Challenges in Real-Time Synchronization*

PHILIPPE STELLWAG and WOLFGANG SCHRÖDER-PREIKSCHAT
Friedrich-Alexander University Erlangen-Nuremberg
Computer Science 4, Martensstr. 1, Erlangen, Germany

Abstract

Upcoming multi-core processors force developers of real-time systems to meet the challenges in real-time synchronization. This paper sketches some results of a novel wait-free, linearizable, and disjoint-access parallel NCAS library, called RTNCAS. It focuses the use in massive parallel real-time systems and enables developers to implement arbitrarily complex data structures in an easy way, while ensuring linearizability, wait-freedom, as well as disjoint-access parallelism. It allows, furthermore, developers to re-use their sequential algorithms without any modifications and care about concurrency. Thereby, the degree of disjoint-access parallelism can be used to trade a low jitter for a higher average-case performance.

1 Introduction

Future CPU generations will offer an increasing amount of parallelism. This fact induces, however, a plenty of challenges, in particular for real-time systems. For the previous decades, increasing clock cycles of uni-processors let real-time systems, such as data and computationally intensive control systems, scale well in terms of performance without violating the time specifications and without further man-power [19]. Thus, scalable performance is not an issue in the field of uni-processor real-time systems. In environments that occupy an increasing amount of parallelism and where clock frequencies of processing elements tend to stagnate, scalable performance, however, becomes a prerequisite. Scalable performance, however, requires the elimination of sequential code, such as critical sections protected by spinlocks, or auxiliary schemes for wait-free computing [6, 13, 5] that usually perform concurrent operations in a strict sequential order. The way of processor manufacturers to continually increase parallelism needs, on the software side, generic mechanisms that allows us to continually concentrate on building complex real-time systems, without digress into building complex [4] special purpose constructs to tackle concurrency. The latter is a rather error-prone process and, hence, induces risks of incorrect real-time systems. These challenges are to be mastered.

In this paper, we sketch some results of our RTNCAS library, introduced previously in a two-paged abstract [16]. RTNCAS tackles all these challenges by offering a linearizable [7], wait-free [6], and disjoint-access parallel [8] NCAS approach that is able to conditionally swap up to 256 words in an atomic manner.

2 System Model

RTNCAS was developed for asynchronous shared memory systems with multiple closely connected processing elements. To show under which conditions RTNCAS works correctly, we shall specify the system model.

First, the number of threads is limited and known in advance. Most real-time applications (especially those with hard timing constraints) fulfill this precondition.

Second, the target machine has to support some atomic instructions. Most multi-core CPUs offer such instructions, i.e. FAA, CAS, and DCAS. Fetch-and-Add (FAA) atomically fetches a memory location and increments it by a given value. By this instruction we can implement atomic increment and decrement operations on word sized counters. Compare-and-swap (CAS) atomically compares a one word sized memory location with a given "old value" and conditionally swaps it to a "new value", if the compare part succeeds. And double-word compare-and-swap (DCAS), which is has the same semantics as CAS, but operates on two consecutive memory words.

Last, the target machine has to guarantee a worst-case execution time (WCET) for these instructions. They may also be implemented in software (e.g., via the load-linked and store-conditional instructions [11]), if the implementation is linearizable and wait-free.

*This work was partly supported by the German Research Foundation (DFG) as part of the Transregional Collaborative Research Centre "Invasive Computing" (SFB/TR 89).

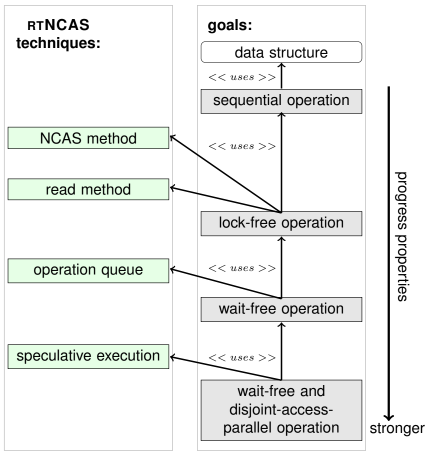

Figure 1: Uses hierarchy [12] of RTNCAS. The right big block depicts the progress properties ensured by the transformation steps on top of a sequential DS operation. Thereby, lower boxes represent stronger properties. The left big block depicts the techniques used within the transformations to achieve these progress properties.

3 RTNCAS

3.1 Terminology

An *NCAS method* is used to atomically change several unrelated variables, if and only if they contain the given expected value. An *NCAS operation* corresponds to an invocation of an NCAS method; the NCAS method implements the NCAS operation.

A *data structure operation* is a user-defined function on a shared data structure (DS), such as enqueue on a FIFO queue. The actual semantics of a DS operation performed by the RTNCAS library is implemented in a user-defined callback function λ activated through an upcall [2] by the RTNCAS library.

3.2 Properties

The user-defined DS operations implemented via RT-NCAS have to be correct and provide strong progress guarantees that allow the definition of upper bounds for their WCET. These properties make such operations applicable in hard real-time applications.

RTNCAS guarantees that such operations are always linearizable. Linearizability [7] defines a correctness condition of concurrent objects. The main benefit of linearizable implementations is the feasibility of local correctness proofs to verify all concurrent execution histories. Implementations of DS operations using RTNCAS also support strong progress properties. To achieve those properties, we use the techniques depicted in Fig. 1.
1. *Lock-free operations* are interrupt-transparent, and neither depend on system libraries nor induce restrictions to the scheduler (we do not consider backoff algorithms, such as [14], here). Under concurrent usage they also frequently yield better performance than blocking counterparts [20]. As no progress guarantees can be given for such operations, it prevented us to determine the WCET of such operations.
2. *Wait-free operations* [6] are lock-free and have a determinable WCET. Hereby, the temporal requirements of real-time applications can be satisfied.
3. *Disjoint-access-parallel operations* [8] offer parallel executions of operations that access to disjoint memory. This yields additional benefits in performance in multi- and many-core environments.

Starting with a sequential implementation of a DS operation, we apply several transformations to it. Each of these transformation ensures the next stronger progress property and finally, a wait-free and disjoint-access-parallel DS operation is reached.

3.3 Design

Sequential operations on data structures rely on exclusive access to the parts of the structure by one unit of execution. Their methods are typically implemented using reads and writes to single words spread out over the whole critical region to manipulate the state of a DS. Fig. 2 depicts an example where three words – comprising the current state of a linked-list – are manipulated when adding an new element. Such operations correspond to sequential operations in Fig. 1.

3.3.1 Lock-Free Operations

In a first step the formerly sequential operation has to be transformed into a lock-free operation. This is accomplished by reading the old state of the DS via a *read method* (see Fig. 1), computing a new state, and, finally, atomically exchanging the old state with the new one by an NCAS operation. In case of concurrent interference the NCAS operation may fail and the complete operation including reading the old state and computing the new state has to be repeated.

The usage of the NCAS operation as described above is both lock-free and linearizable. As depicted in Fig. 1, we require a weakly wait-free NCAS method (requirement **L1**) to support lock-free DS operations and a read method (req. **L2**) to retrieve the actual value from an NCAS word, which are described later in Sec. 4.

3.3.2 Wait-Free Operations

The usage of the NCAS operation within such a retry-loop prevents the estimation of a WCET for the DS operation in the presence of concurrent executions. This is due to the number of retries needed to successfully complete the

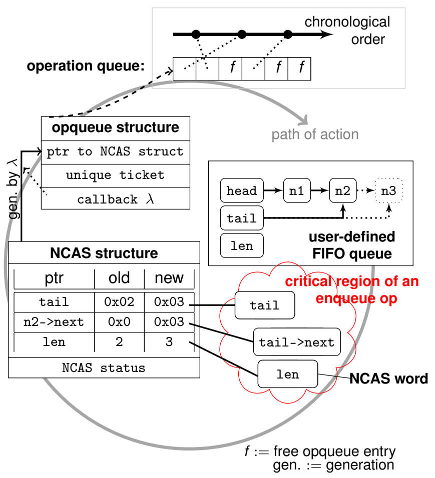

Figure 2: This figure shows all data structures involved in RTNCAS and illustrates an example of adding a node `n3` to a FIFO queue by using RTNCAS. An enqueue operation implemented via locks would modify `tail`, `tail->next`, `len` inside a critical region to ensure atomicity. To add `n3` to the queue using RTNCAS, all words of the DS are encapsulated into NCAS words. The thread performing the enqueue operation has to create an opqueue structure containing a unique ticket to be able to deduce a chronological order and a user-defined callback λ to create an NCAS structure describing the atomic state transformation to enqueue `n3`. This opqueue structure is enqueued to the operation queue providing a wait-free helping scheme, where concurrent threads help each other to perform stalled operations.

DS operation is unpredictable a priori. This is unacceptable under hard real-time conditions. To overcome this, we transform the lock-free operation into a wait-free one by means of a helping scheme implemented via an *operation queue* as depicted in Fig. 2. This operation queue guarantees progress of DS operations built on top of it.

The operation queue is implemented by a wait-free FIFO and works as depicted in Fig. 2: Every thread encapsulates its DS operation into a so called *opqueue structure*. This structure contains a pointer to the callback λ, a pointer to the NCAS structure generated by the callback λ, and a unique ticket to establish a chronological order among all elements within the operation queue.

After enqueuing the own operation, each thread performs the following steps until its own operation is completed: Get the oldest opqueue structure from the operation queue and try to perform the encapsulated operation. The latter is done in the following three steps: 1) The opqueue structure is checked for an active NCAS structure, i.e. an NCAS structure whose associated NCAS operation might still be executed successfully. If one is found, the second step is skipped. 2) The callback λ is used to create a new NCAS structure (see Fig. 2). A thread then tries to replace the reference to the NCAS structure in the opqueue structure with a reference to the newly created one. To ensure that only non-active NCAS structures are replaced, CAS is used here. 3) The reference to the current NCAS structure is reloaded and its NCAS operation is executed. Thus, in case of multiple threads concurrently working on the same opqueue structure, exactly the same NCAS structure is used to perform the actual NCAS operation in a cooperative manner. Finally, in case of successful execution of the NCAS operation, the opqueue structure is dequeued.

The operation queue imposes additional requirements. First, we need a wait-free FIFO that allows us to find and use the oldest entry without removing it. Therefore, we have adapted one of our earlier work [15]. Second, the NCAS implementation has to support cooperation: If all active threads work on the same NCAS operation (by using the same NCAS structure), the described NCAS operation must be completed successfully (req. **W1**), discussed in Sec. 4.5. This helping scheme facilitates the implementation of wait-free DS operations. Thereby, a progress guarantee and an upper bound for the WCET can be given for these DS operations.

3.3.3 Disjoint-Access-Parallel Operations

The helping scheme implemented by the operation queue has one drawback: All DS operations are performed sequentially. This is not disjoint-access-parallel [8], leads to convoy effects [3] (sequential code), and cause poor average performance. To avoid these side effects, we introduce the concept of *speculative execution* (see Fig. 1).

Before enqueueing an opqueue structure into the operation queue each thread tries to execute the DS operation speculatively by the corresponding lock-free operation (see Sec. 3.3.1). If the speculative execution fails, an opqueue structure will be enqueued into the operation queue and the thread carries on as described in the previous section. If the speculative execution is successful, the thread still works on the oldest entry of the operation queue. This is required to guarantee progress for the oldest entry in the operation queue. Otherwise, the oldest entry might starve due to continuously interfering speculative executions. By forcing every thread to execute at least one opqueue structure from the operation queue, it can be guaranteed that the oldest entry in the operation queue is completed in finite time.

The speculative execution creates one new requirement for the NCAS implementation: It has to be able to deal with different priorities of NCAS operations (req. **D1**). An NCAS operation using an opqueue structure stalled within the operation queue has a higher priority than NCAS operations executed speculatively.

Using speculative execution, finally, we support the implementation of wait-free and disjoint-access-parallel DS operations (see Fig. 1). The degree of disjoint-parallel accesses can, furthermore, be chosen with respect to the number of speculative executions that can take place concurrently. Hereby, a low jitter can be traded for a better average performance.

4 NCAS

4.1 NCAS Structure

Every NCAS operation is wrapped into a structure, an *NCAS structure*, that, as depicted in Fig. 2, contains the addresses of the variables that are to be changed, their expected old values, their respective new values, and a status variable. The status variable can adopt the states WORKING, SUCCESS, WRONG_VALUES, and KILLED. Initially it is WORKING and during the lifetime of the structure, it can only be changed from WORKING to a different state. SUCCESS indicates a successful execution of the NCAS operation, WRONG_VALUES means that the values of the variables to be changed differ from the expected old values, and KILLED denotes that this NCAS operation has been cancelled by another NCAS operation. For concurrent usage of NCAS structures, we use Sundell's reference counting idiom [17].

4.2 NCAS Words and Value Function

Every variable that is supposed to be changed by an NCAS operation is wrapped into another structure, called an *NCAS word* (see Fig. 2). It either directly contains the value of one machine word or it contains a reference to another NCAS structure. In our implementation, NCAS words consist of two machine words; there is a status bit that differentiates between "contains value directly" and "contains reference". In either case the remaining bits of the two machine words are interpreted accordingly. To guarantee consistent updates of NCAS words, changes on them are always made by means of DCAS.

On an NCAS word an abstract value function is defined: If the NCAS word contains a value directly, the result of the function is this value. If the NCAS word contains a reference, the result of the value function depends on the values and the status of the referenced NCAS structure. If the status is not equal to SUCCESS, the value of the NCAS word is the value of the corresponding "old value" field in the referenced NCAS structure. If the status is SUCCESS, the value of the NCAS word is the "new value".

4.3 Read Method

We have implemented a read method reflecting req. **L2** in Sec. 3.3.1 that implements the value function, described in the previous section. If this method is performed successfully, it returns a value that has been the value of the NCAS word for some moment during the invocation of the method. The read method is weakly wait-free, i.e. it may fail to reliably read a value if the referred NCAS structure has been deallocated in the meantime. In this case the read method returns an error code.

4.4 NCAS Method

Our NCAS method implements req. **L1**, mentioned in Sec. 3.3.1. It consists of three phases, similar to Ramamurthy's approach [13]. The semantics of these phases are as follows:
Phase 1: A reference to the current NCAS structure `NCS` is inserted in every NCAS word that needs to be changed. Note that inserting these references does not affect the values of the NCAS words.
Phase 2: The NCAS method tries to change the status of the NCAS structure `NCS` from WORKING to SUCCESS by a single CAS instruction. The CAS might fail, if the status of `NCS` has been set either to WRONG_VALUES or KILLED before.
Phase 3: The references to the current NCAS structure `NCS` are replaced by the actual values. If the CAS instruction in phase 2 was successful, all references are replaced by the new values. Otherwise, the old values are restored. This works, since the replacing of the references in phase 1 does not affect the values of the NCAS words.

All manipulations of the NCAS words are made in a concurrent-safe way by means of DCAS. For instance, in the first phase it is ensured that a reference is only inserted if the NCAS word contains the expected value. Furthermore, a reference is only replaced by the actual value, if a NCAS word still contains a reference to the NCAS structure in the last phase.

The NCAS method could fail in two cases. Both of them are detected in the first phase and are taken into account no later than in the second phase. Moreover, both cases only arise in presence of speculative execution, otherwise all NCAS operations are executed sequentially with the help of the operation queue (see Sec. 3.3.2).

Speculative execution might lead to the situation that two threads concurrently work on different NCAS structures `NCS_A` and `NCS_B` containing references to shared NCAS words. Assume that a reference to `NCS_A` has already been inserted into an NCAS word and one of the threads tries to insert a reference to `NCS_B` into the same NCAS word. While it is straightforward to replace the reference to `NCS_A` with a reference to `NCS_B`, it must be ensured that only one NCAS operation succeeds if both are in state WORKING. Otherwise, the data structure would be updated inconsistently in the third phase; this is prevented by setting the status of `NCS_A` to KILLED by means of a CAS instruction before replacing the reference. This ensures that the values of the NCAS words are not changed by the NCAS operation referred to by `NCS_A`.

In the description above we assume that the NCAS operation referred to by the NCAS structure `NCS_B` *wins*, i.e. the NCAS operation on `NCS_B` can be successfully completed while that on `NCS_A` is cancelled. The actual winner, however, will be determined by means of priorities that are explained later on in Sec. 4.6.

Additionally, the DS could have been updated during the current NCAS operation. This can happen, if `NCS_A` has already proceeded to the state SUCCESS in the example above. Here, a thread in phase 1 would detect that an NCAS word does no longer have the expected value. In this case, inserting a reference to the NCAS structure into the NCAS word is not possible without changing values (after setting the reference, the NCAS word would contain the expected old value!). As a consequence, the whole NCAS operation cannot be executed anymore. The status of the NCAS structure is then changed to WRONG_VALUES by means of a CAS instruction. The value of the NCAS word is determined using the read method described in the foregoing section. If the read method fails, the status of the NCAS structure also is updated to WRONG_VALUES and the NCAS operation fails. Again, this ensures that the values of the NCAS words are not erroneously changed by the current NCAS operation.

4.5 Cooperative NCAS

A problem occurs when more than one thread simultaneously work on the same NCAS structure. If one thread is still in phase 1 despite the object's status already being SUCCESS, setting a reference may change the value of the NCAS word. Thus, we introduce pre-references that are similar to a reference, in that an NCAS word that contains a pre-reference has its value determined by the (pre-)referenced NCAS operation object. The difference is that the value is independent of the status of the object; it is always the old value. Thus, setting a pre-reference does not change the value of the NCAS word. By using a unique representation for the pre-reference, it is ensured that a thread does not mistake a pre-reference set by another thread for a pre-reference of its own; thus, ABA problems [10] are prevented. A pre-reference is distinguished from a reference by means of an additional bit next to the status bit in an NCAS word.

Phase 2 of the NCAS method requires that a reference is inserted into every NCAS word, otherwise the values of these NCAS words cannot be changed atomically. For this purpose, every pre-reference has to be replaced by a reference before phase 2 can be completed. The proper use of DCAS ensures that a thread cannot replace its pre-reference by the actual reference if another thread successfully completed the NCAS operation in the meantime. Hence, setting a reference is impossible, after the status of the NCAS operation has already been changed. Thereby, we can guarantee that the values of the NCAS words are not altered in phase 1, although several threads can cooperatively work on the same NCAS structure, hereby reflecting req. **W1** in Sec. 3.3.2.

4.6 Priorities of NCAS Requests

According to req. **D1** in Sec. 3.3.3, different priorities among NCAS operations have to be supported. Thereby, higher priority NCAS operations shall not be disturbed by lower priority ones. This is ensured by a priority comparison in phase 1 of the NCAS method, if a reference to another NCAS structure is encountered when inserting references and its state is WORKING. If this NCAS operation has a higher priority, the own NCAS operation is cancelled by setting its status to KILLED; else the other NCAS operation is cancelled instead.

5 Further Related Work

For brevity, we only present the most important related work. In [6], Herlihy introduced the first wait-free universal construction (UC) that consumes an unbounded amount of memory, serializes all operations on object O, which induces convoy effects [3], and makes copies of potentially large objects. It, furthermore, requires at least N local computation steps to complete a operation [9].

Anderson and Moir introduced in [1] various nonblocking UC to reduce the copying overhead when dealing with large objects on the basis of load-linked and store-conditional [1]. Their constructions do not allow parallelism on disjoint memory locations; this is due to the fact that there are no information about the memory locations available that are to be changed.

In [18], Sundell presented a wait-free NCAS operation with a helping scheme. The scope of his work, however, are transactions of sets of values instead of building DS operations. Building custom DS operations is also supported, but only lock-freedom can be guaranteed.

6 Conclusion and Further Work

Developers of shared data structures now have an easy way to achieve full interrupt-transparency with a strong progress guarantee and minimize sequential code. With RTNCAS, developers can, furthermore, use their sequential algorithms on top of it without modifications. We are currently still working on several optimizations to reduce the WCET of RTNCAS-based DS operations. Additionally, we also evaluate RTNCAS with other approaches, which currently show some promising results.

References

[1] ANDERSON, J. H., AND MOIR, M. Universal constructions for large objects. *in IEEE Transactions on Parallel and Distributed Systems 10*, 12 (December 1999), 1317–1332.

[2] CLARK, D. D. The structuring of systems using upcalls. *in Proc. of the 10th ACM Symp. on Operating Systems Principles* (1985), 171–180.

[3] FRASER, K. Practical lock-freedom. Tech. rep., TR 579, University of Cambridge, February 2004.

[4] GROVES, L. Verifying Michael and Scott's lock-free queue algorithm using trace reduction. *in Proc. of the 14th Computing: The Australasian Theory Symposium (CATS 2008), Wollongong, NSW, Australia. CRPIT, 77. Harland, J. and Manyem, P., Eds. ACS* (2008), 133–142.

[5] HARRIS, T. L., FRASER, K., AND PRATT, I. A. A practical multi-word compare-and-swap operation. *in Proc. of the 16th International Symp. on Distributed Computing* (2002), 265–279.

[6] HERLIHY, M. P. Wait-free synchronization. *in ACM Transactions on Programming Languages and Systems 11*, 1 (January 1991), 124–149.

[7] HERLIHY, M. P., AND WING, J. M. Linearizability: A correctness condition for concurrent objects. *in ACM Transactions on Programming Languages and Systems 12*, 3 (1990), 463–492.

[8] ISRAELI, A., AND RAPPOPORT, L. Disjoint-access-parallel implementations of strong shared memory primitives. *in Proc. of the Symp. on Principles of Distributed Computing* (1994), 151–160.

[9] JAYANTI, P. A lower bound on the local time complexity of universal constructions. *in Proc. of the 17th annual ACM Symp. on the Principles of Distributed Computing* (1998), 183–192.

[10] MICHAEL, M. M. ABA prevention using single-word instructions. Tech. rep., IBM Research Division, RC23089 (W0401-136), January 2004.

[11] MOIR, M. Practical implementations of non-blocking synchronization primitives. *in Proc. of the 16th annual ACM Symp. on the Principles of Distributed Computing, Santa Barbara* (August 1997), 219–228.

[12] PARNAS, D. L. Some hypotheses about the "uses" hierarchy for operating systems. Tech. Rep. BS I 76/1, Fachbereich Informatik, Technische Hochschule Darmstadt, March 1976.

[13] RAMAMURTHY, S. A lock-free approach to object sharing in real-time systems. *PhD thesis, University of North Carolina at Chapel Hill* (1997).

[14] SHAVIT, N., AND TOUITOU, D. Elimination trees and the construction of pools and stacks. *in Proc. of the Theory of Computing Systems* (1997), 645–670.

[15] STELLWAG, P., DITTER, A., AND SCHRÖDER-PREIKSCHAT, W. A wait-free queue for multiple enqueuers and multiple dequeuers using local preferences and pragmatic extensions. *in Proc. of the IEEE Symp. on Industrial Embedded Systems (SIES 2009)* (July 2009), 237–248.

[16] STELLWAG, P., SCHELER, F., KRAINZ, J., AND SCHRÖDER-PREIKSCHAT, W. A wait-free NCAS library for parallel applications with timing constraints. *poster abstract, in Proc. of the 16th ACM SIGPLAN Annual Symp. on Principles and Practice of Parallel Programming (PPoPP 2011)* (February 2011).

[17] SUNDELL, H. Wait-free reference counting and memory management. Tech. rep., 2004-10, Göteborg University, Chalmers, October 2004.

[18] SUNDELL, H. Wait-free multi-word compare-and-swap using greedy helping and grabbing. *in Proc. the 2009 International Conference on Parallel and Distributed Processing Techniques and Applications, Las Vegas, Nevada, USA, CSREA Press* (July 2009), 494–500.

[19] SUTTER, H. The free lunch is over. *in Dr. Dobb's Journal 30*, 3 (March 2005).

[20] ZHANG, Y. Non-blocking synchronization: Algorithms and performance evaluation. *PhD thesis, Göteborg University, Chalmers* (2003).