHz 16 Core/32 Thread Xeon, 4xE7520, Nehalem-based machine with 64 GB of RAM, run-

ning Linux CentOS. The algorithms were implemented in Java 1.7.0. We varied the percentage of Search() method calls and the percentage of Insert() and Delete() method calls. The keys of the LCRS tree are randomly generated, but satisfied the restrictions defined in Section 5.1.1. The integer values of the keys are in the range between 0 and MAX_RANGE which is a power of 2. At the beginning of the benchmark, we initialize the LCRS tree with a certain number (MAX_RANGE/4) of nodes with randomly generated keys within the range. The number of operations (Search(), Insert() or Delete()) being executed is the workload, which is defined as equivalent to MAX_RANGE. The workload is evenly divided by the number of threads and assigned to each thread. Each thread then randomly generates keys in the predefined range and carries out operations with respect to the given percentage of the three types of operations. In all our benchmarks, we measured throughput: the number of operations completed per millisecond over the course of the entire workload, averaged across 100 runs. The standard error of mean of our experiment results ranged from 1% to 5% of the average results which is small enough that, therefore, we did not include error bars in the graphs.

We implemented three algorithms in all:

1. ***Global-rwLock*** we use a global reader-writer lock to ensure mutual exclusive access to the LCRS tree. Search() operations acquire a global read lock before executing. The update operations Insert() and Delete() must acquire a global write lock before executing.

2. ***Lazy-Tree*** we implement the concurrent LCRS tree based on the lazy locking mechanism used in the lazy-list algorithm.

3. ***Optimized*** the optimized algorithm for concurrent LCRS tree based on the lazy locking mechanism used in the lazy-list algorithm.

We compare the three algorithms listed above in the following experimental settings:

Small workload: nTask = MAX_RANGE = 1024

Large workload: nTask = MAX_RANGE = 16384

Search intensive mode: 75% Search; 12.5% Insert; 12.5% Delete

Update intensive mode: 10% Search; 45% Insert; 45% Delete

We run the benchmark in different experimental settings with the number of threads varying from 1 to 64. Figures 5.12 and 5.13 show the test results of the three algorithms in the search intensive mode with a small workload and a large workload respectively. Figures 5.14 and 5.15 show the test results of the three algorithms in the update intensive mode with a small workload and a large workload respectively.

As seen from all the four figures, our optimized algorithm has slightly improved performance over the Lazy-tree algorithm through all four different experimental settings and both the Optimized and Lazy-tree algorithms considerably overtake the performance of the Global-rwLock algorithm. This is reasonable because the Optimized and Lazy-tree both use fine-grained locking strategies which allow concurrent manipulation on the disjoint parts of

the LCRS tree. Comparatively, the Global-rwLock simplifies the algorithm but restricts the concurrency of operations, and the global reader-writer lock tends to become the bottleneck of the algorithm when the number of threads increases.

Noteworthy, the throughput of small workload scenarios (Figures 5.12 and 5.14) shows the same trend in which the throughput grows as the number of threads increases, but reaches a peak at a point and then drops gradually. This is understandable as the number of the threads is close to the number of cores in the multi-core system, our algorithm can better explore the advantage of multicore system. As seen from Figures 5.12 and 5.13, the point of performance peak moves from 4 to 64 working threads in the search intensive mode, while the thread number of the performance peak point moves from 16 to 64 as seen from Figures 5.14 and 5.15. Both cases verify that a larger workload of the program could benefit more from hyper threading, which increases throughput when the number of threads exceeds the number of virtual cores. In the search intensive mode where the majority of the operations are searching, the throughput reaches the peak when the number of threads is 4, as compared to 16 threads in the update intensive mode when the throughput reaches the peak. The throughput of large workload scenarios (Figure 5.13 and Figure 5.15) also shows the same trend, which is different from the small workload scenarios. In the large workload scenarios, the throughput largely overtakes the throughput of small workload scenarios, and the throughput keeps growing as the number of threads increases.

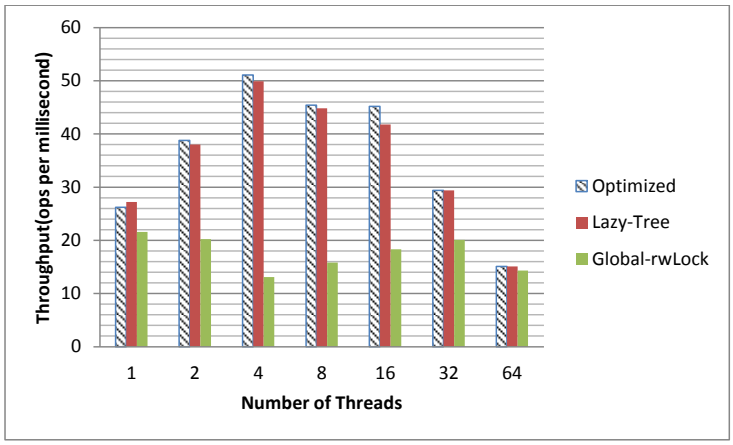


Figure 5.12: Small workload in the search intensive mode

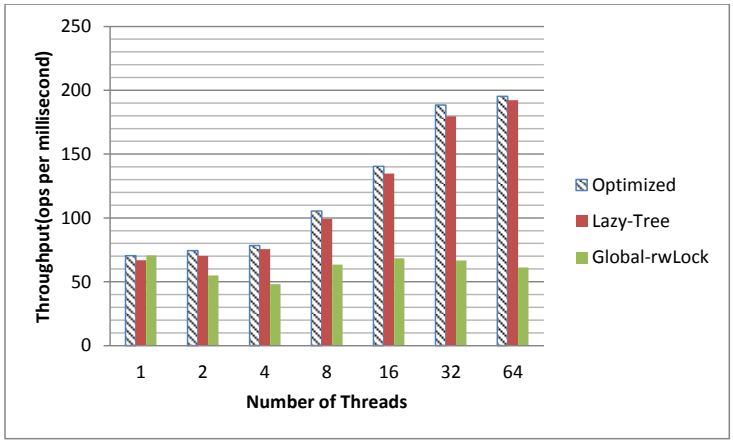


Figure 5.13: Large workload in the search intensive mode

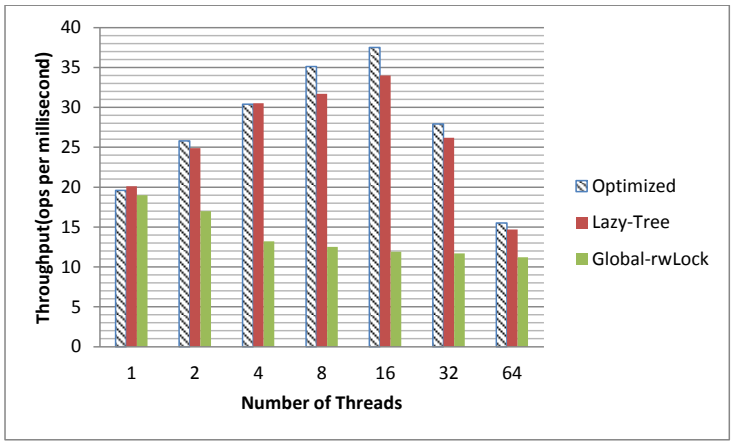


Figure 5.14: Small workload in the update intensive mode

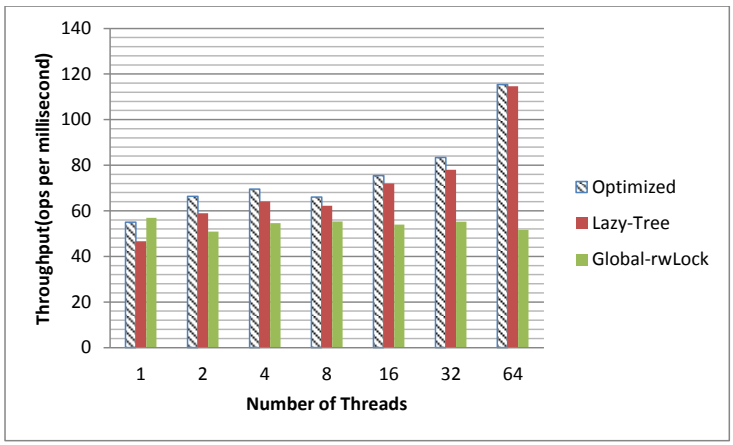


Figure 5.15: Large workload in the update intensive mode

In addition, noticeably the performance of the three algorithms does not differ too much when there is only one thread working. In this case, all the operations are executed in a sequential order. When the number of threads grows, our optimized algorithm and Lazy-tree algorithm gain more concurrency in updating the LCRS tree, therefore the throughput is improved. Figure 5.15 shows that the throughput improvement of Optimized and Lazy-tree over the Global-rwLock is not as much as the other three experiments. The reason is that the high percentage of update operations in the update intensive mode requires more costly locks and higher contention rates can cause validation failure and thus increase the overall overhead.

Overall, our optimized algorithm improves the throughput up to 8% over the Lazy-tree algorithm in the search intensive mode, and 12% in the update intensive mode. Additionally the performance of our optimized algorithm over Global-rwLock algorithm is up to 3 times better.

5.2 Multi-Granularity Locking Scheme based on the Concurrent Multiway Tree

In the last section, we introduced a concurrent algorithm for the multiway tree which represents the containing or ordering relationship among different nodes in the left-child right-sibling tree, and then we also analyzed and evaluated the algorithm. In this section, we present a new efficient approach for multi-tier data synchronization based on the concurrent multiway tree

and then analyze the algorithm and evaluate the performance of the tree-based approach by comparing it with the list-based approach as presented in Chapter 4.

5.2.1 Technical Approach

In order to keep the simplicity of our approach to multi-granularity locking on PackedObjects, we incorporate the insert and search operations for the concurrent multiway tree to our approach which dynamically grows the tree of *PackedMonitorInfo* during the synchronization of derived PackedObjects from the same Packed Region. In order to distinguish from the concurrent linked-list-based approach *MultitierPackedSync* which is present in Chapter 4, we name our new approach as Tree-based MultitierSync, as compared to the List-based MultitierSync for the approach *MultitierPackedSync*.

5.2.1.1 Auxiliary Data Structures

Our approach to multi-tier data synchronization is based upon the Java packed object data model, in which the data of a PackedObject can be nested in another PackedObject. In order to synchronize on the multi-level nested PackedObjects (Multi-tier PackedObjects), we should be able to identify the packed data being synchronized from different tiers and maintain data integrity. We use the structure *PackedMonitorInfo* to keep the information related with each packed data region being synchronized. In order to keep track of the *PackedMonitorInfo* in a concurrent multiway tree, which rep-

resents the containing or ordering relationship among the packed data in different tiers, we re-define the structure *PackedMonitorInfo* as:

```
typedef struct PackedMonitorInfo{
        struct Key{ UDATA offset; UDATA packedLen;} key;
        j9thread_t owner;
        UDATA count;
        U_8 lockState;//0,1,2
        j9thread_monitor_t syncMonitor;
        struct PackedMonitorInfo *next;
        struct PackedMonitorInfo *childList;
        struct PackedMonitorInfo *parent;
        j9thread_monitor_t mutex;
}PackedMonitorInfo;
```

In the structure PackedMonitorInfo, offset and packedLen represent the offset and the length of the packed object data and are defined in the key structure which identifies the data region being guarded by the *PackedMonitorInfo*. The structure key in *PackedMonitorInfo* is corresponding to the key of the concurrent multiway tree that identifies the data range from key.begin (offset) to key.end (offset+packedLen). owner is a pointer to the thread that is currently locking on the data region identified by key. Our approach supports recursive locks which allow the same thread to lock on the same packed data more than once. count indicates how many times the thread owner enters

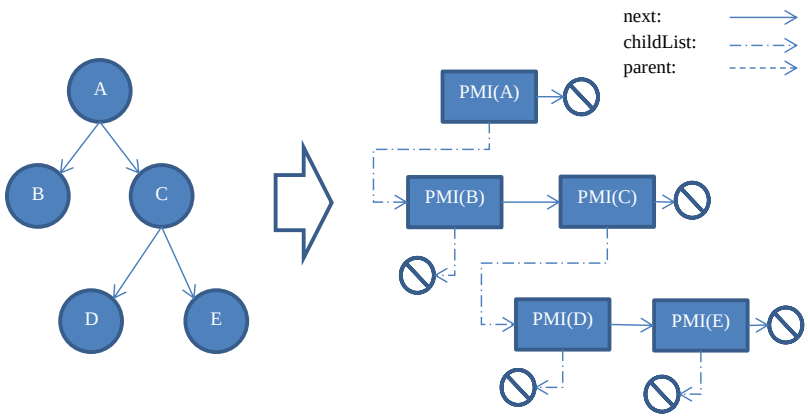


Figure 5.16: A LCRS tree representation for the *PackedMonitorInfo* of the three-tier PackedObject

this *PackedMonitorInfo*.

Figure 5.16 shows the left-child right-sibling tree representation for the *PackedMonitorInfo* for the three-tier PackedObject A defined in Section 4.1. The parent pointers are not shown in the figure. The tree structure on the left of Figure 5.16 shows the containing relationship (represented by the arrows) among different nodes.

In order to synchronize on the multi-tier PackedObject, we define lockState to indicate three types of locking status for the PackedObjects. They are defined as:

```
#define LS_DIRECT      2//Explicit
#define LS_INDIRECT    1//Implicit
#define LS_UNLOCKED    0
```

LS_DIRECT indicates that the packed data region identified by *PackedMonitorInfo*.key is being locked explicitly, as opposed to *LS_INDIRECT* which indicates the packed data region identified by *PackedMonitorInfo*.key is being locked implicitly. *LS_UNLOCKED* indicates the default status of the packed data region. As shown in Figure 5.16, all the PMIs' default lock states are set as *LS_UNLOCKED* after the *PackedMonitorInfo* is created and inserted into the *PackedMonitorInfo* tree. If a thread invokes Packed-Lock on Packed-Object $A.C$, then $A.C$ is considered as being directly locked, thus PMI(C)->lockState is changed to *LS_DIRECT*. As C contains two other packedObjects D and E which are considered as being indirectly locked by the same thread, their lock states are supposed to be updated to *LS_INDIRECT* before the thread eventually obtains the lock on C.

The field syncMonitor is used for mutually exclusive access to the three mutable fields: owner, count and lockState that represent the locking status of the *PackedMonitorInfo*. mutex is used for mutual exclusively updating the multiway tree structure.

In addition, each Packed Region has a ContainerInfo associated with it. The structure ContainerInfo has a pointer PMTree pointing to the multiway *PackedMonitorInfo* tree to keep track of the synchronization information for the Packed Region. treeMutex is used to make sure the PMTree is updated mutual exclusively. The structure ContainerInfo is redefined as:

```
typedef struct ContainerInfo{
        UDATE container;
        PackedMonitorInfo * PMTree;
        J9thread_monitor_t treeMutex;
} ContainerInfo;
```

5.2.1.2 Synchronization Interfaces

Our approach to multi-tier data synchronization provides mutual exclusive access to the multi-tier data by PackedLock and PackedUnlock, which traverses the left-child right-sibling tree PMTree that is associated with the multi-tier packedObject and searches for the corresponding *PackedMonitorInfo* for the data to be synchronized and then carries out the locking/unlocking operations. Like the List-based MultitierSync, there are six cases to be addressed when the methods PackedLock and PackedUnlock are traversing the PMTree to look for the corresponding *PackedMonitorInfo* for synchronization. The six cases are corresponding to the cases defined in Section 5.1.2.1 for the Search method of concurrent multiway tree (the key k identifies the data range on the packedObject to be synchronized; *curr* points to the current node of the PMTree being traversed; *pred* points to the predecessor node of the current node pointed to by *curr*. *pred* is NULL if *curr* points to the root of the PMTree; isLeftChild indicates if *curr* is the leftChild of *pred* and is initialized as false):

1. *curr* is NULL, which means traversal reaches the end of the tree. If

pred is NULL, the PMTree is NULL. Otherwise, either pred->next or pred->childList is NULL which is pointed to by *curr*.

2. k > curr->key, which means the data range identified by k is located after the data range identified by the current node in PMTree being traversed. In this case, continue to traverse to the next sibling node of the current node.

3. k < curr->key, which means the data range identified by k is located before the data range identified by the current node in PMTree being traversed.

4. k == curr->key, which means the current node in PMTree being traversed is found as the target node for the data range identified by k.

5. k $\subset$ curr->key, which means the data range identified by k is the sub-region of the data range identified by the current node in PMTree being traversed. In this case, continue to traverse to the first child node of the current node.

6. curr->key $\subset$ k, which means the data range identified by k is the containing-region of the data range identified by the current node in PMTree being traversed.

The flowchart in Figure 5.17 shows the six cases during the traversal of the PMTree.

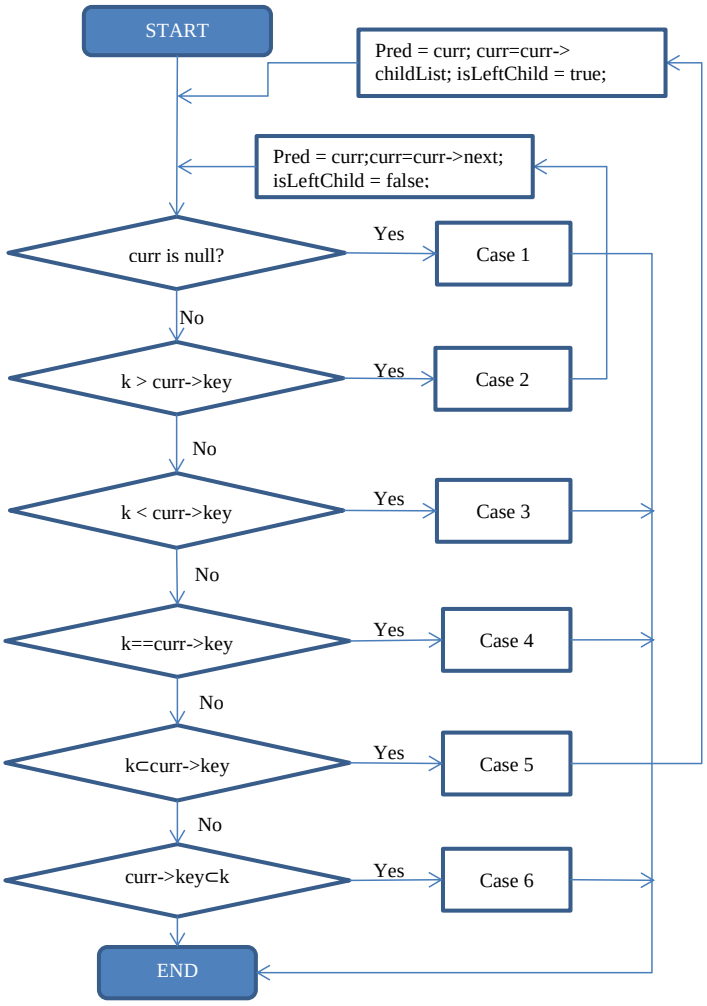


Figure 5.17: Flowchart of the Traversal of PMTree

PackedLock:

A thread calls PackedLock to lock on a PackedObject *PO* whose data region is identified by a key k. The method PackedLock first calls function getPackedContainerInfo(PO) (described in Section 3.3.3) to get the associated containerInfo which has a link to the multiway *PackedMonitorInfo* tree PMTree, and then traverses the PMTree to find the corresponding *PackedMonitorInfo* for *PO* for the locking operation.

When the traversal encounters Case 1 or Case 3, in which the target *PackedMonitorInfo* is not found in the PMTree, a simple insertion operation for the *PackedMonitorInfo* for *PO* is to be carried out. The simple insertion operation for *PackedMonitorInfo* is similar as the simple insertion for the concurrent multiway tree described in Section 5.1.2.2. The only difference is, before the simple insertion is completed, if the newly inserted *PackedMonitorInfo* (let newPMI represent it) has a parent node, newPMI's lock state should be set as consistent with its parent, which means if the newPMI->parent->lockState is *LS_DIRECT*, newPMI->lockState should be set to *LS_INDIRECT*, otherwise, newPMI->lockState is set to be the same lock state as newPMI->parent->lockState. After the insertion is completed, the locks for the tree updating are released. The method PackedLock then acquires newPMI->*syncMonitor* and updates the relevant information for the locking operation. Figure 5.18 shows the flowchart of the PackedLock after the simple insertion.

After the simple insertion is completed, the thread acquires the *syncMonitor*

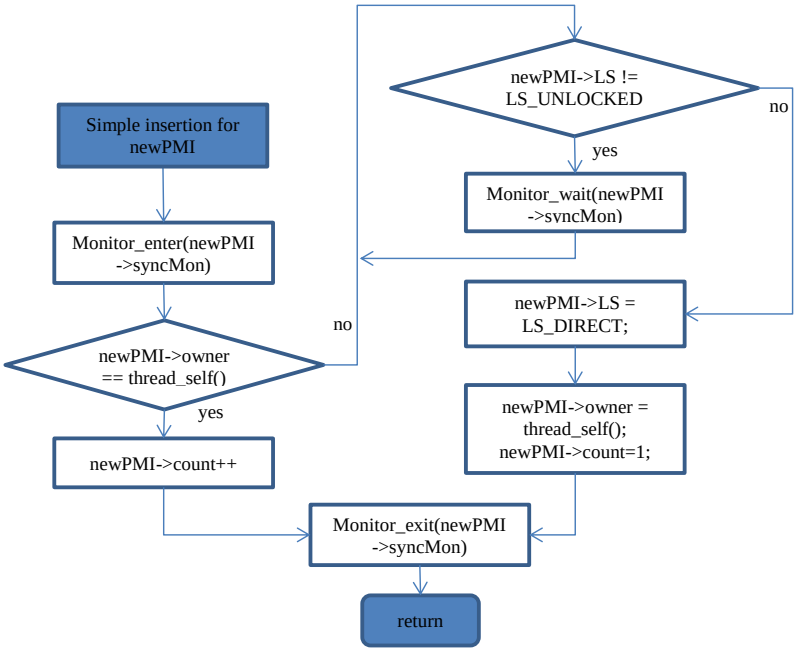


Figure 5.18: Flowchart of PackedLock after the simple insertion

and checks if itself is the owner of the *PackedMonitorInfo* newPMI. If so, it increases the newPMI->count for recursive locking, and then releases the *syncMonitor*. Otherwise, the thread checks if newPMI is unlocked. If not, the thread waits until other threads release the lock on the newPMI and then updates its lockState to *LS_DIRECT* and sets other related information. Once all the updates are completed, the thread releases *syncMonitor*.

When the traversal encounters Case 2, the method PackedLock continues traversing to the next sibling node of *curr*.

When the traversal encounters Case 4, the target *PackedMonitorInfo* for *PO* is found in the PMTree pointed to by *curr*. Figure 5.19 shows the flowchart of PackedLock in Case 4.

In this case, the update of the target *PackedMonitorInfo* is the same as the operations after simple insertion. Once the updates are completed, the thread needs to recursively traverse all the descendants of the target *PackedMonitorInfo* and set their lock states to be *LS_INDIRECT* by calling function TraverseAndSetLockState(). The flowchart of function TraverseAndSetLockState() is shown in Figure 5.20.

In function TraverseAndSetLockState(), if the thread encounters a descendant *PackedMonitorInfo* that is directly locked by other threads, the thread waits until other threads release the lock on the descendant *PackedMonitorInfo* before continuing the rest of the updates. The traversal in the function TraverseAndSetLockState() is a pre-order traversal which traverses the root of the tree before recursively traversing the left sub-tree and then the right

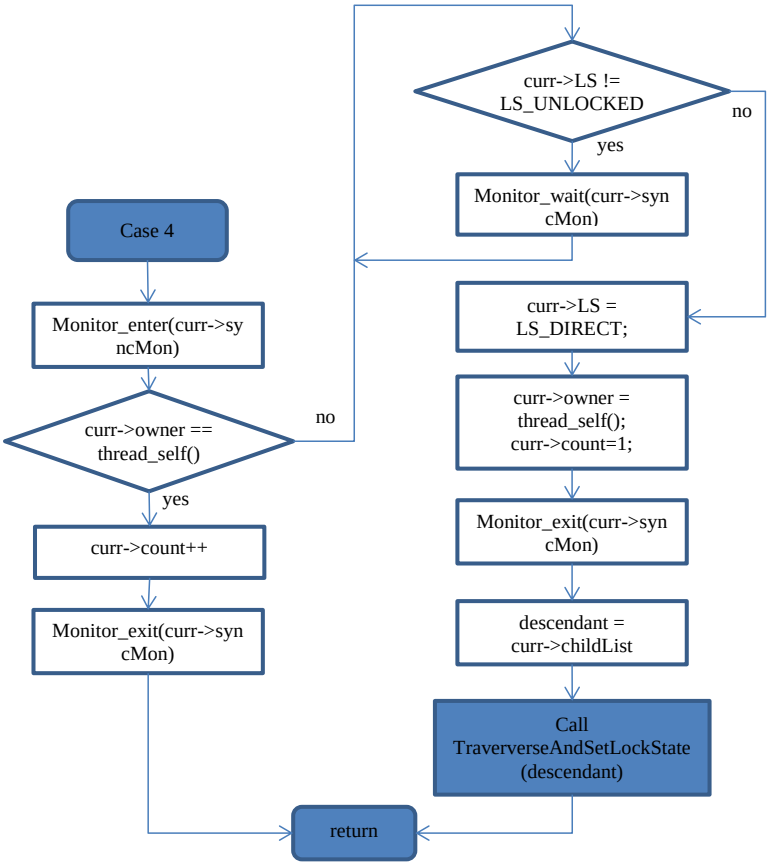


Figure 5.19: Flowchart of PackedLock in case 4

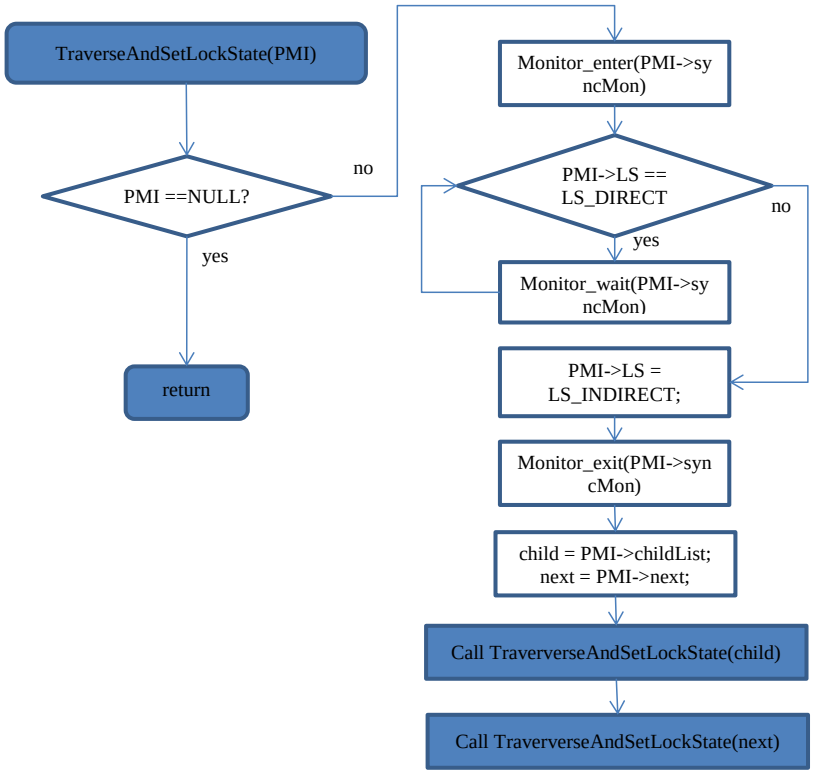


Figure 5.20: Flowchart of function TraverseAndSetLockState()

sub-tree.

When the traversal encounters Case 5, the data region identified by the current node in PMTree is found as the containing-region of PackedObject *PO*. In this case, it continues to traverse to the first child node of *curr*. Before continuing traversal to the child node of curr, if the current PackedMonitorInfo's lock state is *LS_DIRECT*, the thread blocks and waits until other threads release the lock on the current *PackedMonitorInfo*.

When the traversal encounters Case 6, the data region identified by the current node in PMTree being traversed is found as the sub-region of PackedObject *PO*. In this case, a complex insertion operation for the *PackedMonitorInfo* for *PO* is to be carried out. Likewise, the complex insertion operation for *PackedMonitorInfo* is the same as the complex insertion of the concurrent multiway tree except that the newly inserted *PackedMonitorInfo* newPMI's lock state has to be updated to a consistent state before the insertion is completed. After the complex insertion is completed, the method PackedLock acquires *syncMonitor* and updates the relevant information for the locking operation. Figure 5.21 shows the flowchart of PackedLock after the complex insertion.

The operations after the complex insertion for the newly added *PackedMonitorInfo* newPMI are the same as the operations in Case 4. After the insertion is completed, the thread updates the newPMI->lockState and other information accordingly and then invokes the function TraverseAndSetLockState() to traverse down to all the descendants of newPMI recursively and updates

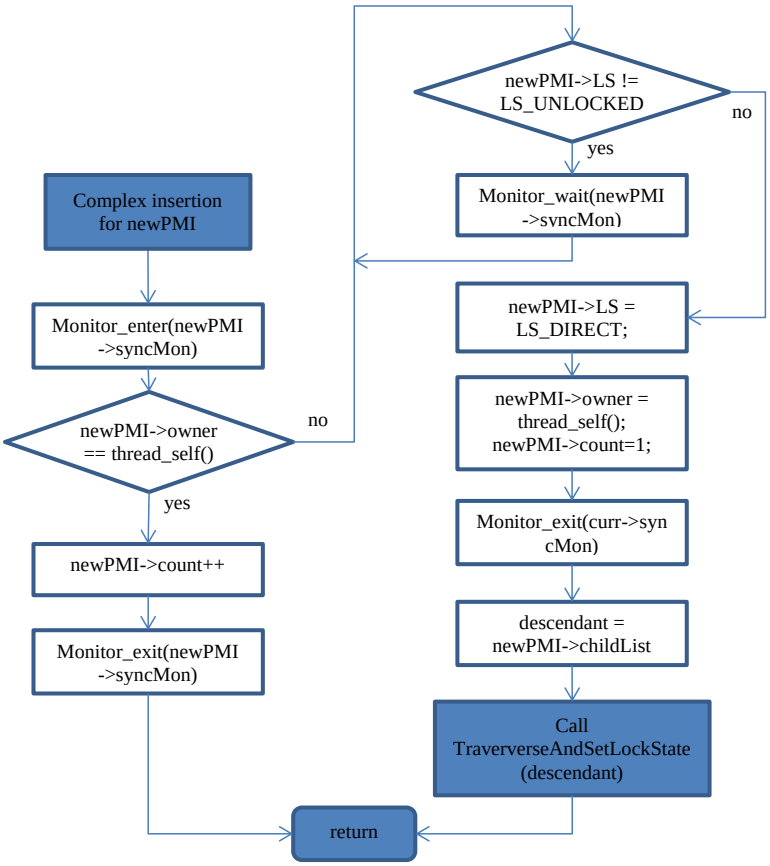


Figure 5.21: Flowchart of PackedLock after the complex insertion

their lock states to *LS_INDIRECT*.

PackedUnlock:

A thread calls PackedUnlock to unlock on a PackedObject *PO* whose data region is identified by a key k. Likewise, the method PackedUnlock calls function getPackedContainerInfo(PO) to get the associated containerInfo that has a link to the multiway *PackedMonitorInfo* tree PMTree, and then tra-

verses the PMTree to find the corresponding *PackedMonitorInfo* for *PO* for the unlocking operation.

When the traversal encounters Case 1 or Case 3, the target *PackedMonitorInfo* is not found in the PMTree. Therefore, PackedUnlock throws a MONITOR_NOT_FOUND exception.

When the traversal encounters Case 2, the method PackedUnlock continues traversing to the next sibling node of *curr* as PackedLock does.

When the traversal encounters Case 4, the target *PackedMonitorInfo* for *PO* is found in the PMTree pointed to by *curr*. Figure 5.22 shows the flowchart of PackedUnlock in Case 4. In this case, the method PackedUnlock acquires curr->*syncMonitor* and then checks if the current *PackedMonitorInfo* is being locked directly. If not, PackedUnlock throws an ILLEGAL_MONITOR_STATE exception. Otherwise, the method decreases curr->count by 1. If curr->count is not 0 after decrement, the current thread is still holding the lock on the current PackedMonitorInfo, and then it simply exits curr-> *syncMonitor* and returns. If curr->count equals 0 after decrement, the method updates the curr->owner to NULL indicating that the thread releases the lock on the current *PackedMonitorInfo*. Then the method calls TraverseAndClearLockState() to recursively traverse down to the descendant nodes of the current *PackedMonitorInfo* and set their lock states to *LS_UNLOCKED*. Once the descendant nodes' lock states are cleared, the method sets the curr->locksState to *LS_UNLOCKED* and notifies other threads waiting for the lock on the current *PackedMonitorInfo* to be released.

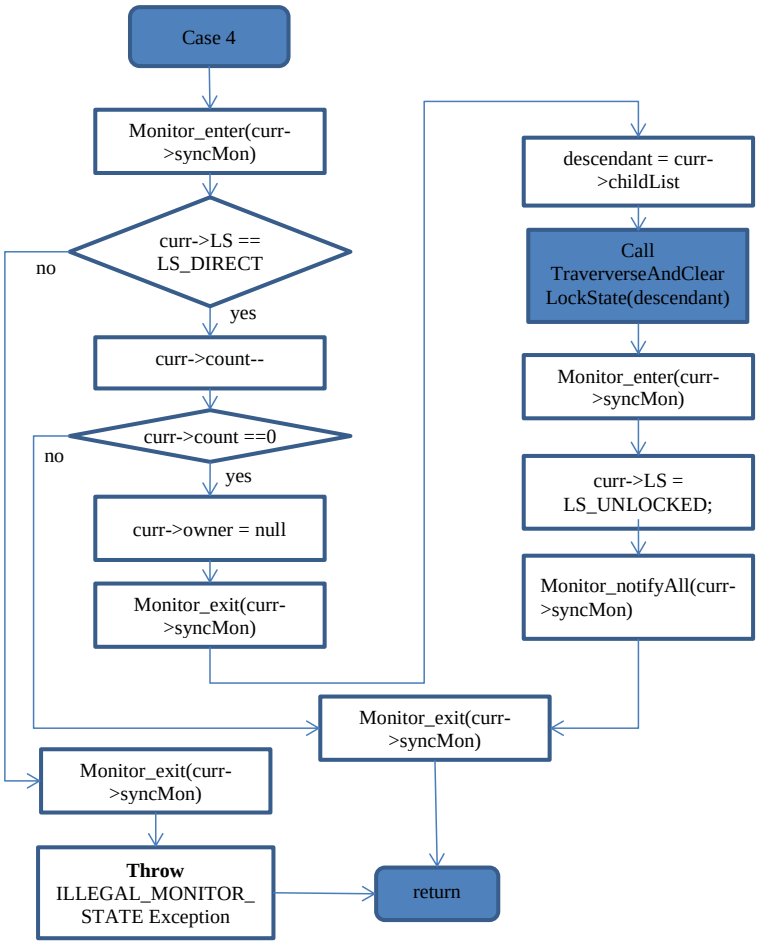


Figure 5.22: Flowchart of PackedUnlock in Case 4

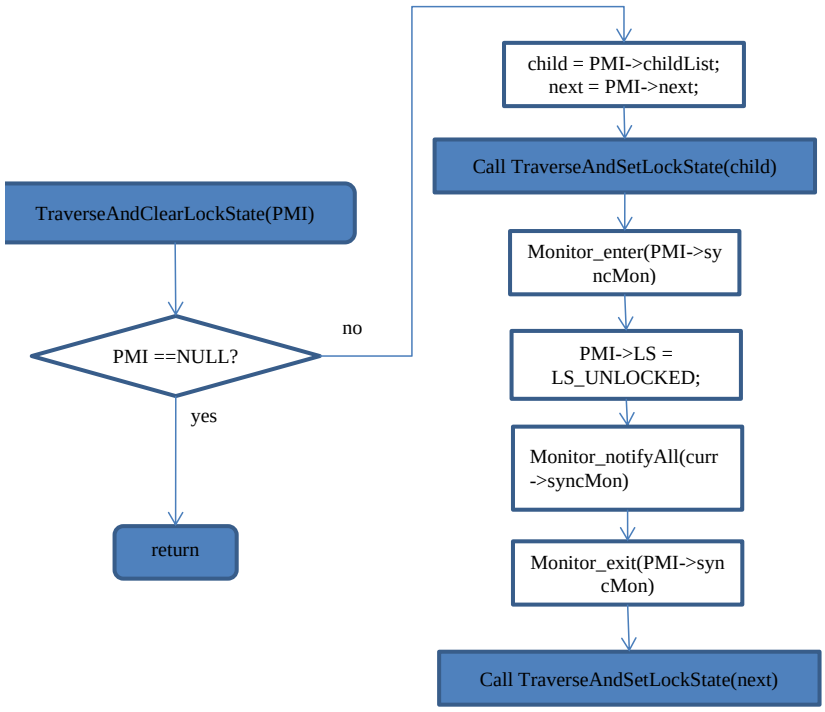


Figure 5.23: Flowchart of function TraverseAndClearLockState()

The flowchart of function TraverseAndClearLockState() is shown in Figure 5.23. The function TraverseAndClearLockState() recursively traverses down to the descendant *PackedMonitorInfo* and sets their lock states to the initial state: *LS_UNLOCKED* and notifies other threads waiting for the implicit lock on the descendant *PackedMonitorInfo* to be released. The traversal in the function TraverseAndClearLockState() is an in-order traversal which recursively traverses the left sub-tree before traversing the root of the tree and then the right sub-tree.

When the traversal encounters Case 5, in which the data region identified

by the current node in PMTree is found as the containing-region of Packed-Object *PO*. In this case, PackedUnlock continues traversal to the first child node of *curr* directly.

When the traversal encounters Case 6, the data region identified by the current node in PMTree being traversed is found as the sub-region of Packed-Object *PO*. It is not possible for a thread to lock on a containing-region (PO) and then unlock from its sub-region. Therefore, PackedUnlock throws an ILLEGAL_MONITOR_STATE exception.

5.2.2 Algorithm Analysis

In this section, we analyze how our algorithm can work properly with the interleaving of concurrent updates of the tree structure and monitor information and demonstrate that our algorithm is linearizable by defining a linearization point for each operation. In order to verify the correctness of our algorithm, we examine all the possible scenarios for multi-granularity locking with the concurrent multiway tree. The scenarios can be categorized as two types: static tree scenario and growing tree scenario.

In the static tree scenario, the tree structure is kept in a static state in which no insertion of tree nodes is taking place and the tree structure remains unchanged. In this scenario, we only need to care about the conflict of updating monitor information. The update conflicts happen whenever one thread is attempting to update a *PackedMonitorInfo* P and another thread is attempting to update one of P's direct ancestors. In such cases, their

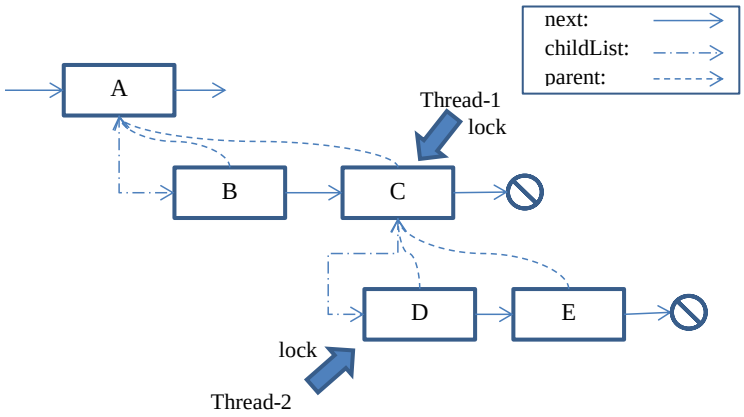


Figure 5.24: Static Tree Scenario

operations should be serialized. Otherwise, their updates can proceed in parallel. Figure 5.24 shows an example of the static tree scenario, in which a thread attempts to lock on PackedMonitorInfo C, while another thread attempts to lock on one of C's descendants.

In our algorithm, a *PackedMonitorInfo* can be updated either by the direct locking threads or indirect locking threads. A thread locking on a *PackedMonitorInfo* has to update the lock states of the direct locking *PackedMonitorInfo* and all of its descendants which are indirectly locked by the same thread (e.g. Thread-1 sets the direct locking PMI C's lock states to *LS_DIRECT* and sets D and E's lock states to *LS_INDIRECT*). The conflict updates can be serialized in a way that the thread blocks and waits for other threads to release the lock on a *PackedMonitorInfo* if it finds the *PackedMonitorInfo* is being locked by other threads directly or indirectly. Therefore, the linearization point for a successful locking operation is at the

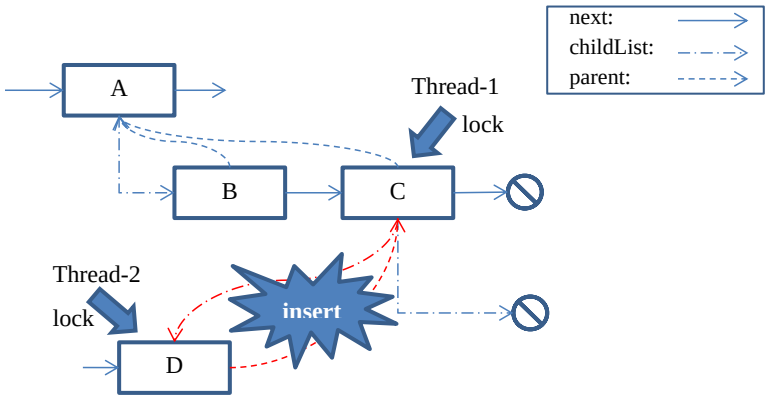


Figure 5.25: An example for case 1 of the growing tree scenario

point when the thread finishes updating the lock states of all the descendants of the direct locking *PackedMonitorInfo* or the lock state of the direct locking *PackedMonitorInfo* if it has no descendants.

In the growing tree scenario, the tree structure is in a dynamically growing state in which the insertion of tree nodes is taking place concurrently. The concurrent insertion operations for the nodes of a multiway tree have been addressed in Section 5.1. In this scenario we only consider the interleaving of updating *PackedMonitorInfo* and inserting *PackedMonitorInfo* to the tree. There are two basic cases for this scenario. All cases requiring concurrent manipulation in the growing tree scenario can be categorized into one of the two cases:

Case 1: Inserting a *PackedMonitorInfo* whose ancestor is concurrently being locked by another thread. Figure 5.25 shows an example of this case.

As shown in Figure 5.25, Thread-1 is attempting to lock on a PMI C in the

PMTree, while Thread-2 is attempting to lock on C's sub-PMI D which is not found in the PMTree, thus thread-2 has to insert D into the PMTree as C's child before locking on D. Thread-1 attempting to lock on C has to acquire the syncMonitor and then update C's lock state and other relevant information, while thread-2 attempting to insert D to C's childList has to acquire the predecessor (which is C)'s mutex for the simple insertion operation. Since the two threads acquire two different monitors for updating the *PackedMonitorInfo* and the tree structure respectively, their operations can proceed concurrently without interfering each other. The contention occurs at the updating of *PackedMonitorInfo*. If thread-1 updates C->lockState to *LS_DIRECT* before thread-2 inserts D to C->childList, then D->lockState will be set to *LS_INDIRECT* to keep consistent with D->parent. Thread-2 continues to lock on D after D is inserted. Since D is indirectly locked by thread-1, thread-2 has to wait until thread-1 releases the lock on C and update D->lockState back to *LS_UNLOCKED*. If thread-2 inserts D to C->childList before C->lockState is set to *LS_DIRECT*, then D's initial lockState is *LS_UNLOCKED*. Thread-2 then continues to lock on D after D is inserted. Thread-1 will update C->lockState to *LS_DIRECT* and recursively traverse its descendants at the same time. If thread-1 happens to acquire D->syncMonitor before thread-2 does and update D->lockState to *LS_INDIRECT*, thread-2 has to wait until thread-1 release the lock on C and update D->lockState back to *LS_UNLOCKED*. Otherwise, thread-2 proceeds before thread-1 and sets D->lockState to *LS_DIRECT*, then thread-1 has to

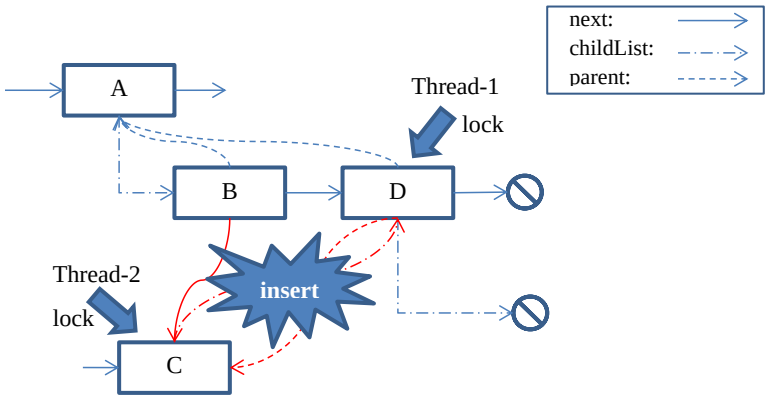


Figure 5.26: An example for Case 2 of the growing tree scenario

wait for thread-2 to release the lock on D before it continues the traversal of C's descendants. Therefore, the linearization point for a successful locking operation for thread-2 is at the point when it sets D->lockState to *LS_DIRECT*, while a successful locking operation for thread-1 is at the point when it finishes updating all the descendants' lock states to *LS_INDIRECT*.

Case 2: Inserting a *PackedMonitorInfo* whose descendant is concurrently being locked by another thread. Figure 5.26 shows an example of this case. As shown in the figure, thread-1 is attempting to lock on a *PackedMonitorInfo* D in the PMTree, while thread-2 is attempting to lock on D's containing-PMI which is not found in the PMTree. Thus, thread-2 attempts to insert C into the PMTree as D's parent. Likewise, the complex insertion for C can be carried out concurrently with the update of *PackedMonitorInfo* D since thread-1 and thread-2 are acquiring two different monitors for their operations respectively. The contention occurs at the updating of the *PackedMon-*

itorInfo. If thread-1 happens to acquire D->*syncMonitor* and updates its lockState to *LS_DIRECT* before thread-2 does, thread-2 will block during the recursive traversal to C's descendants and wait until thread-1 releases D's lock by updating D->lockState to *LS_UNLOCKED*, and then thread-2 continues traversal and sets the descendants' lockState to *LS_INDIRECT*. If D has descendants, thread-1 will recursively traverse all the descendants and set their states to *LS_INDIRECT*. Therefore, a linearization point for a successful locking operation for thread-1 is at the point when it updates D's lockState to *LS_DIRECT* if D has no descendants. Otherwise, the linearization point is when Thread-1 finishes updating all the descendants' lockState to *LS_INDIRECT*. Likewise, the linearization point for a successful locking operation for thread-2 is at the point when it finishes updating all the descendants' lock states to *LS_INDIRECT*.

For the unlocking operation, the method PackedUnlock traverses PMTree and searches for the target PMI whose key identifies the data region being synchronized. Once the target PMI is found in PMTree, the method PackedUnlock recursively traverses all the descendants and clears their lock states (change the lock state from *LS_INDIRECT* to *LS_UNLOCKED*), and set the target PMI's lockState to *LS_UNLOCKED* at last and then notifies other threads waiting for the lock on the target PMI to be released. The linearization point for a successful unlocking operation is at the point when the target PMI's lockState is set to *LS_UNLOCKED*. A linearization point for a failed unlocking operation is at the point when the target PMI is not

found in PMTree.

In addition, as mentioned in Section 5.2.1.2, the method PackedLock calls TraverseAndSetLockState() to recursively traverse the descendants in pre-order and set the lockStates to *LS_INDIRECT*, while the method PackedUnlock calls TraverseAndClearLockState() to recursively traverse the descendants in in-order and change the lockStates from *LS_INDIRECT* to *LS_UNLOCKED*. The traversal orders for the two functions help avoid inconsistent locking states for the PMTree. Figure 5.27 shows an example that an inconsistent locking state occurs if the traversal order for recursive traversal of the descendants is not as we designed.

As shown in Figure 5.27, Thread-1 is directly locking on A, it traverses to A's descendants and blocks at C since C is directly locked by thread-2. Once thread-2 finishes its work on the data guarded by C, it calls PackedUnlock to release the lock on C. If C's lockState is cleared before C->childList, which disobeys the in-order traversal for clearing lock states, then thread-1 is woken up at C and it sets C->lockState to *LS_INDIRECT* and continues to traverse to C's childList. In a concurrent context, it is possible that thread-1 traverses D before thread-2 clears D's lockState. In such a scenario, thread-1 mistakenly considers that D's lockState is already set to *LS_INDIRECT* and continues to traverse down to the other descendants, while thread-2 later visited D and cleared D's lockState. Therefore, an inconsistent locking state occurs as shown in Figure 5.27.

By keeping the pre-order traversal for setting lock states and in-order traver-

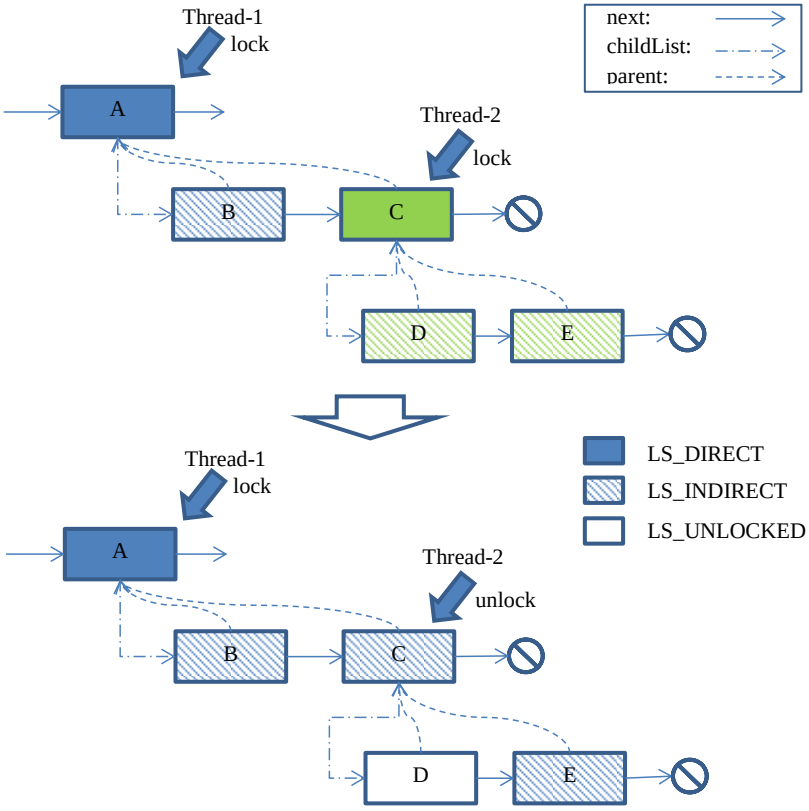


Figure 5.27: An inconsistent locking State caused by wrong traversal order

sal for clearing lock states, when thread-1 is woken up by thread-2, C and all its descendants' lockStates have been cleared, thus inconsistent locking states can be avoided in our algorithm. In addition, a pre-order traversal for setting lock states blocks other threads attempting to lock on the conflicting data region at a higher level of PMTree and keeps the simplicity of our algorithm.

5.2.3 Evaluation and Analysis

In this section, we evaluate our Tree-based MultitierSync approach by comparing it with the List-based MultitierSync approach (MultitierPackedSync) as presented in Chapter 4. Likewise, our experiments were carried out on a 1.8GHz 16 Core/32 Thread Xeon, 4xE7520, Nehalem-based machine with 64 GB of RAM, running Linux CentOS. The algorithms were implemented in the IBM J9 Virtual Machine, and the synchronization interfaces PackedLock and PackedUnlock for mutual exclusion operations were provided. In our benchmarks, we measured the two approaches (List-based and Tree-based MultitierSync) in a multi-granularity locking scenario in terms of the total execution time and throughput (the number of operations completed per millisecond over the course of the entire workload).

In the benchmark defined in Section 4.3 for multi-granularity locking, the same number of threads was assigned to each tier of the three-tier packed-Object array. Therefore, the total number of operations is proportional to the number of threads. In this evaluation, we adjusted the benchmark to

assign a fixed number of tasks evenly to different numbers of threads and let them randomly update the three-tier PackedObject array (same as defined in Section 4.3) in random tiers. Likewise, we set the array size to 128 and vary the number of threads from 1 to 64 and run the benchmark in different workloads:

1. **Small workload**: nTask = arraySize*nTier = 128*3=384.

2. **Large workload**: nTask = arraySize*nTier*10 = 3840.

Our approaches are validated through the benchmark programs which yield the same results under the same experimental setting after the threads finish their tasks. In our benchmark experiments, we calculated the average results across 100 runs. The standard error of mean ranged from 1% to 5% of the average results which is small enough that, therefore, we did not include error bars in the graphs. Figures 5.28 and 5.29 respectively show the total execution time and throughput of the two approaches Tree-based and List-based MultitierSync under a small workload, while Figures 5.30 and 5.31 respectively show the total execution time and throughput of the two approaches under a large workload.

As seen from Figure 5.28 through Figure 5.31, the performance of the Tree-based MultitierSync approach significantly overtakes the List-based approach in both the small workload and the large workload scenarios. In the small workload scenario, the total execution time reaches the lowest point when the number of threads is 8. The throughput in Figure 5.29 shows the same trend

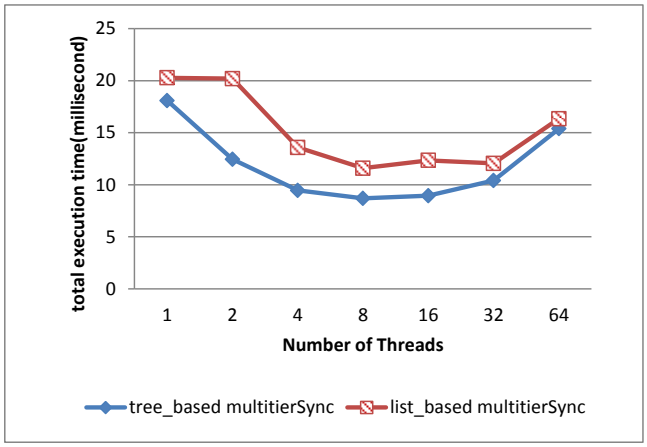


Figure 5.28: Time expense comparison in a small workload (nTask =arraySize*nTier)

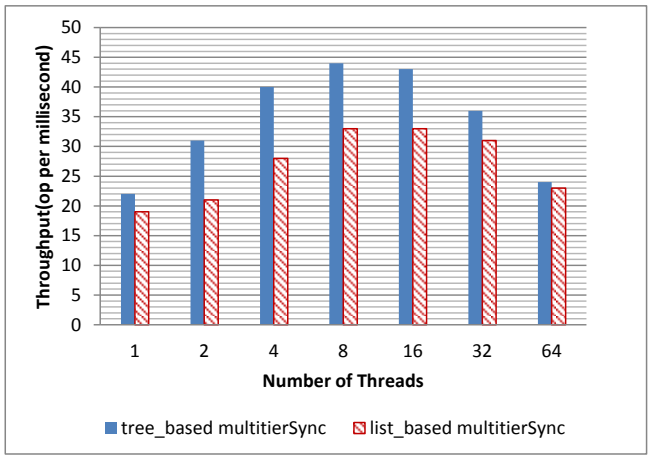


Figure 5.29: Throughput comparison in a small workload (nTask =arraySize*nTier)

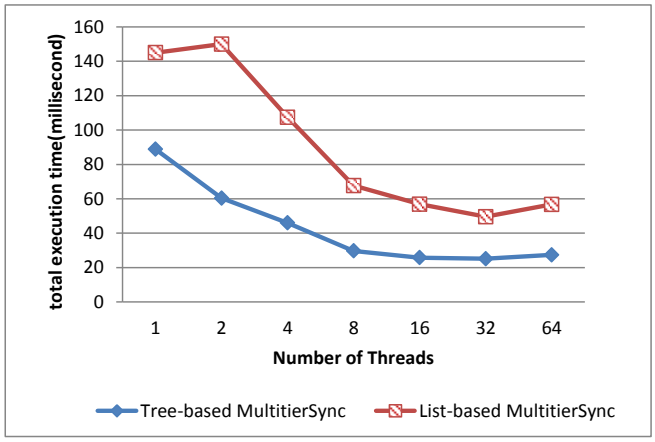


Figure 5.30: Time expense comparison in a large workload (nTask =arraySize*nTier*10)

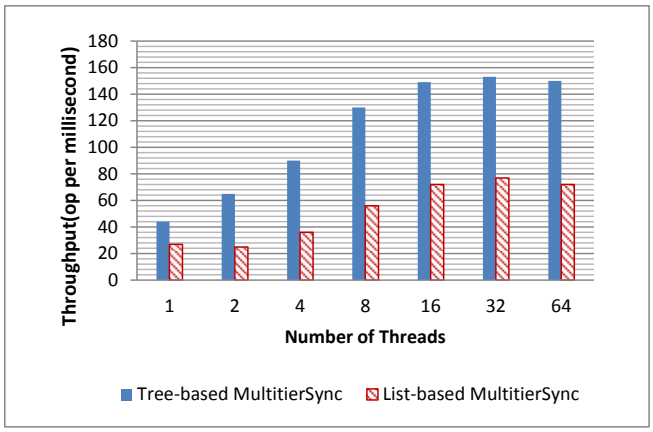


Figure 5.31: Throughput comparison in a large workload (nTask =arraySize*nTier*10)

as Figure 5.28. Comparatively the throughput reaches a peak as the number of threads is 8. This experimental results show that our algorithms achieve the best performance when the number of the working threads is close to the number of core of the system that our benchmark was running on. For a large workload scenario which has 10 times the workload as the small workload scenario, the best performance was achieved when the number of threads is 32. The throughput stopped growing but dropped after a certain number of threads. Therefore, simply increasing the number of working threads does not necessarily mean the performance will be improved all the time. As seen from Figures 5.28 and 5.29, the performance gap between the tree-based and list-based approaches is shrinking as the number of threads goes up. This is due to the contention rate raising and the overall overhead is increased accordingly.

By comparing the performance between the small workload scenario and the large workload scenario, we can tell that the performance gap between the two approaches is enlarged in the large workload scenario. In order to see the impact of the workload on the overall performance, we re-ran the benchmark with 64 working threads and varied the number of tasks from 1 to 30 times that of the small workload (nTask = arraySize*nTier). The results are shown in Figure 5.32 and 5.33.

We can see clearly from Figure 5.32 that the time expense for the two approaches is almost linearly proportional to the amount of workload. As the number of tasks increases, the Tree-based approach improves the perfor-

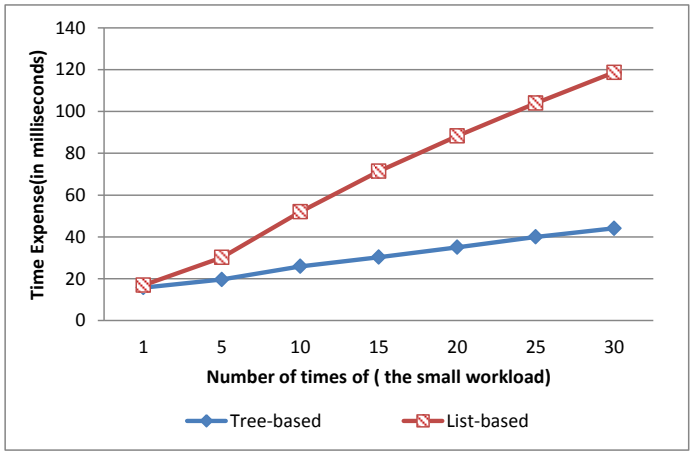


Figure 5.32: The time expense comparison in different workload as the working thread number is 64

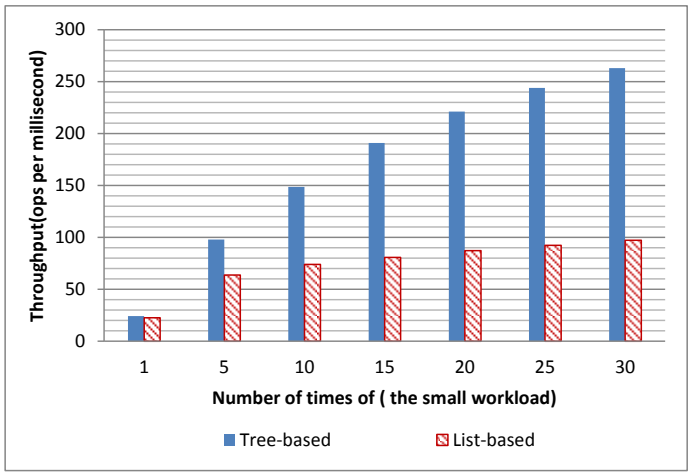


Figure 5.33: Throughput comparison in different workload as the working thread number is 64

mance significantly over the List-based approach, and the more workload, the better performance gain can be achieved through the Tree-based MultitierSync approach. Figure 5.33 shows the throughput of the two approaches with different workload as the working thread number is 64. Noticeably the throughput of the List-based approach grows slowly after the workload is more than 5 times that of the small workload. However, the throughput of the Tree-based approach continues to grow significantly after the workload is more than 5 times that of the small workload, and the gap between the throughputs of the two approaches is enlarged considerably as the workload increases.

Overall, we compared the Tree-based MultitierSync and the List-based MultitierSync (*MultitierPackedSync* as presented in Chapter 4) in the multi-granularity locking scenario with a benchmark which evenly distributes the tasks among the working threads. The test results show that the Tree-based MultitierSync outperforms the List-based MultitierSync considerably in both the small and large workload scenarios in terms of the total execution time and throughput, even though the gap between the two approaches is shrinking in the small workload scenario as the number of working threads is going up. However, the performance gap between the two approaches is enlarged proportionally as the workload increases. The Tree-based approach improved the overall performance significantly in large workload scenarios. According to our benchmark results, the Tree-based approach improves the overall performance by 5.8% to 38.3% in a small workload scenario. The Tree-based

approach works even better with a larger workload.

5.2.4 Space Analysis

The Tree-based MultitierSync is the last and also the most efficient approach we proposed in this thesis for synchronization of the packed object data model in terms of time expense and throughput. However, due to the significant change of the packed object data model over the traditional Java object model, we need auxiliary structures to help synchronize on PackedObjects and unavoidably auxiliary structures increase the overall memory footprint for our approaches. In the Tree-based MultitierSync, we are focused on achieving the best performance in terms of time efficiency. Thus the concurrent multiway tree is only growing but never shrinking during the process of synchronization on the multi-tier PackedObject. This keeps our approach simple and avoids the overhead of deletion/insertion of nodes from the concurrent multiway tree. The space overhead for the Tree-based MultitierSync is proportional to the size of the Multi-tier PackedObjects, and the finer the locking granularity is, the more *PackedMonitorInfo* is created and inserted into the concurrent multiway tree.

The formula to calculate the auxiliary memory footprint of Tree-based MultitierSync is the same as that of List-based MultitierSync (MultitierPackedSync) as shown in Section 4.4:

On-heap PackedObjects:

*(sizeof(PackedRegionInfo) + sizeof(ContainerInfo)) * NumOfContainers +*

*sizeof(PackedMonitorInfo) *NumOfNodes;*

Off-heap PackedObjects:

*(sizeof(NativePackedRegionInfo) + sizeof(ContainerInfo)) * NumOfContainers + sizeof(PackedMonitorInfo) * NumOfNodes;*

However, Tree-based MultitierSync redefined the structure of PackedMonitorInfo, which makes it a different size. In addition, NumOfNodes in Tree-based MultitierSync will increase up to the number of distinct PackedObjects in different tiers being synchronized as there is no node deletion in Tree-based MultitierSync.

Chapter 6

Conclusion

The study of synchronization has grown in urgency with the proliferation of multi-core processors in recent years. In this thesis, we focused on the packed object data model and proposed thread-safe and efficient solutions for synchronization of PackedObjects.

PackedObjects were introduced in IBM's J9 Virtual Machine as an experimental enhancement in order to help users gain greater control over the object layout. This object data model organizes data in a multi-tier manner which is like nested C structs. The packed object data is nested in its enclosing PackedObject instead of being pointed to by an object reference as in the traditional Java object data model. In this thesis, we named such data structures as multi-tier data structures. Our work focuses on how to solve the synchronization of multi-tier PackedObjects efficiently.

PackedObjects can be either allocated on the Java heap or native memory.

Therefore Java applications with PackedObject support gain great flexibility when they work with data structures outside the Java heap and avoid unnecessary data copying and marshalling between the Java heap and native memory. However, the packed object data model also brings challenges due to its significant changes from the traditional Java object data model. The packed object data model removes unnecessary fields in the object header including the lock-word field used in traditional Java synchronization mechanisms, and packed objects can be used to represent data that is located outside the Java heap and thus can be manipulated by both the Java and native code. Synchronization in such a new object data model is out of the scope of traditional Java synchronization mechanisms. Therefore, we proposed an approach PackedSync to PackedObject synchronization, which allows threads to synchronize on the packed data from both Java and native sides concurrently without relying on the lock-word field. We evaluated this approach in both coarse-grained and fine-grained locking scenarios and compared it with the Java built-in synchronization approach used in the IBM's J9 VM. According to the benchmark results, PackedSync performs much better than the Java built-in synchronization in coarse-grained locking scenarios where 18% of the runtime is reduced by PackedSync. For fine-grained locking, PackedSync performs slightly better than or equal to the Java built-in synchronization while array sizes and the number of working threads are small. However, as the array size and thread number increase, the time overhead of PackedSync exceeds the Java built-in synchronization gradually.

The experimental results also show that the PackedSync approach has relatively poor scalability as compared to the Java built-in synchronization in fine-grained locking.

Although PackedSync provides a solution to PackedObject synchronization and it shows performance improvement in certain circumstances, there is still room for improvement. One significant drawback of PackedSync is that it only works with two-tier PackedObjects and supports only the single-granularity locking. If multiple threads are locking at different tiers of a PackedObject at the same time, a new approach is needed. In order to address the multi-granularity locking of multi-tier PackedObjects, we developed MultitierPackedSync based on a concurrent linked-list which allows multiple threads to lock on the data regions of the multi-tier PackedObject from different tiers. The MultitierPackedSync approach optimizes the state-of-the-art approach for linked-list (Lazy-list) by postponing the physical deletion process and thus avoids the overhead of unnecessary repetition of deletion and insertion operations. In the evaluation, we compared MultitierPackedSync against the PackedSync, Lazy_multitierPackedSync (which is based upon the lazy-list algorithm without our optimization) and the Java built-in synchronization, for single-granularity locking in both a coarse-grained and fine-grained manner. The experimental results show that MultitierPackedSync largely overtakes the Java built-in synchronization in coarse-grained locking and greatly reduces the overhead for fine-grained locking compared with PackedSync and Lazy_MultitierPackedSync in the single-

granularity locking scenario, although the performance is not as desirable as the Java built-in synchronization in fine-grained locking. In order to show the performance gain of MultitierPackedSync in multi-granularity locking, we ran the multi-granularity locking benchmark with MultitierPackedSync and the Lazy_MultitierPackedSync. The results show that the optimized approach MultitierPackedSync improves the overall performance significantly compared to the Lazy_MultitierPackedSync.

MultitierPackedSync provides a more efficient approach for PackedObject synchronization supporting multi-granularity locking on the multi-tier PackedObjects with the use of an optimized concurrent linked-list. However, compared to linked-lists, tree structures are better at representing the containing and ordering relationship among different tiers of PackedObjects and more efficient in terms of the average traversal distance. In order to further improve the performance of multi-granularity locking on multi-tier PackedObjects, we implemented a concurrent multiway tree which is represented in a left-child right-sibling tree representation. The concurrent multiway tree algorithm enhances the lazy locking mechanism introduced in the lazy-list algorithm and deals with more complicated scenarios than concurrent linked-lists. The algorithm provides methods Search, Insert and Delete that allows multiple threads to concurrently manipulate the tree structures. With the concurrent multiway tree implemented, we proposed a multi-granularity locking scheme based on the concurrent multiway tree algorithm. In the evaluation, we compared the tree-based MultitierSync with the list-based MultitierSync (which

is MultitierPackedSync). The results show that the tree-based MultitierSync improves the performance considerably over the list-based MultitierSync in different workloads, and the higher the workload, the better the performance gains can be achieved by the tree-based MultitierSync.

6.1 Future Directions

Overall, we developed different approaches to synchronization on the multi-tier data structures based on the Java packed object data model. Our approaches provide efficient solutions to multi-granularity locking and allow multiple threads to synchronize on the packed data from either the Java or native side. There are also drawbacks for each approach and room for further improvement.

The tree-based MultitierSync is the most efficient approach among all approaches we implemented for multi-tier PackedObject synchronization. However, the recursive traversal of the sub-tree while updating the lock states for locking and unlocking operations can have a negative impact on the performance as the tree grows bigger. A possible direction to further improve this approach could be exploring the methods to reduce the overhead of the tree traversal. For example, instead of setting/clearing the lock states of the sub-tree rooted from the target node, less effort might be required to set/clear the lock states of the nodes on the locking path to the target node.

In this thesis, we implemented four different approaches that are lock-based

even though the wait-free Search method is inherited from the Lazy-list algorithm for concurrent multiway trees. The lock-based implementation of concurrent data structures has the advantage of simplicity compared to lock-free approaches but is vulnerable to problems such as deadlock and priority inversion. In addition, locks are blocking. While a process is holding a lock, no other thread can access the portion of the data structure protected by the lock. Therefore, a lock-free implementation could also be a reasonable direction to explore and further improve our approaches in the future.

In many computer applications, the trade between space and time is always inevitable. Our approaches use auxiliary data structures in order to support synchronization on the multi-tier PackedObjects. Our implementation includes deletion operations in the list-based MultitierSync to keep the minimum space usage but sacrifices the time for the extra deletion and insertion operations. In the tree-based approach, we implemented a left-child right-sibling tree that only grows since nodes are not deleted, which reduces the time overhead for extra deletion and insertion operations, but increases the memory usage of this approach. Therefore, how to use the space efficiently and gain the best time-efficiency and how to achieve a balance between the space and time efficiency is another direction to explore for our approaches to synchronization of multi-tier PackedObjects.

In addition, our experiments were running on an x86 system architecture with 16 core processors. It might be worth exploring the performance comparison with our approaches running with a different number of cores or

different system architectures, such as IBM PowerPC, or ARM architecture, if conditions allow in the future.

Bibliography

[1] G. M. Adelson-Velsky and E. M. Landis, *An algorithm for the organization of information*, vol. 146, Proceedings of the USSR Academy of Sciences (in Russian), 1962, pp. 263–266.

[2] Ole Agesen, David Detlefs, Alex Garthwaite, Ross Knippel, Y. S. Ramakrishna, and Derek White, *An efficient meta-lock for implementing ubiquitous synchronization*, Proceedings of the 14th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, ACM, 1999, pp. 207–222.

[3] Joseph Albahari, *Threading in c#*, `http://www.albahari.com/threading/`, Accessed: Dec. 2016.

[4] David F Bacon, Ravi Konuru, Chet Murthy, and Mauricio Serrano, *Thin locks: Featherweight synchronization for java*, ACM SIGPLAN Notices, vol. 33, ACM, 1998, pp. 258–268.

[5] Rudolf Bayer and Mario Schkolnick, *Concurrency of operations on b-trees*, Acta informatica **9** (1977), no. 1, 1–21.

[6] Grady Booch, Douglas L Bryan, and Charles G Petersen, *Software engineering with ada*, Addison-Wesley Professional, 1994.

[7] Nathan G Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun, *A practical concurrent binary search tree*, ACM Sigplan Notices, vol. 45, ACM, 2010, pp. 257–268.

[8] Trevor Brown, Faith Ellen, and Eric Ruppert, *A general technique for non-blocking trees*, ACM SIGPLAN Notices, vol. 49, ACM, 2014, pp. 329–342.

[9] Daniel Cederman, Bapi Chatterjee, Nhan Nguyen, Yiannis Nikolakopoulos, Marina Papatriantafilou, and Philippas Tsigas, *A study of the behavior of synchronization methods in commonly used languages and systems*, Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on, IEEE, 2013, pp. 1309–1320.

[10] Sharad Chauhan, *Programming languages-design and constructs*, Laxmi Publications, 2013.

[11] Shimin Chen and Mengzhi Wang, *Transactional virtual memory with fine-grained range lock support*, Citeseer, 2000.

[12] Norman H Cohen, *Ada as a second language*, McGraw-Hill Higher Education, 1995.

[13] Douglas Comer, *Ubiquitous b-tree*, ACM Computing Surveys (CSUR) **11** (1979), no. 2, 121–137.

[14] Tyler Crain, Vincent Gramoli, and Michel Raynal, *A speculation-friendly binary search tree*, Acm Sigplan Notices **47** (2012), no. 8, 161–170.

[15] Tyler Crain, Vincent Gramoli, and Michel Raynal, *A contention-friendly binary search tree*, European Conference on Parallel Processing, Springer, 2013, pp. 229–240.

[16] S. Dandamudi, S. L. Au, and C. Y. Chow, *A performance study of locking granularity in shared-nothing parallel database systems*, Technical Report TR-97-4, School of Computer Science, Carleton Univeristy, 1997.

[17] David Dice, *Implementing fast java monitors with relaxed-locks.*, Java Virtual Machine Research and Technology Symposium, 2001, pp. 79–90.

[18] Edsger W Dijkstra, *Cooperating sequential processes*, The origin of concurrent programming, Springer, 1968, pp. 65–138.

[19] Marcel Dombrowski, Kenneth B Kent, Michael Dawson, Charlie Gracie, and Rainer Herpers, *Dynamic monitor allocation in the java virtual machine*, Proceedings of the 11th International Workshop on Java Technologies for Real-time and Embedded Systems, ACM, 2013, pp. 30–37.

[20] Dana Drachsler, Martin Vechev, and Eran Yahav, *Practical concurrent binary search trees via logical ordering*, ACM SIGPLAN Notices **49** (2014), no. 8, 343–356.

[21] Ramez Elmasri, *Fundamentals of database systems*, Pearson Education India, 2008.

[22] Faith Fich, Danny Hendler, and Nir Shavit, *On the inherent weakness of conditional synchronization primitives*, Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing, ACM, 2004, pp. 80–87.

[23] James Gosling, Bill Joy, Guy L Steele, Gilad Bracha, and Alex Buckley, *The java language specification*, Pearson Education, 2014.

[24] Goetz Graefe, *Hierarchical locking in b-tree indexes*, BTW, Citeseer, 2007, pp. 18–42.

[25] Vincent Gramoli, *More than you ever wanted to know about synchronization: synchrobench, measuring the impact of the synchronization on concurrent algorithms*, ACM SIGPLAN Notices, vol. 50, ACM, 2015, pp. 1–10.

[26] Jim N Gray, Raymond A Lorie, and Gianfranco R Putzolu, *Granularity of locks in a shared data base*, Proceedings of the 1st International Conference on Very Large Data Bases, ACM, 1975, pp. 428–451.

[27] Michael Barry Greenwald, *Non-blocking synchronization and system design*, Ph.D. thesis, Citeseer, 1999.

[28] Per Brinch Hansen, *Monitors and concurrent pascal: a personal history*, ACM SIGPLAN Notices **28** (1993), no. 3, 1–35.

[29] Timothy L Harris, *A pragmatic implementation of non-blocking linked-lists*, International Symposium on Distributed Computing, Springer, 2001, pp. 300–314.

[30] Steve Heller, Maurice Herlihy, Victor Luchangco, Mark Moir, William N Scherer III, and Nir Shavit, *A lazy concurrent list-based set algorithm*, International Conference On Principles Of Distributed Systems, Springer, 2005, pp. 3–16.

[31] Maurice Herlihy and Nir Shavit, *The art of multiprocessor programming*, Morgan Kaufmann, 2011.

[32] Maurice P Herlihy and Jeannette M Wing, *Linearizability: A correctness condition for concurrent objects*, ACM Transactions on Programming Languages and Systems (TOPLAS) **12** (1990), no. 3, 463–492.

[33] Charles Antony Richard Hoare, *Monitors: An operating system structuring concept*, The origin of concurrent programming, Springer, 1974, pp. 272–294.

[34] KM Franke Hu and R Russell, *Futexes and furwocks: Fast userlevel locking in linux*, Proceedings of the 2002 Ottawa Linux Summit (2002).

[35] *Advantages of using packed objects to access native data*, `https://www.ibm.com/support/knowledgecenter/SSYKE2_8.0.0/com.ibm.java.aix.80.doc/user/packed_native_access.html`, Accessed: Jan. 2017.

[36] *C++ thread support library*, http://en.cppreference.com/w/cpp/thread, Accessed: Jan. 2017.

[37] *Java concurrent utilities*, https://docs.oracle.com/javase/8/docs/technotes/guides/concurrency, Accessed: Dec. 2016.

[38] *Packed data model*, https://www.ibm.com/support/knowledgecenter/SSYKE2_8.0.0/com.ibm.java.lnx.80.doc/user/packed_data_model.html, Accessed: Dec. 2015.

[39] *Thanks for the memory*, http://www.ibm.com/developerworks/linux/library/j-nativememory-linux/index.html, Accessed: Jan. 2016.

[40] Richard Jones, Antony Hosking, and Eliot Moss, *The garbage collection handbook: the art of automatic memory management*, CRC Press, 2016.

[41] Kiyokuni Kawachiya, Akira Koseki, and Tamiya Onodera, *Lock reservation: Java locks can mostly do without atomic operations*, ACM SIGPLAN Notices, vol. 37, ACM, 2002, pp. 130–141.

[42] Andreas Krall and Mark Probst, *Monitors and exceptions: How to implement java efficiently*, Concurrency - Practice and Experience **10** (1998), no. 11-13, 837–850.

[43] Suh-Yin Lee and Ruey-Long Liou, *A multi-granularity locking model for concurrency control in object-oriented database systems*, IEEE Transactions on Knowledge and Data Engineering **8** (1996), no. 1, 144–156.

[44] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley, *The java virtual machine specification*, Pearson Education, 2014.

[45] Hans Henrik Løvengreen, *Synchronization mechanisms*, Course notes, DTU Compute (2016).

[46] Maged M Michael, *High performance dynamic lock-free hash tables and list-based sets*, Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures, ACM, 2002, pp. 73–82.

[47] Mark Moir and Nir Shavit, *Concurrent data structures*, Handbook of Data Structures and Applications, Chapman and Hall/CRC Press, 2007.

[48] Yehudit Mond and Yoav Raz, *Concurrency control in b+-trees databases using preparatory operations*, VLDB, vol. 85, 1985, pp. 21–23.

[49] Rajesh Narang, *Database management systems*, PHI Learning Pvt. Ltd., 2011.

[50] Aravind Natarajan and Neeraj Mittal, *Fast concurrent lock-free binary search trees*, ACM SIGPLAN Notices, vol. 49, ACM, 2014, pp. 317–328.

[51] Patrick Naughton and Herbert Schildt, *C# 3.0: The complete reference*, McGraw-Hill Professional, UK, 2008.

[52] Tamiya Onodera and Kiyokuni Kawachiya, *A study of locking objects with bimodal fields*, ACM SIGPLAN Notices **34** (1999), no. 10, 223–237.

[53] Arunmoezhi Ramachandran and Neeraj Mittal, *Castle: fast concurrent internal binary search tree using edge-based locking*, ACM SIGPLAN Notices, vol. 50, ACM, 2015, pp. 281–282.

[54] Liang Sheng, *Java native interface: Programmers guide and reference*, Addison-Wesley Longman Publishing CO, Boston, MA, USA, 1999.

[55] Michael Spiegel and Paul F Reynolds Jr, *Lock-free multiway search trees*, Parallel Processing (ICPP), 2010 39th International Conference on, IEEE, 2010, pp. 604–613.

[56] John D Valois, *Lock-free linked lists using compare-and-swap*, Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing, ACM, 1995, pp. 214–222.

[57] M.A. Weiss, *Data structures and algorithm analysis in c++*, Pearson Addison-Wesley, 2006.

[58] Bing Yang, Kenneth B Kent, Eric Aubanel, Tobi Ajila, and Angela Lin, *A concurrent multiway tree using the lazy locking mechanism*, Proceedings of the 26th Annual International Conference on Computer Science and Software Engineering, IBM Corp., 2016, pp. 56–63.

[59] Bing Yang, Kenneth B Kent, Eric Aubanel, Angela Lin, and Tobi Ajila, *Multi-tier data synchronization based on an optimized concurrent linked-list*, Proceedings of the 13th International Conference on Principles and

Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools, ACM, 2016, pp. 9:1–9:10.

[60] Bing Yang, Kenneth B Kent, Eric Aubanel, and Karl Taylor, *A monitor-based synchronization approach for java packed objects*, Proceedings of the 25th Annual International Conference on Computer Science and Software Engineering, IBM Corp., 2015, pp. 192–200.

[61] Byung-Sun Yang, Soo-Mook Moon, and Kemal Ebcioğlu, *Lightweight monitors for the java virtual machine*, Software: Practice and Experience **35** (2005), no. 3, 281–299.

Vita

Candidate's full name: Bing Yang

University attended (with dates and degrees obtained):
Master of Computer Science, 2010, Beihang University
Bachelor of Computer Science, 2007, Jilin University

Publications:
[1] B. Yang, K. Kent, E. Aubanel, A. Lin, T. Agila. "A Concurrent Multiway Tree using the Lazy Locking Mechanism." In Proceedings of CASCON 2016, Markham, CA. ACM , Oct. 31- Nov. 2, 2016.
[2] B. Yang, K. Kent, E. Aubanel, A. Lin, T. Agila. "Multi-tier Data Synchronization Based on an Optimized Concurrent Linked-list." Proceedings of the 13th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools. ACM, 2016.
[3] B. Yang, K. Kent, E. Aubanel, K. Taylor. "A Monitor-based Synchronization Approach for Java Packed Objects." In Proceedings of CASCON 2015, Markham, CA. pp. 192-200. ACM, November 2-4, 2015.

Referred Posters:
[1] B. Yang, K. Kent, E. Aubanel, T. Agila, A. Lin., "Multi-granularity locking on multi-tier data structures" in the technology showcase of CASCON 2016, Markham, CA. ACM, Oct. 31- Nov. 2, 2016.