Techniques for Concurrent Access to Different Data Structure: A Research Review

Ranjeet Kaur[†] **and Pushpa Rani Suri** [††],

Kurukshetra University, Kurukshetra

Summary
The data structures used in concurrent systems need to be modified. Modifications of shared data structures are done in several steps. If these steps are interleaved with modifications from other tasks, this can result in inconsistency of the data structure. Therefore the data structure needs to be protected from other tasks modifying it while the operation is executing. This can either be done using mutual exclusion(locks)/non-blocking/optimistic methods. The focus of this paper is to give a preview on data structure with efficient and practical approach of concurrency control.

Key words:
Concurrency, lock-free, non-blocking, memory management, compares and swap, Elimination.

1. Introduction

Multi-processor or multi-core machine is much better at power-performance ratio than a single processor machine with the same performance. In this era every computer must be at least dual-core. A dual-core machine can perform two computing tasks simultaneously.
The basic unit of scheduling is generally the thread; if a program has only one active thread, it can only run on one processor at a time. If a program has multiple active threads, then multiple threads may be scheduled at once. In a well-designed program, using multiple threads can improve program throughput and performance. There is a technical distinction between thread and processes. a process can be whole program such as emacs, word, Mozilla. The OS allows more than one process to run at the same time. A thread is a unit of execution with in a process that has less overhead and quicker context switch time. The multiple threads of a single process can share the variables and data structure i.e. access to the memory. The environment in which threads share the common data structure is called a concurrent access. Below sections discussed concurrency and the various techniques related to it.

2. Concurrency control techniques

Simultaneous execution of multiple threads/process over a shared data structure access can create several data integrity and consistency problems:

- Lost Updates.
- Uncommitted Data.
- Inconsistent retrievals

All above are the reasons for introducing the concurrency control over the concurrent access of shared data structure. Concurrent access to data structure shared among several processes must be synchronized in order to avoid conflicting updates. Synchronization is referred to the idea that multiple processes are to join up or handshake at a certain points, in order to reach agreement or commit to a certain sequence of actions. The thread synchronization or serialization strictly defined is the application of particular mechanisms to ensure that two concurrently executing threads or processes do not execute specific portions of a program at the same time. If one thread has begun to execute a serialized portion of the program, any other thread trying to execute this portion must wait until the first thread finishes.
Concurrency control techniques can be divided into two categories.

- Blocking
- Non-blocking

Both of these are discussed in below sub-sections.

2.1 Blocking

Blocking algorithms allow a slow or delayed process to prevent faster processes from completing operations on the shared data structure indefinitely. On asynchronous (especially multiprogrammed) multiprocessor systems, blocking algorithms suffer significant performance degradation when a process is halted or delayed at an inopportune moment. Many of the existing concurrent data structure algorithms that have been developed use mutual exclusion i.e. some form of locking.
Mutual exclusion degrades the system's overall performance as it causes blocking, due to that other concurrent operations cannot make any progress while the access to the shared resource is blocked by the lock. The limitation of blocking approach are given below

Manuscript received June 5, 2013
Manuscript revised June 20, 2013

  - Priority inversion: occurs when a high-priority process requires a lock holded by a lower-priority process.
- Convoying: occurs when a process holding a lock is rescheduled by exhausting its quantum, by a page fault or by some other kind of interrupt. In this case, running processes requiring the lock are unable to progress.
- Deadlock: can occur if different processes attempt to lock the same set of objects in different orders.
- locking techniques are not suitable in a real-time context and more generally, they suffer significant performance degradation on multiprocessors systems.

2.2 Non-blocking

Non-blocking algorithm Guarantees that the data structure is always accessible to all processes and an inactive process cannot render the data structure inaccessible. such an algorithm ensure that some active process will be able to complete an operation in a finite number of steps making the algorithm robust with respect to process failure

In the following sections we discuss various non-blocking properties with different strength.

Wait-freedom
A method is wait-free if every call is guaranteed to finish in a finite number of steps. If a method is bounded wait-free then the number of steps has a finite upper bound.
From this definition it follows that wait-free methods are never blocking, therefore deadlock cannot happen. Additionally, as each participant can progress after a finite number of steps (when the call finishes), wait-free methods are free of starvation.

Lock-freedom
Lock-freedom is a weaker property than wait-freedom. In the case of lock-free calls, infinitely often some method finishes in a finite number of steps. This definition implies that no deadlock is possible for lock-free calls. On the other hand, the guarantee that some call finishes in a finite number of steps is not enough to guarantee that all of them eventually finish. In other words, lock-freedom is not enough to guarantee the lack of starvation.

Obstruction-freedom
Obstruction-freedom is the weakest non-blocking guarantee discussed here. A method is called obstruction-free if there is a point in time after which it executes in isolation (other threads make no steps, e.g.: become suspended), it finishes in a bounded number of steps. All lock-free objects are obstruction-free, but the opposite is generally not true.
Optimistic concurrency control (OCC) methods are usually obstruction-free. The OCC approach is that every participant tries to execute its operation on the shared object, but if a participant detects conflicts from others, it rolls back the modifications, and tries again according to some schedule. If there is a point in time, where one of the participants is the only one trying, the operation will succeed.

3. Literature reviewed:

Concurrent access to shared data in preemptive multi-tasks environment and in multi-processors architecture has been subject to many works. Based on these works, we will review the work done on blocking as well as non-blocking approach. The last section of the paper comprises of a comparative study of these approaches.

3.1 Blocking based Algorithms:

Based on the approach of locking Philp et al [1] algorithm has property that any process for manipulating the tree uses only a small number of locks at any time, no search through the tree is ever prevented from reading any node, for that purpose they have considered a variant of B* -Tree called B^{link}- tree (in figure 1).

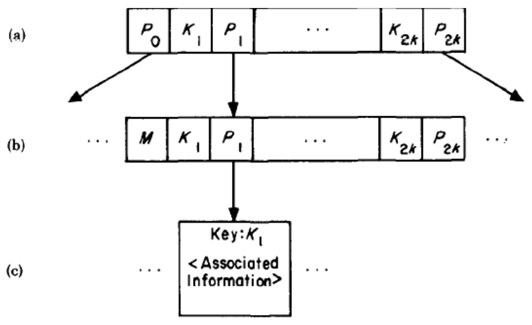

Fig. 1. B*-tree nodes (with no "high key").

The B^{link}-tree is a B*-tree modified by adding a single "link" pointer field to each node This link field points to the next node at the same level of the tree as the current node, except that the link pointer of the rightmost node on a level is a null pointer. This definition for link pointers is consistent, since all leaf nodes lie at the same level of the tree. The B^{link}-tree(in figure 2) has all of the nodes at a particular level chained together into a linked list.

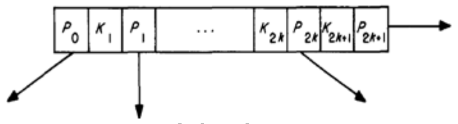

Fig.2. B link-nodes

The main priority given by previous concurrency control algorithm for solution to concurrency control problem was characterized by the use of just one lock types but this solution does not use catenations and distributions and permits the existence of underflown nodes. The system that he has sketched is far simpler than one that requires Underflows and concatenations. It uses very little extra storage under the assumption that insertions take place more often than deletions.
The aim of number of solutions given for handling concurrency control (in pessimistic approach) is to reduce the number of locks required on B-tree data structure. In this direction Mond et. al. [2] suggested the idea based on top-down approach. The idea was immediate splitting or merging of unsafe nodes (those nodes which have no space for more insertion or those which haven't a key for deletion) in order to avoid long chains of locks, By making these preparatory operations only a pair of locks has to be kept: on the current node and on its father node. Thus the portion of the tree locked by the process is getting smaller as the process advances. In order to carry out the idea of preparatory operations basic B+ tree was modified and called preparatory operations B+ tress (PO-B+-tree). This approach has introduced some overhead by increasing the number of operations performed upon the tree, but as the rate of the tree increases the relative number of unsafe nodes in the tree are reduced, hence this overhead become small.

Goal of Concurrency control Algorithms
The goals of concurrency control algorithms are
(I)Reduce the collision occurred during the concurrent execution of transactions using efficient locking mechanism.
(II)Reduce the access time, increase throughput, and minimize the frequency of tree restructuring.
Sakti et al [3] tried to achieve later one goal by introducing node partitioning scheme for large node B-trees to enhance concurrency. In his proposed scheme, each node is partitioned into multiple sub nodes to be distributed for parallel processing.The proposed B-tree structure is called as PNB-trees (partitioned node B-trees). The node size of PNB trees is large, but I/O and computation time improves significantly because large nodes are split into smaller sub nodes and these sub nodes are processed in parallel. Another important factor is how many transactions wait for an access to a data object locked by a given transactions. Avoiding such bottleneck as much as possible is a reasonable Concurrency control on data structures requires solutions that should meet, in addition to correctness criteria, those of high throughput. To achieve high throughput, it is necessary to maximize parallelism of transactions execution. Common factor in parallelism is the period of time for which data objects become unavailable due to the concurrency control technique purpose in design of concurrent data structure algorithms.
Methods for controlling concurrent access to B-trees have been studied for a long time [17,18,19] none of those considered thoroughly the problem of efficiently guaranteeing serializability [20] of transactions containing multiple operations on B-trees, in the face of transaction and system failures, and concurrent accesses by different transactions with fine-granularity locking. [21] Presents an incomplete (in the not found case and locking for range scans) and expensive (using nested transactions) solution to the problem. In spite of the fine-granularity locking provided via record locking for data and key value locking for the index information, the level of concurrency supported by the System R protocols, which are used in the IBM product SQUDS, has been found to be inadequate by some customers [22].There was need of improvement in System R index concurrency control, performance, and functionality.
C. MOHAN [4] drastically improves the problems of System R. He has proposed a method for Concurrency control in B-tree indexes according to that a transaction may perform any number of non index and index operations, including range scans. ARIES/KVL (Algorithm for Recovery and Isolation Exploiting semantics using Key-Value Locking) guarantees serializability and it supports very high concurrency during tree traversals, structure modifications, and other operations. Unlike in System R, when one transaction is waiting for a lock on a key value in a page, reads and modifications of that page by other transactions are allowed. Further, transactions that are rolling back will never get into deadlocks.

3.2 Non-Blocking based algorithm:

With the goal of designing a concurrent queue that supports the normal ENQUEUE and DEQUEUE operations J.D valois et.al[5] implemented a lock-free FIFO queue. the data structure ,in this algorithm is composed of records,each containing two fields:next,a pointer to the next record in the list,and value, the data value stored in the record. Two global pointers , head and tail ,point to records on the list:these pointers are used to quickly find the correct record while dequeuing and enqueuing ,recpectively.
The algorithm first link the new node to the end of the list ,and then updates the tail pointer. the DEQUEUE operation work slightly different, however. Rather than having head point to the node currently at the front of the queue ,it points at the last node that was dequeued.
drawing ideas from previous authors, Maged M.Michel[6] presented a new non-blocking concurrent queue algorithm ,which is simple, fast , and practical. The algorithm implements the queue as a singly-linked list with

Head and *Tail* pointers. As in Valois's [5] algorithm, *Head* always points to a dummy node, which is the first node in the list. *Tail* points to either the last or second to last node in the list. The algorithm uses `compare and swap`, with modification counters to avoid the ABA problem. To allow dequeuing processes to free dequeue nodes, the dequeue operation ensures that *Tail* does not point to the dequeued node nor to any of its predecessors. This means that dequeued nodes may safely be re-used. To obtain consistent values of various pointers we rely on sequences of reads that re-check earlier values to be sure
they haven't changed. These sequences of reads are similar to, but simpler than, the snapshots of Prakash *et al[22']*. (we need to check only one shared variable rather than two). A similar technique can be used to prevent the race condition in Stone's blocking algorithm. We use Treiber's simple and efficient non-blocking stack algorithm [21'] to implement a non-blocking free list. For making the non-blocking algorithms cost-effective, Laden et.al[7] has tried to remove the usage of costly CAS operations. The key idea behind his new algorithm was to replaced the singly-linked list of Michael and Scott[6], whose pointers are inserted using a costly compare-and-swap (CAS) operation, by an "optimistic" doubly-linked list whose pointers are updated using a simple store, yet can be "fixed" if a bad ordering of events causes them to be inconsistent. It was a practical example of an "optimistic" approach to reduction of synchronization overhead in concurrent data structures.
The key idea behind his new algorithm is to (literally) approach things from a different direction... by logically reversing the direction of enqueues and dequeues to/from the list. If enqueues were to add elements at the beginning of the list,they would require only a single CAS, since one could first direct the new node's next pointer to the node at the beginning of the list using only a store operation, and then CAS the tail pointer to the new node to complete the insertion.
The new technique for lock free FIFO queue was introduced by mark et al.[8] that elimination , a scaling technique formerly applied only to LIFO structures, can be applied to FIFO data structures, specifically, to linearizable FIFO queues. They transformed existing nonscalable FIFO queue implementations into scalable implementations using the elimination technique, while preserving lock-freedom and linearizablity.
For that purpose they modified Michael and scott[6] and ladan et al.[7] FIFO queue algorithms. The key feature of MS-queue was that concurrent accesses to the head and tail of the queue do not interfere with each other as long as the queue is non-empty. Ladan et al. [7] introduced an optimistic queue that improves on the performance of the MS-queue in various situations by reducing the number of expensive compare-and-swap (CAS) operations performed. Unfortunately, like all previous FIFO queue algorithms, these state-of-the-art algorithms do not scale.
In all previous FIFO queue algorithms, all concurrent Enqueue and Dequeue operations synchronize on a small number of memory locations, such as a head or tail variable, and/or a common memory location such as the next empty array element. Such algorithms can only allow one Enqueue and one Dequeue operation to complete in parallel, and therefore cannot scale to large numbers of concurrent operations. In the LIFO structures elimination works by allowing opposing operations such as pushes and pops to exchange values in a pairwise distributed fashion without synchronizing on a centralized data structure. This technique was straightforward in LIFO ordered structures. As noticed by Shavit and Touitou [17]: a stack's state remains the same after a push followed by a pop are performed. This means that if pairs of pushes and pops can meet and pair up in separate random locations of an "elimination array", then the threads can exchange values without having to access a centralized stack structure. However, this approach seemingly contradicts the very essence of FIFO ordering in a queue: a Dequeue operation must take the oldest value currently waiting in the queue. It apparently cannot eliminate with a concurrent Enqueue. For example, if a queue contains a single value 1, then after an Enqueue of 2 and a Dequeue, the queue contains 2, regardless of the order of these operations.
The figure.3 shows an example execution

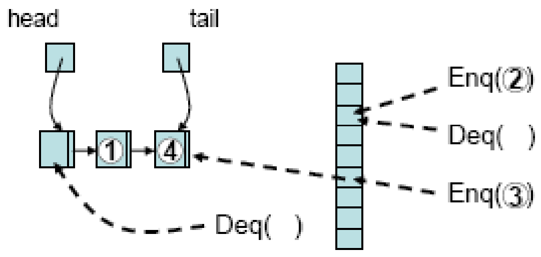

figure.3

Thus, because the queue changes, we cannot simply eliminate the Enqueue and Dequeue. Note that if the queue were empty, we could eliminate an Enqueue-Dequeue pair, because in this case the queue is unchanged by an Enqueue immediately followed by a Dequeue. There algorithm exploits this observation, but also goes further, allowing elimination of Enqueue-Dequeue pairs even when the queue is not empty. To understand why it is acceptable in some cases to eliminate Enqueue-Dequeue pairs even when the queue is not empty, one must understand the linearizability correctness condition [18], which requires that we can order all operations in such a way that the operations in this order respect the FIFO queue semantics, but also so that no process can detect that the operations did not actually occur in this order. If one operation completes before another begins, then we must order them in this order. Otherwise, if the two are concurrent, we are free to order them however we wish. Key to their approach was the observation that they wanted to use elimination when the load on the queue is high. In such cases, if an Enqueue operation is unsuccessful in an attempt to access the queue, it will generally backoff before retrying. If in the meantime all values that were in the queue when the Enqueue began are dequeued, then we can “pretend” that the Enqueue did succeed in adding its value to the tail of the queue earlier, and that it now has reached the head and can be dequeued by an eliminating Dequeue. Thus, they used time spent backing off to “age” the unsuccessful Enqueue operations so that they become “ripe” for elimination. Because this time has passed, we ensure that the Enqueue operation is concurrent with Enqueue operations that succeed on the central queue, and this allows us to order the Enqueue before some of them, even though it never succeeds on the central queue. The key is to ensure that Enqueues are eliminated only after sufficient aging.

A large number of lock-free (and wait-free) queue implementations have appeared in the literature, e.g. [14][12][8] being the most influential or recent and most efficient results. These results all have a number of specialties or drawbacks as e.g. limitations in:

- Allowed concurrency.
- Static in size.
- Requiring atomic primitives not available on contemporary architectures.
- Scalable in performance but having a high overhead.

The algorithm given by Anders et al.[9] improves on previous results by combining the underlying approaches and designing the algorithm cache-aware and tolerant to weak memory consistency models in order to maximize efficiency on contemporary multi-core platforms. The underlying data structure that his algorithmic design uses is linked list of arrays. depicted in below fig.4

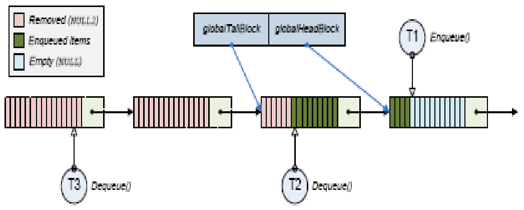

figure.4

The lock-free algorithm has no limitations on concurrency, was fully dynamic in size, and only requires atomic primitives available on contemporary. Anders et al.[9] presented a lock-free FIFO queue data structure that was presented in[22]. The algorithm supports multiple producers and multiple consumers and weak memory models. It has been designed to be cache-aware in order to minimize its communication overhead and work directly on weak memory consistency models (e.g. due to out-of-order execution).

In resemblance to [8][6][23] the algorithm discussed here was dynamic, and in resemblance to [23] removed blocks are logically deleted, blocks are being traversed and creation of long chains are avoided. In contrast to [6][23] the algorithm employs no special strategy for increasing scalability besides allowing disjoint Enqueue and Dequeue operations to execute in parallel.

The algorithm presented by Anders et al. [9] was the first lock-free queue algorithm with all of the following properties:

(i) Cache-aware algorithmic handling of shared pointers including lazy updates to decrease communication overhead.

(ii) Linked-list of arrays as underlying structure for efficient dynamic algorithmic design.

(iii) Exploitation of thread-local static storage for efficient communication.

(iv) Fully dynamic in size via lock-free memory management.

(v) Lock-free design for supporting concurrency.

(vi)Algorithmic support for weak memory consistency models, allowing more efficient implementation on contemporary hardware.

In the above review we talk about tree and FIFO queue now in the remaining section we discuss the concurrent priority queue.

A priority queue is an abstract data type that allows *n* asynchronous processors to each perform one of two operations: an Insert of an item with a given priority, and a Delete-min operation that returns the item of highest priority in the queue.

There exist several algorithms and implementations of concurrent priority queues. The literature on concurrent priority queues consists mostly of algorithms based on two paradigms: search trees and heaps .The majority of the algorithms are lock-based, either with a single lock on top of a sequential algorithm, or specially constructed algorithms using multiple locks, where each lock protects a small part of the shared data structure. lets review the various algorithm proposed for concurrent priority queue either lock based or lock free.

Most of the concurrent priority queue have been proposed ,usually based on heap structure tree,[17,4,15] etc. an obvious problem with these algorithms was that the root is serialization bottleneck. To avoid this problem Theodore et.al [10] proposed a concurrent priority queue based on the B-link tree. It is a B+ tree in which each node has a pointer to its right neighbor. The leftmost child of the B-tree is always containing the smallest key.the search operations continues in this manner until it finds the leaf that might contain the key it is searching for ,at which point it searches that leaf,unlocks the leaf ,and returns. An insert operation starts by searching for the key that is inserting. The insert operations places an exclusive lock on the leaf ,locks the parent , and insert a pointer to the sibling in the parent. if the parent is too full, the parent is half-split and the restructuring continues until a non-full parent node is reached or the root is split.

Empirical evidence collected in recent years [24, 11, 25] shows that heap-based structures tend to outperform search tree structures. This is probably due to a collection of factors, among them that heaps do not need to be locked in order to be \rebalanced," and that Insert operations on a heap can proceed from bottom to root, thus minimizing contention along their concurrent traversal paths. The algorithm given by Michel et.al[11] was based on the array based priority queue heaps. in it the deletions proceed top-down but the insertions proceed bottom- up and consecutive insertions use a bit-reversal technique to scatter across the fringe of the tree, to reduce contention. the algorithm augments the heap data structure with mutual exclusion lock on the heap ‘s size and locks on each node in the heap. Each node also has tag that indicates whether it is empty, valid , or in a transient state due to an update to the heap by an inserting process.

Unfortunately, again the empirical evidence shows, the performance of [11] does not scale beyond a few tens of concurrent processors. As concurrency increases, the algorithm's locking of a shared counter location, however short, introduces a sequential bottleneck that hurts performance. The root of the tree also becomes a source of contention and a major problem when the number of processors is in the hundreds. In summary, both balanced search trees and heaps suffer from the typical scalability impediments of centralized structures: sequential bottlenecks and increased contention.

The solution we propose in this paper by lotal et.al[12] is to design concurrent priority queues based on the highly distributed SkipList data structures of Pugh [26,27].

SkipLists are search structures based on hierarchically ordered linked-lists, with a probabilistic guarantee of being balanced. The basic idea behind SkipLists is to keep elements in an ordered list, but have each record in the list be part of up to a logarithmic number of sub-lists. These sub-lists play the same role as the levels of a binary search structure, having twice the number of items as one goes down from one level to the next.To search a list of N items, $O(\log N)$ level lists are traversed, and a constant number of items is traversed per level, making the expected overall complexity of an Insert or Delete operation on a SkipList $O(\log N)$.

Author introduced the SkipQueue, a highly distributed priority queue based on a simple modification of Pugh's concurrent SkipList algorithm [27]. Inserts in the SkipQueue proceed down the levels as in [27]. For Delete-min, multiple \minimal" elements are to be handed out concurrently. This means that one must coordinate the requests, with minimal contention and bottlenecking, even though Delete-mins are interleaved with Insert operations.

The solution was as follows. keep a specialized delete pointer which points to the current minimal item in this list. By following the pointer, each Delete-min operation directly traverses the lowest level list, until it finds an unmarked item, which it marks as \deleted." It then proceeds to perform a regular Delete operation by searching the SkipList for the items immediately preceding the item deleted at each level of the list and then redirecting their pointers in order to remove the deleted node.

Sundell et.al [13] given an efficient and practical lock-free implementation of a concurrent priority queue that is suitable for both fully concurrent (large multi-processor) systems as well as pre-emptive (multi-process) systems. Inspired by Lotan and Shavit [12], the algorithm was based on the randomized Skiplist [28] data structure, but in contrast to [12] it is lock-free.

The algorithm was based on the sequential Skiplist data structure invented by Pugh [28]. This structure uses randomization and has a probabilistic time complexity of O(logN) where N is the maximum number of elements in the list. The data structure is basically an ordered list with randomly distributed short-cuts in order to improve search times,

In order to make the Skiplist construction concurrent and non-blocking, author used three of the standard atomic synchronization primitives, Test-And-Set (TAS), Fetch-And-Add (FAA) and Compare-And-Swap (CAS).

To insert or delete a node from the list we have to change the respective set of next pointers. These have to be changed consistently, but not necessary all at once. the solution was to have additional information on each node about its deletion (or insertion) status. This additional information will guide the concurrent processes that might traverse into one partial deleted or inserted node. When we have changed all necessary next pointers, the node is fully deleted or inserted.

4. Comparison and Analysis:

Various algorithm based on blocking and non-blocking can be compared by their techniques used for concurrency control and their respective merits and demerits.shown in below table-1

Algorithm	Merits	Demerits
Efficient Locking for Concurrent Operation on B-Tree[1]	Small number of locks used	Expansive locks
Using preparatory operations on B+Trees[2]	High degree concurrency	
ARIES/KVL: key value locking method for concurrency control of multi action transactions operating on B-Trees indexes[3]	-high concurrency during tree traversal. -several B-tree operation be can part of large transaction.	
Parallel processing of large node B-Trees[4]	-Improve response time. -Reduce tree restructuring.	
Implementing Lock-Free queues[5]	Algorithm no longer needs the snapshot of prakash [21],since the only intermediate state that the queue can be in is if the tail pointer has not been updated	It required either an unaligned compare & swap or a Motorola like double-compare –and-swap, both of them are not supported on nay architecture.
Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms[6]	The algorithm was simple,fast and practical.it was the clear algorithm of choice for machine that provides a universal atomic primitive.	Poinets are inserted using costly CAS
An optimistic approach to lock-free fifo queues.[7]	It reduces the synchronization overhead in concurrent data structure	Poor in performance
Using elimination to implement scalable and lock-free FIFO queues.[8]	1. Due to scaling technique, this algorithm allows multiple enqueue and dequeue operations to complete in parallel. 2. The concurrent access to the head and tail of the queue do not interfere with each other as long as the queue is non-empty.	1. The elimination backoff queue is practical only for very short queues as in order to keep the correct FIFO queue semantics, the enqueue operation cannot be eliminated unless all previous inserted nodes have been dequeued. 2. This approach is scalable in performance as compare to previous one but having high overhead.
Efficient lock-free queue that mind the cache.[9]	All above algorithms do not consider the cache behavior ,this one improves on previous result by combining the underlying approaches and designing the algorithm cache aware	This algorithm does not used any strategy for increasing scalability besides allowing disjoint enqueue and dequeue operations to execute in parallel
. A Highly Concurrent Priority Queue Based on the B-link Tree.[10]	Avoid the serialization bottleneck	Needs node to be locked in order to be rebalance
An Efficient Algorithm for Concurrent Priority Queue	Allows concurrent insertion and deletion in opposite direction.	The performance does not scale

Heaps.[11]		beyond a few tens of concurrent processors.
Skiplist-Based Concurrent Priority Queues.[12]	Designed a scalable concurrent priority queue for large scale multiprocessor.	Algorithm based on locking approach.
Fast and Lock-Free Concurrent Priority Queues for Multithread System.[13]	This was a first lock-free approach for concurrent priority queue	

5. Conclusion

This paper reviews the concurrency control techniques with respect to different data structures (tree, queue, priority queue). The algorithms are categorized on the concurrency control techniques like blocking and non-blocking. former based on locks and later one can be lock-free, wait-free or obstruction free .in the last we can see that lock free approach outperforms over locking based approach.

References

[1] P. Lehman and S. Yao, Efficient Locking for Concurrent Operations on B-trees, ACM Trans. Database Systems, vol. 6,no. 4, 1981

[2] Y. Mond and Y. Raz , Concurrency Control on B+-Trees Databases Using Preparatory Operations, Technion - Israel Institute of Technology,1985

[3] Sakti Pramanik and Myoung Ho Kim, Parallel Processing of Large Node B-Trees,IEEE Transactions on Computers, Vol,No. 9, September 1990

[4] C. Mohan, ARIES/KVL: A Key-Value Locking Method for Concurrency Control of Multiaction Transactions Operating on B-tree Indexes, Proc. 16th Int'l Conf. Very Large Data Bases, Sept.1990.

[5] J. D. Valois. Implementing Lock-Free queues. In Seventh International Conference on Parallel and Distributed Computing Systems, Las Vegas, NV, October 1994.

[6] M. M. Michael and M. L. Scott. “Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms.” 15th ACM Symp. On Principles of Distributed Computing (PODC), May 1996. pp.267 - 275

[7] E. Ladan-Mozes and N. Shavit. An optimistic approach to lock-free fifo queues. In proceedings of the 18th International Conference on Distributed Computing (DISC), pages 117–131. Springer-Verlag GmbH, 2004.

[8] Mark Moir, Daniel Nussbaum, Ori Shalev, Nir Shavit: Using elimination to implement scalable and lock-free FIFO queues. SPAA 2005

[9] Anders Gidenstam, Håkan Sundell, Philippas Tsigas Efficient lock-free queue that mind the cache.2010

[10] T. Johnson. A Highly Concurrent Priority Queue Based on the B-link Tree. Technical Report,University of Florida, 91-007. August 1991.

[11] G.C. Hunt, M.M. Michael, S. Parthasarathy and M.L. Scott. An Efficient Algorithm for Concurrent Priority Queue Heaps. In Information Processing Letters, 60(3):151{157, November 1996.

[12] LOTAN, N. SHAVIT. Skiplist-Based Concurrent Priority Queues. International Parallel and Distributed Processing Symposium, 2000.

[13] H.Sundell and P.tsigas. Fast and Lock-Free Concurrent Priority Queues for Multithread System.

[14] Silberschatz, A., Galvin, P.: Operating System Concepts. Addison Wesley (1994)

[15] Dominique Ffober, Yann Orlarey, Stephane Letz Optimised Lock-Free FIFO Queue. Grame-Computer Music Research Laboratory January 2001

[16] Dominique Fober. Yann Orlarey. Stephane Letz Lock-Free Techniques for. Concurrent Access to Shared Objects. Grame - Centre National de Création Musicale sept 2003

[17] N. Shavit and D. Touitou. Elimination trees and the construction of pools and stacks. Theory of Computing Systems, 30:645–670, 1997.

[18] M. Herlihy and J. Wing. Linearizability: A correctness condition for concurrent objects.

[19] Bayer, R., Schkolnick, M. Concurrency of Operations on B-Trees, Acta Informatica, Vol. 9, No. 1. p l-21, 1977

[20] R. K. Treiber. Systems Programming: Coping with Parallelism. In *RJ 5118, IBM Almaden Research Center*, April 1986.

[21] J. Turek, D. Shasha, and S. Prakash. locking without Blocking: Making Lock Based concurrent DataStructure Algorithms Nonblocking. In *Proceedings of the 11th ACM SIGACT-SIGMOD-SIGARTSymposium* on Principles of Database Systems, pages 212–222, 1992

[22] Gidenstam, H. Sundell, and P. Tsigas. Cache-aware lock-free queues for multiple producers/consumers and weak memory consistency. In Proc. of the 14th Int. Conf. on Principles of Distributed Systems (OPODIS 2010), 2010.

[23] M. Hoffman, O. Shalev, and N. Shavit. The baskets queue. In Proc.of the 11th Int. Conf. on Principles of Distributed Systems (OPODIS 2007), volume 4878 of LNCS, pages 401–414. Springer, 2007.

[24] N. Deo and S. Prasad. Parallel Heap: An Optimal Parallel Priority Queue. In *The* Journal of Supercomputing, Vol. 6, pp. 87-98, 1992

[25] N. Shavit and A. Zemach. Concurrent Priority Queue Algorithms. In Proceedings of the Eighteenth Annual ACM Symposium on Principles of Distributed Computing, pages 113-122, Atlanta, GA, May 1999.

[26] W. Pugh. Skip Lists: A Probabilistic Alternative to Balanced Trees. In Communications of the ACM, 33(6):668{676, June 1990.

[27] W. Pugh. Concurrent Maintenance of Skip Lists. Technical Report, Institute for Advanced Computer Studies, Department of Computer Science, University of Maryland, College Park, CS-TR-2222.1, 1989.