Stefan Reif and Wolfgang Schröder-Preikschat

Predictable Synchronisation Algorithms for Asynchronous Critical Sections

Technical Report CS-2018-03

February 2018

Please cite as:
Predictable Synchronisation Algorithms for Asynchronous Critical Sections

Stefan Reif, Wolfgang Schröder-Preikschat
Department of Computer Science
Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Germany

Abstract—Multi-core processors are ubiquitous. Even embedded systems nowadays use processors with multiple cores. Such use cases often impose latency requirements because they interact with physical objects.

One consequence is a need for synchronisation algorithms that provide predictable latency, in addition to high throughput. A promising approach are asynchronous critical sections that avoid waiting even when a resource is occupied.

This paper introduces two algorithms that allow for both synchronous and asynchronous critical sections. Both algorithms base on a novel wait-free queue. The evaluation shows that both algorithms outperform pre-existing synchronisation algorithms for asynchronous requests, and perform similarly to traditional lock-based algorithms for synchronous requests. In summary, our synchronisation algorithms can improve throughput and predictability of parallel applications.

I. INTRODUCTION

Shared-memory multi-core processors have become omnipresent [1], [2]. Even small embedded and portable devices employ processors with more than one core [3].

Systems running on multi-core processors typically need coordination for interacting threads. Most applications ensure data-structure integrity by mutual exclusion of critical sections. The corresponding synchronisation overhead is often performance critical in parallel applications and data structures [4], [5].

With embedded applications running on multi-core processors, the need for predictable synchronisation emerges. Seemingly small delays can accumulate and significantly harm the overall system performance and responsiveness [6], [7], [8]. Blocking operations of traditional lock algorithms are therefore problematic, especially for embedded systems that interact with physical entities [9]. Locks generally delay threads until an associated resource is available, which leads to situation-specific waiting times. In the worst case, a deadlock occurs and threads have to wait forever. Therefore, the goal is a synchronisation algorithm that is fast in in the worst case, without neglecting average-case performance.

A promising concept that avoids blocking, while maintaining the convenient program structure of critical sections, are asynchronous critical sections [10], [11], [12]. This concept means that threads can request the execution of arbitrary critical sections without blocking. For mutual exclusion, only the execution of the critical section is delayed, but the requesting thread can proceed. The critical section is hence decoupled from the requesting thread, and both control flows are potentially concurrent to each other. Traditional synchronous critical sections, in contrast, force threads to wait in case of contention.

For asynchronous critical sections, a run-time system has to provide a mechanism that ensures that each submitted critical section eventually runs. This system also enforces mutual exclusion of all requests.

The contributions of this paper are two general-purpose synchronisation algorithms that support asynchronous critical sections:

• A predictability-oriented synchronisation algorithm for shared-memory multi-core processors
• An adapted version of the above algorithm for many-core platforms where excess processor cores are available

Both algorithms are not limited to asynchronous critical sections. Instead, they support traditional synchronous critical sections as well.

The rest of the paper is structured as follows. Section II introduces existing concepts for delegation-based synchronisation, which is a prerequisite for asynchronous critical sections. Then, Section III presents both novel synchronisation algorithms for asynchronous critical sections. Section IV and Section V examine their correctness analytically and performance empirically. Afterwards, Section VI discusses related work and Section VII concludes the paper.
II. BACKGROUND

Contestion for shared resources generally requires coordination of concurrent threads. In case of conflict, threads have two options available. First, they can wait until the resource is available. This is the traditional lock-based synchronisation model. Second, the thread can encapsulate the critical operation in a dedicated job data structure and submit that job to a synchronisation entity that executes the critical section at the right moment in time [13]. This second concept has been generalised under the term delegation-based synchronisation [4]. Delegation-based synchronisation can be extended for asynchronous requests. By decoupling critical sections from the requesting thread, blocking is not mandatory. While the critical section is executed asynchronously and concurrently, the requesting thread can continue doing meaningful work.

A. Remote Core Locking

Remote Core Locking (RCL) [14] provides general-purpose delegation-based synchronisation. By concept, RCL migrates arbitrary critical sections to dedicated server threads. Other threads thereby take the role of client threads that encapsulate each critical section in a job, which they submit to the server. Such a job describes the critical section in a closure [15]. The server thread iteratively executes all incoming requests. Since only a single server thread exists, all critical sections are executed sequentially. RCL thus guarantees mutual exclusion of all requests.

While the control flow contrasts strongly with lock-based synchronisation, RCL is transparent in functional terms. To achieve compatibility, the RCL protocols force every client to wait for completion of each request. On the one hand, waiting for completion simplifies the implementation because, at any moment in time, each client can submit at most one request. This limit allows for bounded data structures. On the other hand, RCL does not allow for asynchronous critical sections.

This paper presents an alternative implementation for RCL that differs in functional aspects. In contrast to the original solution [14], this version allows for asynchronous critical sections.

B. Queue Delegation Locking

Queue Delegation Locking (QDL) [11] combines a lock with a bounded queue. This queue collects delegated critical sections. At lock release, the old owner executes all pending requests. However, the queue is bounded so the execution is only conditionally asynchronous—If the queue is full, threads are forced to wait. This paper presents an alternative algorithm that uses an unbounded queue. It thus never requires waiting for asynchronous requests.

C. Guards

Guards provide delegation-based synchronisation with focus on predictable execution time. As far as we know, the original implementation [12] was the first to achieve wait-free progress guarantee for the entry and exit protocols of critical sections.

Guards replace the dedicated server thread by an on-demand solution, the sequencer. Every thread that requests execution of a critical section can take the role of the sequencer, if the guard protocols demand for it.

The guard protocols are accompanied by a programming convention, which is shown in Listing 1. Each thread submits critical sections using the vouch function. This function is the entry protocol for the critical section and negotiates the sequencer thread. If the current thread is supposed to take that role, the function returns a job handle. Otherwise, it returns NULL to indicate that no sequencing is required.

The guard concept requires that the sequencer executes all pending requests. This convention is integrated into the clear function. This function returns a handle to the next request, if available. Otherwise, the clear function returns NULL to indicate that no more jobs are pending.

An important aspect of the guard concept is the progress guarantee. Even if the guard is occupied, threads neither block inside vouch nor clear. For non-sequencer threads, this means that they continue their original control flows immediately and run in parallel to their critical sections. For the sequencer, however, the progress guarantee is more complex. It is possible that the sequencer remains in the sequencing loop forever, when other threads submit too many jobs too quickly. For the application, it is therefore mandatory that critical sections are rare. Variants of the guard concept that remove this restriction by safely renegotiating the role of the sequencer are beyond the scope of this paper.

Listing 1: Guard sequencing loop

```
job_t *job = ...
job_t *cur;
if (NULL != (cur = vouch(guard, job))) do {
  run(cur);
} while (NULL != (cur = clear(guard)));
```
Guards rely on an additional reply mechanism. It allows guards to signal that a request has been executed and has terminated, ensuring that all control flow and data dependencies are fulfilled. A typical implementation of such a reply mechanism is a future [16] object that manages results of critical sections. In consequence, guards allow threads to wait immediately (fully synchronous critical section), never (fully asynchronous critical section), or at any later moment in time.

This paper presents an alternative implementation for guards that differs in non-functional aspects. Compared to other solutions, it significantly improves performance and predictability.

D. Actors

Actors [17] are a concept of structuring parallel programs [10]. Such a program consists of sequential entities that communicate via messages [18]. Typically, dedicated actor libraries [19] or languages support programmers to implement software entities as actors. Actors are therefore accompanied by a run-time system that facilitates message passing and actor scheduling.

For the scope of this paper, an actor combines a mailbox data structure and a server thread. Similar to RCL, each thread encapsulates critical sections in jobs and enqueue them in the mailbox. The server thread executes all incoming requests sequentially and thus guarantees mutual exclusion for all critical sections.

This interpretation is a mixture of the RCL and guard concepts. It combines a dedicated server thread with the support for asynchronous requests. Therefore, this paper focuses on efficient message passing, and presents an algorithm that has higher throughput than existing alternatives, especially at high contention. Furthermore, it has lower request latency for asynchronous critical sections.

III. Algorithms

Both synchronisation algorithms presented in this paper operate on a wait-free [20] multiple-producer single-consumer (MPSC) queue. As a distinctive feature, the enqueue operation detects whether the queue was empty beforehand. The synchronisation algorithms utilise this speciality internally.

A. The Guard Algorithm

The guard data structure, as shown in Listing 2, is basically an MPSC queue. Therefore, it has a head pointer referencing the oldest element, and a tail pointer that indicates where new elements can be added.

Listing 2: Data structure definitions

```c
typedef struct
{
    chain_t *next;
} chain_t;

typedef struct
{
    chain_t *head;
    chain_t *tail;
} guard_t;

typedef struct
{
    chain_t *head;
    chain_t *tail;
    sleep_t wait;
} actor_t;
```

Listing 3 summarises all guard-related functions, using atomic load/store, compare-and-swap (CAS), and fetch-and-set (FAS) operations. A setup function initialises the guard data structure, vouch enqueues a critical section to the guard, and clear function removes a request after completion. Figure 1 shows how vouch and clear are mapped to queue operations, and how the queue represents critical section states.

The entry protocol of the guard is the vouch function, which internally performs an enqueue operation. The FAS operation V1 orders request and, consequently, critical sections. Then, V2 detects whether the queue was empty beforehand. If so, the vouch function returns...
Fig. 1: Queue representation of critical section states

non-null, indicating that the current thread is supposed to take the role of the sequencer. As sequencer, the thread is allowed to execute its request immediately. Otherwise, the critical section is already occupied, because the current job is not the first item in the queue. In this case, a sequencer must already be present. Therefore, vouch returns NULL, indicating that the current thread is not supposed to operate as sequencer.

The exit protocol for the sequencer is implemented by the clear function. The sequencer calls this function to remove a request from the queue after completion. If another item is pending in the queue, clear returns a reference to that job. The sequencer is then obliged to execute it. Otherwise, clear returns NULL and the sequencer resumes to its original control flow.

It is possible that calls to vouch and clear overlap. If the queue already contains multiple elements, they do not interfere, because vouch modifies the tail only, and clear operates on the head. However, if the queue contains only a single element, then the two functions interact. Figure 2 details the interaction between concurrent calls to vouch and clear. Inside the vouch operation, it is possible for a short moment that the tail pointer points to a new request while the update of the next pointer is still pending. In this case, the sequencer leaves the queue, because the next element is not yet available. To manage this situation, the FAS operation C2 signals job completion to V2 using a unique magic value, DONE.

B. The Actor Algorithm

The queue algorithms that implement the guard protocols also constitute, with minor modifications, an actor mailbox implementation. Listing 4 summarises the actor protocols, which contain the same MPSC queue as the guard algorithms. Conceptually, the difference to guards is that a server thread is permanently available, instead of an on-demand sequencer thread. In consequence, the interface differs.

The core component of the actor is the server thread. This thread runs the serve function, which executes all

Listing 4: Actor protocols

```c
#include <threads/sys.h>
#include <threads/queue.h>

void actor_setup(actor_t *self) {
    self->head = self->tail = NULL;
    sleep_setup(&self->wait);
    thrd_create(serve, self);
}

void actor_submit(actor_t *self, chain_t *item) {
    item->next = NULL;
    chain_t *last = FAS(&self->tail, item);
    if (last) {
        if (CAS(&last->next, NULL, item))
            return;
        // last->next == DONE
        self->head = item;
        sleep_awake(&self->wait);
    }
    chain_t *actor_shift(actor_t *self, chain_t *item) {
        chain_t *next = FAS(&item->next, DONE);
        if (!next)
            CAS(&self->tail, item, NULL);
        CAS(&self->head, item, next);
        return next;
    }
    void actor_serve(actor_t *self) {
        chain_t *item = NULL;
        while (1) {
            if (!item) {
                item = sleep_wait(&self->wait, &self->head);
                // item != NULL
                run(item);
                item = actor_shift(self, item);
            }
        }
    }
```
incoming requests sequentially. Internally, this function repeatedly calls \texttt{shift}, which dequeues a request from the mailbox. Similarly to the guard algorithm, the first queue element describes the currently running job, and trailing elements represent pending requests. While no requests are pending, the server thread waits passively until a client submits a job, to improve energy efficiency. To this end, the guard queue algorithm needs minor adaptations to signal the presence of requests in the mailbox.

On the client side, the \texttt{submit} function enqueues requests to the actor mailbox. It is equivalent to the guard \texttt{vouch} function, except for an additional wake-up notification for the server thread when work is available. The enqueue operation helps to avoid unnecessary signals, since it detects whether the queue was empty beforehand. Only if the queue is empty, \texttt{submit} sends a wake-up signal. Otherwise, work is already pending, so that no signal is required.

Ideally, the worker thread of each actor is pinned to an exclusive core to avoid scheduler interference. This interpretation of actors assumes that the system contains plenty of cores (“many-core system”). Therefore, the application can dedicate one or more processor cores to the execution of critical sections.

\textbf{IV. Correctness Considerations}

For the sake of comprehensibility and brevity, this paper sketches a proof of correctness for the guard algorithm only informally. To this end, two properties need consideration. First, it is essential that, at every moment in time, at most one critical section is under execution (\textit{mutual exclusion}) at a given guard. Second, every critical section submitted to a guard must eventually be executed (\textit{liveliness}).

\textbf{Mutual exclusion} of critical sections is achieved by ensuring that, at every moment in time, at most one sequencer exists. Here, three scenarios need to be considered. First, mutual exclusion of sequencer threads initially holds because, at initialisation of the guard, no sequencer exists. Second, if the queue already contains at least one element, \texttt{vouch} returns \texttt{NULL}. In consequence, no further thread can take the role of the sequencer. Hence, the property of mutual exclusion remains when adding jobs to a non-empty queue. Third, if the sequencer leaves and another thread enters the sequencing loop, multiple complex overlapping patterns are possible. Figure 2 details the complete state space of interfering \texttt{vouch} and \texttt{clear} operations, considering every possible intermediate state. In the path through the bottom left node, the figure also covers the case were the sequencer leaves first, and afterwards, a new thread becomes sequencer. The top right node, in contrast, is the scenario where the \texttt{vouch} operation completes before \texttt{clear} begins. In summary, it is impossible that multiple sequencers co-exist at any moment in time.

\textbf{Liveliness} of synchronisation based on mutual exclusion is not possible to prove in general, because malicious threads can acquire a resource which they never release. In this artificial case, every further acquisition attempt to that resource is necessarily delayed forever. Therefore, this paper only considers \textit{cooperating} critical sections that certainly release the corresponding resource after a finite number of instructions. In contrast, \textit{non-cooperating} critical sections that hold the resource forever are considered a programming mistake. Under this precondition, liveliness of the guard algorithms can be shown by induction. Thereby, two situations need consideration. First, if the queue is empty, a job request
is granted immediately when vouch returns. Since vouch is wait-free, the job is certainly running after a bounded number of instructions. Second, if the queue is not empty, a previous request exists. By induction, the previous request is eventually executed. Afterwards, the next job is executed, with nothing but a clear operation in between. Since clear is wait-free, every critical section is certainly executed after a finite number of instructions.

We have also verified both mutual exclusion and liveliness properties of the guard algorithm using CDSChecker [21], [22]. This tool applies exhaustive state space exploration to multi-threaded test cases written in C or C++. For the actor algorithm, liveliness considerations are identical because it uses the same queue, and mutual exclusion is trivial because only a single server thread executes critical sections.

V. Evaluation

The evaluation compares both algorithms of Section III with pre-existing synchronisation algorithms. It targets throughput and latency of each variant. For the evaluation, we have implemented micro-benchmarks in C. The benchmarks are tailored for this evaluation because, for delegation-based synchronisation, critical sections must be representable as closures.

A. Evaluation Setup

This evaluation examines the performance of multiple synchronisation algorithms. First, the GUARD and ACTOR synchronisation methods implement the algorithms presented in Section III. Second, The OTHERGUARD variant implements a pre-existing guard algorithm [23] that uses a general-purpose wait-free queue [24]. Third, a fast wait-free MPSC queue [25] is used for an alternative actor (OTHERACTOR) implementation. A variant of this queue is used, for instance, in the akka actor framework [19], [26]. Fourth, locks are represented by TICKET and MCS locks [27]. Furthermore, PTHREAD mutexes serve as a performance baseline. They are the only non-fair synchronisation algorithm considered in the evaluation. Additionally, the ACTOR, GUARD, OTHERACTOR, and OTHERGUARD algorithms support asynchronous critical sections and are therefore evaluated for both synchronous and asynchronous requests.

All experiments were conducted on two computers. The large system has 80 logical cores. It contains four Intel Xeon E5-4640 processors, where each processor has 10 cores and hyper-threading. The processor runs at 2.2 GHz. The machine runs Ubuntu 16.04 and it uses the performance cpufreq-governor, which disables dynamic voltage and frequency scaling (DVFS) for consistent performance. The small system has an Intel Xeon E3-1275 v5 processor with 4 cores and hyper-threading. All eight logical cores run at 3.6 GHz. This machine runs Ubuntu 15.10 and also has DVFS disabled.

B. Throughput Evaluation

The first part of the evaluation focusses on the average case performance. The maximum throughput represents the overhead to synchronise critical sections.

To quantify the throughput, a micro-benchmark application spawns 1 to 79 threads, or 1 to 7 threads, for the large and the small system, respectively. Each thread is thereby pinned to a core. The last core is reserved for the actor server thread. For uniformity, we restrict all measurements to 79 or 7 cores even though guards and lock-based variants need no server thread. Every thread requests execution of critical sections in a tight loop. Thus, the only relevant work performed by the micro-benchmark is the synchronisation overhead to sequentialise the execution of critical sections.

The synchronisation throughput, averaged over $10^7$ requests, is shown in Figure 3. On the large system, the performance drops significantly at 10 cores due to the hardware architecture. In case of more than 10 cores, communication between NUMA nodes is required, which has a higher latency than local memory access operations. The consequence is a significant performance decline. For more than 30 threads, the performance is relatively constant amongst all synchronisation algorithms. On the small system, the performance is relatively constant for more than 4 threads.

For synchronous critical sections, the throughput of the GUARD and ACTOR algorithms is between a TICKET and an MCS lock. This is, however, a test where these algorithms cannot profit from the support for asynchronous execution. PTHREAD mutexes have the highest throughput because they put cores to sleep, which effectively reduces the degree of contention.

For asynchronous critical sections, the ACTOR variant outperforms all lock-based synchronisation algorithms, except for the passively-waiting PTHREAD mutex. The GUARD variant is slightly slower. The OTHERACTOR implementation also has a high throughput, but it is always behind the ACTOR algorithm of Section III. The OTHERGUARD algorithm, however, is bottlenecked by the relatively slow wait-free queue algorithm. Surprisingly, it cannot profit from asynchronous execution because it effectively increases the degree of contention—
as threads need not wait for the completion of jobs, they eagerly submit further requests. In consequence, the contention on the guard queue increases, and hence, performance decreases. The ACTOR and GUARD variant are unaffected because of the efficient queue algorithm.

C. Latency Evaluation

The second part of the evaluation examines the latency of synchronisation requests. Each request causes a specific overhead, compared to the potential non-synchronised execution of the same code. This part of the evaluation therefore measures the overhead associated with each individual synchronisation request.

Similarly to the throughput evaluation, the application consists of identical threads that eagerly request execution of $10^5$ critical sections. The rdtsc instruction measures the associated costs with processor cycle precision.

The average request costs are presented in Figure 4, in processor cycles per request. Similarly to the throughput evaluation, the synchronisation costs are comparable to lock-based synchronisation in the case of synchronous critical sections. On the large system, the latency grows significantly when using more than 10 cores because of the hardware architecture. The non-uniform memory access latency especially affects TICKET locks—they have a low latency at low contention, but they are relatively slow at high contention. For asynchronous requests, however, the GUARD and ACTOR algorithms outperform locks on both systems, because they do not need to wait while the critical section is unoccupied. Again, the OTHERGUARD algorithm is relatively slow because of the complex queue.

The worst-case request costs are presented in Figure 5. The 95% quantile represents the worst-case latency, but ignores hardware unpredictability (such as hardware interrupts) and Linux scheduler interference. The results are similar to the average-case evaluation. As an exception, PTHREAD mutexes fall behind because they are not fair. Notably, the GUARD algorithm scales nearly perfectly: At more than 20 cores, the latency is relatively constant. The asynchronous ACTOR variant has the lowest worst-case latency in most scenarios.

D. Analysis

In summary, the evaluation shows that the GUARD and ACTOR variants are competitive to existing synchronisation algorithms at synchronous requests, and they outperform locks at asynchronous requests.

For synchronous critical sections, the throughput of the ACTOR and GUARD variants is between TICKET and MCS locks. When the application supports asynchronous critical sections, however, delegation-based synchronisation methods outperform their lock-based competitors. Furthermore, the ACTOR algorithm outperforms an alternative, widely used queue algorithm. Locks are competitive at low contention, but at high contention, the ACTOR and GUARD algorithms are faster.
The average-case latency analysis shows similar results. For synchronous critical sections, the latency of the ACTOR and GUARD implementations is between ticket and MCS locks. In this scenario, lock algorithms wait at the beginning at critical sections, while ACTOR and GUARD implementations wait for completion of critical sections. In consequence, the latency differences are small for synchronous requests. For asynchronous critical sections, however, ACTOR and GUARD variants do not need to wait. In this scenario, both are faster than their competitors.

The worst-case latency evaluation highlights the importance of fairness. The non-fair PTHREAD mutex falls behind most competitors, even though the average-case performance is relatively good. Asynchronous variants are very fast, except for the OTHERGUARD algorithm. Asynchronous GUARD requests scale nearly perfectly.
E. Threats To Validity

The performance of synchronisation algorithms always depends on the actual hardware. The evaluation has used two different systems with varying processor speed, core count, and memory uniformity (NUMA and UMA). Other hardware platforms can possibly perform differently. In particular, hardware vendors have started to provide dedicated mechanisms for efficient synchronisation, such as transactional memory [28]. Future instruction sets might therefore exhibit different performance characteristics.

Real-world applications probably use a mixture of synchronous and asynchronous critical sections. Furthermore, they can submit requests before they need the result, so that they can use a combination of both. Then, threads can continue doing meaningful work while the critical section runs concurrently. In the ideal case, the result is already available when the application needs it. However, this effect completely depends on the application and the degree it can utilise asynchronous critical sections. Therefore, writing asynchronous programs remains a challenge for the application programmer. However, we are optimistic that, in many cases, applications can benefit from this form of micro-parallelism.

Actor languages and frameworks can impose an additional run-time overhead to map computations to actor operations. Similarly, run-time overhead related to program transformations required to represent critical sections in closures is outside the scope of the evaluation. However, actor frameworks like akka are successful in industry and academia. Especially on many-core systems, the cost of thread coordination likely dominates the overall system performance.

VI. RELATED WORK

Asynchronous critical sections and similar synchronisation techniques have been used in operating systems [29], [30]. Similarly, the guard concept [12], [23] was originally introduced as a “structuring aid” for multi-core real-time systems.

Many synchronisation concepts support request delegation, but no asynchronous requests. For instance, flat combining [31] delegates data structure operations to an on-demand combiner thread, but enforces request synchronicity. Similarly, RCL [14] and FFWD [32] force threads to wait until requests have completed. The reason is that, without asynchronous requests, every client thread has at most one pending request. This limitation allows for bounded internal data structures, which are often relatively fast and simple. In contrast, the synchronisation algorithms in this paper base on an unbounded queue to fully support asynchronous requests, but they nevertheless achieve competitive performance.

Previous work on general-purpose synchronisation algorithms has often focussed on scalability [27], [33], [34] and the average-case performance [4], [5], [35]. Many lock algorithms have shown good performance in the average case, but they inherently suffer from blocking delays. Similarly, concurrent queue algorithms are typically optimised for the average-case [36], [37], even wait-free implementations [24], [38]. In contrast, both queue-based algorithms in this paper were designed with the worst-case latency in mind.

Recent research on scalable synchronisation algorithms considers timing predictability at hardware level. For instance, locality-aware locking improves the worst-case memory access time by avoiding the overhead of non-local cache operations in NUMA systems. For instance, hierarchical NUMA locks [33], [39], [40] prefer lock transition between cores of the same NUMA node. They thus avoid the additional delay of remote lock transitions, if possible. This optimisation results in higher throughput, but it actually comes at the cost of increased worst-case waiting time. Similarly, thread scheduling can improve system performance by exploiting data locality [41]. The actor concept goes even further and restricts data accesses to a single thread, and thus avoids remote data access operations.

Predictability for synchronisation algorithms often refers to fairness and linear blocking time, with respect to the number of waiting threads [42]. FIFO locks, such as the Ticket Lock [27] ensure that every thread can eventually enter the critical section. They thus avoid starvation, which is an extreme case of unpredictability. Similarly to the algorithms in this paper, some FIFO locks use a queue internally, such as MCS and LH [43]. The difference to the algorithms in this paper is the synchronicity of requests—locks force threads to wait if the critical section is occupied. In consequence, the waiting time can depend on the number of contenders. In contrast, asynchronous requests always complete in a finite number of operations, regardless of the degree of contention.

In general, predictability is important for real-time systems. Hard real-time systems do not depend on average-case performance. Instead, they are designed to meet deadlines, using a sound worst-case execution time (WCET) analysis [9], [44], [45]. Deriving a safe worst-case blocking bound, however, is known to be difficult [46], [47]. Asynchronous critical sections can help
to eliminate the need for a blocking bound estimation because requests cannot block. Then, the WCET of synchronisation algorithms is independent of the durations of critical sections. For synchronous requests, blocking is performed on an optional $future$ variable. In that case, a real-time system designer can apply existing blocking bound estimation techniques to derive the waiting time for the future object.

Besides real-time systems, predictability has also become an issue for high-performance computing (HPC). At large scale, seemingly minor delays can decrease system performance significantly [6], [7], [8]. In consequence, HPC systems often employ application-specific operating systems that minimise system noise [48]. The synchronisation algorithms presented in this paper can help avoiding synchronisation-related jitter.

VII. CONCLUSION

The trend towards embedded multi-core processors is accompanied by a need for fast and predictable thread coordination. This paper has presented two general-purpose synchronisation algorithms that allow for asynchronous critical sections. The benefit of asynchronous execution of critical sections is that threads are not forced to wait if a resource is occupied. Instead, they submit a request that will be executed later. Job management is implemented as an efficient wait-free MPSC queue.

Both algorithms are not limited to asynchronous critical sections. They also support traditional synchronous critical sections by waiting for completion of requests. This extension mimics the behaviour of locking protocols, if required.

The guard algorithm assumes that the number of processor cores is relatively small and therefore negotiates a sequencer thread on demand. This role demands the execution of all pending requests. The actor algorithm, in contrast, assumes that plenty of processor cores are available and therefore permanently occupies a dedicated core for request processing. The evaluation, however, shows that both algorithms are fast at any degree of contention, on large and small systems.

For both algorithms presented in this paper, the number of instructions per critical section has an upper bound. For the guard, the worst-case costs are seven atomic memory operations per asynchronous critical section. The actor has an additional system-specific overhead when the server thread waits passively. Both variants thus provide wait-free progress guarantees to all interacting threads, assuming that all critical sections terminate eventually.

The evaluation shows that both synchronisation algorithms are competitive for synchronous critical sections, and they outperform lock-based variants at asynchronous critical sections. Both throughput and latency are better than lock-based alternatives. On a 80 core machine, the worst-case request latency of guard requests is nearly constant at more than 20 cores. Actors and guard-based variants also scale better than locks in the worst-case latency evaluation. In summary, both algorithms therefore offer high performance and timing predictability. The actor algorithm furthermore outperforms an alternative implementation based on a pre-existing MPSC queue.

In summary, both algorithms show that asynchronous critical sections improve the performance and predictability of parallel programs significantly. Future work will examine how distributed storage systems, video streaming and processing, and other latency-critical compute-intensive applications benefit from this form of micro-parallelism.

ACKNOWLEDGEMENTS

This work is supported by the German Research Foundation (DFG) under grants no. SCHR 603/8-2, SCHR 603/13-1, SCHR 603/15-1, and the Transregional Collaborative Research Centre “Invasive Computing” (SFB/TR89, Project C1). We further thank Timo Höning for his insightful comments.

APPENDIX A

PASSIVE WAITING

For the actor algorithm, a server thread exists permanently. In consequence, it requires an efficient mechanism to wait for synchronisation requests while the actor is idle.

Listing 5 sketches an implementation for the server thread to sleep while it is idle. In this example, the Linux $futex$ system call [49] is used to wait passively until a request is present. A flag variable indicates whether the server thread is currently waiting for further requests, and therefore sensitive to a wakeup signal.

The awake function first checks the sensitivity flag before it sends a wake-up signal. On the other side, await first sets the flag before checking the actual sleeping condition. This combination avoids lost-wakeup problems because, between checking the condition and calling $futex\_wait$, the flag is set.

Passive waiting, in general, is not specific to the Linux $futex$ system call. Alternative implementations could use UNIX signals or hardware interrupts to wait for synchronisation requests.
Listing 5: Sleep operations

typedef struct {
  int state;
} sleep_t;

void sleep_setup(sleep_t *self) {
  self->state = 0;
}

void *sleep_await(sleep_t *self, void **expr) {
  while (1) {
    self->state = 1;
    void *data = *expr;
    if (data) {
      self->state = 0;
      return data;
    }
    futex_wait(&self->state, 1);
  }
}

void sleep_awake(sleep_t *self) {
  if (CAS(&self->state, 1, 0))
    futex_wake(&self->state);
}

Listing 6: Link element with deallocation function

typedef struct {
  chain_t *next;
  work_t work;
  free_t free;
} chain_t;

Listing 7: Guard protocols with request deallocation

chain_t *guard_vouch(guard_t *self) {
  item->next = NULL;
  chain_t *last = FAS(&self->tail, item); // V1
  if (last) {
    if (CAS(&last->next, NULL, item)) // V2
      return NULL;
    // last->next == DONE
    last->free(last);
  }
  self->head = item; // V3
  return item;
}

chain_t *guard_clear(guard_t *self) {
  chain_t *item = self->head; // C1
  // item != NULL
  chain_t *next = FAS(&item->next, DONE); // C2
  bool mine = true;
  if (!next)
    mine = CAS(&self->tail, item, NULL); // C3
  CAS(&self->head, item, next); // C4
  if (mine)
    item->free(item);
  return next;
}

APPENDIX B
MEMORY MANAGEMENT

Memory management is an important issue for parallel algorithms. Since multiple control flows access shared data structures simultaneously, deallocation needs coordination. Otherwise, control flows operate on possibly invalid data.

Both algorithms presented in this paper use a chain_t data structure that represents jobs. Memory management for these queue elements has to consider that multiple control flows access them.

Memory management further has to support both synchronous and asynchronous critical sections. The chain_t data structure in Listing 6 therefore embeds a function pointer (free) which describes the deallocation procedure, depending on the request type.

For asynchronous critical sections, the free function can deallocate the request data structure. Since the critical section is asynchronous, it is safe to assume that no other thread accesses the link data structure afterwards.

For synchronous critical sections, however, a thread waits termination of the critical section. Then, the sequencer must not deallocate the request, since another thread can still access it. In that case, the free function pointer notifies a potentially waiting thread that the critical section has completed. It is then up to the waiting thread to actually deallocate the link element when it is no longer needed. The notification implies that deallocation is safe since no sequencer or server thread accesses the link element any more.

Deallocation is straightforward for the actor algorithm, since a dedicated server thread exists. All queue node deallocations can be performed by this thread. Since only a single server thread exists, it intrinsically knows when a link element is no longer needed. The server thread can thus ensure that every link element is deallocated exactly once.

For the guard algorithm, however, deallocation is more complex, since threads take the role of the sequencer only temporarily. Therefore, the code in Listing 7 identifies the situations where request deallocation is safe. Importantly, the link element is not accessed afterwards, even in the case of a sequencer change.

In most cases, the sequencer is allowed to deallocate a request directly after executing it. However, there is one notable exception when concurrent vouch and clear interfere in such a way that the role of the sequencer...
transitions. Since the `vouch` function accesses the next pointer in V2, a concurrent `clear` must not deallocate the corresponding request. However, `clear` can detect the concurrent `vouch`. If the sequencer manages to reset the tail pointer in C3, no concurrent `vouch` operation is happening. Since the tail pointer is reset to NULL, the current job is no longer accessible through the guard data structure, especially for future `vouch` operations. Deallocation is therefore safe. However, if C3 fails, then a concurrent `vouch` operation is certainly happening, and that `vouch` operation has finished V1 but not V2. Later, the CAS operation V2 will encounter the DONE value which signals the sequencer change. In this case, the new sequencer deallocates a request that its predecessor executed.

In summary, both algorithms reliably detect when link element deallocation is safe, without additional communication. A general-purpose memory management scheme, such as hazard pointers [50], is not needed.

REFERENCES