Theoretical Computer Science 677 (2017) 41–55

Contents lists available at ScienceDirect

Theoretical Computer Science

www.elsevier.com/locate/tcs

Contention-sensitive data structures and algorithms ☆

Gadi Taubenfeld

The Interdisciplinary Center, P.O.Box 167, Herzliya 46150, Israel

A R T I C L E   I N F O

Article history:
Received 30 August 2015
Received in revised form 15 March 2017
Accepted 23 March 2017
Available online 25 March 2017
Communicated by D. Peleg

Keywords:
Contention-sensitive
Interference
Synchronization
Locks
Shortcut code
Disable-free
Prevention-free
Livelock
Starvation
k-obstruction-free
Wait-free

A B S T R A C T

A contention-sensitive data structure is a concurrent data structure in which the overhead introduced by locking is eliminated in common cases, when there is no contention, or when processes with non-interfering operations access it concurrently. When a process invokes an operation on a contention-sensitive data structure, in the absence of contention or interference, the process must be able to complete its operation in a small number of steps and without using locks. Using locks is permitted only when there is interference. We formally define the notion of contention-sensitive data structures, propose four general transformations that facilitate devising such data structures, and illustrate the benefits of the approach by implementing a contention-sensitive consensus algorithm, a contention-sensitive double-ended queue data structure, and a contention-sensitive election algorithm.

1. Introduction

1.1. Motivation

Concurrent access to a data structure shared among several processes must be synchronized in order to avoid interference between conflicting operations. Mutual exclusion locks are the de facto mechanism for concurrency control on concurrent data structures: a process accesses the data structure only inside a critical section code, within which the process is guaranteed exclusive access. Any sequential data structure can be easily made concurrent using such a locking approach. The popularity of this approach is largely due to the apparently simple programming model of such locks, and the availability of lock implementations which are reasonably efficient.

When using locks, the *granularity* of synchronization is important. Using a single lock to protect the whole data structure, allowing only one process at a time to access it, is an example of *coarse-grained* synchronization. In contrast, *fine-grained* synchronization enables to lock "small pieces" of a data structure, allowing several processes with non-interfering operations to access it concurrently. Coarse-grained synchronization is easier to program but is less efficient compared to fine-grained synchronization.

☆ A preliminary version of the results presented in this paper, has appeared in *Proceedings of the 23rd International Symposium on Distributed Computing* (DISC 2009), Elche, Spain, September 2009.

E-mail address: tgadi@idc.ac.il.

http://dx.doi.org/10.1016/j.tcs.2017.03.017
0304-3975/© 2017 Elsevier B.V. All rights reserved.

Using locks may, in various scenarios, degrade the performance of concurrent applications, as it enforces processes to wait for a lock to be released. Moreover, slow or stopped processes may prevent other processes from ever accessing the data structure. Locks can introduce false conflicts, as different processes with non-interfering operations contend for the same lock, only to end up accessing disjoint data.

A promising approach is the design of concurrent data structures and algorithms which avoid locking. The advantages of such algorithms are that they are not subject to priority inversion, they are resilient to failures, and they do not suffer significant performance degradation from scheduling preemption, page faults or cache misses. On the other hand, such algorithms may impose too much overhead upon the implementation and are often complex.

We propose an intermediate approach for the design of concurrent data structures, which incorporates ideas from the work on data structures which avoid locking. While the approach guarantees the correctness and fairness of a concurrent data structure under all possible scenarios, it is especially efficient in common cases when there is no (or low) contention, or when processes with non-interfering operations access a data structure concurrently.

1.2. Contention-sensitive data structures: the basic idea

Contention for accessing a shared object is usually rare in well designed systems. Contention occurs when multiple processes try to acquire a lock at the same time. Hence, a desired property in a lock implementation is that, in the absence of contention, a process can acquire the lock extremely fast, without unnecessary delays. Furthermore, such fast implementations decrease the possibility that processes which invoke operations on the same data structure in about the same time but not simultaneously, will interfere with each other. However, locks were introduced in the first place to resolve conflicts when there is contention, and acquiring a lock *always* introduces some overhead, even in the cases where there is no contention or interference.

We propose an approach which, in common cases, eliminates the overhead involved in acquiring a lock. The idea is simple: assume that, for a given data structure, it is known that in the absence of contention or interference it takes some fixed number of steps, say at most 10 steps, to complete an operation, not counting the steps involved in acquiring and releasing the lock. According to our approach, when a process invokes an operation on a given data structure, it first tries to complete its operation, by executing a short code, called the *shortcut code*, which does not involve locking. Only if it does not manage to complete the operation fast enough, i.e., within 10 steps, it tries to access the data structure via locking. The shortcut code is required to be *wait-free*. That is, its execution by a process takes only a finite number of steps and always terminates, regardless of the behavior of the other processes.

Using an efficient shortcut code, although eliminates the overhead introduced by locking in common cases, introduces a major problem: we can no longer use a sequential data structure as the basic building block, as done when using the traditional locking approach. The reason is simple, many processes may access the same data structure simultaneously by executing the shortcut code. Furthermore, even when a process acquires the lock, it is no longer guaranteed to have exclusive access, as another process may access the same data structure simultaneously by executing the shortcut code.

Thus, a central question which we are facing is: if a sequential data structure cannot be used as the basic building block for a general technique for constructing a contention-sensitive data structure, then what is the best data structure to use? Before we proceed to discuss formal definitions and general techniques, which will also help us answering the above question, we demonstrate the idea of using a shortcut code to avoid locking – in the absence of synchronization conflicts – by presenting a contention-sensitive solution to the binary consensus problem using atomic read/write registers and a single lock.

1.3. A simple example: contention-sensitive consensus

The *consensus problem* is to design an algorithm in which all correct processes reach a common decision based on their initial opinions. A consensus algorithm is an algorithm that produces such an agreement. While various decision rules can be considered such as "majority consensus", the problem is interesting even where the decision value is constrained only when all processes are unanimous in their opinions, in which case the decision value must be the common opinion. A consensus algorithm is called *binary* consensus when the number of possible initial opinions is two.

Processes are not required to participate in the algorithm, however, once a process starts participating it is guaranteed that it may fail only while executing the shortcut code. The algorithm uses an array $x[0..1]$ of two atomic bits, and two atomic registers y and *out*. After a process executes a **decide**() statement, it immediately terminates.

CONTENTION-SENSITIVE BINARY CONSENSUS: program for process p_i with input $in_i \in \{0, 1\}$.

```
shared  $x[0..1]$ : array of two atomic bits, initially both 0
        $y, out$ : atomic registers which range over $\{\perp, 0, 1\}$, initially both $\perp$

1 $x[in_i] := 1$                                                    // start shortcut code
2 if $y = \perp$ then $y := in_i$ fi
3 if $x[1 - in_i] = 0$ then $out := in_i$; decide($in_i$) fi
4 if $out \neq \perp$ then decide($out$) fi                          // end shortcut code
5 $\boxed{\text{lock}}$ if $out = \perp$ then $out := y$ fi $\boxed{\text{unlock}}$ ; decide($out$)   // locking
```