THINK INDIA JOURNAL
ISSN: 0971-1260
Vol-22-NO-30 Special Issue-December-2019

Apragmatic Construction of Non-Blocking Binary Expression Trees

Raj Karan Singh[1], Gauri Mathur
[1] Assistant Professor, School of Computer Science and Engineering, LPU, Jalandhar,India
[2] Assistant Professor, School of Computer Science and Engineering, LPU, Jalandhar, India

Abstract:

The work is based on a new method for procuring non blocking implementation of a binary expression tree. The updates to the tree are used to modify contiguous portions of the tree atomically. We are using modified multi–word abstractions of typical LL, VL and SC elementary which in turn are executed from single-word CAS which is supported by both Intel 64 and AMD64 modern processors.We provide an approach suited for creation of binary expression tree whose pseudocode is given and it can be implemented in any programming language such as C++. This approach yields an effective and easy procedure for generation and traversal of the said data structure. The resultant implementation is free from the disadvantages of implementations obtained by using customary techniques such as locks and CAS.

Keywords:- Non-blocking; binary expression trees; effective CAS; fault tolerant; load-link-extended; locks

1. Introduction:

Non blocking implementations of conventional data structures have gained unparalleled significance in today's era of multi threaded computing paradigms. The vast convenience and advantages offered by non blocking implementations of data structures have structured the present day research in this direction. Although the non blocking implementation of trees may reap huge benefits, however such an implementation is riddled by issues. One of the customary techniques to the implementation of a non blocking implementation in a multi-threaded, parallel system is to utilise locks. Using locks, access to conjunct resources can be synchronized so as to give an impression of a dedicated resource based system. Locking mechanism albeit, a popular and convenient way of providing a platform for implementation of non blocking algorithms is riddled with problems such as being intolerant towards faults. Another major limitation of locks is that it does not handle deadlocks effectively. An alternate approach to using locks is to use abstract data type operations in contrast to using simple memory read/write operations. This approach is also deprecated due to many issues out of which limited concurrency and lack of any scrupulous correctness proof stand out. In order to quash these issues, an interspersed technique favoring the use of hardware synchronization through primitive atomic operations in multithreading environments like compare-and-swap is also in use.
Compare and Swap itself is plagued by ABA problem which generates a false positive match. To decipher and overcome these limitations and to provide an implementation which can boast of high performance, proven correctness and is non-blocking we will introduce a technique for the

implementation of binary expression trees based on a directed acyclic graph in which the number of directed edges entering a vertex is one. This directed acyclic graph will be atomically modifying any connected sub graph of this tree periodically. The technique is aimed at overcoming the deadlock issue as some process will always make progress even if some of them fail.

An expression tree is the data structure of choice for dealing with algebraic and boolean expressions. As with all synchronous implementations, the binary expression tree obtained using the said technique is most efficient when it is localized. The significant operations include insertions and traversal.

A direct consequence of using this technique is that the binary expression tree obtained by this suggested technique is linearizable and hence non-blocking. All the updates can be obtained in the form of atomic steps and hence the process coordination can be carried out in these terms. The C++ implementation of the Binary expression tree seems to surpass implementations obtained via other techniques such as locks. So in essence the implementation exceeds the locked implementation over a variety of workloads, processing time and resource utilization.

2. Related Work:-

A binary expression tree has been implemented using the said technique and many applications of search tree data structures exist using the conventional methods namely CAS and locks [1-9]. We have used the load-link-extended (LLX), validate-extended (VLX) and store-conditional-extended (SCX) which are abstractions of well known load-link (LL), validate (VL) and store-conditional (SC) operations derived from single word CAS. We use a generalization to the single word CAS, choosing Data Records for LLX, VLX and SCX operations. Each data record is composed of fixed number of mutable and immutable fields. For example let us consider a data record (r), LLX(r) is tasked with taking a stock of the changeable fields of r. It may return FAIL, in case it is in conjunction with an SCX which also involves the data record (r). SCX (U,S,gld,new) takes parameters a sequence of Data-record U, a subset S of U, gld is a pointer to the changeable field of data record in U [3]. On the finalization of a data record, then SCX, LLX operations may not change its mutable fields. Any LLX operation on the record will return a status of FINALIZED.

A process must perform LLX (r) for a record r, before invoking SCX or VLX(U). Invocation of last LLX by the said activity is treated asrelated to either SCX or VLX and such an operation shouldgive back a state of r.

The process modifies the data structure by an SCX (U,S,gld,new) if and only if all the records r in U are immutable since its linked LLX(r) barring this, the SCX is unsuccessful. On the same terms, VLX(U) returns TRUE if all records r in U are immutable since its linked LLX(r) by asimilaractivity, else the VLX fails. LLX, VLX and SCX may fail, however this does not stop us from building non blocking data structures. The modifications in these atomic operations are governed by a need to provide an efficient implement ability using single word CAS. The ABA problem is circumvented with the first limitation which applies to the CAS steps involved in updates.

Limitation 1: Each time SCX(U,S,gld, new) is called, it leaves it in a mutually exclusive state. The new SCX implementation involves locking elements of U in the order given. The success of SCX operation is a direct consequence of this ordering.

ISSN: 0971-1260
Vol-22-NO-30 Special Issue-December-2019

Limitation 2: Affixing an identifier such as a version number to every field and sorting the U sequences helps us attain a state named C post which no change in the field of any record will take place.
Limitation 3: The data record to be removed from the tree is finalized.
The successful completion of LLX(r) signifies that the data record r in the tree just before LLX is linearized [10]. In case SCX(U,S,gld, new) is linearized, and a pointer to a record namely new, then this record which is in the tree which is post SCX and is linearized. If pointers read from other data right from the start lead to a data record r, then r was present in the tree at some preliminary time during the operation.

3. Implementation:

The insertion into a binary expression tree is carried via an update mechanism in which a former connected subgraph in a down tree is atomically replaced by a new connected subgraph. All the nodes which are to be modified are included in the old subgraph. The update operation involves a single change of child pointer to some parent node. Figure 1 depicts construction of a binary expression tree from stack. A record with a set number of child pointers as mutable fields represents a tree node. Every child pointer is pointing to a data record or contains NIL. If an update results in the loss of data, a recent copy of node is made with amended data.

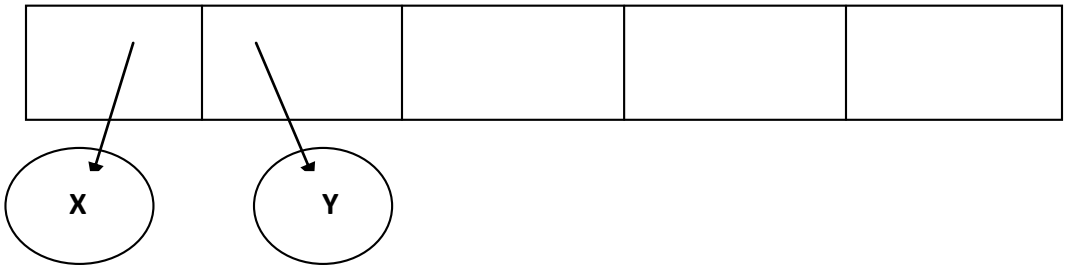
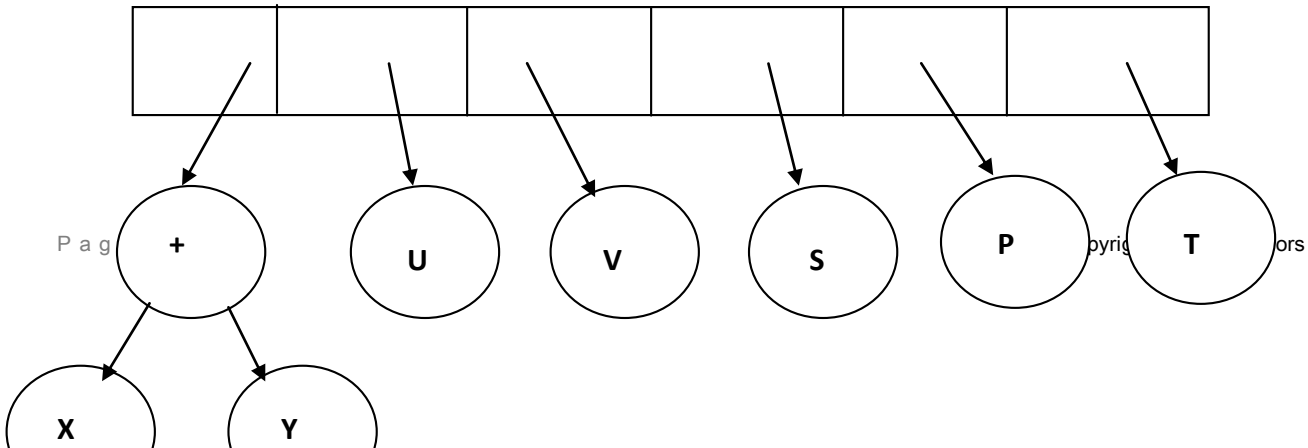


Figure 1. Insertion into a stack from left to right, used in construction of binary expression tree



ISSN: 0971-1260
Vol-22-NO-30 Special Issue-December-2019

Figure 2. Construction of a binary expression tree from a stack

The technique behind the insertions or updates is easy. At first anadjacentpart of the tree is selected and LLX is performed on the nodes including its ancestor and the set V of nodes to be taken out from the tree. After that an SCX is performed which mutates the child pointer. These steps if followed would mean that our update operation is inline with the technique proposed. The update operation reads a set of pointers (child pointers) starting from the entry point to the target node. Then an LLX operation is performed on the sequence (N0,N1,..) of the nodes beginning with N0. A non empty child of one of the preceding nodes is chosen to construct a new node using locally available information i.e the snapshots returned by the LLX on the previous elements. Another local computation tells whether the LLX operation should continue or not. If LLX returns FAIL or FINALIZED, then the attempted update is aborted as well. However on the successful completion of all LLX, SCX is invoked and the result is calculated locally. This method is invoked from TREE_INSERT() method which pushes all the operands into the stack and pops the operators from it, to construct a binary expression tree. Figure 2 lists the creation performed on operands and operators. A formal pseudocode for the above operation is as follows:-

```
INSERT(args)
traverse pointers from the entry point to arrive at target node N0
        counter=0
                FOR all Si in check(S0,S1,S2,S3,..,args) is TRUE
                        Si=LLX(Ni)
                IF Si == FAIL OR Si == FINALIZED THEN
                        RETURN FAIL
                ENDIF
        Si= immutable fields of Ni
        Ni+1=GET_NEXT_NODE(S0,S1,S2,S3,..,args)
        counter=counter+1;
                ENDFOR
        IF SCX(SCX_ARGS(S0,S1,S2,S3,..,args)) THEN
                RETURN RESULT(S0,S1,S2,S3,..,args)
        ELSE
                RETURN FAIL
        ENDIF
```

ISSN: 0971-1260
Vol-22-NO-30 Special Issue-December-2019

```
TREE_INSERT()

Counter=0


WHILE INSERT(args) NOT EQUAL FAIL DO
        FOR all Counter in (S0,S1,..,args)
                IF S(Counter) NOT EQUAL {+,-,*,/..} THEN
                        PUSH(S(counter))
                ELSE
                        POP(S(counter))
                END IF
        END FOR
END WHILE
```

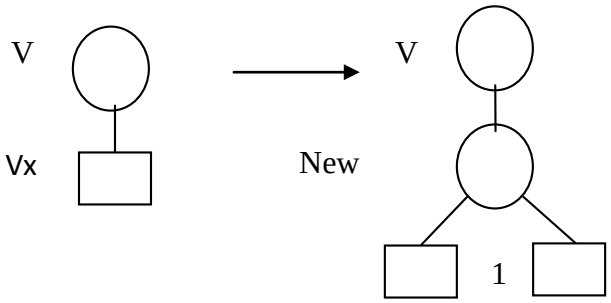


Figure 3. Insertion

The SCX_ARGS method is used to construct the segments U, S, gld and new for SCX. Every operation must result in the removal of a connected set of nodes N and replacement by a joined set N of recently– created nodes. The root of this is new.
Each data record will be represented with two changeable child pointers along with non changeable fields x , t1 are used to store the key, associated value(operands or operators). Figure 3 denotes an INSERT operation. INSERT AND TRAVERSE (key) operations are provided which have been used to implement the binary expression tree. The TRAVERSE (key) starting point is the entry value and uses one of the traversal techniques using READS of child pointers which is then subsequently returned. A leaf's parents always exist, and grandparents exist whenever the binary expression tree is non-empty. If it is void, TRAVERSE(key) returns NULL as an alternative to returning the grandparents. An INSERT(key, value) perform a SEARCH(key) and performs an update on reaching the leaf. The detailed pseudocode for these operations follow:-

```
TRAVERSE (key)
        T0=NULL
        T1=start
        T2=start.left
                WHILE T2 is internal
```

ISSN: 0971-1260
Vol-22-NO-30 Special Issue-December-2019

```
            T0=T1
            T1=T2
            T2=(key<T1.k) ? T1.left : T1.right
            RETURN (T0,T1,T2)
    END WHILE
```

4. Conclusion:

The above technique which has been used to construct and traverse a binary expression tree is based on an approach which involves a slight variation of the CAS technique. The resultant binary expression tree provides a non blocking implementation which is better in comparison to the resultant binary expression tree obtained from other non blocking techniques such as locks and CAS because of more fault tolerance. It also overcomes the ABA problem as is prominent in CAS implementations. The resultant binary expression tree is also very efficient as some overhead has been reduced. We hope to implement the same procedure to obtain non blocking trees and give a thorough and detailed study of performance evaluation vis a vis other non blocking implementations.

References:

[1] Y. Afek, H. Kaplan, B. Korenfeld, A. Morrison, and R. Tarjan, "BTree: A Practical Concurrent Self-Adjusting Search Tree"in Proc. 26th International Symposium on Distributed Computing, volume 7611 of LNCS, pages 1–15, 2012.

[2] K. S. Larsen,"Amortized Constant Relaxed Rebalancing Using Standard Rotations"in Acta Informatica, 35(10):859–874, 1998.

[3] Trevor Brown , Faith Ellen , Eric Ruppert, "A General Technique for Non-Blocking Trees" in Proceedings of the 19th ACM SIGPLAN symposium on Principles and practice of parallel programming, February 15-19, 2014, Orlando, Florida, USA

[4] H. Kung and P. L. Lehman,"Concurrent Manipulation of Binary Search Trees"in ACM Transactions on Database Systems, 5(3):354–382, 1980.
[5] T. Crain, V. Gramoli, and M. Raynal,"A Speculation-Friendly Binary Search Tree"in Proc. 17th ACM Symp. on Principles and Practice of Parallel Programming, pages 161–170, 2012.

[6] L. J. Guibas and R. Sedgewick,"A Dichromatic Framework for Balanced Trees"in Proc. 19th IEEE Symposium on Foundations of Computer Science, pages 8–21, 1978.

[7] A. Braginsky and E. Petrank,"A Lock-Free B+tree"in Proc.24th ACM Symposium on Parallelism in Algorithms and Architectures, pages 58–67, 2012.

[8] T. Brown and J. Helga,"Non-Blocking k-ary Search Trees",in Proc. 15th International Conf. on Principles of Distributed Systems, volume 7109 of LNCS, pages 207–221, 2011.

ISSN: 0971-1260
Vol-22-NO-30 Special Issue-December-2019

[9] T. Brown, F. Ellen, and E. Ruppert,"Pragmatic Primitives for Non-Blocking Data Structures."in Proc. 32nd ACM Symposium on Principles of Distributed Computing, 2013.

[10] N. G. Bronson, J. Casper, H. Chafi, and K. Olukotun,"A Practical Concurrent Binary Search Tree"in Proc. 15th ACM Symposium on Principles and Practice of Parallel Programming, pages 257–268, 2010.

[11] Neeraj Mittal, Aravind Natarajan, "Fast Concurrent Lock-Free Binary Search Trees" in Proceedings of the 19th ACM SIGPLAN symposium on Principles of parallel programming, 2014.

[12] Arunmoezhi Ramachandran, Neeraj Mittal, "CASTLE: Fast Concurrent Internal Binary Search Tree using Edge-Based Locking", in ACM SIGPLAN Notices, 2015.