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

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 [9, 10, 14, 19, 24, 25] 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., [11,23,26]) 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 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 [17, 22, 23, 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 [3, 6, 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 back-

This research was support in part by NSF Grant 1314547.

```
SAFE-ACCESS(x, L)
 1  if h(x) ∈ L
 2      // do nothing
 3  else
 4      M = {i ∈ L : i > h(x)}
 5      L = L ∪ {h(x)}
 6      if M == ∅
 7          ACQUIRE(lock[h(x)]) // blocking
 8      elseif TRY-ACQUIRE(lock[h(x)]) // nonblocking
 9          // do nothing
10      else
11          roll back transaction state (without releasing locks)
12          for i ∈ M
13              RELEASE(lock[i])
14          ACQUIRE(lock[h(x)]) // blocking
15          for all i ∈ M in increasing order
16              ACQUIRE(lock[i]) // blocking
17          restart transaction // does not return
18  access location x
```

Figure 1: Algorithm L, which safely accesses a memory location x within a transaction with local lock-index set L. Each element of the global ownership array $lock[0..n-1]$ contains an antistarvation (e.g., queuing) lock. The owner function $h : U \to \{0,1,\ldots,n-1\}$ maps the space U of all shared-memory locations to indexes in the ownership array *lock*. At transaction start, the transaction's lock-index set L is initialized to the empty set: $L = \emptyset$. When the transaction completes, all locks with indexes in L are released.

off. 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 Figure 1. Each transaction maintains its own local set L of lock indexes, which starts out as the empty set $\emptyset$. 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 [1, 8], and a transaction blocks on acquiring a lock only if it does not hold any higher-indexed locks. The proof, therefore, can follow the standard proof that acquiring locks in order cannot produce a deadlock. For deadlock to occur, there must be a cycle of transactions, each waiting for a lock that another holds. Suppose for the purpose of contradiction that there is such a cycle, and consider the transaction on the cycle that holds the largest indexed lock. Since Algorithm L only blocks when acquiring locks that are larger than any held lock (see lines 7, 14, and 16), this transaction cannot be waiting. Contradiction. □

THEOREM 2. *Every transaction makes forward progress.*

PROOF. Assume that transactions are finite, i.e., no infinite loops. Consider the set L of lock indexes for a transaction at the various times immediately before the transaction restarts. The transaction must eventually be able to acquire the lock with index $h(x)$ and the other locks with indexes in M, because transactions do not deadlock (Lemma 1) and we have assumed that the locks are antistarvation. Consequently, all locks with indexes in L are held by the transaction immediately before the transaction restarts. With each abort, at least one more lock index is added to L before the transaction restarts, namely $h(x)$. Since the ownership array contains only a finite number n of locks, after at most n starts ($n-1$ restarts), the transaction must complete. □

Theorem 2 holds even if on each restart, the transaction can access different shared-memory locations. If a transaction is accesses the same shared-memory locations every time it restarts, a somewhat tighter bound can be obtained. Specifically, the transaction

must complete after executing at most $\min\{n, m-1\}$ times, where m is the total number of distinct locks the transaction must acquire, which is at most the number of distinct shared-memory locations it accesses.

4. DISCUSSION

In practice, a contention-manager must cope with a multitude of system concerns not immediately addressed by Algorithm L. Because of the algorithm's simplicity, it should be possible to adapt it to address many of these concerns. This section discusses the ramifications of Algorithm L, many of which call for future research.

Algorithm L requires neither a global lock nor a backoff strategy to ensure forward progress. A global lock may cause the system to degrade poorly, because all transactions must serialize, even if they are independent. In contrast, Algorithm L degrades gracefully in that two transactions cannot delay one another if there is no path between them in the conflict graph in which two transactions have an edge between them if there exists a lock that they both need to acquire. Because the algorithm is deterministic, requiring no randomization as with backoff methods, it provides a solid guarantee of forward progress.

How big should the ownership array be? Ideally, one would like the chance of false conflicts to be small so that transactions that have nothing to do with each other execute without interaction. The larger the size n of the ownership array, the less the chance that two transactions will conflict, if the owner function h is chosen as a random hash function. Due to the birthday paradox [2, Sec. 5.4], if the total number of shared-memory locations in all concurrently running transactions is m, the expected number of false conflicts is at most 1 if $n = m^2/2$, in which case the impact of the conflict should be negligible. Empirical measurements of conflicts for finite ownership arrays have been studied in [28], but it would be desirable to characterize the impact of ownership-array size theoretically.

Irrevocable transactions [27] are transactions whose side effects, such as I/O and system calls, cannot be rolled back. Algorithm L can support irrevocable transactions if the shared-memory locations are known at the start of the transaction. The transaction sorts the lock indexes of the shared-memory locations and acquires the corresponding locks in order before starting the transaction. This and other advantages of using locks to support transactions are discussed in [4].

It may make sense to wait competitively [13] before aborting a transaction. The idea is that the rollback and lock reacquisition may take some time. Waiting a proportion of that time before giving up on acquiring a lock does not affect overall performance by more than a constant factor if the lock is not soon released. If the lock is released soon, performance can improve dramatically, especially since if the owner function h is a random hash function, the more locks a transaction acquires, the more likely it is that it must perform an aborting acquire instead of a blocking acquire.

The extension to reader/writer locks is straightforward, and for the most part, it follows the same logic as the mutex-based algorithm. When a writer wishes to acquire a lock that it already holds as a reader, however, it must attempt to reacquire the lock in writer mode, aborting if there is a conflict with other readers. When the lock is acquired in writer mode during restart, it is acquired with blocking. The reader/writer lock implementation must ensure that this writer cannot be starved by subsequent readers in order to guarantee forward progress. A nice overview of the different kinds of locks and their properties can be found in [18].

If the compiler understands Algorithm L, it seems there are many opportunities for optimization. For example, if the compiler can analyze a block of transactional code and determine what shared-memory locations it will access, it can sort the corresponding locks in advance of executing the code. If the transaction happens to abort, all the locks can be acquired in the reacquisition phase, rather than just the lock that caused the abort. This optimization would avoid the possibility of aborting over and over, each time on a different lock acquisition.

To support Algorithm L in hardware, the L1-caches can be used to implement the ownership array. The owner function h can map each address to the cache line in L1. The protocol can now piggyback on the cache-coherence mechanism, much as in Herlihy and Moss's original proposal [10]. The more difficult issues in a hardware implementation would seem to be ensuring the queuing behavior for acquisition of reader/writer locks and dealing with issues such as page faults, context switches, and the like. But these are exactly the issues any transactional system works out with more complicated contention managers, and it seems like a good research project to see how they would play out with Algorithm L.

Algorithm L can be used as a locking methodology outside of a transactional-memory context to implement transactions "by hand." For example, many parallel graph algorithms operate atomically on a vertex and all its neighbors by acquiring locks associated with each vertex before performing the operation. By using a finite ownership array for locks, rather than a lock in each vertex, Algorithm L can ensure forward progress even if the graph structure changes from one lock-acquisition attempt to the next. This strategy should work particular well for graphs with bounded degree, since the chance of aborting increases with the number of held locks.

As described in Section 2, the ordering of locks is static. It may be possible for the transactions themselves to define the locking order dynamically, such as in tree locking [26, Sec. 4.3.7]. This strategy may lead to fewer aborts, since a transaction can often define a needed lock to be larger in the linear order as long as no cycles are created, allowing it to be acquired with blocking instead of abort. A downside of this dynamic approach is that it seems to compromise support for irrevocable transactions that access more than one shared-memory location.

An advantage of Algorithm L is that a transactions aborts itself "synchronously" rather than being aborted "asynchronously" by another transaction. Asynchronous aborts are generally harder to manage, because the transaction must always be ready to be aborted, regardless of whether its internal state is consistent. In contrast, synchronous aborts allow a transaction to perform the abort at specific times in the code when its state is consistent. An advantage of asynchronous aborts is that they may be needed them anyway to protect against transactions with infinite loops or large finite delays.

Certainly, there is plenty of room for more research as this simple theoretical algorithm finds its way into practice.

5. RELATED WORK

Algorithm L has roots in prior work. The idea of releasing and reacquiring locks (sometimes termed "resources") has surfaced sporadically in the literature since the early studies of concurrency. The idea of an ownership array is not new. In a sense, the contribution of this paper is simply to recognize that these two ideas can be combined to ensure forward progress. This section outlines this prior work. It also briefly compares Algorithm L with timestamp-based algorithms, a popular way of guaranteeing forward progress.

Havender mentioned a release-reacquire strategy in his seminal 1968 paper [8] as "Approach 3." He does not explicitly mention the idea of reacquiring the locks in a linear order on conflict, even though that is "Approach 1," settling instead for "Approach 2": obtaining the locks collectively, which he explains can be done by acquiring a global lock to ensure that collective lock acquisition is

atomic.

Lampson produced notes [15] for his MIT class 6.826 on operating systems in 1995 on operating systems which includes in-order reacquisition after releasing them due to a conflict. The notes say, "The generic solution is to collect the information you need, one mutex at a time. Then reacquire the locks in a standard order, check that things haven't changed (or that your conclusions still hold), and do the updates. If it doesn't work out, retry. Version number make the 'didn't change' check cheap."

In 2006, Riegel, Felber, and Fetzer [20] make the most explicit reference to a release-and-reacquire-in-order strategy in the context of a more elaborate contention-manager: "When a transaction fails to commit at the end of the optimistic phase, the read/write sets are first sorted deterministically using a total order relation (e.g., the address of the objects in memory). Then, before restarting its execution, the contention manager iterates over all objects in the sorted set and "opens" each object in write (read) mode if it belongs to the write (read) set." The release-and-reacquire-in-order idea is not by itself, however, sufficient to guarantee forward progress. Indeed, Riegel, Felber, and Fetzer later say, "A transaction can thus abort more than once, but it will succeed as soon as the read/write sets stabilize, i.e., do not change between two consecutive executions. Given that we compute the read/write sets as the union of the objects accessed during previous executions, we should quickly obtain a superset of the objects actually accessed by the current execution and hence commit." If there were an infinite number of locations, however, a finite transaction might yet never complete, because each time it is restarted, it might take a completely different code path and acquire a completely different finite set of locks. That is, the read/write sets might not stabilize. To guarantee forward progress, a finite ownership array ensures that every finite transaction completes, because eventually the transaction acquires all the locks, and nothing can block its progress.

The idea of an owner function h and a finite ownership array is mentioned in the literature in 2003 by Harris and Fraser [7]. They do not relate the finiteness of the ownership array to the problem of guaranteeing forward progress, however. Database systems also map object identifiers into a shared data structure of bounded size, usually a hash table of so-called lock control blocks [26, Sec. 10.2].

Timestamp-based algorithms [3,6,21] are a popular way to guarantee forward progress, but these contention-management algorithms tend toward the complex. For example, they require synchronization to ensure that timestamps are distinct and properly ordered. "Wound-wait" algorithms [21] require that a mechanism be provided for one transaction to abort another asynchronously. In this sense, Algorithm L is more like a "wait-die" algorithm [21] in which a transaction only ever needs to abort itself synchronously. Moreover, supporting irrevocable transactions is more complex with timestamp-based algorithms, especially to support multiple irrevocable transactions. Algorithm L supports multiple irrevocable transactions simply by ordering the accesses.

Timestamp-based algorithms may be the preferred way to implement transactions while guaranteeing forward progress, because modern techniques, especially TL2 [3], reduce the chance of restarting considerably. Indeed, Guerraoui, Herlihy, and Pochon [6] have devised a wound-wait scheme with strong bounds on completion. A concern for Algorithm L is that with each lock a transaction acquires, the chances increase that a restart is necessary, assuming that the locks are randomly ordered. From this point of view, Algorithm L may be most suitable for implementing transactions "by hand" as a locking protocol in situations where each transaction (critical region) only accesses a handful of objects.

6. ACKNOWLEDGMENTS

Many thanks to the participants of Dagstuhl Seminar 15021 *Concurrent Computing in the Many-Core Era*, which was held January 4–9, 2015, which is where and when I conceived the algorithm. In particular, thanks to Hans Boehm (Google), Sebastian Burckhardt (Microsoft), Dave Dice (Oracle), Stephan Diestelhorst (ARM), Pascal Felber (University of Neuchatel), Maurice Herlihy (Brown University), Alexander Matveev (MIT), Eliot Moss (University of Massachusetts), Torvald Riegel (Red Hat), Sven-Bodo Scholz (Heriot-Watt University), Michael Scott (University of Rochester), Nir Shavit (MIT), Michael Swift (University of Wisconsin), and Martin T. Vechev (ETH Zurich) for their extensive feedback during the Seminar as I bent their ears attempting to articulate the algorithm. José Nelson Amaral (University of Alberta), David F. Bacon (Google), Daniele Bonetta (Oracle Labs), Antony Hosking (Purdue University), Milind Kulkarni (Purdue University), Viktor Leis (Technical University of Munich), Yossi Lev (Oracle), Maged M. Michael (IBM), Paolo Romano (University of Lisbon), and Sven-Bodo Scholz (Heriot-Watt University) also provided constructive comments during the Seminar.

Thanks to Matteo Frigo (Amazon Web Services), Bradley Kuszmaul (MIT), and Alex Matveev (MIT) for their ideas to simplify the exposition of the algorithm. Thanks to Phil Bernstein (Microsoft), Butler Lampson (Microsoft/MIT), Sam Madden (MIT), Michael Scott (University of Rochester), Gottfried Vossen (University of Muenster), and Gerhard Weikum (Max Planck Institute for Informatics, Saarbrueken) for confirming, to the best of their knowledge, the novelty of the algorithm. Thanks to Pascal Felber (University of Neuchatel), Butler Lampson (Microsoft/MIT), and Michael Swift (University of Wisconsin) for helping to identify relevant prior work. Thanks to Will Hasenplaugh (MIT), Bradley Kuszmaul (MIT), and Yuan Tang (Fudan University) for helpful comments.

7. REFERENCES

[1] E. G. Coffman, M. Elphick, and A. Shoshani. System deadlocks. *Computing Surveys*, 3(2):67–78, 1971.

[2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, third edition, 2009.

[3] Dave Dice, Ori Shalev, and Nir Shavit. Transactional locking II. In Shlomi Dolev, editor, *Distributed Computing*, volume 4167 of *Lecture Notes in Computer Science*, pages 194–208. Springer Berlin Heidelberg, 2006.

[4] Dave Dice and Nir Shavit. TLRW: Return of the read-write lock. In *SPAA*, pages 284–293. ACM, 2010.

[5] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[6] Rachid Guerraoui, Maurice Herlihy, and Bastian Pochon. Toward a theory of transactional contention managers. In *PODC*, pages 258–264. ACM, 2005.

[7] Tim Harris and Keir Fraser. Language support for lightweight transactions. In *OOPSLA*, pages 388–402, Anaheim, California, October 2003.

[8] J.W. Havender. Avoiding deadlock in multitasking systems. *IBM Systems Journal*, 7(2):74–84, 1968.

[9] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer, III. Software transactional memory for dynamic-sized data structures. In *PODC '03*, pages 92–101, Boston, Massachusetts, July 2003.

[10] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *ISCA 20*, pages 289–300, San Diego, California, May 1993.

[11] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008.

[12] Maurice P. Herlihy, Victor Luchangco, and Mark Moir.

Obstruction-free synchronization: Double-ended queues as an example. In *ICDCS*, pages 522–529, Providence, Rhode Island, May 2003.

[13] A.R. Karlin, M.S. Manasse, L.A. McGeoch, and S. Owicki. Competitive randomized algorithms for nonuniform problems. *Algorithmica*, 11(6):542–571, 1994.

[14] Tom Knight. An architecture for mostly functional languages. In *LFP*, pages 105–112. ACM Press, 1986.

[15] Butler Lampson. Practical concurrency. Handout 21a for 6.826 Principles of Computer Systems, MIT, 1995.

[16] Nancy Lynch, Michael Merritt, William Weihl, and Alan Fekete. *Atomic Transactions*. Morgan Kaufmann, San Mateo, CA, 1994.

[17] Virendra J. Marathe and Michael L. Scott. A qualitative survey of modern software transactional memory systems. Technical Report TR 839, Department of Computer Science, University of Rochester, 2004.

[18] Paul E. McKenney. Selecting locking primitives for parallel programming. *CACM*, 39(10):75–82, 1996.

[19] Ravi Rajwar and James R. Goodman. Transactional lock-free execution of lock-based programs. In *ASPLOS-X*, pages 5–17, San Jose, California, October 5–9 2002.

[20] Torvald Riegel, Pascal Felber, and Christof Fetzer. Stateful contention management for software transactional memory: Learning from failure. Technical Report RR-I-06-05.2, Université de Neuchâtel Institut d'Informatique, 2006.

[21] Daniel J. Rosenkrantz, Richard E. Stearns, and Philip M. Lewis, II. System level concurrency control for distributed database systems. *ACM Trans. Database Syst.*, 3(2):178–198, 1978.

[22] William N. Scherer, III and Michael L. Scott. Advanced contention management for dynamic software transactional memory. In *PODC*, pages 240–248. ACM, 2005.

[23] Michael L. Scott. *Shared-Memory Synchronization*. Morgan & Claypool, 2013.

[24] Nir Shavit and Dan Touitou. Software transactional memory. In *PODC '95*, pages 204–213, Ottawa, Ontario, Canada, August 1995.

[25] Janice M. Stone, Harold S. Stone, Philip Heidelberg, and John Turek. Multiple reservations and the Oklahoma update. *IEEE Parallel and Distributed Technology*, 1(4):58–71, November 1993.

[26] Gerhard Weikum and Gottfried Vossen. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann, 2002.

[27] Adam Welc, Bratin Saha, and Ali-Reza Adl-Tabatabai. Irrevocable transactions and their applications. In *SPAA*, pages 285–296. ACM, 2008.

[28] Craig Zilles and Ravi Rajwar. Brief announcement: Transactional memory and the birthday paradox. In *SPAA*, pages 303–304. ACM, 2007.