Concurrent B-trees Using Lock-free Techniques

A thesis presented

by

Afroza Sultana

to

The Department of Computer Science

in partial fulfillment of the requirements

for the degree of

Master of Science

in the subject of

Computer Science

The University of Manitoba

Winnipeg, Manitoba

October 2007

THE UNIVERSITY OF MANITOBA

FACULTY OF GRADUATE STUDIES

*****

COPYRIGHT PERMISSION

Concurrent B-trees Using Lock-free Techniques

BY

Afroza Sultana

A Thesis/Practicum submitted to the Faculty of Graduate Studies of The University of Manitoba in partial fulfillment of the requirement of the degree

MASTER OF SCIENCE

THE UNIVERSITY OF MANITOBA

FACULTY OF GRADUATE STUDIES

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies for acceptance, a MSc thesis entitled:

Concurrent B-trees Using Lock-free Techniques

submitted by:

Afroza Sultana

in partial fulfillment of the requirements for the degree of: *MSc*

Dr. Helen Cameron, co-advisor

Dr. Michael Doob
Mathematics

Dr. Peter Graham, co-advisor

Dr. Ruppa Thulasiram
Computer Science

Date of Oral Examination: *October 24, 2007*

The student has satisfactorily completed and passed the MSc Oral Examination.

Dr. Helen Cameron, co-advisor

Dr. John Anderson
Chair of MSc Oral

Dr. Peter Graham, co-advisor

Dr. Ruppa Thulasiram
Computer Science

Dr. Michael Doob
Mathematics

(The signature of the Chair does not necessarily signify that the Chair has read the complete thesis.)

Thesis advisor

Dr. Helen A. Cameron and Dr. Peter C. J. Graham

Author

Afroza Sultana

Concurrent B-trees Using Lock-free Techniques

Abstract

B-trees and their variants are efficient data structures for finding records in a large collection (e.g. databases). The efficiency of algorithm based on B-trees increases when a number of users can manipulate the tree simultaneously. Many algorithms have been developed over the last three decades to achieve both concurrency and consistency in B-trees. However, current lock-based concurrency control techniques limit concurrency. Moreover, lock-based B-trees suffer from certain negative scheduling anomalies, such as deadlock, convoying and priority inversion. Lock-free concurrency control techniques using, for example, Compare and Swap (`CAS`) can provide improved concurrent access to data structures including B-trees and other search structures. Besides this, correctly designed lock-free techniques prevent deadlock, convoying and priority inversion. Considering the advantages of lock-free techniques for other concurrent data structures, I propose to develop a lock-free B-tree like structure to support high performance concurrent in-memory searching in a Non Uniform Memory Access (NUMA) parallel computing environment.

The use and parallelization of B-trees have both been widely explored in the past – primarily for application to database implementation and, hence, disk-based operations. Moving B-trees into memory for use in new online searching applications,

however, fundamentally changes the characteristics of managing them and will allow me to effectively exploit the use of lock-free techniques, something that has previously not been applicable to B-trees.

Contents

Abstract ii
Table of Contents iv
List of Figures viii
List of Tables xiii
Acknowledgments xiv
Dedication xv

1 Introduction **2**

2 Background **4**
2.1 Parallel Machines 4
2.1.1 Distributed Memory Architectures 5
2.1.2 Shared Memory Architectures 6
2.1.3 Non-Uniform Memory Access (NUMA) Machines 7
2.2 Concurrency 9
2.2.1 Pessimistic Concurrency Control 9
Test-And-Set (TAS) 10
Locks 11
2.2.2 Optimistic Concurrency Control 13
Universal Primitives 14
Compare-And-Swap (CAS) 15
The ABA Problem 15
Double-Compare-And-Swap (DCAS) 17
2.2.3 Lock-free Concurrency Control Techniques 19
2.2.4 Pessimistic Versus Optimistic Concurrency Control 19
2.3 Search Trees 20
2.3.1 Binary Search Trees 21
2.3.2 2-3 Trees 22
2.3.3 B-trees and Their Variants (B^+ Trees, B^* Trees) 23
B-trees 24
B^*-trees 26

B^+-trees 29

3 Related Work **30**
3.1 Concurrent Algorithms for B-trees and Their Variants 30
3.1.1 Lehman and Yao's B^{link}-tree 32
3.1.2 Sagiv's B^{link}-trees with Overtaking 36
3.1.3 Lanin and Shasha's Symmetric Concurrent B-tree Algorithm . 39
3.1.4 Biliris' Operation-Specific Locking Algorithm 40
3.1.5 De Jonge and Schijf's Improved Overtaking Algorithm 42
3.1.6 Das and Demuynck's B^{mad}-tree Algorithm 43
3.1.7 Performance Evaluation of Concurrent B-trees 46
3.2 Relevant Lock-free Data Structures 48
3.2.1 Valois' Lock-free Linked List 50
3.2.2 Michael's Lock-free Linked List 52
3.3 Lock-free Parallel Implementations of Trees 55
3.3.1 Ma's Red-black Tree Insertion Algorithm 55
3.3.2 Kim's Red-black Tree Insertion and Deletion Algorithm 55
3.4 Other Related Work 56

4 Problem Description **57**
4.1 Motivation 57
4.1.1 Problems with Lock-based Algorithms 57
4.1.2 Large Scale In-memory Search Algorithms 58
4.1.3 Parallel Machine Architectures 59
4.2 Problem Statement 59

5 Solution **61**
5.1 The B^{list} Tree 61
5.1.1 Cell 62
5.1.2 Key-List 62
5.1.3 Node 65
5.1.4 The Hazard Pointers Array 67
5.1.5 The Path Stack 68
5.1.6 The Retired-Cell Stack 70
5.1.7 The Please-Scan Array 71
5.2 Benefits of B^{list}-trees 72
5.2.1 Benefits of an In-Memory Structure 73
5.2.2 Benefits of the List Structure 73
5.2.3 Benefits of Lock-free Concurrency-Control Techniques 74

6 Algorithms **77**
6.1 Algorithm Overview and Discussion 77
6.1.1 Search Algorithm 77
Step 1: Find the Appropriate Leaf 78
Step 2: Find the Desired Key in the Leaf 83
6.1.2 Insert Algorithm 86
Step 1: Create a Root 88
Step 2: Find the Appropriate Leaf 88
Step 3: Allocate a Cell for the New Key 89
Step 4: The Search for the New Key's Position 92
Step 5: Try to Insert the Key in the Leaf 92
Step 6: Split the Leaf 94
Step 7: Find the Parent 97
Step 8: Make the Tree Taller 101
Step 9: Insert the Middle Key into the Parent 102
Step 10: Fix the Parent's High-key 105
6.1.3 Delete Algorithm 107
Step 1: Find the Appropriate Leaf 108
Step 2: Search for the Desired Key 108
Step 3: Delete the Requested Key from the Leaf 108
Step 4: Retire the Deleted Cell 110
Step 5: Scan the Retired-cell Stack 111
Step 6: Reclaim the Retired Cells 112
6.2 Concurrency Correctness of the Algorithms 113
6.2.1 Concurrency Control in Search Operations 114
Conflicts with Concurrent Update Operations 114
Conflicts with Concurrent Split Operations 120
6.2.2 Concurrency Control in Update Operations 123
Conflicts with Other Concurrent Update Operations 123
Conflicts with Concurrent Split Operations 131
6.2.3 Concurrency Control in Split Operations 132

7 Assessment **134**
7.1 Experimental Setup 134
7.1.1 Test-bed Environment 135
7.1.2 Programming Language 136
7.1.3 Parameters Varied 136
Different Operation Types 137
Different Values for m 137
Different Key Ranges 138
Different Numbers of Processes 139
Memory Affinity 139

7.2 Results 140
7.2.1 Sun Machine with Eight Processors 140
Experiment 1: 100% Search Operations 141
Experiment 2: 90% Search and 10% Insert Operations 147
Experiment 3: 50% Search and 50% Insert Operations 156
Experiment 4: 10% Search and 90% Insert Operations 163
Experiment 5: Equal Mix of Search, Insert and Delete Operations 169
Summary of the Experiments on the Sun Fire Machines 175
7.2.2 SGI Machines 178
Experiment 1: 100% Search Operations 179
Experiment 2: 50% Search and 50% Insert Operations 179
Experiment 3: Equal Mix of Search, Insert and Delete Operations 184
Summary of the Experiments on the SGI Machines 184

8 Conclusion and Future Work **189**
8.1 Conclusions 189
8.2 Future Work 191

Bibliography **194**

List of Figures

2.1 Distributed memory architecture. 5
2.2 Shared memory architecture. 6
2.3 NUMA architecture. 8
2.4 `TAS` algorithm. 11
2.5 The `lock` algorithm. 12
2.6 The `Unlock` algorithm. 13
2.7 `CAS` algorithm. 16
2.8 The `DCAS` algorithm. 18
2.9 A Binary Search Tree (BST). 21
2.10 A 2-3 Tree. 23
2.11 A B-tree of order 4. 25
2.12 A B^*-tree of order 3. 26
2.13 A B^+-tree. 29

3.1 Relationship between the B-tree and its variants. 33
3.2 An example of a B^{link}-tree. 34
3.3 Valois' lock-free linked list. 50
3.4 Michael's lock-free linked list. 53

5.1 The structure of a cell in a B^{list}-tree. 63
5.2 The `cell` class in a B^{list}-tree. 63
5.3 An example of two consecutive cells in a B^{list}-tree. 64
5.4 The structure of a `key-list` in a B^{list}-tree. 65
5.5 The `key-list` class in a B^{list}-tree. 66
5.6 An example of a `key-list` in a leaf of a B^{list}-tree. 66
5.7 The structure of a `node` in a B^{list}-tree. 67
5.8 The `node` class in a B^{list}-tree. 68
5.9 An example of a `node` in a B^{list}-tree. 69

6.1 Initial locations of the hazard pointers in a node while searching. . . . 79

6.2 Move the hazard pointers to the immediately following cells in the `key-list`. 80
6.3 The appropriate child (shaded node) is selected. 81
6.4 The rightmost child (shaded node) is selected. 82
6.5 The shaded nodes are examined and stored in the `path` stack by the search process to find the appropriate leaf. 83
6.6 Initial hazard pointer locations in a leaf node. 84
6.7 Move the hazard pointers one cell ahead in the `key-list`. 84
6.8 The search process finds the desired key in the leaf. 85
6.9 The search process could not find the desired key in the leaf. 85
6.10 The hazard pointers point to the end of the `key-list`, and the search process could not find the desired key in the leaf. 86
6.11 The key steps of the insert algorithm. 87
6.12 The insert process creates a new leaf for an empty tree. 89
6.13 Allocate a new cell from the `free-list`, and copy the new key into that cell. 90
6.14 Insert the new cell between the `head` and the `tail` of the `key-list`. . 90
6.15 Update the `key-counter` and the `high-key` of the node. 91
6.16 The next field of the new cell is set to point to the current cell. 93
6.17 CAS swings the `next` field of the previous cell to point the new cell. . 94
6.18 Insertion of the new cell fails. 94
6.19 A split process finds the middle key containing 300. 97
6.20 A split process creates a new leaf, and updates the fields of the new leaf. 98
6.21 A split process changes the `right` pointer, the `high-key`, and the `key-counter` of the original leaf. 99
6.22 A split process returns the reclaimed cells to the free list of the original leaf. 100
6.23 The insert process creates a new root for the B^{list}-tree. 102
6.24 Linking the new root into the tree. 102
6.25 The insert process sets the root pointer to point to the new root. . . . 103
6.26 The insert process copies the middle key, the original leaf, and the new leaf to the `key`, `left-child`, and the `right-child` fields of the new cell, respectively. 104
6.27 The locations of `hp1` and `hp2` of the inserter in the `key-list` of the parent node, before the insertion. 105
6.28 The insert process inserts the new cell in the `key-list`. 106
6.29 The insert process fixes the `left-child` pointer of the cell pointed to by `hp2`. 106
6.30 The insert process does not need to fix the `left-child` pointer of the cell pointed to by `hp2`, if the new cell is the rightmost cell in the `key-list`. 107
6.31 The delete process finds the desired key 330 in the key list. 110

6.32 The delete process marks as deleted the current-cell pointed to by hp2. . . . 110
6.33 The CAS operation swings the next field of the previous-cell (pointed to by hp1) from the current-cell (pointed to by hp2) to the new-current cell. . . . 110
6.34 The delete process links the cell to be reclaimed (new-free) to the free-list. . . . 112
6.35 The CAS operation swings the free-head from copy-free to new-free. 113
6.36 During a search operation, a concurrent process wants to insert a new cell between the searcher's hp1 and hp2. . . . 115
6.37 During a search operation, a concurrent inserter is trying to insert a new cell between the searcher's hp1 and hp2. . . . 115
6.38 During a search operation, a concurrent inserter has inserted a new cell between the searcher's hp1 and hp2. . . . 116
6.39 During a search operation, the searcher re-assigns hp2 to the cell immediately following to the cell pointed to by hp1. . . . 116
6.40 During a search operation, a concurrent delete is deleting the cell pointed to by the searcher's hp1. . . . 117
6.41 During a search operation, a concurrent delete is deleting the cell pointed to by the searcher's hp2. . . . 118
6.42 During a search operation, a concurrent delete has already deleted the cell pointed to by the searcher's hp2. . . . 119
6.43 In a search operation, the searcher re-assigns hp2 to the cell immediately after the cell pointed to by hp1. . . . 119
6.44 No conflicts occur if the insertions are taking place to the left of the hp1 of the searcher or to the right of hp2 of the searcher. . . . 120
6.45 No conflicts occur if the deletions are taking place to the left of the hp1 of the searcher or to the right of hp2 of the searcher. . . . 120
6.46 Two concurrent inserts want to insert in the same position. . . . 125
6.47 Inserter 1's CAS succeeds, and Inserter 2's CAS fails. . . . 125
6.48 Two concurrent deleters want to delete in the same position. . . . 127
6.49 One deleter succeeds in marking the cell to be deleted. . . . 127
6.50 One deleter succeeds, but the other deleter fails to delete the cell from the key-list. . . . 128
6.51 Two concurrent deleter operations want to delete two consecutive cells. 129
6.52 Two concurrent deleters want to delete two consecutive cells. . . . 129
6.53 The second deleter deletes the cell before the first deleter marks its cell to be deleted. . . . 130
6.54 The first deleter marks the cell to be deleted before the second deleter deletes current-cell-2. . . . 130
6.55 The second deleter fails to delete and unmarks the cell to be deleted. 131

7.1 Result of 100% search operations on Sun Fire (m is 100). 142
7.2 Result of 100% search operations on Sun Fire (m is 200). 143
7.3 Result of 100% search operations on Sun Fire (m is 500). 144
7.4 Result of 100% search operations on Sun Fire (m is 700). 145
7.5 Result of 100% search operations on Sun Fire (m is 1000). 146
7.6 Result of 90% search operations and 10% insert operations on Sun Fire (m is 100). 149
7.7 Result of 90% search operations and 10% insert operations on Sun Fire (m is 200). 150
7.8 Result of 90% search operations and 10% insert operations on Sun Fire (m is 500). 151
7.9 Result of 90% search operations and 10% insert operations on Sun Fire (m is 700). 152
7.10 Result of 90% search operations and 10% insert operations on Sun Fire (m is 1000). 153
7.11 Result of 50% search operations and 50% insert operations on Sun Fire (m is 100). 157
7.12 Result of 50% search operations and 50% insert operations on Sun Fire (m is 200). 158
7.13 Result of 50% search operations and 50% insert operations on Sun Fire (m is 500). 159
7.14 Result of 50% search operations and 50% insert operations on Sun Fire (m is 700). 160
7.15 Result of 50% search operations and 50% insert operations on Sun Fire (m is 1000). 161
7.16 Result of 10% search operations and 90% insert operations on Sun Fire (m is 100). 164
7.17 Result of 10% search operations and 90% insert operations on Sun Fire (m is 200). 165
7.18 Result of 10% search operations and 90% insert operations on Sun Fire (m is 500). 166
7.19 Result of 10% search operations and 90% insert operations on Sun Fire (m is 700). 167
7.20 Result of 10% search operations and 90% insert operations on Sun Fire (m is 1000). 168
7.21 Result of equal mix of search, insert and delete operations on Sun Fire (m is 100). 170
7.22 Result of equal mix of search, insert and delete operations on Sun Fire (m is 200). 171
7.23 Result of equal mix of search, insert and delete operations on Sun Fire (m is 500). 172

7.24 Result of equal mix of search, insert and delete operations on Sun Fire (m is 700). 173
7.25 Result of equal mix of search, insert and delete operations on Sun Fire (m is 1000). 174
7.26 Result of 100% search operations on SGI (m is 100). 180
7.27 Result of 100% search operations on SGI (m is 500). 181
7.28 Result of 100% search operations on SGI (m is 1000). 182
7.29 Result of 50% search operations and 50% insert operations on SGI (m is 100). 185
7.30 Result of 50% search operations and 50% insert operations on SGI (m is 500). 186
7.31 Result of 50% search operations and 50% insert operations on SGI (m is 1000). 187

List of Tables

3.1 Lock compatibility rules in Biliris' operation-specific locking algorithm. 41
3.2 Algorithm classification in Srinivasan and Carey's experiments. 48
3.3 Results of Srinivasan and Carey's experiments. 49

7.1 Percentage of performance gain in B^{list}-trees compared to the B^{link}-linked-list trees with eight concurrent processes. 176
7.2 Percentage of performance gain in B^{list}-trees compared to the B^{link}-array B^{link}-linked-list trees with 50% search and 50% insert operations on 32 concurrent processes. 188

Acknowledgments

I would like to thank all those magnificent people who believed in me and supported me throughout my research work. I would first like to thank my supervisors Dr. Helen Cameron and Dr. Peter Graham for giving me the opportunity to work with them in such an interesting topic. Without their continuous supervision, valuable inputs, and endless encouragement at every stages of my thesis, this thesis would not have been completed. I would also like to thank all system stuffs of the Department of Computer Science of University of Manitoba, specially Mr. Gilbert Detillieux for not only allowing me to have dedicated accesses to the parallel machines whenever I needed, also for his kind considerations on other resource allocations for my thesis. I am also grateful to westgrid team to allow me to use their SGI machines and Mr. Jonatan Aronsson for giving his valuable time for help me to understand the westgrid environment. Last, but not the least, I would like to thank my family, friends, lab mates and well-wishers for always supporting me unconditionally along the way.

This thesis is dedicated to my parents.

Chapter 1

Introduction

Historically, memory capacity has been limited so that large data collections had to be stored on disk in databases, which use data structures such as B-trees. With the availability of large memories, most notably in shared-memory multiprocessors, this restriction has been relaxed. Correspondingly, a number of new applications have emerged in such fields as bio-informatics and computational linguistics that require searching huge collections in memory. A B-tree-like data structure (built in memory) is still a good solution for such problems.

To achieve concurrency in B-trees, various lock-based algorithms have been designed, but those algorithms suffer from a number of problems. For example, some algorithms limit efficiency since, when a process manipulates some nodes in a B-tree, all other processes that want to manipulate the same nodes are forced to wait, even if concurrent access would not cause any inconsistencies. Additionally, the use of locks introduces some negative side effects related to scheduling. To address these problems, I have designed lock-free algorithms for a B-tree variant, where several

processes can work on the tree simultaneously. More specifically, I have developed a locality-of-reference-efficient version of Lehman and Yao's B^{link}-tree [24] that does not require locking and which exploits memory management ideas from the work of Michael [27]. My algorithms are also designed to tolerate variation in memory access times present in the increasingly common Non Uniform Memory Access (NUMA) time class of parallel machines.

My algorithms will support in-memory applications. This support is in contrast to the existing literature on B-trees, which focuses on operations performed on-disk. Lehman and Yao's lock-based B^{link}-tree achieves better concurrency compared to my B^{list}-tree. However, when comparing a variant of the lock-based B^{link}-tree that has the same structural complexity in the nodes as my B^{list}-tree, my B^{list}-tree performs better than the linked-list-based variant of the B^{link}-tree when there is no thread sharing in the same processor. Furthermore, my lock-free B^{list}-tree is not prone to deadlock, priority inversions and convoying, unlike the lock-based B^{link}-trees.

The rest of this thesis is organized as follows. In Chapter 2, I describe parallel machines, and different basic concurrency-control techniques. Furthermore, I define B-trees, B^*-trees and B^+-trees in the same chapter. I then review some existing lock-based B-tree algorithms and some lock-free data structures in Chapter 3. In Chapters 4 and 5, I give the problem description for my thesis research along with the key features of my algorithm. Chapter 6 describes the implementation details of my algorithms and Chapter 7 describes my experimental assessment of my algorithms. Finally, in Chapter 8, I summarize my thesis, identifying the contributions made, and describing possible future research that could be carried out.

Chapter 2

Background

This chapter provides background on parallel machine architectures, and basic concurrency control techniques as well as a discussion of Binary Search Trees (BSTs), 2-3 trees, and B-trees and their variants. Those who are familiar with this material may skip this chapter.

2.1 Parallel Machines

Parallel machines are collections of computers that communicate with each other to solve problems by working together simultaneously. The machines in a parallel computer system execute programs that have been broken into pieces. Each machine runs different parts of such programs on the different computers simultaneously to obtain faster results. The two major types of architectures that are currently used in parallel machines are the distributed memory architectures and the shared memory architectures.

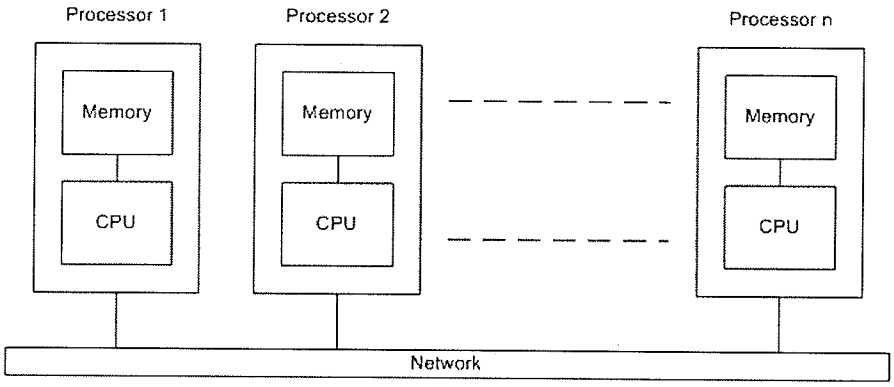


Figure 2.1: Distributed memory architecture.

2.1.1 Distributed Memory Architectures

In a distributed memory architecture [18, 30], the parallel processors consist of local memories tightly-coupled to the central processing units (CPUs). All processors are connected through a network to build a parallel machine. Each processor in the system can only access its own local memory, but not the memories local to the other processors. A computer in a distributed memory architecture machine is known as a *node* of the system. All nodes communicate with each other by sending messages through the network. Figure 2.1 shows a parallel machine using the distributed memory architecture.

There are some advantages and disadvantages of the distributed memory architecture compared to the shared memory architecture (see Section 2.1.2 for details on shared memory architectures). The memory management in distributed memory architecture is similar to single processor systems, since there is no memory sharing among the processors. Distributed memory architectures are relatively cheaper

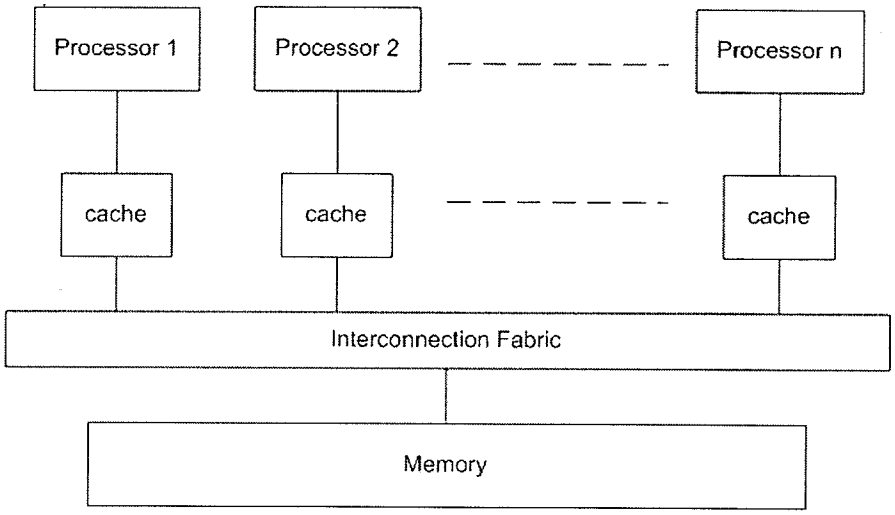


Figure 2.2: Shared memory architecture.

and more scalable than shared memory architectures. However, when many processors must frequently access the network to pass messages to each other, the overall performance of a distributed memory system is normally not as good as the overall performance of a shared memory architecture with the same number of processors.

2.1.2 Shared Memory Architectures

In a shared memory architecture, each processor has its own cache which is connected to a high-speed interconnection fabric. The interconnection fabric is also connected to shared memory so that all parallel processors share a single memory. Figure 2.2 shows a parallel computer system that uses a shared memory architecture.

Shared memory machines also have some advantages and disadvantages. From the user's perspective, they are easier to program and it is convenient to share data in a

shared memory architecture. On the other hand, the shared memory architecture is not very scalable, and is more expensive than parallel machines that use a distributed memory architecture.

In a shared memory architecture, memory access time for the parallel processors may vary according to the location of the memory. When the memory access time for all processors to all memory locations is the same, then the shared memory architecture is known as Uniform Memory Access (UMA)-time architecture [18]. Conversely, when the memory access time for a process depends on how near the memory segment is to the processor (i.e., if the memory address is nearer to the processor, then it takes less access time than when the processor accesses a relatively distant memory location), then that shared memory architecture is called a Non-Uniform Memory Access (NUMA)-time architecture [18].

2.1.3 Non-Uniform Memory Access (NUMA) Machines

In a NUMA architecture, the shared memory is divided into segments and each segment is attached to one processor with a connecting bus. A processor in a NUMA architecture can access its associated memory directly through the bus, but it has to connect through the interconnection fabric to access memory attached to other processors. Therefore, the memory M_i, that is attached to processor P_i, can be treated as P_i's local memory. Thus, P_i has faster access to M_i than other memories in the parallel machine. Figure 2.3 shows a high-level diagram of the NUMA architecture.

The NUMA architecture has some advantages over uniform memory access (UMA)-time shared memory architectures. UMA architectures are hard to scale to more than

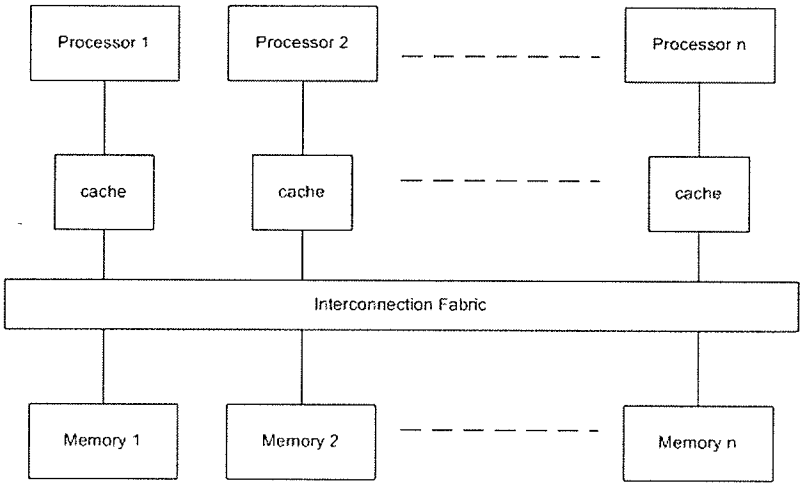


Figure 2.3: NUMA architecture.

a small number (e.g., 16) of processors. The bottleneck that is created on the interconnection fabric to access the single shared memory affects the overall performance of the parallel processors. However, in a NUMA architecture, each processor has a direct bus connection to its local memory. Because a processor does not have to communicate through the interconnection fabric to access its local memory, the interconnection fabric works faster. Therefore, the NUMA architecture is scalable to large numbers of processors without affecting the overall performance. This scalability, however, may require careful placement of data in memory on the part of the programmer.

2.2 Concurrency

In a parallel machine, a large number of operations may be performed simultaneously on a data structure, by different processes or threads[1]. In a shared memory architecture, if more than one process wants to update the same data at the same memory location at the same time, the content of the data might become inconsistent. This inconsistency must be prevented.

To ensure consistency, the simplest solution is to perform only one operation at a time on each data structure. Thus, if one process is searching or updating in a data structure, then other processes must wait until that process is finished accessing the structure. However, this solution is not practical because the overall performance of all processes accessing that data structure will not be efficient compared to their performance in a concurrent execution. To achieve better performance, several concurrency control techniques have been introduced to control concurrent access to, and manipulation of, data structures. Some of these concurrency control techniques offer better performance than others in various situations. I will now describe the two major variants: pessimistic and optimistic concurrency control.

2.2.1 Pessimistic Concurrency Control

By pessimistic concurrency control, I mean any technique that prevents any unsafe concurrent access. More precisely, if there is a vulnerable portion of code (commonly known as a *critical section*) where concurrent execution of that code by different

[1]Without loss of generality, I will hence forth use the term *process* only. Unless explicitly stated, remarks made also apply to other units of concurrency including threads.

processes might cause inconsistency in the data, then only one process is permitted to execute that portion of code at a time. The `Test-And-Set (TAS)` [7] primitive is commonly used to implement most major pessimistic concurrency-control mechanisms, including locks [6].

Test-And-Set (TAS)

Pessimistic algorithms typically use the `Test-And-Set (TAS)` instruction [7] as a primitive building block to ensure that processes can execute their critical sections mutually exclusively. To ensure the safety of execution of a critical section, `TAS` looks at the current status of a shared variable (`lock`). If the critical section is safe because no other process is accessing the shared data (i.e., if `lock == 0`), a process is allowed to execute its critical section of code. Otherwise, when `lock == 1` (i.e., the critical section is not safe due to the execution of another concurrent process), the process must wait until the critical section becomes safe for execution. This waiting is normally accomplished by *busy waiting* on the lock using `TAS` (i.e., `while(TAS(lock));`). The indivisible or atomic[2] code in Figure 2.4 shows how `TAS` works.

Locking is an example of a pessimistic concurrency-control technique that can be built using `TAS`. I will now describe how locking controls concurrency in a multiprocess environment.

[2]An *indivisible* or *atomic* segment of code is guaranteed to execute in its entirety without any other process interfering.

```
TAS(lock){
    /* Begin atomic */
    old_value = lock;
    lock = 1;            /* The shared data object is now locked */
    return old_value;  /* Returns the original status of the lock */
    /* End atomic */
}
```

Figure 2.4: TAS algorithm.

Locks

Locking [6] a shared data item is the main operating-system-supported pessimistic concurrency-control approach. I denote a *lock* associated with a shared data item, X, by, Lock_X. Initially the value of Lock_X is set to 0, as no process has locked the data item X yet. When a process P_1 locks the data item X, the process sets the value of Lock_X to 1. If any process other than P_1 wants to hold the lock of X, it has to wait in a queue until P_1 sets the value of Lock_X back to 0 after it finishes its work on X.

Two operations are performed on locks. They are lock(Lock_X) and unlock(Lock_X). A process locks a shared data item X by executing the lock(Lock_X) operation (see Figure 2.5). In lock(Lock_X), a process tests the value of Lock_X and, if possible (i.e., if Lock_X == 0), the process locks X by setting the value of Lock_X to 1. This operation is normally implemented using TAS. If X is already locked (i.e., Lock_X == 1), the runtime environment or OS blocks the process and forces it to wait in a queue until X is unlocked (instead of busy waiting as was done with CAS). Locking blocks

```
lock(Lock_X) {
  /* Begin Atomic */
  /* If X is not locked, acquire the lock on X */
 B: If (Lock_X == 0)
        Lock_X = 1;
  /* If X is already locked, block the process until X is unlocked */
    else {
        while(Lock_X == 0)
            RunTime.BlockMe();
        go to B;
        }
  /* End Atomic */
}
```

Figure 2.5: The `lock` algorithm.

processes when one process holds the lock of a shared data item. Locking, therefore, guarantees sequential access by all processes to their critical sections. Moreover, this blocking avoids the inefficiency of the busy waiting associated with using `TAS` alone.

When a process wants to release the lock from a shared data item `X`, the process performs the `unlock(Lock_X)` operation (see Figure 2.6). In this operation, the process unlocks `X` by setting the value of `Lock_X` back to 0. After unlocking `X`, it unblocks a process (if any) that wanted to lock `X` and is currently blocked by the runtime environment/OS.

```
unlock(Lock_X) {
    /* Begin Atomic */
    Lock_X = 0;     /* unlock X */
    if (any processes are blocked)
        unblock one of the blocked processes;
    /* End Atomic */
}
```

Figure 2.6: The `Unlock` algorithm.

Unfortunately, the overhead of acquiring locks and certain negative scheduling anomalies, such as deadlock [6], convoying [41] (when a relatively slow process acquires a lock on a data object, and as a result other processes have to wait for a long time), and priority inversion [33] (when a low-priority process holds a lock on a shared data object forcing high-priority processes to wait for it) are undesirable properties of the use of locks.

2.2.2 Optimistic Concurrency Control

Unlike pessimistic concurrency control, optimistic concurrency-control techniques assume that there will seldom be conflicts in accessing a shared data object. They allow concurrent accesses in a critical section without checking whether the access is safe. Instead, after computing the new value of a shared data object, but before storing it, the value used in computing the new value and the current value of the shared data object are compared. If both values are the same, the old value may

be safely replaced by the new value since the shared object has not been changed. Otherwise, the computation must be redone using the changed value. Comparing the values of the shared object and then possibly updating the object must be done atomically. The most common examples of optimistic concurrency control are the so-called lock-free techniques. Lock-free techniques (see Section 2.2.3) use universal primitives [17] like CAS [19] and DCAS [20] to achieve optimistic concurrency control in parallel machines.

Universal Primitives

Before describing universal primitives, the concepts of the consensus problem [14] and consensus number [17] need to be understood. In the consensus problem, multiple concurrent process need to produce a common output from their individual inputs. If a synchronization primitive can solve the consensus problem for a maximum of n processes, then n is known as the consensus number of that synchronization primitive. If there exists a synchronization primitive with a consensus number of infinity, then it can solve the consensus problem for an infinite number of processes. Herlihy [17] named such synchronization primitives *universal* primitives, since they can solve any concurrency problem with any number of processors.

There are several universal primitives, including non-atomic registers, memory-to-memory move, CAS, and DCAS, that can be used to build other universal primitives. CAS and DCAS are examples of primitives that can be used to build other lock-free objects like lock-free linked lists [27], queues [28], and different types of trees [21, 25].

Compare-And-Swap (CAS)

Optimistic algorithms commonly use the `Compare-And-Swap` (`CAS`) primitive [19] for concurrency control. If a process wants to update the value stored in some location `address`, the process first makes a copy, in `old_value`, of the contents stored at location `address`. Then the process calculates a new value, `new_value`. The `CAS` operation takes `address`, `old_value`, and `new_value` as its arguments, and compares the contents stored at location `address` with `old_value` to make sure they have not been changed during the computation of the `new_value`. If no changes were made, `CAS` updates the contents of `address` with `new_value` and returns `TRUE` to indicate that `CAS` has successfully updated the value at the location `address`. Otherwise, `CAS` returns `FALSE`, meaning that `CAS` failed to update the contents at location `address` with `new_value` because some other concurrent process(es) have already manipulated the contents at location `address`. The code in Figure 2.7 shows the atomic `CAS` operation.

The ABA Problem

What has come to be known as the *ABA problem* [20] may occur when a system uses `CAS` for concurrency control. The ABA problem happens when a process is unable to recognize certain changes in a shared variable. Suppose the value `A` is stored at location, `address`. To update the contents at the location `address`, a process P_i stores `A` in the variable `old_value`, and computes the updated value and stores it in the variable `new_value`. While process P_i computes its new value, some other process P_j may update the contents at location `address` to `B`, and then another process P_k

```
CAS(address, old_value, new_value) {
    / * Begin atomic */
    /* The old value is replaced by the new value if it is unchanged */
    if(*address == old_value) {
        *address = new_value;
        return TRUE;
    }
    /* If any changes occured, the algorithm returns failure */
    else {
        return FALSE;
    }
    /* End atomic */
}
```

Figure 2.7: CAS algorithm.

may change the contents at location address back to A. When process P_i uses CAS to compare the contents at location address and old_value, it will succeed since both contain the same value, A. In such a situation, CAS is unable to detect the changes in the contents at location address, and will update the contents at location address with the value stored in new_value. In some cases, the ABA problem may cause incorrect results.

Double-Compare-And-Swap (DCAS)

A `Double-Compare-And-Swap (DCAS)` primitive [20] can be used to solve the ABA problem. A `DCAS` operation can simultaneously and atomically modify two arbitrary data items. `DCAS` takes six arguments, instead of three as with `CAS`. They are:

- Locations (`address1` and `address2`) of two shared data items.
- Old values (`old_value1` and `old_value2`) of the two shared data items.
- New values to be stored in the shared data items (`new_value1` and `new_value2`).

A process using `DCAS` stores the contents at `address1` in `old_value1` and the contents at `address2` in `old_value2`, respectively. The process uses `DCAS` to atomically compare the contents of `address1` with `old_value1` and the contents of `address2` with `old_value2` after computing `new_value1` and `new_value2`. If both comparisons are true, the contents at `address1` are replaced with `new_value1` and the contents at `address2` are replaced with `new_value2`. Figure 2.8 shows the `DCAS` algorithm.

To solve the ABA problem, a process associates a version number with each shared data value. The memory location of the data value that a process wants to update, the original data value itself, and the new value to be stored are used as `address1`, `old_value1`, and `new_value1` in the `DCAS` operation. Besides this, the memory location of the version number, the current version number of the data object, and the new version number (current version number + 1) are sent as `address2`, `old_value2`, and `new_value2` in the same DCAS operation. The DCAS operation compares the contents at `address1` with `old_value1` (data to be updated), and the contents at `address2` with `old_value2` (version number). If both equality checks return true, the

```
DCAS(*address1, *address2, old_value1, old_value2, new_value1,
new_value1) {
    /* Begin atomic */
    if((*address1 == old_value1) && (*address2 == old_value2)){
        *address1 = new_value1;
        *address2 = new_value2;
        return TRUE;
    }
    else {
        return FALSE;
    }
    /* End atomic */
}
```

Figure 2.8: The DCAS algorithm.

process atomically updates the contents at address1 with new_value1 and updates the contents at address2 with new_value2 (the incremented version number) as a record of the update. If other concurrent processes update the contentss of address1 during the computation of the new value to potentially cause an ABA problem, they will also increment the corresponding version number. Thus, the comparison of the old and current version numbers (address2 and old_value2) will detect the changes and thereby prevent the process from updating the contents of address1.

2.2.3 Lock-free Concurrency Control Techniques

In a lock-free environment, the concurrent processes, operating on a shared data object, do not need to wait to access a critical section while another process accesses its corresponding critical section. Unlike with locking techniques, concurrent processing using lock-free techniques do not hold locks on a shared data object while working on that object. With lock-free techniques, processes use optimistic primitives, such as `CAS`, to achieve concurrency.

Non-blocking techniques and *wait-free* techniques are two types of widely-used lock-free techniques. If a lock-free technique guarantees that some processes will complete their operations in a finite number of steps, then that lock-free technique is *non-blocking.* If the lock-free technique guarantees that every process will complete its operations in a finite number of steps regardless of the execution speeds of the other processes, then that lock-free technique is *wait-free* [17].

2.2.4 Pessimistic Versus Optimistic Concurrency Control

Optimistic concurrency control algorithms have some advantages over pessimistic concurrency control algorithms in the absence of frequent access conflicts between concurrent processes. Algorithms based on CAS (and similar primitives) have almost no overhead when compared to locking. Using lock-based algorithms, whether there are conflicts or not, a process needs to invoke the operating system to test the lock of the critical section. Further, other processes cannot access the critical section concurrently. Therefore, if no conflict occurs between processes, optimistic algorithms can accommodate more concurrency with less overhead than lock-based algorithms.

Additionally, optimistic algorithms ensure a deadlock-free environment, whereas in a multiprocess environment using pessimistic algorithms like locking techniques, if, for example, a faulty process acquires a lock and halts, other processes needing the lock can never proceed, causing deadlock [6]. Besides this, there are no priority inversions [33] when using optimistic algorithms. Unlike pessimistic algorithms, with optimistic algorithms, a higher-priority process can perform operations on a concurrent object without even being blocked by a lower-priority process. Furthermore, pessimistic algorithms are prone to convoying [41], whereas optimistic algorithms guarantee progress of some concurrent process(es) during execution of a critical section.

However, in the presence of frequent conflicts, optimistic algorithms may take more time than lock-based algorithms, since processes will need to recompute new values when other processes have changed the shared data objects. Designing an intelligent, cost-effective lock-free algorithm can, however, keep the cost at an acceptable level.

2.3 Search Trees

Searching for a specific object from a collection of data is a very common problem in computer science. Many search algorithms have been designed that take significantly less running time to find a data object than linear search. Search trees are often the basis of such efficient algorithms. In a tree, data objects stored in the nodes can work as *routers* to quickly guide a search to a desired data object. Binary search trees (BSTs), 2-3 trees and B-trees are some examples of such search trees.

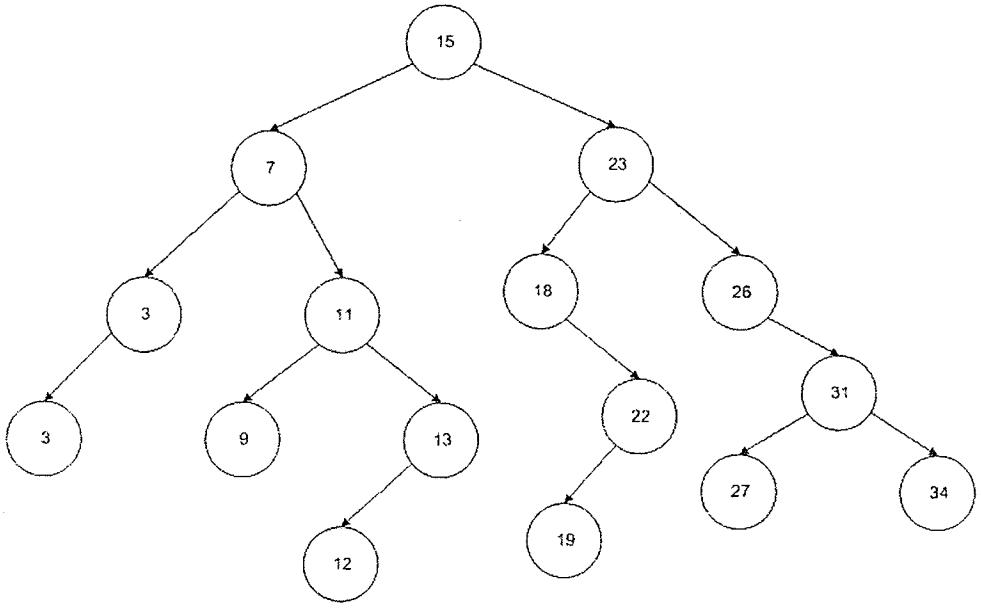


Figure 2.9: A Binary Search Tree (BST).

2.3.1 Binary Search Trees

A binary search tree (BST) [22] is a tree structure in which each node contains a key, and is allowed to have at most two children. The root of the tree is the entry point of the search algorithm. In a binary search tree, any key that is in the subtree rooted at the left child of a node N is less than or equal to the key stored at N, and any key that is stored in the subtree rooted at the right child of N is greater than the key stored in N. A binary search tree allows multiple entries containing the same key. Figure 2.9 shows a binary search tree.

To search for a key in a binary search tree, the search process starts searching for the key in the root. If the given key is not equal to the key that is stored in the root, the process must look for the key in a child of the root. If the given key is less than

the key stored in the root, the process goes to the left child of the root. Otherwise, the process goes to the right child of the root. This process continues to route to the appropriate child of each current node until the process finds the given key. If the search process reaches a leaf of the tree and still does not find the key, it concludes that the given key does not exist in the binary search tree.

2.3.2 2-3 Trees

The 2-3 tree structure (see Figure 2.10) is very similar to the binary search tree structure. The major difference is that 2-3 tree nodes can contain one or two keys, and have two or three children. The keys in a node of a 2-3 tree are stored in sorted order. If the key k_{left} is to the left of the key k_{right}, then $k_{left} < k_{right}$. If the three children of node N are C_{left}, C_{mid}, and C_{right}, where C_{left} is the left, C_{mid} is the middle, and C_{right} is the right child of N, then:

- any key in the subtree rooted at C_{left} is less than or equal to k_{left},
- any key in the subtree rooted at C_{mid} is greater than k_{left} and less than or equal to k_{right}.
- any key in the subtree rooted at C_{right} is greater than k_{right},

The search algorithm in a 2-3 tree works in a similar way to the search algorithm in a binary search tree. A search process in a 2-3 tree starts searching for a key in the root, and it repeatedly moves to a child of the current node until it finds a match for the given key. In a 2-3 tree, if a search process is in a node N, it compares the leftmost key, k_{left}, with the given key. If the current key is equal to the given key, the

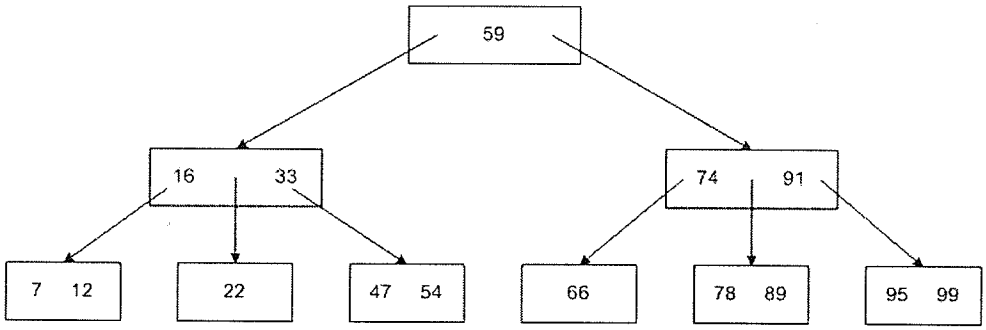


Figure 2.10: A 2-3 Tree.

search algorithm concludes that the key is in node N. If the given key is less than k_{left}, then the search algorithm goes to N's leftmost child, C_{left}. If the given key is greater than k_{left}, then the search algorithm compares the given key with k_{right}. If the given key is equal to k_{right}, then the search algorithm concludes that the key is in the 2-3 tree. If the given key is less than k_{right}, the search algorithm goes to C_{mid}, the middle child of N. Otherwise, the search algorithm goes to N's right child, C_{right}. If the search algorithm goes to the leaf level, yet cannot find the given key in the 2-3 tree, it concludes that the key is not in the 2-3 tree.

2.3.3 B-trees and Their Variants (B^+ Trees, B^* Trees)

The B-tree, introduced by Bayer and McCreight [3], is useful for organizing and maintaining large collections of ordered data. A B-tree is a significantly more efficient index than the search trees described in Section 2.3.1 and Section 2.3.2 when a user wants to index a large collection of many records[3], especially when the records are

[3]A record is a collection of information associated with a key in a tree.

stored on secondary storage. B*-trees, B$^+$-trees, B^{link}-trees and B^{mad}-trees are the key variants of B-trees. I will now describe the properties of, and operations on, regular B-trees and their basic (non-parallel) variants.

B-trees

A B-tree is a tree structure with the following properties:

- Each tree has a branching factor, m, associated with it. The branching factor of a B-tree specifies the maximum number of children that a node can have.
- The root node is either a leaf containing from 1 to $m-1$ keys, or it has d children and $d-1$ keys, where $2 \leq d \leq m$.
- A non-root leaf contains $d-1$ keys, where $\lceil m/2 \rceil \leq d \leq m$.
- A non-root, non-leaf node has d children and $d-1$ keys, where $\lceil m/2 \rceil \leq d \leq m$.
- The leaves of a B-tree are all on the same level of the tree (i.e., the depths of all the leaves are the same).

There are special rules for the keys and pointers in B-tree nodes. Each non-leaf node contains d pointers to its d children. These pointers are referred to as $C_0, C_1, C_2, \ldots, C_{d-1}$. Further, the keys in each node of a B-tree are stored in sorted order. The keys of a node are referred to as $k_0, k_1, k_2, \ldots, k_{d-2}$, where $k_i < k_{i+1}$. In the subtree rooted at such a node, any key less than k_0 resides in the child pointed to by C_0 (or one of its descendants), and any key greater than k_{d-2} resides in the child pointed to by C_{d-1} (or one of its descendants). Any key greater than k_i (where

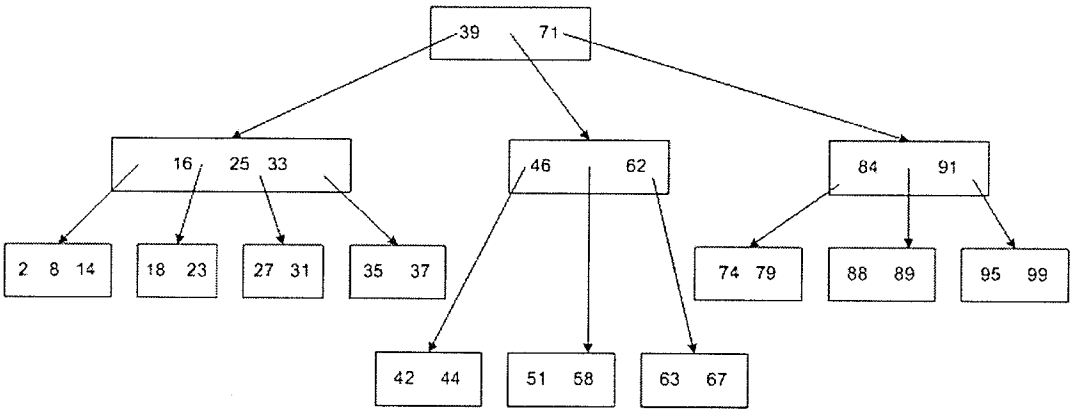


Figure 2.11: A B-tree of order 4.

$0 \leq i \leq m-3$) and less than k_{i+1} resides in the child pointed to by C_{i+1} (or one of its descendants). Figure 2.11 shows a B-tree of order 4.

The branching factor m and the height h of a B-tree determine the cost when a process searches for a particular key in a B-tree. If n is the total number of keys stored in a B-tree, then the height h of the tree is $\log_{\lceil m/2 \rceil}((n+1)/2)+1$. Therefore, a user can store more than 16 million records in only three levels in a B-tree of order 256, whereas a binary search tree needs 23 levels to store the same number of records. Since the cost of searching for a key is directly related to the height of the tree, B-trees take less time for searching than binary trees. This is particularly true for trees on secondary storage where an I/O operation is typically required for each node accessed, but it is also becoming increasingly important in memory as well when locality issues are considered.

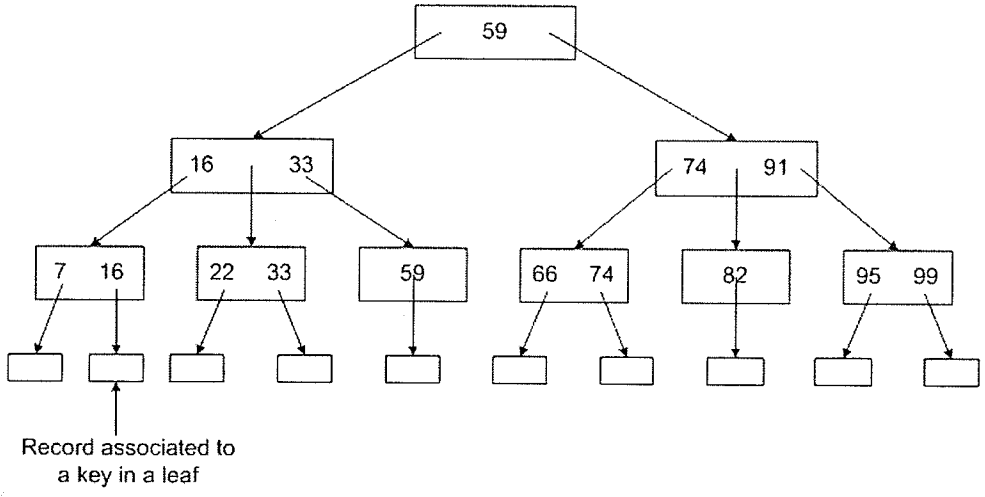


Figure 2.12: A B*-tree of order 3.

B*-trees

Wedekind [38] introduced a variant of the B-tree called the B*-tree. A B*-tree is similar to a regular B-tree, except that the keys and their associated records are stored only in the leaves of the tree. Each key in a leaf is paired with a pointer to the record that is associated with that key. The non-leaf nodes of a B*-tree also contain key values, but those values are not associated with any record. Instead, those keys are only used during searching to locate the keys stored in the leaves. Figure 2.12 shows the structure of a B*-tree of order 3.

B*-tree Operations

To search for a particular key, k, in a B*-tree, the search process starts at the root and proceeds downward to find the leaf that contains k. To move a level down, the search process compares k with the keys in the current node. If $k < k_0$, then

the process follows pointer p_0 to the correct child of the current node. If $k > k_{d-2}$ in the current node, then the search process follows the rightmost pointer p_{d-1} of the current node to find the proper child. If $k_i < k \leq k_{i+1}$, (where $0 \leq i \leq m-3$), then the search process follows pointer p_{i+1}. This procedure continues at subsequent levels until the process finds the leaf node where the key, k, should reside. If the search process finds k in the leaf L, it returns the key k along with the related data from the associated record. Otherwise, the search process returns false since k does not exist in the B*-tree.

A process performs two major steps to complete an insertion operation on a B*-tree. In the first step, the process finds the correct leaf, L, in the B*-tree in which to insert the key k. This step is identical to the search process. In the second step, the process inserts k into the leaf L. If L contains less than $m-1$ keys, then L is an *insertion-safe* leaf since there is room for k. The process terminates by inserting k into L while maintaining the sorted order. If L is full (i.e., if L already has $m-1$ keys), the insertion of k will cause an overflow in L. When this happens, the process splits the leaf L into two leaves $L_{original}$ (the original leaf) and L_{right} (the newly-created leaf). $L_{original}$ is made the left sibling of L_{right}. $L_{original}$ gets the $\lceil m/2 \rceil$ smallest keys and L_{right} gets the $\lceil m/2 \rceil$ largest keys from $L_{original}$. Thus, each of $L_{original}$ and L_{right} gets half of the keys from $L_{original}$, including k, in sorted order. Then, the process inserts a copy of the middle key of L into the parent of L, the node F. Assuming the middle key of L is now the i-th key of F, then the process sets the pointer p_i of F to point to $L_{original}$ and sets the pointer p_{i+1} of F to point to L_{right}. As a result of pushing up the middle key, F may itself overflow. In this case, the process splits F

in the same way and pushes its middle key another level up. This splitting continues until one of the following cases occurs:

1. The process finds a non-full node to accommodate the middle key.

2. The process reaches the root R, and R is full.

In the first case, the insertion process stops since the middle key that was pushed up has been accommodated. In the second case, where the root R is full, the process splits R into two nodes and creates a new root node R_{new}, which contains only the middle key of the previous root R.

Like the insertion operation, the deletion operation also has two major steps. In the first step, the deletion process finds the leaf L containing the key, k, to be deleted. In the second step, k is deleted from L. The leaf L is called *deletion-safe* if L contains more than $\lceil m/2 \rceil$ keys because then a key deletion will not cause an underflow in the B*-tree node.

If L is not deletion-safe, then L needs to merge with one of its adjacent sibling leaves, L'. Suppose the total number of keys in L and L' is n, and that L is immediately to the left of L' (exchange L and L' in the following if L is immediately to the right of L'). If $n > m - 1$, then the keys must be equally distributed between L and L'. Then the key that separates L from L' in their parent, F, must be updated to reflect the re-distribution of keys in L and L'. At this point, the deletion is complete. However, if $n \leq m - 1$, then all keys from L' are shifted to L, and L' is deleted from the tree. Then the key that separates L from L' in their parent, F, is discarded along with the pointer to L'. If F now suffers from underflow, the same merging procedure (with F and an adjacent sibling of F) solves the problem.

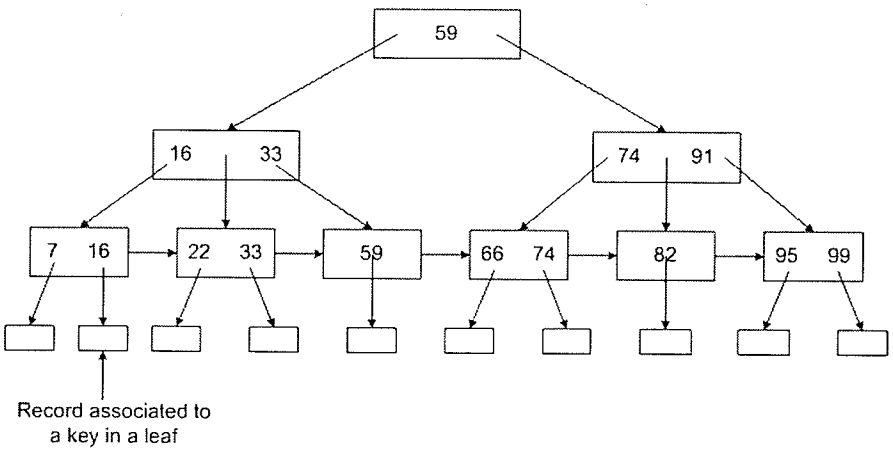


Figure 2.13: A B^+-tree.

If the root has only two children and the two children of the root are merged, then the original root is deleted and the merged child becomes the new root. In this way, the height of the B^*-tree is decreased.

B^+-trees

The B^+-tree, introduced by Comer [8], is another variant of a B-tree. B^*-trees and B^+-trees have similar properties since, in both trees, the keys associated with the records reside only in the leaves of the tree. The only addition in a B^+-tree is that each leaf node of a B^+-tree has an extra sibling pointer that points to the next leaf node to its right. These sibling pointers allow for fast sequential processing of all data in the tree. Figure 2.13 shows a B^+-tree with sibling pointers between the leaf nodes.

Chapter 3

Related Work

This section reviews work that is more directly related to my thesis, focussing mainly on concurrent algorithms for B-trees and their variants.

3.1 Concurrent Algorithms for B-trees and Their Variants

Several concurrent B-tree algorithms were designed for use in databases in the mid 70's, but others are more recent. All of the algorithms presented here are pessimistic lock-based algorithms, where a process locks anywhere between a single node to a portion of the tree while performing an operation on that tree.

The first concurrent B-tree algorithm was developed by Samadi [32], whose approach was very simple. When a process needs to perform an insertion or deletion, Samadi's algorithm locks the entire subtree rooted at the node being manipulated.

Bayer and Schkolnick [4] improved Samadi's algorithm by introducing write-

exclusion locks, where other processes can read, but not write, the data in the portion of the tree locked with a write-exclusion lock. Therefore, even if a process has a write-exclusion lock on a node, other processes can concurrently perform searches through that node, since a search involves only reading the content of each node visited. An update (i.e., insertion or deletion) process must acquire a write-exclusion lock on each node from the root to the appropriate leaf. The write-exclusion locks on those nodes allow other concurrent processes to read the contents of those nodes if necessary. Once the update process finds the proper leaf in which to insert or delete a key, the process acquires an exclusive lock on that leaf, and updates that node (in this case, the leaf). The exclusive lock prevents other processes from either reading or writing that node, and thus maintains the consistency of the tree. The write-exclusion locks increase the concurrency compared to the regular locks used in Samadi's algorithm, but only by a limited amount.

Some other concurrent B-tree algorithms were developed by Miller and Snyder [29], Ellis [13], and Guibas and Sedgwick [16], but none of them make significant enough improvements to the algorithms described above to warrant detailed discussion.

Lehman and Yao [24] came up with an algorithm that introduced a bottom-up restructuring method for B-trees. Most later researchers, such as Lanin and Shasha [23], Biliris [5] and others, were influenced by Lehman and Yao's algorithm. In the following subsections, I describe the B^{link}-tree of Lehman and Yao [24], the overtaking algorithm of Sagiv [31], the symmetric concurrent algorithm of Lanin and Shasha [23], the operation-specific lock algorithm of Biliris [5], the improved overtaking algorithm by de Jonge and Schijf [12], and the B^{mad}-tree algorithm of Das and Demuynck [11].

These works form the core of research related to lock-based concurrency control on B-trees and their variants. Figure 3.1 shows the relationship between the discussed variants of B-trees and algorithms on them.

3.1.1 Lehman and Yao's B^{link}-tree

Lehman and Yao [24] introduced a variation on B*-trees, which they named the B^{link}-tree. The difference between a B*-tree and a B^{link}-tree is that each node in a B^{link}-tree has an extra pointer, known as the link pointer, that points to the node immediately to its right at the same level in the tree. The link pointers of the rightmost nodes at each level are null pointers. When a node is split, the new node is made the sibling immediately to the right of the splitting node, so keys only move to the right in a split. Therefore, as a result of any manipulation in the B^{link}-tree, if any key is moved to a node to the right, the link pointers can be used to find that key. Besides this, each node in a B^{link}-tree has a `highkey` field that contains the value of the highest key in the subtree rooted at N. Figure 3.2 shows an example of a B^{link}-tree.

A process starts a search operation in a B^{link}-tree from the root, and goes down from one level to the next to find the leaf, L, that should contain the key k. When the process is visiting a node, M, and wants to move a level down, the search process compares k with the keys residing in M to decide which child of M is the appropriate child to examine next. If the search process picks N as the appropriate child of M, it compares k with the `highkey`, hk_N, of N. If $hk_N < k$, then the process examines the node immediately to the right of N, and compares k with the `highkey` of that node.

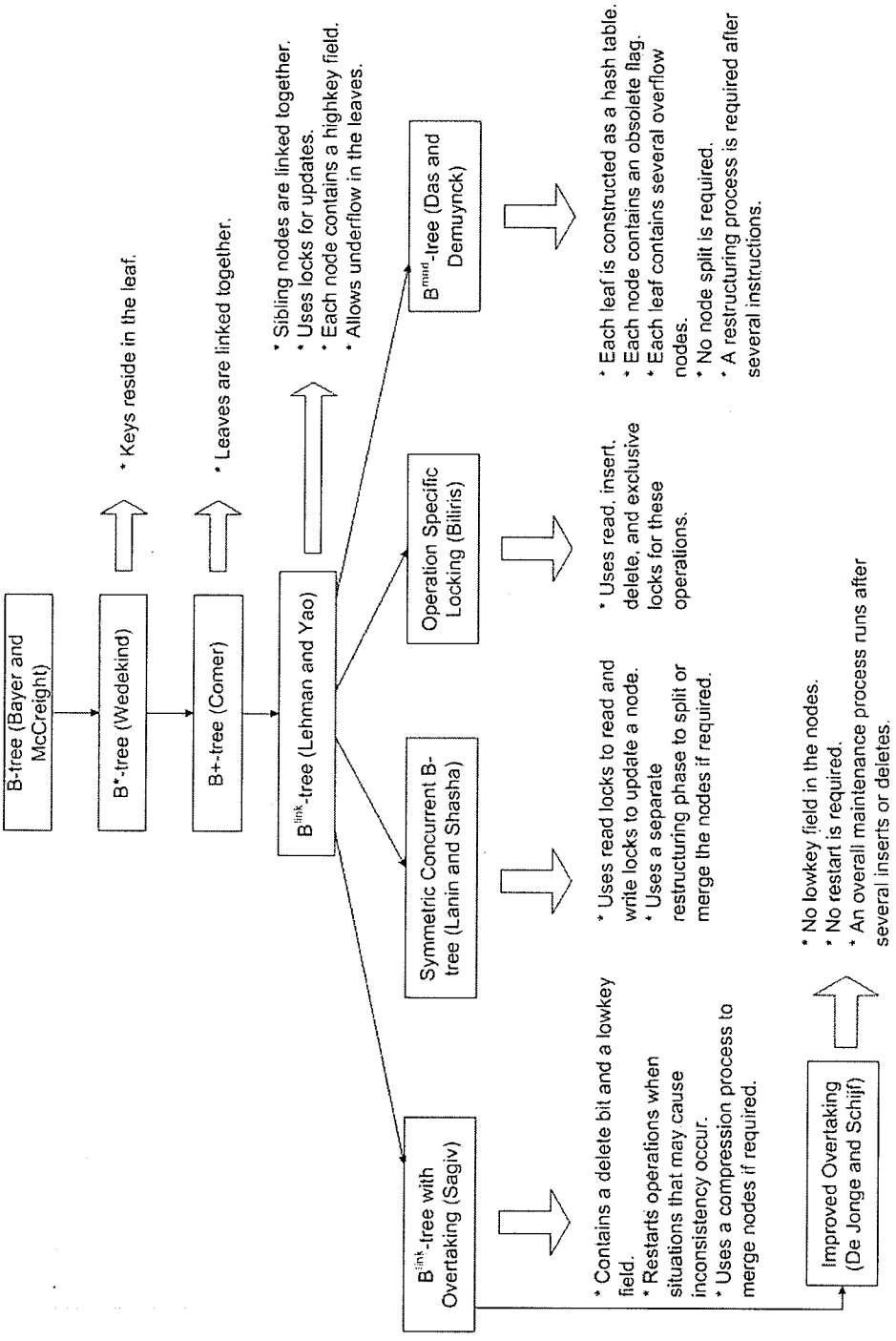


Figure 3.1: Relationship between the B-tree and its variants.

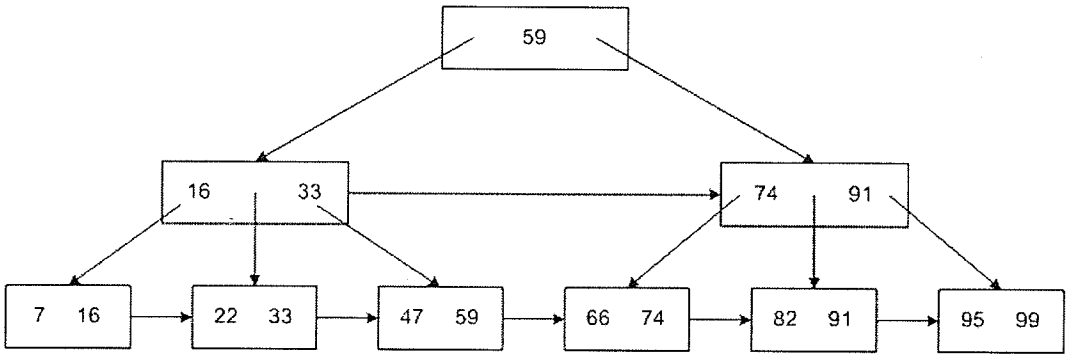


Figure 3.2: An example of a B^{link}-tree.

This process continues until the process finds a node N', where $hk_{N'} \geq k$, at which point the search process knows that this node is the proper child node of M needed to reach L.

After the process moves from a node M to its child N, it again compares the `highkey`, hk_N, of N with k to deal with any issues occurring due to other concurrent operations performed by other processes. If the process finds $hk_N < k$, it follows the link pointer of N to find a sibling node of N to the right where the `highkey` of that node is greater than or equal to k.

Eventually, the process reaches the leaf level and finds the leaf L that contains k. When the search process finds L, then it compares each key of L with the given key k. If the examined key, $k_{examine}$, is less than the given key k, then the search process examines the key immediately after $k_{examine}$. If the search process finds a key in the leaf L that is equal to the given key k, it concludes that the given key k resides in L, and returns L and k. If the search process examines a key in the leaf L that is greater than the given key k, or in the leaf level, while looking for the appropriate leaf to

search for the given key, if the search process comes to the rightmost leaf of the tree and still has not found the proper leaf, the process concludes that k does not exist in the tree. Even though B^{link}-trees use lock-based algorithms, no locking is required during searching in a B^{link}-tree.

Like an insertion in a regular B*-tree, insertion in a B^{link}-tree is performed in two major steps, the search step and the insertion step. The search step is similar to the search process described above. In addition, the insertion process remembers the rightmost node explored at each level of the tree by pushing the node's address onto a stack. The insertion step is also similar to an insertion in a regular B*-tree. However, the insertion process must acquire an update lock on the leaf L so it can insert the key, k, into L. After the insertion is done, the process releases the lock from L. In case of overflow in L, k is inserted into L, and a new node L' is created. The link pointer of L' is set to point where the old link pointer of L used to point, and the link pointer of L is set to point to L'. Then, the rightmost half of the contents of L is moved to L'. After copying the keys to L' the insert process acquires an update lock on F, the parent of L. The node F is found by popping a node from the stack, which is used to remember the path from the leaf to the root traversed by the insert process. Once F is locked, the insert process inserts a copy of the middle key of the original L (before splitting) in the parent F. The pointer from F to L' is set in the same way as in regular B*-trees. If F overflows, necessary steps are taken, in the same way, by splitting F into two nodes, and by linking those nodes to their parents as in the regular B*-tree algorithm. An insertion process locks a node whenever it needs to update that node. After updating a node, the lock is released by the process.

One issue with Lehman and Yao's insertion algorithm is that the nodes stored on the stack may not match the current state of the tree due to other concurrent updates. More precisely, when an insert process gets a parent node F from the stack, due to concurrent splits of F, the node on the stack might not be the current parent of the node N at that moment. To solve this problem, an insertion process compares the highkey, hk_F, of F with the `highkey`, hk_N, of N. If hk_F happens to be less than hk_N, then the insert process has to follow the right link pointer of F to find the appropriate sibling of F that is the correct parent of N.

When a process wants to delete a key, k, from a B^{link}-tree, the search step of the deletion algorithm is identical to that of the insertion algorithm for a B^{link}-tree. In the deletion step, the leaf L is locked by the process and the key k is removed from L. Lehman and Yao's algorithm allows a leaf to have fewer than $\lceil m/2 \rceil$ keys, whereas, in a standard B-tree and its variants, an underfull node is merged with a sibling. Allowing fewer than $\lceil m/2 \rceil$ keys in a node in Lehman and Yao's B^{link}-tree maximizes efficiency since no further locks are required for further merges (as merges are not required). Hence, more concurrency is achieved as other processes can access those unlocked nodes. This enhanced concurrency is useful when different processes perform frequent deletes on a B^{link}-tree. Overall space efficiency may, however, suffer.

3.1.2 Sagiv's B^{link}-trees with Overtaking

The concurrent algorithm developed by Sagiv [31] is very similar to Lehman and Yao's B^{link}-tree algorithm. Like Lehman and Yao's algorithm, Sagiv's algorithm is designed to operate on a B^{link}-tree and Sagiv's algorithm uses only update locks

for insertions and deletions. One big difference between Lehman and Yao's B^{link}-tree algorithm and Sagiv's overtaking algorithm is Lehman and Yao's algorithm do not merge nodes in case of underflow whereas, if a node is underfull because of a key deletion, Sagiv's deletion algorithm merges that node with one of its siblings. Furthermore, during a search or update operation, if any unexpected situation occurs in a node (e.g., if the node has been deleted by another process), the process stops and restarts from the root, since some process has *overtaken* another.

The nodes of the B^{link}-tree in Sagiv's algorithm have fields similar to those of Lehman and Yao's B^{link}-tree. Additionally, each node in Sagiv's B^{link}-tree has two new fields. One field is a `delete bit`, and the other is a `lowkey` field. If a process is about to delete a node, the process sets the `delete bit` of the node to warn other processes that this node is about to be deleted. The `lowkey` field of a node N is similar to the `highkey` field and indicates the lowest key in the subtree rooted at N.

Sagiv's search algorithm is similar to the search algorithm for Lehman and Yao's B^{link}-tree. The only difference is that if a search process accesses a node N with its `delete bit` set by another process, or if the key k to be searched for is less than the `lowkey`, lk_N, of node N, the search process stops and restarts searching from the root again. The process must restart the search because a node N with its `delete bit` set cannot safely be used in searching. Also, if the `lowkey`, lk_N, of node N, is greater than k (which may arise due to concurrent merges caused by other deletions), then k must reside in a node to the left of the node containing lk_N. Since there are no left-pointing links for the nodes in B^{link}-trees, the search process, again, needs to restart from the root.

Sagiv's insert algorithm and Lehman and Yao's insert algorithm are almost identical. One difference is that the search step of Sagiv's insert process remembers the rightmost node accessed at each level so that backtracking is simplified. When the insertion process finds the appropriate node in which to insert the new key, the process locks the node to avoid conflicts due to other concurrent processes. After locking the node, the process inserts the key. The process splits a node, if necessary, in the same way as in Lehman and Yao's algorithm. Further, Lehman and Yao's insertion algorithm ignores the details of new root creation. Sagiv, however, mentions that, to avoid the creation of more than one new root at a time by different concurrent processes, the insertion process must hold an update lock on the previous root. Then it can take the necessary steps to create a new root.

Sagiv's deletion algorithm also has two major steps as in Lehman and Yao's deletion algorithm. While Lehman and Yao's deletion algorithm allows underflow in a node, Sagiv's algorithm periodically runs an independent compression process to ensure that there are at least $\lceil m/2 \rceil$ keys in each node. The compression process locks and examines each node N and each pair of its children A and B, where B is the sibling immediately to the right of A. If the total number of keys in both A and B is less than $m-1$, then the keys of A and B are merged into A, and B is deleted. If B is deleted from the tree, the compression process deletes the key separator of A and B from N, and updates the high key of A in N. If neither A nor B are underfull, the compression process unlocks N, A, and B, and moves to the next sibling nodes C and D, to the right of B. When the compression process moves to C and D, it locks N, C and D. Since the compression process locks the involved parent node and one

pair of its children during the compression, and unlocks all three of them when the compression is done, the compression process can run concurrently with other search, insertion, deletion, or compression processes. If N has an odd number of children, the compression process allows the rightmost child of N to be underfull.

3.1.3 Lanin and Shasha's Symmetric Concurrent B-tree Algorithm

Lanin and Shasha's algorithm [23] also uses locks to do searches and updates. However, a read lock is required to read a node which allows other processes to read, but not write (update), that node concurrently. To insert or delete a key in a leaf, a process must hold the leaf's write lock to prevent other processes from updating that node concurrently.

When a process wants to search for, insert or delete a key using Lanin and Shasha's algorithm, the process needs to first locate the position of the key. This locate phase is similar to Lehman and Yao's search algorithm, but, at each level, when a node N is accessed, the process acquires a read lock on that node. The read lock is released from N when the process finds the proper child node at the next level. The release of the read locks at each level allows more concurrency (since a read lock is incompatible with a write lock on the same node). As a consequence of releasing the read lock from a node N, other concurrent insert or delete processes can acquire a write lock on N if necessary. The insertion and deletion algorithms work in the same way as Lehman and Yao's insertion and deletion algorithms. To insert or delete a key from a selected leaf, a write lock is set on the leaf and then the necessary updates take place.

If the key insertion or deletion does not cause an overflow or underflow in a leaf, the updating process releases the write lock from the leaf. Otherwise, the updating process holds the write lock and calls for a restructuring operation.

A restructuring operation splits a leaf into two when there is an overflow in the node, or merges two nodes into a single node when there is an underflow in the node. The restructuring operation has two steps. The first step is called a *half-split* for insertion, and a *half-merge* for deletion. The restructuring operation creates or deletes a node, and rearranges the keys in the half split or half merge, as needed. After performing the half-split or half-merge, the process releases all locks from the nodes. Therefore, before the completion of the restructuring phase other processes can work on the nodes that are involved in the half split or merge, increasing the concurrency of the update algorithms. During the second step of the restructuring phase, named *add-links* (for insert) or *remove-links* (for delete), the process properly sets up all the pointers to the new nodes from the parent and siblings. If any node at a higher level needs further splitting or merging, the process repeats the steps in the same way.

3.1.4 Biliris' Operation-Specific Locking Algorithm

In Biliris' algorithm [5], four types of operation-specific locks (read lock, insert lock, delete lock and exclusive lock) are introduced. A process holds a specific lock (e.g., an insert lock) for a specific operation (e.g., an insert operation). A read lock is used when any node is accessed for reading during searching. When a node is accessed for an insertion or a deletion operation in one of its descendant leaf nodes,

Table 3.1: Lock compatibility rules in Biliris' operation-specific locking algorithm.

Already acquired a:	**Requesting a read lock**	**Requesting a insert lock**	**Requesting a delete lock**	**Requesting an exclusive lock**
read lock	No	No	No	No
insert lock	Yes	Yes (if insertion safe)	No	No
delete lock	Yes	No	Yes (if deletion safe)	No
exclusive lock	No	Yes	Yes	No

the insert lock or the delete lock is used, respectively. Hence, these locks indicate the intention of performing an insertion (respectively, a deletion) further down the tree. The exclusive lock is used for an update in a node.

Not all lock types are compatible with every other type. Biliris defines specific compatibility rules for pairs of locks ensuring correctness while maximizing concurrency. The rules are given in Table 3.1.

Biliris' search algorithm works the same as Lanin and Shasha's [23] search algorithm since both use read locks. However, according to Biliris' compatibility rules, if any process already holds a node's exclusive lock, another process cannot acquire the node's read lock. So a search process may be temporarily delayed if it attempts to

search through a node that has an exclusive lock on it.

An insertion process starts at the root and initially holds the root's insert lock. An insertion process acquires a read lock on a node after holding an insert lock on that node. The process releases the read lock of the node when it finds the proper child of that node and acquires an insert and a read lock on that child node. When the process finds the proper leaf node, it converts the insert lock to an exclusive lock, and the key is inserted in that leaf. If the leaf is full, the node is split into two nodes as in Lehman and Yao's algorithm. When the insertion is complete, the insert locks are released from all nodes encountered during the insertion.

The deletion process works in a similar way, except that the process holds delete locks rather than insert locks on the nodes that it traverses. After a key deletion, if a node no longer contains sufficient keys, then the node is merged with a sibling. All delete locks are released at the end of the reorganization.

3.1.5 De Jonge and Schijf's Improved Overtaking Algorithm

De Jonge and Schijf [12] suggest some improvements to Sagiv's B^{link}-tree algorithm with overtaking. The insert and delete algorithms of de Jonge and Schijf have two steps. The first step is to search for a key, k, and the next step is either to insert or delete k. Instead of restructuring the B-tree after each insert and delete operation, de Jonge and Schijf suggest an overall maintenance process that runs after several inserts and deletes to restructure the tree as needed. Besides this difference, there is no `lowkey` field in each node as in Sagiv's overtaking algorithm. De Jonge and Schijf's algorithms always move information to the right. Therefore, there is no chance that

the key being searched for will be less than (i.e., to the left of) the `lowkey` of the current node during a search, so no `lowkey` field is needed.

The search algorithm of de Jonge and Schijf is almost the same as Sagiv's algorithm. However, the search algorithm does not need to restart when it encounters a node with its `delete bit` set. When the search algorithm reaches a node with its `delete bit` set, the algorithm follows the node's link pointer to find the correct node to the right at the same level and then moves down to the next level.

There are also slight differences in the insert and delete algorithms between de Jonge and Schijf's and Sagiv's algorithms. In de Jonge and Schijf's update algorithms, the rightmost nodes encountered at each level during the search step need not be remembered. Also, naturally, the update process does not have to be restarted from the root if it encounters a node with its `delete bit` set. The update key step is identical in both Sagiv's and de Jonge and Schijf's algorithm, except for the separate restructuring process.

3.1.6 Das and Demuynck's B^{mad}-tree Algorithm

Das and Demuynck [11] designed a variation on the B-tree, which they refer to as the B^{mad}-tree (B-tree with Multiple Access and Dilation). B^{mad}-trees allow insertions of keys without node splitting. Moreover, B^{mad}-trees allow simultaneous insertions of keys in the same locked node by multiple processes. After a number of insertions, a restructuring algorithm is executed to re-establish the properties of B^{mad}-trees. The B^{mad}-tree algorithm minimizes the overhead of locks used for insertion, and also eliminates the time for node splitting since the insertion algorithm does no splitting.

Therefore, the B^{mad}-tree algorithm achieves noticeable speedup relative to the other lock-based B-tree algorithms.

The B^{mad}-tree data structure is slightly different from the B^{link}-tree data structure. In B^{mad}-trees, each leaf is a hash table with b buckets. The keys in each leaf are stored in semi-sorted order: if L_i and L_j are two arbitrary leaves of a B^{mad}-tree, and if L_i is to the left of L_j, then all keys residing in L_i are less than all keys residing in L_j. Each leaf L in a B^{mad}-tree also contains an `overflow pointer`. When a key needs to be inserted into a full leaf L, instead of splitting L, a new node O with the same structure as a leaf is created to accommodate the new key. The node O is called an `overflow node` for L. The overflow pointer of L is then set to point to O.

Besides the differences in the leaves, each node in a B^{mad}-tree also contains an `obsolete-flag`. During the restructuring phase in a B^{mad}-tree, some nodes may be deleted from the tree. Before deleting any node, the restructuring algorithm sets the `obsolete-flag` of that node to inform other concurrent processes about the upcoming deletion of that node from the tree. Like the `highkey` field of each node, as in Lehman and Yao's B^{link}-tree algorithm, each node of the B^{mad}-tree has a `maximum key` field that contains the largest key associated with that node.

A search process starts at the root node and repeatedly proceeds to a node in the next level to reach the leaf that contains the search key, k. When a search process encounters a new node, it first consults the `obsolete-flag` to check the validity of the node. If the `obsolete-flag` is set, the process proceeds to the node immediately to the right and continues moving right until it finds a node that is not obsolete. When the search process reaches the leaf level, k is hashed to find the appropriate

bucket. If k is not found in the leaf, the search process follows the overflow link to access the chain of overflow nodes. The search process continues until it finds k in an overflow node. If k is not in the overflow chain, the algorithm concludes that k does not exist in the tree. During the execution of the search algorithm, if a concurrent restructuring algorithm is working on the same node as the search algorithm, the search algorithm waits until the restructuring algorithm finishes working with that node. The search algorithm does not hold any locks on any node.

The insertion process works like the search process until it finds the correct leaf, L, in which it will insert the key, k. When the process finds L, and L is insertion-safe, it locks L to insert k. Otherwise, the insertion process locks and follows the overflow pointer to find an overflow node to accommodate k. If no such node exists, the process creates a new overflow node and inserts k in that overflow node. After the insertion, all locks are released by the process.

When concurrent processes perform frequent operations on a B^{mad}-tree, it may cause many overflow nodes in the leaves. Therefore, the tree needs restructuring. The restructuring algorithm works on one level of the tree at a time, starting from the leaf level. The algorithm starts restructuring from the leftmost node in a level, and then proceeds towards the nodes to the right. There are separate restructuring functions for the leaves, the intermediate nodes and the root of the tree.

The B^{mad}-tree restructuring algorithm consumes significant space as the restructuring phase creates new nodes at each level of the tree. However, since nodes are not split during the insertion phase, the insertion algorithm achieves better efficiency than other B-tree algorithms.

3.1.7 Performance Evaluation of Concurrent B-trees

Srinivasan and Carey [34, 35] conducted experiments using some of the B-tree algorithms discussed and published a comparative performance analysis of those algorithms. They divided the algorithms into four categories: the algorithms using exclusive locks (X-LC), the algorithms using shared locks (SIX-LC), the optimistic algorithms (OPT), and the B-link tree algorithms (B-LINK).

Algorithms, such as Bayer and Schkolnick's algorithm, that use exclusive locks to traverse the tree to search for or update a key belong in the X-LC category. The exclusive locks completely shut off the path from the root to the leaf while one process is searching for or updating a key in the tree.

On the other hand, algorithms such as Lanin and Shasha's symmetric concurrent algorithm use read locks from the root to the appropriate leaf in their search and update processes. The read locks allow other concurrent processes to read the locked nodes, which exclusive locks do not permit. These types of algorithms are referred to as SIX-LC algorithms in Srinivasan and Carey's experiments. They differentiate the SIX-LC algorithms into B-SIX algorithms and TD-SIX algorithms. They categorize algorithms like Bayer and Schkolnick's algorithm with shared (SIX) locks as a B-SIX algorithm (the *B* comes from Bayer and Schkolnick's algorithms), and the top-down algorithms, such as Lanin and Shasha's that use SIX locks as TD-SIX algorithms (the *TD* comes from Top-Down algorithms).

The next category introduced by Srinivasan and Carey are the OPT (OPTimistic) algorithms. When an OPT algorithm encounters an unsafe node along a path to the leaf level, the search or update process backs off and restarts the search or update op-

eration from the root. The algorithms designed by Sagiv, and de Jonge and Schijf are examples of OPT algorithms. Srinivasan and Carey categorize the OPT algorithms into B-OPT algorithms, TD-OPT algorithms, and OPT-DLOCK algorithms. Bayer and Schkolnick's algorithm could be categorized as a B-OPT algorithm if it were to restart on detecting a conflict, while the top-down algorithms like Lanin and Shasha's algorithm with back-off could be classified as TD-OPT algorithms. Both the B-OPT and the TD-OPT algorithms restart when they encounter a full leaf. Srinivasan and Carey also introduced a new class of optimistic algorithms, named OPT-DLOCK algorithms, that restart as the other OPT algorithms do, but which also restart in the presence of deadlocks. The OPT-DLOCK algorithms identify deadlocks by recognizing the circular waiting condition required for deadlock.

Lehman and Yao's B^{link}-tree algorithm, naturally, falls into the category of B-LINK algorithms.

These algorithm categories are summarized in table 3.2.

Srinivasan and Carey used mixes of search, insert and delete operations in their simulations to measure the performance of different algorithms under different load conditions. They performed their experiments for three different levels of data contention. Moreover, each experiment, with different levels of data contention, was executed for both high fanout trees (which contained 200 keys/node) and low fanout trees (which contained only 8 keys/node). The results of the experiments are summarized in Table 3.3.

Table 3.2: Algorithm classification in Srinivasan and Carey's experiments.

Type of Algorithms	Sub types	Example
X-LC (exclusive lock-based algorithms)		Bayer and Schkolnick like algorithms
SIX-LC (shared lock-based algorithms)	a) B-SIX	Bayer and Schkolnick like algorithms with shared locks
	b) TD-SIX	Top-down algorithms with shared locks
OPT (optimistic algorithms)	a) B-OPT	Bayer and Schkolnick like algorithms that restart
	b) TD-OPT	Top-down algorithms that restart
	c) OPT-DLOCK	Algorithms that restart in the presence of deadlocks
B-LINK		Lehman and Yao's B^{link}-tree algorithms

3.2 Relevant Lock-free Data Structures

Many data structures, including linked lists [27, 37], queues [28], red-black trees [21, 25], have been implemented using lock-free techniques such as those described in Sections 2.2.2 and 2.2.3. Among them, Valios' [37] and Michael's [27] lock-free linked

Table 3.3: Results of Srinivasan and Carey's experiments.

Data Contention	**Tree Fanout**	**Performance Comparison ("<" means "is better than")**
Low	High	B-LINK and OPT>SIX-LC>X-LC
	Low	**8 disks:** B-LINK & OPT>B-SIX>TD-SIX>X-LC
		∞ disks: B-LINK & OPT>TD-SIX>B-SIX>X-LC
Moderate	High	**8 disks:** B-LINK>TD-OPT>OPT-DLOCK> B-OPT>TD-SIX
		∞ disks: B-LINK>OPT-DLOCK> B-OPT & TD-OPT>TD-SIX
	Low	**8 disks:** OPT-DLOCK>B-LINK> TD-OPT & B-OPT>TD-SIX
		∞ disks: B-LINK>OPT-DLOCK> TD-OPT>B-OPT>TD-SIX
High	High	**∞ disks:** B-LINK>TD-OPT>TD-SIX
		in-memory: TD-SIX>B-LINK & OPT

lists are relevant to my proposed research.

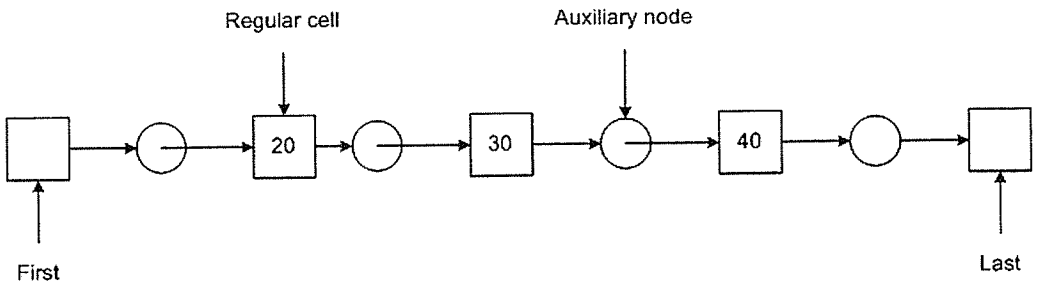


Figure 3.3: Valois' lock-free linked list.

3.2.1 Valois' Lock-free Linked List

Valois [37] designed a concurrent linked-list data structure using the CAS primitive. Like other concurrent algorithms, the main challenge was to maintain the consistency of the linked list. In particular, one process wanting to delete a cell from a linked list that has already been deleted by another process may create an inconsistency in the linked list. To avoid such an inconsistency, Valois introduces *auxiliary nodes*, which contain only a `next` pointer field. In Valois' linked list, each regular cell must be between two auxiliary nodes. However, it is not necessary for the auxiliary nodes to be between two regular cells of the linked list (i.e., the linked list can have consecutive auxiliary nodes). The linked list also contains two dummy cells pointed to by pointers `First` and `Last`. Besides this, Valois' lock-free linked-list contains a list of free cells (the cells that are not currently used in the linked-list). This list is known as the `free list`. Figure 3.3 shows the structure of Valois' lock-free linked list.

A process uses a *cursor* to navigate the linked list. A cursor contains a `target` pointer, a `pre_cell` pointer, and a `pre_aux` pointer. The `target` pointer points to the regular cell of a linked list that a process is currently visiting. The `pre_aux` pointer

points to the auxiliary node that is immediately before the cell pointed to by the `target` pointer. If the `next` field of the auxiliary node pointed to by the `pre_aux` pointer, does not point to the same cell as the `target` pointer, the algorithm decides that some operations have been performed in the linked list by other concurrent processes. The `pre_cell` pointer points to the regular cell that precedes the cell pointed to by the `target` pointer.

The `TryInsert` and `TryDelete` operations are used to insert and delete a node from the list, respectively. After allocating a regular cell and an auxiliary node from the free list, the `TryInsert` algorithm uses a `CAS` operation (see Section 2.2.2) to try to insert the new cell and new auxiliary node before the cell pointed to by the `target` pointer of the process's cursor and after the auxiliary node pointed to by the `pre_aux` pointer of the cursor. Similarly, the `TryDelete` algorithm uses a `CAS` operation to try to delete the node pointed to by the `target` pointer of the cursor. Deleting a node from the linked list may form a chain of auxiliary nodes. The `TryDelete` algorithm also eliminates extra auxiliary nodes created by other concurrent deletion operations on the linked list. After removing the deleted cell and the chain of auxiliary nodes from the linked-list, the deletion process returns them to the free list.

Although Valois uses lock-free techniques for his linked-list algorithms, when a deletion process deletes a cell from the linked-list, and returns the cell to the free list, the process does not do this atomically. Therefore, there is a chance that if multiple processes are trying to delete and return the same cell to the free list, they may interfere with each other. More precisely, at time t_1, two processes may want to delete the same cell, and will read the status of the cell as *not deleted.* Then, at time

t_2, one process deletes the cell and returns the cell to the free list. Since the other process read the status of the cell at time t_1, if it tries to delete the same cell at time t_3, it will still see the status of the cell as *not deleted*. Therefore, the second process will try to delete a cell from the linked list that is now in the free list or, worse, is being allocated as a new cell by some insertion process. Instead of reusing memory from the free list, if Valois' linked list dynamically allocates memory for new cells and auxiliary nodes, then such non-atomic instructions for cell deletion will not cause any problem. This is a known issue with Valois' algorithm.

3.2.2 Michael's Lock-free Linked List

Michael designed a safe wait-free memory reclamation technique [26] for lock-free linked lists, queues, stacks and hash tables. His memory reclamation technique assures that a process can safely return a cell to the free list so that other concurrent processes can reuse the same cell in the future. I describe only Michael's lock-free linked list [27] which uses his safe memory reclamation technique.

Like Valois' lock-free linked list, Michael's lock-free linked list has two dummy cells named `head` and `tail` at the beginning and the end of the linked list, respectively. Moreover, it maintains a free list as Valois' linked list does. However, Michael's linked list does not contain any auxiliary nodes. Figure 3.4 shows Michael's lock-free linked-list structure.

In his safe memory reclamation technique, Michael introduces k (k is usually not more than three) pointers named `hazard pointers` for each process. If a process is working on some cells of a linked list, the hazard pointers of that process point to

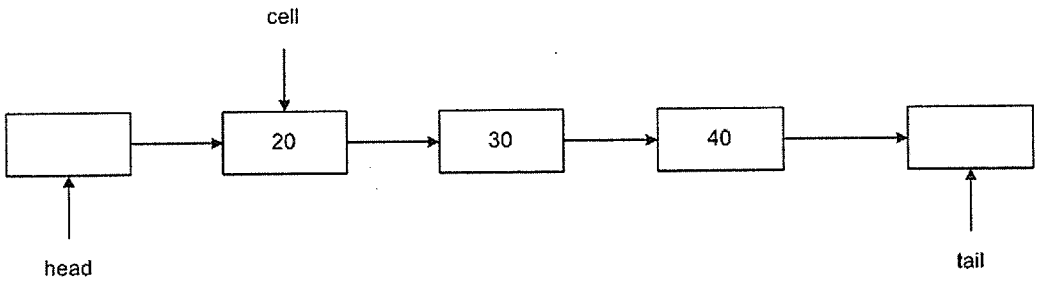


Figure 3.4: Michael's lock-free linked list.

those cells. These hazard pointers can be read by other concurrent processes, but can only be written by the associated process. Besides the hazard pointers, each process has its own stack that is used to store the cells it has deleted. These stacks are known as the `retired-cell-stack`. A `retired-cell-stack` is private to its associated process. Therefore, after the deletion of a cell, when the cell is stored in the `retired-cell-stack`, no process other than the deleter process can access that deleted cell.

The insert algorithm of Michael's lock-free linked list is straightforward. First, an insert process finds the correct (ordered) position in the linked list, and then it inserts the new cell in the linked list using a CAS operation. During these operations, the hazard pointers point to those cells being accessed so that other concurrent deletion processes cannot delete and return those cells to the free list.

The deletion algorithm also works in a similar way. First, it finds the cell to be deleted from the linked list. Then, it must delete and return that cell to the free list, assuming that no other processes are currently pointing their hazard pointers to that cell. To avoid deleting a cell that is pointed to by one or more hazard pointers,

the deletion algorithm does not completely delete a cell from the linked list. Instead, a delete process first *marks* the cell as deleted. To mark a cell as deleted, a delete process adds 1 to the address stored in the cell's `next` field using a CAS operation. Since the least significant bit of any cell address is always 0, adding 1 sets the least significant bit of the `next` field to 1. Thus, the marking distinguishes a deleted cell from a cell that is not deleted.

After marking a cell as deleted, the delete process stores that cell in its associated `retired-cell-stack`. If the number of deleted cells in the `retired-cells-stack` reaches a previously set threshold level, R, where $R = 2 \times k \times$ number of processes, the delete process calls a `scan` routine to return the cells from the `retired-cell-stack` to the free list of the linked list. The scan routine takes a copy of all the hazard pointers of each process. The copies of the hazard pointers are stored in a list named the `p-list`. Once all hazard pointers have been copied by the scan routine, it sorts the `p-list`. Then, the scan routine moves the cells stored in the `retired-cell-stack` into a `temporary-stack`. Then, the scan routine compares the address of each cell in the `temporary-stack` with the addresses stored in the `p-list`. If there is a match, then a concurrent process's hazard pointer is pointing to that deleted cell. In this case, the scan routine re-stores the deleted cell on the `retired-cell-stack`. If there is no match found in the `p-list`, the scan routine returns the deleted cell to the free list of the linked list for further use.

When an insert or delete process finishes its insert or delete operations, it sets all of its hazard pointers to null.

3.3 Lock-free Parallel Implementations of Trees

Besides the simple data structures like stacks, linked lists and queues, Ma [25] and Kim [21] have designed lock-free red-black tree algorithms that are comparatively complicated. The following sections contain brief descriptions of their lock-free red-black tree algorithms.

3.3.1 Ma's Red-black Tree Insertion Algorithm

The concurrent red-black tree insertion algorithm designed by Ma [25] uses CAS-type primitives. Each node of Ma's red-black tree contains a flag field, which indicates that a concurrent process is using the node when the flag is set.

Ma based her concurrent insertion algorithm on the serial red-black tree insertion algorithm from the book by Cormen et al. [9]. Cormen et al.'s serial insertion algorithm performs changes in only a small group of neighboring nodes at any one time. Thus, Ma's concurrent insertion algorithm needs to hold the flags of only a small *local area* of nodes at any one time. Using this approach maximizes concurrency in the tree.

3.3.2 Kim's Red-black Tree Insertion and Deletion Algorithm

Like Ma, Kim also based his concurrent lock-free red-black tree [21] on the sequential red-black tree algorithms of Cormen et al. [9]. Besides this, he uses the *local area* concept from Ma's lock-free red-black tree algorithms. In addition, Kim uses intention markers to indicate that a process wants to move its local area up. The intention markers help to maintain sufficient distance between concurrent processes

so that they do not interfere with each other. Use of CAS and DCAS primitives ensures lock-free insertions and deletions in the red-black trees.

3.4 Other Related Work

The previous work in concurrent B-trees, as reviewed in Section 3.1, was focused on database applications and hence, disk-based operations. As a result, support for fine-granularity parallelism was not an issue and the application of lock-free techniques was not considered. More recently, a small body of work has been done in the area of non-parallel in-memory B-trees. Some of this work has focused on data structures which are cache-conscious [15] (i.e., which perform well due to exhibiting a high degree of spatial locality, which improves their cache performance). Other work has used in-memory B-trees to store the top level of a database index in RAM [40]. In this work, part of the index is replicated across the nodes of a cluster. Note that this is not a parallel B-tree, but a replicated B-tree used in implementing a parallel database. Given the growing importance of in-memory search techniques for very large data sets (e.g., genome searching), designing an in-memory, lock-free B-tree seems timely.

Chapter 4

Problem Description

In this chapter, I first motivate the need for an efficient, in-memory, lock-free B-tree variant and then describe the characteristics of the class of problems that must be addressed using such a new tree structure.

4.1 Motivation

In this thesis, I describe a new B-tree variant, the B^{list}-tree, designed to support in-memory search applications efficiently on modern shared-memory parallel computers. A number of factors have motivated my design choices and these will now be discussed.

4.1.1 Problems with Lock-based Algorithms

In Section 3.1, I reviewed some lock-based concurrent B-tree algorithms. A problem with using such lock-based algorithms is that they do not allow more than one process to share a particular data object at a time (even if no conflict will occur

between those processes). Further, there is a fixed, recurring cost to acquiring locks since an OS system call is normally required. Thus, lock-based algorithms often have relatively high overhead and comparatively little concurrency compared to what lock-free algorithms may potentially offer. Moreover, these pessimistic lock-based algorithms suffer from certain negative scheduling anomalies, such as deadlock, convoying, and priority inversion as described earlier. These are potentially significant issues in highly concurrent systems.

4.1.2 Large Scale In-memory Search Algorithms

Most existing B-tree algorithms have been designed for database applications involving on-disk operations. The use of B-trees, however, is no longer limited to databases. Applications such as text mining (e.g., [10]), gene matching(e.g., [36, 39]), and other such interactive and/or dynamic search algorithms require new approaches to provide efficient concurrency for memory-based operations. Such applications typically require searching very large data collections rapidly, thereby precluding the use of on-disk search structures. Further, in many cases, multiple such independent searches must be carried out at once (e.g., searching for multiple gene sequences concurrently). Considering the problems with lock-based concurrent algorithms and the need for in-memory operations, it seems logical to consider using lock-free techniques for implementing algorithms for these sorts of problems.

4.1.3 Parallel Machine Architectures

As my focus is on in-memory B-trees, I also have to consider the characteristics of different parallel machine architectures to select the most appropriate architecture for my concurrent in-memory B-tree algorithms. There are several different parallel architectures having different properties. Among them, distributed memory architectures (see Section 2.1.1) are popular due to their low cost and good scalability. Unfortunately, they are difficult to program and have limited performance when shared structures must be frequently accessed due to the need to send messages to update the copies of the data structure maintained at each node. Shared-Memory Multiprocessors or SMPs (see Section 2.1.2) are easier to program and perform well, but are costly and typically do not scale up to as many processors as clusters do.

To strike a reasonable balance between cost, performance, and scalability, modern shared-memory machines often exhibit Non Uniform Memory Access (NUMA) times (see Section 2.1.3). A NUMA machine offers the large memory and parallel computation ability required for in-memory search applications, but care will have to be taken in the design of my new B-tree structure to achieve efficient memory access by ensuring good locality of reference.

4.2 Problem Statement

Reviewing the characteristics of existing lock-based B-tree algorithms and considering the characteristics of in-memory search applications discussed earlier, there is a need for a new concurrent B-tree structure that will support faster concurrent ac-

cess than existing structures and which can be used in a wide range of contemporary concurrent in-memory search applications. To satisfy the necessary criteria, I have designed, implemented, and assessed a concurrent lock-free B-tree-like structure that avoids the overhead and negative scheduling anomalies of lock-based algorithms, and which should run efficiently on NUMA-style parallel machines.

Chapter 5

Solution

5.1 The B^{list} Tree

I have named my tree structure the B^{list}-tree (a B^{link}-tree with a list-based node structure). I have based my B^{list}-tree algorithms on Lehman and Yao's B^{link}-tree algorithms [24]. I have chosen the B^{link}-tree because the link pointers in each node support fast searching, which is the primary operation of the applications I am targeting. Though nodes on the same level of my B^{list}-tree are linked with each other as in the B^{link}-tree, I have significantly modified the node structure on my B^{list}-tree to reflect the fact that the tree will be stored in memory, not on disk like a B^{link}-tree.

A B^{list}-tree consists of nodes, both leaf and non-leaf. Each node of a B^{list}-tree has a lock-free `key-list` to store the keys. Each `key-list` consists of a collection of cells where each cell stores a key and descendent nodes with lesser and greater keys, respectively. A B^{list}-tree consists of a root pointer that points to the root node of the tree.

Besides the nodes in the B^{list}-tree, there are some additional structures that help make the tree lock-free and concurrent. Like the B^{link}-tree structure, my B^{list}-tree algorithms use a `path` stack to store the path taken by a process from the root to a leaf of the tree. Besides this, like Michael's lock-free linked-list algorithms [27], my B^{list}-tree algorithms use hazard pointers and a `retired-cell` stack. In addition to these data structures, the B^{list}-tree algorithms use a `please-scan` array with one entry per process to establish communication between the concurrent processes. I describe the data structures that are used in the B^{list}-tree in the following subsections.

5.1.1 Cell

In each node of my B^{list}-tree, the keys are stored as a linked list. I refer to the linked list as the `key-list` of the node. Each `key-list` consists of `cells`, where each cell contains a `key`, two children (a `left-child` and a `right-child`) associated with the key, a `thread-id` to indicate which process (if any) has deleted the cell from the `key-list`, and a `next` field to point to the next cell in the `key-list`. Figures 5.1 and 5.2 show the `cell` structure, and Figure 5.3 shows an example of two consecutive cells in a B^{list}-tree node.

5.1.2 Key-List

Unlike Lehman and Yao's array-based key-list structure in their B^{link}-tree [24], my B^{list}-tree's `key-list` is a lock-free linked list, where keys are stored in sorted order. To design my `key-list`, I have followed the concurrent lock-free linked-list structure of Michael [27] (see Section 3.2.2). However, unlike Michael's lock-free linked list,

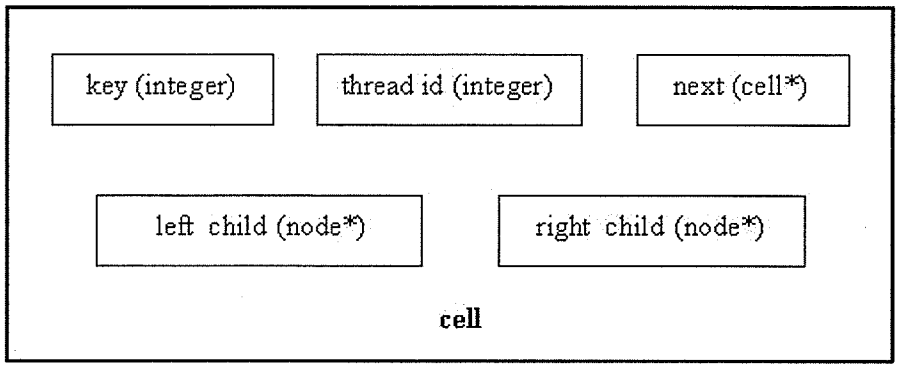


Figure 5.1: The structure of a cell in a B^{list}-tree.

```
class cell{
    int key;              // contains the key
    int thread-id;        // contains the id of the process (thread)
                          // that has deleted this cell
    node *left-child;     // contains the left child of the key
    node *right-child;    // contains the right child of the key
    cell *next;           // contains the address of the next cell
};
```

Figure 5.2: The `cell` class in a B^{list}-tree.

my `key-list` starts and ends with two dummy cells named `head` and `tail`. These dummy cells do not contain any keys. The `next` field of `head` points to the cell that contains the smallest key in the `key-list`, and the `next` field of the cell that contains

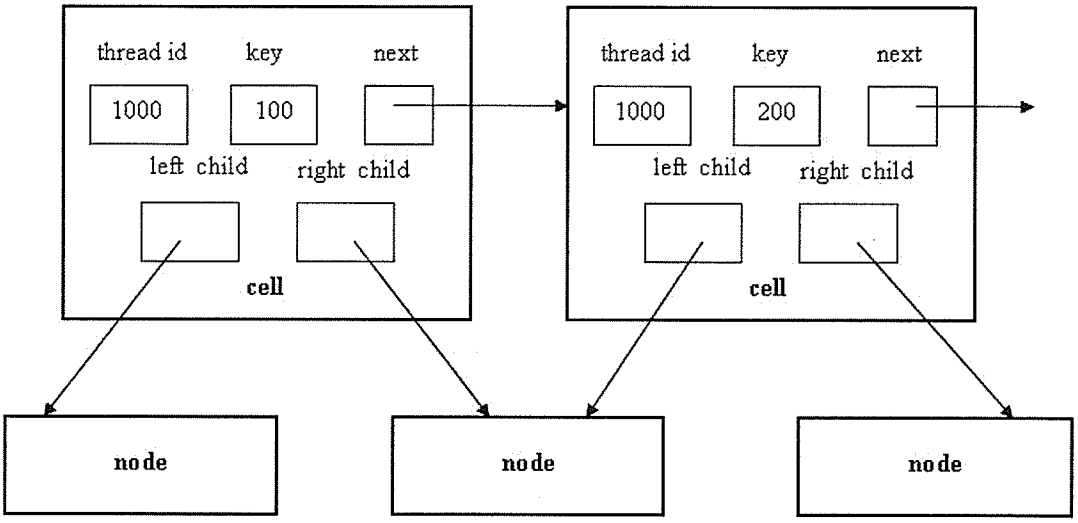


Figure 5.3: An example of two consecutive cells in a B^{list}-tree.

the largest key in the `key-list` points to the `tail` of the key list.

Each `key-list` contains $m+2$ cells, where m is the maximum number of children of a node in the B^{list}-tree (see Section 2.3.3 for details). So that cells within a node are allocated from the same chunk of memory, and therefore have good locality of reference. These cells are implemented as an array of size $m+2$ (pointers between cells are still used to provide the flexibility of a list).

The cells that are not currently being used are referred to as `free cells`, and are stored as a linked list called the `free-list`. The `free-head` pointer points to the first cell of the `free-list`. Figures 5.4 and 5.5 show the `key-list` structure in a B^{list}-tree node. Figure 5.6 shows an example of a `key-list` in a leaf of a B^{list}-tree.

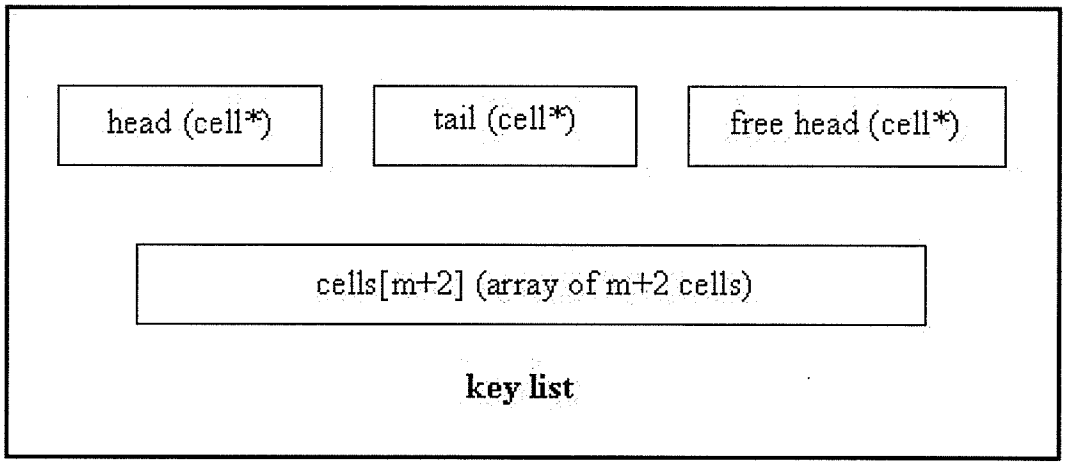


Figure 5.4: The structure of a `key-list` in a B^{list}-tree.

5.1.3 Node

Besides the `key-list`, each node of a B^{list}-tree contains an `is-leaf` field (to indicate whether the node is a leaf or a non-leaf node), a `key-counter` field (to indicate the number of keys currently stored in the node), a `high-key` field (to store the largest key in the descendants of the node), a `split-bit` (to indicate if the node is currently being split by some process), and a `right` field (which points to the node immediately to the right of the node). Moreover, each node of the B^{list}-tree contains a `finally-linked-in-tree` field. When a node is created by a process in a B^{list}-tree, only the process that created the node has authority to access and work on the node. When the creator process turns the node's `finally-linked-in-tree` field on, other concurrent processes working in the tree can access the node. Figures 5.7 and 5.8 show the structure of a `node`, and Figure 5.9 shows an example of a node in a B^{list}-tree.

```
class key-list{
    cell *head;          // the address of the first cell (dummy
                         // cell) in the key-list
    cell *tail;          // the address of the last cell (dummy
                         // cell) in the key-list
    cell *free-head;     // the address of the first cell in
                         // the free cell list
    cell cells[m+2];
};
```

Figure 5.5: The `key-list` class in a B^{list}-tree.

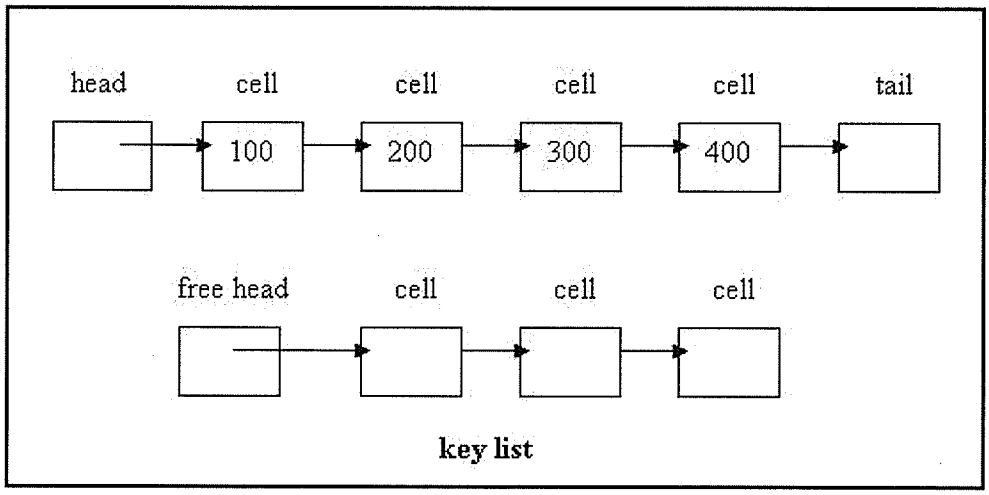


Figure 5.6: An example of a `key-list` in a leaf of a B^{list}-tree.

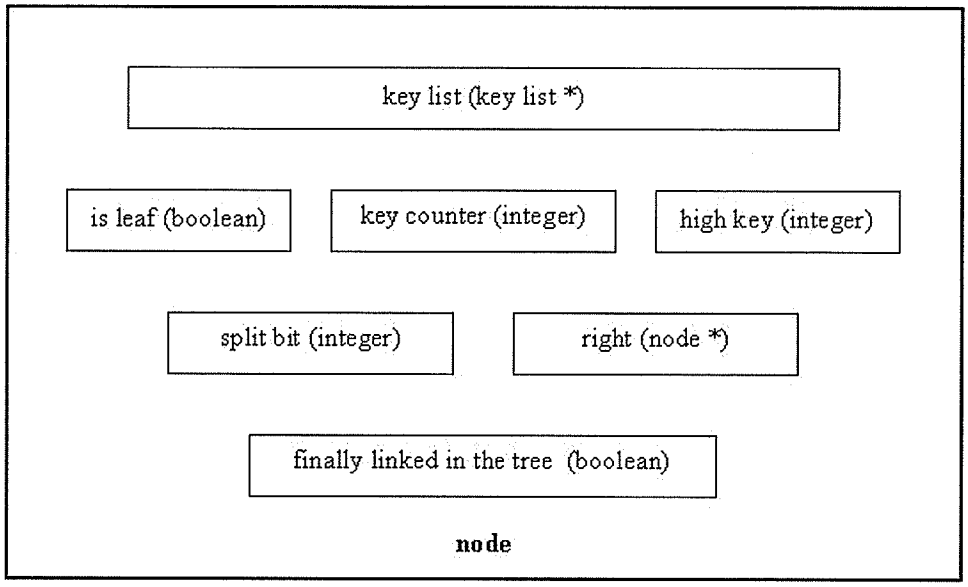


Figure 5.7: The structure of a `node` in a B^{list}-tree.

5.1.4 The Hazard Pointers Array

Since my B^{list}-tree uses a lock-free linked-list structure to store the keys in each node, I needed to implement `hazard pointers` [27] to allow each process to safely perform concurrent operations including deletion on the `key-list`. Two globally-accessible hazard pointers are assigned to each process that may perform operations on the B^{list}-tree. These pointers are use to prevent hazardous concurrent operations (see Section 3.2.2 for a detailed description of how hazard pointers do this). My B^{list}-tree algorithms use two hazard pointers (`hp1` and `hp2`) for each process. A process points its hazard pointers at cells on which the process is currently performing an operation. Hazards pointers keep on pointing to those cells until the the process completes its operation so that any concurrent delete process can be prevented from

```
class node{
    k-list *key-list;    // contains the key-list
    bool is-leaf;        // indicates whether the node is a leaf or
                         // a non-leaf node
    int key-counter;     // contains the number of keys in the node
    int high-key;        // contains the high-key of the node
    int split-bit;       // indicates an ongoing splitting process
    bool finally-linked-in-tree;  // indicates whether the node is
                                  // linked or not in the tree
    node *right;         // contains the address of the immediately
                         // right node
};
```

Figure 5.8: The node class in a B^{list}-tree.

completely deleting a cell (that is in use) from the key-list of the corresponding node.

5.1.5 The Path Stack

Like the B^{link}-tree, each process in a B^{list}-tree has its own path stack that stores the path it has followed from the root to some leaf. Any operation in the B^{list}-tree starts at the root. Until it finds the appropriate leaf, each process stores the rightmost node it encounters on each level of the tree during it search on its path stack (see

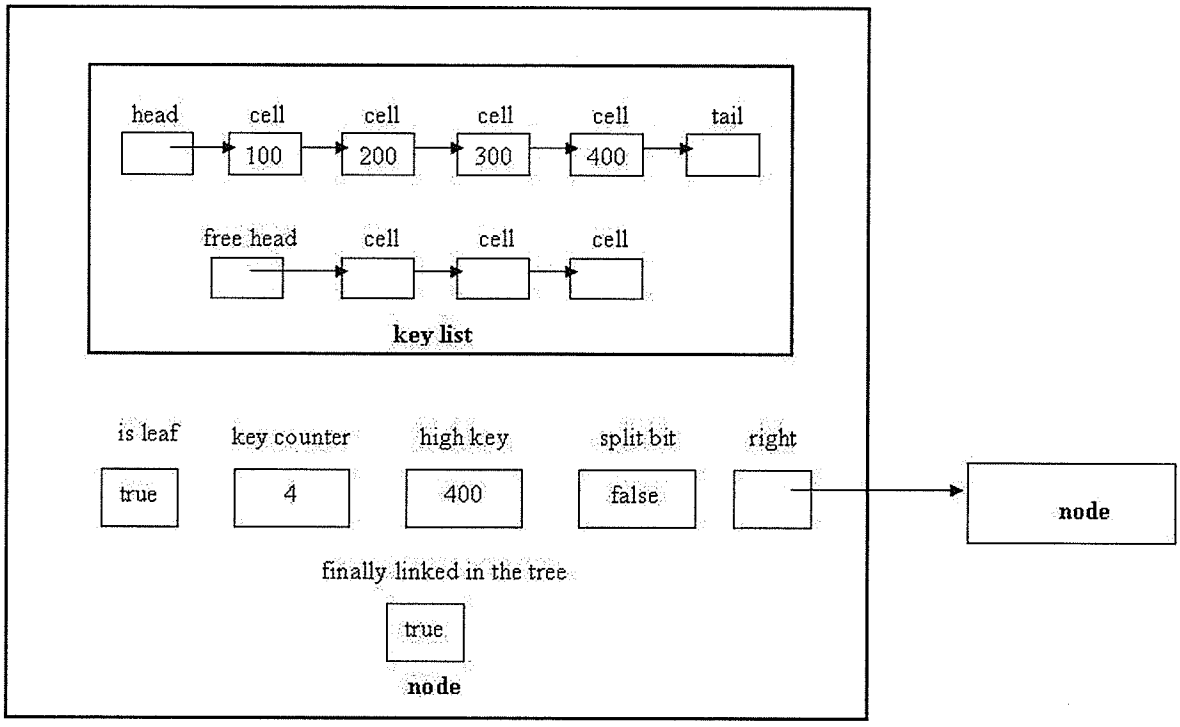


Figure 5.9: An example of a node in a B^{list}-tree.

Section 6.1.1). The stack is later used to find the parent of a node. In an insert operation, when a process is splitting a node, it needs to find and modify the node's parent. An inserting process may also need to find and modify a parent's high-key (see Section 6.1.2). Since the path stack stores the path from the root to a node (or leaf), the parent of that node is stored on the top of the path stack. Therefore, whenever it is necessary to find the parent of a node, a process gets it from the process's path stack. Due to concurrent splits during the insert operations, however, the stored parent may no longer be the parent of a particular node. In this case, the right-link pointer of the stored parent can be followed to find the current parent of

that node.

5.1.6 The Retired-Cell Stack

The B^{list}-tree delete algorithm uses a `retired-cell` stack like the lock-free linked-list delete algorithm of Michael [27] (See Section 3.2.2 to know how Michael's `retired-cell` stack works). Each process has its own `retired-cell` stack which cannot be viewed or accessed by other concurrent processes. After the deletion of a cell, instead of returning that deleted cell to the `free-list`, a process stores the address of the deleted cell in its `retired-cell` stack. The cell remains in the `retired-cell` stack until the process performs the scan algorithm, which returns the cell to the `free-list` if it is safe to do so.

Unlike Michael's lock-free linked list, if a process wants to return a cell from its `retired-cell` stack to the `free-list` in a B^{list}-tree, it has to return the cell to the `free-list` of the node to which the cell belongs. To get the address of the appropriate node, the B^{list}-tree's `retired-cell` stack stores the node address in addition to the cell address.

Like Michael's algorithm, when the number of cells retired by a process reaches a threshold level (for B^{list}-tree algorithms, the threshold level, R, is $m/5$), the process performs a scan routine that examines the retired cells in the process's `retired-cell` stack, and sees if any hazard pointers of any concurrent processes are pointing to those cells. If no hazard pointers are pointing to a cell, the scan routine retrieves the associated node address, and returns the cell to that node's `free-list`. Otherwise, the cell is returned to the process's `retired-cell` stack.

5.1.7 The Please-Scan Array

Every B^{list}-tree has a `please-scan` array shared by all processes. The `please-scan` array contains a `scan-bit` for every process. If a process's `scan-bit` is on, that process is requested to perform a scan by some other concurrent process(es).

In Michael's lock-free linked-list algorithms, only the delete processes check the threshold level, and, if necessary, call the scan routine to return the cells retired by these processes. Similarly, in Section 5.1.6, we have seen that delete processes return a deleted cell to the `free-list` by calling the scan routine in B^{list}-trees. Unlike Michael's lock-free linked-list algorithms which assume that there is only one shared linked list, in B^{list}-trees, each node has its own linked list, which contains a `free-list` of its own. If a process deletes some cells from the same node and stores them in the retired-cells stack, but does not perform a scan to return them in the near future, then the node's `free-list` could become empty. Any update process that is looking for a `free cell` from the list will then have to wait until that delete process returns the retired cells to the `free-list` (see Sections 6.1.2 and 6.1.3). If many processes have to wait for a long time to get a `free cell` from a particular node, there will be a bottleneck. Furthermore, for example, if process P_1 is waiting for a `free cell` that can only be returned by process P_2, process P_2 is waiting for a `free cell` that can only be returned by process P_3, and process P_3 is waiting for a `free cell` that can only be returned by process P_1, then all three processes will be in a deadlock.

To solve this problem, I have added the `please-scan` array to my B^{list}-tree. If a process P_1 is waiting for a `free cell` in a node, N, but the free list is empty, the process P_1 checks its associated `scan-bit` immediately in the `please-scan` array. If

that `scan-bit` is on, the process P_1 calls the scan routine to return the retired cells to their (the cells') associated free lists. If the scan returns some `free cells` in the `free-list` of the node, N, where the process P_1 is waiting, some waiting processes (perhaps P_1) can get `free cells`.

If the scan performed by process P_1 does not return a `free cell` to the `free-list` of node N, process P_1 needs to request other processes to scan so that they can return some cells to the `free-list` of node N where P_1 is waiting for a cell. To send the request, process P_1 goes through each cell of the `key-list` of node N, and looks for a retired cell. If there is a retired cell, then the process P_1 finds the `thread-id` (see Section 5.1.1) of the process that retired the cell to identify which process (say, P_2) has retired this cell. Then, process P_1 turns on process P_2's `scan-bit` in the `please-scan` array. When process P_2 sees that its `scan-bit` is on, it immediately performs a scan and returns some cells to their associated free lists. This action will eventually provide some `free cells` to the processes that are waiting for the `free cells`.

5.2 Benefits of B^{list}-trees

Compared to the B^{link}-tree algorithms and their variants described in Section 3.1, my B^{list}-tree algorithms have some advantages in terms of potential performance. By storing the tree in the main memory of the parallel machine, using a custom linked-list structure to ensure memory locality while maintaining flexibility, and using lock-free operations in the nodes, my B^{list}-tree will potentially be more efficient than concurrent B^{link}-trees.

5.2.1 Benefits of an In-Memory Structure

Access to an in-memory structure will always be faster than one stored on secondary storage. This is necessary for the types of applications I am targeting.

5.2.2 Benefits of the List Structure

In B^{list}-trees, a process does not have to shift the keys in a list-based `key-list` after an update operation, as is required in an array-based `key-list` used in B^{link}-trees. Furthermore, in my B^{list}-tree algorithms, I designed the cells of the `key-list` to be allocated from an array of cells. The benefit of the array of cells implementation is that, when a process accesses one cell in a node, many adjacent cells in the array of that node will also be put in the cache (due to locality of reference since the cells are stored in consecutive memory locations). Hence, searching in the nodes should be fast. My B^{list}-tree algorithms are therefore suitable for shared memory parallel machines. Further, for Non Uniform Memory Access (NUMA) (see Section 2.1.3) machines, with additional effort to localize operations on tree nodes to specific processors, additional locality benefit could be realized at the level of main memory access. In the NUMA architecture, each processor has its own local memory. For a processor, accessing local memory is always faster than accessing non-local memory. By using localized access, NUMA machines can be faster in parallel computations than other parallel machines.

5.2.3 Benefits of Lock-free Concurrency-Control Techniques

All the B-trees and their variants described in Section 3.1 use lock-based pessimistic concurrency-control techniques. Unlike those algorithms, my B^{list}-tree algorithms use lock-free techniques to handle concurrency. Lock-free concurrency control gives my B^{list}-tree a finer granularity of concurrent access, compared to lock-based B-trees and their variants. In B^{list}-trees, each process accesses and manipulates data in smaller chunks (i.e. a cell in a node), whereas a process in a B^{link}-tree accesses and manipulates data in bigger chunks (i.e. a node in the tree). Moreover, in a B^{link}-tree, a process acquires the locks of nodes one at a time. Any concurrent process that wants to work on the same node has to wait until the process that holds the lock of the node finishes its job, and releases the lock. However, in a B^{list}-tree, more than one process can operate on a single node (also, with restrictions, on a single cell) simultaneously. If the result of an operation of one process affects the results of the operation of other concurrent processes, the other concurrent processes either have to redo the work in that cell or the processes have to go to the beginning of the `key-list` of that node and restart their work within that node, yet all processes can redo their work simultaneously. If the concurrent operations have no effect on each other, all processes can continue with their operations simultaneously without any interruption. In a B^{list}-tree, if N processes want to operate in the same node and each operation takes time t, if there is no interference between them, all processes will finish in time t. In a B^{link}-tree, however, all processes will finish in time $N \times t$ (since only one process can work in the node at a time), even if there is no interference between concurrent processes. Therefore, if there is high data access contention in both trees (i.e., many

processes want to concurrently access the same node), the B^{list}-tree algorithms should take comparatively less running time to complete an operation than the lock-based B-tree algorithms and their variants.

Besides this, as described in Section 2.2.4, lock-free concurrency-control techniques assure that my B^{list}-tree algorithms will be free from convoying [41] and priority inversions [33].

Section 2.2.4 also shows that lock-based concurrency-control techniques are prone to deadlocks [6], but the lock-free concurrency-control techniques are not. To prevent deadlocks, my lock-free B^{list}-tree uses a `please-scan` array where every process has a `scan-bit` that is associated to their process identification number (see Section 5.1.7). In the B^{list}-tree update algorithms, if a process P_1 needs a free cell, then P_1 scans its own `retired-cell` stack to get a free cell for itself. If the scan cannot provide any free cell for P_1, then P_1 requests other process(es) (say process P_2) that can reclaim a free cell for the process P_1. To send a scan request to process P_2, process P_1 sets the associated `scan-bit` of process P_2 in the `please-scan` array. In a B^{list}-tree, before starting an operation each process checks their associated `scan-bit` in the `please-scan` array. If the associated `scan-bit` of a process is set, then the process scans its `retired-cell` stack to reclaim free cells.

Without the use of the `please-scan` array in a B^{list}-tree, if process P_1 waits for a cell that can only be reclaimed to the `free-list` by process P_2, and process P_2 is waiting for a cell that can only be reclaimed to the `free-list` by process P_1, then both P_1 and P_2 have to wait forever to get a free cell for themselves. This waiting for free cells will ultimately cause a deadlock in the tree. However, in B^{list}-trees

with a `please-scan` array, when both processes P_1 and P_2 are waiting for a free cell, both processes scan their own `retired-cell` stacks for free cells. If the scans cannot release a free cell for themselves, then process P_1 sends a scan request to process P_2 through the `please-scan` array, and vice versa. When both processes see the scan requests, both processes scan their `retired-cell` stacks and reclaim a free cell (if the cell is not in use by other concurrent processes) for the requested process (i.e., P_1 reclaims a free cell for P_2, and P_2 reclaims a free cell for P_1). This free cell reclamation prevents the potential deadlocks in the B^{list}-tree, and makes the B^{list}-tree algorithms to be deadlock-free.

Chapter 6

Algorithms

6.1 Algorithm Overview and Discussion

My B^{list}-tree algorithms can perform three major operations: search, insert, and delete on a B^{list}-tree. The search operation looks for a particular key in the B^{list}-tree, the insert operation inserts a new key in the tree if that key is not already in the B^{list}-tree, and the delete operation deletes a desired key from the B^{list}-tree, if it exists in the tree.

Sections 6.1.1, 6.1.2 and 6.1.3 describe the search, insert and delete algorithms for a B^{list}-tree, respectively. Section 6.2 discusses the concurrent correctness of the algorithms.

6.1.1 Search Algorithm

The search algorithm works in two major steps. First, the algorithm finds the leaf in the B^{list}-tree where the desired key should reside. Next, the algorithm looks for

the desired key in that leaf.

Step 1: Find the Appropriate Leaf

The search process finds the appropriate leaf in several steps. If the search process searches for the desired key in an empty B^{list}-tree, then the search process terminates right away, and knows that the desired key is not found in the B^{list}-tree. Otherwise, the search process starts looking for the appropriate leaf by looking at the root first, and then, by examining one node at each level, the process propagates to the leaf level. In each level where the node to be examined is a non-leaf node, the search process examines the keys from the key list of the selected node, and tries to find the appropriate child to move to, that is, the root of the sub-tree containing the appropriate leaf and the desired key.

Before starting the key comparisons within a node N, a search process sets one hazard pointer (`hp1`) to point to the `head` of the node's `key-list`, and then sets another hazard pointer (`hp2`) to point to the cell immediately after `head` (see Sections 3.2.2 and 5.1.4 for a description of hazard pointers). The cell pointed to by `hp2` is the leftmost cell in the `key-list`, and contains the smallest key in the `key-list`. (From this point on, I will refer to the cell pointed to by `hp1` as the previous cell, and I will refer to the key stored in the previous cell as the previous key. Furthermore, I will call the cell pointed to by `hp2` the current cell, and the key stored in that current cell as the current key.) Figure 6.1 shows the initial locations of the hazard pointers `hp1` and `hp2` in the `key-list` of a node N. Suppose a search process is looking for the key 350 in a B^{list}-tree. To find the appropriate leaf, in Figure 6.1, it sets `hp1` to

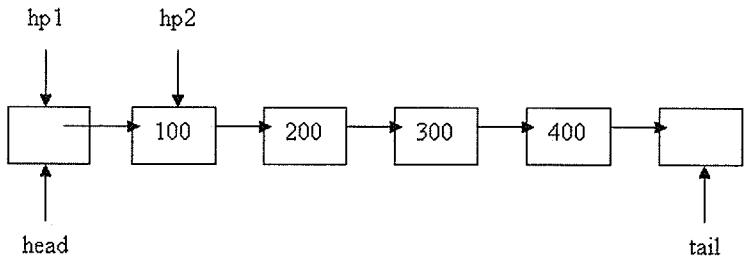


Figure 6.1: Initial locations of the hazard pointers in a node while searching.

point to the `head` of the `key-list`, and `hp2` to point to the cell containing 100, which is the cell immediately after `head`.

After setting the hazard pointers to their initial locations, the search process first compares the key to be searched for with the `high-key` of the node N in case other concurrent processes have split this node. If the `high-key` of the node N is less than the key to be searched for, a split has occurred, so the process follows the `right` link pointer of N to examine the node, R, that is immediately to the right of N. If the `high-key` of the node N is less than the key being searched for, then the search process compares the key to be searched for with the current key. If the current key is less than the desired key, the search process knows that the desired key might be in any of the following cells of the `key-list`. Therefore, it moves `hp1` and `hp2` to the immediately following pair of cells in the `key-list`. Before moving the hazard pointers, the search process examines whether the previous cell has been marked as deleted by another concurrent process, or if the current cell is not the immediate next cell of the previous cell. Both cases prevent the search process from moving to the cells immediately to

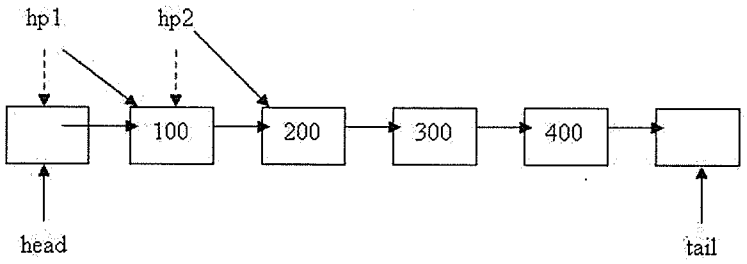


Figure 6.2: Move the hazard pointers to the immediately following cells in the `key-list`.

the right of the previous cell and the current cell. To avoid such inconsistencies, some concurrency-control techniques are applied during the search operation. Please see Section 6.2 for a description of the concurrency-control techniques used in the search operations.

In Figure 6.1, since the current key 100 is less than the desired key 350, the process moves `hp1` and `hp2` to the immediately following cells. After moving the hazard pointers, in Figure 6.2, `hp1` points to the cell containing 100, and `hp2` points to the next cell, containing 200.

Each time the search process moves the hazard pointers to the next cells, it compares the desired key with the current key. It keeps on moving the hazard pointers until it finds a current key that is greater than or equal to the desired key. After finding such a key, the search process knows that the `left-child` of the current cell is the root of the sub-tree that contains the leaf for which it is searching. In Figure 6.3, the current key, 400, is greater than the desired key 350 (also, the previous key 300 is less than the desired key). Therefore, the `left-child` pointer of the current cell

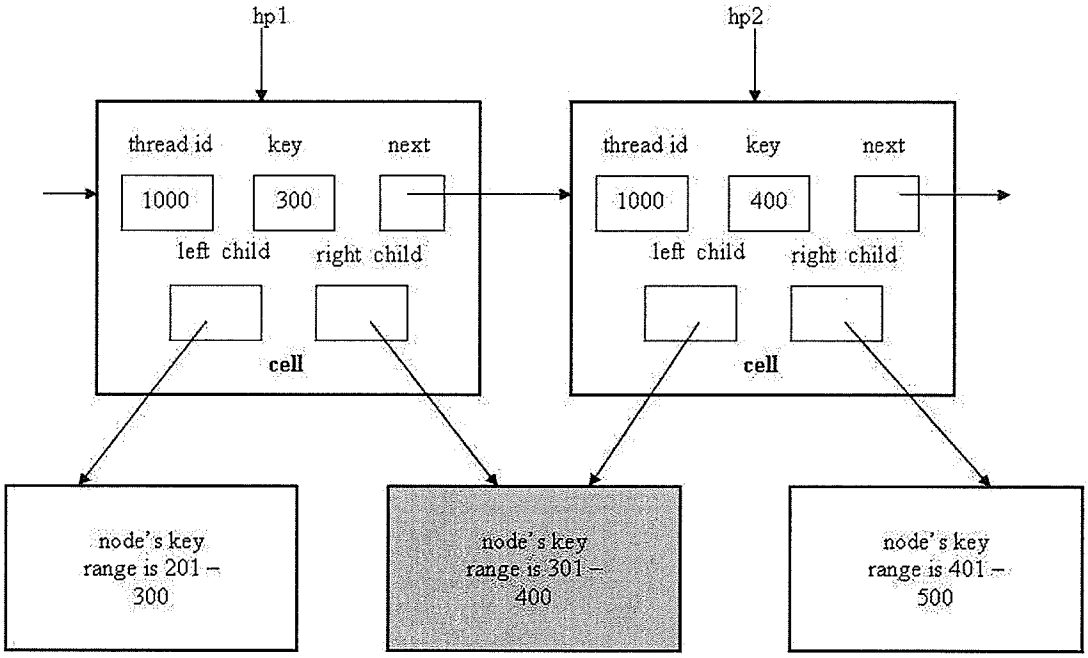


Figure 6.3: The appropriate child (shaded node) is selected.

(also the `right-child` pointer of the previous cell) points to the root of the sub-tree containing all keys valued from 301 to 400. Thus, the search process chooses the left child of the current cell to examine next in the search for the appropriate leaf containing the desired key, 350.

If the search process cannot find any current key in the `key-list` that is greater than the desired key and reaches the end of the `key-list` (i.e., `hp1` points to the cell that contains the largest key in the `key-list`, and `hp2` points to the `tail` of the `key-list`), the search process knows that the `right-child` of the previous cell (pointed to by `hp1`) points to the correct child of this node in which to find the appropriate leaf (see Figure 6.4). Suppose the search process is searching for 450,

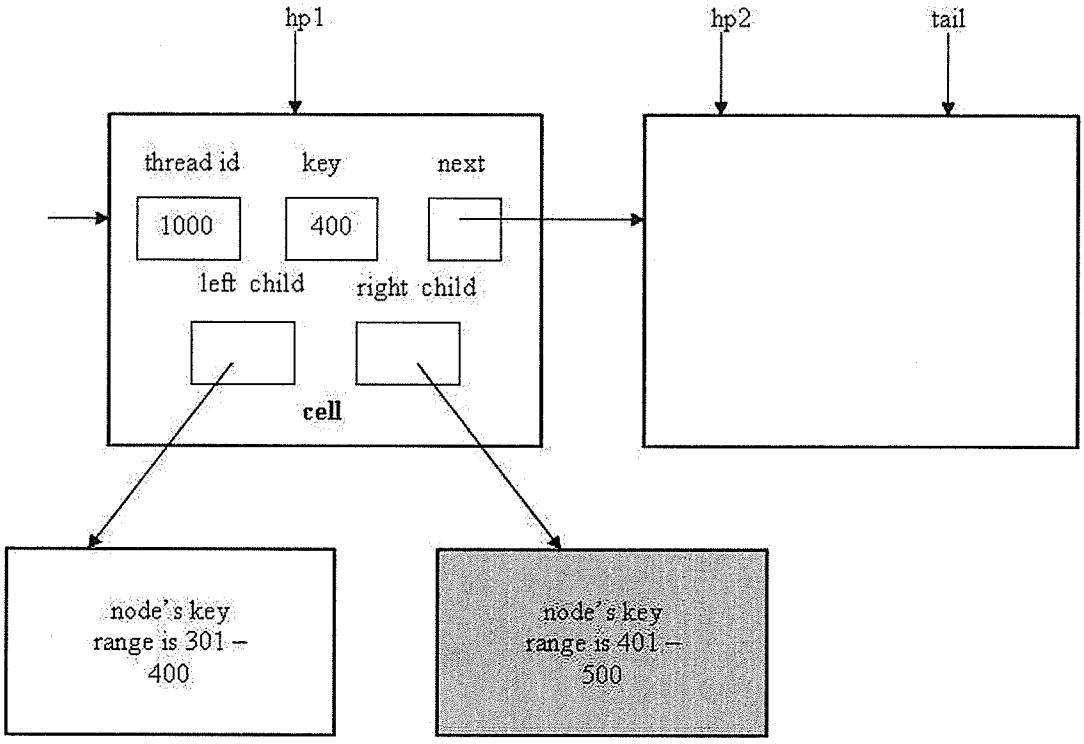


Figure 6.4: The rightmost child (shaded node) is selected.

but the largest key in the `key-list` is 400, and the `high-key` of the current node is 500. In Figure 6.4, `hp1` points to the rightmost cell of the `key-list` containing 400, and `hp2` points to the `tail` of the `key-list`. Thus, the sub-tree rooted at the `right-child` of the cell pointed to by `hp1` (previous cell) contains the keys from 401 to 500. Thus, the search process looks in the `right-child` of the previous cell in the search for the appropriate leaf containing 450.

Before navigating to the selected child, the search process stores the currently examined node in the `path` stack (see Section 5.1.5) for further use. The search process continues to move from each node to a child node until it finds a leaf node.

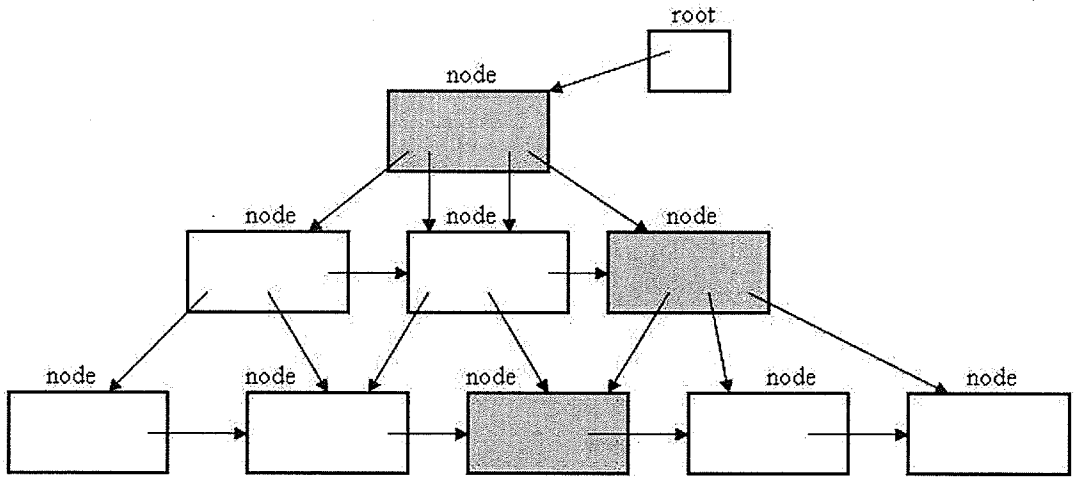


Figure 6.5: The shaded nodes are examined and stored in the `path` stack by the search process to find the appropriate leaf.

Figure 6.5 shows an example of the nodes that are examined in this step and stored in the `path` stack by the search process.

Step 2: Find the Desired Key in the Leaf

When the search process finds the appropriate leaf for a desired key, it compares the keys from that leaf's `key-list`, one by one, with the desired key. First, it sets one of its hazard pointer (`hp1`) to point to the `head` of that leaf's key list, and sets the other hazard pointer (`hp2`) to point to the cell immediately to the right of the cell pointed to by `hp1` (see Figure 6.6). Then, the search process compares the current key with the desired key. If the current key is less than the desired key, the search process knows that the desired key may be in this leaf. Therefore, the search process moves `hp1` to point to the cell currently pointed to by `hp2`, and then, it moves `hp2` to point to the cell immediately after the cell now pointed to by `hp1` (see Figure 6.7).

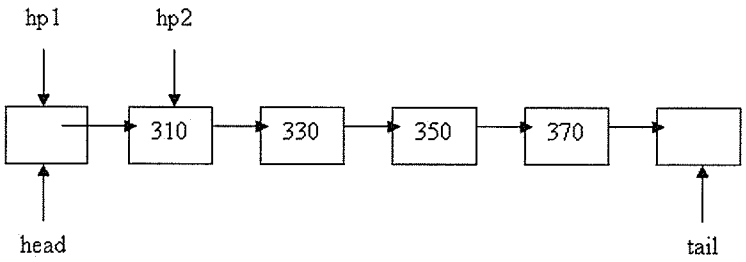


Figure 6.6: Initial hazard pointer locations in a leaf node.

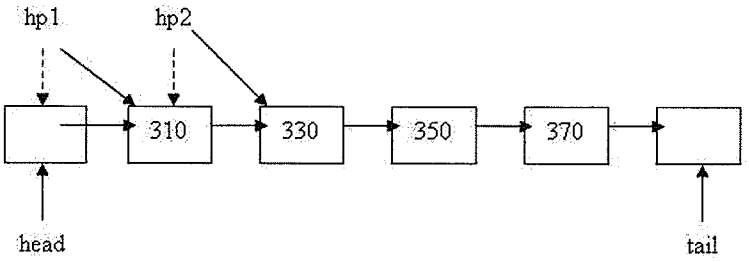


Figure 6.7: Move the hazard pointers one cell ahead in the `key-list`.

In our example, the search process (looking for 350), compares the current key, 310, with the desired key 350 (see Figure 6.6). Since the current key, 310, is less than the desired key, 350, the search process moves its hazard pointers to the immediately following cells (see Figure 6.7).

The search process keeps on moving `hp1` and `hp2`, and comparing the current key with the desired key, until it finds a current key that is greater than or equal to the desired key. If the search process finds a current key that is equal to the desired key, then it has found the key in the leaf, and returns success. In Figure 6.8, the current key is 350, and is equal to the desired key 350. Therefore, the search process

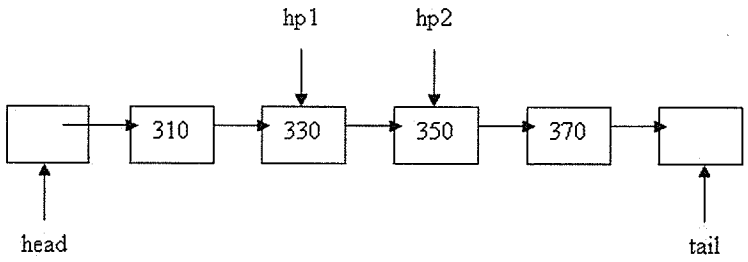


Figure 6.8: The search process finds the desired key in the leaf.

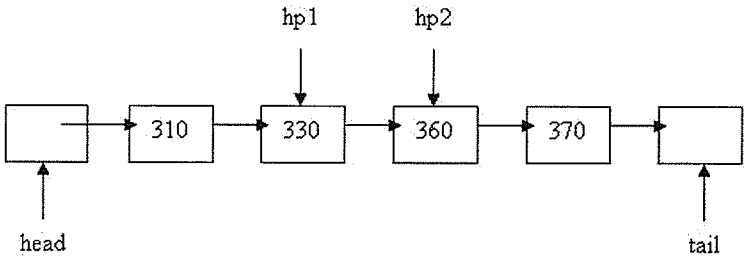


Figure 6.9: The search process could not find the desired key in the leaf.

ends with success. Since the keys in the `key-list` in a node of a B^{list}-tree are stored in sorted order, if the search process finds a current key that is greater than the desired key, then it knows that the desired key does not exist in the leaf. In this case, the search process empties the `path` stack and terminates with a failure result. In Figure 6.9, the current key is 360, which is greater than the desired key, 350. Thus, the search process concludes that 350 does not exist in the B^{list}-tree. If the search process examines all keys from the `key-list` (i.e., `hp1` points to the rightmost cell of the `key-list`, and `hp2` points to the `tail` of the `key-list`), but cannot find the desired key, then, again, it knows that the key is not in the leaf (see Figure 6.10).

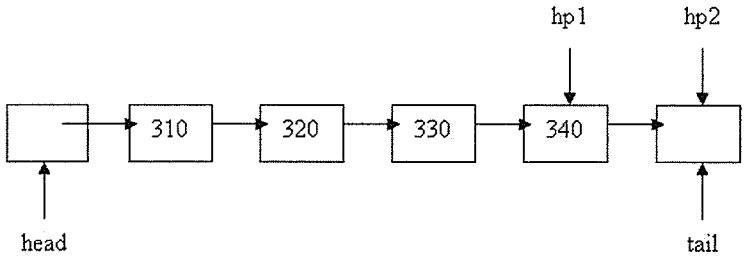


Figure 6.10: The hazard pointers point to the end of the `key-list`, and the search process could not find the desired key in the leaf.

6.1.2 Insert Algorithm

The insert algorithm for a B^{list}-tree is the most complicated algorithm, compared to the search and the delete algorithms. An insertion in a B^{list}-tree works in several steps. In the first step of the insert algorithm, the insert process looks for the new key (the key to be inserted) in the B^{list}-tree. If the key already exists in the tree, the insert process terminates. Otherwise, it inserts the new key in the appropriate leaf of the B^{list}-tree. As a result of the insertion, if the leaf overflows, the process splits that leaf into two leaves, and inserts a copy of the middle key of the original leaf (before it was split) in its parent node. If the middle key insertion causes an overflow in the parent, the insert process splits the parent into two nodes, and inserts the middle key of the parent into the parent's parent. The insert process keeps on splitting a node into two nodes and pushing the middle key of each node to its parent until the parent node does not overflow. In the last step, the insert process updates the `high-key` of the parent node, if required. Figure 6.11 shows a high level flowchart of the insert algorithm.

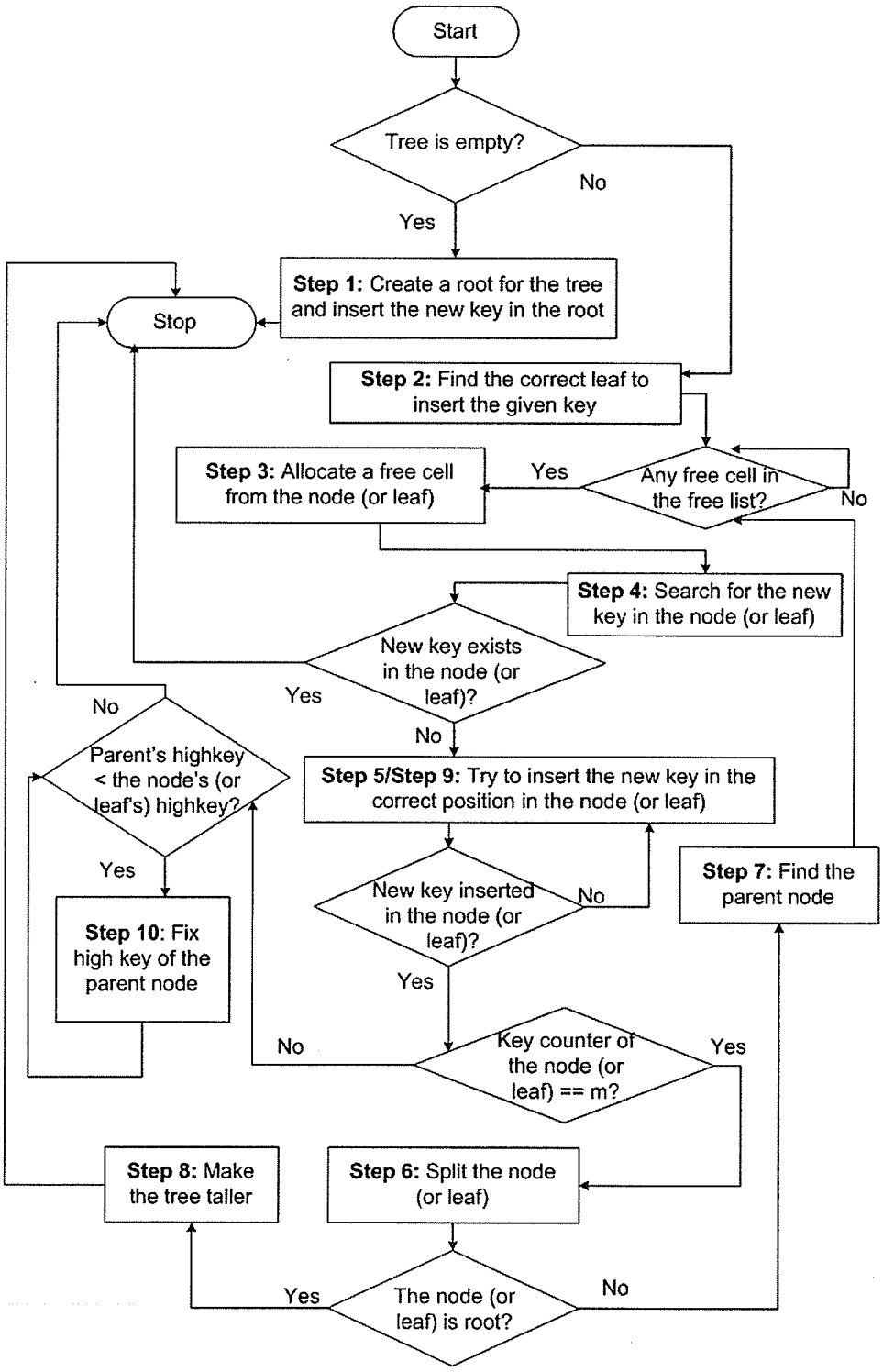


Figure 6.11: The key steps of the insert algorithm.

Step 1: Create a Root

Inserting a key in an empty tree is different than inserting a key in a non-empty tree. If an insert process wants to insert a key in a non-empty B^{list}-tree, the insert process does not execute step 1, it goes directly to step 2 to find the appropriate leaf. Otherwise, if the tree is empty, the process creates a new leaf node for the tree (Figure 6.12). Then, it allocates a cell from that leaf's `free-list`, and copies the new key into the `key` field (see Section 5.1.1) of that cell (Figure 6.13). After that, the insert process inserts that cell between the `head` and the `tail` of the leaf's `key-list` (Figure 6.14). Then, it increases the `key-counter` of the leaf, and updates the `high-key` of the leaf with the new key (Figure 6.15). In the last step, the insert process uses CAS (see Section 2.2.2) to assign the newly-created leaf's address to the `root` of the B^{list}-tree. If other processes already have created another root for the B^{list}-tree, the CAS fails, and the insert process goes to step 2. Otherwise, the CAS successfully assigns the new leaf as the root of the B^{list}-tree. Finally, the insert process makes the new root of the tree accessible to other processes by setting the `finally-linked-in-tree` bit (see Section 5.1.3) of that leaf. From this point on, the new root is available to other concurrent processes.

Step 2: Find the Appropriate Leaf

In the case of inserting a key into a non-empty tree, the insert process needs to find the appropriate leaf for the key insertion. The search for the appropriate leaf in an insert algorithm in a B^{list}-tree is identical to the search for the appropriate leaf in the search algorithm. See step 1 of the search algorithm in Section 6.1.1 for details.

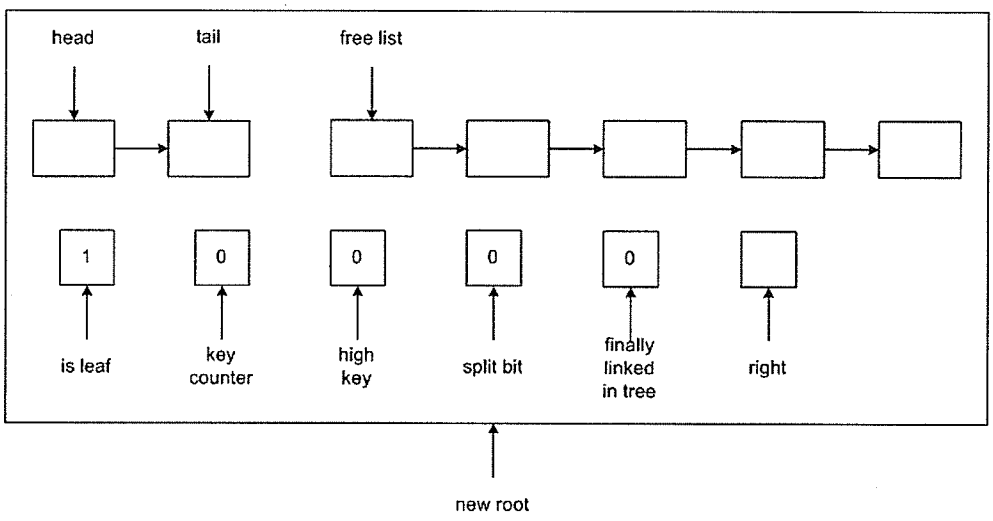


Figure 6.12: The insert process creates a new leaf for an empty tree.

Step 3: Allocate a Cell for the New Key

The insert process needs to allocate a cell from the `free-list` of the leaf to store the new key. Before it allocates the new cell, it checks the status of the `split-bit` (see Section 5.1.3) of that leaf. If the `split-bit` is on, another concurrent insertion process is splitting the same leaf. Therefore, this insert process has to busy wait until the other inserter completes the split process and turns off the leaf's `split-bit`.

Once the `split-bit` is off, the insert process checks whether the leaf is still the appropriate leaf in which to insert the new key after the split operation (see Section 6.2 for a description of how the insert process handles concurrent splits while allocating a new cell for the new key). Once the insert process is in the correct node, it looks for an empty cell in the `free-list` of that leaf's `key-list`. If there is no cell in the `free-list`, the insert process checks the status of the `split-bit`, and the

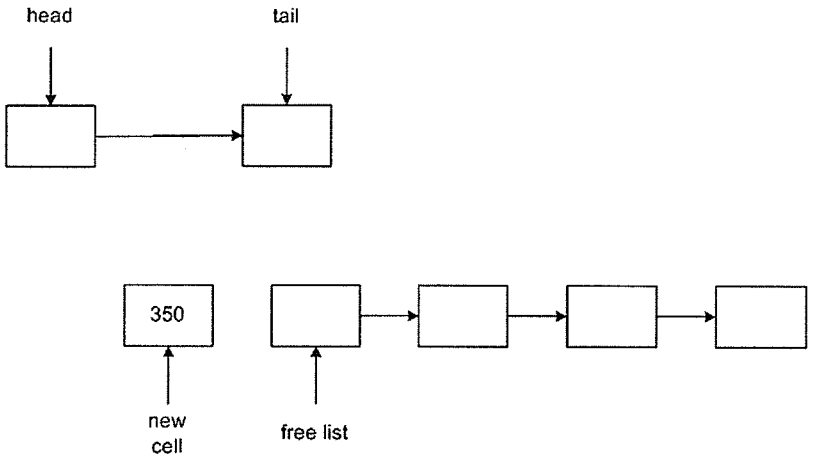


Figure 6.13: Allocate a new cell from the `free-list`, and copy the new key into that cell.

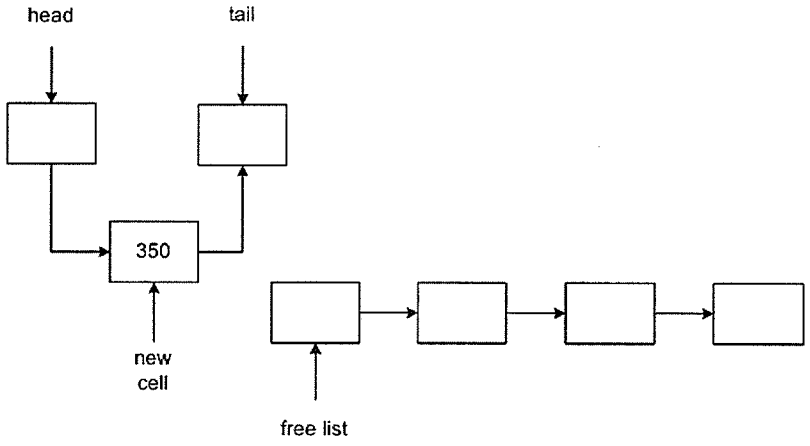


Figure 6.14: Insert the new cell between the `head` and the `tail` of the `key-list`.

`finally-linked-bit` (see Section 5.1.3) of that leaf. It has to busy wait until the `split-bit` is off, or the `finally-linked-bit` is on. If both conditions are satisfied, but there are no cells available in the `free-list`, the insert process has to communi-

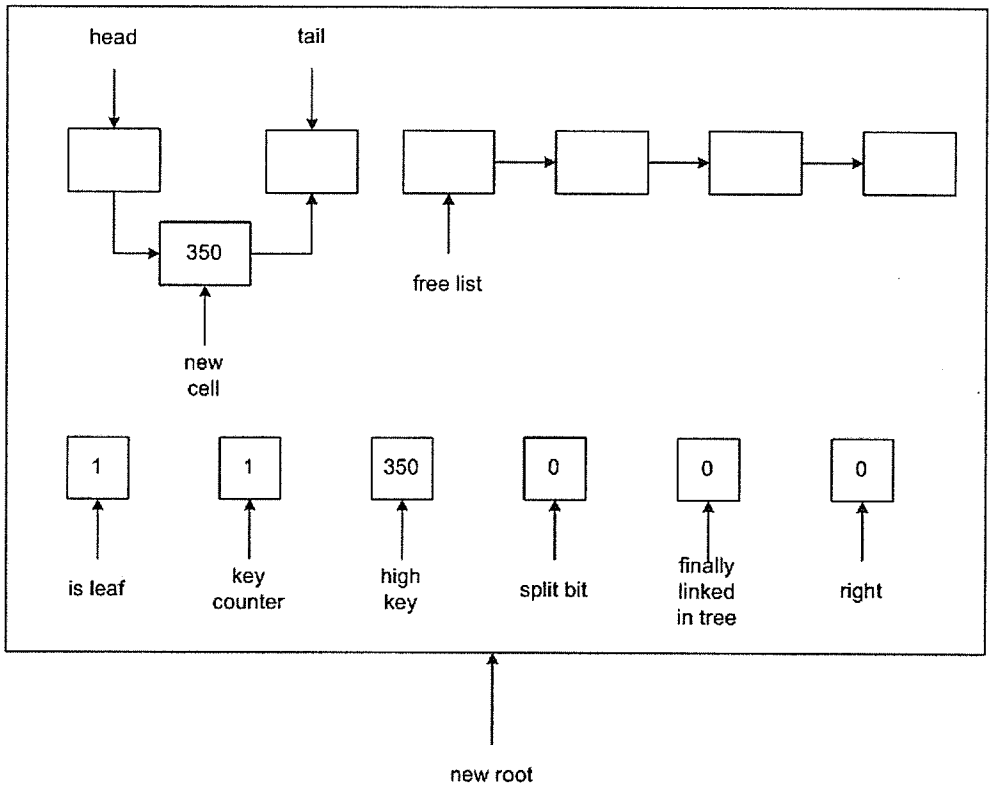


Figure 6.15: Update the `key-counter` and the `high-key` of the node.

cate with all concurrent processes, including itself, to try to get a free cell reclaimed in the `free-list`. Section 5.1.7 describes how a process communicates with other processes using the `please-scan` array.

The insert process keeps on requesting a free cell from all concurrent processes until it gets one. Once it finds a free cell in the `free-list`, it stores the new key in that cell, and goes to step 4 to search for the correct position to insert the cell containing the new key in the leaf's `key-list`.

Step 4: The Search for the New Key's Position

The search for the new key's position in a particular leaf is similar to searching for the desired key in a leaf in the search algorithm. Like the search process, the insert process also begins by setting one of its hazard pointers (`hp1`) to point to the `head` of the `key-list`, and setting its other hazard pointer (`hp2`) to point to the cell immediately after `head`. Then, the insert process starts comparing the new key to the keys of the leaf just like in the search algorithm. If the insert process finds the key in the `key-list`, it knows that the key already exists in the B^{list}-tree, and cannot be inserted in the tree again. So, the insert process returns the allocated cell to the `free-list`, empties the `path` stack that was filled during the search for the appropriate leaf, and then terminates. Otherwise, the insert process goes to step 5 to insert the allocated cell containing the new key between the cells pointed to by `hp1` and `hp2`.

Step 5: Try to Insert the Key in the Leaf

From this point on, I will refer to the cell pointed to by `hp1` as the *previous* cell, the cell pointed to by `hp2` as the *current* cell, and the cell containing the new key as the *new cell*. To insert the new cell in the key list of the leaf, the insert process sets the `next` pointer (see Section 5.1.1) of the new cell to point to the current cell (Figure 6.16). Then, the insert process uses CAS to update the `next` field of the previous cell to point to the new cell. Here, CAS takes the address of the `next` field of the previous cell as the shared variable, the address of the current cell as the old value, and the address of the new cell as the new value. If the current cell is still the

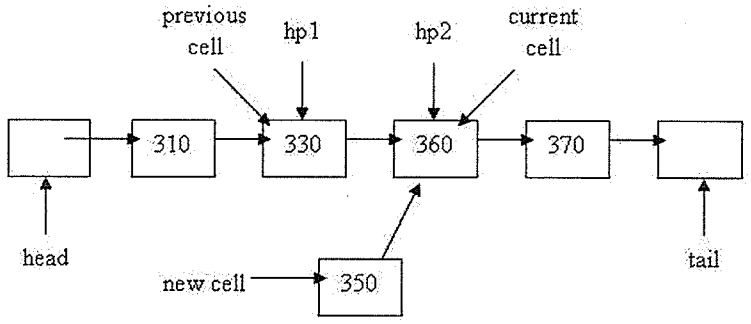


Figure 6.16: The next field of the new cell is set to point to the current cell.

cell immediately after the previous cell, the CAS operation succeeds and sets the `next` pointer of the previous cell to point to the new cell, thereby linking the new cell into the list. In Figure 6.17, before the CAS operation, the `next` pointer of the previous cell was pointing to the current cell (i.e., the old value sent to the CAS operation matched with the value of the shared variable). Therefore, the CAS swings the `next` field of the previous cell from the current cell to the new cell. If the CAS fails, then the insert process has to try again. In Figure 6.18, the CAS fails since the `next` field of the previous cell no longer points to the current cell (due to a concurrent insert by another process). In this case, the insert process repeats steps 4 and 5 until it succeeds in inserting the new cell.

After the insert process inserts the new cell in the `key-list` of the leaf, it increments the `key-counter` of the leaf. If the `key-counter` of the leaf is incremented to m (the maximum number of children that a node of a B^{list}-tree can have), then the insert process knows that the leaf is overflowing, and must split the leaf. Therefore, the process executes step 6. Otherwise, if there is no overflow, the process skips

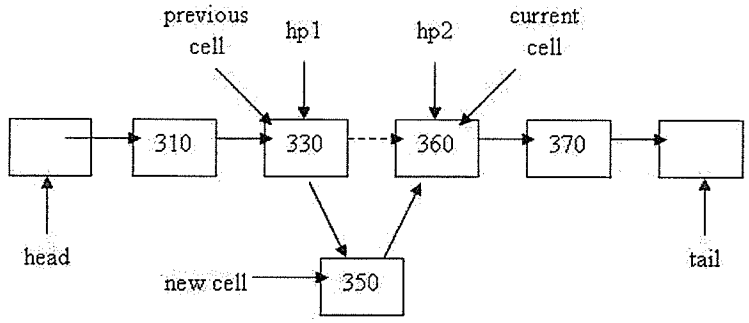


Figure 6.17: CAS swings the next field of the previous cell to point the new cell.

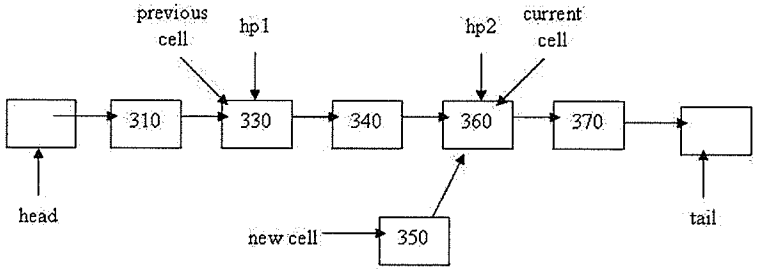


Figure 6.18: Insertion of the new cell fails.

steps 6–9, and goes to step 10 to fix the high-key of the parent node, if the leaf's high-key became greater than the parent's high-key because of the insertion. Otherwise, the insert process empties the path stack that was created during the search for the appropriate leaf, and terminates.

Step 6: Split the Leaf

To split a leaf (or node), the first thing the insert process has to do is to turn on the split-bit of that leaf. Before it tries to turn the split-bit on, it has

to check whether the leaf is accessible to all concurrent processes by checking the `finally-linked-in-tree` bit of the leaf (see Section 5.1.3). Even though an insert or delete process is allowed to insert or delete a key in a leaf (or a node) that is not linked into the tree yet, it is not allowed to split that leaf until the leaf is linked into the tree. If the leaf is not yet linked into the tree, the process needs to busy wait until the leaf is completely linked into the tree and accessible to any concurrent process.

Once the leaf's `finally-linked-in-tree` bit is set, the insert process again checks the `key-counter` of the leaf. This check is necessary because, when the insert process was busy waiting for the leaf to be linked into the tree, the `key-counter` might have gone below m due to some concurrent key deletion(s) or node split(s). In that case, there is no longer an overflow in the leaf, and, hence, there is no need to split the leaf. In this case, the insert process compares the leaf's `high-key` with its (the leaf's) parent's `high-key` (see step 7 of the insert algorithm for description of how an insert process finds the parent of a leaf or a node). If the leaf's (or node's) `high-key` is greater than the parent's `high-key`, then the insert process goes to step 10 to fix the `high-key` of this leaf's parent.

If a split is required, the insert process uses CAS to safely set the `split-bit` of the leaf. The CAS uses the `split-bit` field as the shared variable, 0 as the old value of the `split-bit`, and 1 as the new value of the `split-bit`. If the CAS finds the `split-bit` is already on, then another concurrent process has set the `split-bit` and is currently splitting the leaf. In that case, the CAS fails, and the insert process busy waits until the `split-bit` is off as described. When the busy wait is over, the insert process again checks the `key-counter` of the leaf, and tries to set the `split-bit`

again, if the leaf is still overflowing.

The insert process creates a new leaf for the B^{list}-tree after it sets the `split-bit`. After creating the new leaf, the insert process finds the *middle cell* containing the *middle key* (the $\lceil m/2 \rceil^{th}$ key) from the leaf's `key-list` (see Figure 6.19). Then, it copies the cells that contain keys larger than the middle key from the original leaf's `key-list` to the new leaf's `key-list`. Next, the insert process updates the `key-counter` of the new leaf with the number of cells copied to the new leaf, then updates the `high-key` of the new leaf with the `high-key` of the original leaf, and, finally, sets the new leaf's `right` pointer to point to the leaf currently pointed to by the original leaf's `right` pointer. In Figure 6.20, the middle key of the original leaf is 300. Since the cells containing 400 and 500 have greater key values, the insert process copies them from the original leaf to the new leaf. Then, it sets the `key-counter` to 2 (since the number of cells copied into the new leaf is 2), and copies the `high-key` of the original leaf (500) to the `high-key` of the new leaf. In the end, it links the new leaf's `right` pointer to point to the leaf that is currently pointed to by the original leaf's `right` pointer.

After updating the fields of the new leaf, the insert process changes the necessary fields of the original leaf. First, it sets the `right` pointer of the original leaf to point to the new leaf. Then, it updates the `high-key` of the original leaf to be its middle key. Next, it decreases the `key-counter` of the original node to $\lceil m/2 \rceil$. In Figure 6.21, the insert process sets the original leaf's `right` pointer to point to the new leaf. Besides this, it changes the `high-key` to the value of the middle key, 300, and sets the `key-counter` to 3. Finally, the insert process reclaims the cells that were

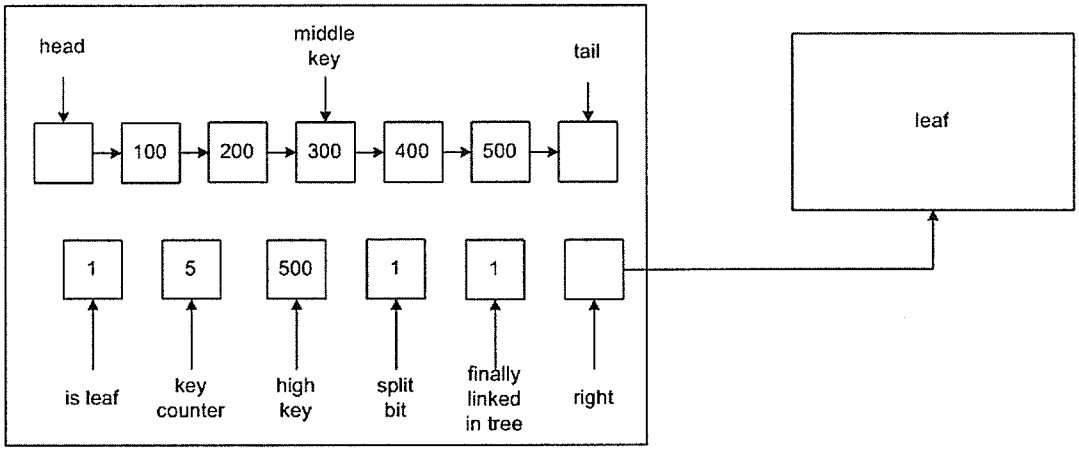


Figure 6.19: A split process finds the middle key containing 300.

copied to the new leaf adding them to the original leaf's `free-list` for reuse (see Figure 6.22).

Step 7: Find the Parent

When the insert process completes the split operation, it needs to find the parent of the original leaf to insert a copy of the original leaf's middle key into its parent. Furthermore, a child pointer to the new leaf from the parent is also required. To find the parent of the original leaf, the insert process gets the topmost node from the `path` stack (see Section 5.1.5) that was created during the search for the appropriate leaf in step 2.

When the insert process has a non-empty `path` stack, it gets the topmost node of the `path` stack which should be the parent of the original leaf. However, due to the possibility of concurrent splits on the parent, the parent node popped off the `path` stack might no longer be the parent of the original leaf. In that case, the insert

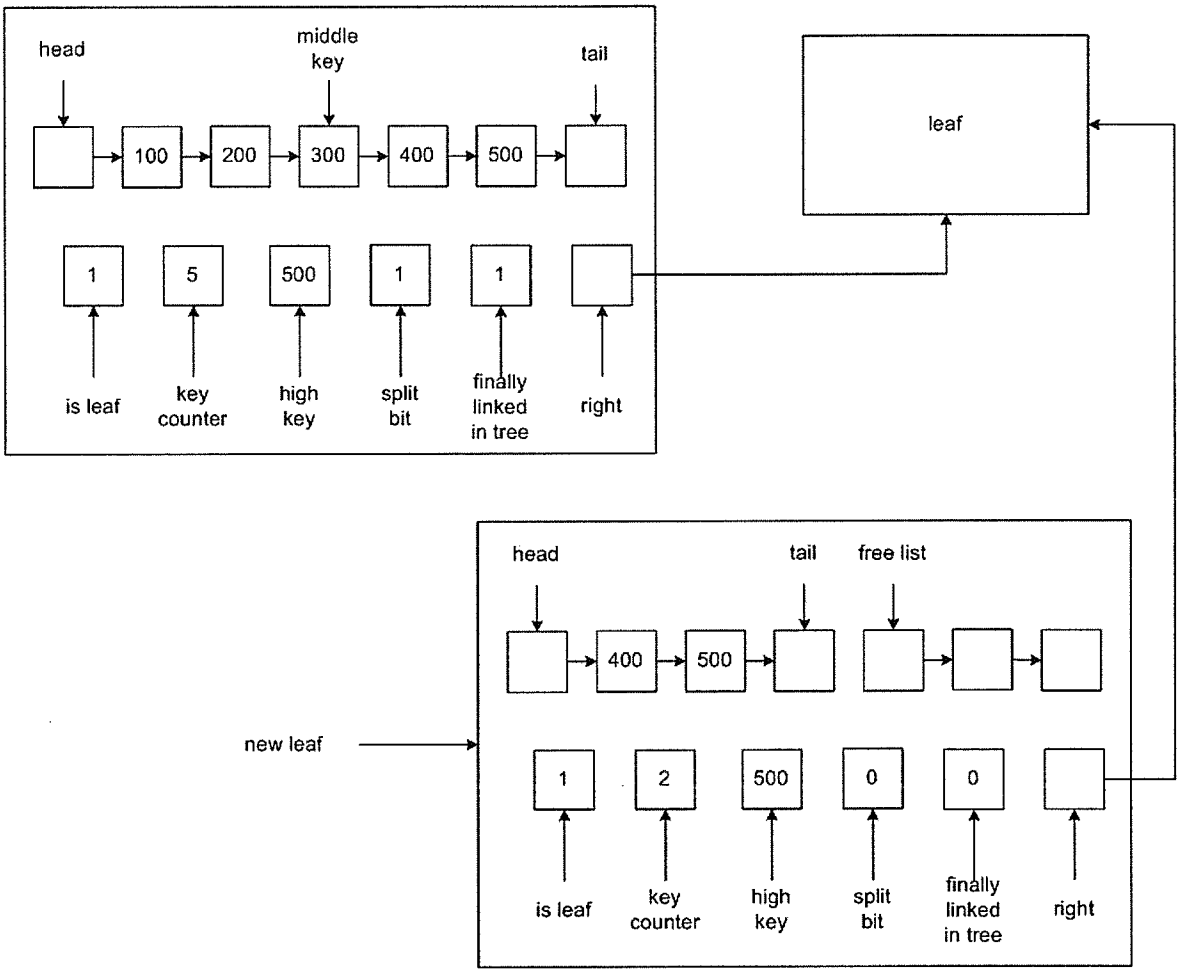


Figure 6.20: A split process creates a new leaf, and updates the fields of the new leaf.

process finds a node to the right of the node stored on the `path` stack as the parent, a node that has a `high-key` greater than or equal to the `high-key` in the original leaf (see Section 6.2 for details). After the split process finds the correct parent for the original leaf, it goes to step 9 for the middle key insertion into the parent.

If the insert process split the root of the B^{list}-tree in step 6, the `path` stack is supposed to be empty (since, the root has no parent), and there is no way to return

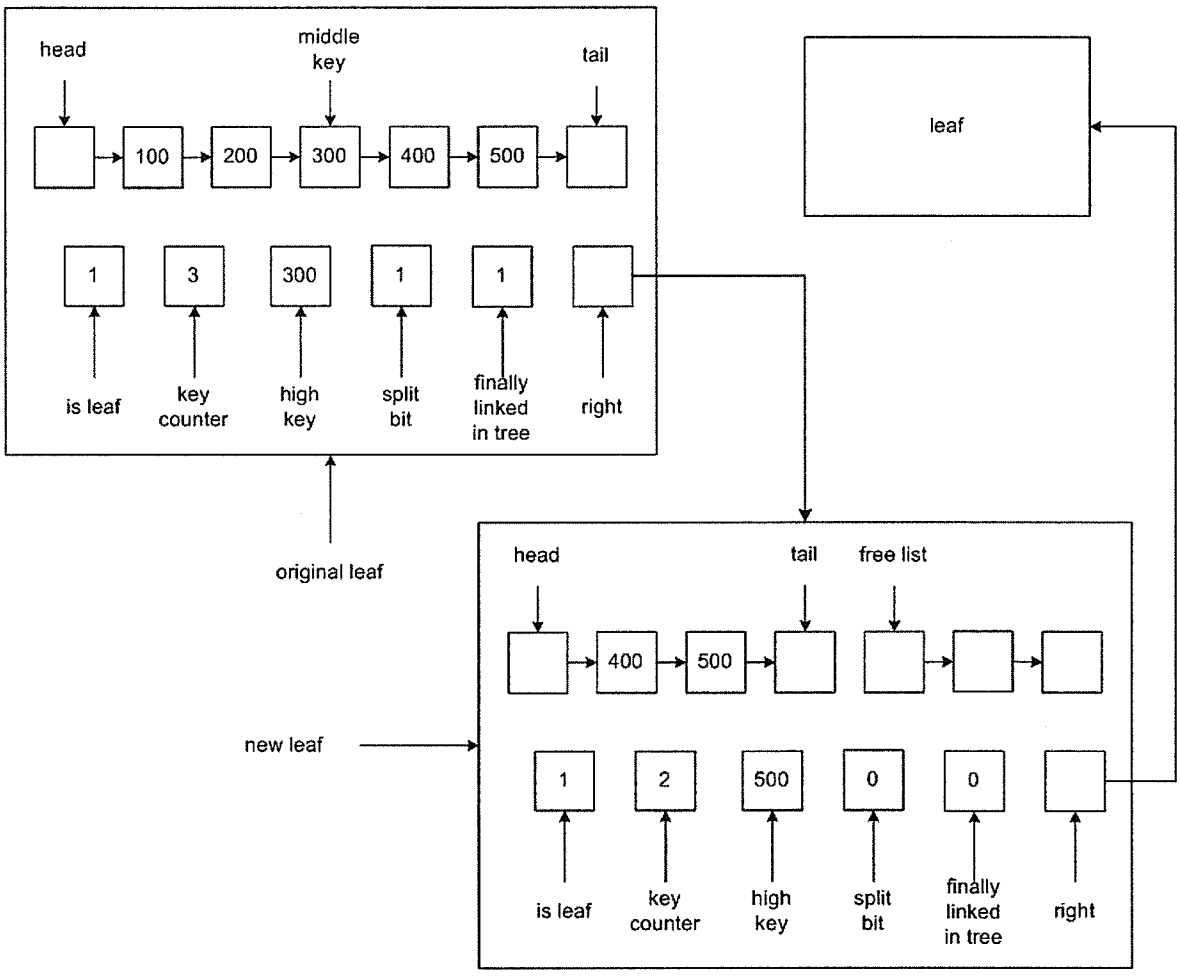


Figure 6.21: A split process changes the `right` pointer, the `high-key`, and the `key-counter` of the original leaf.

any node from the empty `path` stack. In that case, the insert process goes to step 8 to add an extra level to the tree.

Concurrent operations can also cause an empty `path` stack, even if the insert process did not split the root of the B^{list}-tree in step 6. This can happen if, during the operations in step 2, the original leaf was the root of the tree, but due to other

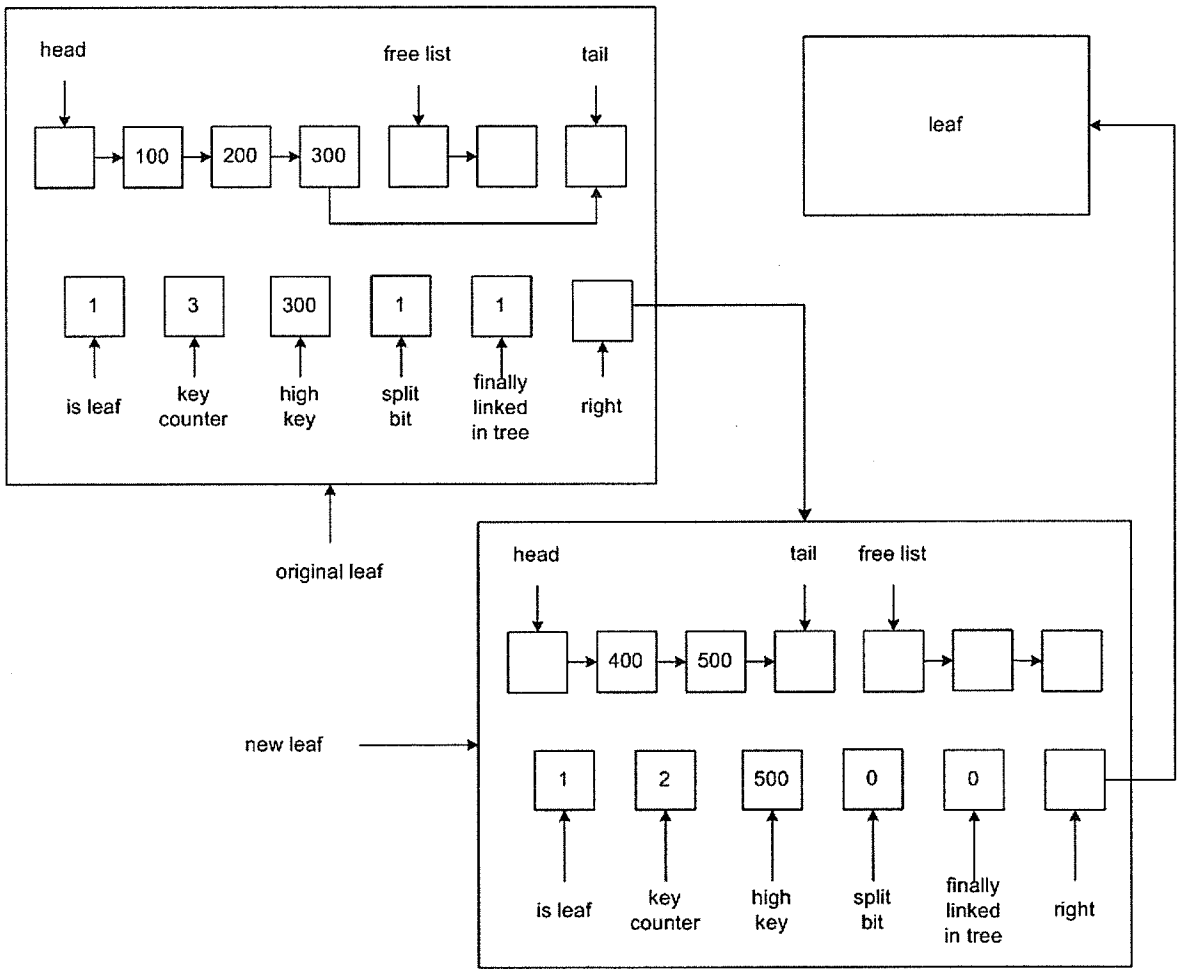


Figure 6.22: A split process returns the reclaimed cells to the free list of the original leaf.

concurrent operations, the tree has had a new root added (and has added levels between the current root and the original leaf). The only way to get the parent of the original leaf is to now re-fill the `path` stack with the visited nodes in the path from the root to the original leaf. In this case, the insert process re-fills the `path` stack in the same way that it filled the `path` stack in step 2. After completing the re-filling

of the `path` stack, the insert process gets the address of the parent node from the topmost node stored in the `path` stack, and goes to step 9 to insert the middle key in the parent node.

Step 8: Make the Tree Taller

In step 6 of the insert algorithm, if an insert process splits the root of the tree, it needs to add an extra level to the tree by creating a new root for the tree (see Figure 6.23). To do this, the insert process allocates a new cell from the new root's `free-list` in the same way it allocates a new cell in step 3. In addition to the operations in step 3, the `left-child` pointer of that new cell is set to point to the original node (or leaf), and the `right-child` pointer of the same cell is set to point to the new node (or leaf) that was created by the split operation in step 6. In Figure 6.24, the insert process allocates a new cell from the `free-list` of the new root's `key-list`, and copies the middle key, 300, into the `key` field of that cell. Then, it copies the address of the original node into the `left-child` field of that cell, and copies the address of the new node into the `right-child` field of the cell. After linking the original and the new node with the new cell, the insert process inserts the new cell between the `head` and the `tail` of the new root's `key-list`. Then, it sets the `key-counter` to 1, and updates the `high-key` with the new node's `high-key`. After that, the insert process sets the `finally-linked-in-tree` bit (see Section 5.1.3) of the new node, and clears the `split-bit` of the original node. Finally, it sets the root pointer of the B^{list}-tree to point to the new root, followed by setting the `finally-linked-in-tree` bit of the new root (see Figure 6.25).

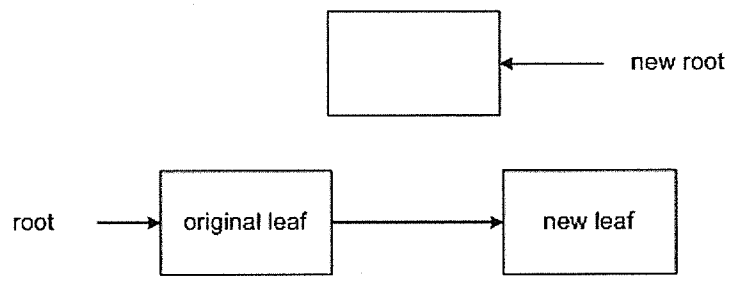


Figure 6.23: The insert process creates a new root for the B^{list}-tree.

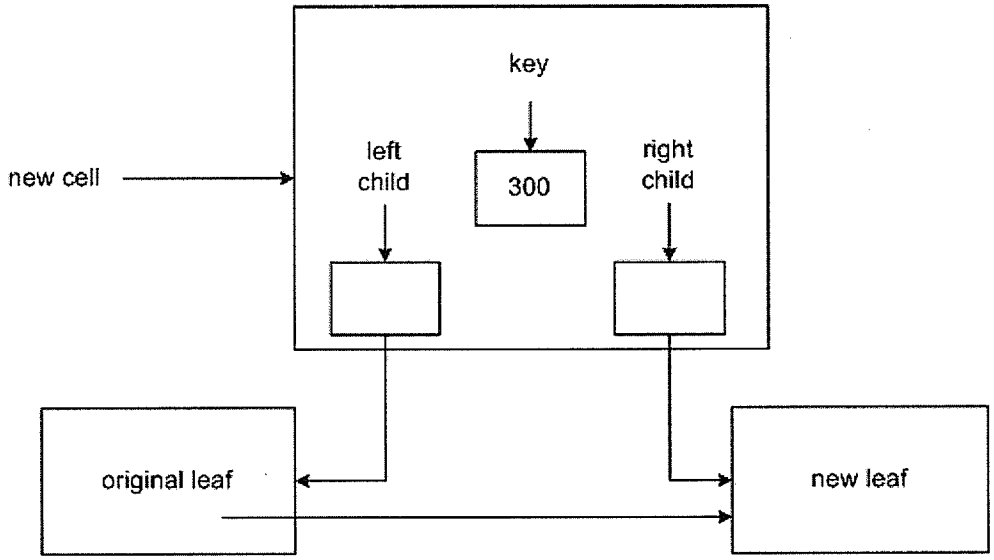


Figure 6.24: Linking the new root into the tree.

Step 9: Insert the Middle Key into the Parent

Once the insert process finds the parent of the original node (or leaf), it allocates a new cell from the `free-list` of the parent, similar to the way it allocates a new cell from a leaf's `free-list` in step 3. The only difference here is that after allocating the new cell from the parent's `free-list`, besides copying the middle key to that cell, it

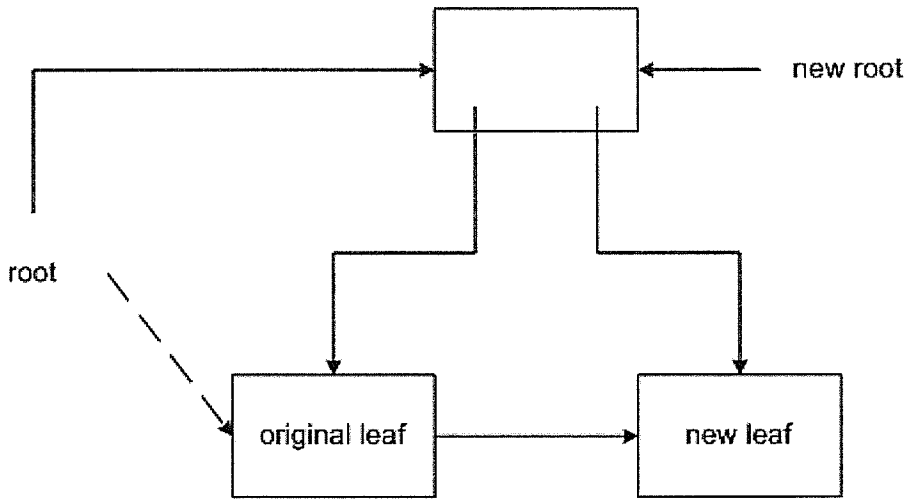


Figure 6.25: The insert process sets the root pointer to point to the new root.

copies the address of the original leaf to the `left-child` pointer of the new cell, and the address of the new leaf to the `right-child` pointer of the new cell (Figure 6.26).

The next step is to search for the middle key in the parent. This search is identical to step 4 of the insert algorithm. If the insert process does not find the middle key in the parent, it tries to insert the new cell between the cells pointed to by its hazard pointers, `hp1` (the previous cell) and `hp2` (the current cell). Figure 6.27 shows the locations of `hp1` and `hp2` in the parent node, before the insertion of the new cell. Insertion of the new cell between the previous and the current cell is identical to step 5.

During a middle key insertion, the insert process needs to fix the child pointers of the previous and current cells after the insertion of the new cell in the `key-list`. If the middle key is not the largest key in the parent node, after the insertion, the `left-child` pointer of the current cell points to the original leaf, as shown in Fig-

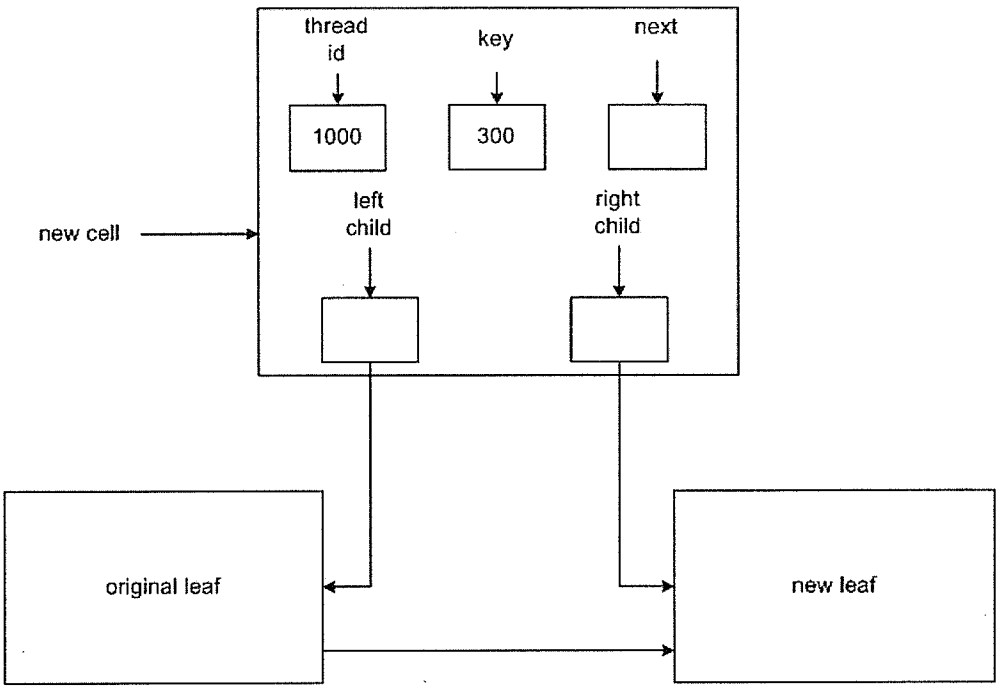


Figure 6.26: The insert process copies the middle key, the original leaf, and the new leaf to the `key`, `left-child`, and the `right-child` fields of the new cell, respectively.

ure 6.28. The insert process changes the current cell's `left-child` pointer to point to the new leaf (see Figure 6.29). If the middle key becomes the largest key in the parent, and the new cell becomes the rightmost cell in the `key-list` (see Figure 6.30), then the insert process sets the `high-key` of the parent to be the `high-key` of the new leaf.

If this insertion creates an overflow in the parent, then the parent needs to be split. In that case, the insert process repeats step 6 to 9. A node split is very similar to a leaf split as described in step 6. The only difference is that when the insert process reclaims the copied cells to the original node's `free-list`, along with the copied cells it also reclaims the middle cell to the `free-list`, since B^{list}-trees store the keys only

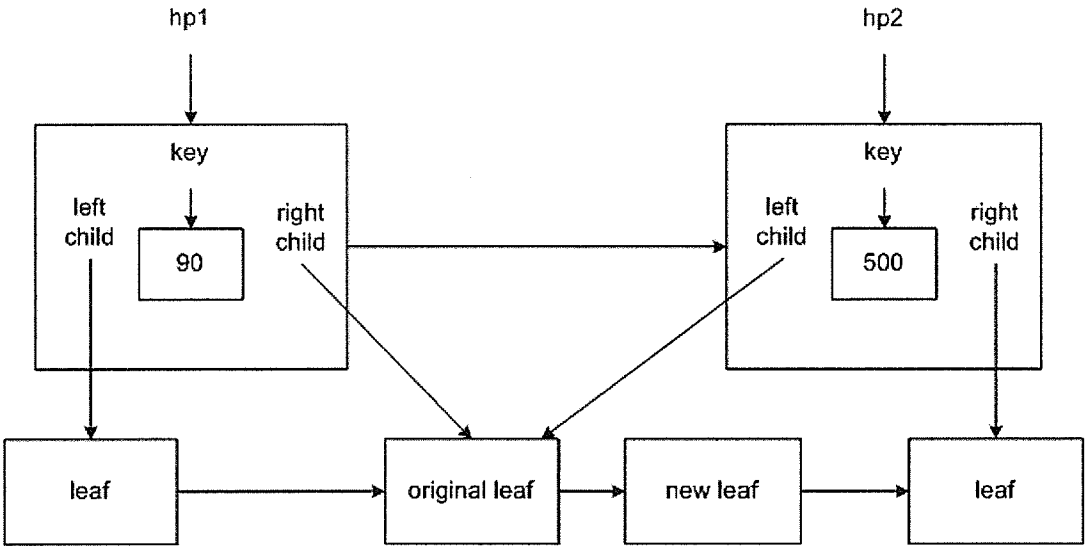


Figure 6.27: The locations of `hp1` and `hp2` of the inserter in the `key-list` of the parent node, before the insertion.

in their leaves. Furthermore, the `key-counter` is decremented to $\lceil m/2 \rceil - 1$, instead of decrementing it to $\lceil m/2 \rceil$.

Step 10: Fix the Parent's High-key

After the key is inserted into a leaf or a non-leaf node, if the `high-key` of that node becomes greater than its parent's `high-key`, then the insert process updates the parent's `high-key` with that node's `high-key`. If, the parent's `high-key` becomes greater than the `high-key` of the parent's parent, then the `high-key` of the parent's parent is updated with the `high-key` of the parent. These `high-key` updates continued until the insertion process finds a node whose `high-key` is less than or equal to its parent's `high-key`, or the node has no parent (i.e., the node is the root of the tree). Once the parent's `high-key` is fixed, the insert process empties the `path` stack

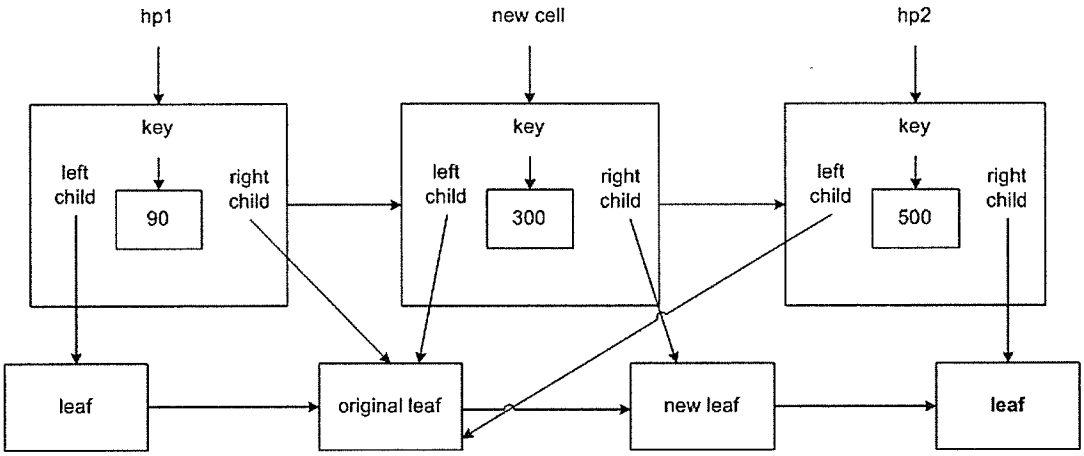


Figure 6.28: The insert process inserts the new cell in the `key-list`.

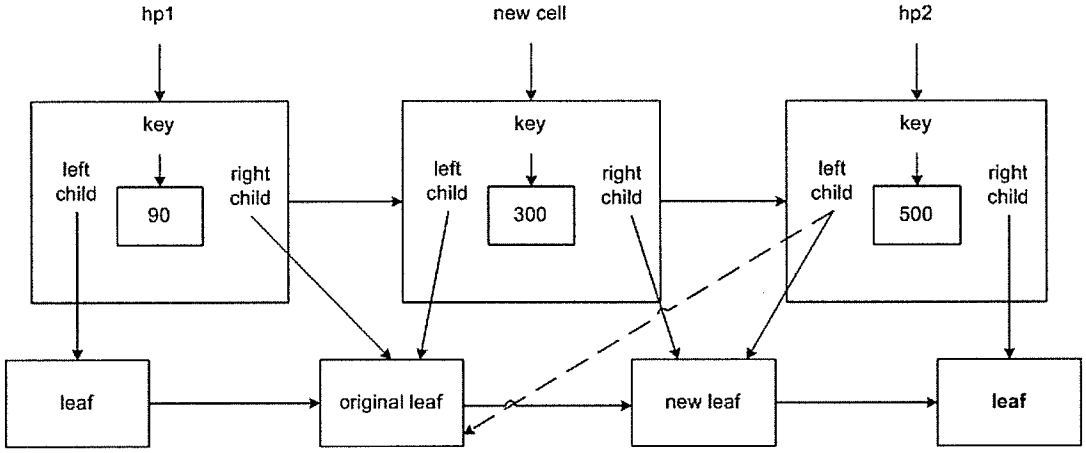


Figure 6.29: The insert process fixes the `left-child` pointer of the cell pointed to by `hp2`.

and terminates.

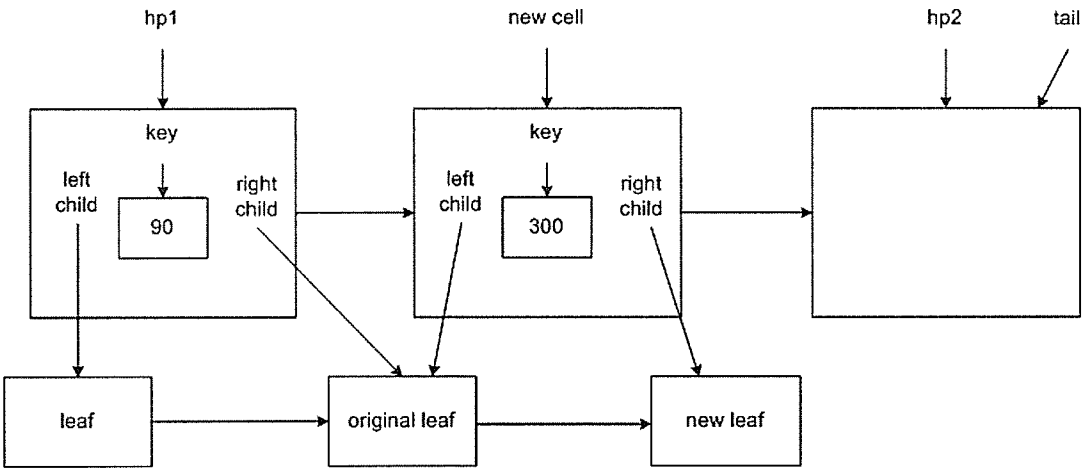


Figure 6.30: The insert process does not need to fix the `left-child` pointer of the cell pointed to by hp2, if the new cell is the rightmost cell in the `key-list`.

6.1.3 Delete Algorithm

The delete algorithm uses multiple steps to delete a key from a B^{list}-tree. In the first step of the delete algorithm, a delete process finds the appropriate leaf in which to look for the key to be deleted. In the second step, it searches for the desired key in that leaf. In the third step, the delete process deletes the key from that leaf's `key-list`, if the desired key exists in the tree. In the fourth step, the deleted cell is retired by the process. If the number of cells retired by this process reaches the given threshold level, in the last step, the delete process performs a scan to return its retired cells to the appropriate `free-lists`.

Step 1: Find the Appropriate Leaf

Finding the appropriate leaf in a deletion is identical to finding the appropriate leaf in a search. See step 1 in the search algorithm (Section 6.1.1) for details.

Step 2: Search for the Desired Key

The search for the desired key in an insertion and the search for the desired key in a delete algorithm are identical. See step 4 of the insert algorithm (Section 6.1.2) for details.

Step 3: Delete the Requested Key from the Leaf

If the delete process does not find the desired key in the appropriate leaf, the process knows that the key does not exist in the B^{list}-tree, so the desired key cannot be deleted, and the deletion process terminates. Before the termination of the deletion process, the `path` stack is emptied.

If the delete process finds the desired key in the appropriate leaf of the B^{list}-tree, it removes the cell that contains the desired key from the `key-list` of that leaf and stores the address of the cell and the leaf in the `retired-cell` stack (see Section 5.1.6). When the delete process finds the desired key in step 3, the hazard pointer `hp2` of the delete process points to the cell that contains the desired key. Let us call this cell the `current-cell`. The other hazard pointer, `hp1`, points to the cell immediately to the left of the current cell. Let us call this cell the `previous-cell`. In Figure 6.31, `hp1` points to the `previous-cell`, containing 320, and `hp2` points to the `current-cell`, containing the desired key, 330.

Before deleting the `current-cell`, the delete process *marks* (see Section 3.2.2 for a discussion of why and how the delete process marks a cell to be deleted) the `next` field of the `current-cell` to be deleted (see Figure 6.32). Marking a cell protects that cell from deletion attempts by other concurrent process(es). Moreover, when another concurrent search or insert process looks for the correct position to search for or insert a key within a node (step 2 of the search algorithm, and step 4 of the insert algorithm), if the search or insert process encounters a cell that is already marked, it immediately goes back to the start of the `key-list` to avoid comparing the desired key with a `key` field of a cell that is already or about to be deleted. Before marking a cell, the delete process has to deal with some concurrency control issues that are described later in Section 6.2.

After marking the cell to be deleted, the delete process uses CAS to update the `next` pointer of the `previous-cell` to point to the cell, `new-current`, immediately following the `current-cell`. The CAS takes the `next` field of the `previous-cell` as the shared variable, the address of the `current-cell` as the old value, and the address of `new-current` as the new value to be stored in the `next` field of the `previous-cell`. If the CAS succeeds, the `next` field of the `previous-cell` points to the `new-current` cell (see Figure 6.33). Then, the delete process decrements the key counter of the leaf, and stores the process identification number of the deletion process in the `current-cell`'s `thread-id` field. This identification number is used later to reclaim the `current-cell` to the `free-list` of the leaf. In case of an unsuccessful CAS (due to some concurrent updates), the delete process unmarks the `current-cell`, and repeats steps 2 and 3 until the CAS operation succeeds.

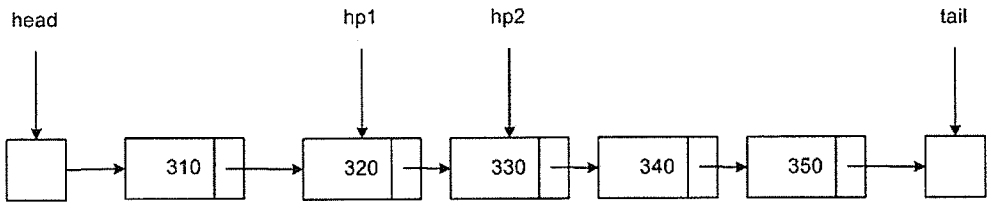


Figure 6.31: The delete process finds the desired key 330 in the key list.

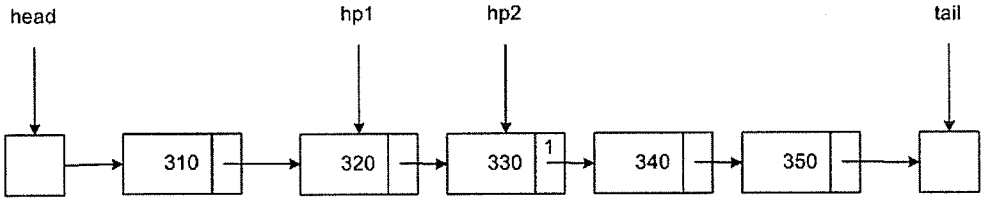


Figure 6.32: The delete process marks as deleted the `current-cell` pointed to by hp2.

Step 4: Retire the Deleted Cell

Since a B^{list}-tree does not use locks for any operation, returning a deleted cell to the `free-list` of a leaf after a deletion is unsafe in a concurrent environment. For

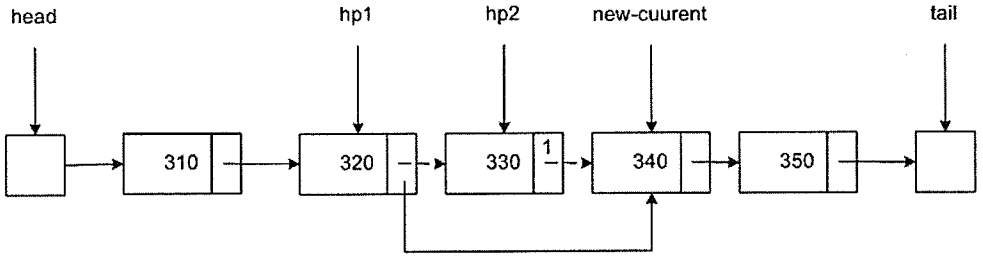


Figure 6.33: The CAS operation swings the `next` field of the `previous-cell` (pointed to by `hp1`) from the `current-cell` (pointed to by `hp2`) to the `new-current` cell.

example, if a deleter returns a deleted cell to the `free-list` while other concurrent processes are concurrently working on that cell, it may create incorrect results (see Section 3.2.2). Therefore, instead of returning a cell directly to the `free-list` of the leaf, my deletion algorithm marks the deleted cell as retired, and stores the address of the cell, along with the address of the associated leaf, in the `retired-cell` stack (see Section 5.1.6). Each process in a B^{list}-tree has its own `retired-cell` stack from which only that process can later reclaim the cells to the `free-list` of the cell's corresponding leaf.

Every B^{list}-tree is assigned a threshold level R (as described by Michael [27]) that is usually related to m. If the size of a process's `retired-cell` stack reaches the given threshold, R, the delete process calls for a scan and goes to the next step. Otherwise, the delete process empties the `path` stack and terminates.

Step 5: Scan the Retired-cell Stack

The scan routine returns retired cells to their free lists, if they are not currently pointed to by the hazard pointers of any process.

First, the scan routine clears the `scan-bit` of the deletion process in the `please-scan` array (see Section 5.1.7). In the next step, it copies the global list of cell addresses, along with their corresponding leaf addresses, of all cells that are currently pointed to by the hazard pointers of any process. These copies are stored in a local list, `p-list`. Then, the scan routine sorts the `p-list` according the cell addresses. Next, the scan routine empties the `retired-cell` stack and stores all the cells, along with their associated leaves, in a temporary stack. The scan routine then compares the

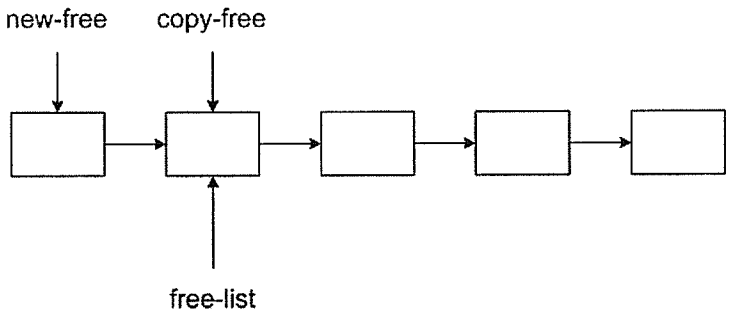


Figure 6.34: The delete process links the cell to be reclaimed (`new-free`) to the `free-list`.

cells from the `p-list` with the cells stored in the temporary stack. Any cells that appear in the `p-list` are being used by some process, so they are returned to the `retired-cell` stack. Any cells that do not appear in the `p-list` are not being used by other processes, and can be safely returned to their free lists as described in step 6.

Step 6: Reclaim the Retired Cells

In this step, the delete process uses CAS to return each safe cell to its corresponding leaf's `free-list`. The CAS takes the address of the `free-head` (see Section 5.1.2) of the `key-list` as the shared variable, a copy of the `free-head` (`copy-free`) as the old value, and the cell to be reclaimed (`new-free`) as the new value. If other processes have not changed the copy of `free-head` (i.e., if `free-head` and `copy-free` are the same), the CAS links `new-free` with the `free-list`, and makes `new-free` the new `free-head` of the `free-list`. If the CAS fails, the delete process repeats this step until the CAS succeeds. Figures 6.34 and 6.35 show the reclamation of a cell to a `free-list`.

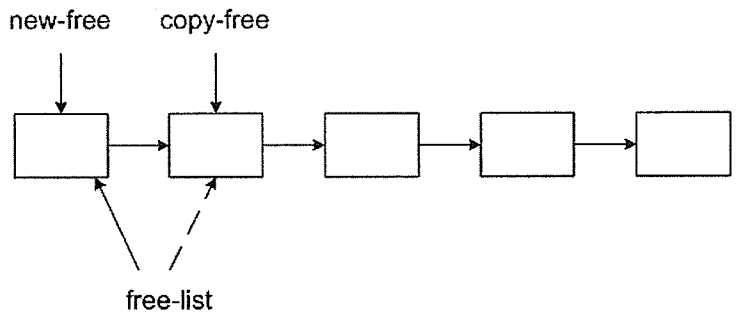


Figure 6.35: The CAS operation swings the `free-head` from `copy-free` to `new-free`.

6.2 Concurrency Correctness of the Algorithms

Unlike the B^{link}-tree algorithms, the B^{list}-tree algorithms do not lock a node for an update or a split operation. Therefore, when a process performs an operation in a node, other concurrent processes can search, insert, or delete some keys in the same node, or can even split the entire node into two nodes. To avoid any inconsistency, my B^{list}-tree algorithms need to handle these concurrent operations without locking the node. In the following subsections, I describe where and how I dealt with the concurrency issues in my B^{list}-tree algorithms.

Consider the three major types of operations in my B^{list}-tree. They are the search operations, where the processes just read, but do not write, any data in the tree; the update operations (insert and delete), where the processes write data in the tree, and finally, the split operations, where the processes split a node into two different nodes. I ensure that none of these types of operations interfere with each other while operating concurrently in a B^{list}-tree.

6.2.1 Concurrency Control in Search Operations

Selecting the correct child to follow to get to the appropriate leaf, and searching for the desired key in the leaf are the key tasks in any search operation in a B^{list}-tree. Since the search tasks do not change any contents of the B^{list}-tree, concurrent search operations in a node (both leaf and non-leaf) do not create conflicts with each other. Conflicts can, however, arise when one or more concurrent update and/or split operations are performed simultaneously with a search operation.

Conflicts with Concurrent Update Operations

When a searcher compares a desired key with the keys of a node's key list, concurrency control is required if a concurrent inserter is inserting a new cell between the cells pointed to by the searcher's `hp1` and `hp2`. To be more precise, in the example in Figure 6.36, the search operation is searching for the key 25. The searcher's hazard pointers are pointing to the cells containing 20 and 30. Concurrently, as shown in Figure 6.37, another inserter is inserting a new cell, containing 25, between the cells containing 20 and 30. Without concurrency control, a search operation would see that its hazard pointer `hp2` points to a cell that contains 30, which is greater than the desired key 25. Even though the key 25 is in the `key-list`, because of the concurrent insertion (as in Figure 6.38), the searcher would conclude that the key 25 is not in the `key-list` of the node.

To avoid this inconsistency, before comparing the desired key with the key in the cell pointed to by `hp2`, and before moving the `hp1` and `hp2` pointers one cell ahead in the `key-list`, the search operation always tests whether the cell pointed to by

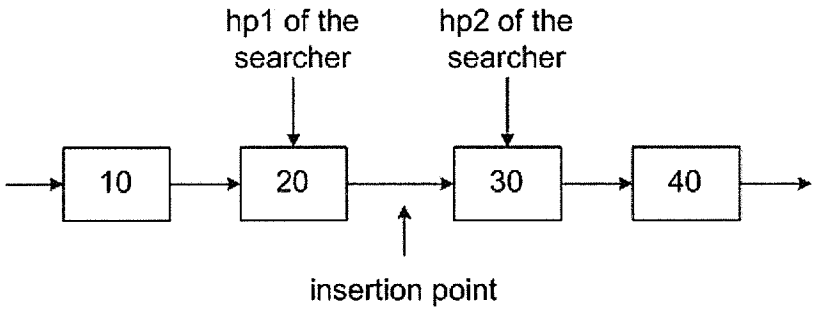


Figure 6.36: During a search operation, a concurrent process wants to insert a new cell between the searcher's `hp1` and `hp2`.

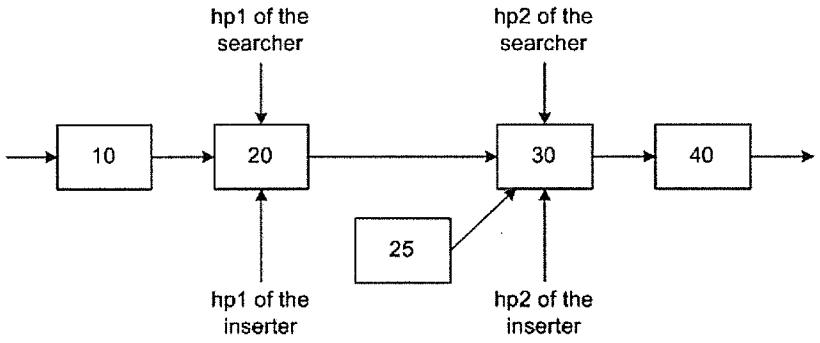


Figure 6.37: During a search operation, a concurrent inserter is trying to insert a new cell between the searcher's `hp1` and `hp2`.

`hp2` is still the next cell immediately to the right of the cell pointed to by `hp1` in the `key-list`. If the test fails (as in Figure 6.38), the search operation must re-assign `hp2` to point to the cell immediately following the cell pointed to by `hp1`, and continue the search for the desired key (Figure 6.39).

Similarly, if a concurrent delete operation deletes any of the cells pointed to by the searcher's hazard pointers `hp1` and `hp2`, the searcher needs to handle the potential inconsistency caused by the concurrent deletion. In the example shown in Figure 6.40,

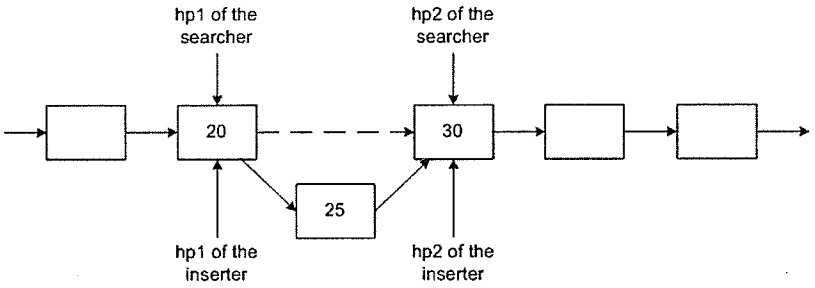


Figure 6.38: During a search operation, a concurrent inserter has inserted a new cell between the searcher's `hp1` and `hp2`.

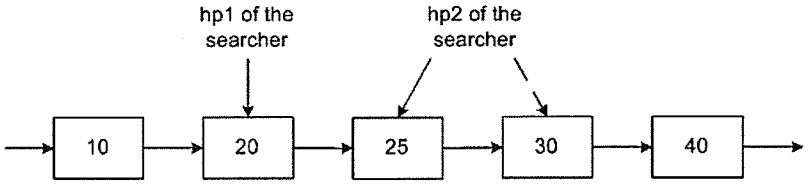


Figure 6.39: During a search operation, the searcher re-assigns `hp2` to the cell immediately following to the cell pointed to by `hp1`.

a searcher's hazard pointers are pointing to the cells containing 20 and 30, and a concurrent delete operation is deleting the cell pointed to by the searcher's `hp1` (the cell containing 20). To delete a cell from a node, a deleter marks the cell containing 20 as deleted, by setting the low order bit of the address stored in the cell's `next` field. Now, if a searcher needs to move its hazard pointers one cell to the right, to their next cells in the `key-list`, because of the marking by the deleter, the cell is no longer linked into the `key-list` (`hp1->next` is no longer a valid cell address). Therefore, moving `hp1` to the immediately following cell, `hp1->next`, in the `key-list` will be an invalid operation for the searcher that will result in an incorrect action in the search

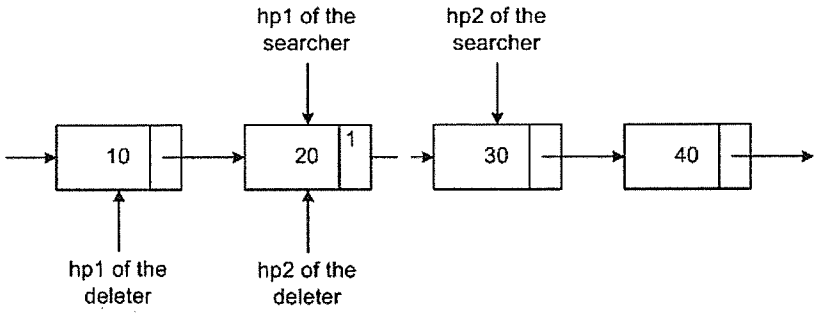


Figure 6.40: During a search operation, a concurrent delete is deleting the cell pointed to by the searcher's `hp1`.

operation.

To handle this situation, before comparing each key in the `key-list`, and before moving to the next cell, the searcher checks whether the `next` field of the cell pointed to by `hp1` is marked as deleted or not. If the cell is marked as deleted, the search operation moves back to the beginning of the `key-list`, and starts searching for the desired key again, from the beginning of the `key-list`.

Handling the deletion of the cell pointed to by a searcher's `hp2` is a little bit different. Here, two cases must be considered. In the first case, the deletion is already in progress (see Figure 6.41), and in the second case the deletion is complete (see Figure 6.42).

In the example in Figure 6.41, while a searcher is comparing the key 30 in the cell pointed to by its `hp2` with the desired key, another concurrent deleter has marked the `next` field of the same cell for deletion. Since, the cell is currently marked, it is no longer linked into the `key-list`. Therefore, before comparing the key of that cell with the desired key, the search operation tests whether the cell is marked as deleted

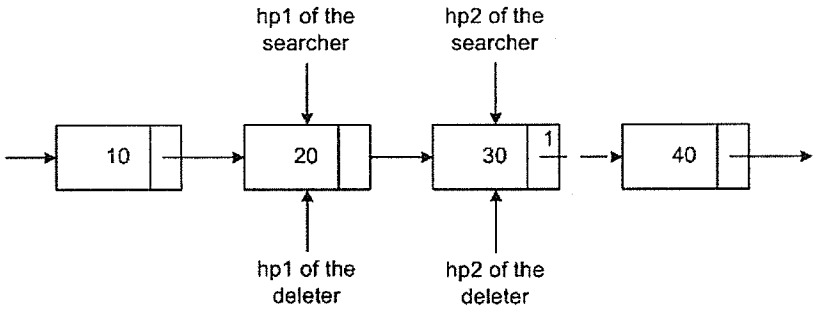


Figure 6.41: During a search operation, a concurrent delete is deleting the cell pointed to by the searcher's `hp2`.

or not. If it is marked as deleted, the search operation goes to the beginning of the `key-list`, and starts searching for the desired key again. The searcher performs the same test before moving `hp2` to the immediately following cell. As the cell is no longer in the `key-list`, there is no cell after the cell pointed to by `hp2`. Therefore, this test prevents the searcher from pointing to some cell that is no longer in the `key-list` of the node.

In the second case, shown in the example in Figure 6.42, the deletion is already complete. The searcher's `hp1` points to a cell that contains 20, and the immediately following cell now contains 40. The pointer `hp2` of the searcher still points to the cell containing 30 which is marked as deleted and is no longer a part of the `key-list` of the node. To avoid inconsistency in a situation like this, before comparing the key in the cell pointed to by `hp2` to the key to be deleted, and before moving to the cell immediately following `hp2`, the search operation must check that the cell pointed to by `hp2` is still the cell immediately after the cell pointed to by `hp1`. When the test fails, the searcher must re-assign `hp2` to point to the cell that immediately follows

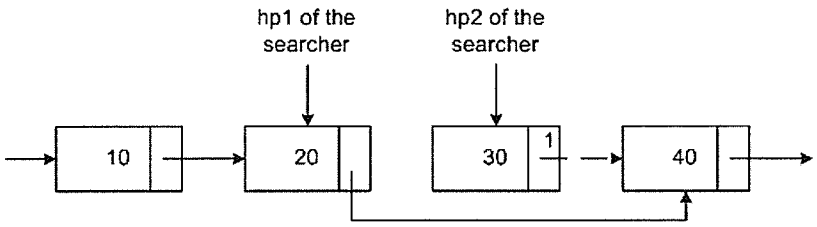


Figure 6.42: During a search operation, a concurrent delete has already deleted the cell pointed to by the searcher's hp2.

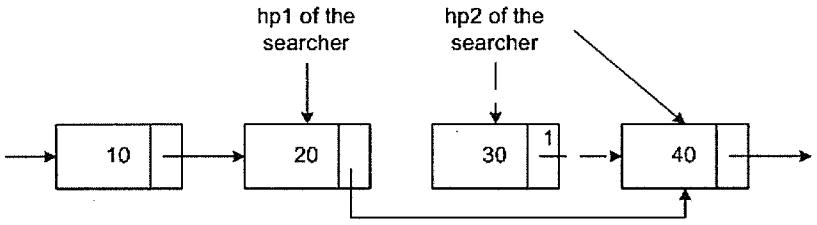


Figure 6.43: In a search operation, the searcher re-assigns hp2 to the cell immediately after the cell pointed to by hp1.

the cell currently pointed to by hp1 (see Figure 6.43). Rather than going back to the beginning of the key-list, re-assigning the pointer hp2, to the cell immediately after the cell pointed to by hp1 saves time in finding the correct position of the desired key in the node, especially when the key-list is long.

In a search operation, a search process only compares the key field of the cell pointed to by its hp2 with the desired key. If a concurrent update operation inserts or deletes a cell to the left of the cell pointed to by the searcher's hp1, or to the right of the cell pointed to by the searcher's hp2 (see Figures 6.44 and 6.45), that concurrent update operation does not affect the search operation. Therefore, concurrency control is not required is such cases. In Figure 6.44, one inserter (inserter 1) inserts 15 and

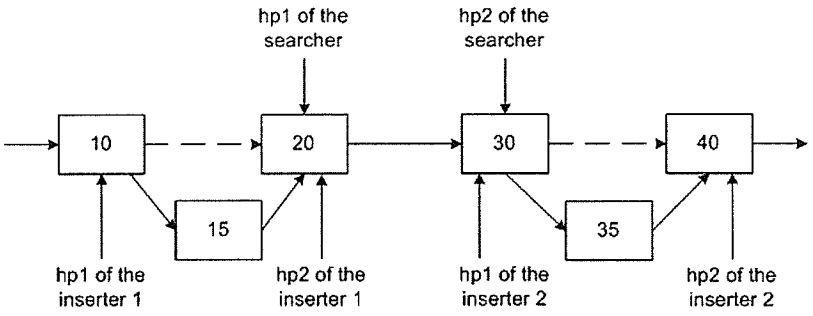


Figure 6.44: No conflicts occur if the insertions are taking place to the left of the `hp1` of the searcher or to the right of `hp2` of the searcher.

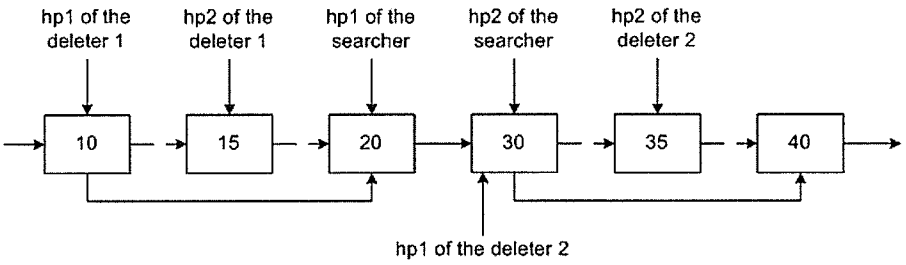


Figure 6.45: No conflicts occur if the deletions are taking place to the left of the `hp1` of the searcher or to the right of `hp2` of the searcher.

another inserter (`inserter 2`) inserts 35 in the `key-list`. These insertions do not create any conflicts with the search operation. Similarly, in Figure 6.45, `deleter 1` is deleting 15 and `deleter 2` is deleting 35 from the `key-list` without any conflict with the search operation.

Conflicts with Concurrent Split Operations

Handling conflicts between a search operation and a concurrent split operation is more difficult than handling conflicts between a search and a concurrent update

operation. This difficulty arises because a search operation can start searching for a desired key in the middle of an ongoing split operation. Moreover, a search process allows a split operation to split the node while the process searches for the desired key. While searching for a key in a node, if the node gets split by a concurrent split process, and the desired key no longer remains in the examined node, then the search operation would conclude that the desired key does not exist in the tree, which is incorrect. The situation gets worse when a cell that is currently being examined by the search process now belongs to the `free-list` of the node due to the split operation (see step 6 of the insert algorithm in Section 6.1.2).

A search operation handles the concurrency issues with split operations in different ways. A search operation compares the desired key with the `high-key` (see Section 5.1.3) of a node before it starts looking for the desired key in that node. As a result of a split operation, if the desired key now resides in the node created by the split operation, then the `high-key` of the original node will be less than the desired key. Therefore, if the search operation finds that the desired key is greater than the `high-key` of the *current* node (the node it is now looking for the desired key in), it must move right to the immediately following node to search for the desired key. In a split operation (described in Section 6.1.2), as we have seen, a splitter copies half the keys from the original node's `key-list` to the new node's `key-list`, then it links the original node to the new node, and finally, it changes the `high-key` of the original node. Therefore, once a concurrent split operation changes the `high-key` of the original node, it has already created the new node, has copied the keys from the original node to the new node, and has linked the original node to the new node.

Thus, it is safe for the search operation to go to the new node, and start searching for the desired key in that node. A search operation not only compares the desired key with the node's `high-key` before it starts searching for the desired key, but also it does the same thing before it compares the desired key with any `key` in the `key-list`. After a comparison, the search operation moves to the node immediately following the current node, if the `high-key` of the current node is found to be less than the desired key.

If the split operation changes the `high-key` of the node after the search operation compares the desired key with the `high-key`, that does not create a problem between the searcher and the splitter because, before comparing the next key in the `key-list` with the desired key, the searcher will do the same comparison again, and will be able to detect the change in the `high-key` of the original node. If the search process is looking at the rightmost key in the `key-list` of the original node, and the concurrent split operation changes the `high-key` after the comparison, that also does not affect the search operation (even though there is no next turn to compare the desired key with the `high-key` of the original node). If the last key is the desired key being searched for, this key has already been copied to the new node which is now linked to the tree. Thus, if the search operation correctly concludes that the desired key exists in the tree. Conversely, if the last key is not the desired key being searched for, then its copy in the new node is also the last key in the new node's `key-list` and the desired key does not exist in the tree. When the search process sees that the last key in the `key-list` is not the desired key, the process knows that the key being searched for does not exist in the tree, which is again a correct conclusion.

During a search process, when a concurrent split operation copies the right half of the `key-list` to the new node, the new node is not yet linked into the tree. Furthermore, the original node contains the right half of the `key-list` until the new node is linked to the original node, and the original node's `high-key` is changed. When the original node's `high-key` is changed, the search process discovers that there is a concurrent split operation in the same node, and moves to the new node, if necessary. As a result, when the split operation removes the right half of the original node's `key-list`, the search process that was working on the right half of the `key-list` is already in the new node. Hence, it is impossible for the search operation to be in the right half of the `key-list` of the original leaf after that portion of the `key-list` is reclaimed to the `free-list` by the split operation.

6.2.2 Concurrency Control in Update Operations

There are two types of update operations in a B^{list}-tree. The first is the insert operation, where an insert process inserts a new cell containing the key to be inserted in the `key-list` of a node or a leaf. The second is the delete operation, where a delete process deletes the cell containing the key to be deleted from the `key-list` of a leaf.

Conflicts with Other Concurrent Update Operations

The most common conflicts that update operations face are conflicts with other concurrent update operations. There can be conflicts between two concurrent insert operations, or between two concurrent delete operations, or between one insert op-

eration and one delete operation. Use of CAS (see Section 2.2.2) in the B^{list}-trees prevents these conflicts between more than one concurrent update operation.

Handling conflicts between two concurrent insert operation is straightforward. When two insert operations want to insert two new cells, each containing the same `key`[1], in the same position of the `key-list` of a node, the situation will look like that shown in Figure 6.46. Both inserters have their `hp1` and `hp2` pointing to the cells containing 20 (this cell will be referred to as `previous-cell`) and 30 (this cell will be referred to as `following-cell`), respectively. Both inserters want to insert a cell (`new-cell-1` by `inserter 1` and `new-cell-2` by `inserter 2`) both containing 25, between `previous-cell` and `following-cell`. The `next` fields of both `new-cell-1` and `new-cell-2` point to `following-cell`, containing 30. Before swinging the `next` pointer of `previous-cell` from `following-cell` to the cell to be inserted, both inserters use CAS to check whether `following-cell` still immediately follows `previous-cell` in the `key-list`. Since only one CAS succeeds, only one of the inserters changes the `next` field of `previous-cell` storing the address of the new cell to be inserted (see step 3 of the insert algorithm in Section 6.1.2). Here, if both inserters want to swing the `next` pointer of `previous-cell` at the same time, because of the atomicity property of CAS, one will succeed, and the other will fail. If `inserter 1` succeeds in swinging the `next` pointer of `previous-cell` from `following-cell` to `new-cell-1`, `inserter 2` will fail (see Figure 6.47), and vice versa. Whichever inserter fails will go back to step 4 of the insert algorithm (see Section 6.1.2) to search again for the new key in the `key-list` of the leaf.

[1]The inserters' keys do not have to be the same to cause a conflict. The keys just have to both fall between the same two keys already in the tree.

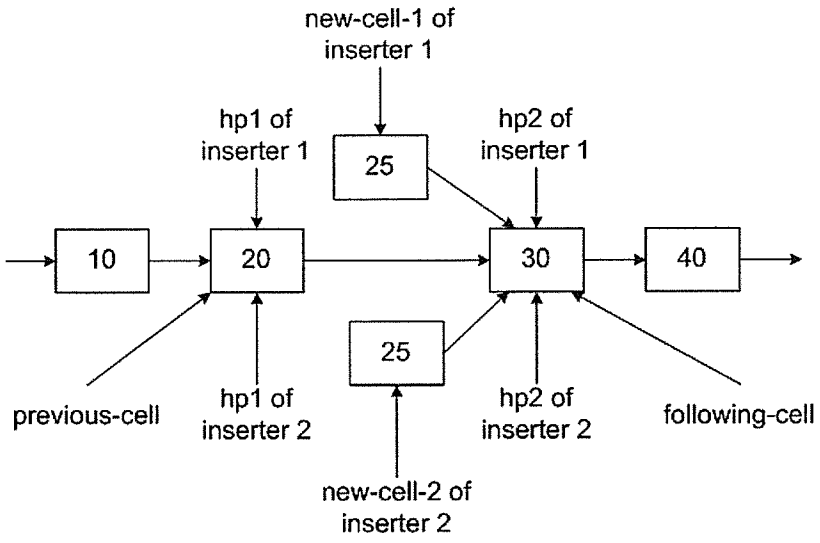


Figure 6.46: Two concurrent inserts want to insert in the same position.

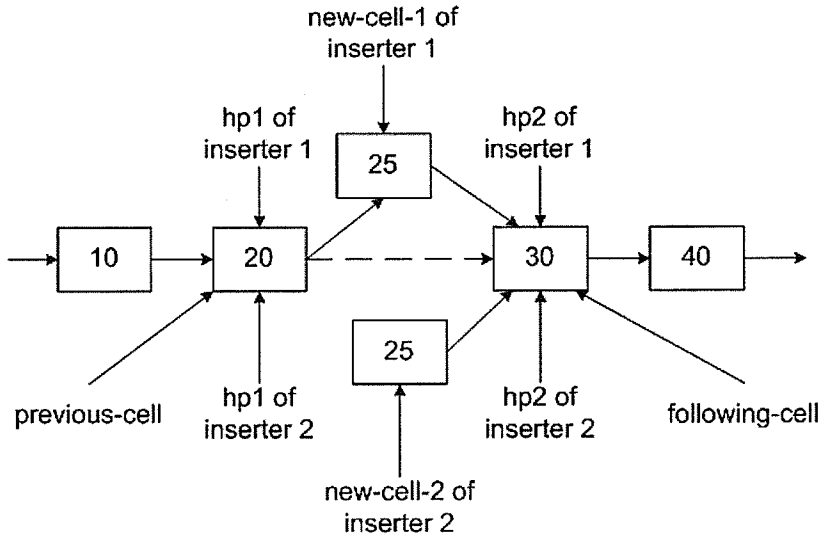


Figure 6.47: `Inserter` 1's CAS succeeds, and `Inserter` 2's CAS fails.

In the case of concurrent delete operations, two different types of conflicts can occur. In the first type, both delete operations want to delete the same cell (see Figure 6.48), and, in the other case, two delete operations want to delete two consecutive cells (see Figure 6.51).

Let us consider the situation shown in Figure 6.48. Here, both delete operations want to delete the cell containing 30. Both of their `hp1` pointers point to the cell containing 20 (this cell will be referred to as `previous-cell`), and both of their `hp2` pointers point to the cell containing 30 (this cell will be referred to as `current-cell`). In the first step of the deletion operation, both delete operations will try to mark `current-cell` as being deleted. A CAS operation is used to change the old status of the `next` field of `current-cell`—*unmarked*—to the new status—*marked*. When marking a cell, a deleter adds 1 to the `next` field of the cell to be deleted (see Section 3.2.2). Adding 1 sets the least significant bit of the `next` field. If a deleter adds 1 to the `next` field of a cell that is already marked, the addition changes the least significant bit of the `next` field back to 0. Thus, CAS is required to ensure that only one of the concurrent delete operations can successfully mark `current-cell` (if it is unmarked), and the other must fail.

Figure 6.49 shows the situation when one deleter succeeds in marking `current-cell` as deleted. No matter which deleter succeeds in marking the cell, once the cell is marked by one of them, both deleter processes will try to swing the `next` pointer of `previous-cell` from pointing to `current-cell` to point to the cell immediately following `current-cell`, using a CAS operation. Here also, because of the nature of the CAS primitive, only one delete operation will succeed in swinging the

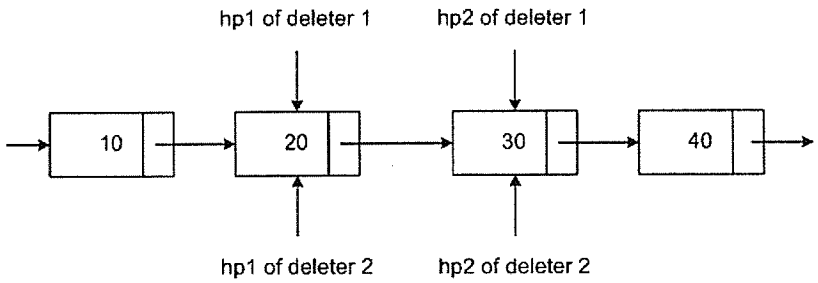


Figure 6.48: Two concurrent deleters want to delete in the same position.

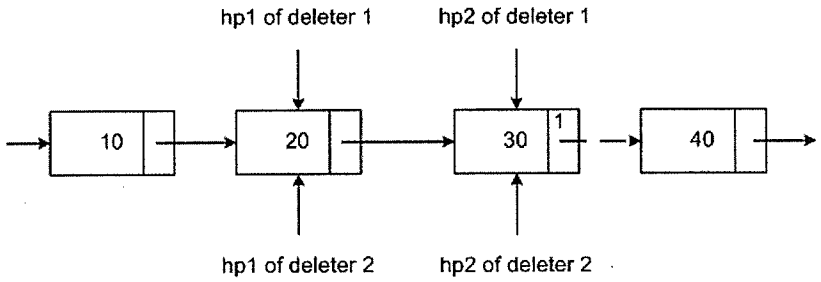


Figure 6.49: One deleter succeeds in marking the cell to be deleted.

`previous-cell->next` pointer to point to the cell immediately following `current-cell`, and the other will fail (Figure 6.50). Whichever deleter fails will go back to the second step of the delete algorithm (see Section 6.1.3), and will start searching again for the key to be deleted in the `key-list` of the same leaf. Use of the CAS operations in the delete algorithm thus ensures no conflict between concurrent delete operations that want to delete the same cell from the same `key-list` of the same leaf.

The second type of conflict between two concurrent delete operations will occur if they want to delete two consecutive cells from the key list of the same node. In the example shown in Figure 6.51, the first deleter (`deleter 1`) wants to delete the cell

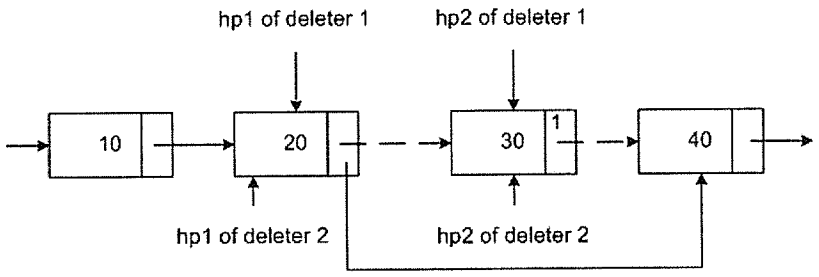


Figure 6.50: One deleter succeeds, but the other deleter fails to delete the cell from the `key-list`.

containing 20, and the second deleter (`deleter 2`) wants to delete the cell containing 30, and these cells happen to be two consecutive cells in the `key-list` of the same leaf. Let us refer to the cell containing 10 as `previous-cell-1`, the cell containing 20 as `current-cell-1`, and the cell containing 30 as `following-cell-1` for the first deleter. Similarly, let us refer to the cell containing 20 as `previous-cell-2`, the cell containing 30 as `current-cell-2`, and the cell containing 40 as `following-cell-2` for the second deleter. From Figure 6.52, we can see that `current-cell-1` and `previous-cell-2` point to the same cell, and `following-cell-1` and `current-cell-2` point to the same cell. Now, if the second deleter can complete the deletion of `current-cell-2` before the first deleter marks `current-cell-1` for deletion, there will be no conflicts among the concurrent delete operations (see Figure 6.53). The first deleter will mark `current-cell-1`, and eventually will delete it without any conflict with the second deleter. However, once the first deleter marks `current-cell-1` (and assuming the second deleter did not delete `current-cell-2` from the `key-list`), the `next` pointer of `current-cell-1` no longer points to `following-cell-1` (see Figure 6.54). This means that, for the second deleter, `previous-cell-2->next` no longer

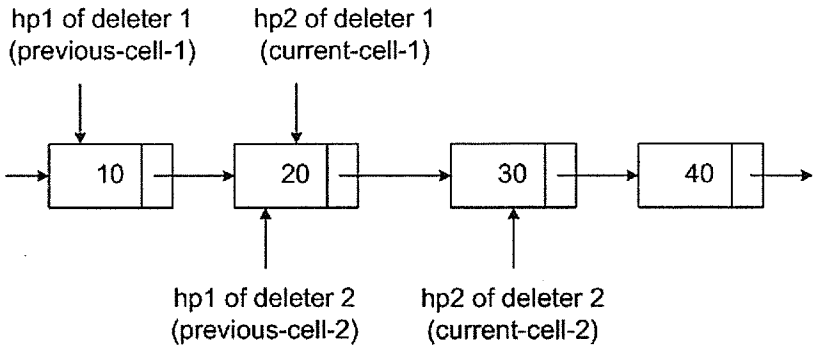


Figure 6.51: Two concurrent deleter operations want to delete two consecutive cells.

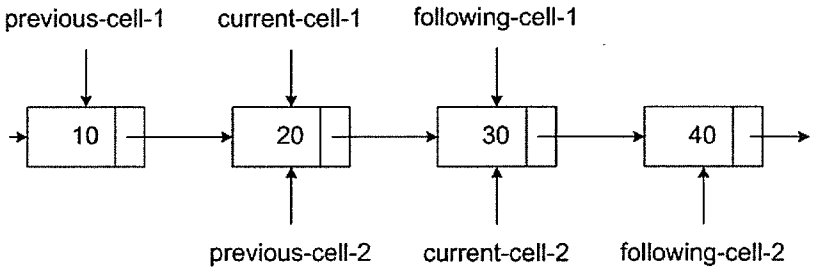


Figure 6.52: Two concurrent deleters want to delete two consecutive cells.

points to `current-cell-2`. Therefore, the CAS operation performed by the second deleter to swing `previous-cell-2->next` from `current-cell-2` to `following-cell-2` will fail. As the CAS operation fails, the second deleter unmarks `current-cell-2` (see Figure 6.55), and goes back to search again for the key to be deleted in the `key-list` of the same leaf.

In addition to the concurrency handling techniques discussed above, using hazard pointers (see Section 5.1.4) avoids conflicts related to memory reuse between concurrent operations in a B^{list}-tree. When a delete operation deletes a cell from the

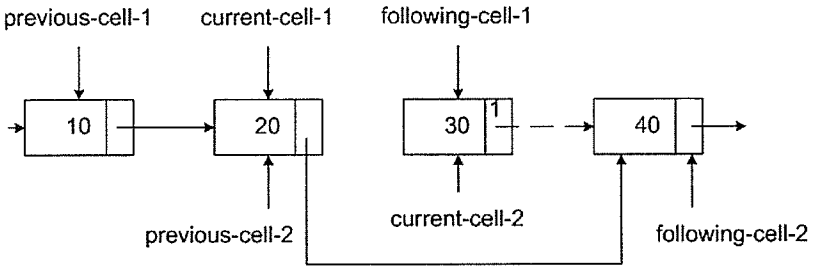


Figure 6.53: The second deleter deletes the cell before the first deleter marks its cell to be deleted.

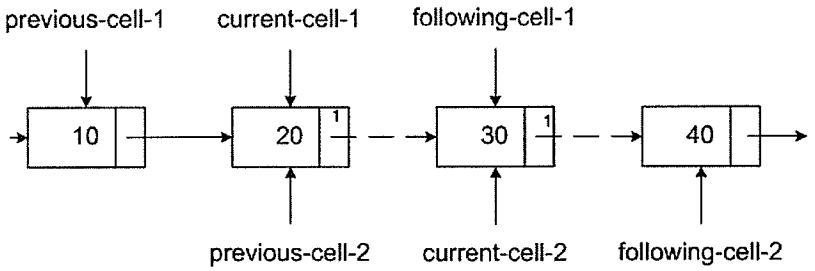


Figure 6.54: The first deleter marks the cell to be deleted before the second deleter deletes current-cell-2.

`key-list` of a leaf, instead of reclaiming that cell to the `free-list` immediately, the delete process stores the address of the cell in its `retired-cell` stack until no hazard pointers of any process point to that cell (see step 4 of the delete algorithm in Section 6.1.3). Since the address of a deleted cell is kept in the `retired-cell` stack, other concurrent operations can finish their jobs on that cell without facing any conflicts. If there are more than two concurrent update operations in the same node that have potential conflicts, the conflicts are handled in the same way as they are handled between two concurrent update operations in the same node.

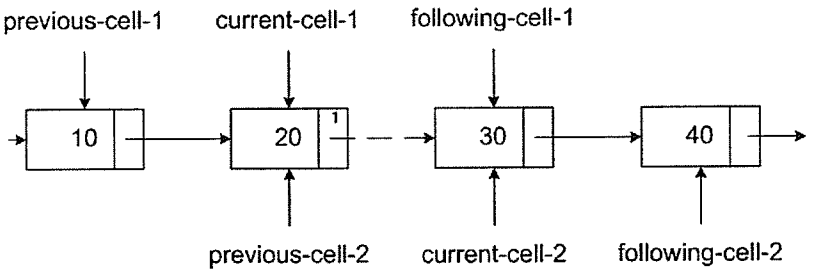


Figure 6.55: The second deleter fails to delete and unmarks the cell to be deleted.

Conflicts with Concurrent Split Operations

The B^{list} algorithms do not have any concurrency control techniques for concurrent update and split since it is impossible to start a split operation in a node (or in a leaf) when there are ongoing concurrent insert or delete operations. Every update process in a node of a B^{list}-tree, whether an insert or delete process, gets a cell from the node's `free-list` before it starts the update operation.

An insert process copies the new key into that cell and inserts that cell in that node's `key-list`. When that cell is completely linked into the `key-list`, then the insert process increments the `key-counter`. An insert process calls a split operation if and only if the `key-counter` is incremented to m. If more than one concurrent insert operations are happening in the same node so that there are no free cells left in the `free-list`, and other update processes are waiting to get a free cell, none of the ongoing insert processes can request a split operation until each of them succeeds in inserting their new key, and the `key-counter` incremented to m. Once the last insert process increments the `key-counter`, and the `key-counter` hits m, that insert process will invoke a split operation on that node. Therefore, a split operation cannot

start when there is another concurrent insert in the same node.

Similarly, if there is a delete process deleting a key from a leaf, no concurrent inserter can perform a split operation. Like the insert processes, a delete process also holds a free cell from the `free-list` before it starts deleting. Suppose a delete process wants to delete a key (and its cell) from a node, and currently the `key-counter` of that node is $m-1$. Since the `key-counter` is $m-1$, we can conclude that there is only one free cell left in the `free-list`, and no concurrent inserter is trying to insert a key in the `key-list`. If there are any concurrent inserters in the same leaf, all of them have already inserted their keys in the `key-list`, and also have finished incrementing the `key-counter` of the leaf. When the delete process grabs the last cell from the `free-list`, the `free-list` becomes empty. Hence, if any update process wants to insert or delete any key in the same node, it has to wait until this deleter process returns that cell to the `free-list`. The deleter process will return the cell only after it decrements the `key-counter` of the node. Whether the deletion takes place or not, when there is a free cell in the `free-list`, the `key-counter` of the leaf will be no more than $m-1$. As a result, no inserter will perform a split operation until the deletion is finished. As it is impossible to have concurrent delete and split operations in the same leaf, there is no need to design concurrency control techniques for concurrent split and delete operations.

6.2.3 Concurrency Control in Split Operations

The B^{list} algorithms do not need any concurrency control techniques for a split operation, since it gets the highest preference among all operations in a B^{list}-tree,

with respect to concurrency. All update and split operations in a node have to wait until an active split operation splits the node completely (see the insert and delete algorithms in Section 6.1.2 and Section 6.1.3, respectively, for details). A search operation can be done concurrently with a split operation, however, since the search operations only read data from the nodes, they do not create any conflict with the split operations. Techniques for handling the concurrency effects of a split operation on search operations were described in Section 6.2.1, and on update operations were described in Section 6.2.2.

Chapter 7

Assessment

7.1 Experimental Setup

To assess the implementation of my B^{list}-tree, I have compared its performance with Lehman and Yao's B^{link}-trees [24]. The B^{list}-tree algorithms are designed for in-memory applications that run efficiently on shared memory machines including Non-Uniform Memory Access-time (NUMA) machines, rather than for the traditional disk-based applications for which Lehman and Yao's B^{link}-tree algorithms were originally designed.

To achieve a fair comparison between B^{link} and B^{list}-trees, I made some modifications to the B^{link}-trees. I have implemented an in-memory version of Lehman and Yao's B^{link}-tree algorithms. In the B^{link}-tree implementation, the keys in the nodes are stored in an array, whereas in my lock-free B^{list}-tree the keys in each node are stored in a *localized* linked list. Furthermore, in B^{link}-trees, the keys residing in a node (both leaf and non-leaf) and the child pointers pointing to the children of a

node are implemented separately as an array of keys and an array of child pointers, respectively. The child pointers and their associated key separators are distinguished using the array indexes (e.g., `key[0]` separates `child[0]` and `child[1]`). However, in my B^{list}-trees, each key and its associated child pointers are stored in a `cell` object that is a part of the localized linked list of a node (see Section 5.1). The complex node structure of B^{list}-trees makes the nodes of the trees noticeably larger than the nodes in B^{link}-trees. Because of the larger size and the complex structure of the nodes, the B^{list}-tree algorithms take significantly higher time to access a key in a node compared to the key access time in a node of the B^{link}-tree algorithms.

In B-trees and their variants, major operations like searches, inserts and deletes take place in a node (or a leaf). To more directly compare the lock-based and lock-free performances within a node, I also implemented a variant of Lehman and Yao's B^{link}-tree in which the keys in a node (both leaf and non-leaf) are stored in a localized linked-list instead of being stored in an array. To distinguish the two different variants of Lehman and Yao's B^{link}-tree, I will refer to the B^{link}-tree using the array-based node structure as the *B^{link}-array*, and to the B^{link}-tree using the linked-list-based node structure as the *B^{link}-linked-list*. The B^{list}-tree will, of course, be referred to by its own name.

7.1.1 Test-bed Environment

I have done a complete set of experiments using the Sun Fire x4600 shared memory machines available in the Department of Computer Science at the University of Manitoba. Each test-bed machine, *Helium-01* through *Helium-05*, has eight proces-

sors with dual cores so that, in total, the machine can work with 16 different threads concurrently. Due to the memory architecture (where each processor has its own DIMMS) the Sun Fire has NUMA behavior but is not optimized for this. In addition to the Sun Fire machines, I have also conducted some of my experiments on an SGI origin 3000 SMP machine, *helios* [1], provided by Westgrid [2]. Helios has 32 MIPS R14000 CPUs and offers highly optimized cache-coherent (CC-NUMA) memory system.

7.1.2 Programming Language

To implement the two variants of B^{link}-tree (the B^{link}-array and the B^{link}-linked list) and the B^{list}-tree, I used C++ as my programming language. Additionally, I used the POSIX threads (pthreads) libraries to create multiple simultaneous threads so that my program can concurrently run on the different processors of the Sun and SGI machines.

7.1.3 Parameters Varied

In my experiments, I recorded the running time of the B^{link}-array, the B^{link}-linked-list, and the B^{list}-tree algorithms for different mixes of operations (search, insert, and delete), different values of m, different key ranges, and different numbers of concurrent processes[1].

[1] Although I used pthreads in my implementation, I will continue to use the term *process*.

Different Operation Types

To begin each experiment on the B^{link}-array, the B^{link}-linked-list, and the B^{list}-tree, I populated the tree with keys inserted by one million uniformly distributed random sequential insert operations. Then, over multiple identical runs, I measured the average execution time for one million concurrent operations consisting of different proportions of search, insert and delete operations. I divided my experiments into five categories according to the ratio of the search, insert, and the delete operations. The first experiment considered only concurrent search operations. In the second, third, and fourth experiments, I included concurrent insert operations along with the search operations. In those three experiments I decreased the number of the concurrent search operations, and increased the number of concurrent insert operations gradually to see the effect of concurrent insertions in the trees. In the fifth experiment, I have examined the running time for equal numbers of concurrent search, insert, and delete operations in all types of trees.

Different Values for m

For each of my five experiments, I also assessed the performance for different values of m for both B^{link}-trees and the B^{list}-tree. In any B-tree variant, using a smaller m, like 100, creates a taller tree than using a larger m, like 1000, assuming the number of keys in the tree is the same. If the tree is taller, then the algorithms should take more time to find the correct leaf to perform the search, insert and delete operations. Conversely, in a shorter tree, with larger m, the algorithms should take more time to find the correct place within a leaf (or node) for the various operations. Moreover, a

larger m might introduce more data contention in the same node. To see the effect of the value of m in the trees, I have experimented with 100, 200, 500, 700, and 1000 as the values of m.

Different Key Ranges

The key range affects the number of nodes in the initial tree constructed by the serial insertions and also by any concurrent update operations performed after the initial tree is constructed. If the key range is small compared to the number of serial insertions, for example, a key range of 1–250,000 and 1,000,000 serial insertions, then the initial tree will likely contain nearly all the keys in the range. Since the range is 1–250,000, the tree will never be very large because there are only 250,000 keys. Furthermore, concurrent insertions performed after initial construction will fail because the keys are all already in the tree, so the insertions degenerate to searches. Therefore, a mix of searches and insertions with no deletions will be similar to 100% searches except that the insertions allocate a free cell in the appropriate leaf, whereas searches do not. Most concurrent deletions performed after the initial construction will succeed because all keys are in the tree initially. Of course, as the number of deletions increases, more insertions will succeed.

Alternatively, if the key range is large compared to the number of serial insertions, for example, a key range of 1–10,000,000 and 1,000,000 serial insertions, then the number of keys in the tree will be close to the number of serial insertions. Furthermore, concurrent insertions after the initial construction are more likely to succeed and concurrent deletions are more likely to fail.

To assess the effect of key range on algorithm performance, I have used the following different key ranges: 1–250,000, 1–1,000,000, 1–2,000,000, and 1–10,000,000. In all cases, I have done 1,000,000 initial serial insertions to populate the tree and then 1,000,000 concurrent operations with mixes, as described earlier.

Different Numbers of Processes

In addition to assessing the effect of operation mix, m and the key range on the algorithms, I have also measured the running time of one million concurrent operations on each tree performed by different numbers of processes. As the number of concurrent processes increases, chances of conflicts also increase. Since the one million concurrent operations are divided among the processes, the number of operations per process decreases as the number of processes increases. Consequently, the overall running time is expected to decrease, subject to concurrency overhead.

To determine the trade-off between the number of processes and the performance gain, I ran all of my experiments for 1, 2, 4, 6, 8, 12, 14, and 16 processes on the Sun machines, and for 1, 2, 4, 8, 16 and 32 processes on the SGI machines. This was done for all values of m, all key ranges, and all concurrent operation mixes.

Memory Affinity

Instead of sequentially building the tree, I also built all three types of trees using eight concurrent processes. Thus, each process should have built its own part of the tree in its associated memory. My expectation was that, during the concurrent operations, accessing the leaves (or nodes) in each process's associated memory would be less time consuming than accessing nodes in another process's associated memory.

However, the experimental results did not show any significant effect of exploiting memory affinity. This lack of effect was likely due to efficient caching because of the design for locality in my algorithm. Therefore, I decided to simply build the tree with a single process running on a single core for the reported experiments.

7.2 Results

In my experiments, I recorded the running times of the B^{link}-array, B^{link}-linked-list, and B^{list}-tree for 10 runs per experiment, and then calculated their average running time to compare their performances. In the next sections, I will describe the results of the various experiments on the Sun machines first. Then, I will describe the results of a subset of the experiments run on the SGI machines[2]. To have a fair comparison in all experiments, I have maintained the same scale for all result graphs.

7.2.1 Sun Machine with Eight Processors

The Sun machines are NUMA-style machines with eight dual-core processors. In my experiments, I created threads only on the second core of each processor to ensure that no processes shared the same data bus to access the shared memory until the number of processes was more than eight. (When there are more than eight processes, two processes simultaneously share the second core of each processor of the Sun machines.) More precisely, if there are 10 processes, process 1 and 9 share the second core of the first processor, and process 2 and 10 share the second core of the

[2]It was not possible to run all experiments on the shared SGI machines due to long waiting times to access them. Hence, a representative subset of the experiments were chosen.

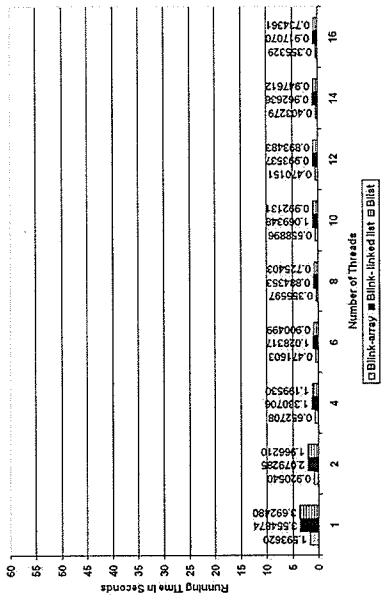
second processor. This was done because during early experimentation on the Sun Fire machines it was discovered that the memory access path shared by the cores on a single processor quickly became saturated and resulted in a performance bottleneck. This phenomenon was not observed on the SGI machines.

Experiment 1: 100% Search Operations

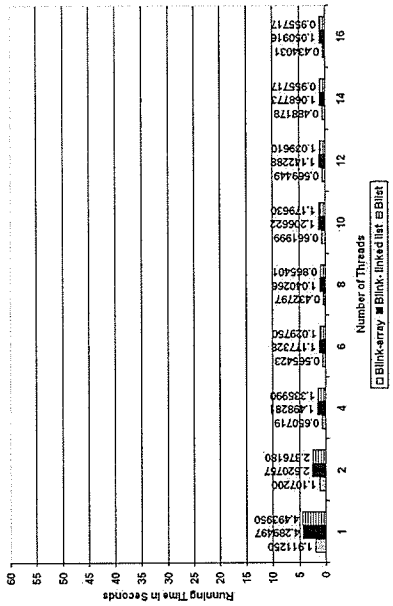
In my first experiment, after building the B^{link}-array, the B^{link}-linked-list, and the B^{list}-tree sequentially, I ran 1 million concurrent search operations on them. Since there are 100% search operations in this experiment, no data is changed in the tree, hence, no conflicts occur. Therefore, none of the trees needed to use any lock-based or lock-free concurrency control techniques during the experiments. Figures 7.1–7.5 show the raw results of these experiments.

For all three types of trees I considered, the time required to finish one million concurrent search operations decreases as the number of processes increases, if there are eight or fewer processes. Sometimes, there is a slight increase in running time when there are more than eight processes (usually when there are 10 processes). In my experiments on the Sun machines, the processes are run only on the second core of each processor. Since there are eight processors, when the number of processes is more than eight, then more than one process must share the same core on the same processor. Even though there are no data conflicts for 100% search operations in all types of B-trees, when some processes share the same core, there is some performance degradation due to thread-switching overhead, memory access contention, etc.

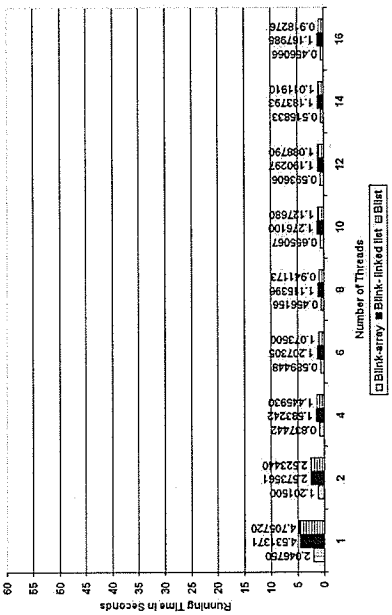
By examining the experimental results, we can see that the B^{link}-array tree is



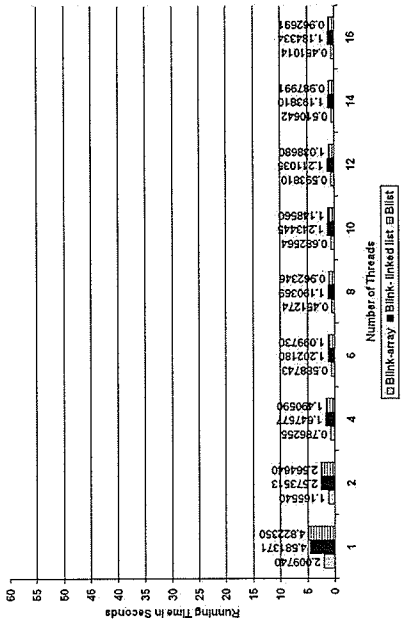
(a) Key range is 250,000.



(b) Key range is 1,000,000.

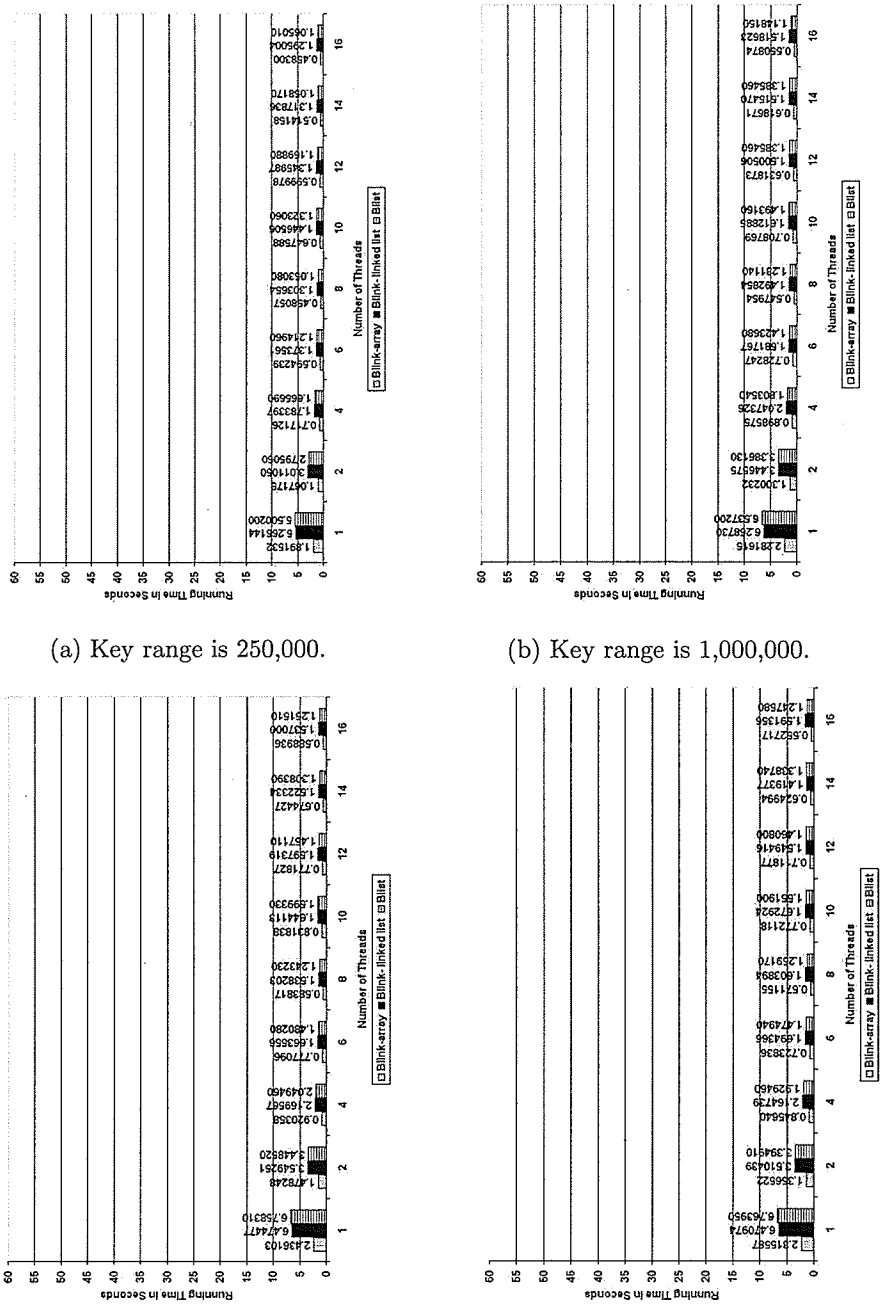


(c) Key range is 2,000,000.



(d) Key range is 10,000,000.

Figure 7.1: Result of 100% search operations on Sun Fire (m is 100).



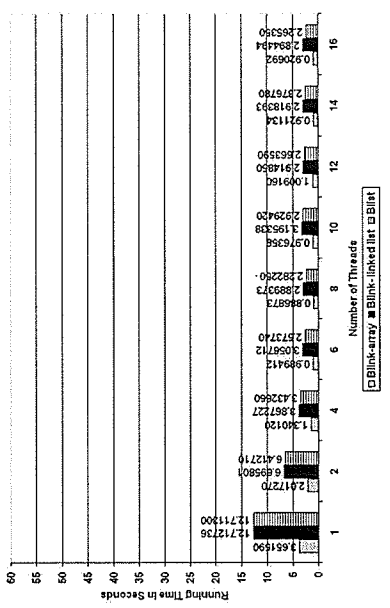
(a) Key range is 250,000.

(b) Key range is 1,000,000.

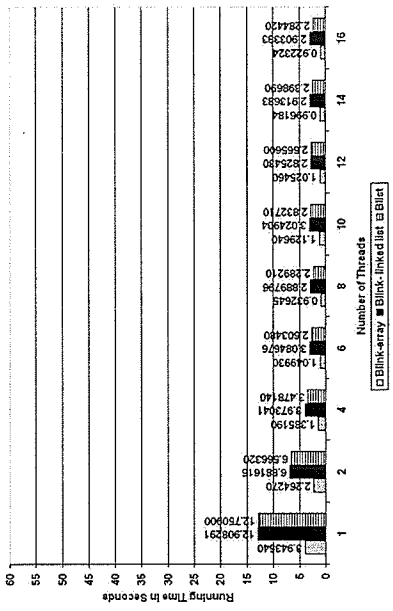
(c) Key range is 2,000,000.

(d) Key range is 10,000,000.

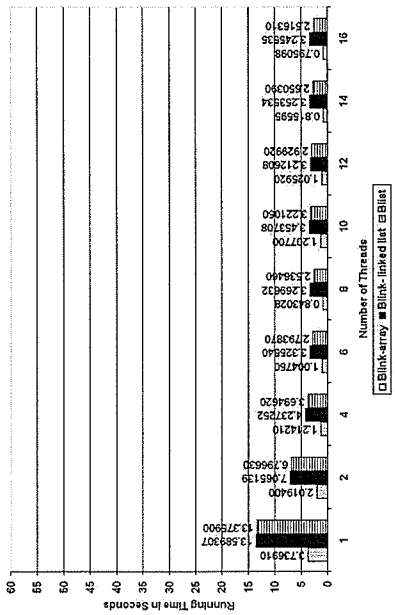
Figure 7.2: Result of 100% search operations on Sun Fire (m is 200).



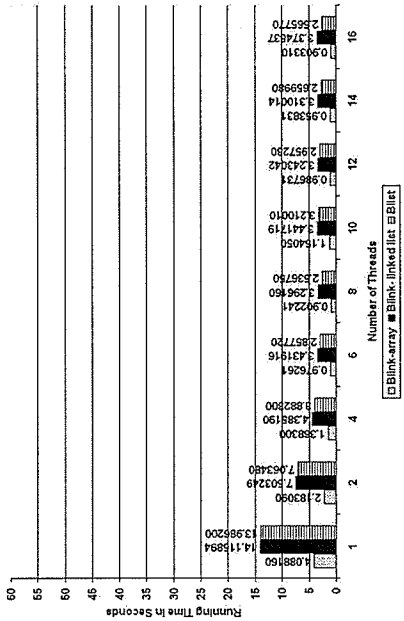
(a) Key range is 250,000.



(b) Key range is 1,000,000.

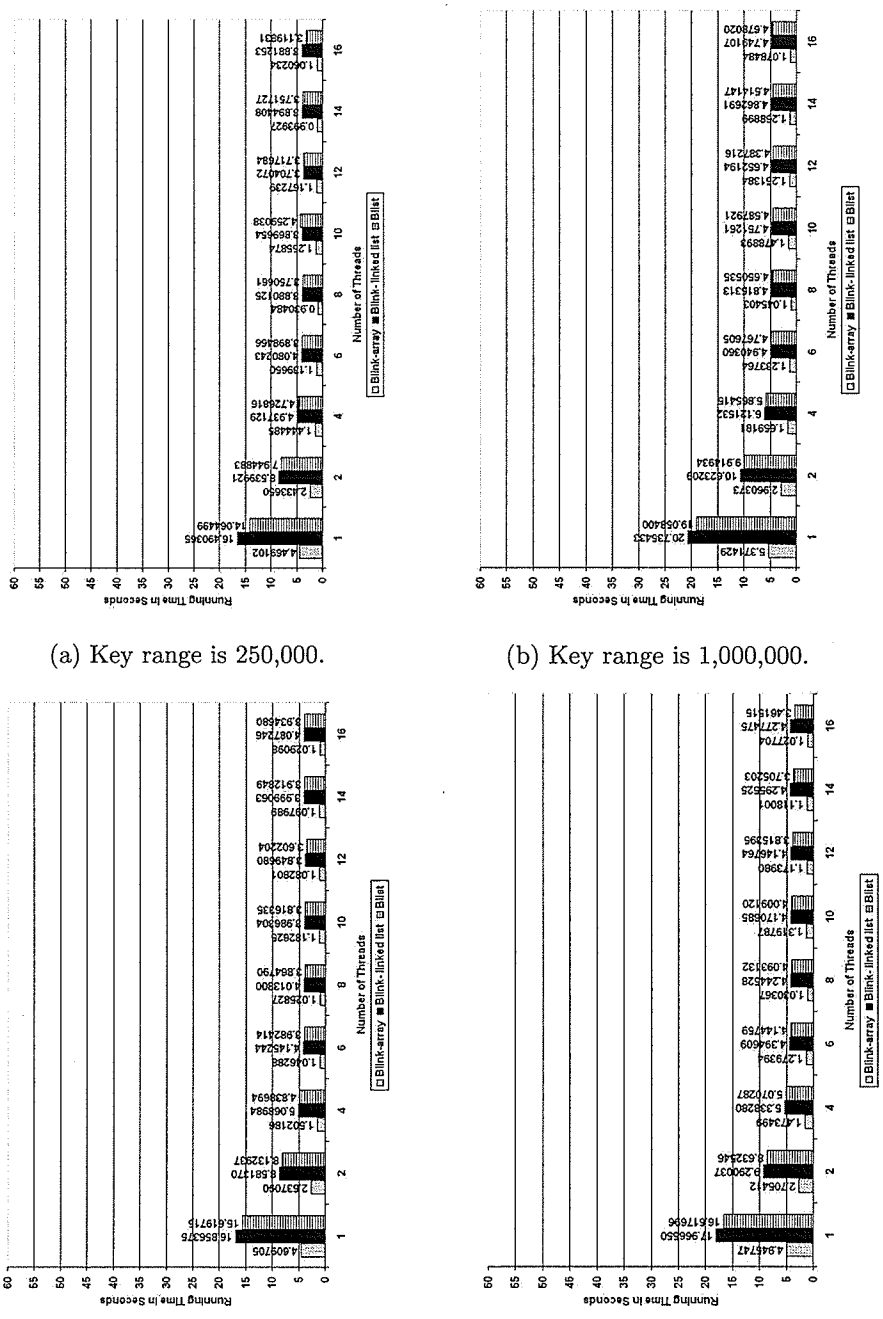


(c) Key range is 2,000,000.



(d) Key range is 10,000,000.

Figure 7.3: Result of 100% search operations on Sun Fire (m is 500).



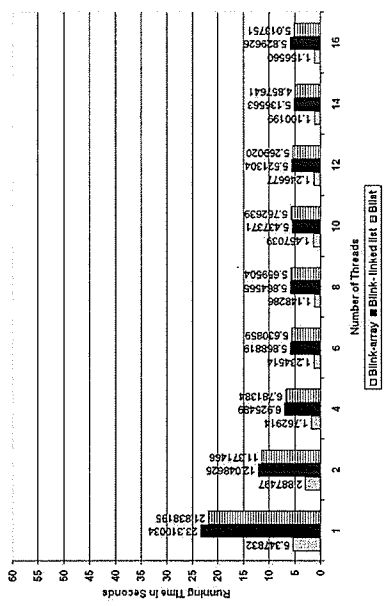
(a) Key range is 250,000.

(b) Key range is 1,000,000.

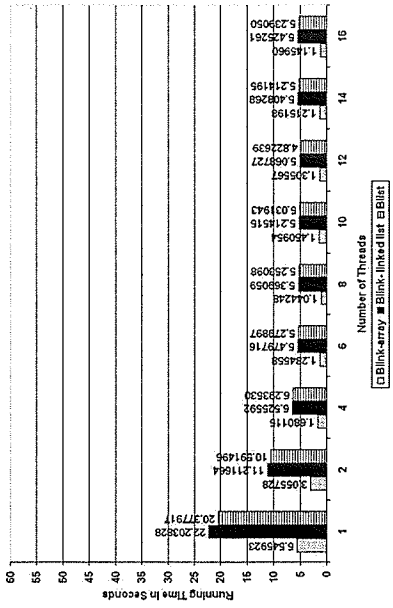
(c) Key range is 2,000,000.

(d) Key range is 10,000,000.

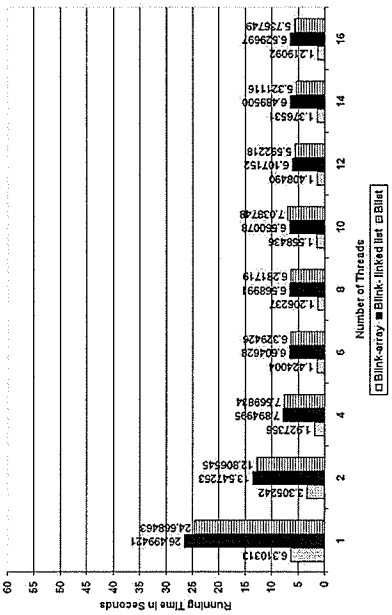
Figure 7.4: Result of 100% search operations on Sun Fire (m is 700).



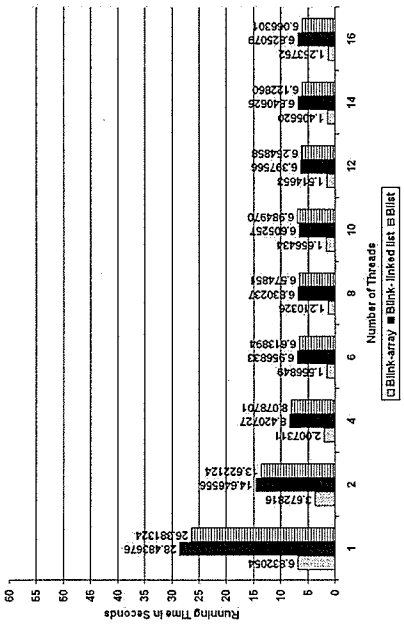
(a) Key range is 250,000.



(b) Key range is 1,000,000.



(c) Key range is 2,000,000.



(d) Key range is 10,000,000.

Figure 7.5: Result of 100% search operations on Sun Fire (m is 1000).

always more efficient than the B^{link}-linked-list and the B^{list}-tree. For a smaller m like 100 or 200 (see Figures 7.1–7.2), when there is only one process, the B^{link}-linked-list is more efficient than the B^{list}-tree. However, when the number of processes increases but m is unchanged, the B^{list}-tree takes less running time than the B^{link}-linked-list tree. When the value of m gets larger than 200 (see Figures 7.3–7.5), then the B^{list}-tree always performs better than the B^{link}-linked-list tree.

There is another interesting thing to notice in these experiment results. For all types of trees, with the same key range, but with different values of m, the running time increases as the value of m increases. For example, in a B^{link}-array tree, with a key range of 1 to 250,000, for one process and $m = 100$, the running time is 1.594 seconds (Figure 7.1a), but the running time is 5.348 seconds when m increases to 1000 (Figure 7.5a). In B-trees and their variants, the search, insert and delete operations take place in a leaf (or node). When the algorithms find the correct leaf (or node) in which to perform an operation, they need to find the correct place in that leaf (or node) for the given key. If the node is larger (i.e., m is larger), then with a linear search, the algorithms take more time to find the correct place in the leaf (or node) than in a node with smaller m. That is why all three types of trees I have examined take more time to complete 1 million concurrent search operations as m gets larger.

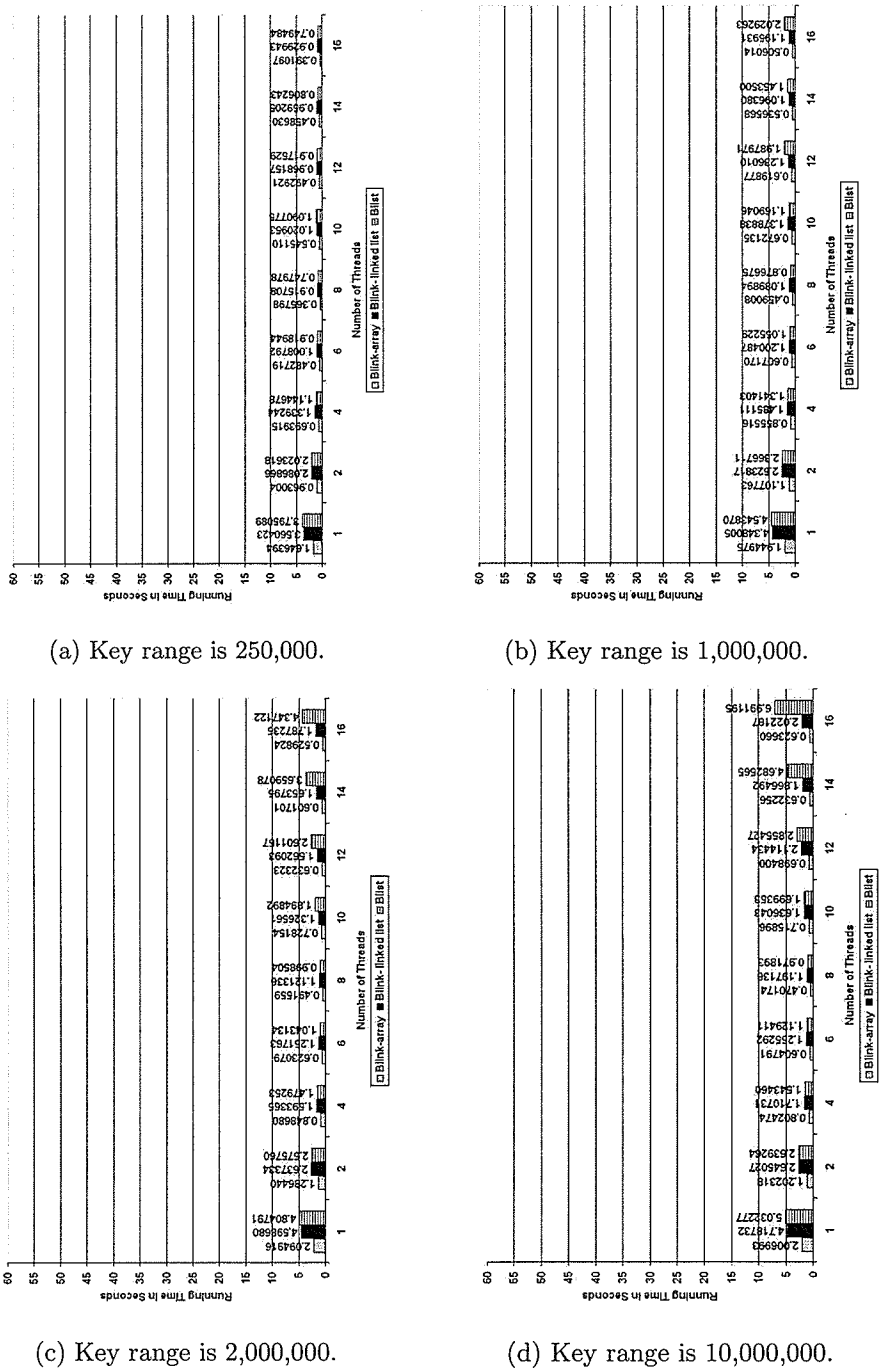
Experiment 2: 90% Search and 10% Insert Operations

My second experiment consists of 90% concurrent search and 10% concurrent insert operations. Due to the insert operations, the processes in this set of experiments could conflict with each other. To manage the inconsistencies caused by these

conflicts, the B^{link}-array and B^{link}-linked-list trees lock a particular leaf (or node) to insert the given key in that leaf (or node). The B^{list}-trees use the compare-and-swap (CAS) operation to avoid the conflicts that may be caused by the concurrent insert operations. Figures 7.6–7.10 show the results of the experiment with 90% concurrent search and 10% concurrent insert operations.

The graphs for this experiment show that the B^{link}-array tree is always more efficient than the B^{link}-linked-list tree and the B^{list}-tree. If the number of processes is eight or fewer, the B^{list}-tree is always more efficient than the B^{link}-linked-list tree. However, if m is 500 or less and only one process is running, the B^{link}-linked-list tree is slightly more efficient than the B^{list}-tree (Figures 7.6–7.8). Besides this, if eight or fewer processes are running, the running time difference between the B^{link}-linked-list tree and the B^{list}-tree increases when m increases from 100 to 500 (Figures 7.6–7.8). In the B^{link}-linked-list tree, the nodes are locked for the insertion of the given key in the correct leaf (or node). As the node size increases, the data contention also increases since the tree becomes shorter than a tree with smaller m. As a result, a process holding the lock of a larger node might create a bottleneck in accessing that node. Because of the locking overhead, the B^{link}-linked-list tree takes more time than the B^{list}-tree to finish. However, when the node size is as large as 700 or more, the chance of re-doing a CAS operation also increases in the B^{list}-tree, since the data contention in a node increases.

The performance results of B^{list}-trees with key range bigger than 1–250,000 becomes more interesting when more than eight processes run simultaneously in the eight processors of the Sun Fire machines. The results show that the efficiency of the



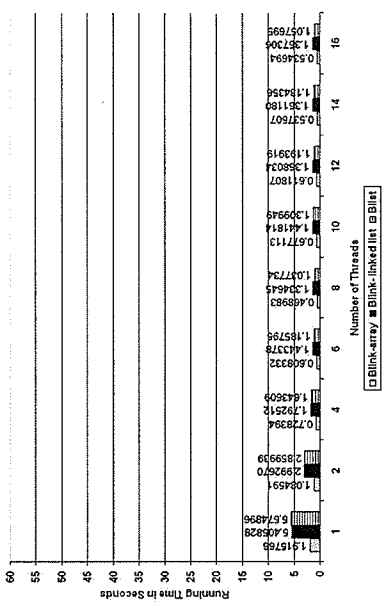
(a) Key range is 250,000.

(b) Key range is 1,000,000.

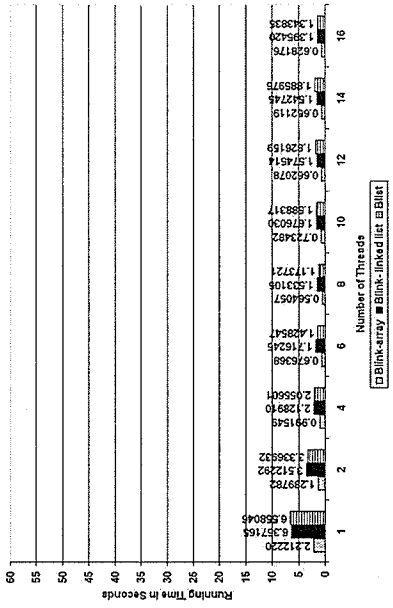
(c) Key range is 2,000,000.

(d) Key range is 10,000,000.

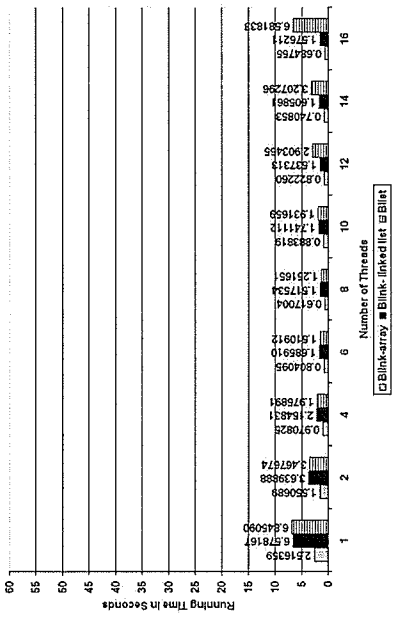
Figure 7.6: Result of 90% search operations and 10% insert operations on Sun Fire (m is 100).



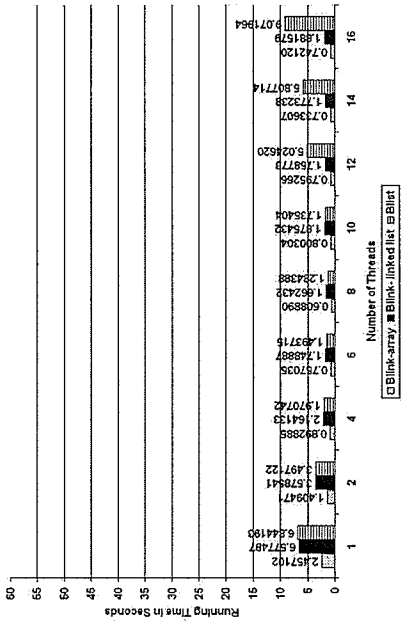
(a) Key range is 250,000.



(b) Key range is 1,000,000.

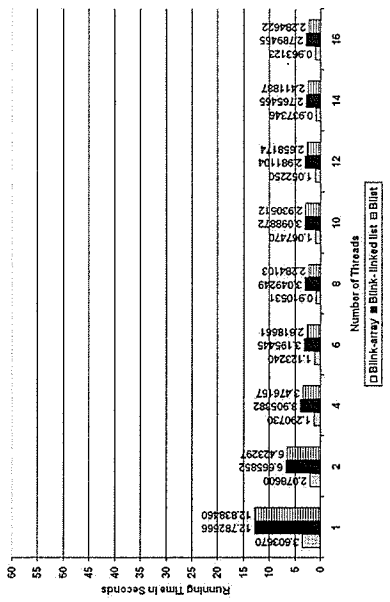


(c) Key range is 2,000,000.

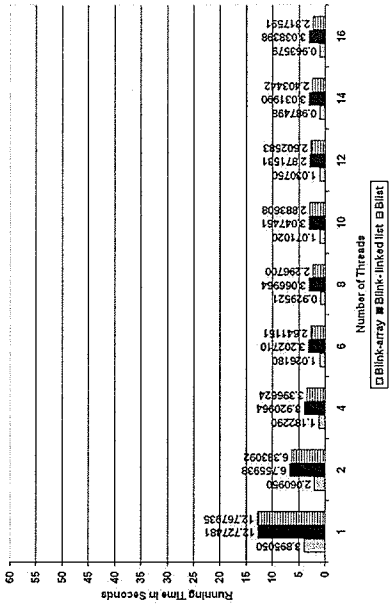


(d) Key range is 10,000,000.

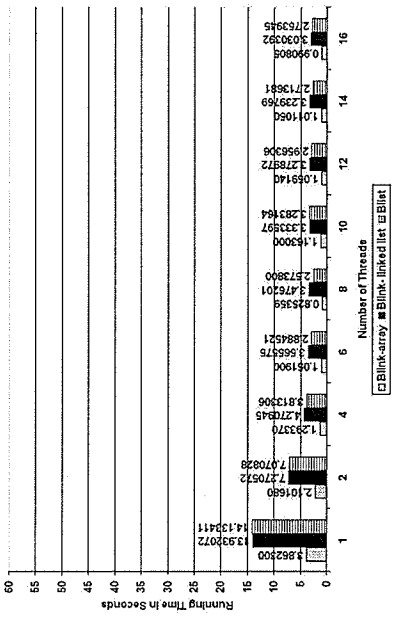
Figure 7.7: Result of 90% search operations and 10% insert operations on Sun Fire (m is 200).



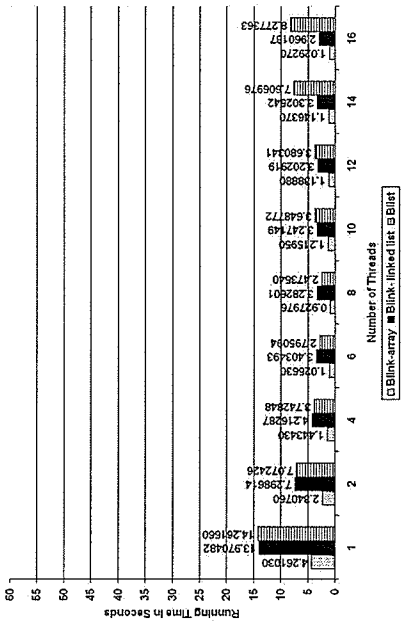
(a) Key range is 250,000.



(b) Key range is 1,000,000.

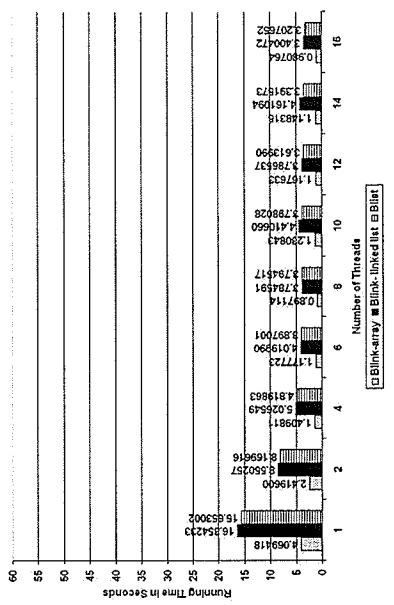


(c) Key range is 2,000,000.

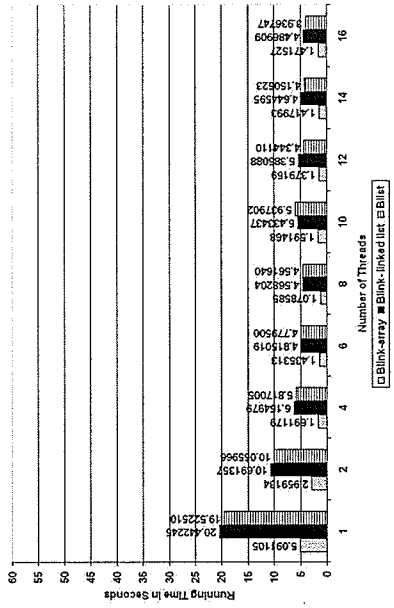


(d) Key range is 10,000,000.

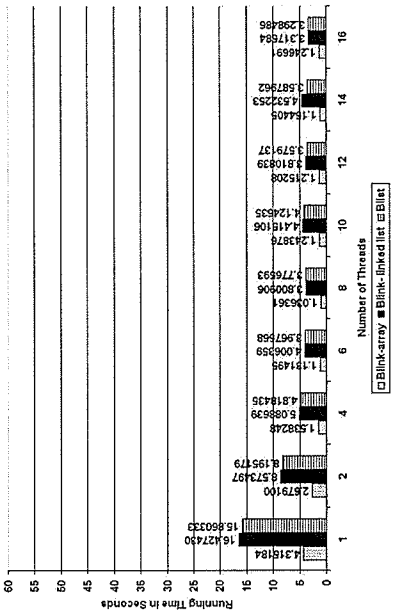
Figure 7.8: Result of 90% search operations and 10% insert operations on Sun Fire (m is 500).



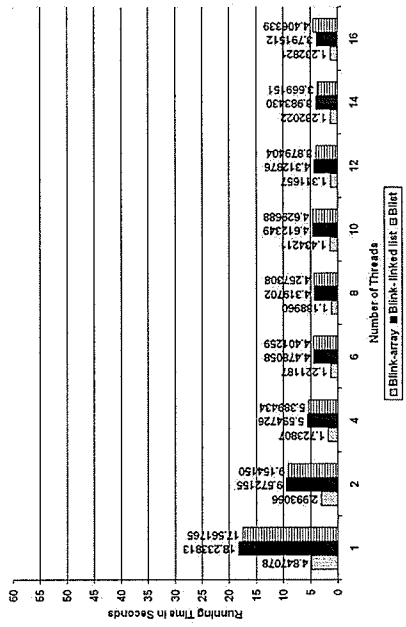
(a) Key range is 250,000.



(b) Key range is 1,000,000.

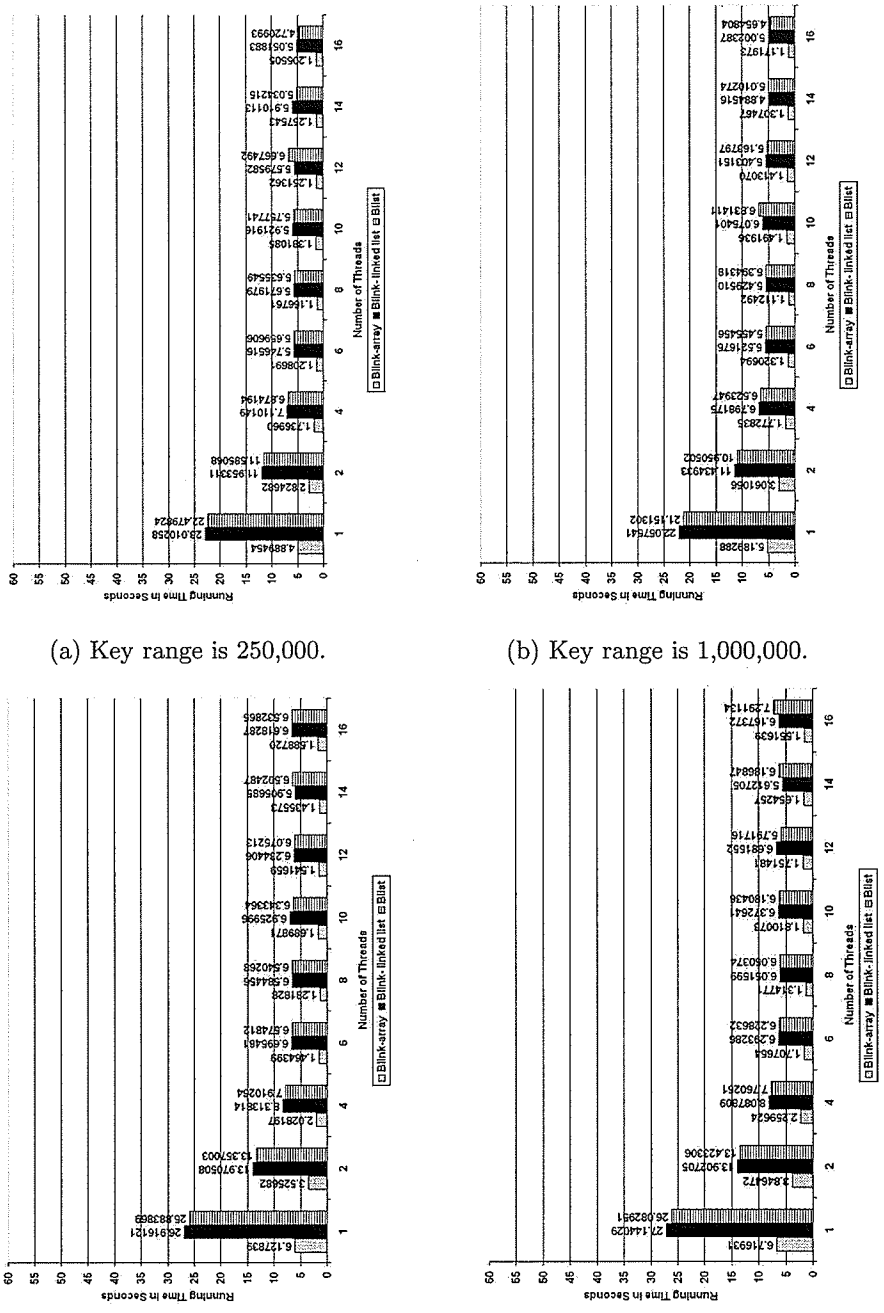


(c) Key range is 2,000,000.



(d) Key range is 10,000,000.

Figure 7.9: Result of 90% search operations and 10% insert operations on Sun Fire (m is 700).



(a) Key range is 250,000.

(b) Key range is 1,000,000.

(c) Key range is 2,000,000.

(d) Key range is 10,000,000.

Figure 7.10: Result of 90% search operations and 10% insert operations on Sun Fire (m is 1000).

B^{list}-trees depends on how big the tree is and how full the nodes (or leaves) are. In a B^{list}-tree with key range of 1–250,000 and 1,000,000 serial insert operations, during the serial insert, most of the keys will be present in the tree. Therefore, when a concurrent insert operation starts, the operation only searches for the given key, and after finding the given key in the tree, the insert operation does not proceed further. As a result, a smaller B^{list}-tree, like a tree with 250,000 keys, always performs better than a B^{link}-linked-list tree with the same number of keys. That is, B^{list}-tree are more efficient when the nodes are not filled up.

As the tree grows bigger with a larger key range (1 to 1 million, 2 million or 10 million), the B^{list}-tree sometimes takes a little more time than both types of B^{link}-trees. When the key range is 1–1,000,000 or 1–2,000,000, and m is 100, the B^{link}-linked-list tree performs slightly better than the B^{list}-tree (Figures 7.6b and 7.6c). Similar results are also observed when m is 200, and the key range is 1–2,000,000 (Figure 7.7c). Otherwise with the key ranges of 1–1,000,000 and 1–2,000,000, the B^{list}-tree performs better than the B^{link}-linked-list tree (Figures 7.7b, 7.7c, 7.8b, 7.8c, 7.9b, 7.9c, 7.10b, 7.10c). Moreover, for the key range of 1–1,000,000, the running time difference between these two types of trees increases as m increases from 200 to 1000 (Figures 7.7b, 7.8b, 7.9b, 7.10b), but when the key range is 1–2,000,000, the running time differences between the B^{link}-linked-list and the B^{list}-trees decrease as m increases from 500 to 1000 (Figures 7.8c, 7.9c, 7.10c).

In the B^{list}-tree insert algorithms, when an insert process finds the correct leaf (or node) in which to insert the given key, it gets an empty cell from that leaf's free list before it starts to search for the correct position in the leaf (or node) to insert

the given key. With a smaller m but bigger key range, the free list of that leaf could be empty. In that case, the insert processes might have to wait to get a free cell in that leaf. The waiting times of all processes in the same leaf accumulate and result in a higher running time. If m increases, the possibility of getting a free cell from a leaf (or node) increases, hence, the B^{list}-tree's running time to complete one million concurrent operations decreases. That is why, for key ranges of 1–1,000,000 and 1–2,000,000, the B^{list}-tree achieves greater efficiency than the B^{link}-linked-list tree when m is 500 or more.

The performance of the B^{list}-tree also depends on the number of successful key insertions and the number of splits, which is directly related to the key range of the tree. Before starting the concurrent operations, the tree is built with one million sequential random key insertions. If the key range is 1–1,000,000, the tree is almost filled during the tree build with close to one million random keys. When the concurrent operations start, most of the keys to be inserted in the tree are already there. Consequently, the concurrent insert operations terminate without inserting. But, when the key range increases to 1–2,000,000 or 10,000,000, after the sequential tree build, there are lots of keys that have not been already inserted. Therefore, in my experiments, when the key range is 1–10,000,000, B^{list}-trees do not perform better than the B^{link}-linked-list trees (Figures 7.6d, 7.7d, 7.8d, 7.9d, 7.10d). The reason for the poorer performance is that the values of m that I have chosen for my experiments are not large enough to eliminate the overhead of the time delay to acquire free cells from a node's `free-list` in a B^{list}-tree, especially with such a big key range. Moreover, more concurrent insertions cause more concurrent splits. Concurrent splits also force concurrent insertion

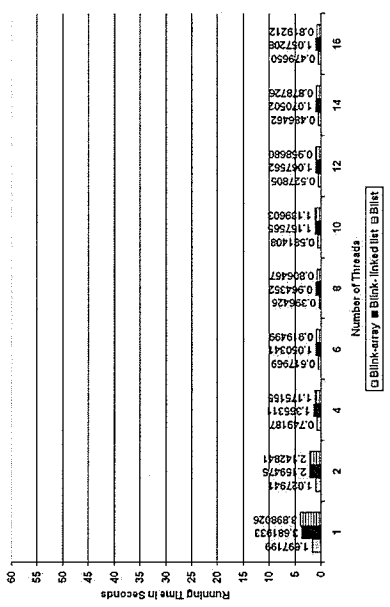
processes to sit idle until the split is done. A B^{list}-tree with a larger m, like 1500 to 2500, might perform better than the B^{link}-linked-list tree with the same key range.

Experiment 3: 50% Search and 50% Insert Operations

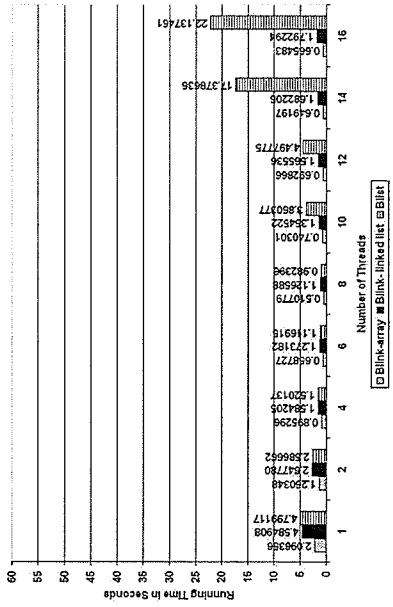
In experiment 3, each process performs 50% concurrent search and 50% concurrent insert operations on the B^{link}-array, B^{link}-linked-list, and B^{list}-trees. Since the percentage of insert operations is higher in experiment 3 than in experiment 2, the chances of conflicts with the same key range and m are also higher in this experiment. To handle the conflicts in this experiment, the algorithms I have examined use more lock-based or lock-free concurrency-control techniques than they needed to use in experiment 2. Figures 7.11–7.15 present the results of experiment 3 with 50% concurrent search and 50% concurrent insert operations performed on the Sun Fire machines.

As in the previous experiments, the B^{link}-array trees outperform the B^{link}-linked-list trees and the B^{list}-trees. Among the B^{link}-linked-list trees and the B^{list}-trees, the B^{list}-trees are more efficient when there are eight or fewer processes concurrently running on the machines (Figures 7.11–7.15).

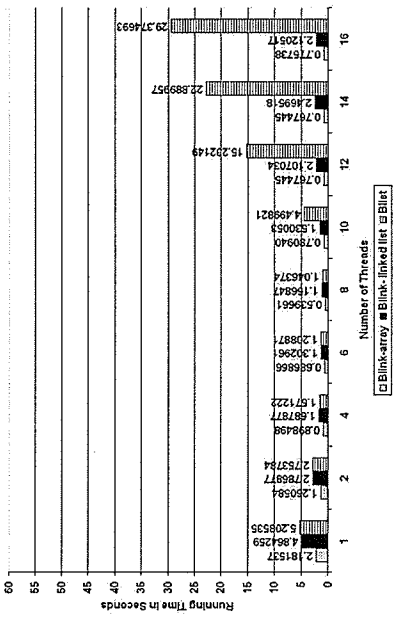
As in experiment 2, when more than one process shares the same core of the same processor (i.e., when more than eight processes run concurrently), sometimes the B^{list}-tree does not perform as well as it performs with eight or fewer concurrent processes running together. Also, like experiment 2, the size and fullness of the tree also affects the performance of the B^{list}-trees when more than eight processes run simultaneously. With a small tree with a key range of 1 to 250,000, the B^{list}-trees



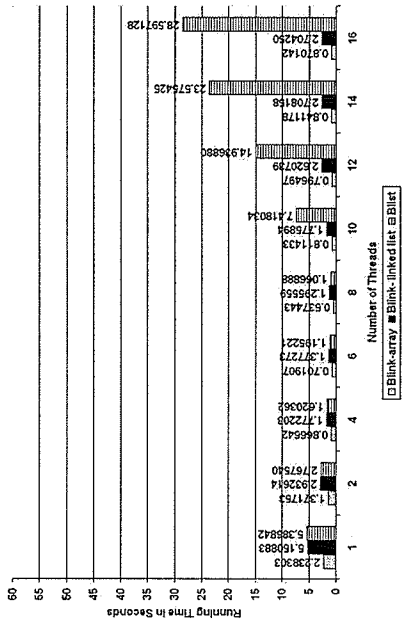
(a) Key range is 250,000.



(b) Key range is 1,000,000.

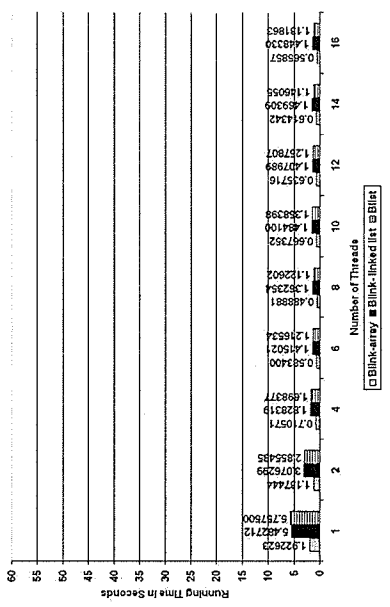


(c) Key range is 2,000,000.

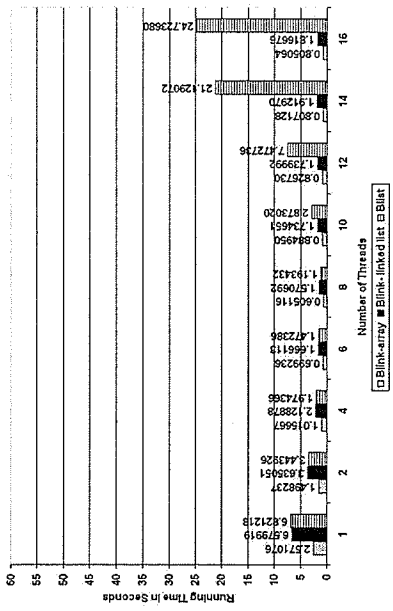


(d) Key range is 10,000,000.

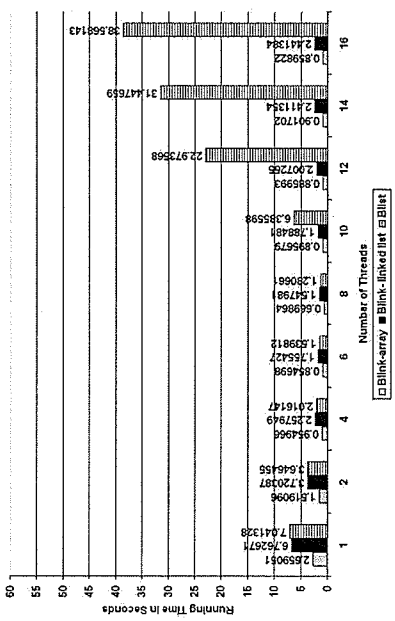
Figure 7.11: Result of 50% search operations and 50% insert operations on Sun Fire (m is 100).



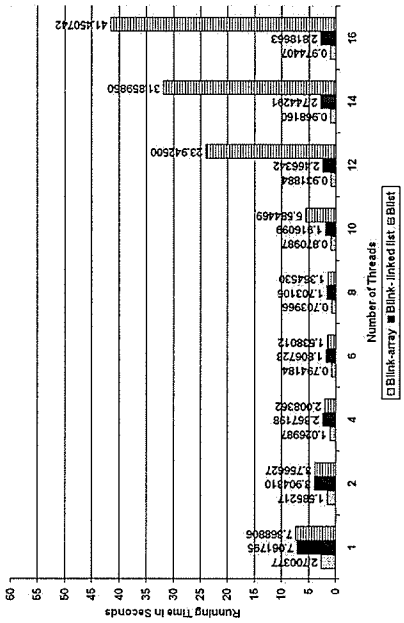
(a) Key range is 250,000.



(b) Key range is 1,000,000.

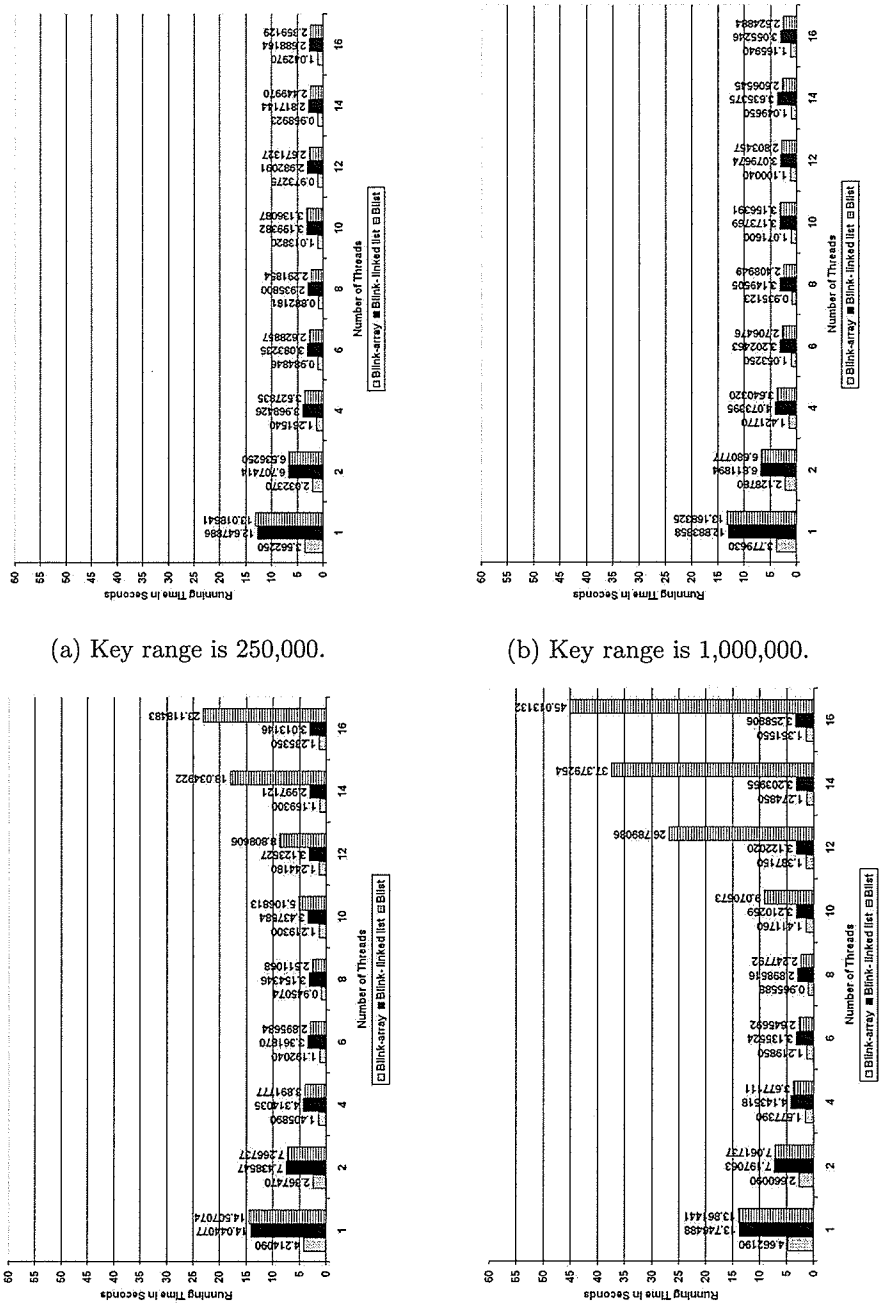


(c) Key range is 2,000,000.



(d) Key range is 10,000,000.

Figure 7.12: Result of 50% search operations and 50% insert operations on Sun Fire (m is 200).



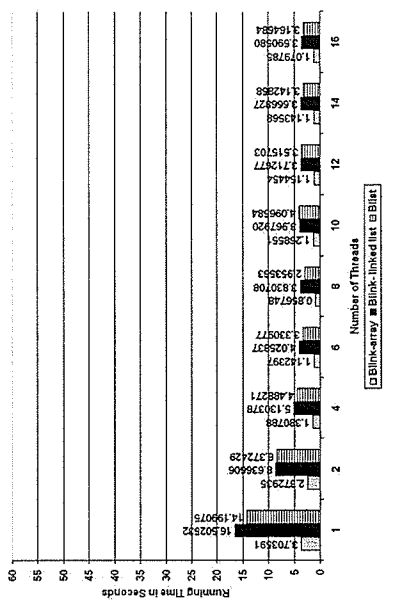
(a) Key range is 250,000.

(b) Key range is 1,000,000.

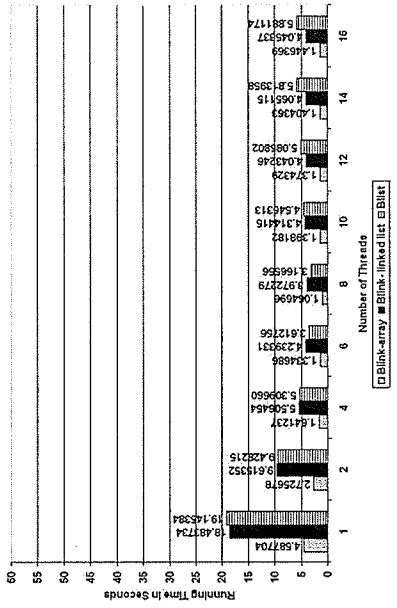
(c) Key range is 2,000,000.

(d) Key range is 10,000,000.

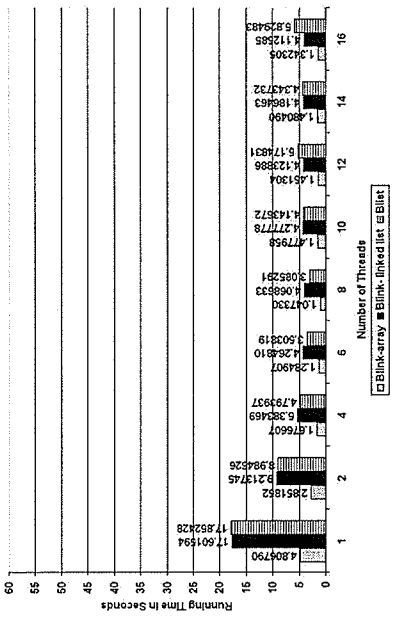
Figure 7.13: Result of 50% search operations and 50% insert operations on Sun Fire (m is 500).



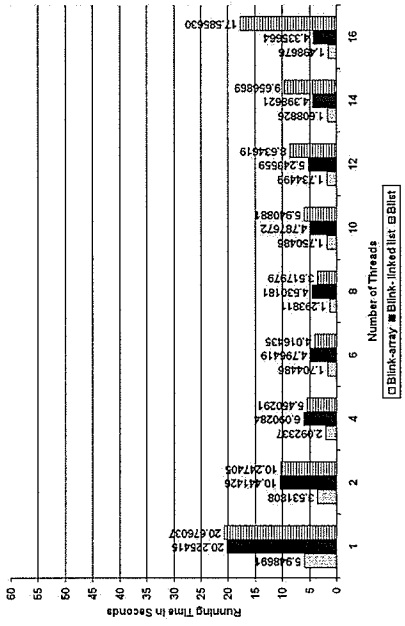
(a) Key range is 250,000.



(b) Key range is 1,000,000.

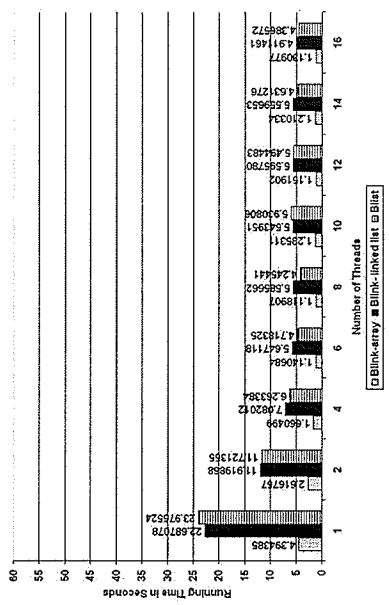


(c) Key range is 2,000,000.

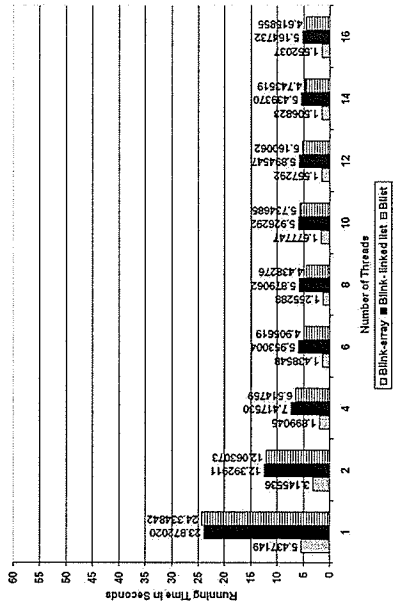


(d) Key range is 10,000,000.

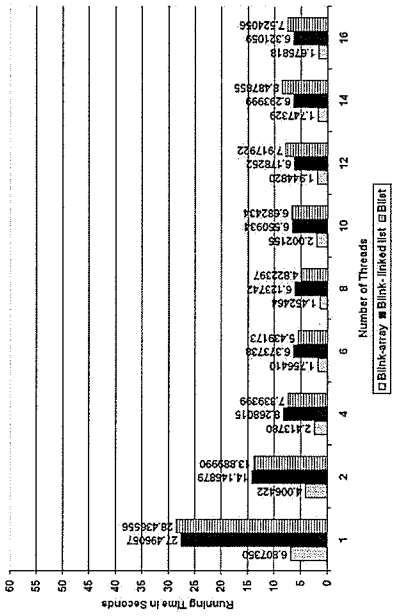
Figure 7.14: Result of 50% search operations and 50% insert operations on Sun Fire (m is 700).



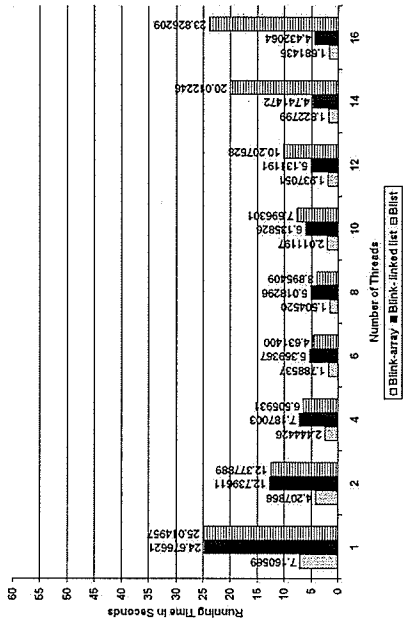
(a) Key range is 250,000.



(b) Key range is 1,000,000.



(c) Key range is 2,000,000.



(d) Key range is 10,000,000.

Figure 7.15: Result of 50% search operations and 50% insert operations on Sun Fire (m is 1000).

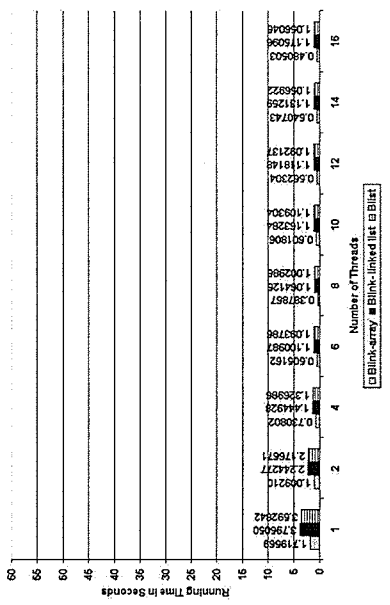
perform better than the B^{link}-linked-list trees. However, when the trees are bigger with a larger key range such as 1–1,000,000, the B^{link}-linked-list trees with a smaller m such as 100 or 200 perform significantly better than the B^{list}-trees (Figures 7.11b, and 7.12b). With the same key range, but with larger m such as 500 or 1000, the B^{list}-trees work more efficiently than the B^{link}-linked-list trees (Figures 7.13b, and 7.15b). When the key range is increased to 1–2,000,000, the B^{list}-trees are less efficient than the B^{link}-linked-list trees. The running time differences between these two types of trees become significantly higher when m is 100, 200, and 500 for such key ranges (Figures 7.11c, 7.12c, and 7.13c). The differences get smaller as m grows to 700 (Figure 7.14c), but when m becomes 1000, a slight increase in time difference is observed (Figure 7.15c). B^{list}-trees with a key range of 1–10,000,000 also do not perform as well as the B^{link}-linked-list trees do. The running time of the B^{list}-trees increases as m increases from 100 to 500 (from 28.597 seconds to 45.013 seconds, see Figures 7.11d, 7.12d, and 7.13d). But, when m hits 700, the running time becomes significantly lower (17.586 seconds, see Figure 7.14d). The running time again goes higher when m becomes 1000 (23.826 seconds, see Figure 7.15d). Additionally, when more than eight processes concurrently run on the Sun machines, the variance in the running time of the same program is higher than when eight or fewer processes run on the same machines. This is because the random seeds across replicated experiments differ so, in some runs some processes likely spent more time waiting during insert operations.

Experiment 4: 10% Search and 90% Insert Operations

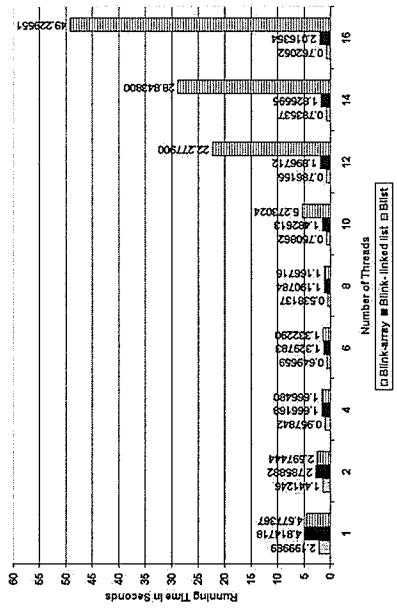
Experiment 4 consists of 10% concurrent search and 90% concurrent insert operations. This experiment allows even more chance of conflicts than experiment 3, hence, more locking delay for the B^{link}-array tree and the B^{link}-linked-list tree, and more chance of unsuccessful CAS operations and free cell allocation waiting time for the B^{list}-trees. Figures 7.16–7.20 show the results of experiment 4 on the Sun Fire machines with eight cores.

In this experiment again the B^{link}-array tree shows the best performance among the three kinds of trees. When comparing the B^{link}-linked-list trees and the B^{list}-trees, the latter performs better with eight or fewer processes running simultaneously.

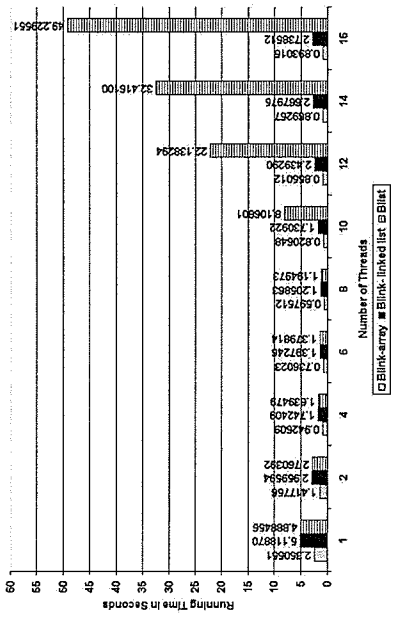
Like experiment 2 (90% search and 10% insert) and 3 (50% search and 50% insert), this experiment also shows some interesting results for the B^{list}-trees when more than eight processes run concurrently in eight processors. The size of the trees, and the number of empty cells in the nodes (or leaves) in the trees affect the performance of the B^{list}-trees. When the key range is small (1–250,000), the B^{list}-trees work more efficiently than the B^{link}-linked-list trees (Figures 7.16a, 7.17a, 7.18a, 7.19a, and 7.20a). Like experiments 2 and 3, when the keys are chosen from a bigger range (1–1,000,000 or 1–2,000,000), B^{list}-trees lose their efficiency when m is small or sometimes moderate (e.g., 100–500, see Figures 7.16b, 7.17b, 7.16c, 7.17c, and 7.18c). When m is 700 or more, then again the B^{list}-trees perform better than they do with a smaller m (Figures 7.19b, 7.19c, and 7.20b). The performance of B^{list}-trees gets worse again when the key range is chosen as large as 1–10 million (Figures 7.16d, 7.17d, 7.18d, 7.19d, and 7.20d).



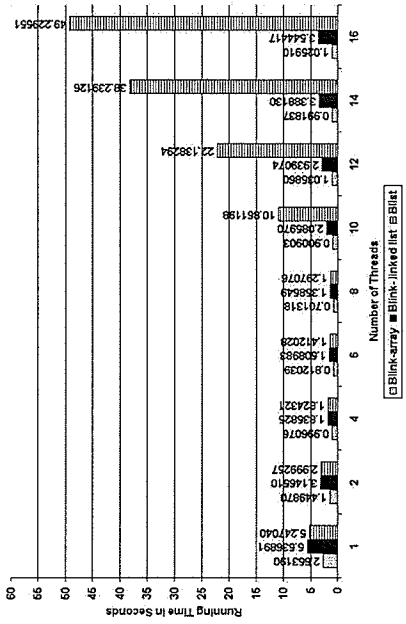
(a) Key range is 250,000.



(b) Key range is 1,000,000.

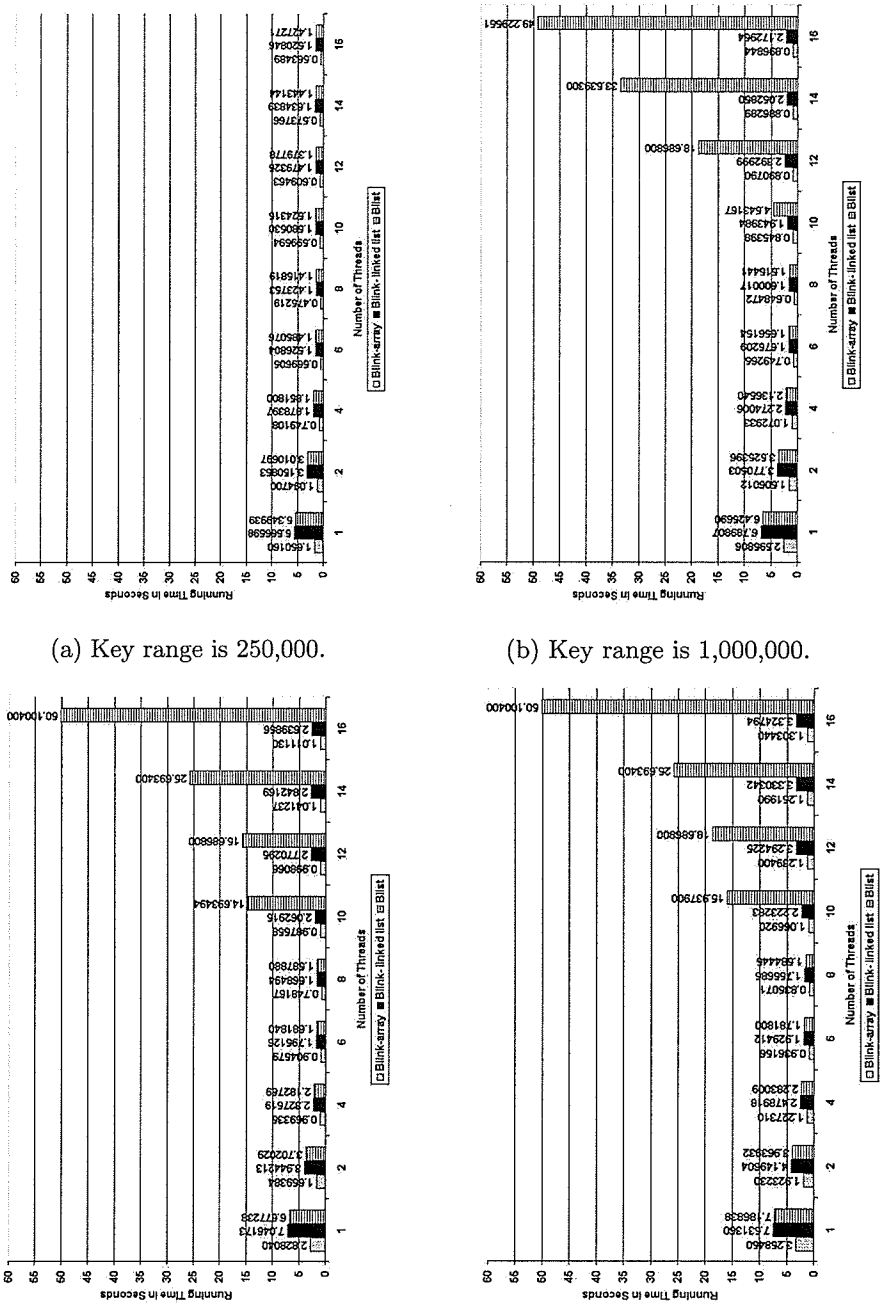


(c) Key range is 2,000,000.



(d) Key range is 10,000,000.

Figure 7.16: Result of 10% search operations and 90% insert operations on Sun Fire (m is 100).



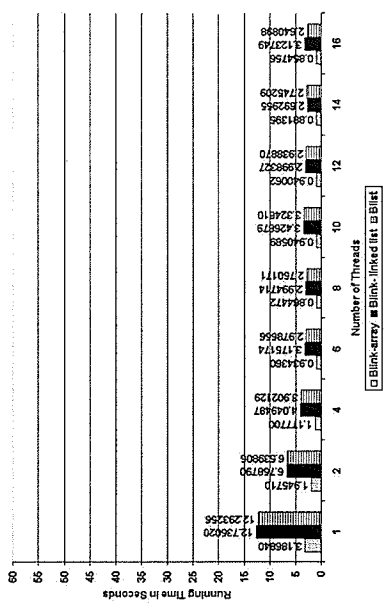
(a) Key range is 250,000.

(b) Key range is 1,000,000.

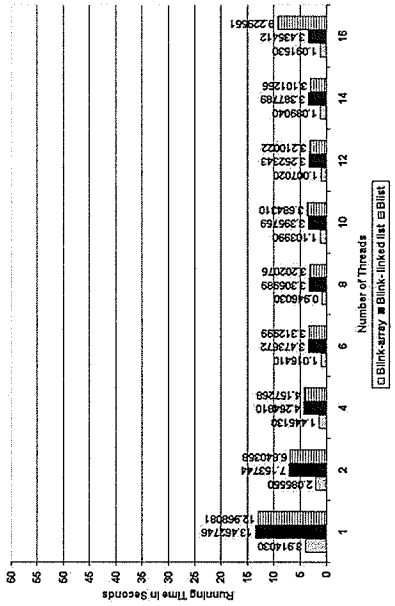
(c) Key range is 2,000,000.

(d) Key range is 10,000,000.

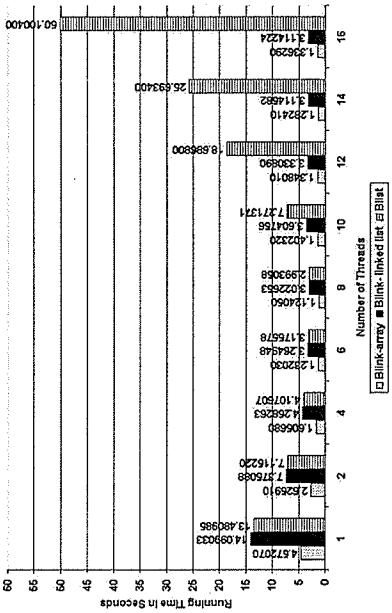
Figure 7.17: Result of 10% search operations and 90% insert operations on Sun Fire (m is 200).



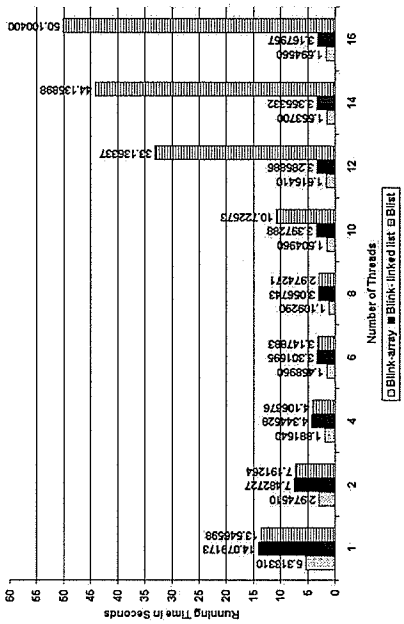
(a) Key range is 250,000.



(b) Key range is 1,000,000.

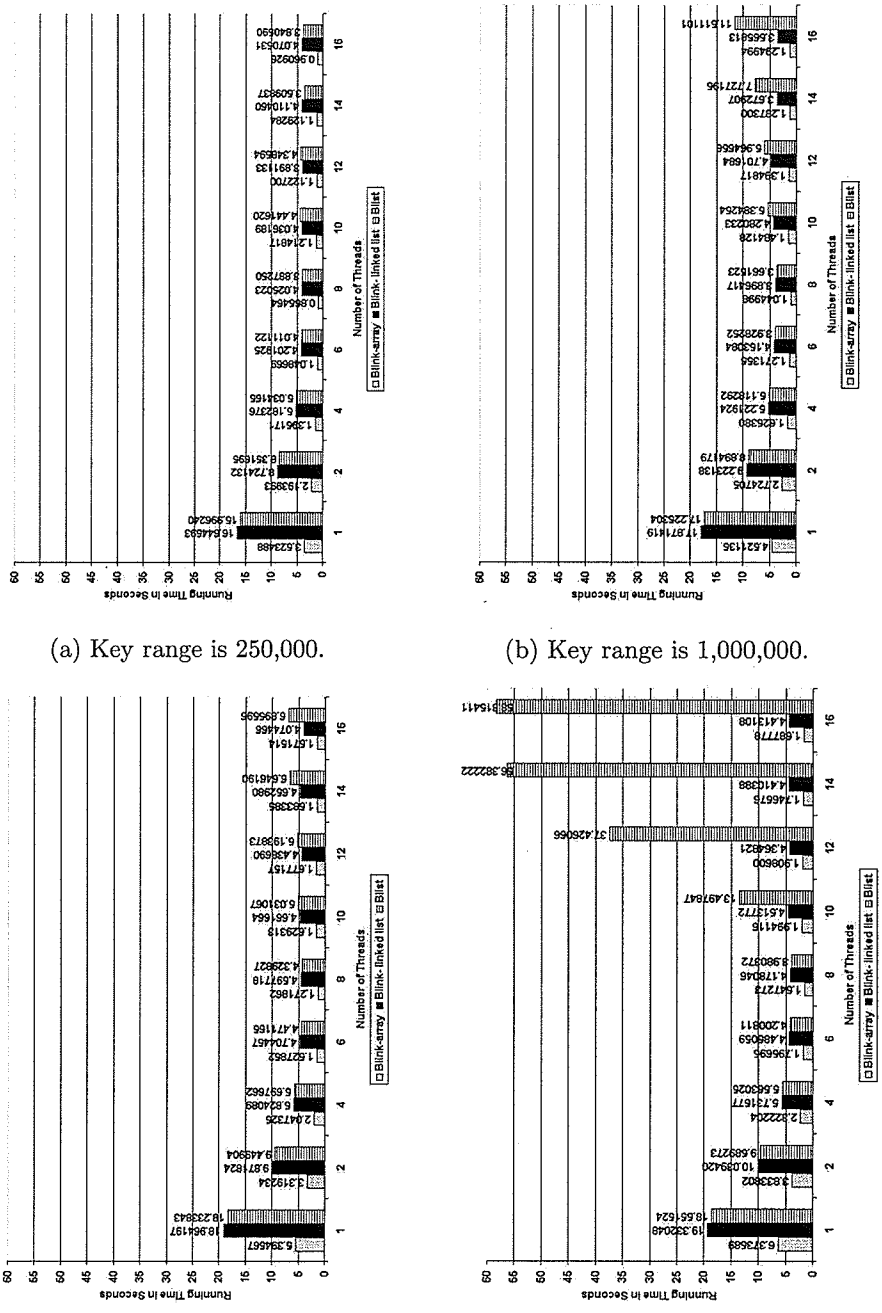


(c) Key range is 2,000,000.



(d) Key range is 10,000,000.

Figure 7.18: Result of 10% search operations and 90% insert operations on Sun Fire (m is 500).



(a) Key range is 250,000.

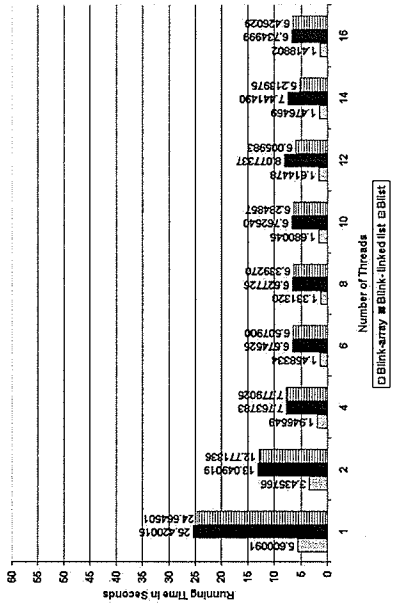
(b) Key range is 1,000,000.

(c) Key range is 2,000,000.

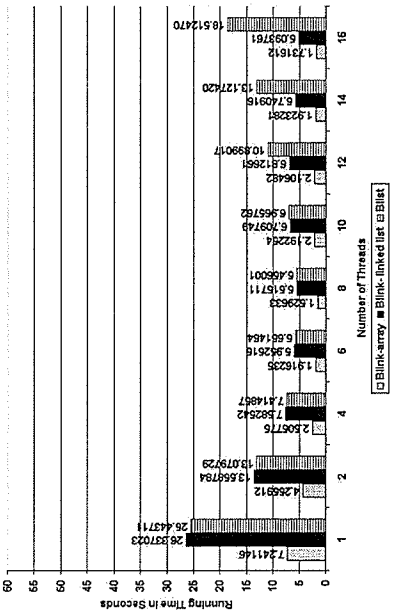
(d) Key range is 10,000,000.

Figure 7.19: Result of 10% search operations and 90% insert operations on Sun Fire (m is 700).

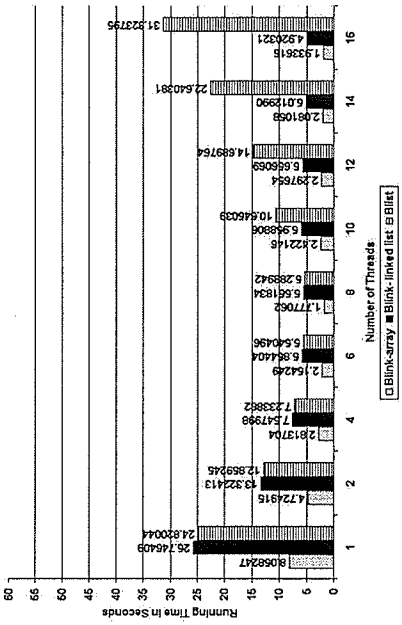
(a) Key range is 250,000.



(b) Key range is 1,000,000.



(c) Key range is 2,000,000.



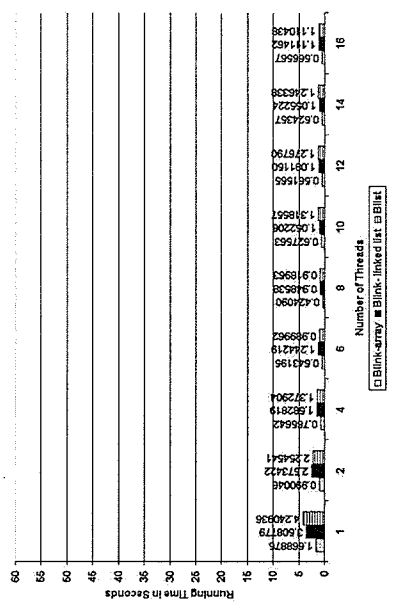
(d) Key range is 10,000,000.

Figure 7.20: Result of 10% search operations and 90% insert operations on Sun Fire (m is 1000).

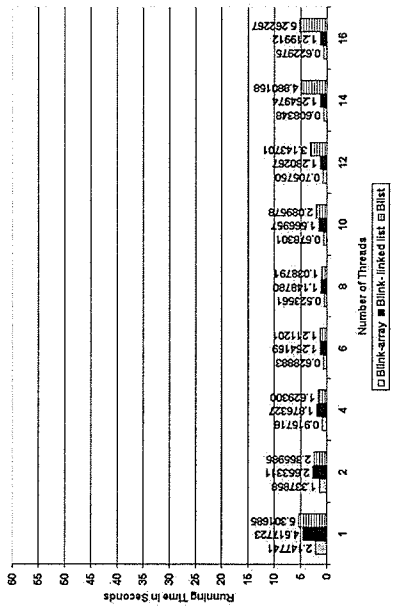
Experiment 5: Equal Mix of Search, Insert and Delete Operations

In my last experiment, I used 1 million concurrent search, insert, and delete operations in equal proportions. As this experiment includes deletion operations, the deletion processes in the B^{list}-trees had to retire deleted cells, and, before returning retired cells to the free list of a node, the deletion processes had to check whether the cell was currently being used by any other concurrent processes. The deletion processes in the B^{link}-array trees and the B^{link}-linked-list trees had to lock entire nodes that contained the keys to be deleted. Figures 7.21–7.25 show the results of this experiment.

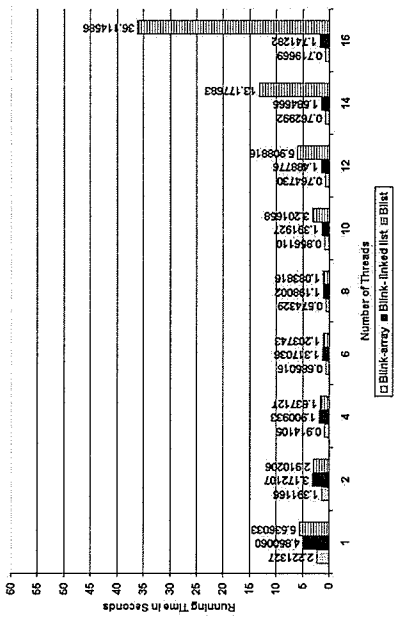
Like the previous experiments, in this experiment, the B^{link}-array tree performs the best among all the types of trees. Additionally, as expected from observing the results of experiments 1–4, B^{list}-trees have better performance than the B^{link}-linked-list trees when eight or fewer concurrent processes run on the Sun machines. When the number of processes increases, the better performance continues within a smaller key range like 1–250,000 (Figures 7.21a, 7.22a, 7.23a, 7.24a, and 7.25a). But, when the key range gets bigger, and m remains small, more cells are held by the concurrent processes, and there are unreturned deleted cells waiting for scanning in the private stacks of the concurrent processes. Therefore, processes are delayed longer waiting to get an empty cell from the free list of a node in the B^{list}-trees, and the delays result in a higher running time than the running time of the B^{link}-linked-list trees (Figures 7.21b, 7.22b, 7.21c, and 7.21c). When m gets larger, the chances of getting an empty cell from the free lists increases, therefore, the performance of the B^{list}-trees gets better than the performance of the B^{link}-linked-list trees (Figures 7.24b, 7.24c,



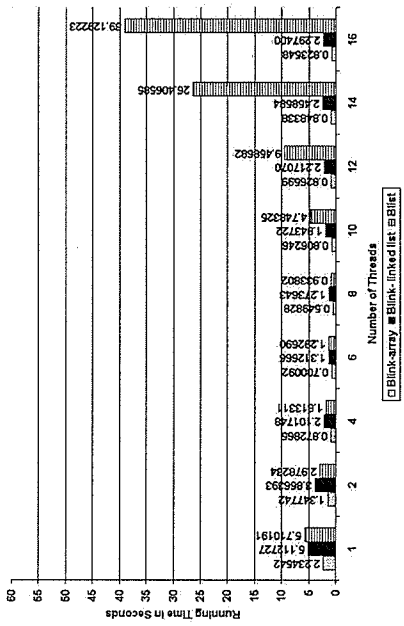
(a) Key range is 250,000.



(b) Key range is 1,000,000.

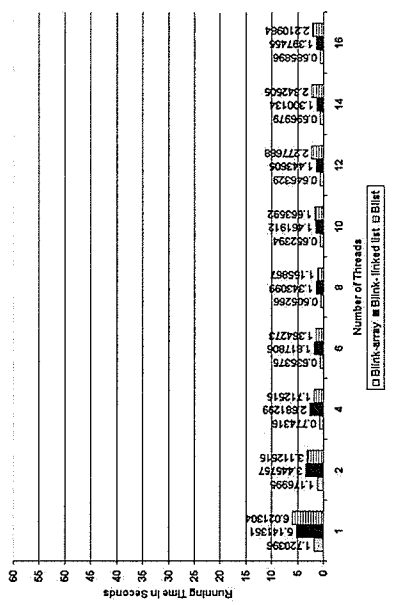


(c) Key range is 2,000,000.

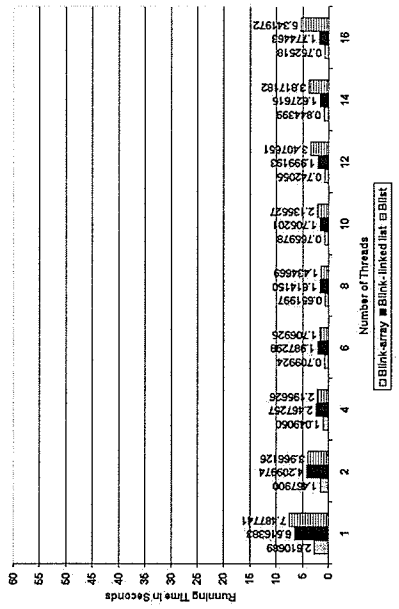


(d) Key range is 10,000,000.

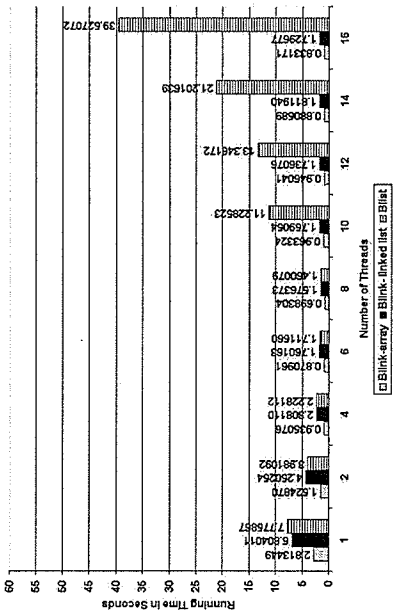
Figure 7.21: Result of equal mix of search, insert and delete operations on Sun Fire (m is 100).



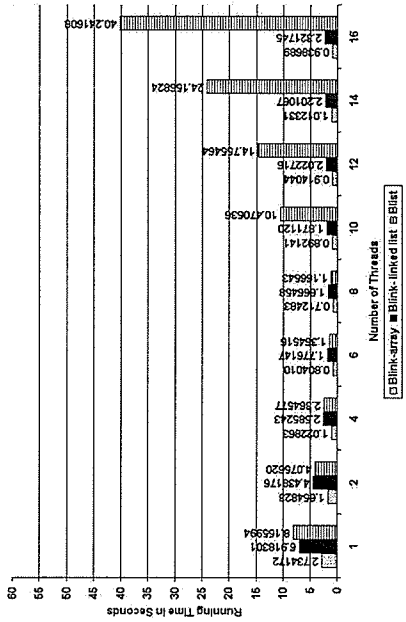
(a) Key range is 250,000.



(b) Key range is 1,000,000.

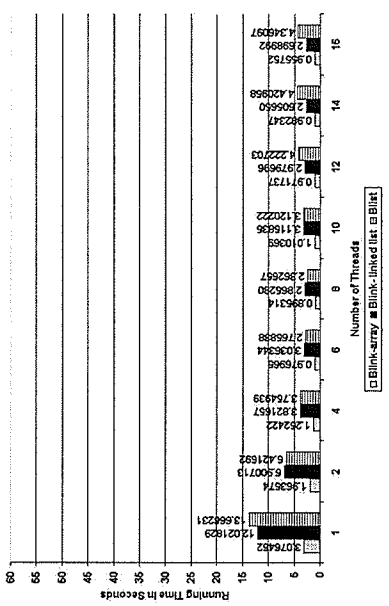


(c) Key range is 2,000,000.

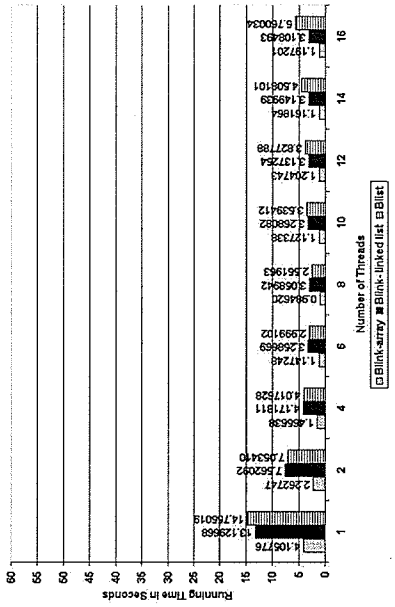


(d) Key range is 10,000,000.

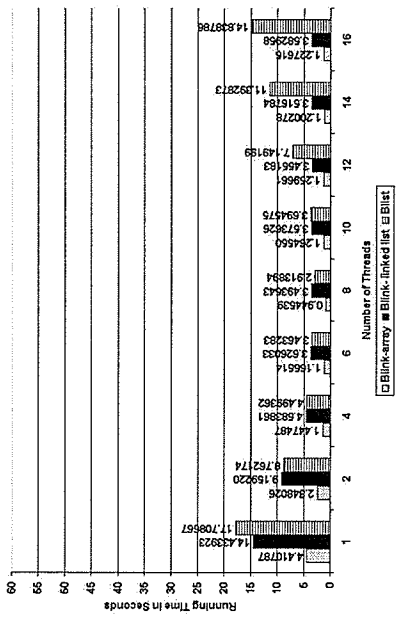
Figure 7.22: Result of equal mix of search, insert and delete operations on Sun Fire (m is 200).



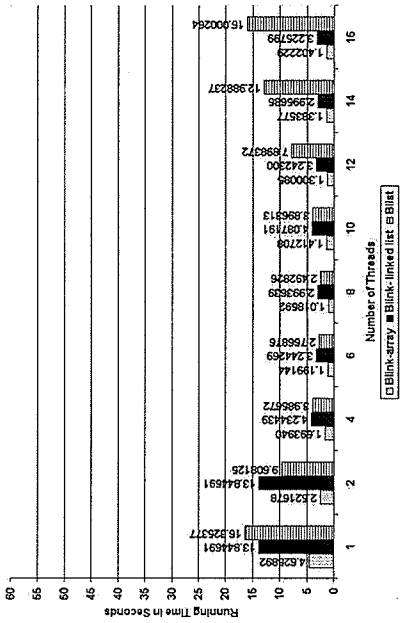
(a) Key range is 250,000.



(b) Key range is 1,000,000.

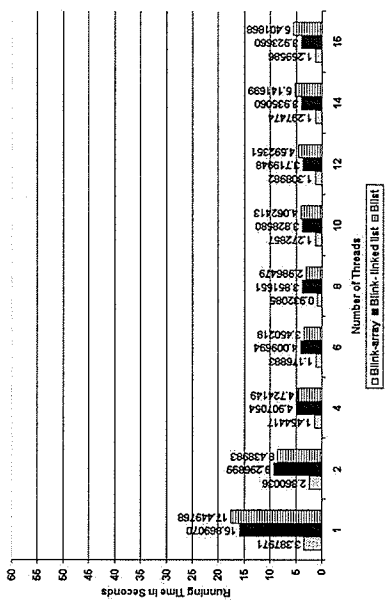


(c) Key range is 2,000,000.

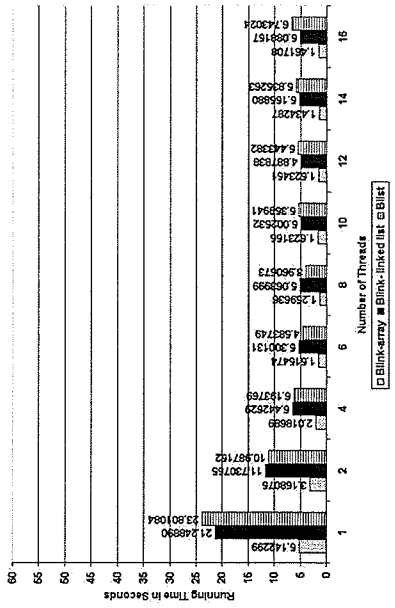


(d) Key range is 10,000,000.

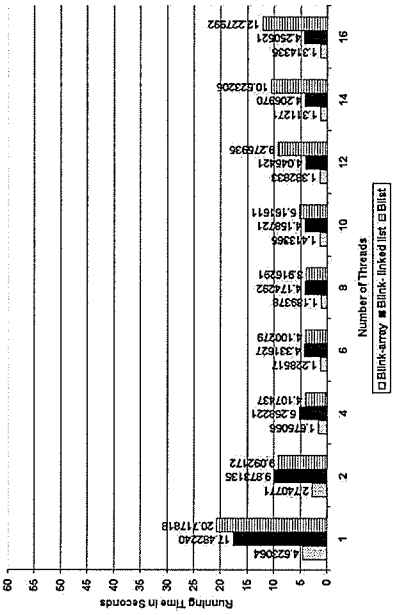
Figure 7.23: Result of equal mix of search, insert and delete operations on Sun Fire (m is 500).



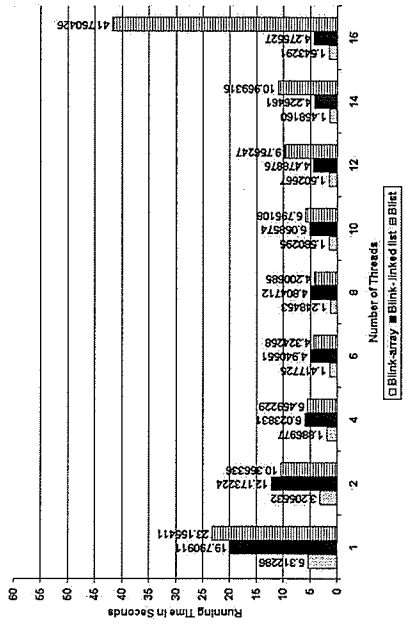
(a) Key range is 250,000.



(b) Key range is 1,000,000.

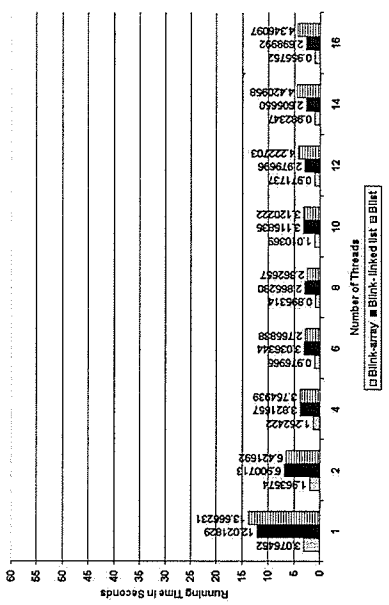


(c) Key range is 2,000,000.

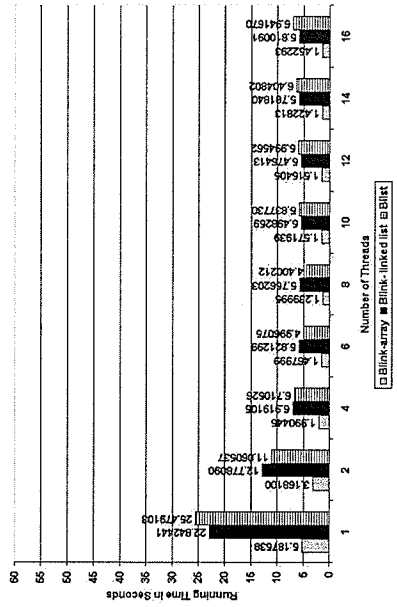


(d) Key range is 10,000,000.

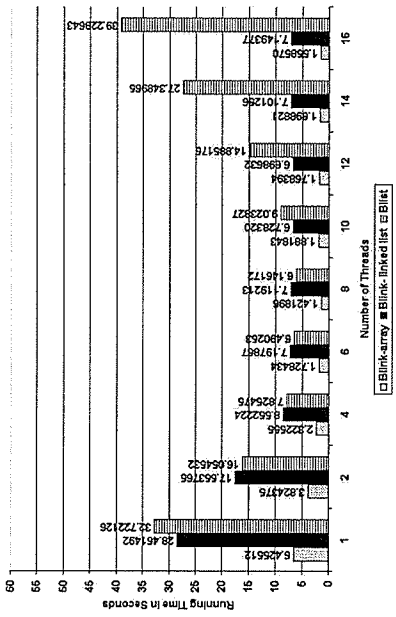
Figure 7.24: Result of equal mix of search, insert and delete operations on Sun Fire (m is 700).



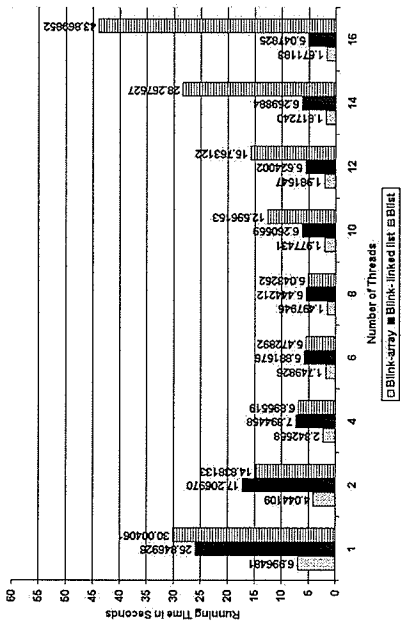
(a) Key range is 250,000.



(b) Key range is 1,000,000.



(c) Key range is 2,000,000.



(d) Key range is 10,000,000.

Figure 7.25: Result of equal mix of search, insert and delete operations on Sun Fire (m is 1000).

7.25b, and 7.25c). However, when the key range is larger (e.g., 1–10,000,000), the B^{list}-tree performance decreases again (Figures 7.21d, 7.22d, 7.23d, 7.24d, and 7.25d).

Summary of the Experiments on the Sun Fire Machines

After conducting my experiments, I have observed that the B^{link}-array tree always performs the best compared to the B^{link}-linked-list and B^{list}-tree. Furthermore, depending on the values of my tested parameters, the B^{list}-tree has noticeable to significant performance gain compared to the B^{link}-linked-list trees. I have summarized the results from experiment 2 to experiment 5 in table 7.1. I did not include the results of the experiment with 100% search operations, since no lock-free or lock-based concurrency-control techniques were used in this experiment. Furthermore, with smaller m like 100 or 200, there were not much performance gain in the B^{list}-trees. Therefore, I did not include the summary of the running time, when m is 100 or 200.

From the experimental results on the Sun machines, it can be concluded that, regardless of the operation type, value of m, and key range, the B^{link}-array trees perform the best compared to the B^{link}-linked-list trees and the B^{list}-trees. This result is expected because of the structural difference between the nodes among the array-based B^{link}-array trees, and the localized linked-list-based B^{link}-linked-list trees, and B^{list}-trees (as described in Section 7.1).

When I compare the lock-based B^{link}-linked-list trees and the B^{list}-trees, there are no structural differences between these two types of tree. The only difference between them is the concurrency-control techniques used—more precisely, the differ-

Table 7.1: Percentage of performance gain in B^{list}-trees compared to the B^{link}-linked-list trees with eight concurrent processes.

m	Key Range	Experiment 2	Experiment 3	Experiment 4	Experiment 5
500	1–250,000	25.09	21.93	8.17	17.54
	1–1,000,000	25.11	23.51	3.14	16.57
	1–2,000,000	25.96	20.39	0.98	16.59
	1–10,000,000	25.65	22.45	2.79	16.73
700	1–250,000	0.01	22.90	3.42	22.46
	1–1,000,000	0.14	20.28	6.03	21.79
	1–2,000,000	0.64	24.17	5.83	6.18
	1–10,000,000	1.44	22.34	4.73	12.57
1000	1–250,000	0.64	23.99	2.53	22.90
	1–1,000,000	0.65	24.51	4.35	23.56
	1–2,000,000	0.67	21.25	1.08	13.67
	1–10,000,000	0.02	22.38	4.74	7.36

ence between lock-based techniques and the lock-free techniques used in the update operations.

In the first experiment, when only search operations were conducted on both types of trees, none of the trees use any concurrency-control techniques since there were no updates on the trees. The results of this experiment show that when there is no conflict in the trees, the B^{list}-tree algorithms perform slightly better than the B^{link}-linked-list tree algorithms.

When update operations were introduced in both types of trees (experiments 2–5), the B^{link}-linked-list tree algorithms had to acquire locks to handle concurrency among concurrent update operations, whereas the B^{list}-tree algorithm used lock-free CAS as a concurrency-control technique to enable concurrent update operations. In these experiments where there was no thread-switching overhead (the experiments conducted with eight or fewer threads), the B^{list}-trees always performed better than the B^{link}-linked-list trees. This means that, in these experiments, the lock-free concurrency-control techniques outperformed the lock-based concurrency-control techniques. Because of the lock-based properties of the B^{link}-linked-list trees, a process acquires the lock of a node for an update operation when the process finds the appropriate node to be updated. Once the node is locked, the rest of the processes must wait (typically be blocked) if they want to update the same node. Conversely, in lock-free B^{list}-trees, there is no need of such waiting. No matter how high the data contention is in a node, all processes can simultaneously work on the same node, if necessary. A process in a B^{list}-tree needs to re-do its computation only if more than one process wants to update the same cell in that node. As long as the concurrent operations do not manipulate the same cell, greater parallelism is achieved. Therefore, as we see the performance analysis of experiment 2–5, the ratio of the running time of the B^{link}-linked-list trees and the B^{list}-trees goes higher when m grows larger. The ratio increases because, when m is 500 or more, the overhead to acquire the lock of a node becomes higher (in the B^{link}-linked-list trees) than the overhead of re-doing the work of updating a cell in the node (in the B^{list}-trees). From the analysis of the results, it can be concluded that when m is relatively larger, the lock-free B^{list}-trees perform

noticeably better than the lock-based B^{link}-linked-list trees.

7.2.2 SGI Machines

In my experiments on the Sun Fire machines, up to sixteen processes performed their operations on eight processors. Therefore, when there were more than eight processes used on the Sun Fire machines simultaneously, more than one process had to share the same processor. As a result of thread-sharing by the same processors, the running time for the experiments with update operations (experiments 2–5 in Section 7.2.1) on B^{list}-trees were sometimes significantly higher than the running time for the B^{link}-array and the B^{link}-linked-list tree algorithms. To see if this was a problem specific to the Sun Fire machines, I did some of my experiments on the SGI machines hosted by WestGrid.

The SGI machines are also NUMA-style Symmetric Multiprocessors (SMP) machines. In my experiments on the SGI machines, I recorded the average running times of ten runs for the B^{link}-array, B^{link}-linked-list, and B^{list}-trees with 1, 2, 4, 8, 16, and 32 parallel processors and with m values of 100, 500, and 1000. With no thread sharing in the SGI machines, the running time of the B^{list}-trees was never greater than the running time of the B^{link}-linked-list trees, suggesting a limitation in the memory system of the Sun Fire machines.

Like the experiments on the Sun Fire machines, in the experiments on the SGI machines, I also populated the tree with one million random keys from different key ranges (as in Section 7.2.1), and then I processes different mixes of search, insert and delete operations on the trees. I tested with 100% search operations (experiment 1 of

Section 7.2.1), 50% search and 50% insert operations (experiment 3 of Section 7.2.1) and equal mix of search, insert and delete operations (experiment 5 of Section 7.2.1) on the SGI machines.

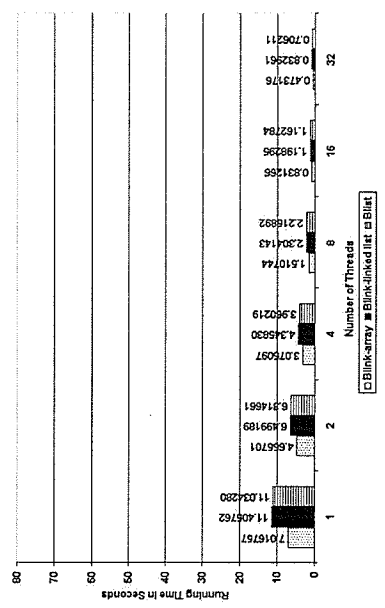
Experiment 1: 100% Search Operations

In my first experiment on the SGI machines, I submitted 100% search operations against the B^{link}-array, B^{link}-linked-list and B^{list}-trees. The search operations only read the keys from the nodes, and conclude whether the key to be searched for exists in the tree or not. Accordingly, there is no data contention within a node of the trees. Like the results for the corresponding experiment on the Sun Fire machines, the B^{link}-array tree performs best when there are only parallel search operations. Furthermore, the B^{list}-tree works more efficiently than the B^{link}-linked-list tree. Figures 7.26–7.28 show the running time comparison for 100% search operations on B^{link}-array, B^{link}-liked-list, and B^{list}-trees on the SGI machines.

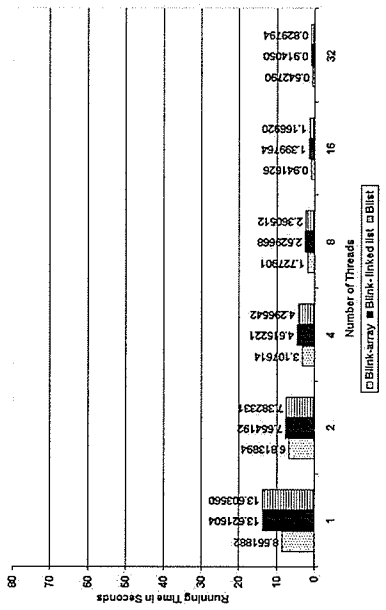
Experiment 2: 50% Search and 50% Insert Operations

In my second experiment with the SGI machines, I calculated the running times of B^{link}-array, B^{link}-linked-list and B^{list}-trees for a mix of 50% search and 50% insert operations. Since there are insert operations in this experiment, updates are observed within the nodes of the trees, hence concurrency-control techniques are used. Locks are used in the B^{link}-array and B^{link}-linked-list trees and CAS is used in the B^{list}-tree. Figures 7.29–7.31 show the results for 50% search and 50% insert operations on B^{link}-array, B^{link}-linked-list and B^{list}-trees on the SGI machines.

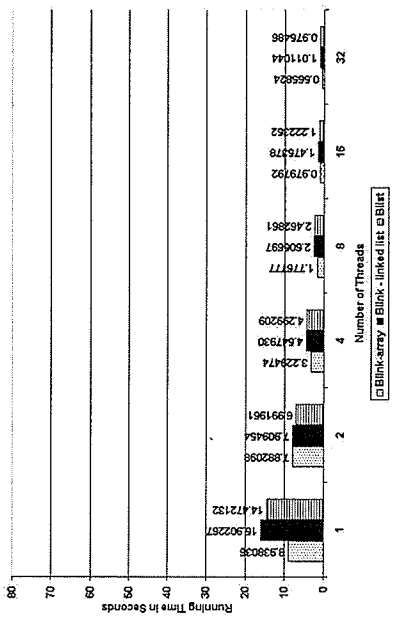
On the SGI machines, the results differ from the corresponding experiment on the



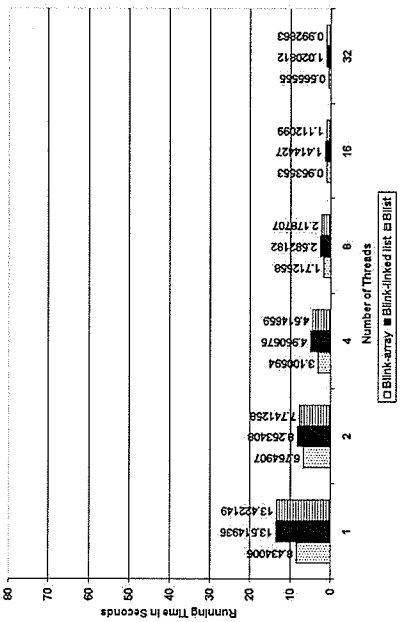
(a) Key range is 250,000.



(b) Key range is 1,000,000.

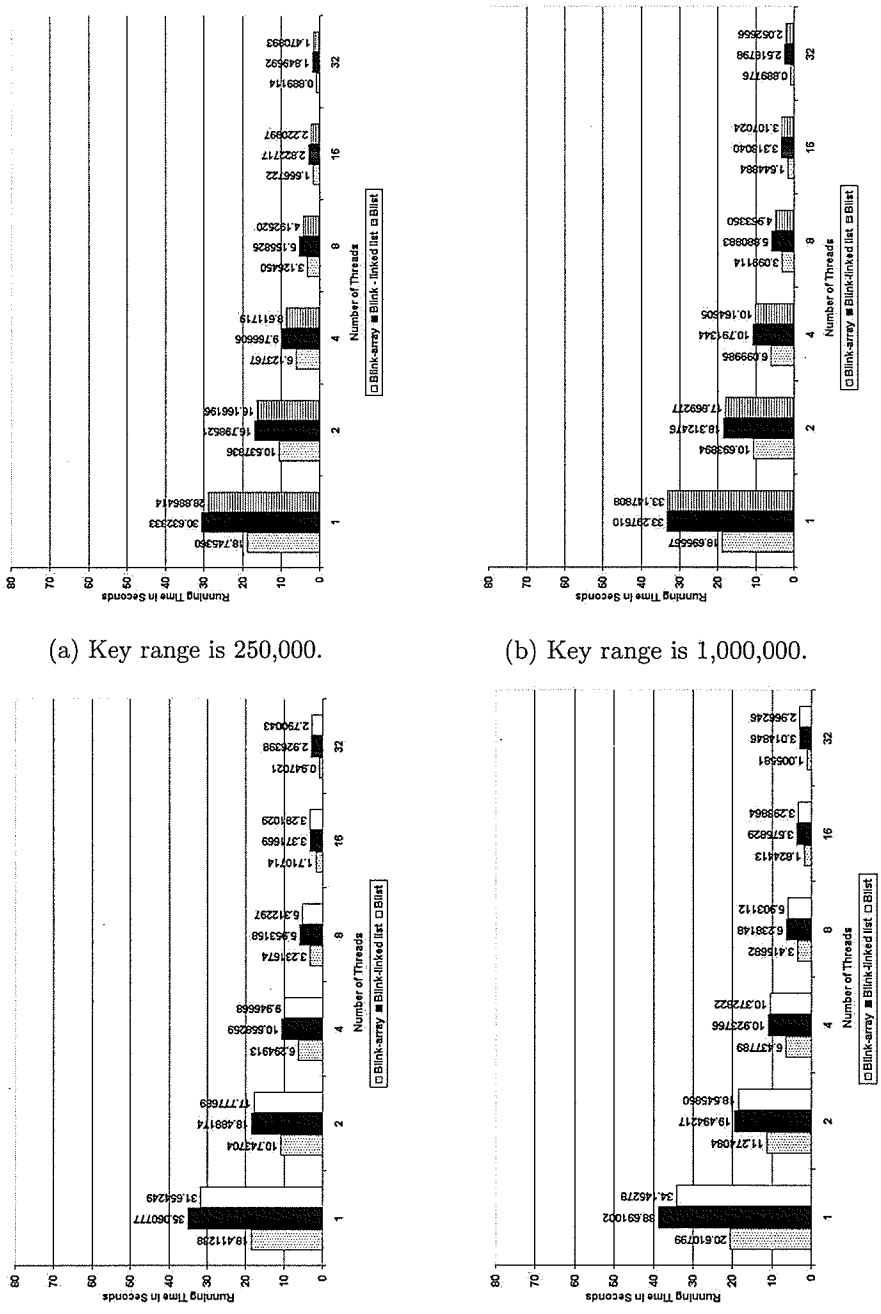


(c) Key range is 2,000,000.



(d) Key range is 10,000,000.

Figure 7.26: Result of 100% search operations on SGI (m is 100).



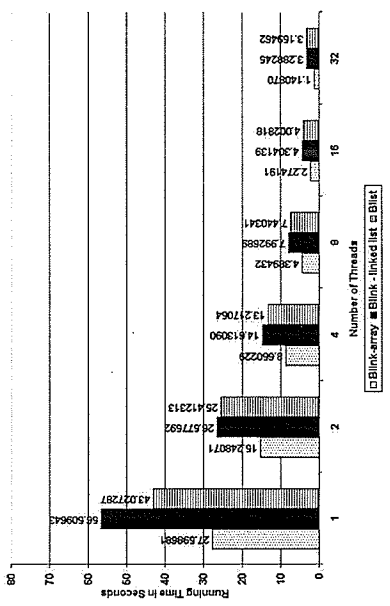
(a) Key range is 250,000.

(b) Key range is 1,000,000.

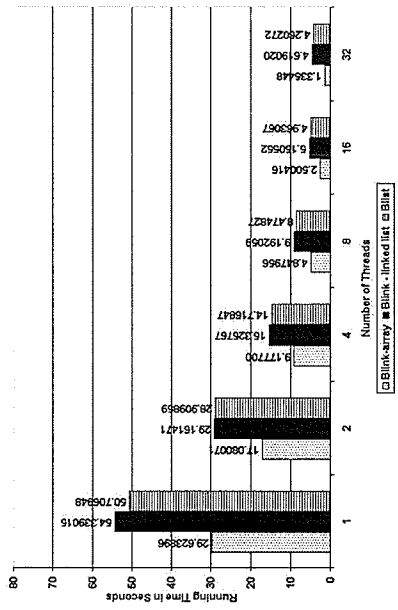
(c) Key range is 2,000,000.

(d) Key range is 10,000,000.

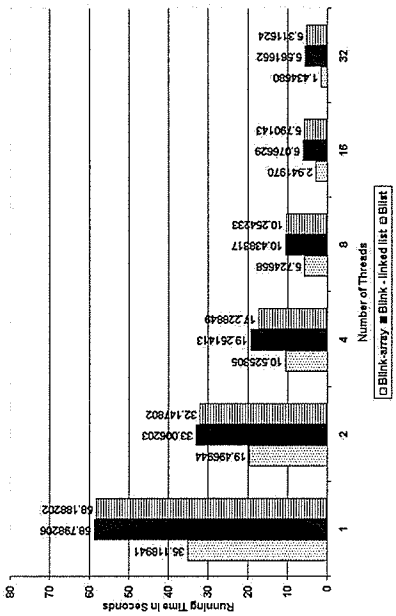
Figure 7.27: Result of 100% search operations on SGI (m is 500).



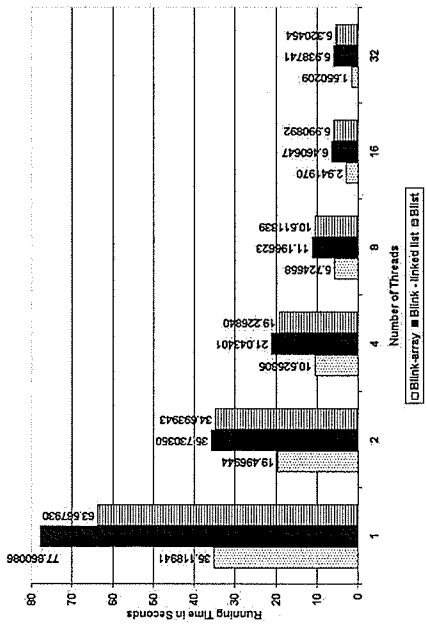
(a) Key range is 250,000.



(b) Key range is 1,000,000.



(c) Key range is 2,000,000.



(d) Key range is 10,000,000.

Figure 7.28: Result of 100% search operations on SGI (m is 1000).

Sun Fire machines. On the Sun Fire machines, regardless of the parameters of the experiment, the B^{link}-array tree always performed the best. Furthermore, with more than eight processes, most of the time, the B^{link}-linked-list tree performed better than the B^{list}-tree (see Experiment 3 of Section 7.2.1). However, the results for the corresponding experiment on the SGI machines show that my B^{list}-tree always outperforms the B^{link}-linked-list tree. Moreover, when m is moderate to large (say, 500 to 1000), the B^{list}-tree performs significantly better than the B^{link}-array tree when more than one concurrent processes are running on the machines (see Figures 7.30–7.31).

The SGI machines have more advanced memory systems and the OS applies more intelligent initial load balancing algorithms than the Sun Fire machines. Furthermore, for the experiments corresponding to those on the Sun Fire machines with more than eight processors, there is no thread sharing in the experiments on the SGI machines. On the SGI machines, even with the more complex node structures of the B^{list}-tree (compared to the node structure of B^{link}-array tree) the lock-free B^{list}-tree out performs the lock-based B^{link}-array trees. Since the B^{link}-tree (both array-based and linked-list-based) algorithms must lock entire nodes, when m grows larger (500 or more) the waiting time to acquire the lock for a node becomes higher, which ultimately causes a larger running times for both the B^{link}-array and the B^{link}-linked-list trees. On the other hand, since in a B^{list}-tree, more than one process can concurrently update a node, with larger m, the probability of getting a free cell from a node's `free-list` becomes higher, and eventually minimizes the waiting time of an update process. Therefore, B^{list}-trees with a larger branching factor (m) show more efficiency

than B^{link}-array and B^{link}-linked-list trees with the same branching factor.

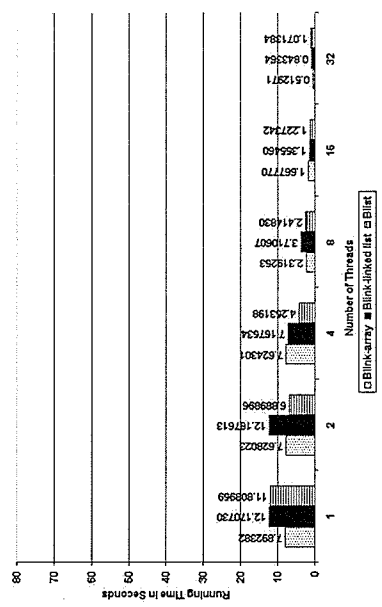
Experiment 3: Equal Mix of Search, Insert and Delete Operations

My last experiment conducted on the SGI machines was with an equal mix of search, insert and delete operations. Like the previous experiments on the SGI machines, I expected to show the results of this experiment on B^{link}-array, B^{link}-linked-list and B^{list}-trees with all parameters. However, due to the long waiting time to access the SGI machines, I could not complete all results for this experiment. My partial results are used as the basis for the following discussion.

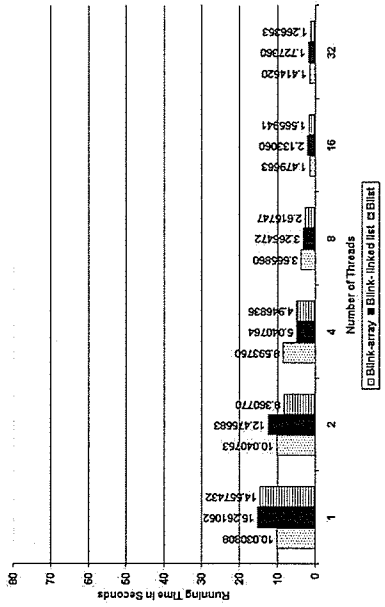
Unlike the experiment with 50% search and 50% insert operations, the subset of my results for this experiment suggest that the B^{link}-array tree shows more efficiency among all B-tree variants considered. Furthermore, the scan operations in the deletion algorithm for B^{list}-trees cause long delays to reclaim the free cells to the node's `free-list`. These long delays sometimes makes the B^{list}-tree less efficient than the B^{link}-linked-list trees, especially when m is smaller (e.g., 100). When m grows larger (e.g., 500 or more), the efficiency of the B^{list}-tree improves compared to the B^{link}-linked-list trees.

Summary of the Experiments on the SGI Machines

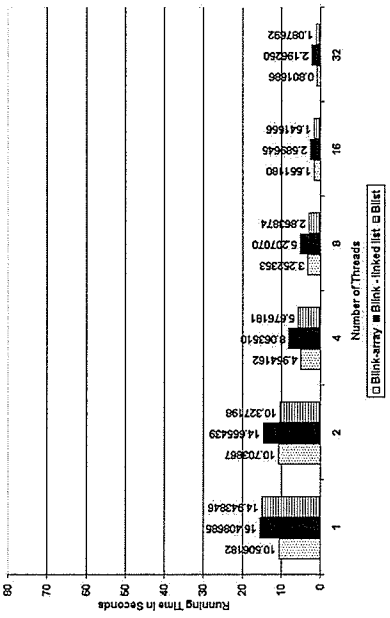
With the experiments on the SGI machines, I have observed due to the cache-coherent nature of the NUMA-style parallel architecture in the SGI machines, sometime my B^{list}-tree outperforms the B^{link}-array tree. With 50% search and 50% insert operations, the performance gain is observed highes. I have summarized the result of the performance gain in B^{list}-tree compared to the B^{link}-array and B^{link}–linked-list



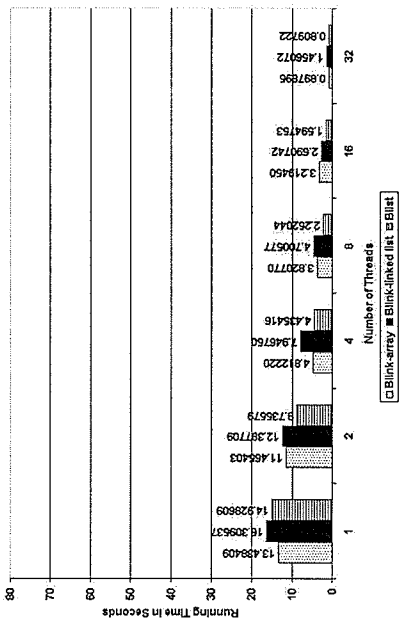
(a) Key range is 250,000.



(b) Key range is 1,000,000.

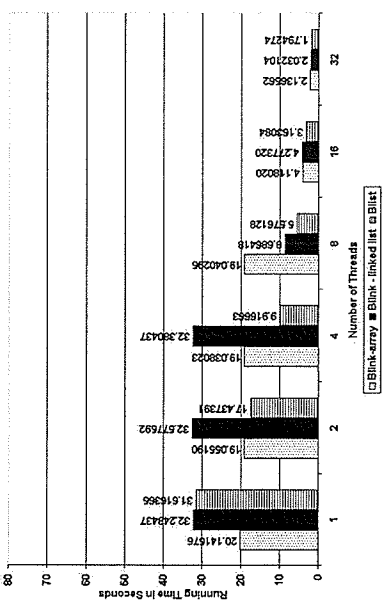


(c) Key range is 2,000,000.

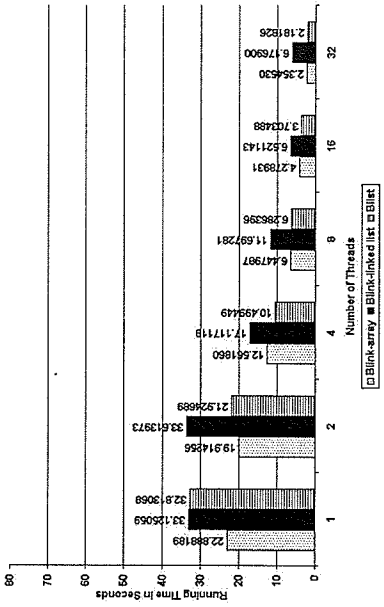


(d) Key range is 10,000,000.

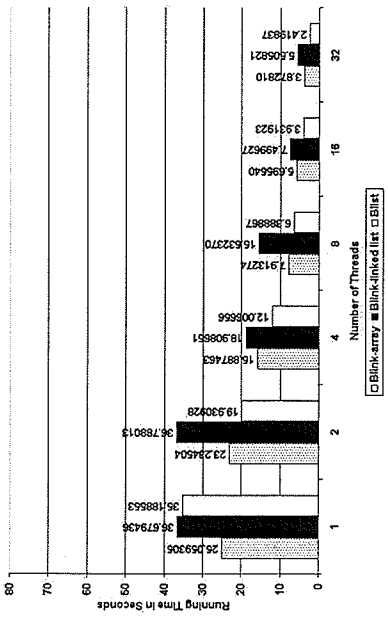
Figure 7.29: Result of 50% search operations and 50% insert operations on SGI (m is 100).



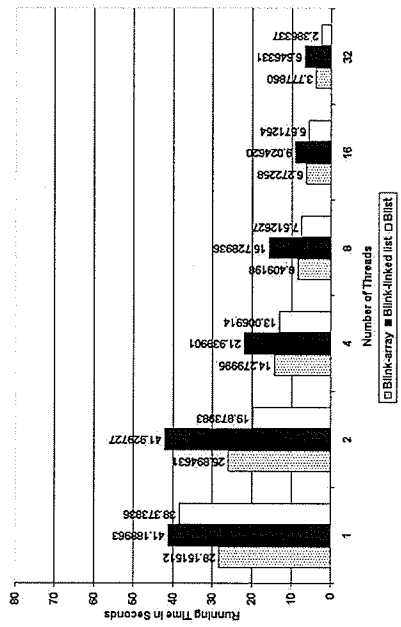
(a) Key range is 250,000.



(b) Key range is 1,000,000.

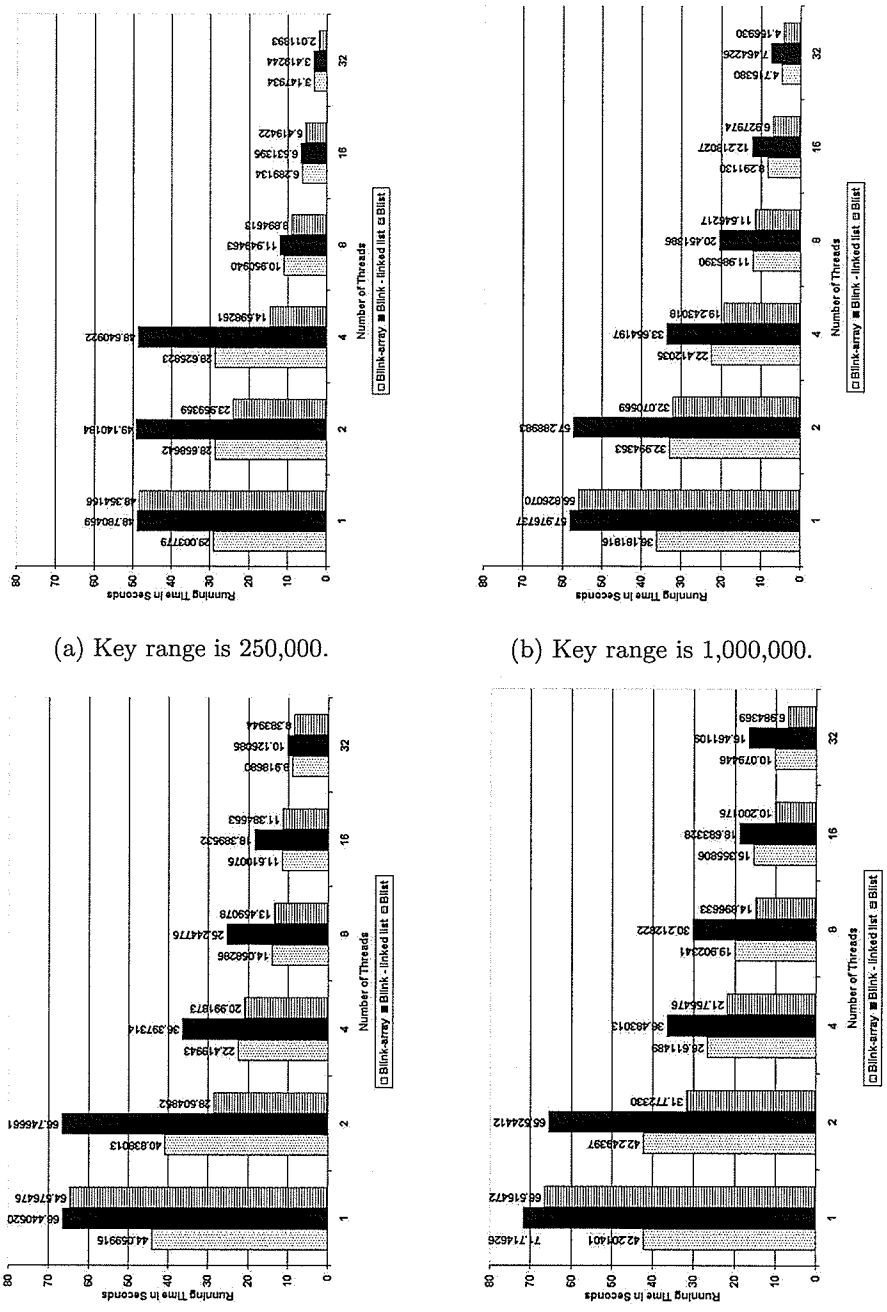


(c) Key range is 2,000,000.



(d) Key range is 10,000,000.

Figure 7.30: Result of 50% search operations and 50% insert operations on SGI (m is 500).



(a) Key range is 250,000.

(b) Key range is 1,000,000.

(c) Key range is 2,000,000.

(d) Key range is 10,000,000.

Figure 7.31: Result of 50% search operations and 50% insert operations on SGI (m is 1000).

Table 7.2: Percentage of performance gain in B^{list}-trees compared to the B^{link}-array B^{link}-linked-list trees with 50% search and 50% insert operations on 32 concurrent processes.

m	**Key Range**	**B^{link}-array tree VS B^{list}-trees**	**B^{link}-linked-list tree VS B^{list}-trees**
100	1–250,000	0	0
	1–1,000,000	10.47	26.69
	1–2,000,000	0	50.48
	1–10,000,000	9.82	44.39
500	1–250,000	16.02	11.706
	1–1,000,000	7.33	64.68
	1–2,000,000	37.52	56.05
	1–10,000,000	36.83	64.10
1000	1–250,000	36.09	41.16
	1–1,000,000	11.84	44.31
	1–2,000,000	6.00	17.20
	1–10,000,000	30.71	57.57

tree in table 7.2.

Chapter 8

Conclusion and Future Work

8.1 Conclusions

B-trees are ideal for large-scale searching since their search, insert and delete operations take only logarithmic time. If a B-tree can be used concurrently by many users, the efficiency of the system increases. There have been many lock-based algorithms designed for concurrent B-trees on disk. These algorithms lock some portion of the tree to allow some concurrency while maintaining consistency. However, these lock-based algorithms may offer limited concurrency in certain cases and have negative side-effects such as deadlock, convoying and priority inversion.

In this thesis, I have presented algorithms for a B^{list}-tree that attempt to provide more concurrency for in-memory applications using lock-free techniques, applying Michael's hazard-pointer techniques to efficiently manage intra-node updates in such a way that memory locality is maintained. Furthermore, after designing the algorithms, I have assessed the performance relative to two variants of Lehman and Yao's B^{link}-

tree (where one variant stores the keys in an array and another variant stores the keys in a linked list). I performed all of my assessments on the NUMA-style Sun Fire and SGI machines. The results of my experiments show that B^{link}-trees with simple array-based key lists perform better than both B^{link} and B^{list}-trees with complex linked-list-based key list in their nodes. The lock-free concurrency-control techniques achieve better performance when the node structure of the trees being compared is similar (i.e., object-oriented linked-list-based key list structure), and when thread-switching overhead is comparable. However, some of the experimental results on the SGI machines show that lock-free B^{list}-tree concurrent insertions are more efficient than insertions in both types of lock-based B^{link}-trees.

My contributions in this thesis include:

- introduce a lock-free locality-of-reference-oriented linked list to store keys in the nodes of B^{list}-trees.
- implement atomic CAS operations for both Sun Fire and SGI machines to perform an update operation.
- introduce a split bit to each node to avoid unsafe concurrent update and split operations.
- introduce a cell allocation in the beginning of an update operation to avoid unsafe concurrent update and split operations.
- introduce a finally-linked-in-tree bit to each node to allow search processes to search in a newly-created node that is not currently accessible from the parent node.

- design a please-scan array to send requests to other concurrent processes to release some free cells for reuse.
- revise the scan algorithm to return the retired cells to their associated nodes for reuse.
- identify a threshold level R as $m/5$ to trigger a scan operation.
- present a comparative assessment on B^{link} and B^{list}-trees experimenting with different parameters.

8.2 Future Work

While doing my research, I have realized that some additional work could be done on my B^{list}-tree in the future to improve its performance. A lock-free array-based B^{list}-tree on a non object-oriented key-list implementation could be designed, maintaining more empty cells in the nodes might limit delays during updates, avoiding unnecessary rollbacks due to concurrent deletions could improve experiment results, and careful work assignment on the processors might improve locality.

Lehman and Yao's array-based B^{link}-trees give the best performance compared to the linked-list-based B^{link}-trees and lock-free B^{list}-trees. The complex linked-list-based node structure has some overhead (as described in Section 7.1) on intra-node operations (i.e., search, insert and delete operations within a node) and causes delay. As short-term future work, B^{list}-trees with lock-free arrays and without using objects could be built, and their performance could be assessed.

Another possibility for short-term future work might be to add extra free-space in each node of the B^{list}-trees. In the experiments on the Sun Fire machines, when more than one thread share the same core, if the nodes are nearly full, the processes require more time to complete the update operations. In my B^{list}-trees, I designed the nodes to have exactly m cells to store the keys. With high data contention, the free list of a node might be empty and any process that wants to do an update operation on the node would have to wait until there is a free cell in the node's free list. If a node is split after the node is 75% full, instead of 100% full, the probability of getting an empty cell will increase. Since memory is cheap, this might be an attractive option. The threshold level for node splitting might also be altered to see the effect of the fullness of nodes on overall performance.

On SGI machines, in particular, performance degradation due to deletes was obvious. This performance degradation was likely due to processes being rolled back when they find a cell that is marked for deletion. To minimize the overhead of such rollbacks, we might maintain pointers to a number of preceding cells while searching. Rather than rolling back to the beginning of the key list, we could then restart our search from a nearby safe position. This future work would be medium term.

In NUMA-style machines, updating a piece of memory is most efficient when the update is performed by the processor associated with the memory segment containing the piece of memory to be updated. Therefore, when a node of a B^{list}-tree needs to be updated, the work could be assigned to the process associated with the piece of memory where the node resides. It would be very interesting to see whether the overhead of this work assignment could be made small enough to benefit from

improved performance due to better locality. This would be a long-term future work.

I hope that the future work I have suggested will improve the performance of the lock-free B^{list}-tree, and it will work more efficiently than the lock-based B^{link}-tree with array-based key lists, under any circumstances.

Bibliography

[1] `http://old.westgrid.ca/support/nexus`.

[2] `http://www.westgrid.ca`.

[3] R. Bayer and E. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1(3):173–189, 1972.

[4] R. Bayer and M. Schkolnick. Concurrency of operations on B-trees. *Acta Informatica*, 9(1):1–21, 1977.

[5] A. Biliris. Operation specific locking in B-trees. In *Proceedings of the Sixth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 159–169, San Diego, CA, U.S.A., 23–25 March 1987.

[6] Per Brinch Hansen, editor. *The Origin of Concurrent Programming: From Semaphores to Remote Procedure Calls*, chapter Cooperating sequential processes, pages 65–138. Springer-Verlag New York, Inc., New York, NY, USA, June 2002.

[7] James E. Burns, Michael J. Fischer, Paul Jackson, Nancy A. Lynch, and Gary L. Peterson. Shared data requirements for implementation of mutual exclusion

using a test-and-set primitive. In *Proceedings of the International Conference on Parallel Processing*, pages 79–87, August 1978.

[8] Douglas Comer. The ubiquitous B-tree. *ACM Computer Surveys*, 11(2):121–137, 1979.

[9] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms.* The MIT Press, Cambridge, MA, U.S.A., first edition, 1990.

[10] Douglass R. Cutting, David R. Karger, Jan O. Pedersen, and John W. Tukey. Scatter/gather: A cluster-based approach to browsing large document collections. In *SIGIR '92: Proceedings of the 15th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 318–329, New York, NY, USA, 1992. ACM Press.

[11] Sajal K. Das and Marie-Anne Demuynck. B^{mad}-tree: An efficient data structure for parallel processing. In *Proceedings of the Eighth IEEE Symposium on Parallel and Distributed Processing*, pages 384–391, New Orleans, LA, U.S.A., 23–26 October 1996.

[12] Wiebren de Jonge and Ardie Schijf. Concurrent access to B-trees. In *Proceedings of the International Conference on Databases, Parallel Architectures and Their Applications (PARBASE-90)*, pages 312–320, Miami Beach, Florida, U.S.A, 7–9 March 1990.

[13] Carla Schlatter Ellis. Concurrent search and insertion in 2-3 trees. *Acta Informatica*, 14:63–86, 1980.

[14] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the Association for Computing Machinary*, 32(2):374–382, April 1985.

[15] Goetz Graefe and Per-Åke Larson. B-tree indexes and CPU caches. In *Proceedings of the International Conference on Data Engineering (ICDE'01)*, pages 349–358, Heidelberg, Germany, 2–6 April 2001.

[16] Leonida J. Guibas and Robert Sedgewick. A dichromatic framework for balanced trees. In *Proceedings of the 19th Annual Symposium of Foundation of Computer Science (FOCS), IEEE*, pages 8–21, October 1978.

[17] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, January 1991.

[18] Kai Hwang and Zhiwei Xu. *Scalable Parallel Computing.* The McGraw-Hill Companies, Boston, MA, U.S.A., first edition, 1998.

[19] IBM Corporation. IBM system/370 principles of operation. *IBM Publication*, Order Number GA22-7000, 1981.

[20] IBM Corporation. IBM system/370 extended architecture principles of operation. *IBM Publication*, IBM Publication Number SA22-7085, 1983.

[21] Jong Ho Kim, Helen Cameron, and Peter Graham. Lock-free Red-Black trees using CAS. *Concurrency and Computation: Practice and experience*, pages 1–40, 2006.

[22] Donald Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching.* Addison-Wesley, U.S.A., third edition, 1997.

[23] V. Lanin and D. Shasha. A symmetric concurrent B-tree algorithm. In *Proceedings of 1986 ACM Fall Joint Computer Conference*, pages 380–389, Dallas, TX, U.S.A., November 1986.

[24] Philip L. Lehman and S. B. Yao. Efficient locking for concurrent operations on B-trees. *ACM Transactions on Database Systems (TODS)*, 6(4):650–670, December 1981.

[25] Jianwen Ma. Concurrent, lock-free insertion in red-black trees. Master's thesis, University of Manitoba, Winnipeg, MB, Canada, June 2004.

[26] Maged M. Michael. Safe memory recalmation for dynamic lock-free objects using atomic reads and writes. In *Proceedings of 21st Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 21–30, Monterey, California, U.S.A., 21–30 July 2002.

[27] Maged M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Transactions on Parallel and Distributed Systems*, 15(6):491–504, June 2004.

[28] Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 267–275, Philadelphia, PA, U.S.A., 23–26 May 1996.

[29] R. Miller and L. Snyder. Multiple access to B-trees. In *Proceedings of the Conference on Information Sciences and Systems*, pages 173–189, John Hopkins University, Baltimore, U.S.A., March 1978.

[30] Seyed H. Roosta. *Parallel Processing and Parallel Algorithms: Theory and Computation.* Springer-Verlag New York, Inc., New York, NY, U.S.A., first edition, 2000.

[31] Yehoshua Sagiv. Concurrent operations on B-trees with overtaking. In *Fourth Annual ACM SIGACT/SIGMOD Symposium on Principles of Database System (PODS)*, pages 28–37, Portland, OR, U.S.A., 25–27 March 1985. ACM.

[32] Behrokh Samadi. B-trees in a system with multiple users. *Information Processing Letters*, 5(4):107–112, 1976.

[33] Lui Sha, Ragunathan Rajkumar, and John P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, 1990.

[34] V. Srinivasan and Michael J. Carey. Performance of B-tree concurrency control algorithms. In *Proceedings of the 1992 ACM SIGMOD international conference on Management of data*, pages 331–340, San Diego, CA, U.S.A., 2–5 June 1992. ACM.

[35] Venkathachary Srinivasan and Michael J. Carey. Performance of B^+tree concurrency algorithms. *The VLDB Journal The International Journal on Very Large Data Bases*, 2(4):361–406, October 1993.

[36] Elizabeth C. Tyler, Martha R. Horton, and Philip R. Krause. A review of algorithms for molecular sequence comparison. *Computers and Biomedical Research*, 24(1):72–96, 1991.

[37] John D. Valois. Lock-free linked lists using compare-and-swap. In *Proceedings of the 14th International Symposium on Principles of Distributed Computing*, pages 214–222, Ottawa, ON, Canada, August 1995.

[38] Hartmut Wedekind. On the selection of access paths in a database system. In J. Klimbie and K. Koffeman, editors, *IFIP Working Conference on Data Base Management*, pages 385–397, North Holland, Amsterdam, New York, 1974.

[39] Dachywan Wu, James Robergé, Douglas J. Cork, Bao Gia Nguyen, and Thom Grace. Computer visualization of long genomic sequences. In *VIS '93: Proceedings of the 4th conference on Visualization '93*, pages 308–315, 1993.

[40] Haruo Yokota, Yasuhiko Kanemasa, and Jun Miyazaki. Fat-Btree: An update-conscious parallel directory structure. In *Proceedings of the International Conference on Data Engineering (ICDE'99)*, pages 448–457, Sydney, New South Wales, Australia, 23–26 March 1999.

[41] John Zahorjan, Edward D. Lazowska, and Derek L. Eager. Spinning versus blocking in parallel systems with uncertainty. In *Proceedings of the International Symposium on Performance of Distributed and Parallel Systems*, pages 455–472, Kyoto, Japan, December 1988.