Hardlock
Real-time multicore locking

Strøm, Tórur Biskopstø; Sparsø, Jens; Schoeberl, Martin

Published in:
Journal of Systems Architecture

Link to article, DOI:
10.1016/j.sysarc.2019.02.003

Publication date:
2019

Document Version
Peer reviewed version

Hardlock: Real-Time Multicore Locking

Tórur Biskopstø Strøm, Jens Sparsø and Martin Schoeberl

Department of Applied Mathematics and Computer Science
Technical University of Denmark
Email: tbst@dtu.dk, jspa@dtu.dk, masca@dtu.dk

***Abstract*—Multiple threads executing on a multicore processor often communicate via shared objects, allocated in main memory, and protected by locks. A lock itself is often implemented with the compare-and-swap operation. However, this operation is retried when the operation fails and the number of retries is unbounded. For hard real-time systems we need to be able to provide worst-case execution time bounds for all operations.**

The paper presents a time-predictable solution for locking on a multicore processor. Hardlock is an on-chip locking unit that supports concurrent locking without the need to get off-chip. Acquisition of a lock takes 2 clock cycles and release of a lock 1 clock cycle.

I. Introduction

Multithreaded programs often communicate over shared data allocated on the heap. To coordinate the access to these shared data structures we use locks. A lock protects a data structure such that only a single thread that is owning the lock can read or write to that data structure. Other threads trying to acquire the lock block until the lock is released by the current lock owner.

To implement a lock, we need some operations that execute atomically, usually a load, a compare, and a conditional store. On a single-core processor, the processor can enforce atomicity by simply turning of interrupts. However, on a multicore processors further hardware support is typically available in the form of compare-and-set (CAS) or load linked and store conditional operations.

CAS operates on a single memory word and has as input two values: an expected old value and a new value. The processor reads the memory location, compares it with the expected old value, and when the read value is identical with the expected value, updates the memory word with the new value. The processor executes this combination of operations atomically. The instruction returns the read value to indicate success or failure. On a failure the CAS operation is simply repeated.

CAS can be used to implement locks, which works efficiently in the common case. However, locks built upon CAS suffer from the potential issue that one thread may starve. When two or more tasks contend for the same memory location, i.e., try to update the value of the same memory location, there is a small possibility that one or more of the tasks will suffer from starvation. This happens if a task tries to swap the value with its own, but with every attempt another task has already swapped it, resulting in the initial task never being able to swap the value and therefore never acquiring the lock. This possible unbounded starvation is not acceptable for real-time systems. We need a system where the worst-case execution time (WCET) of each operation needs to be bounded.

This paper presents and evaluates Hardlock, a hardware unit that provides truly concurrent locks for real-time multicore processors. The acquisition of a free lock is bounded and takes in the worst-case 2 clock cycles. The release of a lock takes 1 clock cycle. Two different locks can be requested concurrently by two different cores. Furthermore, the Hardlock implements a queue in hardware so that every core that requests a lock will be served in the worst-case after all other cores that requested the lock. No unbounded starvation is possible.

To support manycore processors, and for comparison, we also implemented an asynchronous locking unit where cores in different clock domains can access the locking unit. Furthermore, for the evaluation we also implemented a CAS based locking unit for a multicore processor. We integrated all locking units with the T-CREST multicore processor [23], [26]. The comparison of the three locking implementations shows that the Hardlock provides the shortest time for locking and unlocking, and is also the only unit that properly bounds this time. Furthermore, for the number of locks used in this paper, it is also the smallest hardware unit.

This paper is an extension of [33], where we presented an initial version of Hardlock. The additional contributions of this paper are: (1) an implementation of CAS on a shared scratchpad memory for comparison, (2) an implementation of an asynchronous lock for comparison, (3) integration of the three locking units with the pthread library.

This paper is organized in 7 sections. Section II presents related work. Section III provides background on the T-CREST multicore platform, which is used for the implementation and evaluation of the locking units. Section IV describes our designs and implementations. Section V analyses the performance of our implementations whereas Section VI measures the performance our implementations on an FPGA. Section VII concludes the paper.

II. Related Work

The classical issue of priority inversion that can occur on real-time systems when tasks use locks was solved a long time ago for single-core systems with the priority ceiling protocol [29]. Additionally, the protocol also prevents deadlocks. However, no such definite solution has been found for multicore system. Many solutions have been proposed, all with their own benefits and drawbacks. Our work focuses on the hardware acceleration of acquiring and releasing locks to generically reduce the locking overhead, and not any specific protocol, although these protocols can be built on top of our locking unit. We find it easiest to analyze programs that use our hardware locking units if critical sections execute non-preemptively and a global ordering of lock acquisitions is used to prevent deadlocks. Additionally, critical sections should be kept short to reduce the impact of threads' critical sections on each other, as well as gaining the proportionally greatest benefit of the low lock acquisition overhead of the locking units.

Carter et al. [6] compare 6 lock implementations, 2 of which are software implementations with CAS or similar atomic operation support, and 4 hardware implementations. Their findings show that hardware implementations can reduce the lock acquisition and release time by 25-94% compared to well-tuned software locks. Using their own benchmark with heavy contention, the hardware locks outperform the software locks by up to 75%, whilst on a SPLASH-2 benchmark suite, the hardware locks perform 3-6% better.

Patel et al. [21] describe a hardware implementation of MCAS, a multi-word CAS operation for multi-core systems. They find that on average, their implementation is up to 13.8 times faster than locks with critical sections spanning from 40 to 345 cycles. The locks are POSIX mutexes built on Tejas, a Java based architectural simulator, where the lock overhead is not specified. The MCAS operation itself is guaranteed to be atomic and starvation free. However, like CAS, a thread can potentially starve unless additional operations are added to prevent this.

Afshar et al. [5] propose a synchronization unit connected to all cores, like the Hardlock. It also contains a field for each core to register synchronization participation. However, they designed the unit for low-power systems in a producer/consumer relationship, thus only the power consumption, and not the performance, is tested. Additionally, the unit has a shared counter field, meaning that some arbitration, which is not described, must be done to update the counter.

Milik and Hrynkiewicz [19] present a complete distributed control system, that also includes hardware memory semaphores. The semaphore allows consumers to be notified as soon as data is ready, or the producer to be notified when it can update data. The semaphores are not centralized. Instead, each consumer has its own semaphore that notifies, or is notified, when data is consumed, or available, respectively. There can only be one producer per semaphore. It is not clear if the same semaphore allows multiple producers, and if so, how the semaphore handles arbitration.

Braojos et al. [3] investigate pre-emptive global resource sharing protocols. They also present their own protocol that features an increased schedulability ratio of task sets and strong task progress guarantees. The Hardlock operates at the core level and therefore does not make any guarantees about the behaviour of threads on the same core pre-empting each other. This is not an issue in this paper, as the T-CREST platform that we integrate the Hardlock with is not configured for more than one thread per core. However, the platform can easily support pre-emptive global resource sharing protocols by utilizing the Hardlock as a global lock and then managing queues and priorities in software.

Altera provides a "mutex core" [4], which implements atomic test-and-set functionality on a register with fields for an owner and a value. However, that unit does not provide support for enqueuing tasks. Therefore, guaranteeing an ordering of tasks entering the critical section must be done in software.

US Patent 8,321,872 [12] describes a hardware unit that provides multiple mutex registers with additional "waiters" flags. The hardware unit can trigger interrupts when locking or unlocking, such that an operating system can adapt appropriate scheduling. The operating system carries out the actual handling of the wait queue.

The hardware unit described in US Patent 7,062,583 [15] uses semaphores instead of mutexes, i.e., more than one task can gain access to a shared resource. The hardware unit supports both spin-locking and suspension; in the latter case, the hardware unit triggers an interrupt when the semaphore becomes available. Again, queue handling must be done in software. US Patent Application 11/116,972 [38] builds on that patent, but notably extends it with the possibility to allocate semaphores dynamically.

US Patent Application 10/764,967 [20] proposes hardware queues for resource management. These queues are, however, not used for ordering accesses to a shared resource; instead a queue implements a pool of resources, from which processors can acquire a resource when needed.

The hardware-accelerated queue, HAQu [18], is used to improve the performance of software queues by adding a single instruction for enqueueing and dequeueing, whilst keeping the queue data in the application's address space. Hardware locks might not provide the same performance benefits to queues as dedicated queueing units. However, locks have more use-cases than just queues, giving hardware locking units a wider application potential.

Besides using a CAS operation on a shared, external main memory, an on-chip shared scratchpad memory can support synchronization [8]. The shared scratchpad memory is arbitrated in a time-division multiplexing manner for normal read or write operations providing a time-predictable memory for shared data structures. The arbitration scheme is extended, allowing larger access slots where two memory operations, a read and a write, can be performed by a single core. With those two operations executed atomically, locks can be implemented. However, this extension of time slots also leads to higher worst-case access time for normal read and write operations. A variation, with one dedicated synchronization slot, leads to a smaller increase of the worst-case memory access time at the cost of longer lock acquisition times. In contrast, our approach avoids mixing normal access to shared memory and a locking protocol by providing a dedicated locking unit. We envision also using on-chip shared memory for shared data structures protected by a lock from our locking unit.

In our previous work [32] we implement hardware locks for a Java processor that support queues of waiting tasks. The locks support 3 types of atomic operations: requesting a lock, checking ownership, and releasing a lock. Using varying number of processors, we compare the hardware implementation to a software implementation. In all cases the hardware routines are significantly faster than the software routines. This also applies for the benchmarks with a high lock use. The difference between the Java locking unit and the proposed locking unit in this paper, is that the Java locking unit tracks locked memory locations using a content-addressable-memory and has a FIFO queue for each lock. This requires arbitration of requests. The unit in this paper does not rely on FIFO queues and can therefore be without the request arbitration, i.e., cores can concurrently issue requests that the unit processes concurrently, although in the case of contention only one core receives the lock.

An alternative to locks are lock- and wait-free data structures. Instead of acquiring a lock and then updating the data structure, threads manipulate the structure with the help of CAS, or similar atomic primitive, without preventing other threads from concurrently accessing the structure. Lock-free data structures are not starvation free on their own. However, wait-free data structures provide the same guarantees whilst being starvation free. Kogan and Petrank [14] present a wait-free queue with performance comparable to that of a lock-free queue. This approach is also generalized [37] to allow any lock-free data structure to become wait-free. Compared to our locks, the presented algorithms require several additional steps to ensure the correct behavior when manipulating the data structures. Additionally, it is not clear whether the presented algorithms are WCET analyzable, as they contain while-loops and they do not present a WCET analysis. Conversely, the Hardlock is fully analyzable and highly deterministic. However, we acknowledge that a proper comparison requires the wait-free data structures to be implemented on the same platform as ours, and the same benchmarks executed for both.

III. The T-CREST Platform

The locking unit can be used in any multicore processor. For the evaluation we used the open-source T-CREST platform [23], [26]. Figure 1 shows the T-CREST platform, consisting of several processor cores called Patmos [27], [28]. All processor cores are connected to a shared memory via memory network-on-chip (NoC) [25]. For message passing communication between cores, all cores are connected to the Argo NoC [13]. Finally, all cores are connected to the presented locking unit. The implementation of the locking unit is part of the Patmos source repository.

Patmos is a RISC style processor optimized for WCET analysis. Patmos contains special cache types to simplify WCET analysis: a method cache [7] that caches full functions and a stack cache [1] for stack allocated data. Patmos is supported by a compiler that optimizes for WCET [22], based on the LLVM framework [17]. The compiler provides flow facts for the aiT [9] WCET analysis tool from AbsInt and also

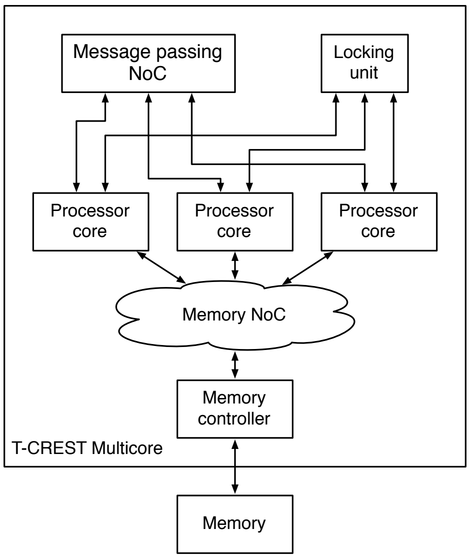


Fig. 1: The T-CREST multicore architecture with several processor cores connected three multicore devices: (1) one NoC for core-to-core message passing, (2) one for access to the shared, external memory, and (3) the Hardlock device.

incorporates WCET path information for aiT. Furthermore, an open-source WCET analysis tool, platin [10], is available for Patmos.

The Argo NoC [13] provides message passing between processing cores via virtual point-to-point channels. Data is pushed from one core's local scratchpad memory to a destination scratchpad memory. To be time predictable, the Argo NoC uses static time-division multiplexing for access control to router and link resources. The network interface between the processor and the Argo NoC is synchronized with the time-division multiplexing schedule. This results in a NoC structure without flow control and no additional buffering.

The Argo NoC has also been used to implement synchronization primitives [30]. Furthermore, the original T-CREST platform supports locks in shared memory with a software-based implementation. It works as follows: to provide a coherent view of the main memory the write-through data cache is invalidated [24] and then Lamport's bakery algorithm [16] is used to implement locks. This implementation of locks is inefficient, so the proposed locking unit replaces it.

IV. Implementation Overview

We have implemented 3 locking units: (1) the synchronous Hardlock, (2) CAS on a shared scratchpad memory, and (3) an asynchronous lock. Furthermore, we have adapted the library to use those units for the pthread mutex.

A. Hardlock

The main goal of the Hardlock is to minimize the WCET when acquiring and releasing locks, thereby potentially reducing critical sections. To achieve this, it is important that requests to one lock are unrelated to requests to a different

TABLE I: The io interface of the Hardlock

Name	Size	Direction	Description
en	1	Input	Request activation
op	1	Input	Request type, i.e., acquisition or release
sel	$\log_2(m)$	Input	Lock ID
blck	1	Output	Core blocked status

lock, i.e., cores can request locks concurrently without going through some arbiter. The secondary goal is to be starvation free, which is an important property for real-time systems. Although this can be achieved without direct support from the hardware, having locks that are inherently starvation free alleviates the burden of users.

Figure 2 shows an overview of the Hardlock. n cores connect to the Hardlock through their respective input and output signals, and can issue requests to any of the m locks. There are two types of requests:

- Acquisitions, where a core requests ownership of a lock
- Releases, where a core releases ownership of a lock

Table I lists the Hardlock interface. In our integration with Patmos we adapt the signals to the T-CREST open core protocol [2] interface.

The *blck* signal indicates whether a core has an outstanding request, i.e., whether the core is awaiting ownership. If a core's *blck* signal is set, the core stalls its pipeline. Therefore, although a core can own multiple locks, it can only have one outstanding request at any time, potentially increasing the WCET of unrelated threads on the stalled core. Stalling is not a requirement dictated by the Hardlock, as cores might also poll or register an interrupt when a lock is available, but stalling simplifies analysis and the implementation. Additionally, we are not running any scheduler on the T-CREST platform, so each core only executes one thread. However, at the cost of a few more clock cycles the locking unit can be extended to support multiple threads for each core. As we focus on minimizing the WCET we will not expound this.

Figure 2 shows how the input signals connect to each lock. The *sel* signal determines which lock a core's request is routed to. Each lock produces one *blck* per core. These are or'ed together with respect to the core number, producing a single *blck* signal per core.

Figure 3 depicts the composition of a single lock. Each lock consists of a queue register and a current owner register. The queue consists of n bits, with each bit belonging to one core. A core's *en* and *op* signals connect to the bit. A set bit indicates that the respective core has issued an acquisition request for the lock. Set bits in the queue therefore indicate that the respective core either owns the lock or is waiting for it. Cleared bits indicate a release request, or simply the absence of a request. The current owner register consists of $\log_2(n)$ bits that specify the identity of the core currently owning the lock. The current owner is set by round-robin arbitration of the queue register. Starting from the last owner, or 0 if there is no previous owner, the arbiter searches through the queue register, looping around at the end, until it encounters a set bit, as shown in Figure 4. When the owner clears its queue bit, the arbiter starts the process again from that position in the queue. The process of finding the next set bit in the queue is done within a single clock cycle.

If there is no contention for a lock, the Hardlock processes acquisition requests in 3 steps:

1) Receive the acquisition request and update the respective queue bit
2) Update the current owner register
3) Notify the core and stop stalling it

Those three steps are executed in 2 clock cycles. Note that the 3rd step does not count towards the acquisition time, as the core executes the next instruction in that clock cycle.

The benefits of iterating round-robin style are twofold:

1) The queue is simpler than, e.g., FIFO, as the order of the requests are irrelevant, so there is no additional hardware for this. Cores only access their own queue bit. This allows cores to truly concurrently issue requests. Additionally, release requests only need to clear the request bit.
2) In contrast with, e.g., compare-and-swap, if no core indefinitely holds a lock, the unit is guaranteed to process all requests in bounded time, preventing starvation.

B. Compare-And-Swap

CAS is one of the most commonly supported synchronization primitives. Comparing the Hardlock to CAS will therefore show how a specialized solution compares to the common solution. However, as Patmos does not support CAS, we implement it as described in this section.

CAS is commonly implemented as a specialized instruction that is supplied with 3 values: The address of the value to be swapped, the expected old value at that address, and the new value to be written at that address if the value at the address and the supplied old value is equal. The address used for CAS can be any memory address in the processor's memory space.

We extend a scratchpad memory (SPM) with the necessary logic to perform CAS operations. Instead of supporting CAS on the full memory space, as might be the general case for commercial processors, we restrict the operation to the memory space of the shared SPM. This significantly simplifies implementation, as:

1) there is no need for adding a specific CAS operation to the instruction set, as explained below
2) we do not have to consider cache coherency issues

The CAS unit contains two registers specific to each core: one containing the expected value at the address to be swapped, and one containing the new value. Each is set by a core issuing a store to them in the CAS unit. These are the only stores that the CAS unit supports. To issue a CAS operation, a core first sets the two registers, after which it issues a load from the address that should be swapped. The CAS unit reads the value from the SPM and compares it to the expected value register. If the values are equal, the value from the new value register is written to the SPM. Regardless

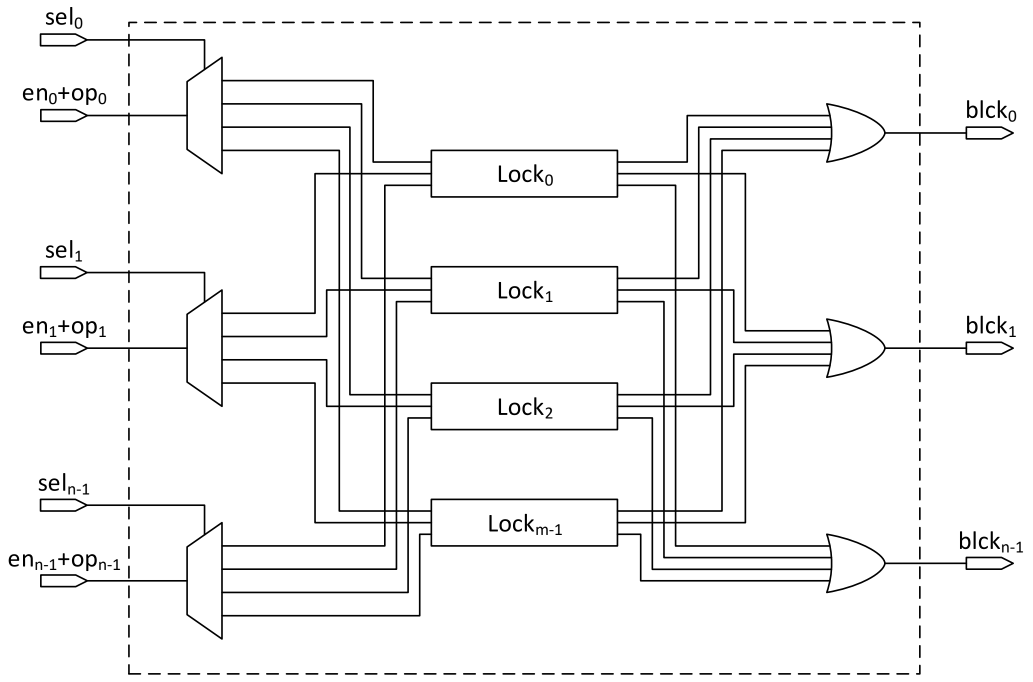


Fig. 2: An overview of the Hardlock showing *n* cores connected to *m* locks

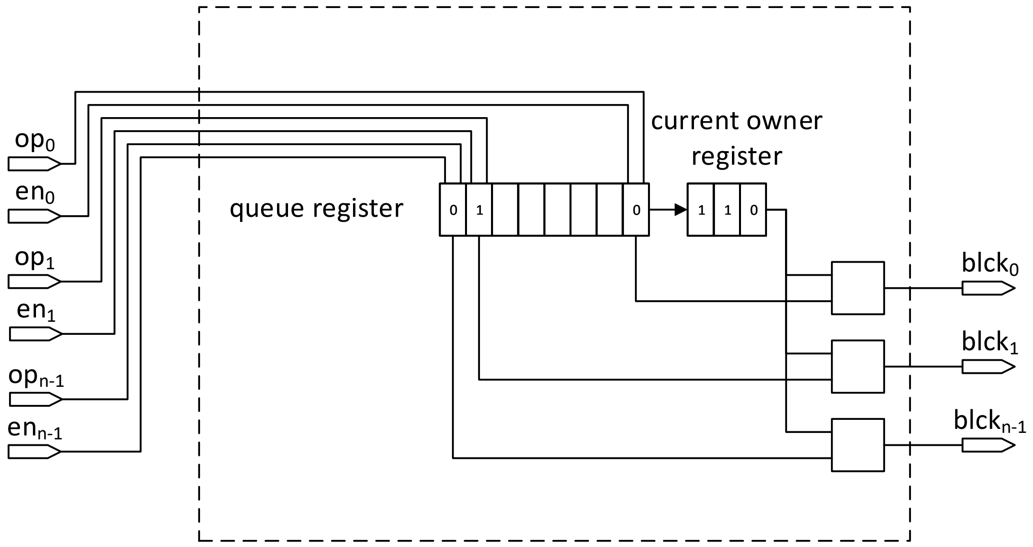


Fig. 3: Composition of a lock from Figure 2 showing *n* cores connected to their respective queue register bits in the lock. A round-robin arbiter updates the current register based on the queue register, and blocking signals are generated from the two registers

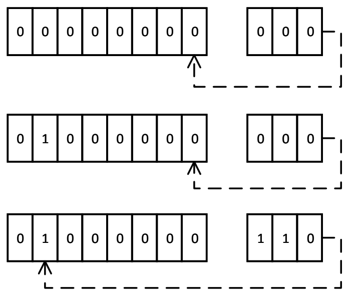


Fig. 4: Current register updated to the next lock owner in a single cycle by a round-robin arbiter

of whether the expected value and the read value are equal, the read value is returned to the core. The core can then check the success of the CAS operation by similarly comparing the expected value with the returned value.

A benefit of storing the expected and new values in core specific registers, is that in the case that the CAS operation fails, the core does not have to re-supply the two values. Instead it can continuously issue loads until the operation succeeds.

Access to the SPM is arbitrated with time-division multiplexing, ensuring time-bounded access. The CAS unit needs one clock cycle to issue a load to the SPM, one clock cycle to receive the value and compare it and one clock cycle to potentially issue a store. The time-division multiplexing slot size is therefore set to 3 clock cycles.

As the expected and new value registers are core specific, and not a part of the SPM, they can be written at any time by a core and are not subjected to the arbitration. Multiple threads running on the same core must not issue multiple interleaved CAS operations, as this would corrupt the registers. Given that the CAS operation is time bounded we can simply disable interrupts for the core instead of having to support register sets for each thread.

There are many solutions to implementing locks on top of CAS, depending on whether the locks support queueing, preemption, etc. However, these all add additional overhead in the form of software steps. To keep the locks simple and fast, each CAS unit address corresponds to a lock. Lock acquisition is then as follows:

```
expected_value = 0;
new_value = 1;
while(cas(lockid) != 0) { }
```

and lock release as follows:

```
expected_value = 1;
new_value = 0;
// Should succeed on first try
while(cas(lockid) != 1) { }
```

C. Asynchronous Lock

The previous solutions require both the processors and the synchronization mechanisms to be in the same clock domain. For very large chips with hundreds and even thousands of processor cores this becomes infeasible. These systems typically use a globally-asynchronous, locally-synchronous timing organization where each processor resides in its own clock domain, and where some form of asynchronous communication protocol is used for communication between clock domains. In such a system, a locking unit could reside in its own clock domain as well. For reliable operation, the locking unit must synchronize incoming request signals (using for example a pair of flip-flops), and the individual processors must synchronize the incoming grant-signals.

A simpler and more elegant solution can be built using a tree of asynchronous arbiters. Figure 5(a) shows the interface between a processor and the arbiter tree and Figure 5(b) shows an arbiter tree for a system with eight processors. The arbiter tree implements one lock and must be copied if more locks are needed. The request-grant interface used on all arbiter ports (inputs as well as outputs) use a 4-phase handshaking protocol and the meaning of the four events of a full handshake are: (1) Request the resource, (2) the resource has been granted, (3) release the resource, and (4) the resource has been released.

The arbiter is a standard textbook design [31][Section 5.8]. The mutual exclusion element that forms the heart of the arbiter requires a non-digital metastability filter circuit. For our FPGA implementation we use a standard gate implementation proposed by Ran Ginosar[1]. Compared to a synchronous locking unit, where synchronizers would have to be used for both signals going to the unit, as well as signals coming from the unit, only the grant-signals into the processors need to be synchronized for the arbiter tree. Furthermore, the arbiter tree itself is very fast, as it is self-timed and signals will traverse the tree as fast as possible, regardless of the clock(s) in the rest of the system. The arbiter tree does not guarantee that processors will get the lock in the same order as they request it, but it does guarantee the same worst-case behavior as the locks discussed in the previous sections: In the worst-case a processor requesting a lock will be granted access to the lock after all the other $n-1$ processors have been granted access once.

In the general case n is not a power of two and in this case one or more leaf nodes are omitted from the arbiter tree. This results in an unbalanced tree, and processors with a shorter distance to the root may obtain the lock two times where other processors obtain it only once. In any case a processor in a system with n processors will wait for at most $n'-1$ other processors to access the lock, where n' is computed as follows (the number of processors rounded up to the nearest power of two):

$$n' = 2^{\lceil \log_2(n) \rceil} \tag{1}$$

D. Pthread Mutex

In the next section we show how our analyzed values correspond to our measured values. To this end we use optimized locking procedures, such as pre-calculating the lock id, as any

[1]http://images.slideplayer.com/16/4906671/slides/slide_13.jpg

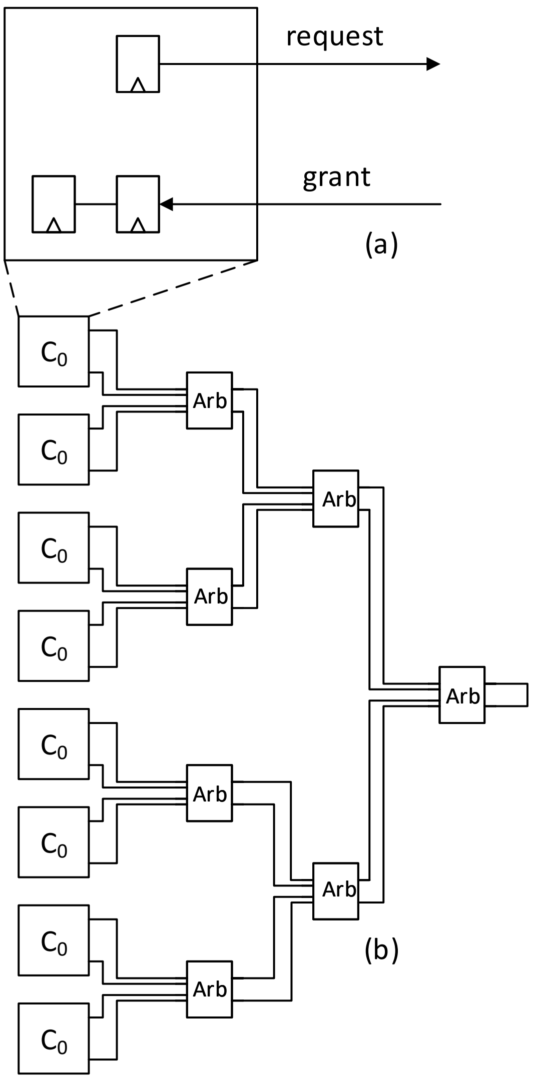


Fig. 5: (a) The request-grant interface between a processor node and the arbiter tree. (b) The asynchronous arbiter tree implementing a single lock for a system with eight processors.

additional operation, other than requesting the lock, adds to the overhead. However, these procedures are not practical for programmers. We therefore build the standard pthread mutex on top of our locks, allowing our locks to be used in "normal" C programs.

We do this by implementing the types *pthread_mutex_attribute* and *pthread_mutex_t*, as well as the following methods

1) *pthread_mutex_initialize*: Initializes a mutex by mapping it to an available hardware lock
2) *pthread_mutex_destroy*: Deallocates the mutex by marking the mapped hardware lock as available
3) *pthread_mutex_lock*: Attempts to acquire the mutex. This method maps almost directly to the hardware lock

TABLE II: Uncontended lock performance in clock cycles

	Acquisition	Release
Hardlock	2	1
CAS	$3n+8$	$3n+8$
Asynchronous Lock	≈ 3	≈ 3

TABLE III: Contended lock acquisition performance in clock cycles

Hardlock	$2n+\sum_{i=1}^{n-1} c_i$
CAS	$3n+8+\sum_{1}^{\infty}\sum_{i=1}^{n-1}(6n+1+c_i)$
Asynchronous Lock	$\approx 6n'-3+\sum_{i=1}^{n'-1} c_i$

4) *pthread_mutex_unlock*: Releases the mutex. As above, this method maps almost directly to the hardware lock

We track available hardware locks by having an array of the same length as the number of hardware locks. One of the hardware locks is reserved for synchronizing access to this structure. During mutex initialization, an available hardware lock is found, marked as reserved, and its id written in the mutex. When the mutex is destroyed the hardware lock is marked available.

Acquiring and releasing the mutex does not incur the above overhead, as these methods map to a hardware lock. The implementation limits the number of mutexes that can be active to the number of hardware locks. However, the number of concurrently active locks is usually low, so this is not necessarily an issue. Additionally, users do not have to consider which specific hardware locks to use, which makes the pthread mutex easier to use than the hardware locks in their "raw" form.

V. Timing Analysis

In this section we analyze the behavior and limitations of each lock. Table II and Table III present the derived WCETs. n is the number of cores, c_i is the WCET of another core's critical section and n' is the variable from equation 1.

None of the locking units dictate how long cores hold the locks, as this is application dependent. Also, the units do not prevent deadlocks. The software must handle these issues, e.g., always acquire locks in the same order, no infinite loops while holding locks, etc.

Another apparent issue with the units is the limited number of locks. Whilst this is configurable during hardware generation, it does not have the benefit of normal CAS where every memory address is a potential lock, i.e., practically infinite locks. However, we find that most applications use a very limited amount of locks.

A. Hardlock

When a core requests an uncontended lock, 1 clock cycle is spent updating the queue register. The round-robin arbiter spends another clock cycle updating the current owner register. In the 3rd clock cycle the core is notified of the ownership, and as it continues execution in the same clock cycle, the

3rd clock cycle does not count towards the lock request, i.e., 2 clock cycles total. A lock release is done by updating the queue register, after which the core can continue execution, i.e., 1 clock cycle total.

When a lock is contended, we assume that all other cores acquire the lock first and must release it. However, this requires the other cores to be enqueued, meaning the first acquisition clock cycle for all other cores does not count towards the total waiting time. We also include other cores' critical sections. We therefore have

$$\begin{aligned}\mathrm{WCET}_{\mathrm{acquisition}} &= 2+\sum_{i=1}^{n-1}(1+1+c_i)\\ &= 2+2(n-1)+\sum_{i=1}^{n-1}c_i \\ &= 2n+\sum_{i=1}^{n-1}c_i\end{aligned} \tag{2}$$

B. CAS

The CAS unit uses 3 clock cycle TDM slots and one slot for each core, i.e., n slots. However, for the WCET the core misses its own slot's first clock cycle, so it must wait 2 clock cycles plus 3 clock cycles for each TDM slot. Additionally, software steps take 6 clock cycles: 2 clock cycles for writing the expected and new value to the unit, and 4 cycles for branching the while loop. If the lock is uncontended, we know that the CAS operation will succeed. The WCET for acquiring an uncontended lock is therefore $3n+2+6=3n+8$ clock cycles. This also applies for a release.

When the lock is contended, we assume that other cores come first, similarly to the Hardlock. However, although a CAS operation is guaranteed to be executed in bounded time, i.e., $3n+8$ clock cycles, the number of times other cores can acquire the lock is not. We can therefore state what the overhead of another core acquiring the lock first will be, but not the number of these acquisitions. The WCET is therefore technically unbounded, represented by the infinite sum.

If another core acquires the lock first, we assume it has the last TDM slot, so $3(n-1)+2$ clock cycles. That core then executes its critical section, c_i, after which it releases the lock, but not before missing its TDM slot, i.e., another $3n+2$ clock cycles. We also add $3n+8$ clock cycles for the current core acquiring the lock, but just missing its TDM slot. The WCET is therefore

$$\begin{aligned}\mathrm{WCET}_{\mathrm{acquisition}} &= 3n+8+\sum_{1}^{\infty}\sum_{i=1}^{n-1}(3(n-1)+2+3n+2+c_i)\\ &= 3n+8+\sum_{1}^{\infty}\sum_{i=1}^{n-1}(6n+1+c_i)\end{aligned} \tag{3}$$

C. Asynchronous Lock

The asynchronous lock involves both synchronous and asynchronous parts. On the synchronous side the lock has one flip-flop per core to maintain the command that the core issued, and two flip-flops to synchronize the answer (grant) from the locking unit. On the asynchronous side the WCET depends on the number of connected cores, i.e., the depth of the arbiter tree. With n processors, the depth of the arbiter tree can be expressed as follows: $d=\lceil \log_2(n)\rceil$.

The depth does not affect the clock frequency of the synchronous side, but as our analysis measures time in clock cycles, the latency of going up and down the arbiter tree must be converted into clock cycles. The critical path in each arbiter stage consists of 4 LUTs down towards the root and 1 LUT upwards. The critical path for the tree is therefore $d*5$ LUTs. Translating this into a number of clock cycles, k, depends on many factors such as FPGA technology used, layout, routing, etc., and must be rounded up to an integer number of clock cycles. As Patmos is clocked at 80 MHz, $k=1$ even for a very large number of processors.

Both uncontended acquisition and release require 1 clock cycle for the command, 1 clock cycle for the synchronization register, and k clock cycles for the arbiter tree, in total $2+k$. With $k=1$, this corresponds to 3 clock cycles. For a contended lock the worst-case number of processors that may be served before a processor is granted the lock is given by equation 1. We therefore have:

$$\begin{aligned}\mathrm{WCET}_{\mathrm{acquisition}} &= 2+k+\sum_{i=1}^{n'-1}(2+k+2+k+c_i)\\ &= 2+k+\sum_{i=1}^{n'-1}(2k+4+c_i)\\ &= 2+k+(n'-1)(2k+4)+\sum_{i=1}^{n'-1}c_i\\ &\approx 6n'-3+\sum_{i=1}^{n'-1}c_i\end{aligned} \tag{4}$$

VI. Evaluation

In this section we evaluate the locking units. First, we compare the resource usage when synthesizing the units on an FPGA. We then compare the performance of the units for both uncontended and contended locks. We also compare the performance of the pthread mutex on top of each locking unit. Finally, we compare the Hardlock to a lock-free implementation.

Throughout the section, n represent the number of cores and m the number of locks.

A. Resource Consumption

We use Quartus II Web Edition version 15.0 [11] and the Altera DE2-115 development board with the Cyclone IV FPGA for the evaluation. Table IV contains the hardware resource usage of the 3 locking units when synthesized with different number of locks and connected to different number of Patmos cores. The resources are specified as logic cells, registers and memory bits. A logic cell on Cyclone IV contains a 4-bit lookup table. To put the numbers in relation to the resource usage of a processor, a simple RISC style processor requires about 3000 logic cells and Patmos consumes about 9000 logic cells.

Overall, the Hardlock has the lowest hardware usage. The number of registers corresponds to the number of queues and current owner registers, one for each lock, i.e., $m \times (n + log_2(n)) + n$. The cell count corresponds to the number of locks and connected cores. When the number of cores increases, the queue and current registers grow, which results in the round-robin arbiter growing. When the lock count increases, the unit grow almost proportionally with the count. Only when both increasing the number of cores and locks does the unit grow significantly.

The CAS unit has the highest register usage, as it must store all the data of a request, such as address, expected value, new value, etc. Additionally, the CAS unit is the only one that consumes memory bits, corresponding to the number and width of the SPM fields, i.e., $m \times 32$ bits. Increasing the number of locks almost only increases the number of memory bits, which is cheaper than the other resources. Therefore, the CAS unit has the best scalability with regards to the number of locks.

Like the CAS unit, the asynchronous lock must also store more request data than the Hardlock, resulting in a higher register usage. However, the cell count is almost twice that of the other units. This is caused by forcing the synthesis tools to keep separate parts of the mutex and arbiter in their own cells. If the unit were to be implemented on an ASIC, the resource usage would be more comparable to, and possibly even lower than, that of the other units.

B. Uncontended Performance

We have created a small test program to measure the performance of uncontended locks when running on the FPGA. Each core has its own lock to ensure that no core experiences contention. The program executes on all cores and loops 1024 times, each time measuring the time before and after acquiring a lock, as well as the time before and after releasing a lock:

```
time1 = read_time();
lock(id);
time2 = read_time();
unlock(id);
time3 = read_time();
```

Table V contains the results of running the program on 8 cores using 16 locks. The tables contain the average, minimum, and maximum time in clock cycles for both operations on all lock types. When using the locking units directly, the measured WCET is exactly equal to our analyzed WCET from Table II, which shows the Hardlock and asynchronous lock having the best performance, although only the Hardlock can guarantee its performance.

Using the pthread mutex on top of each locking unit has absolutely no impact on the performance, compared to using the locking units by themselves. Although this might seem unexpected, the test is optimized to have no overhead in addition to the locking itself. The compiler is therefore able to inline the *pthread_mutex_lock* and *pthread_mutex_unlock* functions, which in turn merely call the raw locking routines. The contended performance test achieves a more accurate view.

C. Contended Performance

We have also created a small test program to measure the performance of contended locks when running on the FPGA. The program runs on 8 Patmos cores. Each core executes a loop where it takes a lock, stalls for a given number of clock cycles and then releases the lock, as shown in the following excerpt:

```
int lckid = 0;
for(int i = 0; i < cnt; i++)
{
        lock(lckid);
        timer_setup(wait_cycles)
        timer_wait()
        unlock(lckid);
        if(++lckid >= lckcnt)
                lckid = 0;
}
```

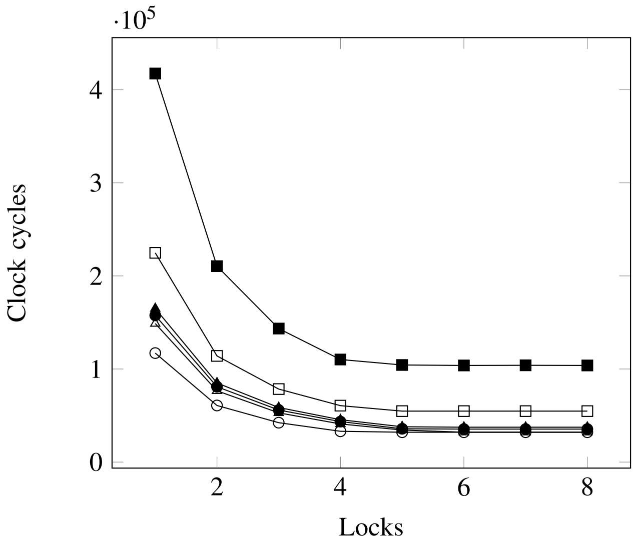
The loop executed 1000 times, and for each iteration, each core tries to acquire a different lock than in the previous iteration. The number of locks used in the loop affects the contention experienced by the cores, i.e., when the lock count is 1, there is full contention, whereas when then lock count is 8, there should be little to no contention. The time a core stalls corresponds to its critical section. All cores have the same critical section length. The time is measured from when the first core starts executing the loop until all cores have finished.

Figure 6 contains the results of executing the program with critical sections of 10 to 10000 clock cycles, using each locking unit with and without pthread mutex. The y-axis represents the execution time in clock cycles and the x-axis represents the number of locks used.

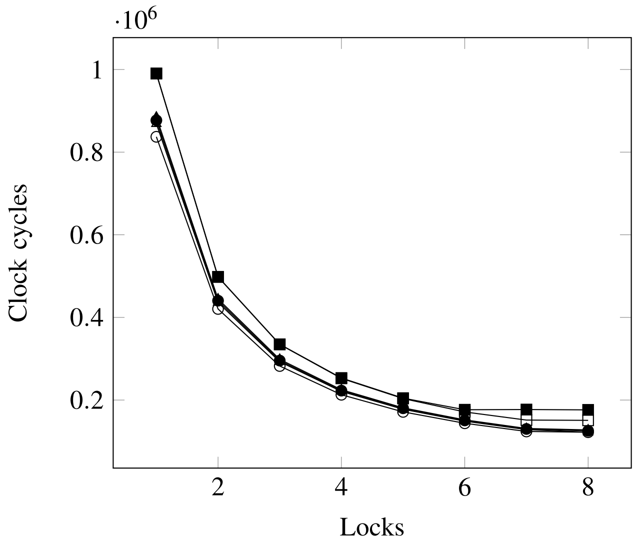
The results show that Hardlock generally performs the best, with the asynchronous lock performing close to the Hardlock, and the CAS being the slowest. However, we mainly observe this difference with shorter critical sections. As the critical sections grow larger, the difference in performance becomes negligible, e.g., when the critical section is 1000 or 10000 clock cycles. We also observe the overhead of the pthread mutexes for the lower critical sections, particularly for CAS. However, this overhead also becomes negligible as the critical sections increase. Using pthread mutexes therefore does not infer a much larger overhead.

D. Comparison with a Lock-Free Algorithm

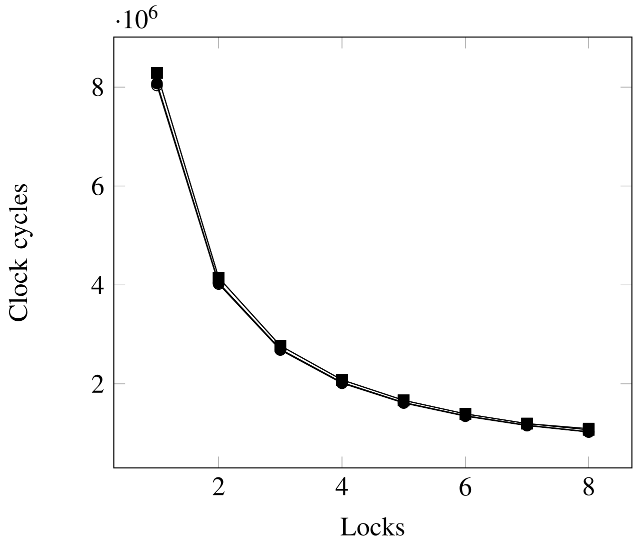
An alternative to using locks when accessing a data structure, is to use a lock-free implementation of the access to that structure. We have created two implementations of a stack, one using the Hardlock to synchronize access, and the other using CAS for a lock-free access. For both implementations, the stack consists of a top pointer that points to the top element



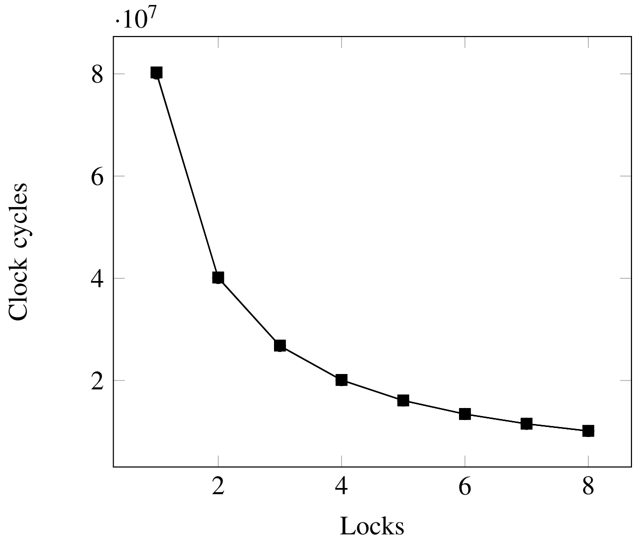
(a) 10 clock cycle critical section



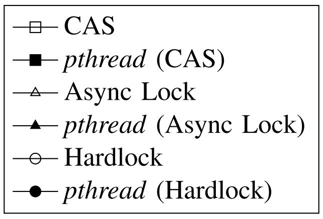
(b) 100 clock cycle critical section



(c) 1000 clock cycle critical section



(d) 10000 clock cycle critical section



(e) Legend

Fig. 6: Contended lock performance of the different locking units with different critical section durations

TABLE IV: Hardware resource usage for each locking unit

Lock type	Cores	Locks	Logic Cells	Registers	Memory Bits
Hardlock	8	16	807	184	0
CAS	8	16	850	629	512
Asynchronous Lock	8	16	1452	392	0

TABLE V: Measured uncontended lock performance in clock cycles

	Acquisition			Release		
Lock type	Avg.	Min.	Max.	Avg.	Min.	Max.
Hardlock	1.9	1	2	1	1	1
CAS	18	11	32	23	23	23
Asynchronous Lock	3	3	3	3	3	3
Pthread Mutex						
Hardlock	1.9	1	2	1	1	1
CAS	18	11	32	23	23	23
Asynchronous Lock	3	3	3	3	3	3

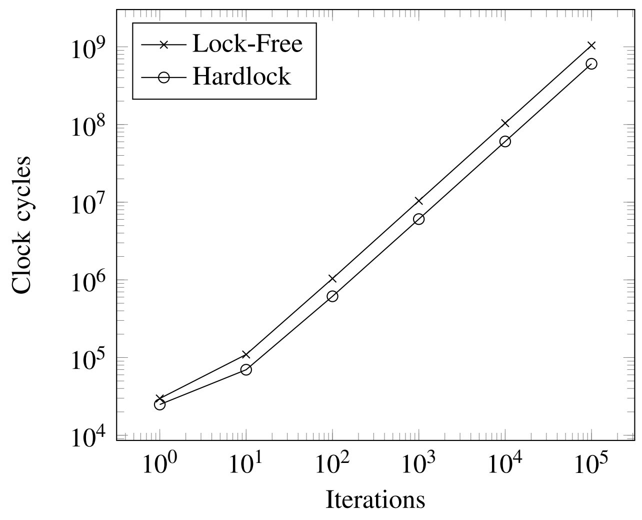


Fig. 7: Performance comparison of the implementation of a stack data structure with the Hardlock or a lock-free algorithm stack. The number of iterations indicates how many times each core will execute two pops and two pushes, in that order

of the stack, and elements that contain a value and a next element pointer, i.e., a single linked list.

The lock-free implementation uses CAS, although only for manipulating the top. The top is therefore located in the CAS unit, whereas all the elements are locate in main memory. To ensure a fair comparison, this is also done for the Hardlock implementation. Some lock-free implementations rely on double-word CAS to avoid the ABA-problem by swapping not only the top pointer, but also a element version counter, which ensures that a pop knows whether other threads popped and pushed the element before the pop finalizes the swap. Our CAS unit is only single-word, so we reserve the 8 most significant bits of the top pointer for the version, and add a version counter to each element. Before pushing an element onto the stack, a thread increments the element's version counter, and then combines this value with the address of the element to create the new top pointer.

We created a program to test the performance of the two stack implementations. The main core starts by generating 2 elements per core and pushing them all on the stack, after which it takes a time stamp, and starts all the cores. Each core then pops two elements and pushes them in reverse order. Each core does this for a number of iterations specified by an iteration constant. When all cores finish, the main core takes the end time stamp.

Figure 7 shows the result of running the program on 8 cores. The x-axis specifies the number of iterations that each core popped and pushed 2 elements, and the y-axis specifies the number of cycles it took for all the cores to finish. Note, that both scales are logarithmic, so the differences look far less than on a linear scale. For a single iteration, the lock-free implementation executes for roughly 19% more

cycles. Most likely, the cores spend most of the time on setup overhead and not on modifying the stack. However, as the number of iterations increases, the difference becomes 57%, increasing towards 72%, showing that the lock-free stack has some additional overhead at high congestion.

Source Access

The T-CREST project is an open-source project and therefore we also provide the locking units and the evaluation benchmarks as open source. Our work is available at [34]. A README explaining how to run the tests and reproduce the results is available at [35] and we provide an Ubuntu virtual machine [36] containing all the build tools.

VII. Conclusion

Performance improvements are currently achieved by building multicore processors. Tasks split into several threads need to communicate and coordinate their work, which is commonly done with shared data structures protected by locks. Therefore, the performance of locking is an important aspect of the overall performance.

In this paper we presented 3 dedicated locking units in hardware: the Hardlock, an asynchronous locking unit and compare-and-swap on a shared scratch-pad memory. The Hardlock bounds uncontended lock acquisition to 2 clock cycles and releases to 1. Neither the asynchronous lock nor compare-and-swap properly bound acquisition. However, the measured performances show that the asynchronous lock performs nearly as well as the Hardlock, and for critical sections of 1000 cycles or higher, the performance difference between the locks is negligible. Additionally, building pthread mutexes on top of the locking units does not incur a large overhead, particularly for locking routines. Comparing a stack implementation using the Hardlock to a lock-free implementation shows that the lock-free implementation incurs an overhead of 19-72

Generally, we find that for normal systems, which of the locking units one might use does not significantly impact the performance. However, for time-predictable computer architectures, the starvation free behavior of both the Hardlock and the asynchronous lock is a benefit. Additionally, the predictable and low worst-case execution time of the Hardlock gives it a benefit over both other two locking units. We therefore find that the Hardlock is the best fit for time-predictable systems.

Acknowledgment

This work was partially funded by the Faroese Research Council under project no. 0607.

References

[1] Sahar Abbaspour, Florian Brandner, and Martin Schoeberl. A time-predictable stack cache. In *Proceedings of the 9th Workshop on Software Technologies for Embedded and Ubiquitous Systems*, 2013.

[2] Accellera Systems Initiative. Open Core Protocol specification, release 3.0. Available at http://accellera.org/downloads/standards/ocp/, 2013.

[3] Sara Afshar, Moris Behnam, Reinder J Bril, and Thomas Nolte. Per processor spin-based protocols for multiprocessor real-time systems. *Leibniz Transactions on Embedded Systems*, 4(2):03–1, 2017.

[4] Altera. Embedded peripherals IP user guide, June 2011.

[5] Rubén Braojos, Ahmed Dogan, Ivan Beretta, Giovanni Ansaloni, and David Atienza. Hardware/software approach for code synchronization in low-power multi-core sensor nodes. In *Proceedings of the conference on Design, Automation & Test in Europe*, page 168. European Design and Automation Association, 2014.

[6] John B Carter, Chen-Chi Kuo, and Ravindra Kuramkote. A comparison of software and hardware synchronization mechanisms for distributed shared memory multiprocessors. *University of Utah, Salt Lake City, Utah*, 84112, 1996.

[7] Philipp Degasperi, Stefan Hepp, Wolfgang Puffitsch, and Martin Schoeberl. A method cache for Patmos. In *Proceedings of the 17th IEEE Symposium on Object/Component/Service-oriented Real-time Distributed Computing (ISORC 2014)*, pages 100–108, Reno, Nevada, USA, June 2014. IEEE.

[8] Henrik Enggaard Hansen, Emad Jacob Maroun, Andreas Toftegaard Kristensen, Jimmi Marquart, and Martin Schoeberl. A shared scratchpad memory with synchronization support. In *2017 IEEE Nordic Circuits and Systems Conference (NORCAS): NORCHIP and International Symposium of System-on-Chip (SoC)*, pages 1–6, Oct 2017.

[9] Reinhold Heckmann and Christian Ferdinand. Worst-case execution time prediction by static program analysis. Technical report, AbsInt Angewandte Informatik GmbH. [Online, last accessed November 2013].

[10] Stefan Hepp, Benedikt Huber, Jens Knoop, Daniel Prokesch, and Peter P. Puschner. The platin tool kit - the T-CREST approach for compiler and WCET integration. In *Proceedings 18th Kolloquium Programmiersprachen und Grundlagen der Programmierung, KPS 2015, Pörtschach, Austria, October 5-7, 2015*, 2015.

[11] Intel Corporation. Quartus. http://dl.altera.com/15.0/?edition=web, 2017.

[12] R Terrell II James. Reusable, operating system aware hardware mutex, November 27 2012. US Patent 8,321,872.

[13] Evangelia Kasapaki, Martin Schoeberl, Rasmus Bo Sørensen, Christian T. Müller, Kees Goossens, and Jens Sparsø. Argo: A real-time network-on-chip architecture with an efficient GALS implementation. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 24:479–492, 2016.

[14] Alex Kogan and Erez Petrank. Wait-free queues with multiple enqueuers and dequeuers. In *ACM SIGPLAN Notices*, volume 46, pages 223–234. ACM, 2011.

[15] Pasi Kolinummi and Juhani Vehvilainen. Hardware semaphore intended for a multi-processor system, December 26 2006. US Patent 7,155,551.

[16] Leslie Lamport. New solution of Dijkstra's concurrent programming problem. *Commun Acm*, 17(8):453–455, 1974.

[17] Chris Lattner and Vikram S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization (CGO'04)*, pages 75–88. IEEE Computer Society, 2004.

[18] Sanghoon Lee, Devesh Tiwari, Yan Solihin, and James Tuck. Haqu: Hardware-accelerated queueing for fine-grained threading on a chip multiprocessor. In *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, pages 99–110. IEEE, 2011.

[19] Adam Milik and Edward Hrynkiewicz. Distributed plc based on multicore cpus-architecture and programming. *IFAC-PapersOnLine*, 49(25):1–7, 2016.

[20] Dale Parson. Resource management in a processor-based system using hardware queues, July 28 2005. US Patent App. 10/764,967.

[21] Srishty Patel, Rajshekar Kalayappan, Ishani Mahajan, and Smruti R Sarangi. A hardware implementation of the mcas synchronization primitive. In *2017 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 918–921. IEEE, 2017.

[22] Peter Puschner, Raimund Kirner, Benedikt Huber, and Daniel Prokesch. Compiling for time predictability. In Frank Ortmeier and Peter Daniel, editors, *Computer Safety, Reliability, and Security*, volume 7613 of *Lecture Notes in Computer Science*, pages 382–391. Springer Berlin / Heidelberg, 2012.

[23] Martin Schoeberl, Sahar Abbaspour, Benny Akesson, Neil Audsley, Raffaele Capasso, Jamie Garside, Kees Goossens, Sven Goossens, Scott Hansen, Reinhold Heckmann, Stefan Hepp, Benedikt Huber, Alexander Jordan, Evangelia Kasapaki, Jens Knoop, Yonghui Li, Daniel Prokesch, Wolfgang Puffitsch, Peter Puschner, André Rocha, Cláudio Silva, Jens Sparsø, and Alessandro Tocchi. T-CREST: Time-predictable multi-core architecture for embedded systems. *Journal of Systems Architecture*, 61(9):449–471, 2015.

[24] Martin Schoeberl, Florian Brandner, Stefan Hepp, Wolfgang Puffitsch, and Daniel Prokesch. Patmos reference handbook. Technical report, Technical University of Denmark, 2014.

[25] Martin Schoeberl, David VH Chong, Wolfgang Puffitsch, and Jens Sparsø. A time-predictable memory network-on-chip. In *Proceedings of the 14th International Workshop on Worst-Case Execution Time Analysis (WCET 2014)*, pages 53–62, Madrid, Spain, July 2014.

[26] Martin Schoeberl, Luca Pezzarossa, and Jens Sparsø. A multicore processor for time-critical applications. *IEEE Design Test*, 35:38–47, 2018.

[27] Martin Schoeberl, Wolfgang Puffitsch, Stefan Hepp, Benedikt Huber, and Daniel Prokesch. Patmos: A time-predictable microprocessor. *Real-Time Systems*, 54(2):389–423, Apr 2018.

[28] Martin Schoeberl, Pascal Schleuniger, Wolfgang Puffitsch, Florian Brandner, Christian W. Probst, Sven Karlsson, and Tommy Thorn. Towards a time-predictable dual-issue microprocessor: The Patmos approach. In *First Workshop on Bringing Theory to Practice: Predictability and Performance in Embedded Systems (PPES 2011)*, pages 11–20, Grenoble, France, March 2011.

[29] Lui Sha, Ragunathan Rajkumar, and John P Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on computers*, 39(9):1175–1185, 1990.

[30] Rasmus Bo Sørensen, Wolfgang Puffitsch, Martin Schoeberl, and Jens Sparsø. Message passing on a time-predictable multicore processor. In *Proceedings of the 17th IEEE Symposium on Real-time Distributed Computing (ISORC 2015)*, pages 51–59, Auckland, New Zealand, April 2015. IEEE.

[31] Jens Sparsø. Asynchronous circuit design – a tutorial. In Jens Sparsø and Steve Furber, editors, *Principles of asynchronous circuit design – A systems perspective*, chapter 1-8, pages 1–152. Kluwer Academic Publishers, 2001.

[32] Tórur Biskopstø Strøm, Wolfgang Puffitsch, and Martin Schoeberl. Hardware locks for a real-time java chip multiprocessor. *Concurrency and Computation: Practice and Experience*, 29(6), 2017.

[33] Tórur Biskopstø Strøm and Martin Schoeberl. Hardlock: A concurrent real-time multicore locking unit. In *2018 IEEE 21st International Symposium on Real-Time Distributed Computing (ISORC)*, pages 9–16, May 2018.

[34] T-CREST Group. Patmos source. https://github.com/t-crest/patmos, 2017.

[35] T-CREST Group. Readme explaining how to run the hardlock tests. https://github.com/t-crest/patmos/tree/master/c/apps/hardlock, 2017.

[36] T-CREST Group. Virtual machine with all the necessary T-CREST tools and sources. http://patmos.compute.dtu.dk/patmos-dev.zip, 2017.

[37] Shahar Timnat and Erez Petrank. A practical wait-free simulation for lock-free data structures. In *ACM SIGPLAN Notices*, volume 49, pages 357–368. ACM, 2014.

[38] Cheng-Ming Tuan. Apparatus and method for hardware semaphore, June 22 2006. US Patent App. 11/116,972.