POSTER: A Predictable Synchronisation Algorithm

Stefan Reif
Friedrich-Alexander University Erlangen-Nürnberg
reif@cs.fau.de

Wolfgang Schröder-Preikschat
Friedrich-Alexander University Erlangen-Nürnberg
wosch@cs.fau.de

Abstract

Interaction with physical objects often imposes *latency* requirements to multi-core embedded systems. One consequence is the need for synchronisation algorithms that provide predictable latency, in addition to high throughput. We present a synchronisation algorithm that needs at most 7 atomic memory operations per *asynchronous critical section*. The performance is competitive, at least, to locks.

CCS Concepts • **Theory of computation → Parallel computing models**; *Shared memory algorithms*; • **Software and its engineering → Process synchronization**; *Concurrency control*;

Keywords Concurrent Data Structures, Parallel Algorithms, Synchronization, Predictability

ACM Reference Format:
Stefan Reif and Wolfgang Schröder-Preikschat. 2018. POSTER: A Predictable Synchronisation Algorithm. In *PPoPP '18: 23nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, February 24–28, 2018, Vienna, Austria.* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3178487.3178533

1 Introduction

With embedded applications running on multi-core processors, the need for predictable synchronisation emerges. Locks, however, delay threads until an associated resource is available, causing 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 the average case, and also in the worst case.

To avoid blocking, synchronisation requests have to be executed *asynchronously*—the execution of the critical section is possibly delayed ensuring mutual exclusion, but the requesting thread can proceed. The critical section is thus decoupled from its inquirer. For asynchronous requests, a run-time system has to ensure that each submitted critical section eventually runs. This system also enforces mutual exclusion of all submitted critical sections.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

PPoPP '18, February 24–28, 2018, Vienna, Austria
© 2018 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-4982-6/18/02.
https://doi.org/10.1145/3178487.3178533

Listing 1. Guard sequencing loop

```
job_t *job = /* critical section */;
job_t *cur = vouch(guard, job);
if (NULL != cur) do {
  run(cur);
} while (NULL != (cur = clear(guard)));
```

The contribution of this paper is a synchronisation algorithm that supports asynchronous requests. It achieves better performance and predictability than alternative algorithms.

2 Background and Related Work

For *remote core locking (RCL)* [2], threads *delegate* their critical sections to a *server thread* for execution. RCL is transparent to locks in functional terms and therefore forces threads to wait for completion of each request. However, delegation-based synchronisation can be extended for asynchronous requests. By decoupling critical sections from the requesting thread, blocking becomes unnecessary. Asynchronous requests can thus eliminate unpredictable blocking delays.

Guards [1, 3] support asynchronous requests based on an unbounded job queue. They further use an on-demand solution, the *sequencer*, instead of a dedicated server thread. If the guard protocols decide so, every thread that submits a critical section can become the sequencer. It is then responsible to execute all pending requests. For mutual exclusion, at most one sequencer exists at any moment in time.

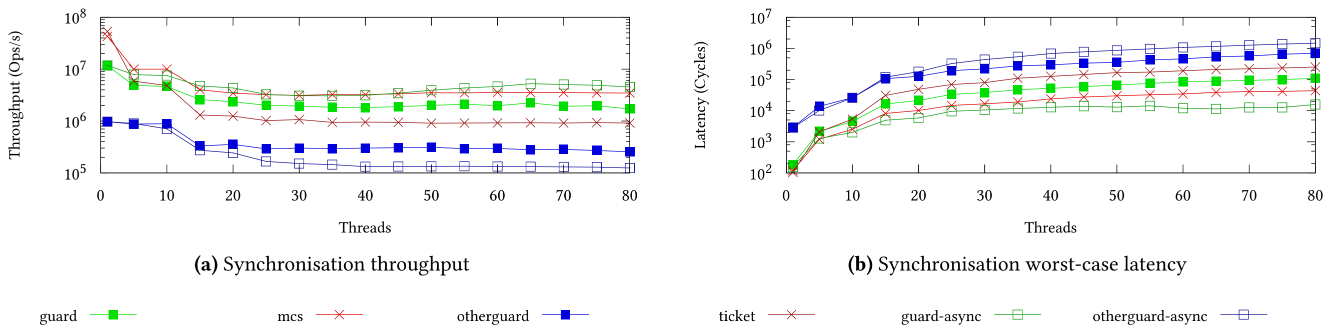
The guard protocols are accompanied by a programming convention, which is shown in Listing 1. Threads can submit critical sections using `vouch`, which internally negotiates the sequencer thread. When a thread becomes the sequencer, it executes all pending requests. After completion of a critical section, the sequencer calls `clear`. This function returns the next pending request, if available.

Guards rely on an additional *reply* mechanism that provides results of critical sections in a *future* variable. This is optional, to respect data dependencies. Guards thus allow threads to wait immediately (*synchronous* critical section), never (*asynchronous* critical section), or at any later moment.

We present an alternative guard implementation that significantly improves performance and predictability compared to previous algorithms.

3 Algorithm

Our predictable synchronisation algorithm operates on a wait-free multiple-producer single-consumer queue. As a



(a) Synchronisation throughput

(b) Synchronisation worst-case latency

Figure 1. Throughput and worst-case latency for synchronous and asynchronous critical sections

Listing 2. Guard data structures and protocols

```
typedef struct { chain_t *next; /* ... */ }      chain_t;
typedef struct { chain_t *head; chain_t *tail; } guard_t;

void setup(guard_t *self) {
  self->head = self->tail = NULL;
}

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

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

distinctive but necessary feature, the enqueue operation detects whether the queue was empty beforehand. Listing 2 summarises all guard functions. A `setup` function initialises the guard data structure, `vouch` submits a critical section, and `clear` removes a request after completion. In total, each request requires at most 7 atomic memory operations.

The `vouch` function enqueues a critical section. Key moment is the V1 operation, which orders concurrent calls to `vouch`, and thus, critical sections. Then, V2 detects whether the queue was hitherto empty. If so, the `vouch` function decides that the current thread becomes the sequencer and returns non-NULL. Otherwise, a sequencer must already be present because another job is enqueued.

The sequencer calls `clear` after completing a request, to remove it from the queue. If another item is enqueued, `clear` returns a reference to it. The sequencer is then obliged to execute that request. If a `vouch` operation happens concurrently, the sequencer role can transition. Internally, `clear` signals this situation to V2 using a unique magic value, DONE.

4 Performance Evaluation

The evaluation compares the GUARD algorithm to a pre-existing variant (OTHERGUARD) [3], and TICKET and MCS locks. Both guard algorithms are also evaluated for asynchronous requests. All experiments were conducted on a machine with 80 logical cores (4x Intel Xeon E5-4640). We refer to the accompanying technical report [4] for more details.

A micro-benchmark spawns 1 to 80 threads that all submit empty critical sections in a tight loop[1]. The *throughput*, averaged over 10^7 requests, is shown in Figure 1. The NUMA hardware causes performance drops at 10 cores. The GUARD is between TICKET and MCS locks, and GUARD-ASYNC outperforms them at high contention. Due to internal overhead, the OTHERGUARD cannot benefit from asynchronous requests.

Figure 1 also shows the per-request *latency* of 10^5 critical sections. The 95 % percentile represents the worst-case latency, excluding hardware unpredictability and OS noise. Again, the synchronous GUARD is between TICKET and MCS locks. The GUARD-ASYNC variant scales nearly perfectly and thus achieves the best predictability.

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).

References

[1] G. Drescher and W. Schröder-Preikschat. 2015. Guarded Sections: Structuring Aid for Wait-Free Synchronisation. In *IEEE ISORC '15*. 280–283.

[2] J.-P. Lozi, F. David, G. Thomas, J. Lawall, and G. Muller. 2012. Remote Core Locking: Migrating Critical-Section Execution to Improve the Performance of Multithreaded Applications. In *USENIX ATC '12*. 65–76.

[3] S. Reif, T. Hönig, and W. Schröder-Preikschat. 2017. In the Heat of Conflict: On the Synchronisation of Critical Sections. In *IEEE ISORC '17*. 42–51.

[4] S. Reif and W. Schröder-Preikschat. 2017. *Predictable Synchronisation Algorithms for Asynchronous Critical Sections.* Technical Report.

[1]Empty critical sections are a stress test for the guard, since the avoidance of blocking is especially beneficial when long requests are pending.