Information Systems 57 (2016) 69–74

Contents lists available at ScienceDirect

Information Systems

journal homepage: www.elsevier.com/locate/infosys

A simple deterministic algorithm for guaranteeing the forward progress of transactions ☆

Charles E. Leiserson

MIT Computer Science and Artificial Intelligence Laboratory, 32 Vassar Street, Cambridge, MA 02139, United States

ARTICLE INFO

Article history:
Received 13 October 2015
Accepted 28 October 2015
Available online 21 December 2015

Keywords:
Algorithm
Contention manager
Deadlock
Forward progress
Mutual exclusion
Ownership array
Proof
Synchronization
Transaction

ABSTRACT

This paper describes a remarkably simple deterministic (not probabilistic) contention-management algorithm for guaranteeing the forward progress of transactions — avoiding deadlocks, livelocks, and other anomalies. The transactions must be finite (no infinite loops), but on each restart, a transaction may access different shared-memory locations. The algorithm supports irrevocable transactions as long as the transaction satisfies a simple ordering constraint. In particular, a transaction that accesses only one shared-memory location is never aborted. The algorithm is suitable for both hardware and software transactional-memory systems. It also can be used in some contexts as a locking protocol for implementing transactions "by hand."

1. Introduction

Transactional memory [14,10,25,19,24,9] has been proposed as a general and flexible way to allow programs to read and modify disparate shared-memory locations atomically. The basic idea of transactional memory rests on ***transactions*** [5,16], which offer a method for providing mutual synchronization without the protocol intricacies of conventional synchronization methods, such as ***locking*** or ***nonblocking synchronization***. Many textbooks on concurrency (e.g., [11,26,23]) treat the basics of synchronization methods, including transactional memory.

A ***transaction*** is a delimited sequence of instructions performed as part of a program. If a transaction ***commits***, then all its instructions appear to have run atomically with respect to other transactions, that is, they do not appear to have interleaved with the instructions of other transactions. If a transaction ***aborts***, then none of its stores take effect, and the transaction may be restarted from its first instruction as if it had never been run. From the programmer's perspective, all that needs to be specified is where a transaction begins and where it ends, and the transactional support, whether in hardware or software, handles all the complexities.

Under the covers of a transactional-memory system is a collection of mechanisms, implemented in hardware or software, which perform basic bookkeeping for the transaction. For example, the system must have some means to detect when two concurrent transactions ***conflict***: both transactions access the same shared-memory location, and one of them attempts to modify the location. ***Read/write sets*** of shared-memory addresses accessed by the transaction must be maintained, so that the transaction can be ***rolled back*** if it is aborted or committed when it completes. These particular mechanisms are amply described in the literature (see, for example [11]), and are not the focus of this paper.

This paper focuses on another under-the-covers mechanism dubbed the ***contention manager*** [12], which ensures that transactions complete. A contention manager

☆ This research was supported in part by NSF Grant 1314547.
E-mail address: cel@mit.edu

http://dx.doi.org/10.1016/j.is.2015.10.013
0306-4379/

can be viewed as a distributed program with a module in each transaction. The modules coordinate to ensure forward progress, typically using mutual-exclusion locks, nonblocking synchronization, and other hardware support. When two transactions conflict, the contention manager chooses whether one of the transactions should abort or whether one transaction should wait for the other so that the two transactions appear to execute in a serial order. The contention manager ensures that the system does not ***deadlock***, where transactions are caught in a cycle of waiting and cannot progress. The contention manager ensures that the system does not ***livelock***, where transactions are repeatedly aborted and restarted without making progress. In short, the contention manager guarantees that transactions make forward progress, ideally with as little overhead as possible.

The literature is replete with contention-management schemes, many of which can be quite complex. (See [22,23,17,26] for overviews.) Some contention-management strategies employ probabilistic backoff, where an aborting transaction progressively delays its restart by increasing amounts to avoid livelock. Other contention managers use timestamps to ensure that the "oldest" transaction makes progress when a conflict occurs [6,3,21]. Some contention managers abort whichever of two conflicting transactions has a smaller read/write set in order to minimize the wasted work. Heuristic strategies abound, many of which — as a last resort to guarantee forward progress if some problematic transaction aborts frequently enough — grab a global lock and execute all transactions serially, even transactions that are completely independent of the problematic one.

This paper describes a simple contention-management algorithm, called Algorithm L, which guarantees forward progress. Before a transaction accesses a shared-memory location, Algorithm L checks whether the access is safe, that is, no other transaction conflicts, which makes the algorithm ***eager***, ***pessimistic***, or ***conservative***, in the varied parlance of the concurrency literature, as opposed to ***lazy*** or ***optimistic***. (See [23] for a taxonomy of contention-management strategies.) Although a transaction may abort, it always completes in a bounded number of retries. Algorithm L is deterministic and contains no probabilistic elements, such as backoff. The algorithm can be adapted for either hardware or software implementation.

The remainder of this paper is organized as follows. Section 2 presents Algorithm L, and Section 3 briefly argues its correctness. Section 4 provides a short discussion of ramifications, and Section 5 concludes by surveying antecedents in the literature.

2. Algorithm L

This section describes Algorithm L. The algorithm employs a finite ***ownership array*** [7] $lock[0..n-1]$ of locks, which is a global array accessible by all the transactions. Typically, a contention manager of this nature needs reader/writer locks, not just mutual-exclusion locks (mutexes), but since this issue can be readily handled at the cost of some additional complexity, let us assume for simplicity that the locks are mutexes. It is important for the guarantee of forward progress, however, that the locks be antistarvation (e.g., queuing). A simple spin-lock will not do. A good discussion of locking alternatives can be found in [18].

Before accessing a shared-memory location x, a transaction must acquire the lock in the ownership array associated with x. An arbitrary many-to-one ***owner*** function $h: U \to \{0, 1, \ldots, n-1\}$ maps the set U of all shared-memory locations to one of the n slots in the ownership array. (All transactions must agree on the same owner function h.) To acquire the lock associated with x, the transaction may perform one of two operations:

- ACQUIRE ($lock[h(x)]$), which blocks on the lock acquisition until the lock becomes free.
- TRY – ACQUIRE ($lock[h(x)]$), which either successfully acquires the lock and returns the Boolean TRUE, or fails and returns FALSE.

The finite ownership array introduces the possibility of a ***false conflict***, where two transactions accessing different locations conflict by requiring the same lock, when they would not have conflicted had the locks been on the locations themselves. The larger the size n of the ownership array, the less the chance of a false conflict. On the other hand, larger values for n lead to weaker bounds on the number of restarts a transaction might endure before it completes.

Pseudocode for Algorithm L is shown in Fig. 1. Each transaction maintains its own local set L of lock indexes, which starts out as the empty set $\varnothing$. Whenever the transaction encounters a new shared-memory location x, it greedily attempts to acquire $lock[h(x)]$ and add $h(x)$ to L. Specifically, it performs one of the following two actions:

(A) If $h(x)$ is smaller than the largest value in L, the transaction aborts if the $lock[h(x)]$ is held by another transaction.
(B) If $h(x)$ is larger than the largest value in L, the transaction blocks if $lock[h(x)]$ is taken. Once the transaction acquires the lock, it performs the access of x.

If an abort occurs, the transaction rolls back its transactional state and releases all locks with indexes larger than $h(x)$. It then acquires $lock[h(x)]$ and reacquires in increasing order all the locks it previously held, blocking along the way if any of these locks is taken. The algorithm then restarts the transaction, which once again attempts to acquire any additional locks it needs greedily as it encounters them.

3. Correctness

This section shows that Algorithm L avoids deadlock and guarantees forward progress.

Lemma 1. *Transactions do not deadlock.*

Proof. The locks in the ownership array are linearly ordered [8,1], and a transaction blocks on acquiring a lock