Modelling and optimization of transactional sections execution by the example of thread-safe hash-tables and search trees.

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

***Abstract.*–Algorithms using software transactional memory for implementing thread-safe associative arrays (a red-black tree, a hash table with open addressing based on the Hopscotch method hashing collision resolution) are proposed. The analysis of the efficiency of associative arrays with different number of involved threads and processor cores is given, comparison with data structures based on coarse-grained and fine-grained locks is given, algorithm selection recommendations for performing transactions are also formulated. The basics of software transaction memory, various policies for updating objects in memory and strategies for conflict detection are described. Various locking methods for using transactional memory implemented in the GCC 5.4.0 compiler are presented. The alternatives currently in use are briefly considered, advantages and disadvantages are also highlighted.**

Key words: Transactional memory; thread-safe data structures; red-black tree; hash table; concurrent programming; concurrent synchronization.

I. INTRODUCTION

Multi-core computing systems with shared memory are currently used to solve various complex problems. These include SMP and NUMA systems. With the increase of the processor cores number, the problem of providing scalable access to parallel streams to shared data structures is very important.

Synchronized access to shared resources is one of the important and complex tasks in the development of concurrent programs. Nowadays, the main methods for solving this problem are:

- Using locks (mutex, spinlock, critical section e. t. c.).
- Lock-free algorithms and data structures.
- Transactional memory.

Using of traditional synchronization primitives (semaphores, mutexes, spinlocks, etc.) has to be provided with ensurance of not only the correctness of the program - the absence of mutual locks and race conditions for data (data races), but also minimization of the waiting time for access to critical sections (shared resources). Classic synchronization methods, based on the locking mechanism, allow to organize

critical sections in parallel programs, where the execution is possible only by one thread at a time [1].

Using of coarse and fine-grained locks is the alternative for usual locks. Thread-safe structures based on coarse-grained lock are easy to implement, but they do not provide the best performance, as they have limited concurrency of operations. Fine-grained locks provide good performance, but using of them is a difficult task [2].

Lock-free data structures are built on atomic operations such as atomic store, atomic load, compare and swap, etc. This approach allows you to completely get rid of mutual locks, however can lead to the appearance of active locks (livelock) - situations where two threads simultaneously try to change the data structure, but each of them cycles through its operation due to changes made by another thread. Also, the disadvantages of this approach include the complexity of the implementation of lock-free structures [3], and the problem ABA.

II. SOFTWARE TRANSATIONAL MEMORY

Nowadays, transactional memory is one of the most promising synchronization mechanisms, it allows you to perform non-conflicting operations concurrently. Transactional memory simplifies parallel programming, allocating instruction groups to atomic transactions - the ultimate sequences of transactions for transactional memory read / write [4]. Changes made by thread within the transaction sections are invisible to other threads until the transaction is committed. If during the execution of a transaction threads are accessing one area of memory and one of the threads performs a write operation, the transaction of one of the streams is canceled (cancel, rollback). When a transaction section is executed by a single thread, other threads are in the state either directly before or immediately after the transaction. An important feature of transactional memory is the execution linearity of transactions: a series of successfully completed transactions is equivalent to some sequential execution of transactions. To perform transaction sections, the runtime-system of the compiler generates transactions. The transaction read operation copies the contents of the specified shared memory part to the corresponding part of the thread local memory. Transactional record copies the contents of the specified area of local memory to the corresponding section of shared memory accessible to all threads [1].

The main differences that determine the effectiveness of software transactional memory are the updating objects policy and the conflict detection strategy.

The updating objects policy determines when changes to objects within a transaction are committed. There are two main policies - lazy and early [5]. In the case of lazy policy (lazy version management), all operations with objects are deferred until the transaction is committed. All operations are recorded in a special log of changes (redo log), which is used for pending operations, when the transaction is committed. Using the change log slows down the commit operation, but greatly simplifies the procedure for undoing and restoring it.

The early policy of updating objects in memory (eager version management) assumes that all changes to objects are immediately written into memory. If the conflict occurs, the original state is restored using the undo log. It is characterized by fast fixation of the transaction, but by slow execution of the procedure for its cancellation [5].

The time when the conflict detection algorithm is initiated is determined by the conflict detection strategy [5]. With lazy conflict detection, this procedure is started at the commit stage of the transaction. The disadvantage of this strategy is the later detection of the conflict and, the implementation of unnecessary operations.

Pessimistic conflict detection (eager conflict detection) triggers an algorithm for detecting conflicts during each memory access operation. This approach avoids the disadvantages of a deferred strategy, but can lead to significant overhead.

This work represents the implementation of transactional memory in the GCC compiler (libitm library), which uses the early update policy of objects in memory and implemented a combined approach to conflict detection - the deferred strategy is used in conjunction with the pessimistic [1].

Using of software transactional memory for the implementation of thread-safe data structures is considered in this paper. One of the most common data structures in multithreaded programs is thread-safe associative arrays. Currently, there are many implementations of associative arrays, for example: a map container in the C ++ STL library, a C # type Dictionary, a Ruby Hash class, a Python dictionary type, and Java.util.map in Java.

The article suggests algorithms for implementing thread-safe hash tables (based on the Hopscotch algorithm hashing collision resolution) and red-black search trees using transactional memory.

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 large number of parallel programs [6]. In a hash table with open addressing, unlike a hash table with private addressing, the cell does not store a pointer to a linked list, but one element (key, value). During the inserting of an element, the corresponding cell is occupied, the insertion algorithm checks the following cells until a free cell is found. This approach is characterized by good cache location, since each cache line contains several hash table entries at once. A significant drawback of this approach is the performance decrease as the hash is filled.

Hopscotch hashing [6] combines the advantages of three approaches: Cuckoo hashing [7], the chain method [8], and the method of linear hashing [8]. The algorithm has a high cache hit ratio. In the worst case, the time complexity of the insert operation is O (n), at best O (1). Search and delete operations are performed in constant time.

The basic idea of Hopscotch hashing is to use the spatial cache locality feature. The required element is in the neighborhood of the cell pointed to by the hash function. In this paper, the size of the neighborhood is H = 32. The time for finding the element in a neighborhood is close to the search time in one cell. Is is achieved during inserting the element by displacing other elements.

Each cell contains information about which cells in the neighborhood have a key with the same hash value. In this implementation, this information is presented in the form of a linked list: the cell contains the relative position of the next and the first cell in the list.

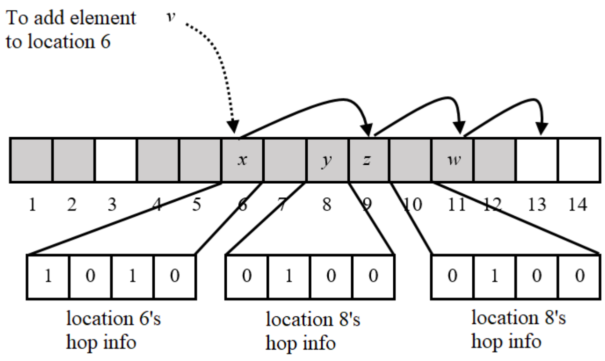
The function of inserting to a hash table is presented below. The critical section is highlighted in the transaction (lines 4-23). This allows threads to insert in the hash table concurrently. If the element with the given key is already in the hash table, the function returns false (line 6). If the empty cell could not be found within ADD_RANGE (in this implementation ADD_RANGE = 256), false returns, and the Resize () function changes the size of the hash table and performs the rehearsal (lines 21-22). If an element is successfully inserted 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
```

Fic. 1. Hash-table insert function.

Insert operation example is presented on Fig.2



Fic. 2. Insert operation example.

The following is a function to delete an item from a hash table. The critical section of the delete function is also highlighted in the transaction (lines 4-19). If the element was deleted 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
```

Fic. 3. Hash-table delete 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 are widely used in the implementation of associative arrays, provide a logarithmic increase in the height of the tree depending on the number of nodes and performs the basic operations of the search tree: insert, remove, and search for the node (O (logn)). Balancing is achieved by introducing an additional tree node attribute - "color" [9].

Red-black tree insert function is presented below. The critical section is highlighted in the transaction (lines 3-10), which allows threads to insert 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
```

Fic. 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 hence, to reduce the number of conflicts between different transactions.

V. Experimental results

The experiments were carried out on a computer system equipped with a 4-core Intel i5-3470 processor (no hardware transaction memory support) and a L1 cache size of 32 KB, L2 256 KB, L3 6 MB. The size of RAM is 8 GB. Used software: Linux Mint 18.1, GCC compiler 5.4.0.

The test consisted of running p threads performing the operations of adding, removing, and searching for an element. The operation rate was chosen randomly. Number of search operations is 40%, the insertion of the element is 30%, the deletion is 30%. The number of threads p in the test programs ranged from 2 to 32. At the first stage, the simulation was performed for p = 1, ..., 4 threads, which does not exceed the number of processor cores. In the second stage, the number of threads reached 32. In this experiment, four methods of performing transactions implemented in the GCC compiler were used:

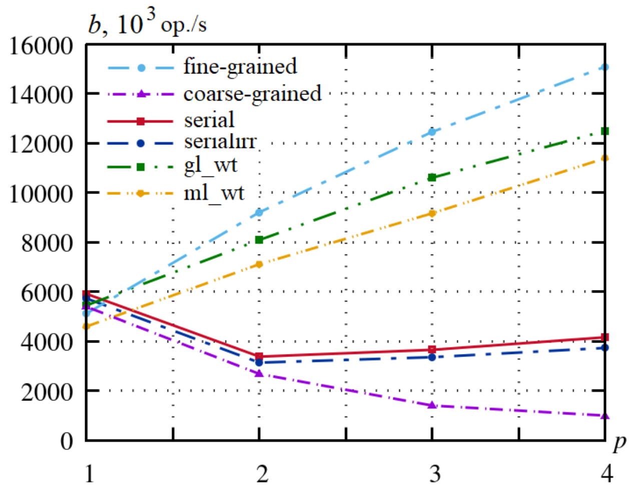
- Global locking method (gl_wt) - threads execute transactions concurrently, global locking occurs when threads start changing one part of the memory.
- Multiple-blocking (ml_wt) - threads execute transactions concurrently until they write to one memory

location, multiple transaction locking occurs when threads write to one memory location.

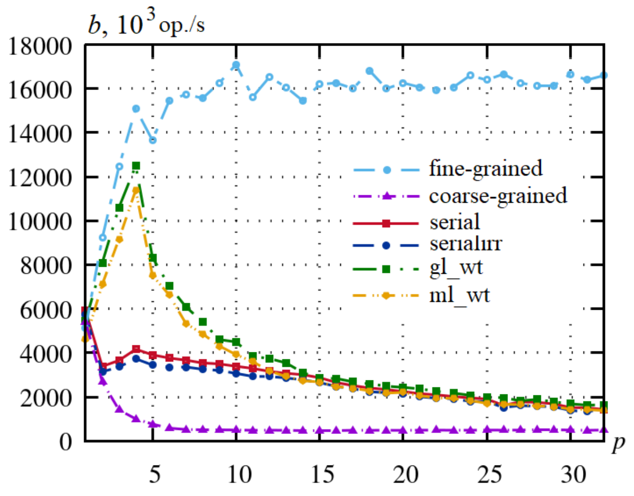
- Serial methods (serial, serialirr) - all transactions are executed sequentially. In serialirr, reading is concurrent, and when a write operation occurs, the transaction goes into irrevocable mode, preventing unauthorized entries.

Also implementations of data structures (red-black tree and hash-table) based on coarse-grained and fine-grained locks (only for a hash table) were used for the experiments.

Hash-table modelling results are shown on Fig. 5-6.



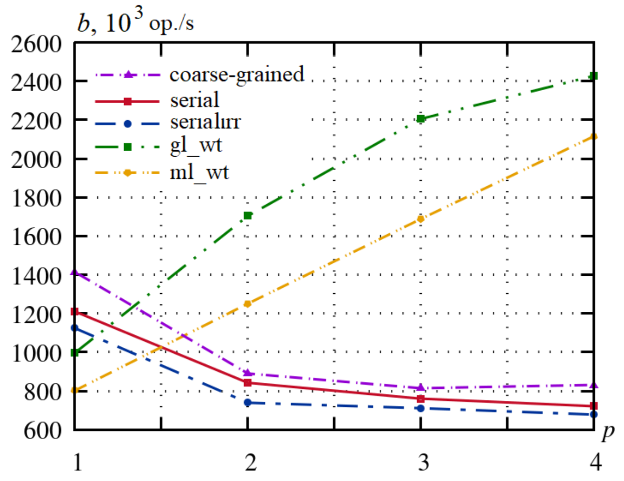
Fic. 5. Hash-table throughput for 1-4 threads.



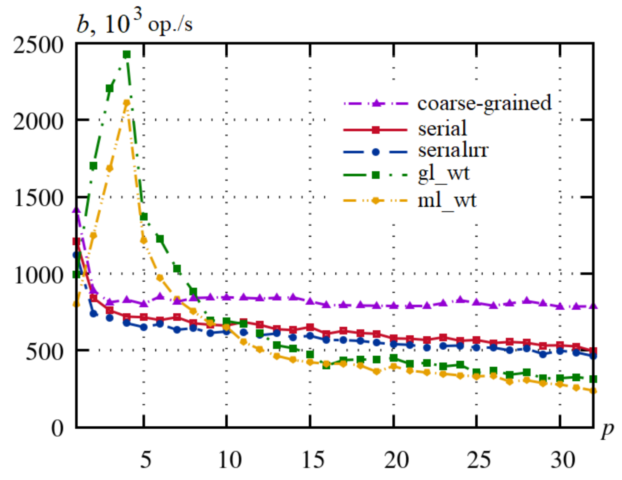
Fic. 6. Hash-table throughput for 1-32 threads

A hash table based on transactional memory provides greater throughput, compared to a coarse-grained lock implementation, with any number of threads (Picture 6). The methods gl_wt and ml_wt show an increase in throughput with an increase in the number of threads and are almost the same in performance as the fine-grained locks, if the number of threads does not exceed the number of processor cores (Picture 5). If the number of threads is more than 16, the effectiveness of all transaction execution methods is comparable. If the number of threads exceeds the number of processor cores, any of the four presented methods of performing transactions is inferior to fine-grained algorithms.

Pictures 7 and 8 show thread-safe red-black tree modelling results.



Fic. 7. Throughput of red-black tree for 1-4 threads.



Fic. 8. Throughput of red-black tree 1-32 threads.

Transactional memory red-black tree's throughput is higher than the implementation based on locks only when the number of threads is less or equal to 8 and for the gl_wt and ml_wt methods of performing transactions (Figure 8). Serial methods of performing transactions (serialirr and serial) are inferior to locks for any number of threads.

CONCLUSION

Transactional memory thread-safe red-black tree and a transactional memeory hash table based on the Hopscotch hashing collision resolution method are considered in this paper.

Modeling has shown that the efficiency of a hash table based on transactional memory is superior to similar implementations based on coarse-grained locks. Red-black tree is inferior in throughput to the analogue based on fine-grained locks when number of threads exceeded the number of cores, this is due to the multitude of conflicts between transactions and, consequently, their multiple cancellations. The recommended method for performing transactions for a hash table is the global locking method (gl_wt). The most effective red-black tree throughput is provided by gl_wt (with the number of threads not exceeding the number of processor cores) and serial (in case the number of threads exceeds the number of processor cores).

ACNKNOWLEGMENT

The paper was supported by Russian Federation President Council on Grants for governmental support for young Russian scientists (project SP-4971.2018.5) and within the scope of the state project “Initiative scientific project” of the main part of the state plan of the Ministry of Education and Science of Russian Federation (task № 2.6553.2017/BCH Basic Part).

REFERENCES

[1] Kulagin I. I, Kurnosov M.G. Optimization of Conflict Detection in Concurrent Programs with Transactional Memory // Vestnik UUrGU. Series: Computational Mathematics and Informatics.-2016.-T. 5; № 4. P.46-60.

[2] Kwiatkowski, J. Evaluation of Parallel Programs by Measurement of Its Granularity // Parallel Processing and Applied Mathematics, Naleczow, Poland, September 2001. pp. 145–153.

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

[4] Shavit, N., Touitou, D. Software Transactional Memory // In PODC’95: Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing, New York, NY, USA, Aug. 1995. ACM, pp. 204–213.

[5] Spear, M., Marathe, V., Scherer, W., Scott M. (2006) Conflict Detection and Validation Strategies for Software Transactional Memory. In: Dolev S. (eds) Distributed Computing. DISC 2006. //Lecture Notes in Computer Science, vol 4167. Springer, Berlin, Hedelberg Symposium on Parallelism in Algorithms and Architectures, June 2008, P. 275–284.

[6] Herlihy, M., Shavit, N., Tzafrir, M. Hopscotch Hashing // Proceedings of the 22nd international symposium on Distributed Computing. Arcachon, France: Springer-Verlag. pp. 350–364.

[7] Pagh, R., Rodler, F.F. Cuckoo hashing // Journal of Algorithms. – 2004. – №51, - C. 122–144.

[8] Knuth, D. E. The art of computer programming, volume 1 (3rd ed.): fundamental algorithms. Addison Wesley Longman Pulishing Co., Inc., Redwood City, CA, USA, 1997.

[9] Introduction to Algorithms. / Cormen T., Leiserson C., Rivest R., Stein C.; MIT Press, 2001.