A Wait-Free NCAS Library for Parallel Applications with Timing Constraints

Philippe Stellwag, Fabian Scheler, Jakob Krainz, Wolfgang Schröder-Preikschat

Friedrich-Alexander University Erlangen-Nuremberg
Computer Science 4
Martensstr. 1, Erlangen, Germany
{stellwag,scheler,krainz,wosch}@cs.fau.de

Abstract

We introduce our major ideas of a wait-free, linearizable, and disjoint-access parallel NCAS library, called RTNCAS. It focuses the construction of wait-free data structure operations (DSO) in real-time circumstances. RTNCAS is able to conditionally swap multiple independent words (NCAS) in an atomic manner. It allows us, furthermore, to implement arbitrary DSO by means of their sequential specification.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming

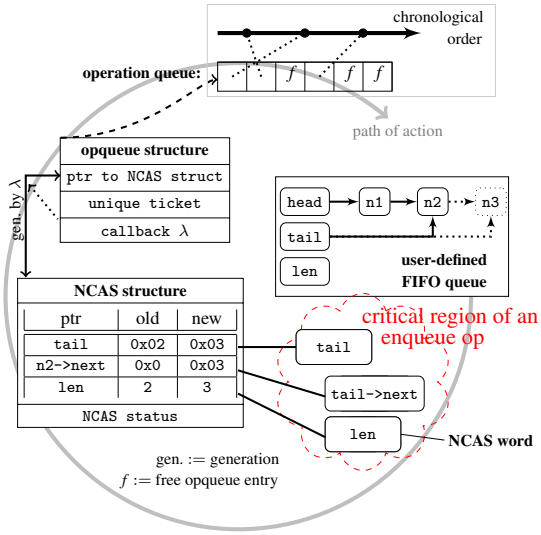
General Terms Algorithms, Design

1. Introduction

Sequential DSO usually rely on exclusive access to the parts of the data structure (DS). 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. This induces convoy effects and other timing drawbacks. Also prior universal constructions, such as Herlihy's work [1], serialize all DSO; this induces priority inversion in real-time circumstances.

Our Contribution

RTNCAS guarantees that DSO built on top of it are linearizable [2] and offer the following properties: **1. Lock freedom** offers interrupt transparency, and neither depends on system libraries nor induces restrictions to the scheduler (we do not consider backoff algorithms here). Under concurrent usage they also frequently yield better performance than blocking counterparts. As no local progress guarantees can be given for such operations, it is usually not possible to determine an upper bound for the worst-case execution time (WCET) of such DSO. We use lock-free DSO in RTNCAS to gain a stronger progress guarantee later on. **2. Wait freedom** [1] is similar to lock freedom, but additionally guarantees a definable and bounded WCET. Hereby, the temporal progress requirements of real-time applications can be satisfied. Note that correctness in real-time systems is both, functional correctness as well as timeliness. **3. Disjoint-access parallelism** [3] enables parallel executions of operations accessing disjoint memory locations. This yields additional benefits in terms of parallel performance and minimal priority inversions.



Starting with a sequential implementation of a DSO, we apply several transformations to it. Each transformation ensures the next stronger progress property (from 1. to 3.) and finally, a wait-free and disjoint-access-parallel DSO is reached.

Our work was inspired by Ramamurthy [4, pp. 138 ff.]. His NCAS, however, only focuses priority-based real-time systems on uni-processors. To the best of our knowledge, RTNCAS is the first generic approach to build interrupt-transparent DSO even in parallel real-time systems, and also considers disjoint-access parallelism. It is, furthermore, not restricted to priority-based real-time systems.

2. Design of RTNCAS

The construction of an NCAS library that satisfies the above stated properties is a rather complex task. Thereby many serious problems arise on the implementation level that we cannot discuss completely within these two pages. Thus, we only adumbrate the design of RTNCAS, and point to some interesting problems and results.

The figure shows the major idea and all data structures involved in RTNCAS. It 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 data structure are encapsulated into NCAS words. The thread performing the enqueue operation has to create an

PPoPP '11, February 12–16, 2011, San Antonio, Texas, USA.
ACM 978-1-4503-0119-0/11/02.

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 those stalled operations.

2.1 Lock-Free Operations

In a first step the formerly sequential DSO has to be transformed into a lock-free operation. This is accomplished by reading the old state of the DS via a *read method*, 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. We have implemented a NCAS method to support lock-free DSO and a read method to retrieve the actual value from the DS. Both are weakly wait-free, i.e. they respond within finite time, but may fail due to concurrent interferences and then have to be repeated.

2.2 Wait-Free Operations

The usage of the NCAS operation within such a retry-loop prevents the estimation of a WCET in the presence of contention. This is due to the unknown number of retries needed to successfully complete the DSO; under hard real-time conditions this is unacceptable. To overcome this, we transform the lock-free DSO into a wait-free one by means of a helping scheme implemented via an *operation queue*. This helping scheme facilitates the implementation of wait-free DSO. Thereby, wait freedom allows to determine the WCET that is essential in real-time circumstances.

The operation queue is implemented by a wait-free FIFO and works as follows: Every thread encapsulates its DSO into a so-called *opqueue structure*. This structure contains a pointer to a *callback* λ, a pointer to the *NCAS structure* (which contains the addresses of the words to be changed, and the corresponding old/new values) generated by the callback λ, and a unique ticket to establish a chronological order among all elements within the operation queue. λ is a user-defined callback that uses the sequential DSO to build the NCAS structure with old and new values.

After inserting its own DSO to the operation queue, each thread performs the following steps until its own DSO is completed: Get the oldest opqueue structure from the operation queue and try to perform the encapsulated operation. The latter is done as follows: **1.** The opqueue structure is checked for an active NCAS structure, i.e. a structure whose associated NCAS operation might still be executed. If one is found, the second step is skipped. **2.** λ is used to create a new NCAS structure. A thread then tries to atomically replace the reference to the NCAS structure in the opqueue structure with a reference to the newly created one by means of a CAS instruction. **3.** The NCAS structure is then executed in a cooperative manner. Thereby all threads always working on the same active NCAS structure (consensus object). Finally, in case of a successful execution, the opqueue structure is dequeued.

The operation queue, however, requires (1) a wait-free FIFO that allows us to find and use the oldest entry, and (2) the NCAS implementation has to support cooperation: If all active threads work on the same NCAS operation (using the same NCAS structure), the described NCAS operation must be completed successfully.

2.3 Disjoint-Access-Parallel Operations

The helping scheme implemented by the operation queue has one disadvantage: All DSO are performed sequentially as be known from previous approaches that may results in convoy effects, and may cause poor performance. To avoid this, we introduce the concept of *speculative execution*.

Before inserting an opqueue structure into the operation queue, each thread tries to execute the DSO speculatively by the corresponding lock-free operation. 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.

Using speculative execution, finally, we support the implementation of wait-free and disjoint-access-parallel DSO based on top of the wait-free operations provided by the helping scheme introduced in Sec. 2.2. Furthermore, the degree of disjoint-parallel accesses can be chosen with respect to the number of speculative executions that can take place concurrently.

3. Preliminary Results

We have implemented several DSO, such as push/pop on a stack or enqueue/dequeue on a queue, by means of spinlocks as well as RTNCAS. Numerous benchmarks that perform DSO (a) with increasing concurrency, and (b) with an increasing number of words to be touched simultaneously show very encouraging results: On the one hand and without speculative executions, RTNCAS operations show at average four times less jitter than operations with spinlocks. Moreover, RTNCAS-based operations show about ten times higher average times; however, the throughput of RTNCAS-based operations per time unit are slightly higher compared to the spinlock case. This is due to our helping scheme and is the crucial factor for our significant low execution jitter. On the other hand, these advantages go along with comparable maximal response times of operations built on top of RTNCAS.

Furthermore, with an increasing number of speculative executions, we are able to trade a higher degree of disjoint-access parallelism to a higher jitter. This allows us to achieve a higher average-case performance for appropriate (e.g., soft real-time) use cases.

4. Conclusion and Further Work

We have sketched the design of RTNCAS, a library offering linearizable, lock-free, wait-free and disjoint-access-parallel interfaces for reading and conditionally swapping multiple words in an atomic manner. Developers of shared data structures now have an easy way to achieve full interrupt-transparency with a strong progress guarantee. With RTNCAS, developers can, furthermore, use their sequential algorithms on top of it without modifications. Currently, we are still working on several optimizations to reduce the WCET of RTNCAS-based DSO. Moreover, we are also close to complete a formal proof of correctness for RTNCAS and its components.

References

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

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

[3] A. Israeli and L. Rappoport. Disjoint-access-parallel implementations of strong shared memory primitives. *in Proc. of the Symp. on Principles of Distributed Computing*, pages 151–160, 1994.

[4] S. Ramamurthy. A lock-free approach to object sharing in real-time systems. *PhD thesis, Univ. of North Carolina at Chapel Hill*, 1997.

A Wait-Free NCAS Library for Parallel Applications with Timing Constraints

Philippe Stellwag, Fabian Scheler, Jakob Krainz, and Wolfgang Schröder-Preikschat

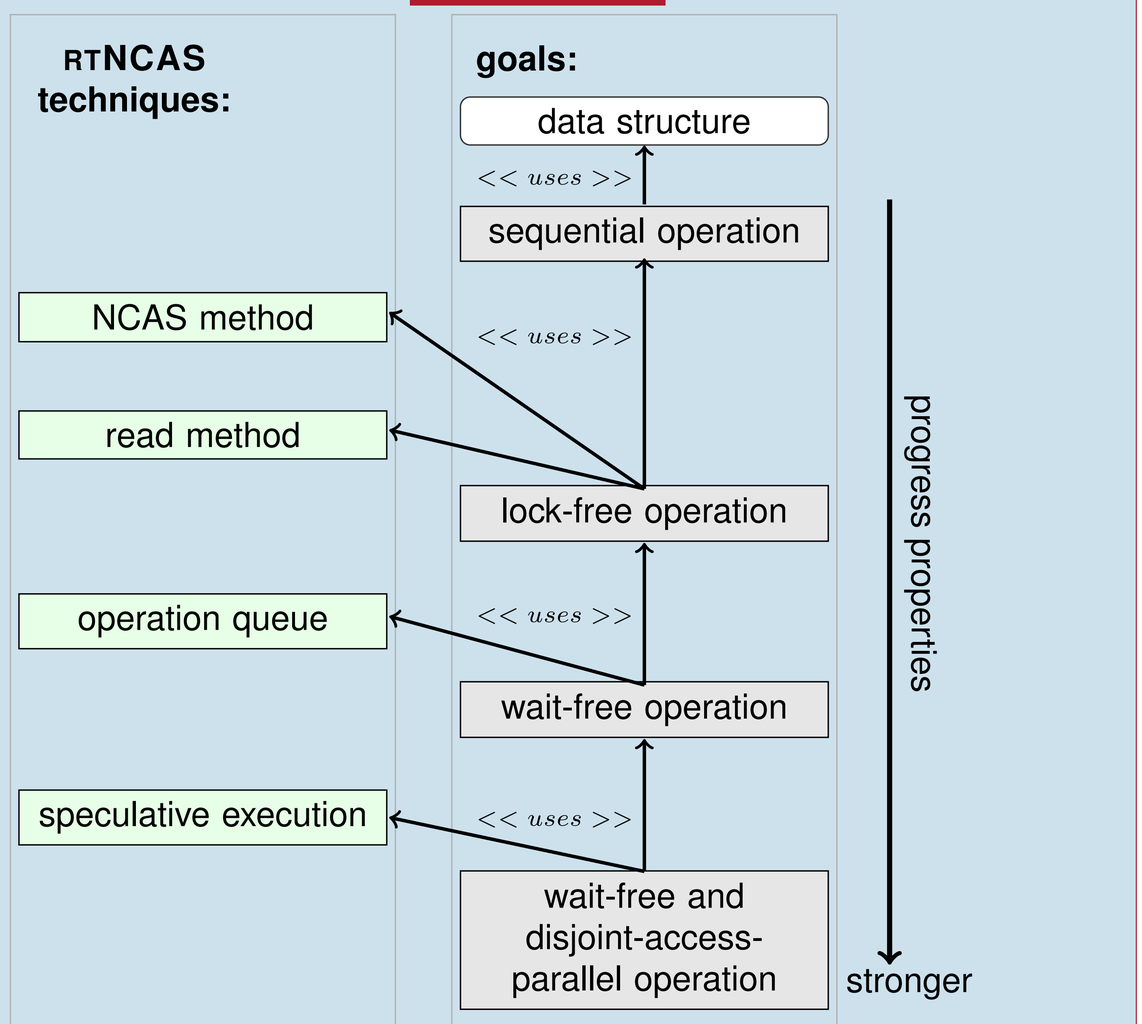
Introduction

Implicit Transaction per Mutual Exclusion	Explicit Transaction per Cooperation

`--Begin CS--`
`read WORD{1-3}`
`NEW{1-3} = calc(WORD{1-3})`
`write NEW{1-3},WORD{1-3}`
`--End CS--`
	`read WORD{1-3}`
	`NEW{1-3} = calc(WORD{1-3})`
	`mk_ncas(NEW{1-3},WORD{1-3})`

- **Problem.** Sequential code is the root of many problems in parallel real-time systems, e.g., priority inversion/violation, convoy effects, jitter, or scalability issues. Not only mutual exclusion, but also previous work on wait-free universal constructions, induces "sequential code". These interrupt-transparent mechanisms are based on auxiliary schemes and perform operations usually cooperatively, one after another.
- **rtNCAS.** Our approach allows building linearizable wait-free operations. Developers are, furthermore, able to re-use their sequential algorithms in multi-core environments without further care about concurrency. In addition, rtNCAS is able to speculatively execute operations that reduces sequential code.

Architecture



Implementation

NCAS method

- gets an NCAS struct as input generated through callback λ
- tries to swap the NCAS words:
  1. insert refs. to NCAS struct (consensus obj.) into words
  2. change values of all words simultaneously
  3. replace refs. of the NCAS words with their actual values

read method

- constant-time wait-free value function on NCAS words
- returns the value of the word that has been valid for some moment during the invocation of the read method
- may fail to reliably read the value, if the NCAS struct has been deallocated

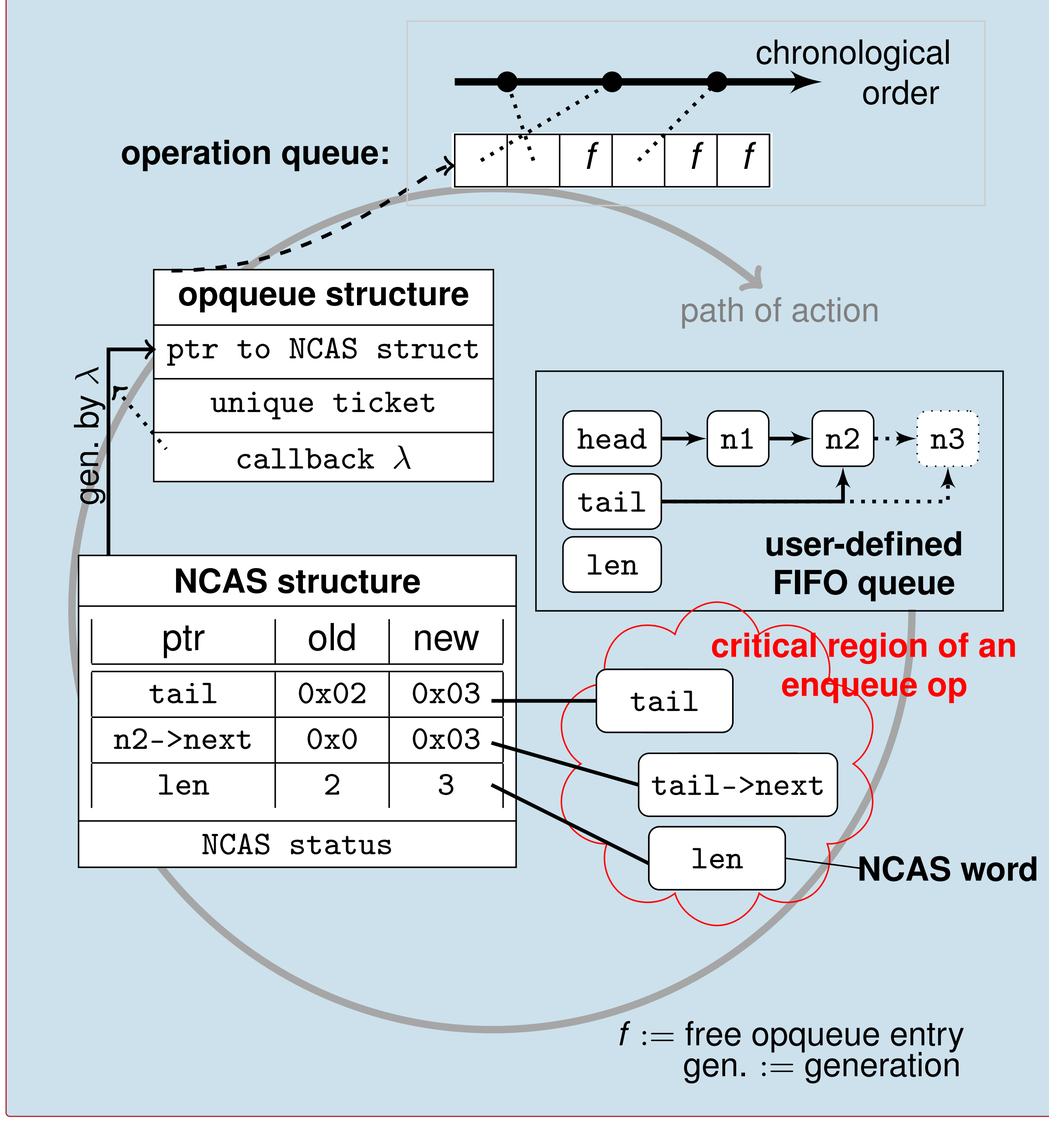
operation queue

- wait-free FIFO on the basis of an array of size N (number of threads)
- uses tickets to build a chronological relation
- uses fetch-and-add instructions to create unique tickets
- avoids ticket cycles, if number of threads $<= 4096$

speculative execution

- similar to the lock-free case
- might violate the progress of stalled operations; to avoid this, rtNCAS uses priorities of NCAS structs to be performed
- threads that succeed in speculative perform operations additionally have to help to perform stalled ones

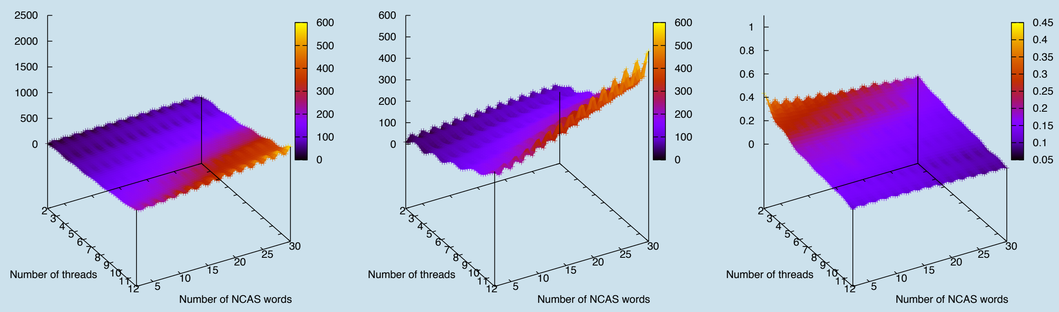
Involved Data Structures



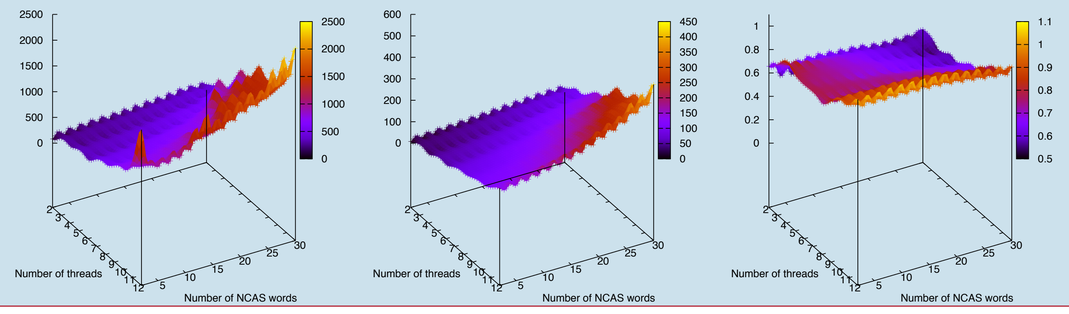
Micro-Benchmarks

The plots show the **maximal** and **average** execution times as well as the **coefficient of variation** as indication for jitter. The testbed uses four Intel Xeon E7340 quad-core CPUs, and Linux 2.6.29.4 with RTAI 3.7.1 as real-time extension.

(1) rtNCAS benchmark without speculative executions:



(2) rtNCAS benchmark with five speculative executions:



Friedrich-Alexander-Universität Erlangen-Nürnberg

Chair in Distributed Systems and Operating Systems