IJARCCE

ISSN (Online) 2278-1021
ISSN (Print) 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering

Vol. 8, Issue 4, April 2019

Eliminate a Parallelization Technique to Implement Expandable Lock-Free Deques

Archana Srivastava

Associate Professor, BBDU, Lucknow, India

Abstract: 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 or non-blocking methods. The focus of this paper is to give a review on efficient and practical non-blocking implementations of shared data structure and suggest alternative scheme in this direction.

Keywords: Concurrency, Lock-Free Deques, Non-Blocking, Memory Management, Compare and Swap, Elimination

I. INTRODUCTION

The aim of the computer science is to design an efficient algorithm for data structures that can be shared among several execution entities. Where the execution entity can be either a process or a thread. These execution entities can access the data structure concurrently[1].This concurrent access can be either in an interleaved manner, which means execution entities gets continuously pre empted or fully in parallel.

When the several tasks are related to a shared data structure then there is need to synchronize these tasks, otherwise the concurrent access of same shared resource will leads to inconsistency. The traditional approach is to implement shared data structure objects for multiprocessor systems. Non-blocking implementations allow multiple tasks to access a shared object at the same time, but without enforcing mutual exclusion to accomplish this. Since in non-blocking implementations of shared data structures one process is not allowed to block another process, non-blocking shared data structures have the following significant advantages over lock based ones.

1. They avoid lock convoys and contention points (locks).
2. They provide high fault tolerance (processor failures will never corrupt shared data objects) and eliminate deadlock scenarios, where two or more tasks are waiting for locks held by the other.
3. They do not give priority inversion scenarios.

Traditionally there are two basic levels of non-blocking algorithms, called lock-free and wait-free. A shared data object is lock-free if it guarantees that whenever a process executes some finite number of steps toward an operation on the object ,some process (possibly a different one) must have completed an operation on the object ,during the execution of these steps. Although lock-free guarantees system-wide progress it does not ensure that individual operations eventually complete since, in theory, an operation may continually be deferred while its process helps a never-ending sequence of contending operations. In some applications a fairer condition such as wait-free may be desirable. A data structure is wait-free if and only if every operation on the structure completes after it has executed a finite number of steps. Since locks are disallowed in lock-free algorithms, they instead use the Compare-&-Swap (CAS) primitive to execute atomic read-modify-write operations on shared memory locations. This primitive takes three arguments: a memory location, a value that is expected to be read from that location, and a new value to write to the location if the expected value is found there. The most common use of CAS is to read from a memory location, perform some computation on the value, and then use CAS to write the modified value while ensuring that the location has not meanwhile been altered. CAS is supported in hardware by most modern multiprocessor architectures. Those that do not implement CAS provide alternative machine instructions that can be used to emulate CAS with very little overhead.

Many researchers have proposed lock-free algorithms for concurrent access to shared data structures like link list, Stack, Dictionary, Queue, and DEQueue. Valois et.al[3][4] has constructed a lock free implementation of singly linked list using the CAS atomic primitive and introduced the cursor concept in order to keep track of each process' relative position while traversing the list. There is a more efficient solution that eliminates the need for auxiliary nodes as it can update the next pointers atomically together with a deletion mark using the standard CAS operation suggested by

Harries et.al [5]. Valois et.al[3] has presented Lock-free implementation of double linked list using multiples of auxiliary nodes and the CAS atomic primitive. However, the presented implementation is not general and complete as it lacks the possibility to delete nodes. Due to these limitations M. Greenwald et.al [6] has presented Lock-free doubly linked list implementation based on the non-available CAS2 atomic primitive. The contribution of this paper is to give exhaustive survey of solutions given for lock-free concurrent access to queue data structure and introduced new ideas in this direction. The research in this area can be primarily classified in to the following broad directions.

1. CAS based non-blocking concurrent queue algorithms.
2. By applying optimistic approach to lock-free FIFO queue.
3. CAS based non-blocking concurrent deques algorithms.
4. Elimination technique for scalable lock-free FIFO queue.

II. CAS BASED NON-BLOCKING CONCURRENT QUEUE ALGORITHMS

Concurrent queues are widely used in parallel applications and operating system. Memory reuse in link-based lock-free data structures like queue requires special care. Many lock-free algorithms require deleted nodes not to be reused until no active pointers point to them. Also, most lock-free algorithms use the compare and swap atomic primitive, which can suffer from the “ABA problem” associated with memory reuse[7]. (Valois [3])[10] proposed a memory management method for link-based data structures that addresses these problems. The method associates a reference count with each node of reusable memory. A node is reused only when no processes or data structures point to it. The method solves the ABA problem for acyclic link-based data structures, and allows lock-free algorithms more flexibility as nodes are not required to be freed immediately after a delete operation (e.g. dequeue, pop, delete min, etc.). However, there are race conditions that may corrupt data structure that use this method. Michael.et.al [11] has proposed a lock-free algorithm for concurrent queue and it also corrected race conditions in the memory management mechanism.

III. BY APPLYING OPTIMISTIC APPROACH TO LOCK-FREE FIFO QUEUE

The main source of inefficiency of Michael [11] algorithm is that, it’s dequeue operation requires a single successful CAS in order to complete, while the enqueue operation require two such successful CASs. This may increases the chances of failed CAS operations, even on the modern multiprocessors ,the successful CAS operations cost an order-of-magnitude longer to complete than a load or a store, since they require exclusive ownership and flushing of the processor’s write buffers([18][19 optimistic approach to queue])[18][19]. The algorithm of Edya.et.al [20] presented a new dynamic-memory lock-free FIFO queue algorithm that performs consistently better than the MS-queue [11]. It is a practical example of an “optimistic” approach to reduction of synchronization overhead in concurrent data structures. At the core of this approach is the ability to use simple stores instead of CAS operations in common executions, the key idea 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.

IV. CAS BASED NON-BLOCKING CONCURRENT DEQUES ALGORITHMS

The double-ended queue (deque) object type ([11 CAS based deques])[12] supports four operations on an abstract sequence of items: PushRight, PudhLeft, PopRight and PopLeft. PushRight (PushLeft) inserts a data item onto the right (left) end. PopRight (PopLeft) removes and returns the rightmost (leftmost) data item, if any. M.M.Michal [13] gives first CAS based lock-free algorithm for shared deques, because the prior lock-free algorithms for shared deques ([1, 4, 5 CAS based deques])[14.15,16] depend on the strong DCAS(Double-Compare-and Swap) atomic primitive , which is not supported on most current processor architectures. The algorithm can use single-word CAS or LL/SC, if the nodes available for use in the deque are pre-allocated statically. This algorithm offers significant advantages over prior lock-free algorithms for shared deques with respect to the strength of required primitive and performance but This algorithm requires CAS of double-width for supporting fully dynamic sizes and is not disjoint-access-parallel. Sundell et.al [17], improves this algorithm by presenting the general operations for a lock-free doubly linked list as well as operations for a deque abstract data type. The algorithm is implemented using common synchronization primitives that are available in modern systems. The underlying data structure is fully consistent when it is idle, and the algorithm supports bidirectional traversals also through deleted nodes and the data structure is fully dynamic in size and there is an upper bound of the amount of memory occupied at any time. The algorithm, which also allows disjoint accesses on the data structure to perform in parallel.

ISSN (Online) 2278-1021
ISSN (Print) 2319-5940

Vol. 8, Issue 4, April 2019

V. ELIMINATION TECHNIQUE FOR EXPANDABLE LOCK-FREE FIFO QUEUE

A recent research by [20] introduced an optimistic queue that improves on the performance of the MS-queue [11] 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. Moir et.al [21] shows how existing non scalable queue implementations including both of the above state-of-the-art queues can be modified to support scalable FIFO elimination; this yields the first scalable non-blocking FIFO queue algorithms. Elimination is a technique to achieve scalability in shared pool and counter [22] .elimination can be used as a backoff technique that achieves scalability for LIFO stacks while preserving linearizability [23]. Linearizability is a standard correctness condition for shared data structures.[24]. Elimination works by allowing opposing operations such as pushes and pops to exchange values in a pair wise distributed fashion without synchronizing on a centralized data structure. The 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. Despite this inherent difficulty, moir has introduced additional FIFO queue operation: NumDeqs and NumEnqs. These can be transformed into a scalable elimination queue, while preserving lock-freedom and linearizablity.

VI. CONCLUSION & FUTURE WORK

The work done by researchers for over two decades is difficult to compile in this short index, however an effort has been done to broadly categorize the work and include the major milestones in the implementation of non-blocking concurrent queues and deques algorithms. I have studied that, with care, the elimination technique already known to be useful for making stacks, pools and counters scalable, can also be applied to FIFO queues. Future work includes applying elimination a parallelization technique of scaling to double ended queue for increasing number of concurrent operations.

REFERENCES

[1]. Hakan Sundell. Efficient and Practical Non-Blocking Data Structures, Ph.D. dissertation, Chalmers University of Technology, Nov. 2004.
[2]. M. Greenwald and D. Cheriton. The synergy between non-blocking synchronization and operating system structure. In Proceedings of the Second Symposium on Operating System Design and Implementation, pages 123–136, 1996
[3]. J. D. Valois, "Lock-free data structures," Ph.D. dissertation, Rensselaer Polytechnic Institute, Troy, New York, 1995.
[4]. J. D. Valois, "Lock-free linked lists using compare-and-swap," in Proceedings of the 14th Annual Principles of Distributed Computing, 1995, pp. 214–222.
[5]. T. L. Harris, "A pragmatic implementation of non-blocking linked lists," in Proceedings of the 15th International Symposium of Distributed Computing, Oct. 2001, pp. 300–314.
[6]. M. Greenwald, "Two-handed emulation: how to build non-blocking implementations of complex data-structures using DCAS," in Proceedings of the twenty-first annual symposium on Principles of distributed computing. ACM Press, 2002,pp. 260–269.
[7]. J. D. Valois. Lock-Free Data Structures. Ph. D. dissertation, Rensselaer Polytechnic Institute, May 1995.
[8]. M. M. Michael and M. 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 '96), pages 267–275, New York, USA, May 1996. ACM
[9]. Donald E. Knuth. The Art of Computer Programming, Volume 1: Fundamental Algorithms. Addison-Wesley, 1968.
[10]. M.M. Michael, CAS-based lock-free algorithm for shared deques, in: Proceedings of the 9th International Euro-Par Conference, in: Lecture Notes in Computer Science, Springer Verlag, 2003.
[11]. Ole Agesen, David L. Detlefs, Christine H. Flood, Alexander T. Garthwaite, Paul Martin, Nir N. Shavit, and Guy L. Steele, Jr. DCAS-based concurrent deques. In Proceedings of the 12th Annual ACM Symposium on Parallel Algorithms and Architectures, pages 137–146, July 2000.
[12]. David L. Detlefs, Christine H. Flood, Alexander T. Garthwaite, Paul Martin, Nir N. Shavit, and Guy L. Steele, Jr. Even better dcas-based concurrent deques. In Proceedings of the 14th International Symposium on Distributed Computing, LNCS 1914, pages 59–73, October 2000.
[13]. Michael B. Greenwald. Non-Blocking Synchronization and System Design. PhD thesis, Stanford University, August 1999.
[14]. Haken Sundell
[15]. Weaver, D., (Editors), T.G.: The SPARC Architecture Manual (Version 9). PTR Prentice Hall, Englewood Cliffs, NJ) (1994)
[16]. Intel: Pentium Processor Family User's Manual: Vol 3, Architecture and Programming Manual. (1994)
[17]. 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.
[18]. Mark Moir, Daniel Nussbaum, Ori Shalev, Nir Shavit: Using elimination to implement scalable & lock-free FIFO queues. SPAA 2005: 253-262
[19]. N. Shavit and D. Touitou. Elimination trees and the construction of pools and stacks. Theory of Computing Systems, 30:645–670, 1997.
[20]. D. Hendler, N. Shavit, and L. Yerushalmi. A scalable lock-free stack algorithm. In SPAA '04: Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures, pages 206–215. ACM Press, 2004.
[21]. M. Herlihy and J. Wing. Linearizability: A correctness condition for concurrent objects. ACM Transactions on Programming Languages and Systems, 12(3):463–492, July 1990.