Optimization of Concurrent Associative Arrays Using Speculative Execution of Critical Sections

Vadim A. Smirnov[1], Artur R. Omelnichenko[2], Alexey A. Paznikov[3]
Saint Petersburg Electrotechnical University "LETI"
Saint Petersburg, Russia
[1]smirnov3308@gmail.com, [2]omelnichenko3308@gmail.com, [3]apaznikov@gmail.com

***Abstract*— Algorithms for implementing thread-safe associative arrays (red-black tree, van Emde Boas tree, hash-table with open addressing based on the Hopscotch method hashing collision resolution) using software transactional memory and speculative execution of critical sections are proposed in the paper. Efficiency analysis of associative arrays depending on the number of flows involved is presented, comparisons are made with analogous data structures based on coarse-grained and fine-grained locks, recommendations on the choice of algorithms for performing transactions are formulated. The principles of software transaction memory, the policy of updating objects in memory and the strategy of conflict detection are described. There are various methods of performing transactions implemented in the GCC 4.8.0 compiler. Modeling has shown that data structures based on transactional memory outperform similar implementations based on coarse-grained locks, but are inferior to implementations based on fine-grained locks.**

Keywords*— *transactional memory; thread-safe data structures; van Emde Boas tree; red-black tree; hash table; multithreaded programming; synchronization of multithreaded programs

I. Introduction

Access synchronization to shared resources is one of the most important tasks in developing of multithreaded programs. Nowadays, the main methods for solving this problem are using of: locks (mutex, spinlock, critical section), algorithms and data structures, non-blocking concurrent data structures and transactional memory.

Classic synchronization methods, based on the locking mechanism, allow you to organize critical sections in parallel programs, but they can be performed by one thread at a time.

There is an alternative to using usual locks: coarse and fine-grained locks. Thread-safe structures based on coarse-grained locks are easy to implement, but they are not effective enough, because they have limited concurrency of operations. Fine-grained locks provide good performance, but their usage presents considerable complexity [1].

Lock-free data structures are built on the basis of atomic operations. This approach allows to get rid of mutual locks, inversion of priorities and fasting threads completely. The drawbacks of this approach include the complexity of implementing thread-safe structures [2] and the problem of ABA.

II. Software transactonal memory

Today transactional memory is one of the most promising synchronization mechanisms; its usage allows to perform operations that change various parts of memory concurrently. Transactional memory simplifies parallel programming, allocating instructions to atomic transactions – the final sequences of transactions of transactional memory read / write [3]. The transaction section is speculative, and if there is a conflict in accessing the memory between the two transactions, one of the transactions is interrupted and all the changes in the memory are returned to the original state (rollback). Then the transaction is repeated. An important feature of transactional memory is the linearizability of the transactions' execution: a series of successfully completed transactions is equivalent to some sequential execution of transactions [4]. The main differences that determine the effectiveness of software transactional memory are the policy of updating objects in memory and the strategy of conflict detection.

At the moment, there are various implementations of transactional memory; This work uses the implementation of transactional memory in the GCC compiler (libitm library), which uses the early update policy for objects in memory and implemented a combined approach to conflict detection – the deferred strategy is used in conjunction with the pessimistic [1].

Algorithms for implementing thread-safe hash tables (based on the Hopscotch algorithm hashing collision resolution), Van Emde Boas trees and red-black search trees using transactional memory are proposed in this paper.

This work was supported by the Council on Grants of the President of the Russian Federation for the state support of young Russian scientists (project SP-4971.2018.5). The publication was carried out within the framework of the state work "Initiative scientific projects" of the basic part of the state task of the Ministry of Education and Science of the Russian Federation (ACT No. 2.6553.2017 / BC).

III. Hopscotch hashing algorithm.

The hash table is one of the most widely used data structures for implementing associative arrays. Increasing its efficiency will reduce the execution time of a significant number of parallel programs [6]. Hash tables with open addressing are characterized by good cache location. Hopscotch hashing [6] combines the advantages of three approaches: Cuckoo hashing, the chain method, and the method of linear hashing. The algorithm has a high cache hit ratio. In the worst case, the time complexity of the addition operation is $O(n)$, at best $O(1)$. Search and delete operations are performed in constant time.

An example of an insert operation is shown in Fig. 1.

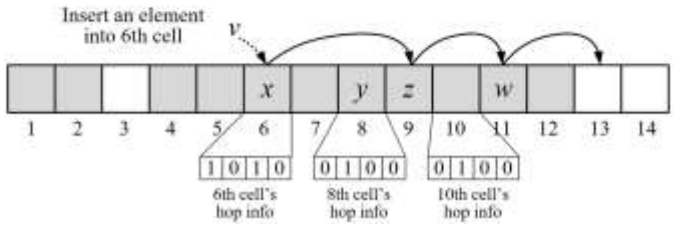

Fig. 1. Insert operation example.

The basic idea of Hopscotch hashing is to use the spatial cache locality property. The required element is in the neighborhood of the cell pointed to by the hash function. Fig. 2 shows the insert function of the hash table. The critical section is highlighted in the transaction (lines 4-23). This allows threads to add to the hash table concurrently. If the element with the given key is already in the hash table, the function returns false (line 6). If the first *ADD_RANGE* cells are not empty (in this implementation *ADD_RANGE* = 256), false returns, and the Resize () function changes the size of the hash table and performs the rehashing (lines 21-22). If an element is successfully added to the table, the function returns true (line 19).

```
1: procedure HopscotchInsert
2:   hash = HashFunc(key)
3:   start_bucket = segments_arys + hash
4:   transaction
5:     if Contains(key) then
6:       return false
7:     end if
8:     free_bucket_index = hash
9:     free_bucket = start_bucket
10:    distance = 0
11:    FindFreeBucket(free_bucket, distance)
12:    if distance < ADD_RANGE then
13:      if distance > HOP_RANGE then
14:        FindCloser(free_bucket, distance)
15:      end if
16:      start_bucket.hop_info |= (1 << distance)
17:      free_bucket.data = data
18:      free_bucket.key = key
19:      return true
20:    end if
21:    Resize()
22:    return false
23: end transaction
```

Fig. 2. Insert function exapmle

The following is a function to remove an item from a hash table (Fig. 3). The critical section of the delete function is also highlighted in the transaction (lines 4-19). If the element is removed successfully, the function returns true (line 14), otherwise false (line 18). To ensure maximum performance, the minimum possible size of the transaction section was chosen.

```
1: procedure HopscotchRemove
2:   hash = HashFunc(key)
3:   start_bucket = segments_arys + hash
4:   mask = 1
5:   transaction
6:     hop_info = start_bucket.hop_info
7:     for i = 0 to HOP_RANGE do
8:       mask <<= 1
9:       if mask & hop_info then
10:          check_bucket = start_bucket + i
11:        if key = check_bucket.key then
12:          check_bucket.key = NULL
13:          check_bucket.data = NULL
14:          start_bucket.hop_info &= ~(1 << i)
15:          return true
16:        end if
17:      end if
18:    end for
19:    return false
20: end transaction
```

Fig. 3. Hash table element remove function

IV. Thread-safe red-black tree

The implementation of a thread-safe red-black tree based on transactional memory is also proposed in this paper. Red-black trees provide a logarithmic increase in tree height depending on the number of nodes, which allows you to perform basic operations in time $O(\log_2 n)$.

Following is the function of adding an element to a red-black tree (Fig. 4). The critical section is allocated to the transaction (lines 3-10), which allows the threads to perform the addition of elements, without violating the integrity of the data and the balance of the tree.

```
1: procedure RBtreeInsert
2:   x = NewNode(data)
3:   transaction
4:     if !FindParent(x) then
5:       return false
6:     end if
7:     InsertNode(x)
8:     InsertBalance(x)
9:   return true
10: end transaction
```

Fig. 4. Red-black tree insert function

The INSERTNODE function of inserting a node (line 8) specifies the left or right fields of the parent node. If the parent node is missing, the added node becomes the root of the tree. The function of creating a new node NEWNODE (line 9) is moved beyond the transactional section: this allows to reduce its size, and therefore, to reduce the number of conflicts between different transactions.

V. THREAD SAFE VAN EMDE BOAS TREE.

The Van Emde Boas tree is a search tree for storing integer m-bit keys. The basic operations (Insert, Delete, Lookup, Min, Max) are performed in the time $O(\log_2(\log_2 U))$, where U is the universe size (the set of all possible elements), which is asymptotically better than the logarithmic complexity in balanced binary search trees.

VI. RESULTS OF THE EXPERIMENTS.

In this experiment, four methods of performing transactions implemented in the GCC compiler were used:

- Global locking method (gl_wt) – threads are synchronized with global locking.
- Multiple locking method (ml_wt) – threads are synchronized using multiple locks.
- Serial methods (serial, serialirr) – serial all transactions are executed sequentially. In serialirr, reading is performed concurrently, and when the write operation appears, the transaction goes into irrevocable mode (the transaction is not canceled), preventing unauthorized entries.

Also, the implementation of data structures based on coarse-grained and fine-grained locks was used for experiments (only for a hash table).

The efficiency $b = N / t$, where N is the number of operations performed, t is the time of execution of all operations, was used as an efficiency measure.

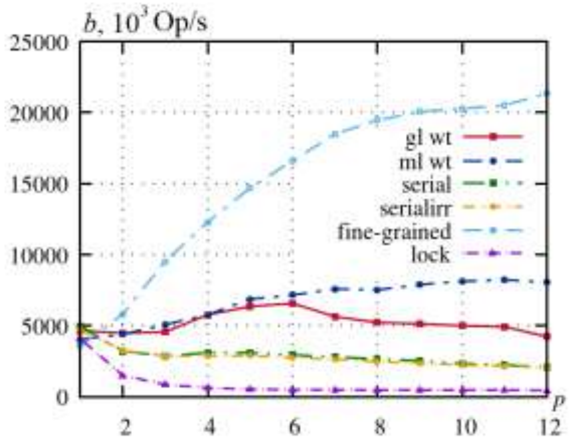

Fig. 5. Throughput of hash table, p = 1, …, 12

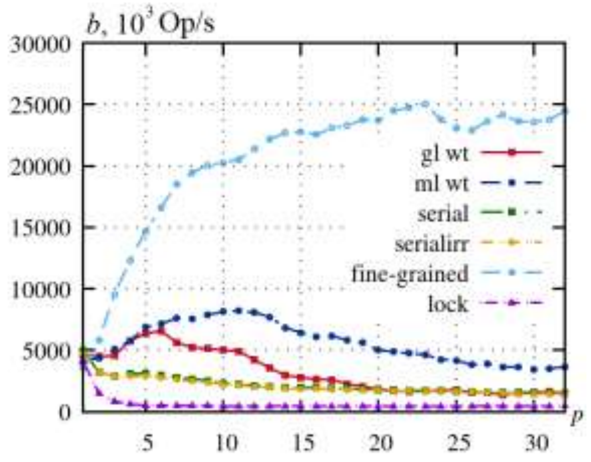

Fig. 6. Throughput of hash table, p = 1, …, 32

A hash table based on transactional memory provides greater throughput, compared to a coarse-grained lock implementation, for any number of threads (Fig. 5, 6). The methods gl_wt and ml_wt show an increase in throughput with an increase in the number of threads, provided that the number of threads does not exceed the number of processor cores. These methods are much more effective than the implementation based on coarse-grained locks, but inferior to the implementation based on fine-grained locks.

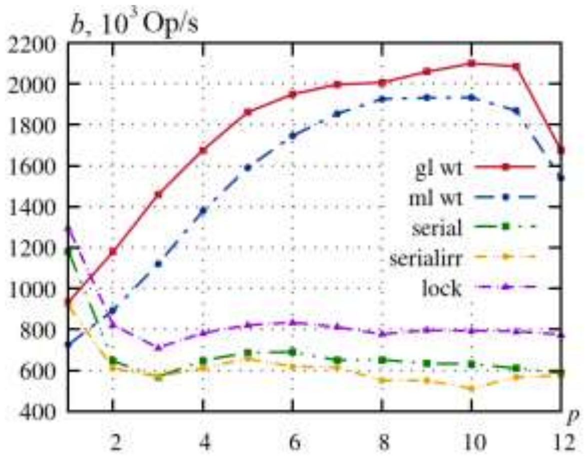

Fig. 7. Throughput of red-black tree, $p = 1, \ldots, 12$

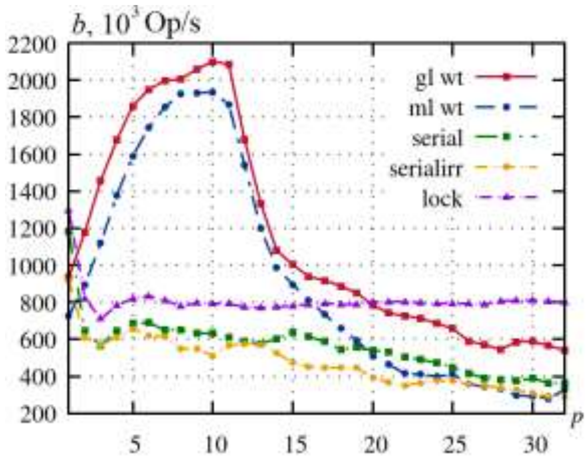

Fig. 8. Throughput of red-black tree, $p = 1, \ldots, 32$

The transactional memory red-black tree's throughput is higher than the implementation based on locks only when the number of threads is less than or equal to 20 for the method of executing transactions gl wt. Serial methods of performing transactions (serialirr and serial) are inferior to locks for any number of threads (Fig. 7, 8).

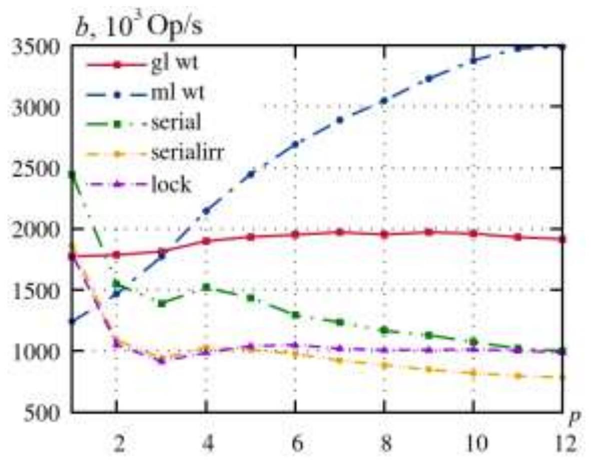

Fig. 9. Throughput of Van Emde Boas tree, $p = 1, \ldots, 32$

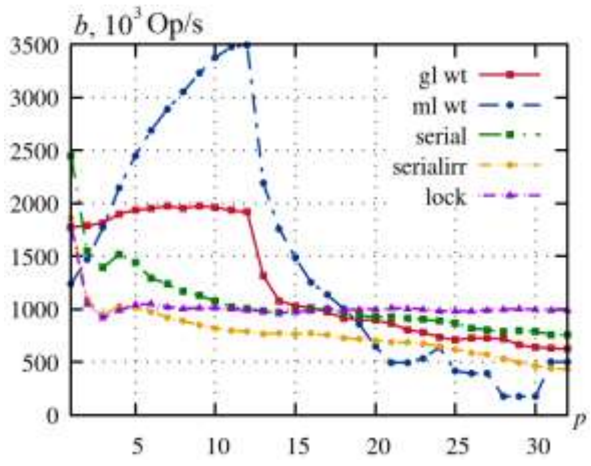

Fig. 10. Throughput of Van Emde Boas tree, $p = 1, \ldots, 32$

The results of experiments for the van Emde Boas tree are similar when using all methods except gl wt. The efficiency of gl wt does not depend on the number of threads, if the number of threads does not exceed the number of processor cores (Fig. 9, 10).

Conclusion

Transactional memory thread-safe red-black tree, van Emde Boas tree and a hash table based on the Hopscotch hashing resolution method are presented in this work.

The effectiveness of a hash table based on transactional memory is superior to similar implementations based on coarse-grained locks. The hash table is inferior in bandwidth to the analogue based on fine-grained locks when implemented with the number of threads exceeding the number of cores, this is due to the multitude of conflicts between transactions and their multiple cancellations. The recommended method for performing transactions for a hash table is the global locking method (gl_wt). The large red-black tree bandwidth is provided by gl_wt and serial (if the number of threads exceeds the number of processor cores).

References

[1] Kwiatkowski J. Evaluation of Parallel Programs by Measurement of Its Granularity. Parallel Processing and Applied Mathematic. Berlin. 2001, pp. 145–153.

[2] Fraser K. Practical lock freedom. PhD thesis. Cambridge University, 2003. 116 p.

[3] Kulagin I.I., Kurnosov M.G. Optimization of Conflict Detection in Concurrent Programs with Transactional Memory. Vestnik UUrGU. Series: Computational Mathematics and Informatics. 2016, Vol. 5; No. 4., pp. 46-60. (in Russian).

[4] Shavit N., Touitou D. Software Transactional Memory. Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing. Ottowa. 1995, vol. 10(2), pp. 204–213.

[5] Herlihy M., Shavit N., Tzafrir M. Hopscotch Hashing. International Symposium on Distributed Computing. Berlin. 2008, vol. 5213, pp. 350–364.