Techniques for Reader-Writer Lock Synchronization

Bharath Reddy and Richard Fields
Process Automation R&D, Schneider Electric, Lake Forest, CA, U.S.A
Email: {Bharath.reddy, Richard.fields}@se.com

***Abstract*—A shared resource synchronization amongst many processes trying to acquire it is a major source of complexity in uniprocessor and multiprocessor systems. The common way of dealing with such complexity is to exclusively acquire the shared resource by a lock, work on it and then let go after the resource is no longer needed. A reader-writer lock paradigm is unique in a way that it allows multiple readers to share the resources amongst them as readers are not changing the state of the shared resource or allow a single writer exclusively to write into/or change the shared resource state from A–B.**

A reader-writer problem is a challenging topic and deserves its own identity. There are many reader-writer algorithms that try to solve and improve the efficiency of this synchronization. With great improvements in computer hardware and subsequently the synchronization methods in the last 2 decades, we attempt to bring all reader-writer locks together. The paper would first examine a general lock and later move on to different kinds of reader-writer locks in the literature to the best of our knowledge.

***Index Terms*—exclusively, lock, processes, reader, synchronization, writer**

I. Introduction

We begin this section with the definition of a mutual exclusion as it is important to understand that the reader-writer problem is a by-product of the mutual exclusion. In a mutual exclusion problem, a process trying to access the shared resource executes a critical section. The process which is trying to access the critical section can execute other activities before and after it executes this critical section, they are generally accepted as entry and exit points of the critical section. This process can exit in the non-critical abruptly but not in the critical section. Therefore, the objective of the mutual exclusion is to design the entry and exit section so that it follows the below requirements.

a) *Exclusion*: There is only one process which executes the critical section at any given time.

b) *Freedom*: If there is a process in the critical section, then that process must execute and complete the critical section.

II. Background

A Reader-Writer lock synchronization problem relaxes the above mutual exclusion by allowing n processes to read the critical section. However, when it comes to writing in the critical section, there is only one process which can execute. Such mechanism is quite necessary (synchronize reader-writers) and ubiquitous in the real world, more so now than 2 decades earlier because of relatively inexpensive sensors and research in this direction. One common scenario everyone is exposed to nowadays, is that of autonomous driving, where the car is continuously acquiring sensor data and GPS data while moving. In such systems, the cost of acquiring and releasing these locks which employ a Cache-Coherent (CC) shared memory with multiple processors can have a large impact on the performance (time taken to reach destination as well of safety) of such parallel systems.

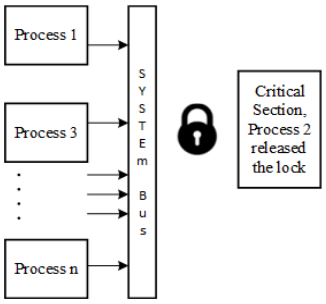


Figure 1. Example of a naive RW lock problem.

Let us begin by understanding a naive implementation of a global reader-writer lock, where the processor acquires a lock on a global variable while the other processors wait on it (to write). This entire mechanism inherently has one major problem in it, that is, the memory containing the lock (in a RAM) and the contention on the interconnection between the RAM and the processors (System bus) would become a problem when the lock is eventually freed up.

Let us see this, in the Fig. 1, as soon as the process 2 comes out of the critical section and frees the lock, there are n - 1 processors which are fighting for this global lock, and since, all n - 1 processors are trying to grab the lock at once, there is an overload on the system bus connecting the processor and the RAM. Therefore, there is a need for a better procedure to synchronize readers and writers.

III. Centralized Reader-Writer Locks

We after understanding the naive implementation of the lock, we will discuss reader-writer locks in detail in the coming sections. All reader-writer locks can be classified into centralized or queue based. Centralized

Manuscript received August 8, 2020; revised November 10, 2020.

doi: 10.18178/ijeee.8.4.63-73

reader-writer locks are those which synchronize reader-writer using centralized data while the queue-based reader-writer locks using a queue to synchronize between the readers and writers. We first begin the section by explaining the centralized reader-writer locks and move on to queue based reader-writer locks in the subsequent sections.

A. Preference Reader-Writer Lock

In this section, we are going to talk about the preference locks. A preference lock is a lock which is biased either to writers or readers. Such locks were first described by Courtois and others in [1]. These two locks are semaphore based and inherently have a problem in them. In the case of reader preference locks, the readers are prioritized ahead of the writers, thereby leading to writer's starvation. This means that the writers have high probability of deadline misses if there are consecutive readers when there is an already existing writer lock request.

When we look at the writer's preference locks, the same exact problem would persist albeit in this case, the writers taken the precedence over the readers and the readers are starved but the there is no deadline misses as in the case of the reader preference locks, at least in the real time systems. Preference locks allow fairness properties in that, in the case of reader preference, a newly arriving reader lock need not have to wait but can join and execute critical section simultaneously even if there is a writer waiting. In the case of writer preference locks, the updates are soon as there is one, even waiting writer or by a group of readers

B. Task-Fair Reader-Writer Lock

Task Fair Reader-Writer locks were first introduced by [2]. These locks are based on the task-Fair Mutex introduced in [2]. In task fair reader-writer locks, readers and writer are given same priority unlike the preference locks. As a result, both access the critical section in a strict FIFO (first in first out) fashion. More than one reader can also read the critical section but when a reader is reading the critical section, no writers can write at the same time.

The task-fair reader-writer locks is definitely an improvement to the reader or writer preference reader-writer locks, but there is a possibility that it may face a worst case scenario where the readers and writers are interleaving such that it looks more like a task-fair mutual exclusion condition rather than a reader-writer lock where multiple reader parallelism is eliminated.

Consider a condition as shown in Fig. 2, in condition a, all readers are executing after T2 writer finishes and T2 Writer enters the critical section after the T1 writer finishes and therefore, everything seems working for the task fair Reader writer algorithm.

However, there is a case where things might not look so rosy. Condition b shows a sequence of requests which will reduce the task fair Reader writer algorithm to work like a mutex. The only difference between condition a and condition b is that T1 and T4 arrival times are changed. This change now would eliminate the reader parallelism, in Condition b, T4 reads and then Writer T2 and T3 reads and then another writer T1 and eventually Reader T5, and T5 misses its deadline while it is executing the exit section of the critical section.

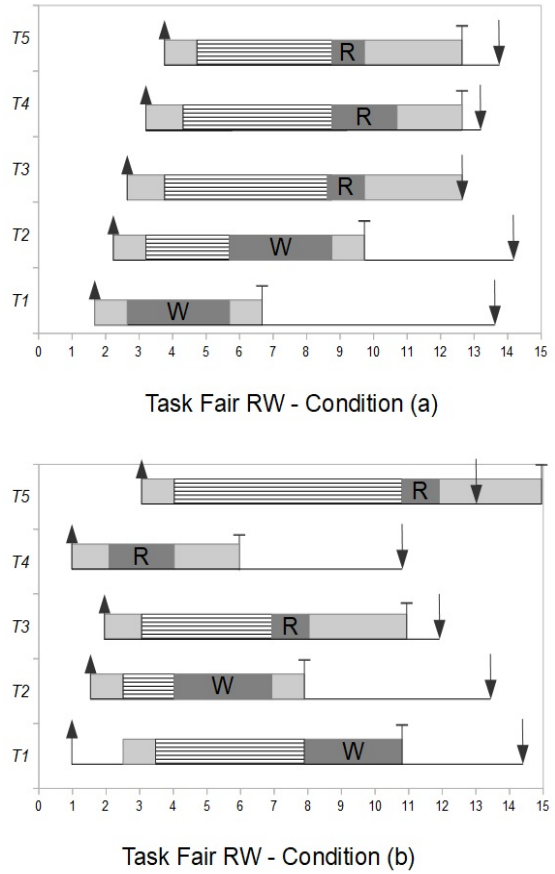


Figure 2. Example of task fair - readers parallelism and parallelism eliminated [3].

C. Reader-Writer Locks with Exponential Back-off Time

Author [4] gives three generic construction of the reader-writer lock with exponential back-off time. By exponential back-off, what the author means is a delay in time which is a strategy first introduced by [5]. Consider a situation, where all threads are waiting to get into a critical section, but they are spinning locally on their cache. Once the thread which is executing the critical section finishes and updates the counter (centralized counter), the threads respective cache gets triggered, meaning the threads now know the counter is free and they can acquire the lock.

Instead of allowing all threads to pounce at once, which is going to create havoc, and overload the central bus, [5] make the threads wait. But the question of how long the threads should wait is answered by delaying the threads to arrive at the lock at different time or spread the threads such that they all get the critical section. Initially, the threads are made to wait for a minimum time t, and the algorithm hopes that this minimum time t is good enough to spread the threads such that they all get to the critical section. When the threads fail to get the lock in the minimum delay time initially, the threads are made to

wait for double the minimum time t set earlier, if that delay time is not enough then they are doubled again, and it goes on. This exponentially backing off the threads to make sure, that there is no serious contention on the central bus is called exponential back off.

1) Reader-Writer locks with exponential back-off time

In this algorithm, the state of the lock is represented by an unassigned integer. The lowest bit is used to indicate a write is active or not. The rest of the bits are used to count the readers. When a reader comes, it increments the upper bits atomically and waits until there are no active writers. When a writer comes, it waits until all bits are clear, meaning there are no readers in the active section or are interested and no other writer is active. Since the reader only waits when there is an active writer and as soon as the writer finished the readers can proceed, no exponential back off for readers is required. On the other hand, writers may be delayed during a random number of critical sections, they are made to use exponential back off, just to reduce the contention amongst them and to minimize the load on the central bus.

2) Writer preference with exponential back-off

This algorithm is a symmetric case of the algorithm above. There are three different kind of information gathered in this algorithm. Count of the number of active readers - this is to know when all of them have finished the critical section, interested writers - to know whether a newly arriving reader to wait and lastly, the writer which is active in the critical section. This algorithm uses a single word with one bit used as a Boolean flag to indicate the writer is active in the critical section. A reader waits until there are no active or waiting writers; writer waits until there are no active readers. Exponential back off delay is used on the writers as they are unordered and to minimize contention. Readers use a ticket style proportional back-off to defer to all writers which are waiting.

D. Fair Reader-Writer Lock with Exponential Back off

This algorithm is based off Ticket lock [3], [6]. Ticket lock is an improvement on the test-and-set algorithms introduced by [5], [7]. The test and set algorithms were inherently unfair. Even with the back-off strategy introduced by [5], it was entirely possible for a thread that was waiting for a long time to be passed by a relative newcomer. In other words, the thread would starve, Ticket lock addressed this problem, it granted locks to competing threads on a first come first serve basis. This algorithm, which is based off ticket lock, has 2 pairs of counters. Each pair is a single word: its upper half keeps the count of the readers while the lower half keeps the count on the writers. Since there are 2 pairs, there are 2 words, one called request word and the other completion word. The counter values of the request word would tell how many threads requested the lock while the completion word would tell us how many threads have acquired and released the lock. Readers wait or spin until all writers are finished. Writers spin until all readers and writers have completed the critical section. The difference between requested and completed numbers of writers is used to estimate the expected wait time for both readers and writers.

E. Phase-Fair Reader-Writer Lock

Brandenburg [8] introduced a new kind of centralized reader-writer lock called the Phase-fair lock. The lock is an improvement on preference lock which caused extended blocking as one kind of threads (reader or writer). Task Fair reader-writer locks on the other hand cause blocking for a long time and parallelism is not efficiently realized. The authors [8] introduced this algorithm to overcome the short coming.

Phase-Fair lock introduced some new properties. They are:

- Reader-Writer Phases alternate between each other.
- Writers are allowed in a FIFO order.
- If there are any unsatisfied readers, then they are satisfied in the next reader phase.
- If there are writer which have not completed the critical section, then the waiting readers must wait until the next reader phase and let the writer complete the critical section.

The lock has 4 counters that count the number of issued (*rin, win*) and completed (*rout,wout*) read and write requests shown in Fig. 3. *rin* has many uses: bits 8-31 are used to count the number of issued / granted read permissions, bit 1 signifies the presence of writers who were not granted write request (unsatisfied writers). Bit 0 is used to tell the phase it is in, If the bit 0 is zero, then it is in the reader phase, if it is 1, then it is the writer phase.

When the readers come, they automatically increment *rin*, if there are no writer/s present, then the reader is admitted immediately else they spin until the bit one becomes zero. For reader phase to begin, both bit 0 and 1 must be set to zero respectively, indicating it is the beginning of the reader phase and there is no writer in the middle of executing critical section. Readers exit follow a protocol, they increment *rout*, rout signal the waiting writer that all readers have now relinquished the lock but it (writer) still needs to check the bit 0 to go to Zero before starting the writer phase and it then increments the *wout*.

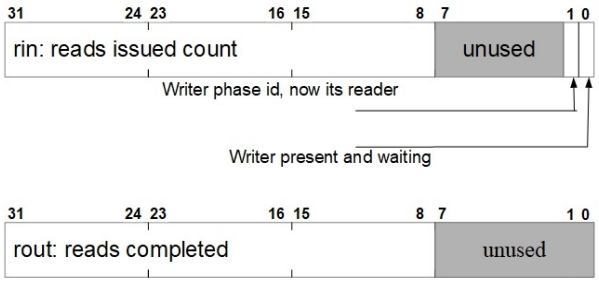


Figure 3. Bit allocation in the reader entry and exit counter [8].

Writers enter the section in a complete FIFO with a ticket abstraction [2]. The writers come and increment win and wait for the prior writers to finish. If the writer is head of the header queue, it can enter the critical section. As it enters the critical section, it observes the number of

reader requests issued. The writer exit protocol is to first make all bits from 0-8 zero, signaling the beginning of the reader phase.

IV. Queue Based Reader-Writer Locks

A. Mellor-Crumney and Scott Scalable (MCS) Spin Based Reader-Writer Locks

This Spin based reader writer lock is definitely an improvement on the Naive reader writer lock shown in section II and is derived from their exclusive lock introduced in the same paper [2]. This algorithm uses a series of atomic operations like 'atomic write', 'atomic fetch and Store', 'Compare and swap and atomic increment' and 'atomic decrement' instructions to build a single linked list for the waiting processors. So, the head of the linked list is the current processor executing the critical section while the next process to be executed is the first one in the queue.

A major improvement over the naive lock is that the processors now are not spinning on the global variable rather than on the local variable. If there is a global variable which becomes available, the first in queue processor's local variable is made zero by the current processor which holds the critical section before it exits the critical section. This means that there is always one processor which sees the global variable as zero and there is no contention in the intermediate bus connecting the processor and RAM.

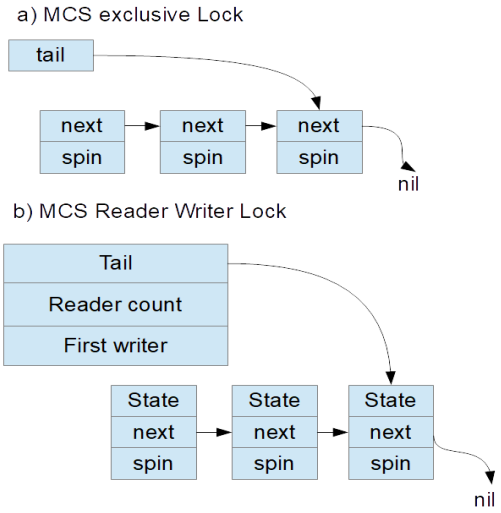


Figure 4. MCS lock and its reader-writer variant [2].

In the Reader-Writer (RW) variant, each queue element has a state element to maintain the state of request. If there is a new reader coming, then previous element is examined to check if new request must block. Since the readers can exit the queue in any order, the global variable in this kind of lock gets a little complicated in that, it now introduces two new variables, 'count of the readers' and 'first writer'. The 'count of the readers' is obvious - to keep the count of the readers. As readers come and exit, they update the global variable. If there are no global readers then it unblocks the writer, this writer is pointed by the global variable 'first writer'. If there is another writer, then it attaches itself to the first writer shown in Fig. 4.

B. Kreiger, Stumm, Unrau and Hanna MCS Improvement

Kreiger and Co. in their paper [9] improved MCS algorithm further by removing the global state by distributing the state of the readers and writers across the queue by adapting a doubly linked list for the readers. In this case (doubly linked list) for the readers, when a reader exits the queue (a reader can exit any time with our affecting others), then the reader existing would synchronize and move the pointers appropriately. However, the last existing pointer would point to the writer. If there are no readers then there is only writer, then that writer would be executing the critical and after it finishes the queue would be empty. Therefore, the improvement is not in creating a doubly linked list but the removal of global state, which in turn removes the central contention for the MCS global variable along the interconnect bus. This algorithm is shown in Fig. 5.

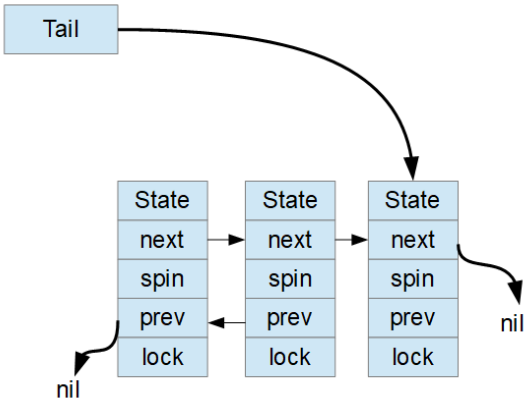


Figure 5. Kreiger and co, MCS improvement with 2 readers and one writer which is blocked [9].

C. Hsieh and Weihl Improvement

Hsieh and Weihl [10] introduce two algorithms which is an improvement over the previous algorithms. The algorithm is a little more radical than the then Lamport Lock algorithm [11] which concentrated on minimizing the number of remote database accesses but falls short as the algorithm leads to spinning over the network in the absence of cache. In the above 2 algorithms shown, we see that, both MCS and Kreiger both used fetch-add-add to eliminate remote spinning. But these two algorithms fall short when it comes to scalability, by scalability, we mean increase in readers and writers. These algorithms scale well when scalability is defined by acquisition of a lock and not by the number of readers and writers. Hsieh and Weihl try to do this, by their 2 algorithms, we will discuss in the next two sections.

1) Static reader-writer algorithm

This algorithm is shown in Fig. 5. This algorithm contains 2 semaphores, one locally by each processor in a uniprocessor or by each thread in a multi-processor system today and the other which act as a gatekeeper for the writers. By semaphore, we mean an addressable

memory address. When a reader comes, in order to acquire the lock, the reader just acquires the local semaphore, whereas the writer should acquire all the semaphores as well as the semaphore which is acting as a gate keeper for the writer to enter the critical section. The algorithm is shown in the Fig. 6. When releasing a local, a reader let goes the local semaphore, but the writer should release all the local semaphores plus the gate keeping semaphore. This algorithm has some important properties, in that, the readers do not interfere with one another and do not have to go over the network to acquire the lock. This algorithm has some short comings too, in that, it performs extremely well when there are more readers or substantially more readers than the writers but when the writers exceed the readers the performance is very poor.

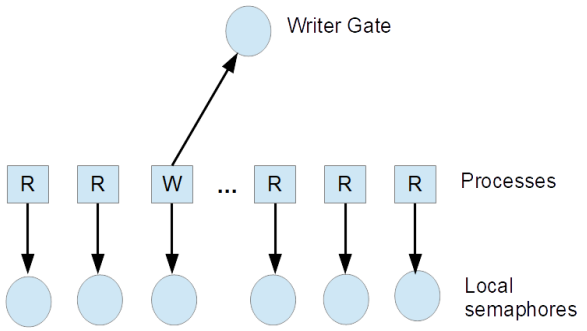


Figure 6. Hsieh and Weihl static RW lock [10].

2) Static reader-writer algorithm dynamic reader-writer algorithm

Dynamic Reader-Writer algorithm is a small improvement over the static algorithm in that, all readers have a semaphore and a bit which indicates whether the semaphore (local) is a valid or not. It also keeps track of the active readers in a centralized valid list, a list which relieves the writers from going to each core (in multiprocessor) or each processor to acquire the semaphore, writers can directly look at this list and see which processor is holding the reader lock. This saves time and the writer queue is separately maintained but mostly this queue is dynamic.

The valid list incorporated in this algorithm is a list of pointers to a shared memory which is protected by a mutex (MCS lock). If a reader wants to acquire the dynamic lock, its valid bit is set, meaning its semaphore is valid, then it can acquire its local semaphore. If the valid bit is not set, then it should try to acquire the mutex protecting the valid list. If the reader has acquired the mutex on the valid list then it checks the write bit to see if there are any active writers, if there are, then the reader releases the mutex, and joins the valid list and marks its semaphore as valid (1) and releases the mutex. If the readers want to release the dynamic lock, it must release just the local semaphore it is holding.

For the writers to acquire the lock, it first must acquire the mutex (MCS lock), then it must check if there are any active writers, if there is one, then it enlists in writer queue and waits. If not, then it sets the writer bit to 1, and invalidates all semaphores in the list, clears and lists and waits until all local semaphore of each reader to be cleared. To unlock, the writer first acquires the mutex first. if there is a writer waiting it wakes the sleeping writers in the writer queue. Otherwise, the writer checks if there are waiting readers, if there are then, it swaps waiting readers with pointers to their semaphores to the valid list, clears the writer bit and releases the mutex. If there are no readers waiting, it just clears the writer bit and releases the mutex.

D. CLH Lock

The authors in [12], [13] introduced this algorithm although they introduced the concept independently, it is collectively called a CLH lock after the authors. CLH lock is a variant of the MCS lock where each lock requester, instead of spinning on its own flag, spins on its predecessors. Each queue node in a CLH lock maintains a pointer to its predecessor, whereas in the MCS lock it maintains a pointer to its successor.

The head of the queue is a dummy node whose succ_must_wait flag is set to false. A newly arriving thread then obtains a pointer to this node (dummy node) by executing a compare and swap operation on the tail pointer. It then spins on this node. Before returning from acquire, it stores the pointer in its qnode. At the time of releasing the lock, thread writes false in the succ_must_wait field in its own node and then leaves that qnode and moves to predecessor's node. Likewise, all threads coming after executing the same way until there is no thread waiting, in which case the state of the lock return to dummy node.

Since CLH reclaims the predecessor's node, the CLH lock additionally needs better memory management. [4] proposed a new improvement where in by marking the nodes as abandoned nodes and skipping them at release time, the bottle neck scenario where in one thread taking a longer time to execute the critical section can be improved in CLH lock.

1) OrderedReadWriteLock

Authors in [14] introduce a new lock called 'Ordered Lock 'and extend this lock to reader-write lock calling it as 'OrderedReader-WriterLock'. The premise for this lock and in turn their reader-write lock is that, read-lock functionality requires additional mechanism, which are expensive in terms of overhead. These overheads are atomic operations which are more prominent than the benefits the concurrent read accesses provide. The authors argue that the excessive threads using reader-writer locks in either real time systems or otherwise systems would degrade the performance to a level of the fine-grained lock we mentioned in the previous section. The authors focus on reducing the overhead of the read-lock operation while the overhead of the write-lock operation is like a general lock. The idea to achieve the above is to reduce or remove any atomic operations other than for increment/decrement operations.

Orderedlock: An ordered lock contains 2 important steps: 1: Determine the synchronization order of the threads or processes among critical sections on a first-come-first served basis. This is like the ticket locks in [15]. An example of this step is shown in Fig. 7. There

are 4 threads T1, T2, T3 and T4, and their order is T1, T2, T1, T4 and T5. 2: Have a point-to-point synchronization between two sections with continuous order using arrays as in Anderson's array-based queue lock [5]. The first step needs an atomic operation to determine the synchronization order of each process and the second step is used to preserve the order by point-to-point synchronization.

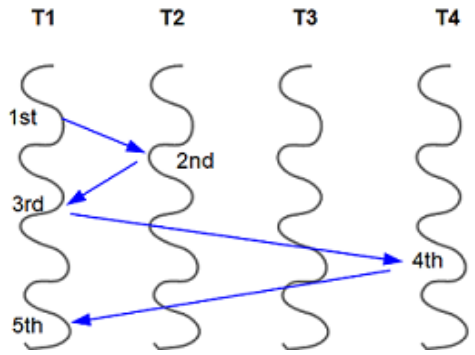


Figure 7. Synchronization order by 4 threads using ordered lock [14].

OrderedReadWritelock: The OrderedReadWrite lock write lock/unlock is very similar to a general lock/unlock write operation but the read lock/unlock operation consists of the following steps.

- Read-Lock
  - First reader: Invoke general lock OrderedLock.lock to obtain lock
  - Other readers: Wait for the first reader to obtain lock
- Read-unlock
  - Last reader: Invoke general unlock OrderedLock.unlock to release lock
  - Other readers: No-op

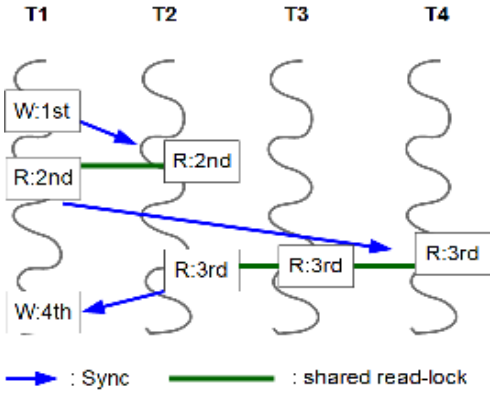


Figure 8. OrderedReadWriteLock by 4 threads [14].

In this reader-writer lock, both a reader operation and writer operation have the same priority when the write lock is released, and once a read-lock is obtained, the subsequent readers can share the read lock even though there are waiting writers. Fig. 8 shows an example for OrderedReadWriteLock by four threads T1, T2, T3, and T4. The writers, for example, W:1st and W:4th by T1, require a mutual exclusion and their own unique synchronization orders, while a synchronization order is assigned to several different readers, such as R:2nd (by T1 and T2), and R:3rd (by T2, T3, and T4). Here R:2nd by T1 is the last reader to releases the lock, and then R:3rd by T4 becomes the next first reader.

OrderedReadWritelock: The authors [16] introduce a new set of reader-writer locks which they collectively call as OLL reader writer locks. The improvement over other algorithms are their scalability over MCS algorithm and removes the need of updating a central shared data such as the queue in this case, when acquired for reading and these algorithms scale well for hundreds of threads under heavy read contention. The factor that facilitates this scalability is the total removal of knowledge of number of active readers holding the lock at any given time but to know that there are readers present. In this context they introduce a new data structure which they call it as Scalable Nonzero Indicator (SNZI) objects [17].

SNZI Object: A SNZI object introduced in [17] is a shared object that supports arrive and depart operations as well as a query operation shown below. The query operation or function returns a Boolean if there are surplus readers entering than they are readers releasing the data structure. A SNZI object authors contend that is easy to implement with a shared counter, which increments and decrements, in these operations, the SNZI object returns the shared counter value before increment or decrement it. Very similarly the read operation returns the value of the counter without modifying it as it should. The authors contend that this approach of a shared variable is more effective when it comes to scalability, than the primitive compare-and-swap operation to update a shared counter.

The authors in [16] extend the SCZI abstract data type object by adding two more operations to it, open () and close () calling it C-SNZI. This is done mainly to facilitate reader-writer synchronization. When the SCNI data type is closed, no arrivals may occur, this is synonymous with the writer blocking all readers until the write completes writing the critical section. Once the writer comes out of the critical section, it does not change the CSNZI to close but decrements the surplus. At this stage, we have writer coming out, there is no surplus and the C-SNZI is closed. Now readers can open the C-SNZI object and now there is no surplus. This is the key for the development of this data type.

Any arrivals readers or writers that come when the CSNZI return false, such arrivals do not increase the surplus. Depart function always decrements the surplus unless the Surplus is zero and the C-SNZI is closed. Query operation will always return whether there is surplus and whether C-SNZI is open. C-SNZI is initially open. Open operation is successful only when C-SNZI is closed and there is no surplus. Close operation may be called at any time; it returns true if and only if the C-SNZI was open and its surplus was zero before initialization.

FOLL Reader-Writer Lock: First-in First-out fairness OLL Reader-Writer Lock or FOLL Reader-Writer lock uses C-SNZI object on a single node which is shared amongst the readers. This is efficient as the successive

readers need not write the tail pointer in the queue as in MCS algorithm but just execute arrive and depart operation on the C-SNZI object. Writers however, first close the C-SNZI object because they need exclusion here, therefore insert their own node into the queue as in the MCS algorithm. FOLL locks has two nodes, one for readers and the other for writers. Writer nodes are very similar to MCS, but the readers nodes have three extra fields, one being a pointer to C-SNZI object. The other two being AllocReaderNode and FreeReaderNode. Every thread has its own writer node too.

A writing thread executes in pretty much the same way as the MCS algorithm, it enque itself in the queue, waits for all the readers to empty itself, and if there are no readers then would execute the critical section but with a small difference in the FOLL. The difference being, if the writer's predecessor is a reader, then the writing thread sets the predecessors qNext pointer, it closes the C-SNZI object in this readers node. This is done to prevent any existing readers to arrive at the C-SNZI pointed by this reader thread.

When a reader wants to acquire the lock, its first examines the tail of the queue. If the tail is pointing to the reader node, then it just attempts to execute the arrive () operation of the C-SNZI object of that node (already existing reader node). If the arrive operation is successful, meaning, the C-SNZI surplus is added by 1, then it waits for the spin flag of the predecessor to be false (meaning the previous reader is now executing critical section, and this present node is next in line- waking up in front of queue). If arrive operation is false, then some writer must have closed the C-SNZI after enqueing behind that node.

If the tail is null, meaning there are no current readers or writers, the reader gets the unused reader node of the lock, which just allocated, has a closed C-SNZI with no surplus and qnext = null. The reader now set the reader node of the lock's spin flag to false, attempts to enque the reader node which just got allocated to the queue and sets the tail of the lock to this reader node and enters the critical section. If it fails to get the reader node of the lock, it means that the tail of the queue must have changed, and some other thread must have enqued before it and the reader thread again tries to acquire the reader lock. If the tail of the queue is pointing to writer node, the reader tries to enque the reader node after writer node and changes the predecessor (writer) qNext pointer to reader node which it allocated. It also set its readers node spin flag to true as it needs to wait for writer before it to finish critical section.

In an MCS or Kruger queue-based reader writer lock explained in the previous section, a thread trying to acquire a lock, whether for reading purpose or writing purpose has exactly one node it enques. However, this is not the case for the FOLL lock, each reader thread is sharing the reader node amongst other readers threads. Therefore, the FOLL lock maintains a pool of reader nodes, that the threads can acquire the lock for reading. To manage this pool of readers, two more procedures are provided they are AllocReaderNode which returns a reader node that is not used and FreeReaderNode takes the node which was returned by AllocReaderNode and makes it available for future calls of AllocReaderNode. AllocReaderNode is called by the reader lock when it needs to enque a new reader node to the queue. FreeReaderNode is called when the last node from the queue is removed either by the last reader or by the writer which closes the C-SNZI of a prior Node to signal there are no readers and surplus is now zero and then enters the critical section.

ROLL Reader-Writer Lock: ROLL lock stands for reader preference FOLL lock mentioned in the previous section. In the queue of the lock, we need to find the reader nodes which is the main requirement for the ROLL. Since the nodes in the existing queue can only traverse forward, to point to next node in the queue, it might be a problem when there are readers and writers interleaved amongst in the queue. To facilitate finding the readers in the queue, the queue is made into a doubly linked list. This way, when a reader tries to acquire the lock, it travels from the tail to the head of the queue, in search of a reader node.

If it finds one, then it checks the spin flag to see if the readers using that node are still waiting to acquire the lock. If the flag is true, meaning there is spinning, the reader executes the arrive () operation of the C-SNZI object, joining other readers and sets the spin flag to true. If it does not find any other, then the thread creates a new reader node, enqueues it as described in the FOLL.

E. Biased Locking

Authors introduce biased locking mechanisms in [18]-[22] which allows same thread to acquire and release the lock without having to require atomic instruction except in the beginning, this is also called dominant thread. If another thread attempts to acquire the lock, then this results in an expensive revocation to wrest he bias away from the existing biased thread. Then revoked lock is not back to normal nonbiased mode for some period until the thread holding the lock becomes biased again. Conceptually, one can think as though the lock is left to one thread forever, where it can lock and unlock for eternity until there arises a contention for the lock, if the contention succeeds then the lock is back to the normal lock and the thread holding it would eventually become biased towards itself and the cycle continues.

1) Biased reader writer lock

Authors in [22] introduce Biased reader-writer lock using a 2-process lock and *n* process lock scheme. We will explain what this 2-process lock is, and *n*-process lock in this section. [22] use a scheme where they implement biased locks with a two-process mutual exclusion algorithm between the dominant thread and a single representative of all the other threads, chosen with a generic *n* process mutual exclusion algorithm. For the 2-process lock, they use a modified version of Petersons algorithm [23]. The Flag variable can take three values: READ, WRITE, and UNLOCK. When a dominant (biased) thread *i* tries to obtain a read lock, it spins if at the same time there is another thread *j* writing. When the dominant thread tries to obtain a write lock, it waits if there is another thread that is either reading or writing.

For non-dominant process to acquire a write-lock, it first must acquire a normal n-process write lock. This write lock is contended only by non-dominant processes or threads. Once this lock is obtained, the process checks if the dominating process is in the unlock state and then enters the critical section. At this point the non-dominant process is the only process in the critical section because the read-write lock in the algorithm (*rwlock*) provides exclusive access among the non-dominant processes. The Peterson-like algorithm which follows immediately follows it provides exclusive access from the dominant thread. For a non-dominant process to acquire a read lock, it first acquires a normal n-process read lock, which is explained in this section. Non-dominant thread acquires a read lock by contending amongst themselves in a normal n-process read lock. Once the non-dominating thread is identified, it then completes with the dominant processes or thread. if the dominating thread is writing, the non-biased or non-dominating spins on the flag variable. The last non-dominating thread to come out of the critical section would then set the flag variable to UNLOCK. The first and the last readers are checked by the variable non owner readers and this variable is protected by the n-process lock.

V. NUMA Aware Reader Writer Locks

With ever increasing presence of multi-node (chip), with many processors, there is a need to develop new set of concurrent algorithms and synchronization mechanisms. Authors [24] have developed certain reader-writer locks to such an environment. Before, we try to address the reader-writer locks, it's imperative that we discuss the term NUMA architecture. NUMA (Non-Uniform Memory Access) architecture is such that, it contains multiple nodes, where each node has local memory, cache and many cores. Such systems are envisioned as a globally visible cache coherent (meaning, if there is change of data in one shared memory address, then it is visible for all processors). Cache coherent communication channel or interconnect is present between the nodes. This is shown in Fig. 9.

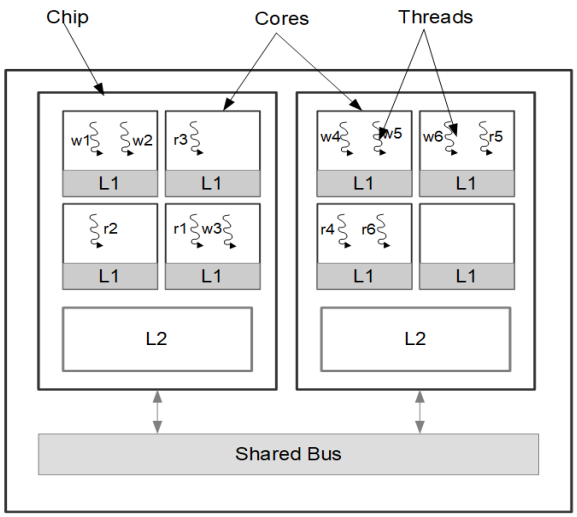


Figure 9. Multi core multi chip NUMA architecture - an example.

A. Hierarchial Backoff Lock (HBO)

Dice [25] along with Hagersten [26] were the first pioneers in throwing some light on the advantages of designing locks that improve the locality of reference on a Cache Coherent NUMA aware system by developing NUMA aware locks. By locality of reference, we mean the ability of the processes to access instructions whose memory addresses are very close to each other. These general-purpose locks were intended to encourage threads with mutual cache locality to acquire the locks consequently, thereby reducing the overall cache coherence fails [24] thereby creating an environment where the lock migration is minimized. By Lock migration, we mean 2 different threads running on two different chip set shown in Fig. 9.

[26] introduced HBO: a test-and-test-and-set lock which is augmented with a back off scheme. The idea is to reduce the cross-interconnect contention on the global lock. Because the back-off delay is dynamic, a thread if it notices that another thread which is currently holding the lock, is in the same cluster as its own, it can dynamically adjust the delay such that it is the next thread to acquire the lock. However, due to test-and-test-and-set locks, in-validations are performed immediately on the shared global lock variable, bringing more traffic which is costly in a NUMA environment.

B. HCLH Lock

To address the short comings of the HBO, [27] by developing a HCLH lock, which is a Hierarchical version of the CLH lock [12], [13]. This algorithm collects requests from each cluster into a local CLH queue, A separate thread then integrates the local CLH queue into a one global queue. This avoids the spinning on a shared location. The algorithm drawback is that the thread which is integrating the local queues must wait for a long time or must complete integrating the queue into a smaller global queue which then becomes unsatisfactory queue. To overcome this synchronization problems of the HCLH lock, [28] showed us method using a flat-combining technique of [29] and then splicing them into a global queue. The resulting lock called FC-HCLH lock outperformed the other NUMA lock by a factor of 2 [29].

C. Lock Cohorting

Lock Cohorting [30] is a technique to develop NUMA locks from NUMA-oblivious mutex locks. It has 2 basic principles. (1) Cohort detection - meaning, the lock holding thread can determine if there are other threads waiting. (2) Thread-obliviousness, this is peculiar in the sense that if the lock is acquired by one thread, then the lock can be unlocked by other thread. Lock Cohorting is hierarchical with one lock at the top and many lock holding threads on the second level, one for each node in the NUMA architecture.

A cohort lock is held by a thread when the threads owns the top-level lock. To acquire the Cohort lock, the thread must first acquire the ownership of the lock assigned to its node and then acquire the top-level lock. So, after the thread owning this cohort lock executes the critical section, it uses its cohort property to check if there

are any threads waiting, and then hands over the ownership to those threads. When the local lock hand off, the owner also passes the ownership of the top-level lock to the next thread which was waiting. If there are no threads, then the last thread releases the top-level lock. So, in Cohort locking scheme, FIFO / FCFS fairness is traded for efficiency. This is done so that there are reduced coherence traffic, improved cache residency, and to reduce interconnect bus coherence traffic and their misses (data loss).

1) C-PLT-TKTCohortLoc

Authors in [24] have developed a new cohort lock which they call C-PTL-TKT which uses ticket lock [31] for the NUMA node-local locks and a partitioned lock [32] for the top-level lock. In this lock, they expose an interface called islocked interface that allows the readers to determine if there is a write lock which is already held. This interface function is determined by comparing the request and grant count of the top-level partitioned ticket lock.

2) Neutral-Preference lock

This algorithm is in other words called a C-RW-NP which stands for (Cohort; ReadWrite; Neutral-Preference) and attempts to give more fairness between readers and writers. All arriving threads are channeled through the central cohort lock.

Each thread, either a reader or writer would first acquire a cohort lock. The reader thread uses a central lock to obtain permission to arrive at ReadIndr.arrive(), it then releases the lock immediately and enters the critical section. Writers on the other hand also acquire the cohort lock to begin, the writers then must make sure there are no lingering readers and spin on the ReaderIndr. This algorithm is simple but not readers friendly as all readers would to acquire the cohort lock. This increases the latency of read requests.

3) Reader-Preference lock

To overcome the shortfalls in the previous RW lock, reader preference lock was developed by the authors [24]. This lock is called C-RW-RP standing for Cohort-Reader Writer-Reader preference Lock. Readers and writers declare their respective existence to others and check for the status of the other threads. Readers come and check if there are any writers present in queue for the cohort lock, if they are no writers in queue, only then can the readers proceed to finish their critical section If there are writers, then the readers spin until the writers finish the critical section. The departing reader release the lock from ReadIndr.

Writers acquire the CohortLock and check of there are any active readers. If there are any readers (by looking at ReadIndr), the writers would release the CohortLock and wait for all readers to finish their critical section. This writer could potentially wait forever if there is continuous stream of readers, thereby starving the waiting writer. To avoid such a scenario, a special reader barrier called (RBarrier) which is abbreviation for reader barrier is introduced. This barrier is implemented as a central variable, in that, the writers wait for a pre-determined amount of time before which writers blocks any readers from entering the critical section. This barrier time is adjustable. When there are no readers in this barrier time, the writer would enter the critical section and then release the CohortLock.

4) Writer-Preference lock

Authors [24], which they call C-RW-WP. This is an exact symmetric form of their previous algorithm discussed in the previous section. The difference in that if the interaction of readers and writers are swapped. Readers arrive at the ReadIndr, check if there are any writers, if there are then the reader depart from the ReadIndr and spin, wait for all writers to finish their critical section and drain the queue. The readers wait for a predefined time Wbarrier which is variable and can be changed. When the reader run out of patience, it can raise the Wbarrier to block all writers from acquiring the cohort lock.

Writers on the other hand examine the Wbarrier checking if it is not raised, if not, then they go head and acquire CohortLock and they make sure there are no concurrent readers before entering critical section.

D. Augmented General Locks

This is a special kind of general lock category we had to define, because of the lock we are about to discuss. The lock does not fall into either centralized or queue based or NUMA locks but is lock which augments all existing locks.

1) BRAVO - Biased reader writer lock

Authors in [33] introduce BRAVO algorithm which is a simple transformation of all existing reader-writer locks. They do that by adding 2 integer fields to the lock instance. Readers make the writers know their presence by hashing their thread identity with the lock address, essentially creating an index to an address space which acts as a visible reader table. Readers attempt to put the lock address into this table. Since each lock reader would put the lock address in difference addressable space by using the index, it reduces coherence traffic in a NUMA environment. In this environment all locks and threads in the addressable space share the visible reader table. BRAVO can transform any existing reader-writer lock 'A' into a Bravo-A by inserting a Boolean field called RBi (reader Bias). The arriving readers would first check the RBi first, if it is set, they hash the address of the lock with the identity of the thread to form an index into the reader table which is exposed to all locks and threads as explained earlier. This addressable space the authors have experimented with is 1024 entries. The elements in this readable table are either null or pointer to a reader-writer lock instance. The reader then would use Compare-and-swap atomic operation to attempt to change the element in the table from null to the lock address (pointer to the addressable lock). If the CAS is successful then the reader checks the RBi again, if the RBi is set, it then enters the critical section. If the recheck on the RBi fails, meaning the RBi is not set, meaning there was a writer interleaved before the reader and cleared the RBi, then readers clears the index in the reader table and goes and tries to acquire the lock based off of the reader writer lock which is it based off of (hard and slow method). Writers

first acquire the write lock on the reader-writer lock they are augmenting. They then check the Rbi lock, if set, then writer performs Revocation, first clearing the Rbi flag and then checks all elements in the table for readers which have the same lock. If there are elements in the table, then it waits for all readers to clear the slot (making the element null), the writer would now enter the critical section having checked the RBi and checking. The writers unlock in a way like the underlying lock they have augmented off.

VI. Conclusion

Reader-Writer synchronization methods have come a long way, from single processor shared memory to NUMA architecture, numerous locks have been introduced. In this paper, due to the broad spectrum, some locks are deliberately omitted, such as abort locks and the paper did not talk in depth about real time locks. Also, Cohorts locks, some abortable and some non-abortable (C-BO-BO, C-TKT-TKT, C-BO-MCS, C-MCS-MCS, C-TKT-MCS, A-C-BO-BO and A-C-BO-CLH [30]) have not explained in detailed to cut the scope of this paper.

With NUMA architecture more prevalent and Moore's law more visible in our commercial life, we think that locking protocols and subsequent locks in a NUMA architecture can be further looked upon more closely in the future. We also feel that the integration protocols in merging local queues in a lock such as HCLH can be looked upon as the details on integration method is deployed is not found in the literature.

Conflict of Interest

The authors declare no conflict of interest.

Author Contributions

All the research was conducted by Bharath Reddy's, Richard Fields conducted the data analyses and proof reading. All authors had approved the final version.

Acknowledgment

The first author would like to thank Schneider-Electric in sponsoring travel expenses.

References

[1] P. Courtois, F. Heymans, and D. Parnas, "Concurrent control with "readers" and "writers"," *Communications of the ACM*, vol. 14, pp. 667-668, 1971.

[2] J. Mellor-Crummey and M. Scott, "Scalable reader-writer synchronization for shared-memory multiprocessors," *ACM SIGPLAN Notices*, vol. 26, no. 7, pp. 106-113, 1991.

[3] M. Fischer, N. Lynch, J. Burns, and A. Borodin, "Resource allocation with immunity to limited process failure," in *Proc. 20th Annual Symposium on Foundations of Computer Science*, 1979, pp. 234-254.

[4] M. Scott, "Shared-memory synchronization," in *Synthesis Lectures on Computer Architecture*, Madison: Morgan & Claypool, 2013, ch. 6, pp. 87-101.

[5] T. E. Anderson, "The performance of spin lock alternatives for shared-money multiprocessors," *IEEE Transactions on Parallel and Distributed Systems*, vol. 1, no. 1, pp. 6-16, 1990.

[6] P. D. Reed and K, R. Kanodia, "Synchronization with event counts and sequencers," *Communications of the ACM*, vol. 22, no. 2, pp. 115-123, 1979.

[7] L. Rudolph and Z. Segall, "Dynamic decentralized cache schemes for MIMD parallel processors," *ACM SIGARCH Computer Architecture News*, vol. 12, no. 3, pp. 340-347, 1984.

[8] B. Brandenburg and J. Anderson, "Reader-Writer synchronization for shared-memory multiprocessor real-time systems," in *Proc. 21st Euromicro Conference on Real-Time Systems*, 2009, pp. 184-193.

[9] O. Krieger, M. Stumm, R. Unrau, and J. Hanna, "A fair fast scalable reader-writer lock," in *Proc. International Conference on Parallel Processing*, August 16-20, 1993, pp. 201-204.

[10] W. Hsieh and W. Weihl, "Scalable reader-writer locks for parallel systems," in *Proc. the 6th International Parallel Processing Symposium*, 2002, pp. 656-659.

[11] L. Lamport, "A fast-mutual exclusion algorithm," *ACM Transactions on Computer Systems*, vol. 5, no. 1, pp. 1-11, 1996.

[12] S. T. Craig, "Building FIFO and priority-queuing spin locks from atomic swap," Technical Report TR 93-02-02, Department of Computer Science and Engineering, University of Washington, 1993.

[13] S. P. Magnusson, A. Landin, and E. Hagersten, "Queue locks on cache coherent multiprocessors," in *Proc. International Symposium on Parallel Processing*, 1994, pp. 165-171.

[14] J. Shirako, N. Vrvilo, E. Mercer, and V. Sarkar, "Design, verification and applications of a new read-write lock algorithm," in *Proc. the Twenty-Fourth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, 2012, pp. 48-57.

[15] J. Mellor-Crummey and M. Scott, "Algorithms for scalable synchronization on shared-memory multiprocessors," *ACM Transactions on Computer Systems*, vol. 9, pp. 21-65, 1991.

[16] Y. Lev, V. Luchangco, and M. Olszewski, "Scalable reader writer locks," in *Proc. the Symposium on Parallelism in Algorithms and Architectures*, 2009, pp. 101-110.

[17] F. Ellen, Y. Lev, V. Luchangco, and M. Moir, "SNZI: Scalable NonZero indicators," in *Proc. the Annual ACM Symposium on Principles of Distributed Computing*, 2007, pp. 13-22.

[18] D. Dice. (2006). Biased locking in HotSpot. [Online]. Available: https://blogs.oracle.com/dave/ biased-locking-in-hotspot.

[19] D. Dice, M. Moir, and N. W. Scherer III, "Quickly reacquirable locks," U.S. Patent 7,814,488, 2002.

[20] F. Pizlo, D. Frampton, and A. Hosking, "Fine-Grained adaptive biased locking," in *Proc. the 9th International Conference on Principles and Practice of Programming in Java*, Kongens Lyngby, Denmark, 2011, pp. 171-181.

[21] K. Russell and D. Detlefs, "Eliminating synchronization-related atomic operations with biased locking and bulk rebiasing," *ACM SIGPLAN Notices*, vol. 41, no. 10, pp. 263-272, 2006.

[22] N. Vasudevan, K. Namjoshi, and S. Edwards, "Simple and fast biased locks," in *Proc. the 19th International Conference on Parallel Architectures and Compilation Techniques*, 2010, pp. 65-74.

[23] G. L. Peterson, "Myths about the mutual exclusion problem," *Information Processing Letters*, vol. 12, no. 3, pp. 115-116, June 1981.

[24] I. Calciu, D. Dice, Y. Lev, V. Luchangco, V. Marathe, and N. Shavit, "NUMA-aware reader-writer locks," in *Proc. the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2013, pp. 157-166.

[25] D. Dice. (2003). US Patent number 07318128: Wakeup affinity and locality. [Online]. Available: http://patft.uspto.gov/netacgi/nphParser?patentnumber=7318128

[26] Z. Radovic and E. Hagersten, "Hierarchical backoff locks for nonuniform communication architectures," in *Proc. the 9th International Symposium on High-Performance Computer Architecture*, 2003, pp. 241- 252.

[27] V. Luchangco, D. Nussbaum, and N. Shavit, "A hierarchical CLH queue lock," in *Proc. the 12th International Conference on Parallel Processing*, 2006, pp. 801-810.

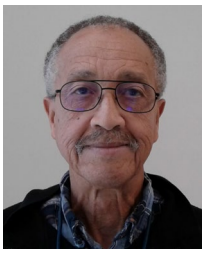
[28] D. Dice, V. Marathe, and N. Shavit, "Flat-combining NUMA locks," in *Proc. the 23rd Annual ACM Symposium on Parallelism in Algorithms and Architectures*, 2011, vol. 23, pp. 65-74.

[29] D. Hendler, I. Incze, N. Shavit, and M. Tzafrir, "Flat combining and the synchronization-parallelism tradeoff," in *Proc. Annual*

ACM Symposium on Parallelism in Algorithms and Architectures, 2010, pp. 355-364.
[30] D. Dice, V. Marathe, and N. Shavit, "Lock cohorting: A general technique for designing NUMA locks," *ACM SIGPLAN Notices*, vol. 47, pp. 247-256, 2012.
[31] J. Mellor-Crummey and M. Scott, "Synchronization without contention," *ACM SIGARCH Computer Architecture News*, vol. 19, no. 2, pp. 269-278, 1991.
[32] D. Dice, "Brief announcement: A partitioned ticket lock," in *Proc. the 23rd Annual ACM Symposium on Parallelism in Algorithms and Architectures*, 2011, pp. 309-310.
[33] D. Dice and A. Kogan, "BRAVO—Biased locking for reader writer locks," in *Proc. USENIX Annual Technical Conference*, 2019, pp. 315-328.

Bharath Reddy is a senior software designer at Schneider-Electric in US, Lake Forest, CA. He received his M.S. (Computer Science) from University of Northern British Columbia, Prince George, Canada in 2009 and M.B.A (General) from University of Phoenix, Vancouver, Canada in 2008. He received his bachelor's degree in Computer Science Engineering from Bangalore University, Bangalore, India in 2001.

He has more than 20 years of professional experience in information systems and technologies. He has published more than 9 research articles in leading journals, conference proceedings and books including ACM Transactions, and IEEE and recently published a book on Bioinformatics.



Richard Fields is an Offer Safety Literature Review Committee member, design, software/hardware specification and technical product manual review, design and maintenance of Triconex safety systems order processing procedures at Schneider-Electric in US, Lake Forest, CA. He received his B.Sc. (Physics), from University of the West Indies, Kingston, Jamaica, in 1967.

He has more than forty years of professional experience in design, software/hardware specifications and technical product manual review, design and maintenance of Triconex safety systems order processing procedures.

Mr. Richard Fields is a member of Schneider-Electric safety Literature Review Committee member.