Transactional Memory Contention Managers: Analysis and Comparison

Rafael Dias Pedrosa Universidade Federal de Pernambuco rdp@cin.ufpe.br

Wellington Oliveira Universidade Federal de Pernambuco woj@cin.ufpe.br

ABSTRACT. Over the years, parallel programming is gaining utmost importance, whether in the area of energy consumption or in other areas like high performance computing. In this context, several techniques have been developed to take advantage of the parallelism among the cores of the processors, such as the use of transactional memory. The contention manager is the judge who validates a transaction within the transaction model. There are several types of contention managers, each implementing a different solution to validate the conflicting transactions.

Objective: This is a review article demonstrating the different types of existing contention managers. Examples and illustration are provided to explain how they function. The paper aims to serve as a guide for those interested in concurrent programming, focusing on transactional memory.

Method: Several articles on transactional memory have been read, focusing particularly on contention manager. Since the topic has received scholarly attention in recent years, there are many recent articles on transactional memory and contention managers.

Result: A structure identifying the main factors of each of the various types of contention managers, as well as a comparison between them, facilitating consultation and research for those interested.

Conclusion: This article contains information about eleven kinds of contentions managers and demonstrates succinctly and objectively how they work.

Keywords: Concurrent Computing, Transactional Memory, Contention Managers, Software Transactional Memory.

1 INTRODUCTION

Parallel programming is a way to share the work that will be performed on smaller processes to ensure better performance of the program in question. When two or more threads are performing work sharing memory address or variables, we call this concurrent programming and ensure the proper functioning of a concurrent system can be very hard [14]. However, there are several techniques [12] to solve the concurrent problems, each with advantages and disadvantages. Unfortunately, none of them is perfect, after all, a solution might address a problem but not another. Among those solutions, this article focuses on transactional memory, more precisely, in the component that deal with the conflict between transactions, the contention manager.

The article aims to analyze the various existing solutions for contention management encompassing: a) objective; b) succinct explanations; and c) figures to facilitate understanding. This analysis will serve as a review for those interested in the subject.

The paper is divided into:

- CONCEPTUAL BACKGROUND: with a short summary of transactional memory and an explanation of each one of the strategies used by contention managers;
- RESEARCH METHOD: demonstrates how the research was developed.
- RESULTS: a comparative table of the different solutions, classifying them according to their overall performance in several benchmarks
- CONCLUDING REMARKS: general remarks on the development of the article and the authors' conclusions
- ACKNOWLEDGMENTS;
- REFERENCES:

2 CONCEPTUAL BACKGROUND

Because contention managers are an important component of the transactional memory model, will be given a brief overview of this solution. Reminding that although they are part of the same model, the implementation of the contention manager is decoupled from the implementation of transactional memory. Changes in implementation of contention manager can be made without changing the implementation of transactional memory with very different results.

Transactional Memory

There are two types of transactional memory, the hardware transactional memory (HTM) and software transactional memory (STM); this article has focused only STM. Transactional memory is an alternative to traditional treatment of conflicts based on locks, is a non-blocking, optimistic solution. It assumes that conflicts will occur and adopts a measure to deal with similar transactions in the database. Important to mention that the transactions of transactional memory are not subject to deadlocks and convoying and also they are compositional and obstruction-free.

A transaction can be in one of three states: ACTIVE, ABORTED or COMMITED. Each state represents the current status of the transaction. ACTIVE is a transaction which is being executed normally. If it is properly finished, then the transaction will be COMMITED if there is a conflict, then one of two conflicting transactions shall receive the ABORTED status.

To resolve a conflict between two, the contention manager is used. It is a component which, according to a preset strategy, in case of conflict selects which of the two transactions is completed successfully and which will be aborted. There are several ways to implement the contention manager, both taking into account whether it will be implemented at the application level or kernel [17].

One may adopt several strategies to prioritize one or other transaction [1] [2] [3] [9] [6]. Next, eleven policy of containment will be described. Studies show that a better solution would be to use not only one of the following solutions, but several, modifying its implementation according to the need of the program [16].

Aggressive

This type of policy always aborts the conflicting transaction in case of dispute. It does not seem to have a good use, as it is very prone to livelock, however, it exists and can be used depending on the type of problem, or in comparison experiments [1] [2] [9].

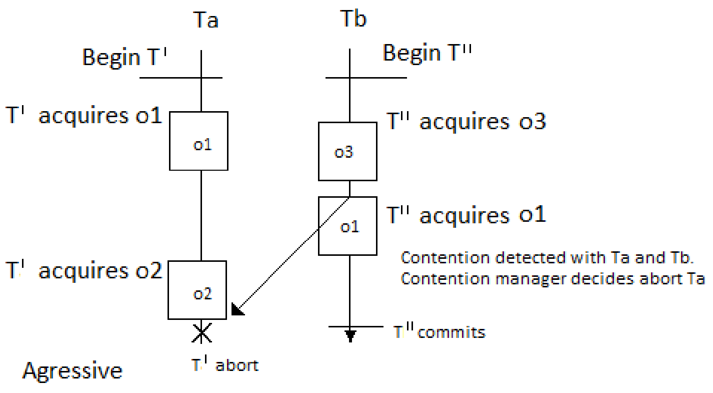

Figure 3. Illustration of Agressive strategy.

Polite or Backoff

This policy uses exponential backoff, waiting for time to resolve the conflicts that grow every time there is contention. This waiting time is proportional to 2n, where n is the number of attempts made. The thread is still trying to complete their transaction until you reach a certain maximum number of attempts (in Figures 3 and 4, eight attempts).

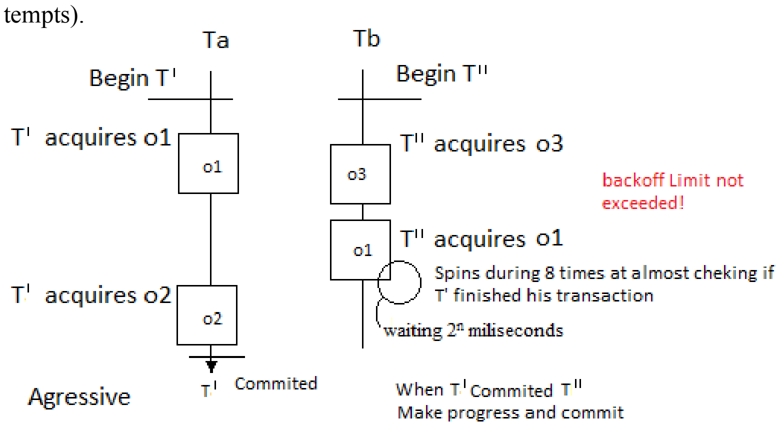
Ta
Tb
Begin T'
Begin T''
T' acquires o1
o1
T'' acquires o3
o3
backoff Limit not exceeded!
o1
T'' acquires o1
Spins during 8 times at almost cheking if T' finished his transaction
T' acquires o2
o2
waiting 2^n miliseconds
Agressive
T' Commited
When T' Commited T'' Make progress and commit

Figure 2. Illustration of Polite strategy.

In other words, when T 'achieves the maximum amount of attempts and T' attempts to abort it, T will be aborted so that T' can end the process. This solution may lose performance due to not existing preemption and page faults.

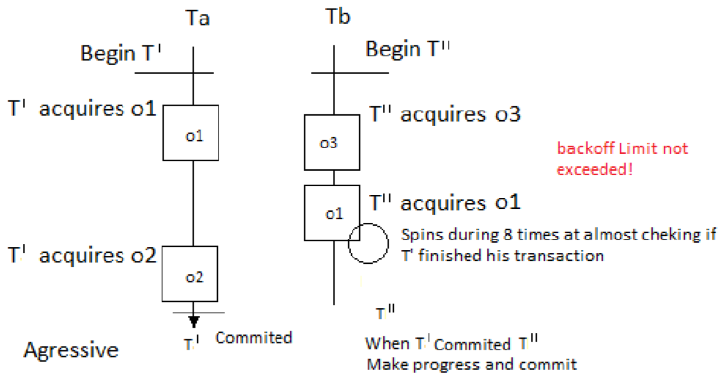
Ta
Tb
Begin T'
Begin T''
T' acquires o1
o1
T'' acquires o3
o3
backoff Limit not exceeded!
o1
T'' acquires o1
Spins during 8 times at almost cheking if T' finished his transaction
T' acquires o2
o2
T''
Agressive
T' Commited
When T' Commited T'' Make progress and commit

Figure 3. Illustration of Polite strategy.

Random

Performs a "heads or tails" deciding whether the conflicting transaction will wait a random time interval (determined by the developer) or be aborted.

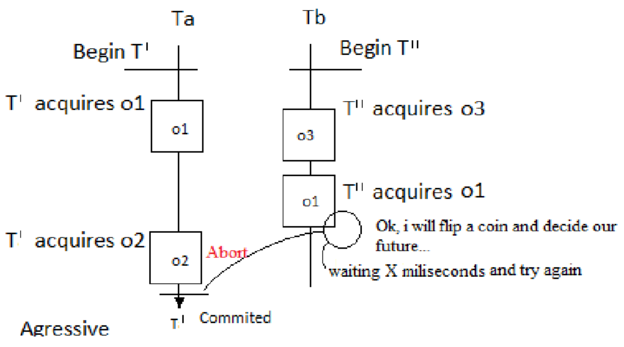

Figure 3. Illustration of Random strategy.

Karma

Karma manager is a bit more elaborate than the previous. It judges the conflicting threads comparing the amount of work done by each. Although it is not trivial to estimate the work done by the transaction, several techniques can be used to estimate work

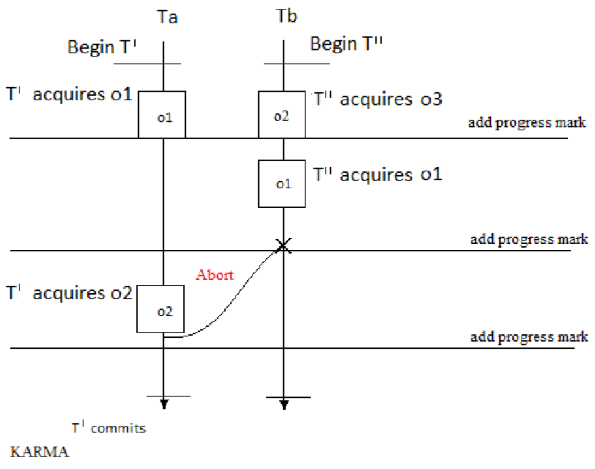

Figure 5. Illustration of Karma strategy.

such as: the number of transactions previously open to finish one transaction or the number of transactions opened by the thread executing the transaction.

If a thread T conflicts with T', the less advanced will be aborted, giving preference to T'. If the conflict comes from T, the manager performs a backoff and tries again until the number of attempts made is higher than the priority of T, then aborts T'.

If one thread perform the commit, its priority resets, if the thread is spinning, it's priority increases to gain priority. However, this alternative is not enough, because if a thread T has a higher priority than T', T' will abort T, but can be aborted by another thread T" which has a higher karma and will not guarantee starvation-free. For this reason, the implementation should not reset when aborted, however, should add current priority with past and save the amount of attempts. Comparing the priorities of conflicting transactions, it is possible to obtain a fairer policy for transactions that were aborted able to perform the commit.

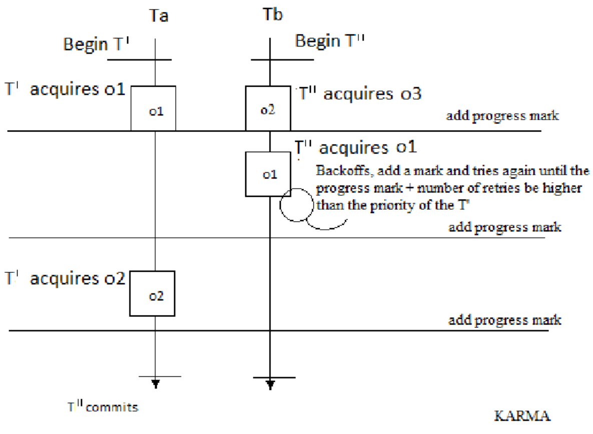

Figure 6. Illustration of Karma strategy.

Eruption

It is much like the karma manager, as both create an estimate through points in the code. The big difference is that if two threads are in conflict, the thread with the highest priority will win another point in their estimate and the other will take a backoff.

That is, if T identify that T' has more priority than it, so T adds its priority to T'. This ensures that if a resource is being widely used among threads, the more advanced will gain priority to finish faster fails to lock the resource for other threads, then releasing it. It is possible that many transactions increase the priority of a conflicting transaction, then it can be aborted. This possibility can not be ruled out, therefore, every time a transaction is aborted his priority is halved, preventing aborted transactions to accumulate and aborting other transactions that have high priority.

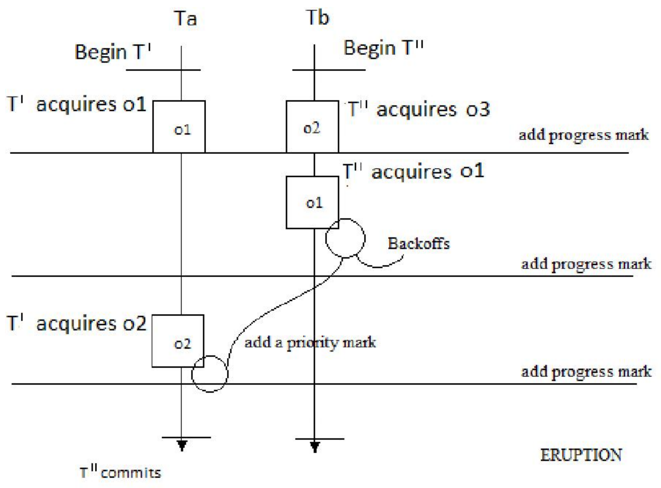

Figure 7. Illustration of Eruption strategy.

Kindergarten

Kindergarten manager handles transactions accessing a blacklist. When there is conflict, the manager aborts the competitor transaction if it is already in this list. Otherwise, the manager puts the competitor on the list and awaits for a limited number of intervals, giving opportunity to other transaction (the most advanced) to complete the process.

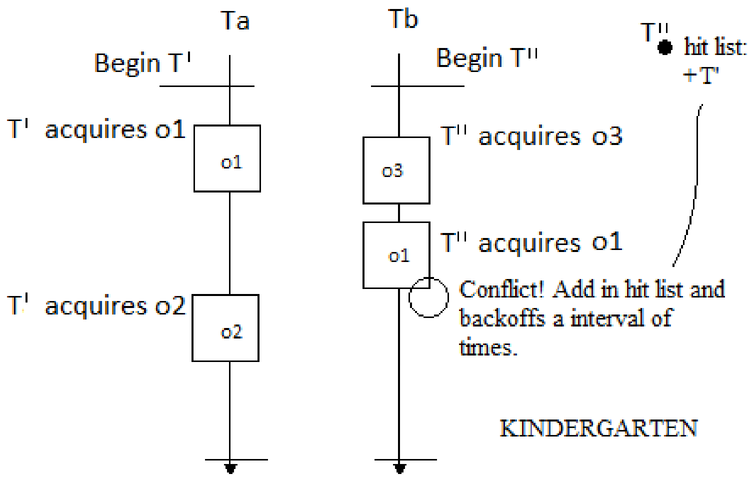

Figure 8. Illustration of Kindergarten strategy.

In each interval, we see the conflicting transaction ended to continue. If the conflicting transaction is not completed, the manager aborts T' forcing it to start over, but if T' is not finished yet, then T' will be aborted, giving the conflicting transaction, T’’, an opportunity to finish.

Timestamp

The policy of this manager aims to be fair to the transactions. This is the default type used in database transactions. The time is stamped at the beginning of each transaction.

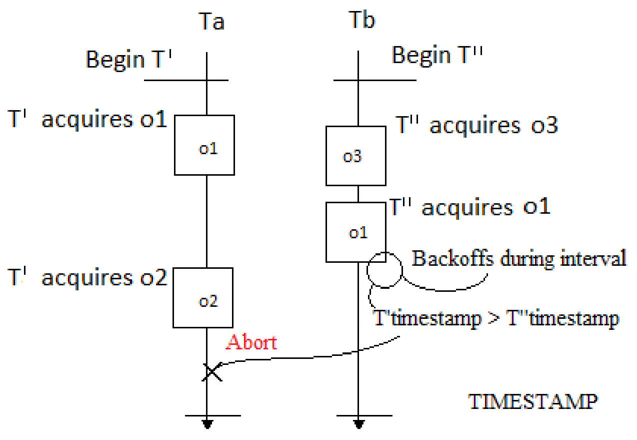

Figure 9. Illustration of TimeStamp strategy.

In case of contention, is comparing the timestamps of threads T’ and T’’ if T’’ began after T’, the manager aborts T’’. Otherwise, it waits for a given time interval.

If the number of unsuccessful attempts T’’ reach half of the maximum number of retries determined, transaction T’ conflict flag is set to a "zombie", if T’’ reach the maximum number of retries and the flag T’ is set, then T’’ aborts T’. During this process, if carried out any operation in T’, the defunct flag is cleared. This flag serves as a feedback mechanism to distinguish one transaction active transaction from a dead stop for long.

KillBlocked

KillBlocked manager is not as complex as karma or eruption, and has a rapid elimination of cyclic block. The manager marks the transaction as locked when not successful in acquiring an object. In contention, the transaction aborts the conflicting transaction every time: the conflicting transaction is also blocked or if the maximum backoff time is reached.

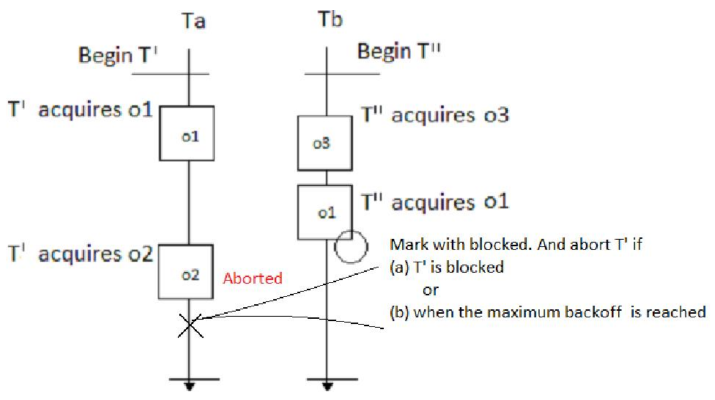

Figure 10. Illustration of KillBlocked strategy.

QueueOnBlock

This manager when there is contention T'' if the link to a list transasaction T'. After this it keep spinnig by a flag "finish" that is eventually set by the confliting transaction. Alternatively, if the transaction is waiting for a long time, it aborts the enemy and continues.

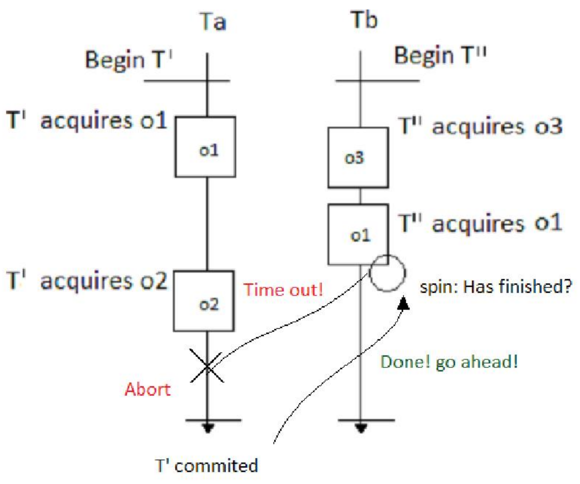

Figure 11. Illustration of QueueOnBlock strategy.

Without it, it would not be possible to guarantee obstruction-freedom. In this implementation, when two transactions are competing neither wins until timeout reaches its maximum.

Certainly, the manager does not have an effective cycle, after a transaction has to go into timeout to ensure progress. However, without this dependency would be impossible to abort the conflicting transaction.

Polka

In the article [2] was realized that the Polite and Karma managers had slightly better performance than other managers. And in [1] was designed to the junction of backoff karma and policies, using their best resources, would bring improvements to the benchmarks. In other words, polka was the name of the new manager increases the priority of the transaction as the Karma, and in the case of restraint, priorities and attempts are summed and compared as Polite.

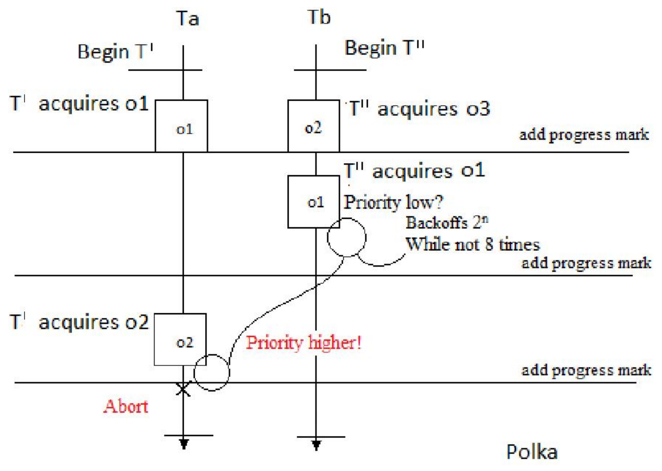

Figure 12. Illustration of Polka strategy.

If the priority of conflicting thread is larger and the number of times this thread has not reached its maximum, then this conflicting transaction expects a 2n milliseconds exponentially, as the in the polite contention manager, otherwise, it will be aborted.

3 RESEARCH METHOD

This article intends to be a review of contention manager solutions. No new data has been added to this article.

Threats to Validity

The conclusions of this article are based on the results and conclusions of articles and books listed on the references. Any threat to validity of these may be considered a threat to the validity of this article.

Other contention manager policies may exist, but, as all information was taken from the references, any other policy that were not in the scope of the references will not be described here.

4 RESULTS

According to the analysis of benchmarks [1] [2] [9] was possible to develop a ranking for the various strategies used in implementing contention manager. It is worth noting that during the development of this article, we did not implement or test the analysis was done according to the reference articles. But we can show a summary of the related work and discuss about it.

Managers	IntSet	IntSetUpgrade	RBTreeTMNodeStyle	Stack	ArrayCounter	LFUCache
Karma	-	X	X	-	-	-
Eruption	-	X	X	-	-	-
Polite	-	X	-	-	-	-
Kindergarten	-	X	X	-	-	-
Timestamp	-	-	-	-	-	-
Polka	X	X	X	X	X	X

Table 1. Review of strategies of contention managers

The table 1 attempts to identify a general behavior of the benchmarks, classifying them according to their average performance in many cases experience. The purpose of this table is to be used as a reference of general strategy would work in most cases effectively. Anyway, checking this summarized table, you can see that: in the benchmark tests, Polka obtained a great performance in every environments of analysis performed by the article [1]. So, it's the best choice for a default contention manager, but is not the best for every cases. To adopt an optimal strategy, it is necessary to know both the context of problem that you want to solve, as the specification of the hardware where the implementation of transactional memory is performed.

In the same article [1] have a fairness analyses. The Polka was best on benchmarks but was not good on fairness like the timestamp. Like we said, timestamp had the best fairness polite, so, his threads do not wait too long for commit like the others managers. Again, the date leaves us with the uncertainty of what is the best manager

During this research, it was noticed that it seems very unlikely that there is a definite strategy for contention manager, making it important to pay attention to choosing the best strategies to solve the problem in the shortest possible time is necessary to study the context in which solution is being employed.

5 CONCLUDING REMARKS

In this paper, we made no new findings on the subject or tests, only described the unique way each contention manager strategy and their functioning, trying to compare the performance of each solution without going into the details of the tests. Moreover You can analyze the reference articles to find accurate comparisons between these contention managers. However, in all the articles that have comparisons, it is apparent that there is no perfect solution contention manager, ie each contention manager works uniquely for each test performed and the adoption of several strategies being dynamically modified to solve a problem [16] shows a viable option for the development of transactional memory. Therefore, it is clear that to implement transactional memory technique, choosing a contention manager is essential, because it will behave differently in every case.

6 ACKNOWLEDGMENTS

We aknowledge the financial support of Fundação de Amparo à Ciência e Tecnologia do Estado de Pernambuco (FACEPE).

7 REFERENCES

1. Scherer III, William N., and Michael L. Scott. "Advanced contention management for dynamic software transactional memory." Proceedings of the twenty-fourth annual ACM symposium on Principles of distributed computing. ACM, 2005.
2. Scherer III, William N., and Michael L. Scott. "Contention management in dynamic software transactional memory." PODC Workshop on Concurrency and Synchronization in Java programs. 2004.
3. Shavit, Nir, and Dan Touitou. "Software transactional memory." Distributed Computing 10.2 (1997): 99-116.
4. Rossbach, Christopher J., Owen S. Hofmann, and Emmett Witchel. "Is transactional programming actually easier?." ACM Sigplan Notices 45.5 (2010): 47-56.
5. Herlihy, Maurice, and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. Vol. 21. No. 2. ACM, 1993.
6. Guerraoui, Rachid, Maurice Herlihy, and Bastian Pochon. "Polymorphic contention management." Distributed Computing. Springer Berlin Heidelberg, 2005. 303-323.
7. Guerraoui, Rachid, Maurice Herlihy, and Bastian Pochon. "Toward a theory of transactional contention managers." Proceedings of the twenty-fourth annual ACM symposium on Principles of distributed computing. ACM, 2005.
8. Scherer III, William N., and Michael L. Scott. "Contention Management in Dynamic Software Transactional Memory–Errata." (2005).
9. Herlihy, Maurice, Victor Luchangco, and Mark Moir. "Obstruction-free synchronization: Double-ended queues as an example." Distributed Computing Systems, 2003. Proceedings. 23rd International Conference on. IEEE, 2003.
10. Herlihy, Maurice, and Nir Shavit. The Art of Multiprocessor Programming, Revised Reprint. Elsevier, 2012.McKenney, Paul E. "Is parallel programming hard, and, if so, what can you do about it?." Linux Technology Center, IBM Beaverton (2011).
11. S.V. Adve et al. (Novembro de 2008). "Parallel Computing Research at Illinois: The UPCRC Agenda". Parallel@Illinois, Universidade de Illinois.
12. Guerraoui, Rachid, Maurice Herlihy, and Bastian Pochon. "Polymorphic contention management." Distributed Computing. Springer Berlin Heidelberg, 2005. 303-323.
13. Maldonado, Walther, et al. "Scheduling support for transactional memory contention management." ACM Sigplan Notices. Vol. 45. No. 5. ACM, 2010.
14. Spear, Michael F., et al. "A comprehensive strategy for contention management in software transactional memory." ACM Sigplan Notices. Vol. 44. No. 4. ACM, 2009