Algorithms for access optimization to relaxed concurrent queues in shared memory systems

Vadim A. Smirnov
Department of computer science and engineering of Saint Petersburg Electrotechnical University LETI
Saint Petersburg, Russian Federation
smirnov3308@gmail.com

Alexey A. Paznikov
PhD in techniques, associate professor of department: computer science and engineering of Saint Petersburg Electrotechnical University LETI
Saint Petersburg, Russian Federation
apaznikov@gmail.com

Artur R. Omelnichenko
Department of computer science and engineering of Saint Petersburg Electrotechnical University LETI
Saint Petersburg, Russian Federation
omelnichenko3308@gmail.com

***Abstract* — The paper proposes algorithms for the implementation and optimization of a thread-safe queue pool with weakened semantics of operations for computing systems with shared memory using transactional software that implements speculative execution of critical sections. 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 8.4.0 compiler. Modeling showed that data structures based on transactional memory are superior to similar implementations based on locks, provided that the number of threads does not exceed the number of processor cores. For more threads, lock-based implementations are more suitable.**

Keywords— transactional memory; thread-safe data structures; relaxed concurrent data structures; queue pool, multithreaded programming; synchronization of multithreaded programs. (key words)

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.

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).

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. Ease of Use

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) [4]. 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 [5]. 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 [6].

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].

III. Relaxed concurrent queues

In this paper, we propose implementations of a thread-safe queue with weakened semantics of performing operations based on programmatic transactional memory. Queue is an abstract data type with the discipline of accessing the elements "first come, first come out" (FIFO). Adding an item is possible only at the end of the queue, selection - only from the beginning, while the selected item is removed from the queue.

Queues are often used in programs to implement a buffer in which you can put an element for further processing, preserving the order of receipt. In the average and worst case, all operations (adding and removing a node) are performed in O(1). Elements are added to the tail, retrieved from the head [N5].

In this paper, two approaches to the implementation of queues are used: based on linked lists (queue length is limited only by the amount of available memory) [7] and based on static arrays (the maximum queue length is specified) [8].

Implementation of a queue based on linked lists:

- Queue elements are stored in a singly linked list.
- To quickly add and retrieve elements from the list, a pointer to the last element (tail) is supported.
- New items are added to the end of the list.

An example of such a queue is presented in Figure 1.

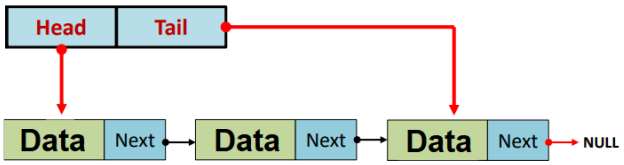

Fig. 1. Example of queue based on linked lists.

Advantages: Queue length is limited only by the amount of available memory.

Disadvantages (comparing with the implementation based on arrays): working with a queue is a bit slower, it requires more memory to store one element. Functions enqueue and dequeue are shown in Figures 2 and 3.

```
 1: procedure BufferEnqueue
 2:   item.data = value
 3:   item.timestamp = timestamp
 4:   item.next = NULL
 5:   if left = NULL then
 6:     left = item
 7:     right = item
 8:   else
 9:     right.next = item
10:     right = item
11:   end if
```

Fig. 2. Enqueue function in the queue based on linked lists.

```
 1: procedure BufferDequeue
 2:   if left != NULL & right != NULL then
 3:     if left = right then
 4:       value = left.data
 5:       Delete(left)
 6:       return value
 7:     else
 8:       item = left
 9:       left = left.next
10:       value = item.data
10:     Delete(item)
11:     return value
12:   end if
```

Fig. 3. Dequeue function in the queue based on linked lists.

Implementation of a queue based on cyclic arrays:

- Queue elements are stored in an array of fixed length [0…L – 1].
- The array is logically represented as a ring.
- In the empty queue tail = 0, head = L.

An example of such a queue is presented in Figure 2.

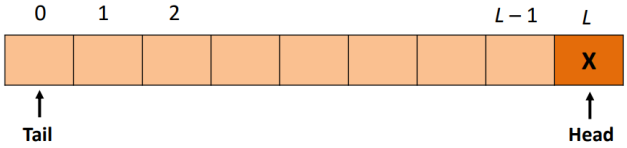

Fig. 4. Example of queue based on cyclic arrays.

Functions enqueue and dequeue for are shown in Figures 5 and 6.

```
 1: procedure BufferEnqueue
 2:   item.data = value
 3:   item.timestamp = timestamp
 4:   item.next = NULL
 5:   if left = NULL then
 6:     left = item
 7:     right = item
 8:   else
 9:     right.next = item
10:     right = item
11:   end if
```

Fig. 5. Enqueue function in the queue based on cyclic arrays.

```
 1: procedure BufferDequeue
 2:   if left != NULL & right != NULL then
 3:     if left = right then
 4:       value = left.data
 5:       Delete(left)
 6:       return value
 7:     else
 8:       item = left
 9:       left = left.next
10:       value = item.data
10:     Delete(item)
11:     return value
12:   end if
```

Fig. 6. Dequeue function in the queue based on cyclic arrays.

While developing scalable thread-safe data structures for computing systems with shared memory, an approach based on weakening the order of operations looks promising [9]. The weakened data structures built on its basis make it possible to achieve an increase in throughput in comparison with structures focused on ensuring strict semantics of operations.

The basis of this approach is a compromise between scalability and the correctness of the semantics of operations. It is proposed to weaken the semantics of operations to increase the scalability. For example, when searching for the maximum element in an array, a thread may skip parts of the array blocked by other threads to increase the performance of the search operation, and the accuracy of this operation will be lost [10].

Most well-known non-blocking thread-safe structures and blocking algorithms have a single point for performing operations on the structure [11]. For example, when inserting an element into a queue, you must use a single pointer to the first element of the structure. In the case of a multi-threaded system, this fact is a bottleneck, since each thread is forced to block one element, causing other threads to wait. In weakened data structures, a single structure is replaced by a set of simple structures, the composition of which is considered as a logically single structure. As a result, the number of possible points of access to this structure increases, which helps to avoid bottlenecks.

Within this approach, each simple structure is usually protected by a lock. While performing the operation, the thread accesses a random structure from the set and tries to block it. If the structure is successfully locked, the thread completes the operation; otherwise, it randomly selects a new structure. Thus, thread synchronization is minimized, but loss of accuracy of operations is acceptable [12].

A queue pool is a composition of simple queues that are protected by locks. Operations are performed on randomly selected queues from the pool. An example of such a pool is shown in Figure 7.

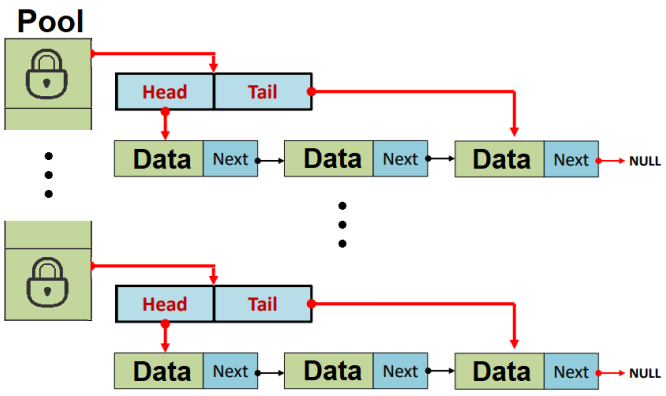

Fig. 7. Example of pool of queues based on linked lists.

The functions of operation enqueue in a thread-safe pool based on locks shown in Figure 8.

```
1: procedure LOCKENQUEUE
2:   queue_id = RANDOM % numOfQueues
3:   MUTEXLOCK(queue_id)
4:   counter = counter + 1
5:   BUFFERENQUEUE(queue_id, value, counter)
6:   MUTEXUNLOCK(queue_id)
```

Fig. 8. Enqueue function in the pool of queues with lock synchronization.

Following is the functions of operation dequeue in a thread-safe pool based on locks (Fig. 9). First, two random adjacent queues are selected (lines 2-9). Then the selected queues are blocked and the queue with the oldest element is determined. Performing an dequeue operation (line 15, 17).

```
1: procedure LOCKDEQUEUE
2:   first_queue_id = RANDOM % numOfQueues
3:   if first_queue_id % 2 then
4:     if first_queue_id = numOfQueues – 1 then
5:       first_queue_id = first_queue_id – 1
6:     else
7:       first_queue_id = first_queue_id + 1
8:   end if
9:   second_queue_id =  first_queue_id + 1
10:  MUTEXLOCK(first_queue_id)
11:  MUTEXLOCK(second_queue_id)
12:  first_TS = GETOLDESTTS(first_queue_id)
13:  second_TS = GETOLDESTTS(second_queue_id)
14:  if first_TS < second_TS then
15:    value = BUFFERDEQUEUE(first_queue_id)
16:  else
17:    value = BUFFERDEQUEUE(second_queue_id)
18:  end if
19:  MUTEXUNLOCK(first_queue_id)
20:  MUTEXUNLOCK(second_queue_id)
21:  return value
```

Fig. 9. Dequeue function in the pool of queues with lock synchronization.

The functions of operation enqueue in a thread-safe pool based on software transactional memory in Figure 10.

```
1: procedure STMENQUEUE
2:   queue_id = RANDOM % numOfQueues
3:   transaction
4:   counter = counter + 1
5:   BUFFERENQUEUE(queue_id, value, counter)
6:   end transaction
```

Fig. 10. Enqueue function in the pool of queues with transactional memory synchronization.

Following is the functions of operation dequeue in a thread-safe pool based on transactional memory (Fig. 9). The critical section is allocated to the transaction (lines 10-18), which allows the threads to perform the operations with shared resources, without violating the integrity of the data.

```
1: procedure STMDEQUEUE
2:   first_queue_id = RANDOM % numOfQueues
3:   if first_queue_id % 2 then
4:     if first_queue_id = numOfQueues – 1 then
5:       first_queue_id = first_queue_id – 1
6:     else
7:       first_queue_id = first_queue_id + 1
8:   end if
9:   second_queue_id =  first_queue_id + 1
```

```
10: transaction
11:   first_TS = GETOLDESTTS(first_queue_id)
12:   second_TS = GETOLDESTTS(second_queue_id)
13:   if first_TS < second_TS then
14:     value = BUFFERDEQUEUE(first_queue_id)
15:   else
16:     value = BUFFERDEQUEUE(second_queue_id)
17:   end if
18: end transaction
19: return value
```

Fig. 11. Dequeue function in the pool of queues with transactional memory synchronization.

IV. Results of the Experiments

The first experiment was the launch of p threads performing enqueue and dequeue operations. The second experiment contains 40% of enqueue operations, 40% of dequeue and 20% of reading operations. The type of operation was randomly selected. The number of threads p in test programs varied from 2 to 32. At the first stage, modeling was performed for p = 1, ..., 4 threads, which does not exceed the number of processor cores. At the second stage, the number of threads reached 32. In this experiment, we used four transaction execution methods implemented in the GCC compiler:

- 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 locks was used for experiments.

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.

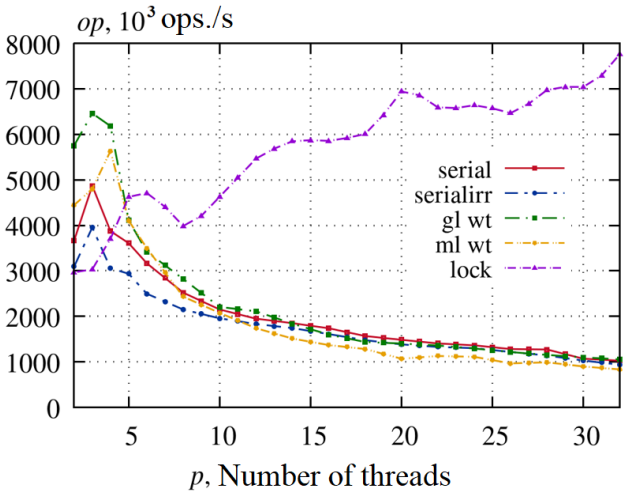

Fig. 12. Throughput of Linked List Queue Pool, enqueue and dequeue operations

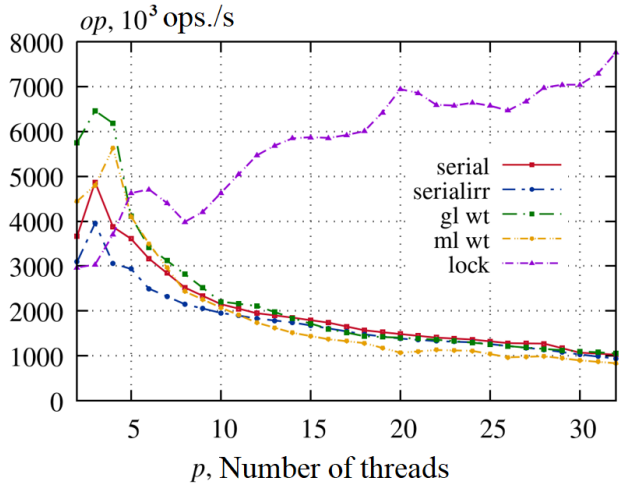

Fig. 13. Throughput of Cyclic Arrays Queue Pool, enqueue and dequeue operations

A pool of queues based on transactional memory provides greater throughput, compared to a lock implementation, for number of threads exceeding the number of cores (Fig. 12, 13). Implementation using locks shows good scalability. The methods gl_wt and ml_wt are much more effective than the sequential methods serial and serialirr.

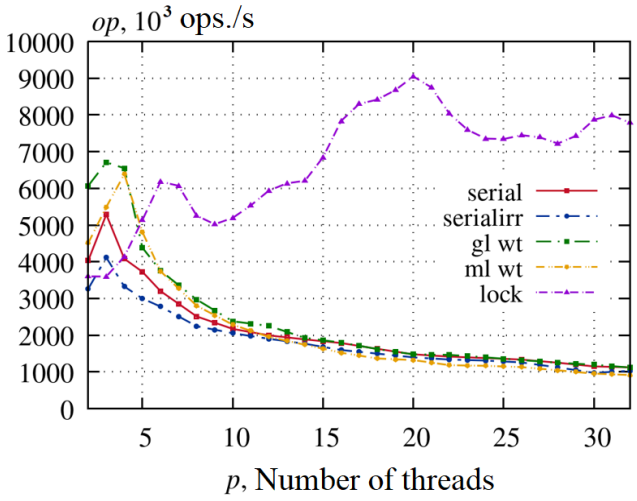

Fig. 14. Throughput of Linked List Queue Pool, mixed operations

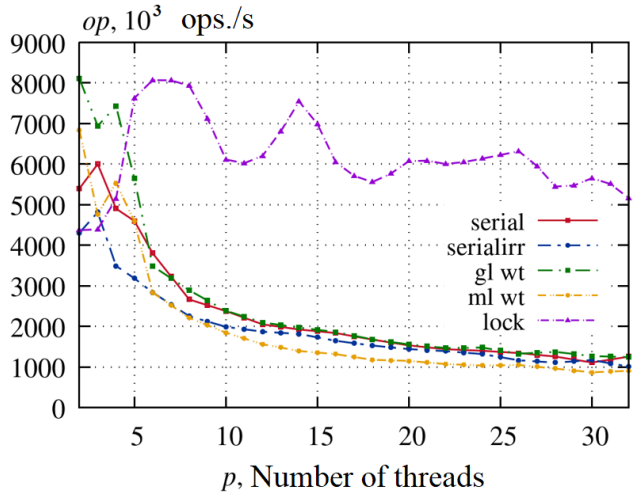

Fig. 15. Throughput of Cyclic Arrays Queue Pool, mixed operations

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. In other cases, a locking implementation shows better results.

V. CONCLUSION

In this work were implemented thread safe relaxed data structures linked list queue pool and cyclic arrays queue pool with locks. We also optimized these structures using software transactional memory. The optimized data structure has a 2 times performance boost in comparison with lock-based structures provided that the number of threads not exceeding the number of cores. Exceeding the number of processor cores leads to a drop in throughput, this is due to the multitude of conflicts between transactions and their multiple cancellations.

The recommended method for performing transactions is the global locking method gl_wt.

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, T. 5; № 4. pp. 46-60.

[4] Larus, J., Kozyrakis, C. Transactional memory // Communications of the ACM, v.51 n.7, July 2008. pp. 80–88.

[5] 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.

[6] 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.

[7] 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.

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

[9] Y. Afek, G. Korland, E. Yanovsky. Quasi-Linearizability: Relaxed Consistency for Improved Concurrency OPODIS, V. 6490, 2010, pp. 3-10.

[10] M. Dodds, A. Haas, and C. M. Kirsch. Fast concurrent data-structures through explicit timestamping. Technical Report TR 2014–03, Department of Computer Sciences, University of Salzburg, 2014.

[11] M. Dodds, A. Haas, and C.M. Kirsch. A scalable, correct time-stamped stack. In POPL. ACM, 2015.

[12] M.M. Michael and M.L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In PODC. ACM, 1996.