A sequential and MR-Lock Reimplementation of : Lock-free Transactions without Rollbacks for Linked Data Structures

Jigar Jain

University of Central Florida
jigarjain@knights.ucf.edu

Rushikesh Gaidhani

University of Central Florida
rushikeshgaidhani@Knights.ucf.edu

Pranjal Bahuguna

University of Central Florida
pranjal.bahuguna@Knights.ucf.edu

ABSTRACT

Non-blocking transactional structures are now been there for quite a while now. With the transactional structure we want to achieve atomicity of multiple operation of a transaction and consistency of the structure after rollback. Older solutions like software transactional memory (STM) and transactional boost provide synchronization at an external layer over the structure itself. This introduces an overhead which is not necessarily required. Thus, it is a probable problem. To solve this issue, researchers provided with a solution which make structural changes in the existing data structure and make it transactional, lock-free.

In this work we present a sequential re-implantation of a the above provided solution. We applied transactional transformation to a linked-list and made a sequential version of it. Next, we introduced multi-resource lock version of the same which will used locking to support multi- thread operation on the linked list.

As anyone would expect, we get constant time graph for a sequential version of the transactional structure. But the case for locked based multi thread version is different. We get worse performance as compared to sequential version as we get the overhead of maintain a resource vector. As the number of threads increase the size of resource vector increase and thus contention increases.

In future we plan to completely re-implement a lock free transactional linked data structure and yet again compare its result with the results from this paper.

Keywords

Transactional Data Structure, Lock free, Rollback

1. INTRODUCTION

Multicore CPUs are now more common and affordable and thus it has put more pressure on the researchers and developers to develop data structure which can take maximum advantage out of these multi-core systems. Programming languages like Java and C++ are continuously putting out libraries for developers to use it to their advantage. These libraries work fine when we execute individual operation concurrently the problem occurs when a set of operations are to be done which need to

operate atomically (performing a transaction atomically). This problem is solved by making transactional transformation to the linked data structure [1].

In this paper we reimplement and discuss the results of sequential and locked version of transactional linked list. For our study we have used linked list. A linked list is an ordered data set with each node carrying the address of the next/upcoming node. We have a set of transaction which will perform on the linked list and each operation will have a set of operations which must operate atomically. In the sequential version we need not worry of multiple transaction working at the same time. Also, this will act as a pilot reading for our further multi resource locked based concurrent implementation (MR Lock) [2]. In the MR Locked version will try to achieve concurrency and atomicity of transaction by locking shared objects.

This paper is organized as follows. Section 2 we mention our implementation of both sequential and concurrent lock-based structure. In Section 3 we interpret the results. And in Section 4 we talk about our future goals.

2. IMPLEMENTATION

We implement our solution in Java. We use Java over C++ because of the familiarity of the language.

2.1 Sequential Implementation

Our goal of this implementation is to create a reference point from where we can compare our results with concurrent structures and obtain results and feasibility. We create a standard linked list structure and use method provided with the base data structure (Find, Insert and Delete). We hold all the transaction in an array. The transaction contains operations in the following format <operation, value>. Since all transaction are sequential thus, all operation of a transaction can be put in the array. We execute each operation one by one in order that they come. Since it is sequential, we do not care of the conflicts and rollback as the success of all operations are guaranteed.

For 150000 random insert, find and delete operation

Figure 1

2.2 Locking Based Concurrent Implementation

We re-implement the MR-Lock¹ provided by the researchers and use it. We hold all resource required for a transaction in a resource vector. Each level in the vector hold all resources required by a transaction. We check for conflicting resources that might be in use by some older thread. It can be done by just looking up the vectors for duplicate [2]. If duplicate exists, the transaction waits until the resource is available that is the older transaction is committed. In our implementation we use acquire function to set the transaction to the tail of the vector and the check if all the resources it requires to proceed are available. Once all the resources are available (i.e. all the keys are available) we use Do operation to start the execution. Finally, after all operations are done, we Release all the locked. We consider each node of the linked list as a resource. As each node is a resource, we lock all required a node for an operation thus multi resource locking.

There is no need to consider rollbacks as we detect conflicting transaction and make them wait until older transaction has completed its operation.

3. RESULT

We consider 100000 operations per thread. As anyone familiar with concurrent programming would know we get a constant time irrespective of number of threads which are spawned. It is useless to spawn multiple threads as there a single point of entry to the transaction queue and each operation is executed to sequentially.

For lock based multi thread version we get an unexpected slower performance than the sequential version. We interpret it as a result of contention caused because of creating and handling a resource vector. Every time a new transaction comes in we have to check all the active transactions and check for conflicting resources (in our case node). We no not let the thread proceed until all required resources are available. Thus, slower performance. We also do not provide any helping using Descriptors [3]. Thus, a thread waits do nothing to increase the

overall throughput of the system. Figure 1 shows the MR Lock implementation using random insert, find and delete operation.

4. CONCLUSION

We conclude by the above given results that MR Locking does not provide speed up in our implementation. On paper if we consider the same scenario, we should get a speed up. Also, We observe that the time of execution increases as we increase the number of threads. It is a result of increase in contention cause by creating and handling the resource vector.

5. FUTURE IMPLEMENTATION

As this paper is still work in progress, we plan on re-implanting the lock-free transactions on linked list [1] and comparing its results with the MR Locked version. We will focus on the synchronization techniques and efficiency of the algorithm. We aim to get similar or better results by making changes in provided structure. Also, we may implement another solution to this problem and discuss the advantages and disadvantages of the same.

6. REFERENCES

- [1] Zhang, D., & Dechev, D. (2016). Lock-free Transactions without Rollbacks for Linked Data Structures. SPAA '16 Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures, 325-336
- [2] Zhang, D., Lynch, B., & Dechev, D. (2015). Queue-Based and Adaptive Lock Algorithms for Scalable Resource Allocation on Shared-Memory Multiprocessors . *International Journal of Parallel Programming*, 721-751.
- [3] Dechev D., Pirkelbauer P., Stroustrup B. (2006) Lock-Free Dynamically Resizable Arrays. In: Shvartsman M.M.A.A. (eds) Principles of Distributed Systems. OPODIS 2006. Lecture Notes in Computer Science, vol 4305. Springer, Berlin, Heidelberg

¹<http://cse.eecs.ucf.edu/gitlab/deli/mrlock>