Blocking and non-blocking concurrent hash tables in multi-core systems

ÁKOS DUDÁS
Budapest University of Technology and Economics
Department of Automation and Applied Informatics
1117 Budapest, Magyar Tudósok krt. 2 QB207
HUNGARY
akos.dudas@aut.bme.hu

SÁNDOR JUHÁSZ
Budapest University of Technology and Economics
Department of Automation and Applied Informatics
1117 Budapest, Magyar Tudósok krt. 2 QB207
HUNGARY
juhasz.sandor@aut.bme.hu

Abstract: Widespread use of multi-core systems demand highly parallel applications and algorithms in everyday computing. Parallel data structures, which are basic building blocks of concurrent algorithms, are hard to design in a way that they remain both fast and simple. By using mutual exclusion they can be implemented with little effort, but blocking synchronization has many unfavorable properties, such as delays, performance bottleneck and being prone to programming errors. Non-blocking synchronization, on the other hand, promises solutions to the aforementioned drawbacks, but often requires complete redesign of the algorithm and the underlying data structure to accommodate the needs of atomic instructions. Implementations of parallel data structures satisfy different progress conditions: lock based algorithms can be deadlock-free or starvation free, while non-blocking solutions can be lock-free or wait-free. These properties guarantee different levels of progress, either system-wise or for all threads. We present several parallel hash table implementations, satisfying different types of progress conditions and having various levels of implementation complexity, and discuss their behavior. The performance of different blocking and non-blocking implementations will be evaluated on a multi-core machine.

Key–Words: blocking synchronization, non-blocking synchronization, hash table, mutual exclusion, lock-free, wait-free

1 Introduction

Parallel programs have been around for more than a quarter of a century, and recently they have moved into the focus of research again. The general availability of multi-core processors brings the demand for high performance parallel applications into mainstream computing promoting new design concepts in parallel programming [1].

The most crucial decision when designing parallel data structures and algorithms is finding an optimal balance between performance and complexity. Shavit in [1] argues that "concurrent data structures are difficult to design." Performance enhancements often result in complicated algorithms requiring mathematical reasoning to verify their correctness, whereas simpler solutions fall short of the required performance or have other unfavorable properties.

Critical sections assure correct behavior by limiting access to the shared resources. Every runtime environment supports various types of synchronization primitives supporting this type of parallel programming, and they usually require only a few changes in code or data. Critical sections and other primitives correspond to the level of abstraction of the programming environment making the life of programmers easier. But simplicity comes with the cost of unfavorable properties, such as the risk of deadlocks, livelocks, priority inversion, and unexpected delays due to blocking [2, 3, 4, 5], not mentioning the increased risk of programming errors [6].

Non-blocking synchronizations promote themselves as true alternatives to locking. Such algorithms employ atomic read-modify-write operations supported by most modern CPUs. The basic idea is that instead of handling concurrency in a pessimistic way using mutual exclusion, concurrent modifications of a data structure are allowed as long as there is a fail-safe guarantee for detecting concurrent modification made by another thread. (Rinard used the term "optimistic synchronization" to describe this behavior [5].) Complex modifications are performed in local memory and then in an atomic replace step they are committed into the data structure.

The biggest advantage of non-blocking synchronization is complete lock-freeness. As there are no locks, the problems mentioned at blocking synchronization do not arise. These methods also provide better protection from programming errors and from

faulty threads not releasing locks, resulting in higher robustness and fault tolerance [7]. The tradeoff with non-blocking algorithms is that they are more complex and harder to design, even when they seem simple [8].

The right choice depends on the required performance and expected guarantees. There are two types of requirements: safety and liveness. *Safety* means that data structures work "correctly" without losing data or breaching data integrity. The result of concurrent operations performed on a parallel data structure are correct, according to the linearizability condition [9, 10], if every operations appears to take effect instantaneously at some point between it's invocation and its response. The *liveness* property captures the fact whether the interfering parallel operations will eventually finish their work. Liveness conditions are different for blocking- and non-blocking algorithms. For blocking algorithms, usually deadlock-freedom or starvation-freedom are expected, while non-blocking algorithms can be lock-free [11, 12] or wait-free [13, 14].

In this paper hash tables are used to illustrate the wide range of design options for parallel data structures. We discuss the implementation details and the performance costs of applying various approaches designing a concurrent hash table.

The rest of the paper is organized as follows. Section 2 presents the related literature of parallel algorithms and hash tables. Lock based solutions are discussed in Section 3 followed by the non-blocking algorithms in Section 4. After the implementation details, their performance is evaluated in Section 5 through experiments. The conclusions are discussed in Section 6.

2 Blocking- and non-blocking synchronization in hash tables

Hash tables [17] store and retrieve items identified with a unique key. A hash table is basically a fixed-size table, where the position of an item is determined by a hash function. The hash function is responsible for distributing the items evenly inside the table in order to provide O(1) lookup time. The difference between the two basic types of hash tables (bucket- and open address), is the way they handle collisions, i.e. when more than one item is hashed (mapped) to the same location. Bucket hash tables chain the colliding items using external storage (external means not storing within the original hash table body), while open-address hashing finds a secondary location within the table itself.

Hash tables are perfect candidates for optimization in parallel environments. The hash table itself is an abstract concept and there are various implementation options, which all demand different parallelization approaches. Best performance is observed when the hash table uses arrays as external storage [18, 19, 20], which will require structural changes in case of a non-blocking implementaion.

The most notable downside of blocking synchronization is the level of parallelism they provide; or rather they fail to provide. Limiting the number of parallel data accesses to a single thread at one time by a single critical section guarding the entire hash table significantly reduces overall performance. To overcome this issue, finer grained locking can be applied. Larson et al. in [21] use two lock levels: a global table lock and a separate lightweight lock (a flag) for each bucket. The implementation by Lea in [22] uses a more sophisticated locking scheme with a smaller number of higher level locks (allocated for hash table blocks including multiple buckets) allowing concurrent searching and resizing of the table. When considering large hash tables with a couple million or more buckets the use of locks in high quantities can be challenging. Most environments do not support this many locks, and with custom implementations one has to be concerned with memory footprint and false sharing limiting the efficiency of caches [24, 25, 26].

There is no general consensus about blocking synchronization. Some argue that mutual exclusion locks degrade performance without any doubt [5, 11] and scale poorly [16], while others state that contention for the locks is rare in well designed systems [2, 27], therefore contention does not really impact performance [28]. The strongest reservation about blocking synchronization, and what is probably most relevant for advocating non-blocking solutions, is that they build on assumptions, such as critical sections are short, and the operating system scheduler is not likely to suspend the thread while it is in the critical section. These assumptions, however, may not always hold.

To ensure greater robustness and less dependency on the operating system a parallel data structure can be implemented using the compare-and-exchange/compare-and-swap (CAS) operations. This method is founded on the fact that all modern architecture provide hardware support for atomic combined read-write operations (e.g the CMPXCHG instruction in the x86 architecture) that allow the non-destructive manipulation (setting the value of a memory location with somehow preserving its old content) of a single machine word. Herlihy describes the basic concept of transforming sequential algorithms into non-blocking protocols this way [29]: first the object to be modified is copied into local memory in its current state, then the modifications are performed on this local copy in-

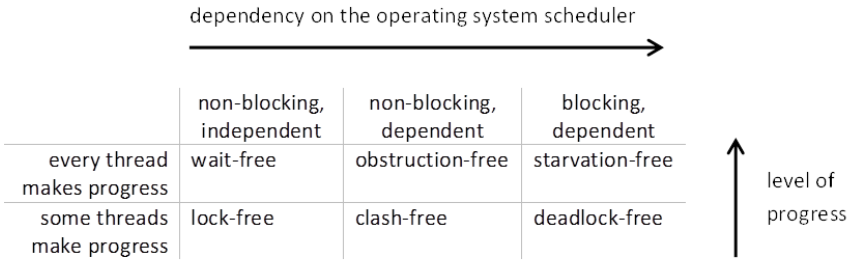

Figure 1: Progress conditions of parallel data structures described by Herlihy and Shavit [33].

stance, and finally the change is committed back using the CAS instruction. If another thread has modified the same data in the meantime, the operation is retried.

With this method complete lock-free data structures and algorithms can be built, as it was to linked list by Harris [30], Valois [12] and Michael [25], whose works were further extended to resizable hash tables by Shalev and Shavit in [31]. Both Gao el al. [8] and Purcell and Harris [32] built a lock-free open address hash tables.

Non-blocking algorithms are generally hard to design. This is in fact the case for parallel hash tables built on the CAS primitives. The entire organization of the data has to be revised: arrays having given the best performance in single threaded case are no longer a storage option as an additional level of indirection has to be introduced into accessing an element in order to allow transactional modification with a single CAS operation. Failing to perform CAS successfully the complete transaction has to be retried which requires restructuring of code to allow this fallback execution path. All in all, a completely new hash table is required with careful design of placement and alignment in memory.

Other concerns about non-blocking implementations include the fact that atomic operations are more expensive than simple writes [8]. Michael even went as far as saying that they are "inefficient" [11], and so are the algorithms, that avoid the use of locks [16].

There is an inherent hierarchy, or rather structure among both the blocking and the non-blocking algorithms with regards to the assumptions they are based on, and the level of progress they provide. Several liveness conditions are known in the literature for quite some time, yet there has been no handle on their relation until recently. Herlihy and Shavit presented a "unified explanation" [33] of the progress properties. They placed these properties in a two-dimensional structure, where one axis defines maximal or minimal progress (informally: expect at least a single thread minimal, or all threads to complete actions - maximal); the other axis defines the assumptions of the operating system scheduler. Algorithms are either blocking or non-blocking, and they can depend on the scheduler or can guarantee the level of progress regardless of schedulers support. Fig. 1 describes this structure of progress conditions (as in [33]).

This system of classification of parallel algorithms helps with comparing different implementations focusing on their merits both in terms of performance and in robustness. Wait-free algorithms are the "best" but usually the most complex, while lock-free algorithms are simpler to create. There is a similar relationship between starvation-free and deadlock-free algorithms, given that they are also more dependent on the operating system. The practical importance of the obstruction-free and clash-free conditions is yet to be discovered.

3 Blocking hash tables

Blocking synchronization mechanisms are usually easier to implement. Locks and critical sections are inherent part of all programming languages and environments and they are described on an abstraction level the programmer is familiar with, therefore is is straightforward to use them.

3.1 Deadlock-free hash table

Being conceptually simpler does not mean that blocking algorithms require no design. The basic expectation from lock-based data structures is to be **deadlock-free**, that is, threads cannot delay each other forever. The simplest way of guaranteeing deadlock-freedom is using a single lock in the data structure, which basically serializes all accesses. This single lock protects the data structure at each entry point. Alg. 1 shows the implementation of the insert method in a deadlock-free hash table. A similar guard is to be

Algorithm 1 Deadlock-free concurrent hash table using a single table lock.

```
initialize:
   table_lock: mutex

insert( data ):
   table_lock.enter()        // acquire lock, blocks until successful
   insert_internal( data )   // perform the insert as usual
   table_lock.release()      // release the lock
```

Algorithm 2 Deadlock-free concurrent hash table using fine-grained locking.

```
initialize:
   bucket_locks[]: mutex

insert( data ):
   bucketIdx = hash( data )            // maps the item to a bucket
   bucket_locks[bucketIdx].enter()     // acquire lock, blocks until successful
   insert_internal( data )             // perform the insert as usual
   bucket_locks[bucketIdx].release()   // release the lock
```

placed around each operation to make the data structure completely thread-safe and deadlock-free.

More sophisticated locking mechanisms use multiple locks. Fine-grained locking can deliver much better level of parallelism and performance. For example in case of bucket hash tables each bucket can be protected with a dedicated lock. Since each basic operation (except for rename) works in a single bucket, the implementation is still deadlock-free. In case of large hash tables, it might not be feasible to use tens of thousands, or even millions of locks. In such cases, the buckets can be assigned to groups, where each bucket is only member of one group but one group contains multiple buckets [23]. Protecting each group with a single lock delivers good level of parallelism (depending on the number of threads and the number of groups). Alg. 2 shows the deadlock-free hash table with fine-grained locking.

The only requirement about the locks used in the aforementioned implementations is that they work correctly, that is, they allow a single thread to reside within the critical section. There is no fairness guarantee, such as allowing the access to the critical section only in the same order as it was requested. Simple spin-locks (also known as busy-wait loops) spinning on a test-and-set instruction [34, 24] as well as mutexes offered by the programming environment can be used. Other options include using reader-writer locks, which allow concurrent readers and a single writer. This type of lock, as long as only a single one is accessed by each operation, is still deadlock-free; the only difference is that the appropriate type of access (shared or exclusive) is determined by the operation (i.e. find requires read access while insert requires write access).

3.2 Starvation-free hash table

Deadlock-freedom guarantees only that there are threads that make progress (specifically the one that currently owns the lock). It does not specify that all threads must complete their operations. A more restrictive condition is **starvation-freedom**, which requires that *all* threads get the access to the critical section eventually. With test-and-set locks this cannot be guaranteed, as it is possible that a thread always looses in the race for the lock.

In order to guarantee starvation freedom with mutual exclusion, threads should enter the critical section in the same order as they request it. Queue-locks satisfy this requirement. Using Mellor-Crummey and Scott's *ticket lock* [24] the bucket hash table is modified as shown in Alg. 3 for a single table lock; fine-grained bucket level locks can be implemented similarly. (The *interlocked-inc* instruction increases the value of the integer in an atomic step and returns the new value.)

4 Non-blocking hash tables

Their undesired properties often limit the usability of blocking synchronizations. Even in the absence of deadlocks and starvation, the level of parallelism is limited: threads hinder each others progress causing priority inversion and problems that occur when a thread crashes and does not release a lock by error. Non-blocking synchronization mechanisms promote themselves as alternatives to lock based methods offering better scalability and robustness.

Algorithm 3 Starvation-free concurrent hash table using a single table lock.

```
initialize:
   currently_served: int, initially 1
   ticket_counter: int, initially 0

insert( data ):
   my_ticket = interlocked_increment( ticket_counter )   // get ticket
   while( my_ticket != currently_served )                // wait until my ticket is served
       nop                                               // no operation, spin-wait
   insert_internal( data )                               // perform the insert as usual
   interlocked_increment( currently_served )             // notify next in line
```

4.1 Linked list-based lock-free bucket hash table

The sensitive operation within the hash table is committing a new entry into the bucket, or removing an entry. Using a linked list for realizing the buckets the hash table can be implemented in a non-blocking fashion. Alg. 4 shows the CAS based linked-list lock-free bucket hash table algorithm. This algorithm works only if there are no deletes from the hash table. Deletes can be supported as well, with some modifications [12].

It is easy to see that this implementation is lock-free. There are no locks or mutual exclusion in the code, hence there are no indefinite delays. That is to say, there is one case: the CAS operation can fail if the target memory location has been changed by another thread. It will, however, always succeed for a single thread, thus the data structure is lock-free. As for correctness, this algorithm is linearizable, the CAS being the linearization point (where the changes appear atomically for the entire system).

4.2 Array based lock-free bucket hash table

Linked-lists are not always the best choice for bucket hash tables. Data prefetch mechanisms and cache lines containing multiple items mask the memory access latency when iterating through the items in arrays while prefetching is of no use when traversing a linked list. This can yield an increase in the amount of cache misses deprecating performance.

The incompatibility between compare-and-swap and arrays lies in the fact that CAS works with machine word size values only, meaning that an item in the array larger than 4 or 8 bytes cannot be written atomically. The compromise is as follows. The buckets are implemented with arrays, but when a new item is to be added, a new array is allocated copying the previous contents and adding the new one, then CASing the pointer to this new array into the hash table body (see Alg. 5). Only one concurrent insert will succeed causing the other one to retry. Deleting items works the same way, except the new array will not contain the item to remove. Search operations are not affected by this as long as they hold the pointer to the bucket as they know it.

This implementation is, similarly to the linked-list based one, lock-free but not wait-free for the same reasons discussed previously. The linearization point is the CAS operation.

This approach wastes the new array allocation and copy operation in case of contention for the same bucket, which is an obvious performance drawback. The gain on the other hand is the faster searching capability in the array.

De-allocating the unused arrays is also a problem. When a new array replaces the previous one, the old one should be de-allocated, given that no one else uses it. In garbage collected systems this is handled by the environment, but in other cases this is the responsibility of the programmer. The solution is either reference counting, or if possible, not deallocated memory until it is safe (after parallel operations have been completed).

4.3 Wait-free hash table

The last hash table presented in this section takes a different approach compared to all previously discussed implementations. All the solutions above required, to some extent, the modification of the hash table code: placing the locks and the critical sections, reorganizing the entire table into linked lists or use the compare-and-exchange operation. Let us present an algorithm, which uses no locks and is not only non-blocking but completely wait-free, and requires no modifications inside the hash table. To our knowledge this is the first wait-free hash table in the literature.

The main problem with parallelism from the point of view of data structures is concurrent modification of the same storage slot. If different threads worked with different hash tables, this would not be a problem anymore. Instead of the traditional scenario where each thread is capable of executing any work we specialize our threads by assigning individual work areas to them in a way that these working areas do

Algorithm 4 CAS based lock-free concurrent bucket hash table with linked list.

```
typedef node: key, value, next

initialize:
   buckets[]: pointer to node

insert( data ):
   myItem = create( data )                  // creates the node from the data
   l = buckets[ hash( data ) ]              // linked list representing the bucket
   while( true ):                           // loop for CAS retries
      curr = l;
      while curr.next != NULL:              // check each item in the list
         if( matches( data, curr ) ):
            return curr;
         curr = curr.next
      if( CAS( curr.next, myItem, NULL ) )  // link the new item to the end
         return myItem                      // CAS succeeded (item is inserted) or retry
```

Algorithm 5 CAS based lock-free concurrent bucket hash table with array.

```
initialize:
   buckets[]: pointer to array

insert( data ):
   while( true ):                                      // loop for CAS retries
      h =  hash( data )
      myArray = buckets[ h ]                           // the current bucket
      idx = find_in_array( myArray, data )             // check the current data
      if idx != -1:                                    // found
         return myArray[ idx ]
      newArray = allocate_new_copy( myArray, data )    // create new array, copy existing
                                                       //   data and insert new
      if( CAS( buckets[ h ], newArray, myArray ) )     // replace bucket
         return                                        // CAS succeeded (item is inserted)
                                                       //   or retry
```

not overlap (separate functions, pipeline stages, spatial domains or graph branches). A specific request (insert/find) can only be served by one thread, and it is up to the right selection of domains that provides the load balance. The selection rules are usually based on data decomposition as functional decomposition generally provides limited scalability and we easily reface the bottleneck problem of critical sections on the level of control organization. According to our knowledge this kind of cooperation is never mentioned in the literature of shared memory algorithms, but the idea is not unknown in distributed systems, where there low cost shared memory synchronization is not available, thus a coarser grained cooperation must be maintained between the nodes by directed point-to-point messages or implicit work allocation rules (e.g. distributed file servers, horizontally partitioned data bases, documents groups allocated to separate web servers).

The universe of items that the concurrent hash table stores is partitioned into several subtables, and each of these hash table partitions is unambiguously mapped to a single working thread. When a thread is ready to perform work, it checks the next work item in a central queue, and decides if it should process it. The decision is completely decentralized, and all threads must come to the same conclusion: one and only one of them chooses the item for processing. There is no communication between the threads, thus no overhead.

The decision whether to process an item is based on the item itself. Each item is unambiguously mapped to a thread by calculating a hash value from the data. In case of a hash table this is quite easy, as the items have a key, which, by definition, uniquely determines them. (This concept, however, we note, is generalizable to other data structures and problems. For example, sorting can be performed this way as well: the hash function can divide the items by their value: smaller than a threshold goes to one thread, larger to another [35].)

The critical element in this case is the central queue. It must be wait-free for the hash table system to be wait-free, and it must map the items to the requesting threads. There are wait-free queue algorithms in the literature, such as the one by Michael et al. [**?**]. Let n be the number of processing threads (and consequently the number of hash tables). Using

Algorithm 6 Wait-free algorithm for hash tables.

```
initialize:
   hash_tables[]: hash table, one for each thread
   queues[]: wait free queues, one for each thread

addRequest(data):
   h = hash( data )
   queues[h].enqueue( data )

processingThreadFunction():
   while true
     i = queues[threadId].dequeue()      // get item from the threads queue
     h = hash( i )
      hash_tables[threadId].insert( i )  // insert into the table
```

n wait-free queues every processing thread receives a dedicated queue. Assigning an item to a thread needs to hash the data and add it to the right queue. The overhead this time is the pre-calculation of the hash by the process that would generally only insert the request into a queue. Algorithm 6 presents the solution.

The biggest advantage of this solution, besides being wait-free, is that the internal hash tables are unaware that they are used in a multithreaded environment. Any hash table implementation could be used and its source code does not need to be altered.

The difference between Alg. 6 and traditional parallel hash tables is that here the threads need to be aware that they are accessing different hash tables. It is not the responsibility of the hash table to handle parallelism, but it is left up to the threads themselves. This can be viewed as a "relaxation" of the requirements of the data structure, as Shavit calls it [1]. It might require changes in the algorithms using the hash table to support this behavior, but the gain is wait-freedom. (We also note that data parallel algorithms are likely to be easier to adopt to this behavior as they already distribute work among threads in a similar fashion.)

This wait-free implementations cannot be described by any of the generally used correctness criteria. It is not linearizable, as changes are not atomic. It is not sequentially consistent [3] either, because it is not clear which point is the invocation of a hash table operation, and when the method returns: all operations are split into two phases, first insertion into the right queue, and the completing the work. This type of operation is not compatible with the general description of concurrent objects by histories, invocations and response events.

5 Performance evaluation

Vastly different hash table implementations were presented in the previous sections. Some are already known in the literature, while others present new ideas. They all feature different properties: the non-blocking solutions are all more robust, while the lock based tables are simpler. Let us examine how simplicity and compromises in the implementation translates into performance.

All off the parallel hash tables were implemented in C++ with careful optimization. They are all of the same structure: a bucket hash table with either a linked list or an array representing the buckets. The tables were initialized by inserting 8 million items into them, and a subsequent 8 million find queries were executed. The wall-clock execution time was measured. The insert and find operations were executed under two circumstances: in the first case the number of buckets of the hash tables was chosen to result in a single item per bucket on average, thus the contention for the buckets is small (Fig. 2); the latter case used tenth of the number of buckets causing high contention (Fig. 3). The results are an average of 5 independent executions on a state-of-the-art Intel Core i7-2600 CPU with 8 (logical) execution units.

Using a single lock is not evaluated below as it is expected to have very little speedup, if any due to high contention for the single lock. The speedup is measured by comparison to the single threaded execution of the same operations using a very similar hash table implementation without locks or CAS; as a matter of fact, the very same table is used by *rule based cooperation*.

The lock based (*bucket lock w/ spin locks* and *bucket lock w/ ticket lock*) and the *linked list based w/ CAS* have quite similar performance. The maximal speedup is between around 3-3.5 and 4-4.5 for 8 concurrent threads depending on the number of buckets used by the table. The lock-free solutions are both faster than the lock based solutions by a small margin under little contention, but the true power of the lock-free solutions show with high contention: *array based w/ CAS* is by far the best one reaching almost 6.5 speedup. Considering the overhead with the con-

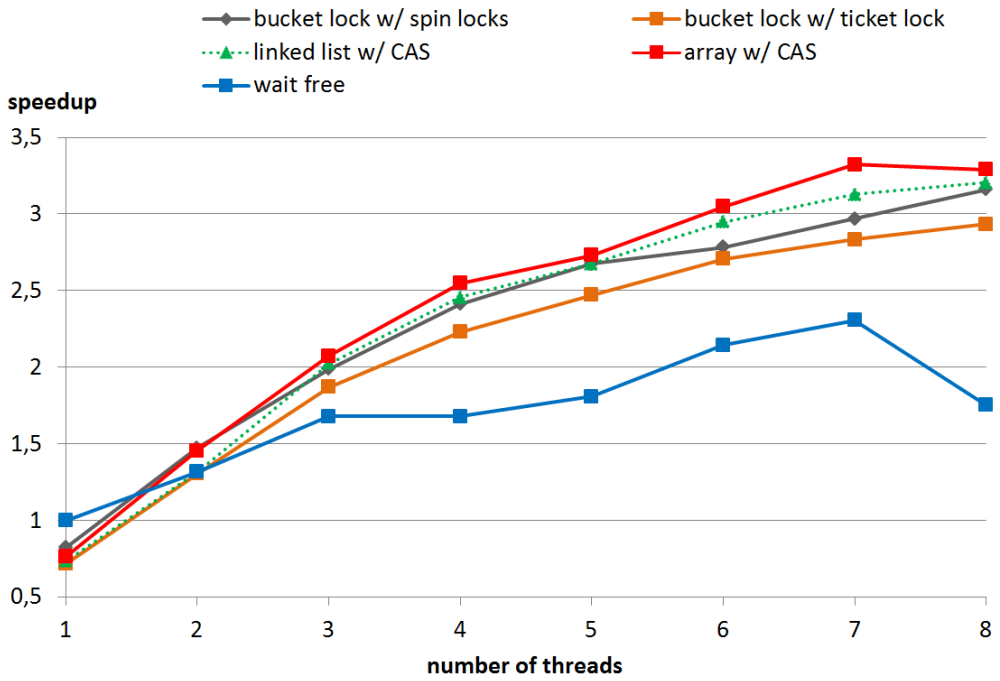

Figure 2: Speedup of the various parallel hash tables with little contention for the buckets.

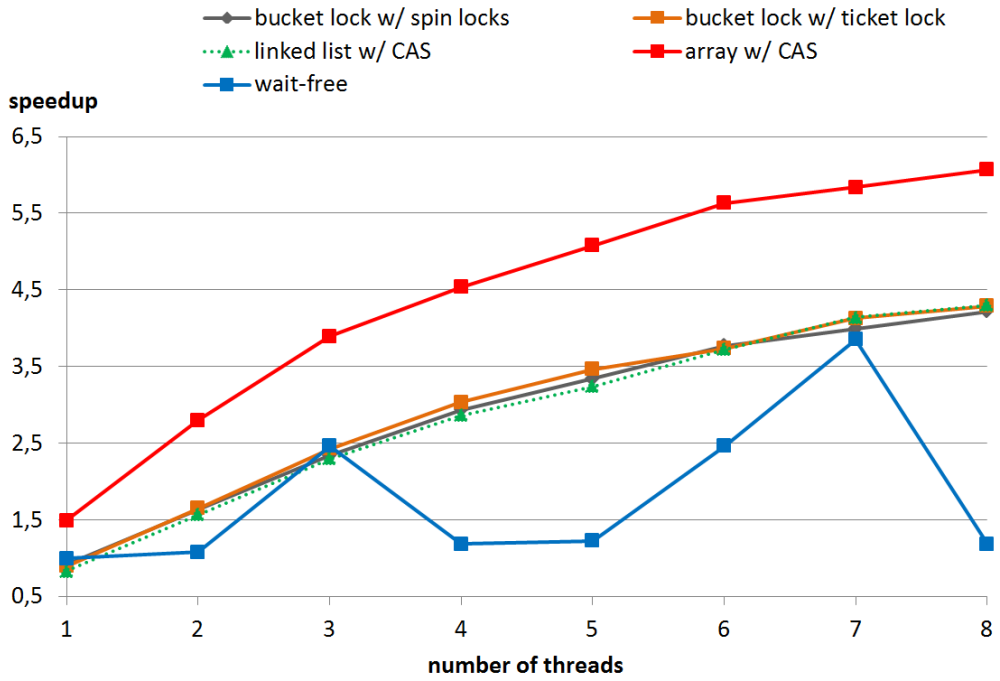

Figure 3: Speedup of the various parallel hash tables with high contention for the buckets.

Table 1: Hash table realizations for the various progress conditions of parallel data structures.

	non-blocking, independent	blocking, dependent
every method makes progress	wait-free w/ queues	table lock w/ ticket lock, bucket lock w/ ticket lock
some method makes progress	linked list-based w/ CAS, array-based w/ CAS	table lock w/ spin lock, bucket lock w/ spin lock

stant array allocations and copy operations can only be due to the good cache behavior and prefetch friendliness exhibited by the array.

The wait-free solution, *rule based cooperation*, achieves a 2.5-3.5 speedup at its peek performance for 7 threads. It is not as fast as either the lock-based or the lock-free solutions, but at the same time it guarantees progress for all threads at all times and that internally it can work with arbitrary hash table implementations. The drop in performance under heavy loads for 4 and 5 threads in Fig. 3 is due to uneven load balancing of items among the threads. This points out another bottleneck in this wait-free implementation.

Lock-free solutions can deliver the same, if not better performance than lock based solutions. The cost is the complexity of the algorithms. The wait-free algorithm requires restructuring of code, but not the structure of the hash table, and in exchange for global progress guarantee we pay with a significant performance loss.

Guaranteeing stronger progress conditions, such as starvation-freeness or wait-freeness, result in performance loss. As an engineering choice the faster but less resilient solutions seem like a good trade-off, while the complicated progress guarantees should only be chosen when they are indeed necessary. Performance in lock-based solutions is manageable by choosing the level of granularity for locking, which is closer to an engineering solution, while there is no such directly tunable parameter for lock-free algorithms.

6 Conclusion

Parallel data structures are hard to design in a way that they remain simple while delivering good performance at the same time. Lock based solutions are simpler, but cannot guarantee much in terms of safety and robustness; non-blocking algorithms are much more complicated, and a compromise is required to balance performance and simplicity. In this paper we used hash tables to illustrate the vast amount of options for creating concurrent data structures. Their structure is relatively simple and both locking and non-blocking algorithms can be implemented with some effort. Table 1 lists the various hash table types organized according to the progress conditions they satisfy [33].

Lock based solutions, when designed carefully, such as using fine-grained locking for regions in the bucket hash table, have really good performance. Such locking solutions are relatively lightweight and inter-thread communication in multi-core machines (via the locks) is very cheap allowing to achieve up to 4.5 times speedup for 8 concurrent threads.

CAS based non-blocking solutions represent a different type of parallel hash tables. Their internal structure require careful redesign, whereas, when carefully tuned, they have the same performance as the lock based solutions. The usual argument that they are more robust is masked by the fact that they are complex and therefore hard to verify.

The final options we presented, *rule based cooperation* is in our knowledge the first wait-free hash table in the literature. It wraps the hash table with an addition layer, which handles the parallel access, thus requires no internal modifications of the original hash table and is still able to achieve a reasonable speedup.

Acknowledgements: This work was partially supported by the European Union and the European Social Fund through project FuturICT.hu (grant no.: TÁMOP-4.2.2.C-11/1/KONV-2012-0013).

References:

[1] N. Shavit, Data structures in the multicore age, Communications of the ACM 54 (3) (2011) 76.

[2] J. H. Anderson, Y.-J. Kim, T. Herman, Shared-memory mutual exclusion: major research trends since 1986, Distributed Computing 16 (2-3) (2003) 75–110.

[3] L. Lamport, How to make a correct multiprocess program execute correctly on a multiprocessor, IEEE Transactions on Computers 46 (7) (1997) 779–782.

[4] H. H. Michael Hohmuth, Pragmatic Nonblocking Synchronization for Real-Time Systems, in: Proceedings of the General Track: 2002 USENIX Annual Technical Conference, 2002.

[5] M. C. Rinard, Effective fine-grain synchronization for automatically parallelized programs using optimistic synchronization primitives, ACM Transactions on Computer Systems 17 (4) (1999) 337–371.

[6] P. Gill, Operating Systems Concepts, 1st Edition, Vol. 2006, Laxmi Publications, New Delhi, India, 2006.

[7] G. E. Blelloch, B. M. Maggs, Parallel algorithms, in: Algorithms and theory of computation handbook, 2010, Ch. 25, p. 25.

[8] H. Gao, J. F. Groote, W. H. Hesselink, Lock-free dynamic hash tables with open addressing, Distributed Computing 18 (1) (2005) 21–42.

[9] M. P. Herlihy, J. M. Wing, Linearizability: a correctness condition for concurrent objects, ACM Transactions on Programming Languages and Systems 12 (3) (1990) 463–492.

[10] M. P. Herlihy, J. M. Wing, Axioms for concurrent objects, in: Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages - POPL '87, ACM Press, New York, New York, USA, 1987, pp. 13–26.

[11] M. Michael, Nonblocking Algorithms and Preemption-Safe Locking on Multiprogrammed Shared Memory Multiprocessors, Journal of Parallel and Distributed Computing 51 (1) (1998) 1–26.

[12] J. D. Valois, Lock-free linked lists using compare-and-swap, in: Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed computing, ACM Press, New York, New York, USA, 1995, pp. 214–222.

[13] M. Herlihy, Wait-free synchronization, ACM Transactions on Programming Languages and Systems 13 (1) (1991) 124–149.

[14] M. P. Herlihy, Impossibility and universality results for wait-free synchronization, in: Proceedings of the seventh annual ACM Symposium on Principles of distributed computing - PODC '88, ACM Press, New York, New York, USA, 1988, pp. 276–290.

[15] M. Herlihy, V. Luchangco, M. Moir, Obstruction-Free Synchronization: Double-Ended Queues as an Example, in: Proceedings of the 23rd International Conference on Distributed Computing Systems, IEEE Computer Society Washington, DC, USA, 2003, pp. 522–259.

[16] H. Attiya, R. Guerraoui, D. Hendler, P. Kuznetsov, The complexity of obstruction-free implementations, Journal of the ACM 56 (4) (2009) 1–33.

[17] D. E. Knuth, The art of computer programming, Vol 3, Addison-Wesley, 1973.

[18] M. Mitzenmacher, Good Hash Tables & Multiple Hash Functions, Dr. Dobbs Journal (336) (2002) 28–32.

[19] G. L. Heileman, W. Luo, How Caching Affects Hashing, in: Proc. 7th ALENEX, 2005, pp. 141–154.

[20] A. Sachedina, M. A. Huras, K. K. Romanufa, Resizable cache sensitive hash table (2003).

[21] P.-A. Larson, M. R. Krishnan, G. V. Reilly, Scaleable hash table for shared-memory multiprocessor system (Apr. 2003).

[22] D. Lea, Hash table util.concurrent.ConcurrentHashMap, revision 1.3, in JSR-166, the proposed Java Concurrency Package (2003).

[23] Á. Dudás, S. Juhász, S. Kolumbán, Performance analysis of multi-threaded locking in bucket hash tables, ANNALES Universitatis Scientiarum Budapestinensis de Rolando Eötvös Nominatae Sectio Computatorica 36 (2012) 63–74.

[24] J. M. Mellor-Crummey, M. L. Scott, Algorithms for scalable synchronization on shared-memory multiprocessors, ACM Transactions on Computer Systems 9 (1) (1991) 21–65.

[25] M. M. Michael, High performance dynamic lock-free hash tables and list-based sets, in: ACM Symposium on Parallel Algorithms and Architectures, 2002, pp. 73–82.

[26] M. Greenwald, Non-blocking Synchronization and System Design, Phd thesis, Stanford University (Aug. 1999).

[27] L. Lamport, A fast mutual exclusion algorithm, ACM Transactions on Computer Systems 5 (1) (1987) 1–11.

[28] F. Fich, V. Luchangco, M. Moir, N. Shavit, P. Fraigniaud, Obstruction-Free Step Complexity: Lock-Free DCAS as an Example, in: P. Fraigniaud (Ed.), Lecture Notes in Computer Science, Vol. 3724 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, Berlin, Heidelberg, 2005, pp. 493–494.

[29] M. Herlihy, A methodology for implementing highly concurrent data structures, in: Proceedings of the second ACM SIGPLAN symposium on Principles & practice of parallel programming - PPOPP '90, ACM Press, New York, New York, USA, 1990, pp. 197–206.

[30] T. L. Harris, A Pragmatic Implementation of Non-blocking Linked-Lists, in: DISC '01 Proceedings of the 15th International Conference on Distributed Computing, 2001, pp. 300–314.

[31] O. Shalev, N. Shavit, Split-ordered lists, Journal of the ACM 53 (3) (2006) 379–405.

[32] C. Purcell, T. Harris, Non-blocking hashtables with open addressing, in: Proceedings of the 19th International Symposium on Distributed Computing, Springer-Verlag GmbH, Krakow, Poland, 2005, pp. 108–121.

[33] M. Herlihy, N. Shavit, On the Nature of Progress, in: A. Fernàndez Anta, G. Lipari, M. Roy (Eds.), Principles of Distributed Systems, Lecture Notes in Computer Science, Vol. 7109 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, Berlin, Heidelberg, 2011, pp. 313–328.

[34] T. Anderson, The performance of spin lock alternatives for shared-money multiprocessors, IEEE Transactions on Parallel and Distributed Systems 1 (1) (1990) 6–16.

[35] E. Horowitz, A. Zorat, Divide-and-Conquer for Parallel Processing, IEEE Transactions on Computers C-32 (6) (1983) 582–585.