Hardlock: a Concurrent Real-Time Multicore Locking Unit

Tórur Biskopstø Strøm and Martin Schoeberl

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

***Abstract*—To use multicore processors, an application needs to split computation into several threads that execute on different processing cores. As those threads work together towards a common goal, they need to exchange data in a controlled way. A common communication paradigm between cooperating threads is using shared data structures protected by locks.**

Implementing a lock on top of shared memory can easily result in a bottleneck on a multicore processor due to the congestion on the shared memory. However, the number of locks in use is usually low and using the large external memory to support locks is over-provisioning a resource. This paper presents an efficient implementation of locking by providing dedicated hardware support for locking on-chip. This locking unit supports a restricted number of locks without the need to get off-chip. The unit can process lock acquisitions in 2 clock cycles and releases in 1 clock cycle.

I. Introduction

On any system where multiple tasks interleave, and/or execute concurrently, the need for synchronization arises to prevent data model corruption. An example would be 2 tasks, τ_1, that updates a record structure, and τ_2, which reads the record. If τ_2 reads the record before τ_1 is finished updating the entire record, τ_2 may read incorrect data.

One solution to this issue is locking. A lock is a synchronization mechanism that protects some critical region and employs the notion of ownership, e.g., if τ_1 has acquired the lock that protects the record, τ_2 will not read the record until τ_1 has released the lock and τ_2 has acquired it.

Locks, as well as other synchronization mechanisms (mutexes, semaphores, etc.), consist of several operations that must execute in a synchronous manner, i.e., be atomic. For locks, this means that a task should not be able to partially own a lock. It either owns it or not. Acquiring or releasing a lock should mimic a single operation. Therefore, locks and other synchronization mechanisms commonly employ hardware supported atomic operations to ensure that the higher-level operations become atomic.

Compare-And-Swap (CAS) is an example of a common hardware supported atomic operation. It checks whether a memory location has the expected value and, if so, swaps it with a newly supplied value, all in a single operation. Locks built upon CAS work generally well but suffer from two potential issues. Firstly, 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. Secondly, the locking operation requires multiple memory operations in addition to CAS. When multiple cores connect to the same memory, an arbiter handles access to the memory. There are different arbitration policies, but often, in the worst case, the time that an arbiter prevents a core from accessing the memory scales linearly with the number of cores, because another core is using it. This potentially results in the locking time scaling linearly as well.

Whilst these issues are undesirable on any system, they are egregious on real-time systems where tasks' worst-case execution time (WCET) affects, or is the basis for, their scheduling. If the potential for starvation is present, hard guarantees on the WCET cannot be made, and the schedules are therefore potentially broken.

In this paper we present and evaluate our design and implementation of Hardlock, a concurrent real-time hardware locking unit. We designed the unit with the goal of minimizing the WCET and being starvation free. It is based on previous work from our patent application [1], although this is the first time it is implemented in Chisel, as well as integrated with a multicore processor and evaluated.

This paper is organized in 5 sections: The next section, Section II, presents related work in the field and Section III provides background on the T-CREST multicore platform, which is used for the implementation and evaluation of the locking unit. Section IV describes our design and our implementation. Section V evaluates our implementation and Section VI concludes the paper.

II. Related Work

Carter et al. [2] compare 6 lock implementations, 2 of which 2 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. [3] describe a hardware implementation of a multi-word CAS (MCAS) for multi-core systems. They find that on average their implementation is up to 13.8 times faster than locks. They also claim that starvation does not occur, however, it is unclear whether the claim applies to MCAS requests themselves or whether it also applies to multiple cores continuously making MCAS requests. From the implementation description it seems to be the former, which means that the potential starvation issue of CAS also applies to MCAS.

Afshar et al. [4] 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 [5] 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 multiple producers for the same semaphore are allowed, and if so, how the semaphore handles arbitration.

Braojos et al. [6] 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.

US Patent 5,276,886 [7] provides atomic access to single-bit locking flags, but does not provide any support for more sophisticated locking protocols.

Altera provides a "mutex core" [8], 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 (according to priorities of a FIFO policy) must be done in software.

US Patent 8,321,872 [9] 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 actual handling of the wait queue is carried out by the operating system.

The hardware unit described in US Patent 7,062,583 [10] 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 [11] builds on that patent, but notably extends it with the possibility to allocate semaphores dynamically.

US Patent Application 10/764,967 [12] 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.

Besides using a CAS operation on a shared, external main memory, an on-chip shared scratchpad memory can support synchronization [13]. 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 [14] 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.

III. The T-CREST Platform

This section provides an overview of the T-CREST multicore platform that we use to implement and evaluate our

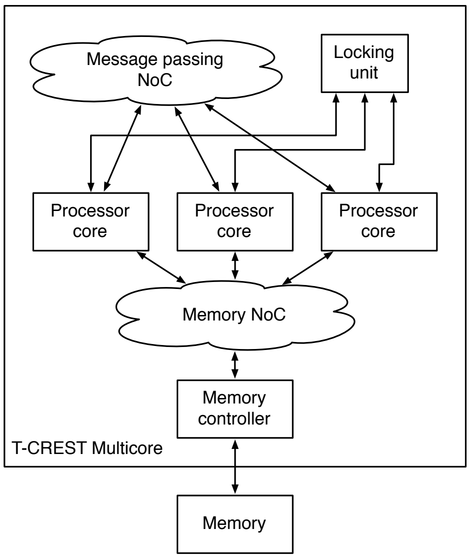

Fig. 1: The T-CREST multicore architecture with several processor cores connected to two NoCs: one for core-to-core message passing and one for access to the shared, external memory and connected to the locking unit.

synchronization unit. Note that our locking unit is not restricted to the implementation language, nor the processor, used. However, integrating it with a working platform shows the feasibility of our design. Figure 1 gives an overview of the T-CREST multicore processor, including the locking unit. Several processor cores connect to shared memory, a message passing network-on-chip, and the locking unit.

The T-CREST multicore is a processor optimized for low WCET and to simplify static WCET analysis [15], [16]. All components have been designed to be time-predictable and to enable WCET analysis. The platform consists of several processing cores connected to two networks-on-chip (NoCs): (1) a NoC for message passing between cores, called Argo [17] and (2) a memory NoC for access to shared, external memory [18].

The processing cores are RISC style processors, called Patmos [19], [20]. Patmos is written in Chisel [21], a new hardware construction language. Therefore, to easily integrate with the cores, our locking unit implementation is also written in Chisel. The implementation is part of the Patmos source repository and therefore does not require separate retrieval.

Patmos contains special cache memories optimized for WCET analysis and a local scratchpad memory. Patmos is supported by a compiler that optimizes for WCET [22], based on the LLVM framework [23]. The compiler provides flow facts for the aiT [24] WCET analysis tool from AbsInt and also incorporates WCET path information for aiT. Furthermore, an open-source WCET analysis tool, platin [25], is available for Patmos.

The Argo NoC [17] 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.

TABLE I: *io* bundle composition

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

The Argo NoC is also useful in supporting synchronization primitives [26]. 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 [27] and then Lamport's bakery algorithm [28] is used to implement locks. This implementation of locks is inefficient, so the proposed locking unit replaces it.

IV. Design of the Locking Unit

We designed our locking unit with two main goals in mind:

- Minimizing WCET
- Being starvation free

To this end, it is important that cores interacting with unrelated locks do not experience interference from each other's interactions, i.e., cores can access the unit concurrently and not through some arbiter.

Figure 2 depicts the general architecture of the Hardlock. *n* cores connect to the Hardlock through their respective *io* signal bundle and can issue requests to any of the *m* locks. Requests take one of two forms:

- Acquisitions, that request ownership of a specific lock
- Releases, that release ownership of a specific lock

Table I shows the composition of an io bundle. Depending on the specific implementation, it might use the signals directly or adapted them to another protocol. In our Patmos integration we adapt the signals to the T-CREST Open Core Protocol (OCP) implementation. The Hardlock will stall the Patmos core's pipeline while it processes the core's request or if the core is 'blocked', i.e., the core is awaiting a lock. Therefore, although a core can own multiple locks, it can only have one outstanding request at any instance. This can potentially increase the WCET of unrelated threads on the stalled core. However, the T-CREST platform only executes one thread per core without any context switching, making the point moot. Furthermore, stalling is is not a requirement dictated by the Hardlock, as cores might also make multiple requests and poll, or register an interrupt, when a lock is available. Nevertheless, stalling simplifies the analysis and the implementation.

Figure 2 further shows how the input signals from each core travel to each lock, depending on their *sel* signals. Each lock produces a blocking signal for each connected core. These

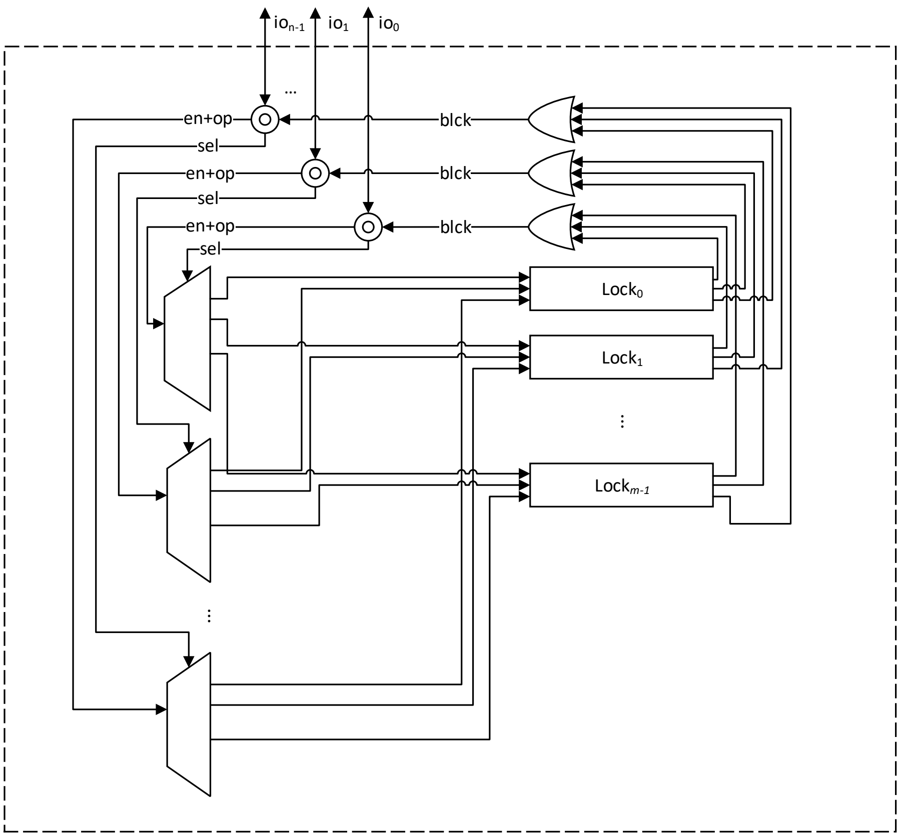

Fig. 2: The overall architecture of the Hardlock connecting n cores to m locks

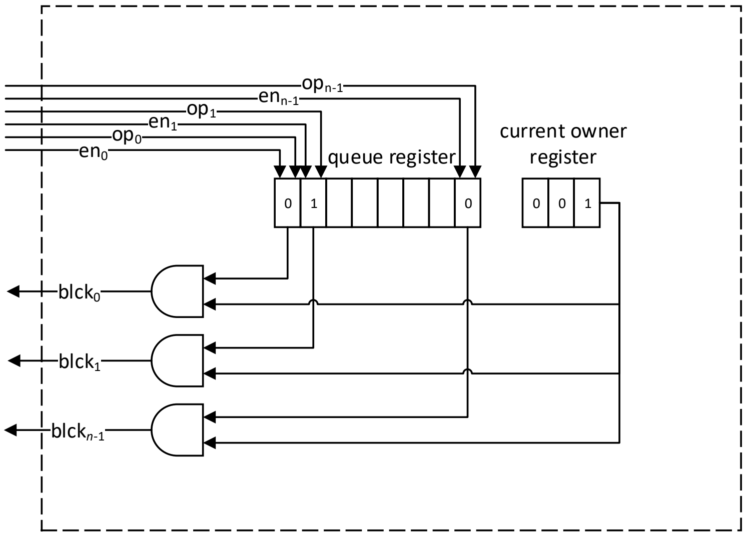

Fig. 3: Composition of a lock from Figure 2 showing core signals connected to respective queue register bits and the current register

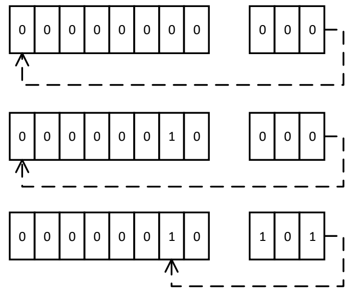

Fig. 4: Current register updated to the next lock owner with a priority-encoder

signals merge with respect to the core number, producing a single *blck* signal per core, which connects with the *io* bundle.

Figure 3 shows the composition of a single lock. Each lock consists of a queue and a current (owner) register. The queue consists of *n* single bit registers where each bit belongs to one core. The *en* and *op* signals from each core connect to the respective registers. A set bit in the register 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 it is waiting for it. An unset bit indicates a release request, or simply the absence of a request.

The current owner register consists of $\log_2(n)$ bits and holds the number of the core currently owning the lock. The unit looks through the queue in round-robin fashion, starting from the last owner and looping around at the end. When the unit reaches a register containing a set bit, it sets the current register to that register. When the queue bit currently pointed at is unset, the unit again looks through the queue until it reaches another set bit. A circular priority encoder does this processing. Therefore, finding a set queue bit and updating the current register is a single cycle operation, as shown in Figure 4. If no other core is contending for the same lock, the unit processes acquisition requests in 2 cycles:

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

Note that the core executes the next instruction during 3rd cycle, which is why it does not count towards the lock acquisition time.

The benefits of iterating round-robin style are twofold:

1) It simplifies the hardware compared to a real queue, such as FIFO, as the order of the requests are irrelevant, so there is no hardware tracking the order. Cores only access their own queue bit. This allows cores to truly concurrently issue requests. Additionally, release requests only need to unset the request bit.
2) In contrast with, e.g., Compare-and-swap (CAS), if no core indefinitely holds a lock, the unit guarantees to eventually process all requests, preventing starvation.

TABLE II: WCET Hardlock request performance in cycles

Acquisition	2
Release	1

An upper bound for this is the sum of acquisition time a, release time r and critical section c, of all cores potentially involved in the lock for 1 round of round-robin, i.e., $T_{WCET} = \sum_{i=0}^{n} a + r + c_i = n * (2+1) + \sum_{i=0}^{n} c_i$, where i is the id of the core, and n is the number of potentially involved cores.

Figure 3 also shows how the current register value together with a core's queue register generate the core's *blck* signal for the lock. Shown in Figure 2, the unit merges each core's *blck* signal from each lock, i.e., the unit merges $blck_0$ from each lock to create a single *blck* which it routes to io_0.

V. Evaluation

Our evaluation consists of simulation, synthesis, and measurements.

Simulation

We simulate the Hardlock using Chisel's built in testing framework. From the simulation we can verify that the unit behaves correctly and derive WCET performance. Table II shows the WCET of a lock request and release for the Hardlock. The circular priority encoders in the Hardlock find the next awaiting cores within a single cycle. We add 1 cycle for making the request and updating the queue register.

The Hardlock does not dictate how long cores hold the locks, as this is application dependent. Also, the unit does 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 unit is the limited number of locks. Whilst this is configurable during hardware generation, it does not have the benefit of CAS where every memory address is a potential lock, i.e., practically infinite locks. However, given an application requiring a limited number of locks, the unit provides nearly optimal performance with regards to cycle count.

Synthesis

For synthesis we target the Cyclone IV FPGA used on the Altera DE2-115 development board. We use the Quartus II Web Edition version 15.0 [29]. Table III contains both the total hardware resources of the unit, as well as the resources used by the OCP connection wrapper, when synthesized with different number of locks and connected to different number of Patmos cores, in logic cells and registers (a logic cell contains a 4-bit lookup table in the Cyclone IV). 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. Therefore, the hardware resource consumption of the Hardlock, which serves several processors, is negligible.

The number of registers used correspond to the number of queue and current registers, i.e., $m \times (n + log_2(n)) + n$.

TABLE III: Hardlock hardware resource usage

		Total		OCP Wrapper	
Cores	Locks	Logic Cells	Registers	Logic Cells	Registers
2	2	15	8	3	2
2	4	32	14	2	2
2	16	105	50	5	2
4	2	43	16	4	4
4	4	96	28	4	4
4	16	331	100	14	4
8	2	110	30	8	8
8	4	229	52	8	8
8	16	807	184	17	8

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 priority encoder 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. Therefore, we find that the unit is useful with a high number of cores as long as we keep the number of locks low, and vice versa.

Another potential issue with connecting a high number of cores is the limitation on the clock frequency. The circular priority encoder must check all queue registers, which can lead to a very long critical path. However, with the number of processing cores fitting into the FPGA, we did not run into that limitation.

Measurement

We have created a small test program to measure the performance of uncontended locks when running on the FPGA and using the Hardlock. The program contains 2 loops that execute on all cores and each run 1024 iterations. The first loop measures the time before and after acquiring a lock, as well as the time before and after releasing a lock:

```
stop1 = TIMER_CLK_LOW;
lock(id);
stop2 = TIMER_CLK_LOW;
unlock(id);
stop3 = TIMER_CLK_LOW;
```

The second loop measures the time before and after acquiring and immediately releasing a lock:

```
stop1 = TIMER_CLK_LOW;
lock(id);
unlock(id);
stop2 = TIMER_CLK_LOW;
```

For both loops, each core has its own lock to ensure that no core experiences contention.

Table IV contains the results of running the program on 8 cores using 8 locks. Within an operation and core, the average, minimum, and maximum cycle count is identical. In addition, the measured time is equal to our WCET from Table II. Only core 7 has a different acquisition time. This is a result of the circular priority encoder setting the current register to the highest value when the queue is empty. When core 7 then requests the lock, it is already the owner, and therefore allowed to continue a cycle sooner. This is therefore the best-case scenario.

Note, that we also tested the program with different numbers of cores and locks. If each core has its own lock, the results are the same, with the last core always spending 1 cycle less during acquisition.

We have also created 2 test programs to measure and compare the performance of the Hardlock with the SSPM lock [13] for contended locks. The programs are functionally identical, but one uses our locking unit and the other uses the SSPM lock. We therefore refer to them simply as a single program.

The program runs on 4 Patmos cores that all share a single integer array. The size of the array corresponds to the number of locks used, as each lock protects its respective element in the array. All cores also share a counter (*cnt1*) that starts at 10000. Before a core tries to acquire an element specific lock, it first locks the counter, reads its value, decrements it, and unlocks it, as shown in the following excerpt:

```
lock(0);
if(cnt1 == 0)
{
  unlock(0);
  while(cnt2 > 0) {asm("");}
  break;
}
int _cnt = cnt1--;
unlock(0);
```

The core then calculates element and lock to use, after which it acquires the lock, increments the array element, and then executes a busy-wait to simulate a longer critical section, before releasing the lock:

```
int fldid = _cnt%i;
int lckid = fldid%LCK_CNT;

lock(lckid);
data[fldid]++;
for(int j = 0; j < WAIT; j++)
  asm("");
unlock(lckid);
```

When the counter reaches 0 the cores have incremented the array elements a total of 10000 times. The program then verifies correct execution and lock behaviour by summing all entries and comparing the sum to 10000.

The counter creates a possible point of contention across all cores. The individual element locks allow contention reduction by increasing the number of elements/locks, i.e., using 1 element lock means all cores try to update the same element, whereas with 8 locks, cores update different elements. The number of busy-wait iterations controls the length of each element lock's critical section.

Figure 5 and Figure 6 contain the results of executing the programs. The y-axis represents the total time in cycles it takes all cores to decrement the counter to 0. The x-axis represents the number of element locks used, i.e., for each busy-wait variant we vary the number of elements used from 1 to 8. The legend contains the lock types (SSPM and Hardlock) and the number of iterations in the busy-loop (10-10000).

TABLE IV: Measured uncontended lock performance in cycles

	Acquisition			Release			Acquisition + Release		
Core Nr.	Avg.	Min.	Max.	Avg.	Min.	Max.	Avg.	Min.	Max.
0	2	2	2	1	1	1	3	3	3
1	2	2	2	1	1	1	3	3	3
2	2	2	2	1	1	1	3	3	3
3	2	2	2	1	1	1	3	3	3
4	2	2	2	1	1	1	3	3	3
5	2	2	2	1	1	1	3	3	3
6	2	2	2	1	1	1	3	3	3
7	1	1	1	1	1	1	2	2	2

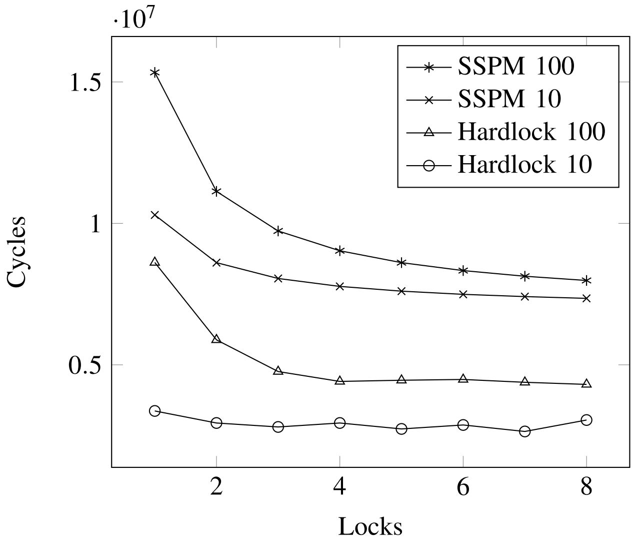

Fig. 5: Contended lock performance with small critical sections, showing the Hardlock performing faster

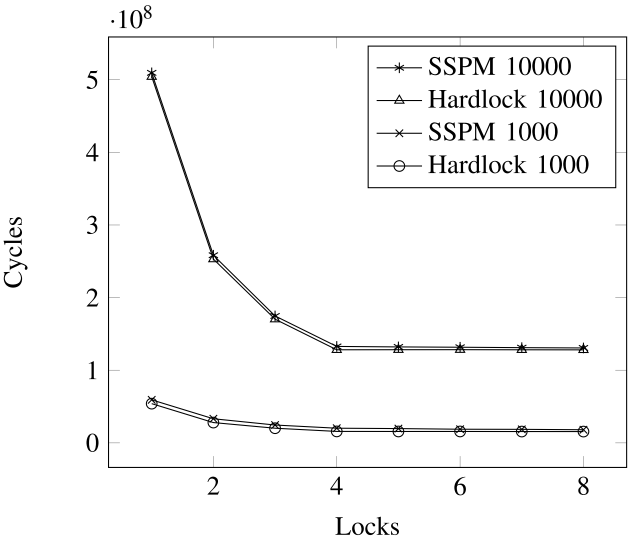

Fig. 6: Contended lock performance with large critical sections, showing the Hardlock and SSPM performing equally fast

The results show that the smaller the critical section is, the more significant the improvement of the Hardlock over the SSPM is, and vice versa, e.g., Figure 6 shows that if the critical section busy-waits for a thousand iterations, the benefits of the Hardlock are negligible, although the Hardlock always performs at least as well as the SSPM. For both locking mechanisms it is evident that having a single lock, and thereby the biggest contention, causes significant overhead. Increasing the lock count beyond the core count has no benefit. We attribute the slight variation seen for Hardlock 10 to locking the counter, as the lock used for the counter is the same as for the first array element.

Source Access

The T-CREST project is an open-source project and therefore we also provide the Hardlock unit and the evaluation benchmarks in open source. Our work is available at [30]. A README explaining how to run the tests and reproduce the results is available at [31] and we provide an Ubuntu virtual machine [32] containing all the build tools.

VI. 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 a dedicated locking unit in hardware that bounds the acquisition of locks to 2 cycles, if the lock is free, and releases in 1 cycle. With an implementation of the locking unit in an FPGA we showed that the hardware cost is relatively low, using 807 logic cells when supporting 8 cores and 16 locks.

We also ran test programs that verified the acquisition and release performance. Additionally, we compared the unit to another locking implementation and showed that, for short critical sections, programs can perform twice as fast.

Additionally, the process for selecting the next lock owner prevents starvation. Our unit is therefore a good solution for time-predictable computer architectures.

Acknowledgment

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

References

[1] Strøm, T.B.: A lock circuit for a multi-core processor (September 2015) WO Patent App. PCT/EP2015/054,491.

[2] Carter, J.B., Kuo, C.C., Kuramkote, R.: A comparison of software and hardware synchronization mechanisms for distributed shared memory multiprocessors. University of Utah, Salt Lake City, Utah **84112** (1996)

[3] Patel, S., Kalayappan, R., Mahajan, I., Sarangi, S.R.: A hardware implementation of the mcas synchronization primitive. In: 2017 Design, Automation & Test in Europe Conference & Exhibition (DATE), IEEE (2017) 918–921

[4] Braojos, R., Dogan, A., Beretta, I., Ansaloni, G., Atienza, D.: Hardware/-software approach for code synchronization in low-power multi-core sensor nodes. In: Design, Automation and Test in Europe Conference and Exhibition (DATE), 2014, IEEE (2014) 1–6

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

[6] Afshar, S., Behnam, M., Bril, R.J., Nolte, T.: Per processor spin-based protocols for multiprocessor real-time systems. Leibniz Transactions on Embedded Systems **4**(2) (2017)

[7] Dror, A.: Hardware semaphores in a multi-processor environment (January 4 1994) US Patent 5,276,886.

[8] Altera: Embedded peripherals IP user guide (June 2011)

[9] Terrell, II, J.R.: Reusable, operating system aware hardware mutex (November 27 2012) US Patent 8,321,872.

[10] Kolinummi, P., Vehvilainen, J.: Hardware semaphore intended for a multi-processor system (June 13 2006) US Patent 7,062,583.

[11] Tuan, C.: Apparatus and method for hardware semaphore (June 22 2006) US Patent App. 11/116,972.

[12] Parson, D.: Resource management in a processor-based system using hardware queues (July 28 2005) US Patent App. 10/764,967.

[13] Hansen, H.E., Maroun, E.J., Kristensen, A.T., Marquart, J., Schoeberl, M.: 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). (Oct 2017) 1–6

[14] Strøm, T.B., Puffitsch, W., Schoeberl, M.: Hardware locks for a real-time java chip multiprocessor. Concurrency and Computation: Practice and Experience **29**(6) (2017)

[15] Schoeberl, M., Abbaspour, S., Akesson, B., Audsley, N., Capasso, R., Garside, J., Goossens, K., Goossens, S., Hansen, S., Heckmann, R., Hepp, S., Huber, B., Jordan, A., Kasapaki, E., Knoop, J., Li, Y., Prokesch, D., Puffitsch, W., Puschner, P., Rocha, A., Silva, C., Sparsø, J., Tocchi, A.: T-CREST: Time-predictable multi-core architecture for embedded systems. Journal of Systems Architecture **61**(9) (2015) 449–471

[16] Schoeberl, M., Pezzarossa, L., Sparsø, J.: A multicore processor for time-critical applications. IEEE Design & Test **35(2)** (2018) 38–47

[17] Kasapaki, E., Schoeberl, M., Sørensen, R.B., Müller, C.T., Goossens, K., Sparsø, J.: Argo: A real-time network-on-chip architecture with an efficient GALS implementation. IEEE Transactions on Very Large Scale Integration (VLSI) Systems **24** (2016) 479–492

[18] Schoeberl, M., Chong, D.V., Puffitsch, W., Sparsø, J.: A time-predictable memory network-on-chip. In: Proceedings of the 14th International Workshop on Worst-Case Execution Time Analysis (WCET 2014), Madrid, Spain (July 2014) 53–62

[19] Schoeberl, M., Schleuniger, P., Puffitsch, W., Brandner, F., Probst, C.W., Karlsson, S., Thorn, T.: 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), Grenoble, France (March 2011) 11–20

[20] Schoeberl, M., Puffitsch, W., Hepp, S., Huber, B., Prokesch, D.: Patmos: A time-predictable microprocessor. Real-Time Systems **54(2)** (Feb 2018) 389–423

[21] Bachrach, J., Vo, H., Richards, B., Lee, Y., Waterman, A., Avizienis, R., Wawrzynek, J., Asanovic, K.: Chisel: constructing hardware in a scala embedded language. In Groeneveld, P., Sciuto, D., Hassoun, S., eds.: The 49th Annual Design Automation Conference (DAC 2012), San Francisco, CA, USA, ACM (June 2012) 1216–1225

[22] Puschner, P., Kirner, R., Huber, B., Prokesch, D.: Compiling for time predictability. In Ortmeier, F., Daniel, P., eds.: Computer Safety, Reliability, and Security. Volume 7613 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg (2012) 382–391

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

[24] Heckmann, R., Ferdinand, C.: Worst-case execution time prediction by static program analysis. Technical report, AbsInt Angewandte Informatik GmbH [Online, last accessed November 2013].

[25] Hepp, S., Huber, B., Knoop, J., Prokesch, D., Puschner, P.P.: 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)

[26] Sørensen, R.B., Puffitsch, W., Schoeberl, M., Sparsø, J.: Message passing on a time-predictable multicore processor. In: Proceedings of the 17th IEEE Symposium on Real-time Distributed Computing (ISORC 2015), Auckland, New Zealand, IEEE (April 2015) 51–59

[27] Schoeberl, M., Brandner, F., Hepp, S., Puffitsch, W., Prokesch, D.: Patmos reference handbook. Technical report, Technical University of Denmark (2014)

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

[29] Intel Corporation: Quartus. http://dl.altera.com/15.0/?edition=web (2017)

[30] T-CREST Group: Patmos source. https://github.com/t-crest/patmos (2017)

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

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