Algorithms for Optimization of Relaxed Concurrent Priority Queues in Multicore Systems*

Andrey V. Tabakov[1], Alexey A. Paznikov[2]
Department of Computer Science and Engineering
Saint-Petersburg Electrotechnical University "LETI"
St. Petersburg, Russia
[1]komdosh@yandex.ru, [2]apaznikov@gmail.com

***Abstract*— Design of scalable concurrent data structures for shared memory systems is one of promising approach to relaxation of operation execution order. Relaxed concurrent data structures are non-linearizable and do not provide strong operation semantics (such as FIFO/LIFO for linear lists, delete max (min) element for priority queues, etc.). In the paper, we use the approach based on design of concurrent data structure as multiple simple data structures distributed among the threads. For operation execution (insert, delete), a thread randomly chooses a subset of these simple structures and make actions on them. We propose optimized relaxed concurrent priority queues based on this approach. We designed algorithms for optimization of priority queues selection for insert/delete operations and algorithm for balancing of elements in queues.**

Keywords*— *concurrent data structures; relaxed data structures; relaxed semantics; multicore systems; multithreading

I. INTRODUCTION

The Multicore computing systems (CS) with shared memory are the primary means of solving complex problems and processing large amounts of data. It is used both autonomously and as part of distributed CS (cluster, multi-cluster systems, systems with mass parallelism). Such systems include number of multicore processors that share a single address space and have a hierarchical structure (Fig. 1). Thus, the Summit (OLCF-4) supercomputer (17 million processor cores), which is first in the TOP500 rating, includes 4608 computing nodes, each of which is equipped with two 24-core IBM Power9 family of universal processors and six NVIDIA Tesla V100 graphics accelerators (640 cores) [1]. The Sunway TaihuLight supercomputer (more than 10 million processor cores, second in the TOP500 rating), is equipped with 40,960 Sunway SW26010 processors, including 260 computing cores [2].

Among the existing CS with shared memory, there are SMP-systems, providing the same speed of access of processors to memory, and NUMA-systems, represented as a set of nodes (composition of processor and local memory) and characterized by different latency for access of processors to local and remote memory segments.

The reported study was funded by RFBR according to the research project № 18-57-34001 and by Russian Federation President Council on Grants for governmental support for young Russian scientists (project SP-4971.2018.5).

Parallel programs for shared-memory CS are created in a multithreaded programming model. The main problem arising in the development of programs is the organization of access to shared data structures. It is necessary to implement the correct execution of operations by parallel threads (no race conditions, dead-locks, etc.) and to provide scalability for a large number of threads and a high intensity of operations. For this, it is necessary to develop means for synchronizing threads and constructing algorithms for performing operations for thread-safe data structures. The main methods of thread synchronization are locks, non-blocking thread-safe data structures and transactional memory.

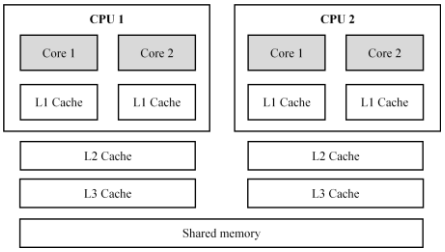

Fig. 1. Multicore system hierarchical structure

Locks (semaphores, mutexes) implement access to shared memory areas with a single thread at any time. Among the locking algorithms, we can distinguish TTAS, CLH, MCS, Lock Cohorting, Flat Combining, Re-mote Core Locking, etc. [3]. The disadvantages of locks are the possibility of deadlocks (livelocks), threads starvation, priority inversion and lock convoy.

Non-blocking concurrent data structures have the property of linearizability, assuming that any parallel execution of operations is equivalent to some sequential execution of them, and the completion of each operation does not require the completion of any other operation [4]. Among non-blocking data structures, there are three classes: wait-free – each thread completes the execution of any operation in a finite number of steps; lock-free – some threads complete the execution of operations in a finite number of steps; obstruction-free – any thread completes the operations for a finite number of steps, if the execution of other threads is suspend-ed. Among the most common non-blocking algorithms and data structures, we could highlight Treiber Stack, Michael & Scott Queue, Harris & Michael List, Elimination Backoff Stack, Combining Trees,

 978-1-7281-0339-6/19/$31.00 ©2019 IEEE

Diffracting Trees, Counting Network, Cliff Click Hash Table, Split-ordered lists, etc. [6]. The disadvantages of non-blocking data structures include the high complexity of their development (compared to other approaches), the lack of hardware support for atomic operations for large or non-adjacent memory (double compare-and-swap, double-width compare-and-swap), problems of freeing memory (ABA problem).

Within the approach of transactional memory, the program organizes transactional sections in which the protection of shared memory areas is implemented. Transaction is the finite operations sequence of the transactional read / write actions [6,7]. The transactional read operation copies the contents of the specified section of shared memory into the corresponding section of the local thread memory. The transactional write copies the contents of the specified section of local memory into the corresponding section of shared memory acces-sible to all threads. After completion of the execution, the transaction can either be committed or canceled. Committing a transaction implies that all changes made to memory become irreversible. There are software (LazySTM, TinySTM, GCC TM, etc.) and hardware (Intel TSX, AMD ASF, Oracle Rock, etc.) transactional memory. Among the shortcomings of the transactional memory, it is possible to distinguish restrictions on certain types of operations within the transactional sections, overhead in tracking changes in memory, the complexity of debugging the program.

Given the above-discussed shortcomings of the existing synchronization tools, their performance may be insufficient for modern multi-threaded programs. In this paper, the method of increasing scalability is used due to relaxation of the semantics of data structures.

II. RELAXED CONCURRENT DATA STRUCTURES

The basis of this approach is a compromise between scalability (performance) and the correctness of the semantics of operations. It is proposed to relax the semantics of the implementation of operations to increase the possibility of scaling. For example, when searching for the maximum element in an array, a thread can skip area of other arrays blocked by other threads to improve the performance of the search operation, while the accuracy of the operation is lost.

The principle of quasi-linearizability [4] is applicable to this approach, which assumes that several events may occur during the execution of some operations, simultaneously changing the data structure in such a way that after performing one of the operations, the state of the data structure is undefined. Thus, the result of the operation should not coincide with the implied.

In most existing non-blocking thread-safe structures and blocking algorithms, there is a single point of execution of operations on the structure. For example, when inserting an element into a queue, it is necessary to 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 relaxed data structures, a single structure is replaced by a set of simple structures, the composition of which is considered as a logical single structure. As a result, the number of possible points for access to this structure increases, thus avoiding the occurrence of bottlenecks.

In this approach, each simple structure is usually protected by a lock. When performing an operation, the thread accesses a random structure from the set and tries to lock it. In case of successful locking of the structure, the thread completes the operation; otherwise, the thread randomly selects a new structure. Thus, synchronization of threads is minimized, losses in the accuracy of operations are acceptable [8]. The main representatives of relaxed data structures are SprayList, k-LSM, Multiqueue.

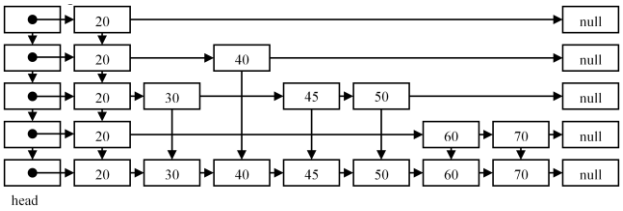

Fig. 2. SprayList relaxed concurrent data structure

SprayList [9] (Fig. 2) is based on the SkipList structure [10]. SprayList is a connected graph, where at the lower level of the structure there is a connected sorted list of all elements, and each next level with a given fixed probability contains the elements of the list of the lower level. The search for this structure is carried out linearly from top to bottom and from left to right; one pointer follows each iteration. Unlike the SkipList, SprayList assumes not a linear search from top to bottom and from beginning to end, is uses a random movement from top to bottom and from left to right. If the search fails or another thread locks the item, the algorithm returns to the previous item. After finding the necessary element, the operations with the list are performed in the same way as in the SkipList. However, in the worst case, inserting items into a SprayList causes significant overhead due to the need to maintain an ordered list.

The log-structured tree with merge (LSM) is used as the basic structure of k-LSM [11]. Each structure change is recorded in a separate log file, tree nodes are sorted arrays (blocks), each of which is at the L level of the tree and can contain N elements ($2L - 1 < N \leq 2L$). Each thread has a local distributed LSM. Common LSM is the result of the merging of several distributed LSM structures. All threads can access common LSM using a single pointer. As a result of combining common and distributed LSM structures, a k-LSM structure was obtained (Fig. 3). When performing the insert operation, the thread stores the element in the local LSM structure. During a delete operation, the search for the smallest key in the local LSM is used. If the local structure is empty, and not insert operation is required, an attempt is made to access foreign LSM structures, the search is performed among all distributed and common LSM structures, and if it was found the structure that is not locked, an operation is performed on it. The disadvantages of this structure are the synchronization of calls to a common LSM and an attempt to call someone else's LSM, since there is no guarantee that it is not empty.

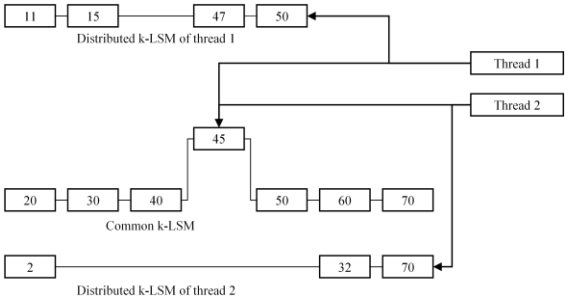

Fig. 3. k-LSM relaxed concurrent data structure

Multiqueues [12] (Fig. 4) is a composition of simple priority queues protected by locks. Each thread has two or more priority queues. The operation of inserting an element is performed in a random, unlocked by another thread, queue. The operation of deleting an element with the minimum key is performed as follows: two random unlocked queues are selected, their values of the minimum elements are compared and the element with the smallest value from the corresponding queue is deleted. This element is not always the minimum inserted in the global structure, but it is close to the minimum and for real problems, this error can be neglected.

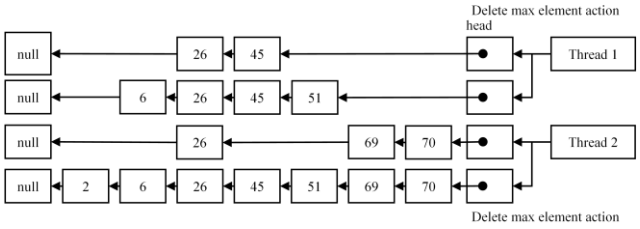

Fig. 4. Multiqueues relaxed concurrent data structure

This paper proposes algorithms to optimize the execution of operations for priority queues based on Multiqueues. Algorithms allow you to optimize the choice of structure for performing the operation.

III. Operation execution optimization

The disadvantage of the current implementation of insert and delete operations in Multiqueues is the algorithm for searching for a random queue. The thread performing the operation most likely turns to queues locked by other threads. The structure of Multiqueues includes *kp* queues, where *k* is the number of queues per thread, *p* is the number of threads.

A. Evaluation Queue Identifier by Thread

Heuristic selecting queue algorithms performing an operation are constructed. Methods are proposed for reducing the number of collisions based on the limitation of the range for randomly choosing a structure. By collisions is meant access a memory area that was already locked by another thread. The set of queues and threads is divided in half. Each thread during the selection of the queue refers to only half of all queues, thereby reducing the likelihood of choosing a locked queue.

Let i in (1) be the thread identifier, then for the first half of all threads (2), the random queue selection operation is performed among the queues (3), where q is selected to perform the operation queue in the Multiqueue structure. For the second half of threads (4), queue searching in the second half of queues (5).

$$i \in \{0, 1, \ldots, p\} \quad (1)$$

$$i \in \{0, 1, \ldots, \lfloor p / 2 \rfloor\} \quad (2)$$

$$q \in \{0, 1, \ldots, \lfloor kp / 2 \rfloor\} \quad (3)$$

$$i \in \{\lfloor p/2 \rfloor + 1 \ldots, p\} \quad (4)$$

$$q \in \{\lfloor kp / 2 \rfloor + 1, \ldots, kp \rfloor\} \quad (5)$$

An approach to optimizing the selection of queues, based on the "binding" of queues to threads, has also been developed. This scheme allows you to specify the order in which the thread accesses queues. In the implementation of fastening, the following model is used: in total, the set contains *kp* queues, then each thread has (6) fixed queues. When performing an operation, the thread first turns to the one of (6) queue, thereby minimizing calls to blocked queues. If all queues from the set (6) are blocked, then the original queue selection algorithm among all queues is used.

$$q \in \{pi, \ldots, pi + k\} \quad (6)$$

B. Implementation of Optimization Algorithms

Algorithms presented in this section have the following semantics: Lock/Unlock – lock and unlock thread mutex for specified queue, isLocked – check that queue is not processing by another thread, RandQueue – return one queue from set of all queues by its identifier that in the specified range, Rand2Queue – same as a RandQueue, but returns pair of queues.

Algorithm 1 represents an optimized insertion algorithm (OptHalfInsert) of an element in a Multiqueues structure. The queue is selected depending on which half of the threads the current thread identifier belongs to.

```
do
   if i ∈ {0, 1, …, p / 2} then
      q = RandQueue(0, kp/2)
   else
      q = RandQueue(kp / 2 + 1, kp)
   end
while isLocked(q)
Lock(q)
q.insert(Element)
Unlock(q)
```

Algorithm 1. OptHalfInsert

Algorithm 2 provides pseudo-code of the optimized algorithm for deleting the maximum element (OptHalfDelete), the queue is selected in the same way as in the OptHalfInsert algorithm.

```
do
   if i ∈ {0, 1, …, p / 2} then
```

```
        (q1, q2) = Rand2Queue(0, kp/2)
    else
        (q1, q2) = Rand2Queue(kp / 2 + 1, kp)
    end
    q = GetMaxElementQueue(q1, q2)
while isLocked(q)
Lock(q)
q.removeMax()
Unlock(q)
```

Algorithm 2. OptHalfDelete

Algorithm 3 contains the pseudo-code of an alternative optimized algorithm for deleting the maximum element (OptExactDelete), first the algorithm try to selects a queue from the "attached" to the thread, and only after failing it search queues among all.

```
do
    if iteration == 0 then
        (q1, q2) = Rand2Queue(pi, pi+k)
    else
        (q1, q2) = Rand2Queue(0, kp)
    end
    ++iteration;
    q = GetMaxElementQueue(q1, q2)
while isLocked(q)
Lock(q)
q.removeMax()
Unlock(q)
```

Algorithm 3. OptExactDelete

IV. Balancing Algorithm

As a result of continuous running program, there may be an imbalance in a relaxed priority queues: some queues may contain significantly more elements than others. This circumstance leads to a decrease in the performance of the algorithms, since empty queues become unsuitable for a delete operation, which increases the search time for suitable queues to perform the operation. A balancing algorithm (Algorithm 4) for the complete Multiqueues structure has been created, for a uniform distribution of elements among the queues.

```
q1 = FindLargestQueue()
q2 = FindShortestQueue()
if q1.size() > AvgSizeOfAllQueues()*0.2 then
        Lock(q1)
        Lock(q2)
        sizeToTransfer = q1.size()*0.3
        TransferElements(q1, q2, sizeToTransfer)
        Unlock(q1)
        Unlock(q2)
end
```

Algorithm 4. Balancing

V. Evaluation

A. Testing Platform

All experiments reported in this paper were processed on one node of cluster, which equipped with dual-socket Intel Xeon X5670 where each socket contains six cores (Hyper-Threading is disabled).

TABLE. Target Machine Specification

Processor	**Intel Xeon X5670**
Clock Frequency	2.93 GHz
Number of Sockets	2
Cores per Socket	6
L3 Size	12 MB
L2 Size	256 KB
L1 Size	6 × 32 KB
RAM	16 Gb DDR3 1066 MHz

B. Measurement Technique

As an indicator of efficiency, the throughput was used, which is calculated as the sum of the carrying capacities of the threads (7), where n is the number of insert / delete operations by the thread i, t is the execution time of operations.

$$b_i = n / t \tag{7}$$

C. Benchmarks

The first experiment, the effectiveness of the original and optimized multiqueues was compared. Individual insert / delete operations were investigated. Each thread was allocated $k = 2$ queues. Each thread performed $n = 10^6$ insert operations and $n = 0{,}5 \times 10^6$ delete operations.

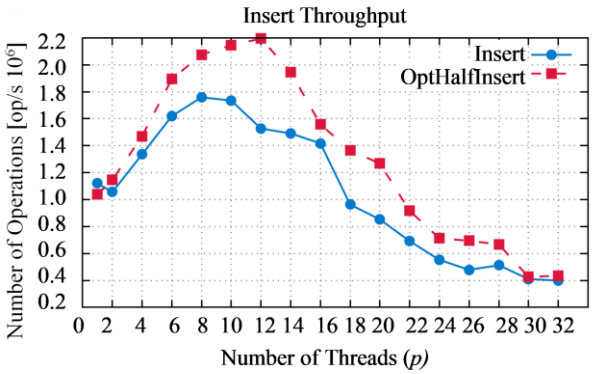

Fig. 5. Insert Throughput by Threads

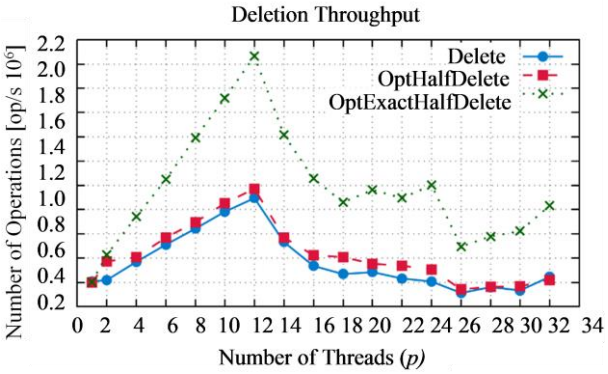

Fig. 6. Deletion Throughput by Threads

The following experiment shows the dependence of the number of random operations (insertion, deletion) on the number of threads used. The following options for using the insertion and deletion algorithms were analyzed:

1. The original insertion algorithm (Insert) and the original element deletion algorithm (Delete)
2. Optimized insertion algorithm (OptHalfInsert) and optimized element deletion algorithm (OptHalfDelete)
3. Optimized insertion algorithm (OptHalfInsert) and an alternative optimized element removal algorithm (OptExactDelete)

The graph at fig. 7 shows the throughput dependencies of random operations for the original and optimized algorithms. From the graph, it can be seen that random operations OptHalfInsert and OptExactDelete on 12 threads show throughput 30% higher than random operations in the original insert and delete algorithms.

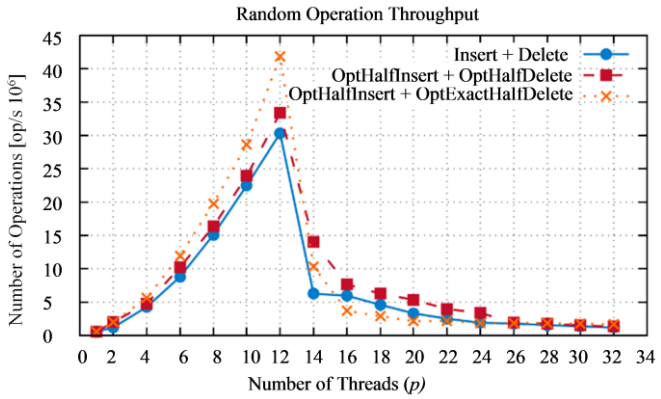

Fig. 7. Random Operation Throughput

The analysis of the optimal number of k queues per threads (Fig. 8 and fig. 9). We used a fixed number of threads p = 12. The algorithms OptHalfInsert and OptExactDelete has the maximum throughput of the system is achieved when the number of queues k = 4 per thread. However, when using the original insert / delete algorithms or OptHalfDelete, it is more advantageous to increase the parameter k, since in this case, the probability of a collision decreases and, as a result, the search time for an unlocked resource decreases.

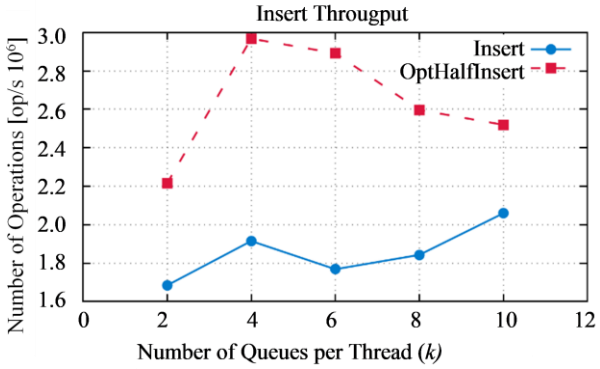

Fig. 8. Insert Throughput by Number of Queues

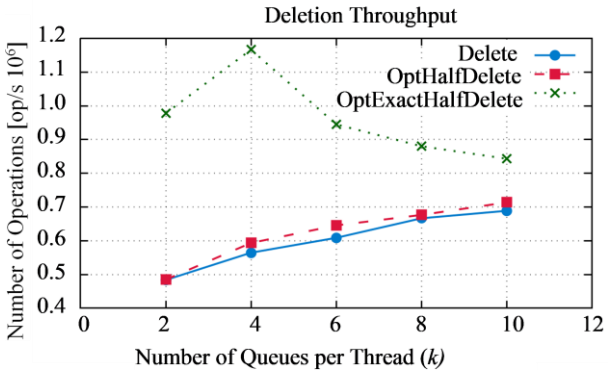

Fig. 9. Deletion Throughput by Number of Queues

VI. Conclusion

An optimized version of a relaxed concurrent priority queue based on Multiqueues has been developed. The developed insertion and deletion algorithms has a 1.2 and 1.6 times performance boost, respectively, compared to the original insertion and deletion algorithms. Optimization is achieved by reducing the number of collisions based on the limitation of the random structure selection.

The implementation of these algorithms is publicly available at https://github.com/Komdosh/Multiqueues.

Acknowledgment

We gratefully acknowledge the computing resources provided and operated by Novosibirsk State University. We thank to Vladislav Kalyuzhny for supporting as to work with computing cluster.

References

[1] TOP500 supercomputers list URL: https://www.top500.org/news/summit-up-and-running-at-oak-ridge-claims-first-exascale-application/ (date of visit 25.06.2018).

[2] Dongarra J. Report on the sunway taihulight system. 2016. pp. 12-13. URL: http://www.netlib.org/utk/people/JackDongarra/PAPERS/sunway-report-2016.pdf / (date of visit 12.05.2018).

[3] Herlihy M., Shavit N. The art of multiprocessor programming, Morgan Kaufmann, Boston, 2011, p. 141-174.

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

[5] Cederman D. et al. Lock-free concurrent data structures, Programming Multicore and Many-core Computing Systems, V. 86, 2017, p. 59.

[6] Shavit N., Touitou D. Software transactional memory, Distributed Computing, V. 10, 1997, pp. 99-116.

[7] Kulagin I. I., Means of architectural-oriented optimization of parallel program execution for computing systems with multilevel parallelism, NUSC, Novosibirsk, 2017, pp. 77-82

[8] A. V. Tabakov, L. M. Veretennikov, Relaxed Concurrent Data Structures, Science of Present and Future Digest of Conference, ETU, Russia, 2018, pp. 105-107

[9] Alistarh D. et al. The SprayList: A scalable relaxed priority queue, ACM SIGPLAN Notices, V. 50, 2015, pp. 11-20.

[10] Pugh W. Skip lists: a probabilistic alternative to balanced trees, Communications of the ACM, V. 33. 1990, pp. 668-676.

[11] Wimmer M. et al. The lock-free k-LSM relaxed priority queue, ACM SIGPLAN Notices, V. 50, 2015, pp. 277-278.

[12] Rihani H., Sanders P., Dementiev R. Multiqueues: Simpler, faster, and better relaxed concurrent priority queues, arXiv pre-print arXiv:1411.1209. 2014 URL: https://arxiv.org/pdf/1411.1209.pdf (date of visit 11.03.2018).

[13] Blumofe R. D., Leiserson C. E. Scheduling multithreaded computations by work stealing, Journal of the ACM (JACM), V. 46, 1999, pp. 720-748.

[14] P. Sanders. Randomized priority queues for fast parallel access, Journal Parallel and Distributed Computing, Parallel and Distributed Data Structures, 1998, pp. 86-97

[15] T.A. Henzinger et al. Quantitative relaxation of concurrent data structures, V. 48, ACM SIGPLAN Notices, 2013, pp. 317-328.