On lock-free programming patterns

DMITRY NAMIOT
Faculty of Computational Mathematics and Cybernetics
Lomonosov Moscow State University
MSU, Russia, 119991, Moscow, GSP-1, 1-52, Leninskiye Gory
RUSSIA
dnamiot@gmail.com

Abstract: - Lock-free programming is a well-known technique for multithreaded programming. Lock-free programming is a way to share changing data among several threads without paying the cost of acquiring and releasing locks. On practice, parallel programming models must include scalable concurrent algorithms and patterns. Lock-free programming patterns play an important role in scalability. This paper is devoted to lock-free data structures and algorithms. Our task was to choose the data structures for the concurrent garbage collector. We aim to provide a survey of lock-free patterns and approaches, estimate the potential performance gain for lock-free solutions. By our opinion, the most challenging problem for concurrent programming is the coordination and data flow organizing, rather than relatively low-level data structures. So, as the most promising from the practical point of view lock-free programming pattern, we choose the framework based on the Software Transactional Memory.

Key-Words: - parallel programming, thread, lock, mutex, semaphore, scalability, atomic

1 Introduction

When writing multi-threaded code, it is often necessary to share data between threads. If multiple threads are simultaneously reading and writing the shared data structures, memory corruption can occur [1]. The simplest way of solving this problem is to use locks [2].

A lock is a thread synchronization mechanism. Let us see this classical example in Java code (Code snippet 1):

```
public class Counter{

  private int count = 0;

  public int increase(){
    synchronized(this){
      return ++count;
    }
  }
}
```

Code snippet 1. Thread synchronization

In this code, the *synchronized (this*) block in the *increase()* method makes sure that only one thread can execute the return ++*count* at a time.

A package *java.util.concurrent.lock* supports another thread synchronization mechanism. The main goal is similar to synchronized blocks, but it is more flexible and more sophisticated. It is a classical implementation of mutex [3]

```
Lock lock = new ReentrantLock();
lock.lock();
/*
  here is a critical section
*/
lock.unlock();
```

Code snippet 2. A critical section

First a Lock object is created. Then it's lock() method is called and the instance of Lock object is locked. Now any other thread calling lock() will be blocked until the thread that locked the lock calls unlock(). In the critical section, we can perform some calculation with the shared data. Finally unlock() method is called, and the instance of Lock object becomes unlocked, so other threads can lock it.

But locking includes various penalties. And it is not only the performance. For example in Java, one serious question is the ability for Java Virtual Machine (JVM) to optimize the workflow (change the order of calculations). Locks are very often

preventing JVM from this optimization. By this reason, lock-free patterns and data structures are an important part of the programming techniques.

Lock-free programming (lockless in several papers [4]) covers not only programming without the locks (mutexes). More broadly (practically), the lock is the ability for one thread to block by some way another thread. So, in the lock-free application, any thread cannot block the entire application.

From the latest definition, we can conclude that in lock-free programming one suspended thread will never prevent other threads from making progress, as a group, through their own lock-free operations. It highlights the importance of lock-free programming for real-time systems, where certain operations must be completed completely within a certain time limit. And the ability to finish operations should not depend on the rest of the program [5]. A methodology for creating fast wait-free data structures is presented in [6]. There are two basic moments: low-level lock-free data structures and relatively high-level solutions (e.g., frameworks) lock-free programming models [7].

The rest of the paper is organized as follows. In section 2 we discuss atomic operations. In section 3 we describe the basic lock-free patterns. And section 4 is devoted to the usage of lock-free approach.

2 Atomic operations

Atomic operations are the key moment for lock-free programming support. The definition for atomic operation is very simple - no thread can observe the operation half-complete. Actually, on modern processors, many operations are already atomic. For example, aligned reads and writes of simple types are usually atomic. Figure 1 illustrates the tree:

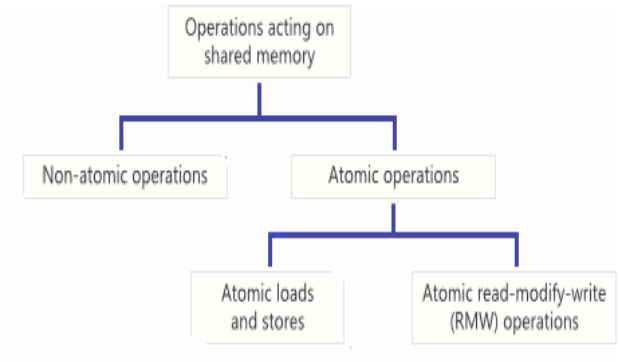

Fig. 1. Atomic vs. non-atomic operations [8]

An operation, acting on shared memory, is atomic as soon as it completes in a single step relative to other threads. This “single step” is the warranty that no other thread can observe the modification half-complete.

When an atomic load is performed on a shared variable, it reads the entire value as it is at a single moment of time. Non-atomic loads and stores do not make those guarantees. For example, in Java atomic operations are:

- all assignments of primitive types except for long and double
- all assignments of references
- all operations of classes of the *java.concurrent.Atomic* package

Without atomic operations, lock-free programming would be impossible. The reason is obvious – we cannot let different threads manipulate a shared variable at the same time without atomic operations. Suppose we have the two threads perform write operations on a shared variable concurrently. Because the thread scheduling algorithm can swap between threads at any time, we do not know the order in which the threads will attempt to gain access the shared variable. So, the result of the change in data depends on the thread scheduling algorithm. In other words, both threads are racing to gain access the data. It is so-called race condition [9].

This problem often occur when one thread performs so-called a "check phase-then-act phase" (e.g. "check" if the value is 5, then "act" to do something that depends on the value being 5), and another thread performs something with the value in between the "check phase" and the "act phase". For example,

```
if (x == 5) // Check phase
{
   x = x * 2; //  Act phase
}
```

Code snippet 3. Act & check phases

Another thread can change x value in between "if (x == 5)" and "x = x * 2" operators.

The point being, x could be 10, or it could be anything, depending on whether another thread changed x in between the check and act phases. So, as soon as two threads operate (access to write) on a shared variable concurrently, both threads must use atomic operations. Code snippet 4 illustrates an

example of atomic operations in Java (*AtomicInteger* class).

The next set of very important atomic elements are so-called Read-Modify-Write (RMW) operations [10]. They let us perform more complex transactions atomically. RMW operations are especially useful when our algorithm must support multiple writers. When the several threads attempt to perform a RMW operation on the same data, they will be effectively serialized (line up) in a row and execute those operations one-at-a-time. Here is a typical example, presented in [10]. Two transactions perform operations on their snapshot isolated from other transactions and eventually check for conflicts at commit time. The transactions temporarily store data changes (insert, update, and delete operations) in a local buffer and apply them to the shared data store during the commit phase. The commit of a transaction T1 will only be successful if none of the items in the write set have been changed externally by another transaction T2 which is committing or has committed since T1 has started. Suppose, T2 and T1 are running in parallel and modify (update operations) the same item. If one transaction writes the changed item to the shared store before it is read by the second, then we should notify the second transaction about the conflict. The typical solution involves atomic RMW operations such as load-link and store-conditional (LL/SC) [11]. LL/SC is the pair of instructions that reads a value (load-link) and allows update the latter atomically only if it has not changed in the meantime (store-conditional). Actually, it is a key operation for conflict detection in transactions. If one of the LL/SC operations fails, the respective data item has been modified.

Note, the implementation for LL/LC depends on the hardware (CPU). E.g., all of Alpha, PowerPC, MIPS, and ARM provide LL/SC instructions.

```
public class AtomicIntegerTest{
AtomicInteger counter= new
AtomicInteger(10);
class AddThread implements
Runnable{
@Override
public void run() {
//ads the 5 in current value
counter.addAndGet(5); } }
```

Code snippet 4. Atomic operations

Technically, the following atomic operations are mentioned in the computer science papers:

- Atomic read-write
- Atomic swap
- Test-and-set
- Fetch-and-add
- Compare-and-swap
- Load-Link/Store-Conditional

These instructions operations can be used directly by compilers and operating systems. Also, they can be abstracted exposed as libraries (packages) in higher-level languages.

Technically, when there are multiple instructions which must be completed without interruption, we can use a CPU instruction which temporarily disables interrupts. So, potentially, we can achieve the atomicity of any sequence of instructions by disabling interrupts while executing it. However, it means the monopolization of the CPU and may lead to hang and infinite loops. Also, in multiprocessor systems, it is usually impossible to disable interrupts on all processors at the same time. The compare-and-swap pattern, described below, allows any processor to atomically test and modify a memory location, preventing such multiple-processor collisions. It is one of the reasons for its popularity.

3 Lock-free programming patterns

In this section, we would like to discuss so-called Compare-And-Swap operations. Technically, it is also atomic operation (as it is described in section 2). As we have mentioned in section 2, the implementation for atomic operations depends on the hardware. So, depending on the hardware platform, there are two main atomic operations hardware implementations: the above-mentioned Load-Link/Store-Conditional (LL/SC) on Alpha, PowerPC, MIPS, ARM and Compare-And-Swap (CAS) on x86 line [12].

The above-mentioned Load-Link/Store-Conditional uses the following pattern: the application reads the data with load-link, computes a new value to write, and writes it with store-conditional. If the value has changed concurrently, the store-conditional operation will fail.

CAS compares a memory location with a given value, and if they are the same the new value is set. The returned value from CAS operation is the value before the swap was attempted.

In the both cases, we can know if the memory location was written to between our read and write. This conclusion is leading to the common programming pattern, used here. It is so-called read-modify CAS sequence (CAS loop): the application reads the data, computes a new value to write, and writes it with a CAS. If the value changes concurrently, the CAS writing will fails then the

application tries again (repeats the loop). Let us see this Java code:

```
AtomicReference<Object> cache =
new AtomicReference<Object> ();
Object cachedValue =
new Object();
cache.set(cachedValue);
```

Code snippet 5. Atomic operations

AtomicReference is an object reference that may be updated atomically. Later in the code, we can calculate a new value and conditionally update (Code snippet 6)

```
Object cachedValueToUpdate = someFunction(cachedValue);
boolean success = cache.compareAndSet(cachedValue,cachedValueToUpdate);
```

Code snippet 6. CAS operation

Another example is a well-known Singleton pattern. Here is the classical implementation (Code snippet 7)

```
private Singleton singleton = null;
protected Singleton createSingleton()
{ synchronized (this)
{ // locking on 'this' for simplicity
if (singleton == null) { singleton = new Singleton(); }
return singleton; }}
```

Code snippet 7. Classical singleton

This classical model contains a synchronized block. And the code blow presents the lock-free implementation for singleton pattern. It uses the same *AtomicReference* object. The method *weakCompareAndSet* atomically reads and conditionally writes a variable, but does not create any happens-before orderings. This method provides no guarantees with respect to previous or subsequent reads and writes of any variables other than the target of the *weakCompareAndSet.* It is illustrated in Code snippet 8.

```
private AtomicReference singletonRef = new AtomicReference(null);
protected Singleton createSingleton()
{
   Singleton singleton = singletonRef.get();
   if (singleton == null)
   {
        singleton = new Singleton();
        if (!singletonRef.weakCompareAndSet(null, singleton))
        {
            singleton = singletonRef.get();
        }
    }
   return singleton; }
```

Code snippet 8. Lock-free singleton

In order to estimate the performance gain in lock versus lock-free implementations, we have followed to the test suite, provided by M.Thompson [13]. It is a concurrent test for reading/writing x-y coordinates for some moving object. Figure 2 [13] illustrates the performance for 2 readers – 2 writers test case. In our own tests, it shows the following approximate performance for reading (in thousands reads/sec):

Synchronized	3100
ReadWriteLock	2700
ReentrantLock	5900
LockFree	21000

The performance gain in writing is less, but still is significant (in thousands writes/sec):

Synchronized	3300
ReadWriteLock	4500
ReentrantLock	4600
LockFree	9500

The average performance gains (in 20 tests, Java language) for lock-free vs. traditional (in percents) are:

Reading:

Synchronized	650%
ReadWriteLock	760%
ReentrantLock	320%

Writing:

Synchronized	280%
ReadWriteLock	210%
ReentrantLock	205%

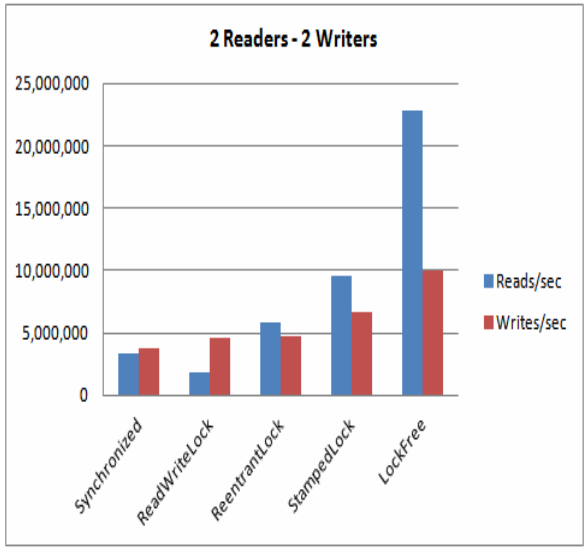

Fig. 2. Lock vs. lock-free performance

The common problem for CAS-based solutions is so-called ABA problem [14]. The definition for ABA is very simple. Between the time that our code reads the original value and try to replace it with the new value, it could have been changed to something else and back to the original value. Obviously, in this case, we cannot detect the intermediate change. The simplest solution to this problem is to associate the counter with our values. This counter could be incremented at each operation. So, we can compare not only values but counters also. The counter will be changed in case of intermediate operations.

4 Lock-free programming use cases

Usually, in computer science papers, non-blocking synchronization and non-blocking operations are divided for three primary classes. They have own set of warranties for non-blocking actions:

• Obstruction Freedom [15]. The application (algorithm) provides for any single-thread progress guarantees in the absence of conflicting operations. It is the weakest requirement. An obstruction-free operation does not need to wait for actions by other threads, regardless of where they have stopped or not.

• Lock Freedom [16]. The application provides system-wide progress warranties. At least one active invocation of an operation is guaranteed to complete in a finite number of steps. Lock-free progress combines obstruction-free progress with live-lock freedom [17].

• Wait Freedom [18]. The application supports per-operation progress warranties. Every active invocation of an operation completes in a finite number of steps. It combines lock-free restrictions with starvation-freedom [19]. So, any wait-free operation is guaranteed to complete in a finite number of its own steps, regardless of the actions or inaction of other operations.

Conventional locks are neither non-blocking nor wait-free. Any wait-free algorithm is also lock-free and obstruction-free. Any lock-free algorithm is also obstruction-free. A non-blocking data structures are structures with operations that satisfy the progress warranties in the above-mentioned non-blocking hierarchy. But there are two important remarks. It is allowed to implement a data structure that provides a non-blocking status for a limited number of readers or writers only. And secondly, it is possible also for a data structure to provide different non-blocking warranties for the different sets of operations [20].

The typical use case for non-blocking solutions is quite obvious. We have multithreading system and some of the threads have wait for actions (data) by other threads. Of course, any such solution (design) makes the whole system potentially vulnerable to deadlocks, live-locks, or simply to long delays. Examples of such uses are [21]:

• Kill-safe systems. Such systems should stay available even if processes may be terminated at arbitrary points. The typical example is a server-side multi-user framework. Any process could be killed by the user. And, of course, any process could be terminated while it is operating on shared structures

or services. So, if other processes are waiting for some actions from the terminated process (they are blocked), it will never happen. It means non-blocking operations are the only possible solution for this case.

• Asynchronous signal safety. The handlers of asynchronous signals should be able to share data with the interrupted threads and never wait for actions by the interrupted thread. In other words, interrupted threads should not stop signals handlers. Asynchronous handlers should be safe.

• Priority inversion and preemption tolerance. The active thread should not be blocked awaiting action by other threads that are delayed for a long time. It is very important for stream processing, for example.

In the connection with non-blocking operations, we would like to mention three key elements. At the first hand, it is automatic code generation. Technically, a code for non-blocking algorithms is more complex. That is why it is very important to simplify the development. Plus, automatic code generation allows some formal methods for verification. We discuss this stuff in the connection with IoT programming in our papers [22, 23].

Traditionally, much attention is paid to the so-called non-blocking data structures [24, 25]. The practical design of non-blocking data structures may be different. One of the widely used approaches can present non-blocking data elements as shared data structures that support linearization of concurrent operations as the main correctness invariant. They support seemingly instantaneous execution of every operation on the shared data, so that an equivalent sequential ordering of the concurrent operations can always be constructed [26]. In the parallel (concurrent) programming developers will obviously the increased use of data and resource sharing for utilizing. It is what parallelism is about. The whole applications are split into subtasks. And subtasks (processes) share data. So, data structures are crucially important for the performance.

Classical (standard) implementations of data structures are based on locks in order to avoid inconsistency of the shared data due to possible concurrent modifications. Of course, it reduces the possibility for parallelism and leads to possible deadlocks. So, lock-free implementations of data structures should support concurrent access. It means that all steps of the supported operations for a lock-free structure can be executed concurrently. It means they should employ an optimistic conflict control approach and allow multiple processes to access the shared data object at the same time. The delays should be suffered only when there is an actual conflict between operations. This conflict can cause some of the parallel operations to retry. Of course, such architecture increases the scalability.
According to the above-mentioned definitions, an implementation of a data structure is lock-free if it allows multiple processes to gain access to the data structure concurrently and guarantees that at least one of those processes finishes in a finite number of its own steps regardless s of the state of the other processes. A consistency requirement for lock-free data structures could be described via linearizability [27]. It means ensures that each operation on the data appears to take effect instantaneously during its actual duration. The effect of all operations is consistent with the object's sequential specification [28].

Methods for implementing the non-blocking data structures vary depending on the type of structure. For example, let us see non-blocking stack implementations. Stack (queue) is one of the simplest sequential data structures. The obvious implementation for the stack is a sequentially linked list with a pointer to the top and a global lock for access control to the stack. The minimal lock-free concurrent stack implementation [29] includes a singly-linked list, a pointer to the top, but uses CAS to modify the top pointer atomically. But this lock-free solution does not solve the scalability issues, because the top pointer is still a bottleneck. So, the real implementation will include more sophisticated code. E.g., it could be so-called elimination technique [30]. It introduces pairs of operations with reverse semantics (push and pop for the stack), which complete without any central coordination. The idea is that if a pop operation can find a concurrent push operation to associate with, then this pop operation can use the value from the detected push. In other words, both operations eliminate each other's effect. So, both operations can return values immediately.

This last statement actually leads to the next important conclusion. The most important elements for concurrent programming are the coordination and data flow organizing, rather than relatively low-level data structures. The various elements of concurrent programming are used together in real applications. It is the biggest issue. Each individual element (the individual abstraction) solves

effectively own tasks only. But it does not solve the complexity problem for the whole application. So, by our opinion, the proper solution here should be based on the frameworks [31] and programming languages especially oriented to concurrent programming [32].

For example, languages like Erlang have built-in concurrency features. In Erlang, concurrency is achieved through Message passing and the Actor Model [33]. On the top level, process mailbox in Erlang is lock-free. Although, a locking scheme can be used internally by the virtual machine [34]. Concurrent Haskell [35] uses so-called Software Transactional Memory (STM) [36].

STM is a low-level application programming interface for synchronizing shared data without locks [37]. It uses the standard definition for transactions – it is a sequence of steps executed by a single thread. Transactions are atomic. So, each transaction either completes all steps or aborts completely (all step/changes should be discarded). And transactions are linearizable (see above the remarks about consistency checking for lock-free data structures). Transactions take effect in one at a time order. So, STM API lets applications mark the sequence of operations on possibly shared data as a transaction. This mark is just a pair of calls: start transaction and commit transaction. If the commit succeeds, the transaction's operations take effect, otherwise, they are discarded. Originally this conception was proposed for hardware architecture and later extended as a software API. It has several implementations (including Open Source) for various programming systems (languages). STM enables to compose scalable applications. It is, probably, the biggest advantage. The performance of STM applications depends on several factors. One of them, by our experience, is a size for transactions. But it is a directly managed parameter and it could be easily changed in the source code.

5 Conclusion

It this paper, we discuss lock-free programming. The aim of the study was the choice of the methods of work with memory (memory management) in a parallel implementation of a virtual machine. Lock-free programming provides a model to share changing data among several threads without paying the cost of acquiring and releasing locks. We provide a survey of lock-free data structures and algorithms, lock-free programming patterns and approaches. Also, we estimate the potential performance gain without the cost of locks. It lets us estimate the applicable overhead for multi-threading support. The most promising lock-free programming pattern, by our opinion, is STM.

References:

[1] Szenasi, S., "Difficulties and Solutions in the Field of Teaching Parallel Programming", MAFIOK 2013, Miscolc, 26-28 Aug. 2013, pp.1-6, ISBN 978-963-358-037-0.

[2] Lee, Edward A. "The problem with threads." Computer 39.5 (2006): 33-42

[3] Andrews, Gregory R. Concurrent programming: principles and practice. Benjamin/Cummings Publishing Company, 1991.

[4] van den Brink B. Providing an Efficient Lockless Hash Table for Multi-Core Reachability in Java. – 2014.

[5] Ravindran, B., Jensen, E. D., & Li, P. (2005, May). On recent advances in time/utility function real-time scheduling and resource management. In Object-Oriented Real-Time Distributed Computing, 2005. ISORC 2005. Eighth IEEE International Symposium on (pp. 55-60). IEEE.

[6] Kogan, A., & Petrank, E. (2012, February). A methodology for creating fast wait-free data structures. In ACM SIGPLAN Notices (Vol. 47, No. 8, pp. 141-150). ACM.

[7] Prokopec, A., Miller, H., Schlatter, T., Haller, P., & Odersky, M. (2013). FlowPools: A lock-free deterministic concurrent dataflow abstraction. In Languages and Compilers for Parallel Computing (pp. 158-173). Springer Berlin Heidelberg.

[8] Atomic vs. Non-Atomic Operations http://preshing.com/20130618/atomic-vs-non-atomic-operations/ Retrieved: Jan, 2015.

[9] Abadi, M., Flanagan, C., & Freund, S. N. (2006). Types for safe locking: Static race detection for Java. ACM Transactions on Programming Languages and Systems (TOPLAS), 28(2), 207-255.

[10] Loesing, S., Pilman, M., Etter, T., & Kossmann, D. (2013). On the Design and Scalability of Distributed Shared-Memory Databases.

[11] E. Jensen, G. Hagensen, and J. Broughton. A new approach to exclusive data access in shared memory multiprocessors. Technical Report UCRL-97663, 1987

[12] Petrović, D., Ropars, T., & Schiper, A. (2014, February). Leveraging hardware message passing for efficient thread synchronization. In Proceedings of the 19th ACM SIGPLAN

symposium on Principles and practice of parallel programming (pp. 143-154). ACM.
[13] Lock-Based vs. Lock-Free Concurrent Algorithms http://mechanical-sympathy.blogspot.ru/2013/08/lock-based-vs-lock-free-concurrent.html Retrieved: Jan, 2015
[14] Michael, M. M. (2003). CAS-based lock-free algorithm for shared deques. In Euro-Par 2003 Parallel Processing (pp. 651-660). Springer Berlin Heidelberg.
[15] Herlihy, M., Luchangco, V., & Moir, M. (2003, May). Obstruction-free synchronization: Double-ended queues as an example. In Distributed Computing Systems, 2003. Proceedings. 23rd International Conference on (pp. 522-529). IEEE.
[16] Fraser, K. (2004). Practical lock-freedom (Doctoral dissertation, University of Cambridge).
[17] John D. Valois. Lock-Free Linked Lists Using Compare-and-Swap. In Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing, pp. 214–222, August 1995.
[18] Barbera, M. V., Epasto, A., Mei, A., Perta, V. C., & Stefa, J. (2013, October). Signals from the crowd: uncovering social relationships through smartphone probes. In Proceedings of the 2013 conference on Internet measurement conference (pp. 265-276). ACM.
[19] Rajwar, R., & Goodman, J. R. (2002). Transactional lock-free execution of lock-based programs. ACM SIGOPS Operating Systems Review, 36(5), 5-17.
[20] Herlihy, M., Luchangco, V., Moir, M., & Scherer III, W. N. (2003, July). Software transactional memory for dynamic-sized data structures. In Proceedings of the twenty-second annual symposium on Principles of distributed computing (pp. 92-101). ACM.
[21] Michael, M. M. (2013). The balancing act of choosing nonblocking features. Communications of the ACM, 56(9), 46-53.
[22] Namiot, D., & Sneps-Sneppe, M. (2014). On IoT Programming. International Journal of Open Information Technologies, 2(10), 25-28.
[23] Namiot, D., & Sneps-Sneppe, M. (2014). On M2M Software. International Journal of Open Information Technologies, 2(6), 29-36.
[24] Shavit, N. (2011). Data structures in the multicore age. Communications of the ACM, 54(3), 76-84.
[25] Cederman, D., Gidenstam, A., Ha, P., Sundell, H., Papatriantafilou, M., & Tsigas, P. (2013). Lock-free concurrent data structures. arXiv preprint arXiv:1302.2757.
[26] Li, X., An, H., Liu, G., Han, W., Xu, M., Zhou, W., & Li, Q. (2011, May). A Non-blocking Programming Framework for Pipeline Application on Multi-core Platform. In Parallel and Distributed Processing with Applications (ISPA), 2011 IEEE 9th International Symposium on (pp. 25-30). IEEE.
[27] Herlihy, Maurice, and Nir Shavit. "The art of multiprocessor programming." PODC. Vol. 6. 2006.
[28] M. Herlihy and J. Wing. “Linearizability: a Correctness Condition for Concurrent Objects.” ACM Transactions on Programming Languages and Systems, 12(3): 463–492, 1990
[29] Kaur, Ranjeet, and Pushpa Rani Suri. "Concurrent Access Algorithms for Different Data Structures: A Research Review." Global Journal of Computer Science and Technology 14.3 (2014).
[30] Michael, M. M., & Scott, M. L. (1996, May). Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing (pp. 267-275). ACM.
[31] N. Shavit and D. Touitou. Elimination trees and the construction of pools and stacks. Theory of Computing Systems, 30:645–670, 1997
[32] Ricci, Alessandro, Mirko Viroli, and Giulio Piancastelli. "simpA: An agent-oriented approach for programming concurrent applications on top of Java." Science of Computer Programming 76.1 (2011): 37-62.
[33] Feo, John T., ed. A comparative study of parallel programming languages: the Salishan problems. Elsevier, 2014.
[34] Armstrong J. Programming Erlang: software for a concurrent world. – Pragmatic Bookshelf, 2007.
[35] Jones, S. P., Gordon, A., & Finne, S. (1996, January). Concurrent haskell. In POPL (Vol. 96, pp. 295-308).
[36] Di Sanzo, P., Ciciani, B., et.al. (2012). On the analytical modeling of concurrency control algorithms for software transactional memories: the case of commit-time-locking. Performance evaluation 69(5), 187-205.
[37] Dragojevic, A., Harris, T., (2012, April). STM in the small: trading generality for performance in software transactional memory. In Proceedings of the 7th ACM European Conference on Computer Systems (pp. 1-14). ACM.