

Turbocharging Boosted Transactions or: How I Learnt to Stop Worrying and Love Longer Transactions

Chinmay Kulkarni

Birla Institute of Technology and
Science, Pilani, India
chinmay007@gmail.com

Osman Unsal

Adrián Cristal

Barcelona Supercomputing
Center-Centro Nacional de
Supercomputación
{osman.unsal, adrian.cristal}@bsc.es

Eduard Ayguadé

Mateo Valero

Barcelona Supercomputing Center and
Technical University of Catalunya
{eduard.ayguade, mateo.valero}
@bsc.es

Abstract

Boosted transactions offer an attractive method that enables programmers to create larger transactions that scale well and offer deadlock-free guarantees. However, as boosted transactions get larger, they become more susceptible to conflicts and aborts. We describe a linear-time algorithm to detect transactions that cannot make progress, *which* transactions need to be aborted, and *when*. The algorithm guarantees zero false positives with minimal aborts. Our proposals, as implemented in DSTM2, increase the transactional throughput of the system, often by more than 30%.

Categories and Subject Descriptors D [1]: 3—Concurrent Programming— Parallel Programming; D [4]: 1—Process Management— Synchronization; Threads; Concurrency

General Terms Algorithms, Performance

Keywords Concurrency, parallel programming, transactional memory, deadlocks, deadlock-detection

1. Introduction

Software Transactional Memory (STM) has emerged as an alternative to locking and monitors which scale poorly and require programmer effort to ensure deadlock freedom [1].

Boosted transactions are a special class of transactions that offer better performance (vis-a-vis “non-boosted” STMs) for certain kinds¹ of operations[2]. The power, and the limitations, of boosted transactions stem from the fact that the runtime system maintains a log, not of memory accesses, but of *commutative, reversible operations* that transactions perform.

This method of logging significantly reduces the overhead, so that in the absence of aborts, transactional operations are nearly as fast as non-transactional code. Boosting thus allows programmers to write longer transactions than was previously possible, without creating a performance bottleneck, as long as aborts are rare.

¹ Informally, boosting applies to operations whose inverses are known.

However, when a transaction does abort, it affects the overall performance of the system very adversely. Prior work has focused on reducing this overhead somewhat; for example, using checkpointing or nested transactions [3]. However, little attention has been paid to *which* boosted transaction needs to be aborted and *when*.

Through this current work, we make the following two contributions:

- We make a precise characterization of the nature of conflicts in a boosted transactional system and provide a simple criterion to determine when it is inevitable to abort.
- We describe a simple, linear-time algorithm that embodies this theoretical criterion with minimal aborts

Our results show that with our proposed algorithms, it is possible to obtain a significantly higher throughput (over twice as high in some cases), with a corresponding decrease in the number of transactions aborted (by up to 30%).

2. Background and Theory

Boosted transactions apply only to “commutative” operations (such that the state of the system is identical regardless of the order in which operations are performed).

In practice, when a transaction wants to perform a transactional operation, it first acquires the *abstract lock* associated with the operation; to ensure that no other transaction can perform a non-commutative operation. These locks are “abstract”, because they do not guard access to data, but are instead acting as a test for commutativity. If an operation A commutes with an operation B ($A \leftrightarrow B$), then the inverse operation A^{-1} also commutes with B (Proof available in [2]). Therefore, a single lock is sufficient both for an operation and its inverse.

Transactions cannot release locks while running before a commit or abort, leading to two-phase locking. This is because operations which do not commute with the given operation could be then performed by other threads (before the commit), or the transaction may need to undo this operation in case of an abort. For the remainder of this paper, we consider a *conflict* as an event in which a transaction trying to acquire an abstract lock is not successful; and an *aborting condition* as a state of the system where one or more transactions need to be aborted in order that transactions that are not aborted will not conflict anymore. (We show below that the two definitions are not necessarily equivalent.)

In the timeout-based contention management as implemented in current boosted transactional frameworks, all conflicts that fail to resolve in a pre-determined time are treated as an abortable. Setting

the timeout is tricky since it is impossible to predict in general how long other transactions might take to complete (even at runtime). However, using a timeout is entirely unnecessary if we only abort transactions that are deadlocked (other conflicts will eventually resolve). The general problem of detecting deadlocks is hard; and involves performing checks periodically (such as “probing”).

3. A practical linear-time algorithm

For boosted transactions, deadlock detection is easier because cycles, if any, present in a wait-for graph are non-intersecting, as at any point, a transaction can only wait on no more than one lock. Below, we describe an algorithm (adapted from [4]) that finds such cycles. Due to its distributed nature, this algorithm can be executed in parallel on all conflicting threads.

We associate public and private “colors” with each running transaction. In addition, we define a total order among colors. *Private* colors are unique and non-decreasing with time. Initially, the public color, L_i is set to the private color H_i .

The algorithm allows us to determine a victim transaction in a cycle based on a policy such as timestamps, operations performed etc.. To do this, we use another public label— like the color, the priority P_i has a hidden counterpart Q_i .

The first time a transaction fails to acquire a given lock, it performs a *Block* operation. If it fails subsequently, it performs either a *Transmit* action repeatedly, until it either acquires the lock, or performs the *Detect* action and aborts.

In the state diagrams that follow², each transaction is represented by a circle, the upper and lower left, and upper and lower right quadrants show the public and private color, and the public and private priority, respectively. An arrow represents the “wait-for” relation between two transactions.

- **Block** The process acknowledges a *block*. If transaction T_i with public color L_i , is waiting on T_j with public color L_j , T_i sets L_i to $inc(L_i, L_j)$, which is a value greater than both L_i and L_j . It also sets its private color H_i to L_i , and updates its public priority $P_i = Q_i$ (Q_i is a private priority, which in our case counts the number of operations performed) See figure 1 below.

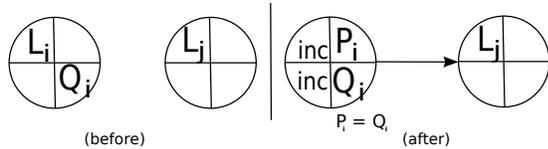


Figure 1. The Block Step (T_i starts to wait on T_j .)

- **Transmit** If $L_j > L_i$ or $L_i = L_j$, $P_j < P_i$, T_i sets L_i to L_j , and P_i to $min(P_i, P_j)$. (Figure 2)

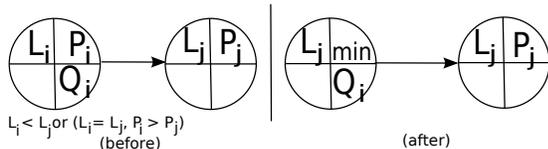


Figure 2. The Transmit Step

- **Detect** If $L_j = L_i$, and $P_j = Q_i$, the transaction T_i detects itself as the deadlock victim, and aborts. (Figure 3)

A maximum of $N - 1$ operations will carry the largest public color across the cycle, and at worst $N - 1$ more cycles are required to propagate the lowest priority, giving us the linear time guarantee.

² We retain Mitchell and Merritt’s representation

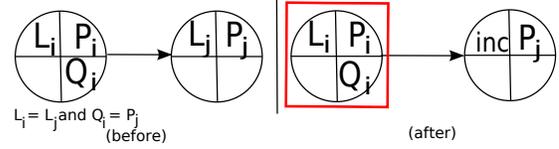


Figure 3. The Detect Step

4. Evaluation and results

We wrote a synthetic benchmark that deliberately considers long transactions, with high possibilities of conflict. We use a Skiplist-based Set (that contains unique elements) and insert or remove a set of elements from it. We show high and low contention cases (low having half the conflicts). With 8 threads on an Intel Xeon (32GB RAM), we are able to get approximately a 30% improvement.

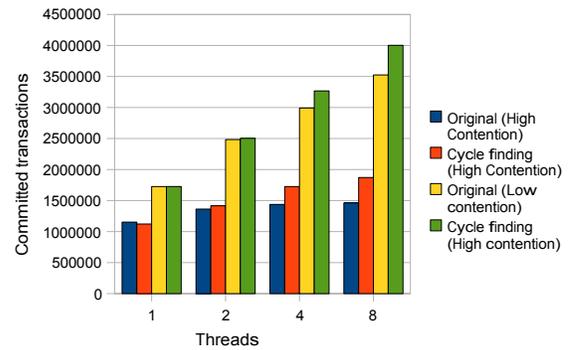


Figure 4. Number of committed transactions vs. Thread count

5. Conclusions and future work

In this paper, we propose a linear-time conflict management algorithm to minimize aborts in a boosted transactional system. Preliminary results show promise. We are also working on finding approximate solutions for cases where a zero-false-negative guarantee is not required. Future work will include further testing of this and the existing algorithm, as well as exploring the possibility of partially rolling back deadlocked transactions.

This work has been partially supported by the Ministry of Education of Spain under contract TIN2007-60625 and the European Commission in the contexts of the HiPEAC network of excellence and the VELOX FP7 STREP (216852) project. Chinmay’s summer internship supported by the Barcelona Supercomputing Center.

References

- [1] T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy. Composable memory transactions. *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 48–60, 2005.
- [2] M. Herlihy and E. Koskinen. Transactional boosting: a methodology for highly-concurrent transactional objects. *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 207–216, 2008.
- [3] E. Koskinen and M. Herlihy. Checkpoints and continuations instead of nested transactions. *Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, pages 160–168, 2008.
- [4] D. Mitchell and M. Merritt. A distributed algorithm for deadlock detection and resolution. *Proceedings of the third annual ACM symposium on Principles of distributed computing*, pages 282–284, 1984.