# Time Analysable Synchronisation Techniques for Parallelised Hard Real-Time Applications

Mike Gerdes, Florian Kluge and Theo Ungerer
University of Augsburg, Germany
Email: {gerdes,kluge,ungerer}@informatik.uni-augsburg.de

Christine Rochange and Pascal Sainrat
University of Toulouse, France
Email: {rochange,sainrat}@irit.fr

***Abstract*—In this paper we present synchronisation techniques for hard real-time (HRT) capable execution of parallelised applications on embedded multi-core processors. We show how commonly used software synchronisation techniques can be implemented in a time analysable way based on the proposed hardware primitives. We choose to implement the hardware synchronisation primitives in the memory controller for two reasons. Firstly, we remove pessimism in the WCET analysis of parallelised HRT applications. Secondly, we enable that the implementation of synchronisation techniques is mostly independent of the chosen instruction set architecture (ISA) which allows to use the existing ISAs without enhancements. We analyse the presented synchronisation techniques with the static worst-case execution time (WCET) analysis tool OTAWA. In summary, our specifically engineered synchronisation techniques yield a tremendous gain on the WCET of parallelised HRT applications.**

## I. Introduction

For a long time research in parallel applications and architectures was bound to the domain of high-performance computing. With the upcoming of multi-core processors, parallelisation became also important in other domains, namely desktop end-user systems and embedded systems as well. However, embedded systems have different needs and must fulfil other requirements than high-performance systems. Today's HRT applications in the automotive, avionic or machinery industry are executed on single-core processors. The new trend of using multi-cores in safety-critical domains sparks off research on running HRT tasks in parallel with other tasks to execute mixed-critical application workloads. Our research goes even one step further: we target multi-core execution of parallelised HRT tasks without sacrificing timing guarantees.

In this paper, we focus on the timing predictability of synchronisation in parallelised HRT applications using the static WCET analysis tool OTAWA [1]. The contributions of this paper are as follows: we show that specifically engineered software synchronisation techniques, busy-waiting as well as blocking synchronisation methods, are timing analysable. We assess these synchronisation primitives and their hardware respectively software implementation with respect to their impact on the WCET, and the possible gain for the WCET of parallelised HRT applications.The presented synchronisation techniques can be implemented independently of the ISA, as the logic for the hardware synchronisation primitives is nested in the memory controller.

978-3-9810801-8-6/DATE12/

Moreover, we present WCET analysis results for two different parallelised applications: a data parallel application, namely matrix multiplication, and a multi-stage producer-consumer application, that is IFFT (Integer Fast Fourier Transformation). Both applications have been implemented with different synchronisation primitives to depict their impact on the application's WCET.

In Section II we present related work for real-time synchronisation and WCET analysis of parallel HRT applications. Section III illuminates the implemented HRT capable hardware and software synchronisation techniques. In Section IV we show by a static WCET analysis the gain in the WCET for parallelised HRT applications achieved with the proposed HRT capable hardware and software synchronisation techniques.

## II. Related Work

Monchiero et al. [2] present an augmented global memory controller, the Synchronisation-operation Buffer (SB), to reduce contention for busy-waiting synchronisation primitives in future mobile systems with complex Network-on-Chips (NoCs). Their main focus is on reducing contention, and therefore enabling an efficient use of busy-waiting synchronisations like spin locks. However, the goal of synchronisation buffers is to decrease the average-case execution time by speeding up slow synchronisation primitives, while also enabling a fine-grained synchronisation. Contrarily, we focus with our augmented memory controller on implementing WCET-efficient HRT capable synchronisation primitives, while also reducing WCET overestimation.

Anderson [3] introduces queuing spin locks using unique IDs in shared memory multiprocessors with cache coherence. The approach is similar to ticket locks established in [4] by Mellor-Crummey and Scott despite that the ticket locks are implemented with the fetch-and-increment (F&I) primitive. We also implemented ticket locks with the F&I primitive, but with focus on assuring fairness between HRT threads without requiring a specific bus arbitration. In [5] Molesky et al. present an arbitration for a bus, the *Deferred Bus* theorem, which is the baseline for the bus arbitration we are using to assure fairness of spin locks [6]. Molesky et al. show that their *Deferred Bus* enables synchronisation mechanisms for mutual exclusion with linear waiting, and bounded semaphores for predictable synchronisation in multiprocessor real-time systems. Though, the use of ticket locks is more flexible

concerning the bus arbitration, which is also the reason why ticket locks are used in the Linux Kernel since version 2.6.25 (January 2008) as a fair spin lock mechanism.

Further spin lock implementations, which also allow timing predictability in shared memory multiprocessors, like e.g. MCS locks [4] or CLH locks [7], require a cache coherence protocol or, for MCS locks, a complex allocation and pointer arithmetic in local memory on non-cache-coherent systems. However, we do not use a cache coherence protocol, because it would need complex hardware and/or software solutions, which hinder a WCET analysis or even render it impossible [8]. Also, queued spin locks are introduced to reduce the overhead and contention of busy-waiting synchronisation primitives to improve the average-case execution time. But in our case—for a tight WCET analysis—we focus on reducing the WCET overestimation introduced from (slower) synchronisation primitive accesses on shared resources.

For parallelised HRT applications, it is not possible to duplicate all shared resources, because doing so would lose the mandatory requirement of communication between the threads of a parallelised application. Hence, it is essential to assert time bounding access to shared resources, as well as an upper bound to waiting time introduced by the execution of synchronisation primitives. Only very few publications have been targeting WCET analysis of parallel HRT applications so far. Gustavsson et al. [9] present the chain of a possible static WCET analysis of multi-core architectures. They use timed automata to model the various components of a multi-core architecture, including private and shared caches, but also software-level shared resources like spin locks. The WCET of the parallel program is then derived by model checking.

In [10], the basic principles of analysing the worst-case waiting times in synchronisation functions are introduced. The idea is to determine all the paths on which a thread holds any system-level or application-level synchronisation variable. Their estimated WCETs are combined to compute the worst-case waiting times at synchronisation points. In [11] we present first results on the static WCET analysis of an industrial, parallel HRT application. They consider a limited set of synchronisation functions based on test-and-set. The grain of the parallelism in their application is coarser than in the programs we consider here so that the cost of synchronisations compared to the computation time is relatively smaller.

## III. Hard Real-Time Capable Synchronisation Techniques

We use a WCET model of a HRT capable multi-core processor [12] for the WCET analysis in this paper. The modelled multi-core processor features a configurable number of HRT capable cores. We consider in this paper that only HRT threads are executed concurrently, one per core. Also, the memory controller and interconnect cannot isolate concurrent accesses of different cores. Besides, a partitioning of global memory would impede the use of a global address space, and hence narrow down the programmability for users. Therefore, we have chosen to allow shared resources. Interferences are handled by an upper bounding of accesses to shared resources like a real-time capable bus [6] as interconnect to memory and cores, as well as a real-time capable memory controller. As local memories we use scratchpad memories for each core, namely a data scratchpad (DSP) and a dynamic instruction scratchpad (D-ISP) [13], but no caches for the HRT threads. However, we allow caches to be used by non-hard real-time (NHRT) threads. Fig. 1 depicts an overview of our multi-core processor [12]. It is binary compatible with the Infineon TriCore architecture, which is widely used in the automotive and construction machinery domains.

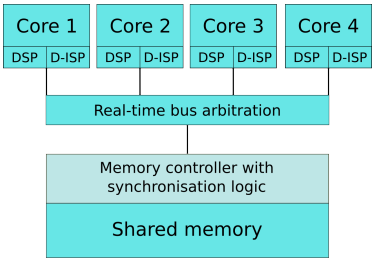


Fig. 1. Overview of our multi-core processor, stressing the embedded hardware synchronisation primitives in the real-time memory controller.

### A. Hardware Synchronisation Support

To support synchronisation methods in parallel applications, the hardware needs to provide atomic operations. We focus on well known read-modify-write (RMW) operations [14] like test-and-set (TAS) and fetch-and-increment/decrement (F&I/F&D) and on their use in a HRT capable multi-core processor. We decided to implement the logic for the RMW operations in the memory controller (see Fig. 1) for two reasons: Firstly, with this approach we can leave the TriCore ISA untouched. For our atomic operations we reuse the *swap* instruction of the TriCore ISA and the needed logic is placed in the memory controller. The *swap* operation is usually used in the single-core TriCore to atomically swap a data register value with a memory word. As we do not need the data which is usually swapped by the *swap* instruction for our hardware synchronisation primitives we use it to encode the corresponding RMW operations. In the memory controller we can then decode the data value to identify a TAS, a F&I, or a F&D operation. The *swap* instruction is reused for simplification reasons in this paper, however, it should be assured that the reused instructions are not generated by the compiler. Hence, the advantage of this approach is that we do not lose the generality of our approach as it is possible to reuse instructions of other ISAs for RMW operations in the memory controller as well.

Secondly, this approach allows assuring atomicity while also achieving a tight time bound when the logic is embedded in the memory controller. Other possibilities to achieve atomicity, e.g. by locking the interconnect are not advisable for a tight WCET analysis [10]. Contrarily, with our implementation of synchronisation logic in the memory controller, we reduce this overhead for concurrent, atomic accesses in HRT systems.

*1) Test-and-Set (TAS):* The test-and-set primitive is implemented in the memory controller by first loading the lock value $val \in \{0, 1\}$ from memory address *addr*. Subsequently, the memory controller stores a '1' at *addr*. These operations are executed atomically. Then, the loaded lock value *val* is returned to the thread issuing a test-and-set, where $val = 0$ signals that the lock is free and the thread got the lock, whereas $val = 1$ signals that the lock is held by another thread.

*2) Fetch-and-Increment/Decrement (F&I/F&D):* F&I/F&D operations need to be initialised with a software construct in the RTOS as follows. The upper two bytes of the four byte memory word are used to store an upper limit value *lim*, whereas the lower two bytes are used for the actual *count*. At runtime, if e.g. the memory controller recognises an F&I instruction, it loads the four byte word from memory. Then, it sends the lower two bytes, the actual *count*, as fetched value to the core issuing the F&I instruction. After that it checks if the lower two bytes are equal to the initialised upper limit *lim*, i.e. if the counter *count* would overflow *lim* when being incremented. If this is not the case, *count* is incremented and stored back together with *lim*. The same holds for F&D, but with $lim = 0$ when decrementing. Otherwise, if F&I increments so that $count > lim$ (or F&D decrements so that $count < 0$) the lower two bytes of the store back value *count* are set to *lim* for F&I (or respectively $lim = 0$ for F&D). Also, as we need to manipulate the loaded value and store the incremented or decremented value back to memory for F&I/F&D, the latency of a F&I/F&D is higher than for a TAS. In detail, one extra cycle in the memory controller is needed to increment/decrement the loaded value.

### B. Software Synchronisation Techniques

Software synchronisation techniques presented in this section are part of a HRT capable Real-time Operating System (RTOS) extending the RTOS presented in [10]. In the following we introduce a FIFO buffer implemented with the F&I primitive, and we describe the implemented synchronisation constructs (see Table I): a *mutex lock* implementation according to POSIX, using the TAS hardware primitive, as well as *ticket locks*, *semaphores*, and *software barriers* that are using the F&I/F&D hardware synchronisation primitives.

Synchronisation methods can be separated into two different categories, that is either blocking or busy-waiting. Busy-waiting means that the thread executes the synchronisation function as long as the lock is not gained, whereas blocking means that a thread tries to get the lock and is suspended if not succeeding. Busy-waiting algorithms can consume a lot of processor time and add contention on the memory system (and other shared resources which need to be taken into account when accessing a synchronisation variable). With suspension it is possible to avoid unnecessary accesses, but suspending and waking up may take longer than busy-waiting.

Synchronisation primitives need to fulfil the following two requirements for a safe implementation of critical sections: *mutual exclusion*, and *progress*. In addition, some further restrictions for synchronisation primitives are needed to fulfil the special requirements for being HRT capable. Concerning *progress*, we need to achieve fairness between threads that are competing to access a critical section. In the following we show that not all implementations, e.g. barriers commonly implemented with conditionals, or mutex locks with TAS, are strictly fair (see Sections III-B1, and III-B4). Hence, fair progress between competing threads, achieving as low as possible upper bounded waiting times for critical sections, is needed to enable a tight WCET analysis.

TABLE I
OVERVIEW OF THE IMPLEMENTED SOFTWARE SYNCHRONISATION TECHNIQUES, THE USED HARDWARE SYNCHRONISATION PRIMITIVES, AND THE CATEGORISATION AS EITHER BUSY-WAITING OR BLOCKING.

| SW Synchronisation | HW Primitive | Strategy |
|---|---|---|
| Mutex lock (Section III-B1) | TAS | blocking |
| Ticket lock (Section III-B2) | F&I | busy-waiting |
| Semaphore (Section III-B3) | F&I/F&D | blocking |
| Software barrier (Section III-B4) | F&I | blocking |

*FIFO Buffer:* The F&I primitive can be used to implement a shared FIFO buffer. In that case, F&I is used to increment the index for the insert/remove operation in the FIFO buffer. The only requirement is that the upper limit of the FIFO buffer needs to be known a-priori, that is the upper bound needs to be set for the F&I primitive. With our implementation of F&I, it is then easily possible to manage a FIFO buffer applying the implemented cyclic counting of F&I. In detail, we can use F&I to increment the index atomically, and, if the last index is reached, the next fetched index is '0' again, as the upper limit for F&I is reached. This allows to implement a FIFO ordered waiting list for HRT threads with two indexes. Overwriting of HRT threads in the list is not possible, as the number of threads is known at design time. The advantage of this approach is that, contrarily to a fully software managed linked list, we do not need to secure the access to the list with locks. Therefore, we save computation and waiting time in the WCET. We use this FIFO buffer with F&I to manage the waiting list in the semaphore implementation in Section III-B3. It is also applicable for other blocking software synchronisations. However, for comparison, the waiting list for mutex locks has been implemented with a linked list.

*1) Mutex Lock:* The mutex lock is a blocking synchronisation function implemented according to POSIX. The RTOS uses a TAS spin lock for critical sections inside the mutex lock. We can assure fairness and progress of the used spin lock by specific arbitration strategies in the real-time bus [6]. Additionally, a linked list contains all waiting HRT threads in FIFO ordering to guarantee fairness, that is the longest waiting thread wakes up and acquires the mutex lock when the previous holding HRT thread releases it.

For HRT capability, the POSIX mutex lock implementation has been refined to ensure that the woken HRT thread acquires the mutex lock, and not other HRT threads which are not suspended yet and that are competing for the mutex lock. Therefore, the active mutex lock holder does not release the

spin lock after waking up a waiting HRT thread. Instead, if a HRT thread, other than the woken HRT thread, tries to acquire the spin lock inside the mutex lock, it finds it still locked and suspends. The woken HRT thread does not check the spin lock again, and hence acquires the lock. If we take the FIFO ordering in the waiting list into account as well, we can assure a fair handling of HRT threads for our mutex lock implementation using a real-time aware memory bus arbitration.

*2) Ticket Lock:* The semantic of *ticket locks* [4], based on Lamport's bakery algorithm, is quite easy. Each HRT thread gets a unique *ticket_id* when trying to access a critical region. HRT threads are allowed to enter the critical region when their *ticket_id* matches the current value of *now_served*. A HRT thread leaving the critical section increments *now_served*, and the HRT thread with the appropriate *ticket_id* can then enter the critical section. The atomic incrementing of *ticket id* and *now_served id* is done with the F&I primitive in the memory controller. Thus, ticket locks implement a busy-waiting spin lock, which is, contrary to simple TAS spin locks, fair, independently of the arbitration strategy in a bus-based memory interconnect.

*3) Semaphores:* Our semaphores are implemented according to POSIX. The implementation uses the F&I/F&D hardware primitives. In POSIX, the original *P-operation* and *V-operation* from Dijkstra are being referred to as *wait()* and *post()*, which we also use in the following. The *wait()* method first needs to check if the resource secured by the semaphore is free. This is done by using F&D on the semaphore counter. A resource is successfully acquired if a value $> 0$ is fetched, otherwise the thread enters a waiting list and suspends. Inserting and removing a thread from the waiting list is secured in a critical section using F&D with lower limit 0 (see Section III-A2) instead of TAS as for mutex locks. This critical section is needed as otherwise a thread calling *post()* might conflict with a thread that is calling *wait()* and trying to enter the waiting list. In that case, it is crucial that the thread executing *post()* wakes a thread from the waiting list. Otherwise, if no thread is already waiting, the thread executing *post()* increments the semaphore counter without waking a thread. Instead, the thread executing *wait()* needs to check if the resource was released while the thread was competing to enter the waiting list. Therefore, we added an additional F&D primitive inside the critical section for the waiting list. The suspended threads are managed in a waiting list that uses the FIFO buffer implementation with the cyclic F&I as described in Section III-A2. Please also note that a binary semaphore has the same functionality as a mutex lock with the difference that the semaphore implementation does not use the concept of an owner contrarily to the mutex lock implementation (see WCET comparison in Section IV).

*4) Software Barriers:* Barriers are a very useful construct to synchronise starting or re-starting of threads at a specific point, e.g. to organise parallel progress in different phases of an application. But, typically implemented barriers with conditionals are not HRT capable respectively lead to an overestimation in the WCET. This is because it is possible that threads exiting a barrier compete with threads that are trying to reenter the barrier, if the code between barriers is too short. In detail, it is possible that a reentering thread acquires the mutex lock to enter the barrier again, before a thread that is trying to leave the barrier acquires the mutex lock for the conditional that is needed to leave the barrier. One solution to this is to use subbarriers introduced in [15], and e.g. later used in the Legion OpenSPARC simulator. In that implementation the competition between threads at a barrier in different phases of the application is solved by switching from one subbarrier to another when all needed threads have reached the barrier. Thus, leaving threads are exiting one subbarrier whereas other threads, which again execute the barrier code, enter the other.

Another solution is to implement barriers with the F&I/F&D primitives and use a waiting list for suspended threads at the barrier as described for the FIFO buffer with F&I in Section III-A2. Using F&I/F&D for barriers is a well known concept, and overcomes the WCET overestimation of the implementation of barriers with conditionals. All threads that enter a barrier are suspended, as long as the needed number of threads is not reached. The waiting list for threads is organised as for the blocking semaphores with a FIFO buffer managed with F&I. When the last thread enters the barrier, it wakes all waiting threads and all threads continue. Also, threads that try to reenter the barrier in the next iteration are busy-waiting until the last thread of the actual iteration leaves the barrier.

## IV. WCET Analysis - Evaluation

### A. Methodology

Approaches to estimate the WCET of critical tasks have received much attention in the last fifteen years [16]. Those based on static analysis techniques aim at determining guaranteed upper bounds on the real WCET, taking into account the specificities of the target hardware.

In this work, we use a static WCET analysis tool that implements state-of-the-art algorithms for WCET analysis [1]. It supports our target multi-core architecture and accounts for possible contentions on the shared bus and memory controller by considering worst-case latencies. In this architecture, predictability is enforced by processing the requests to the main memory in a round-robin fashion. As a result, the worst-case latency experienced by one core occurs when all the other cores have pending requests and are served before it, and when these requests exhibit the longest processing time by the memory system. This is illustrated in Fig. 2 that assumes a four-core configuration. For the sake of simplicity, we consider that the latency of read-modify-write access ($t_{RMW}$) is twice the latency of a simple load or store ($t_{mem}$). In the worst-case, core $C_0$ will be served after the other cores that might execute RMW operations. This must be assumed for safety although synchronisation instructions are relatively unfrequent in a program, and it contributes to WCET overestimation. As a result, the worst-case latency of an access to the memory in

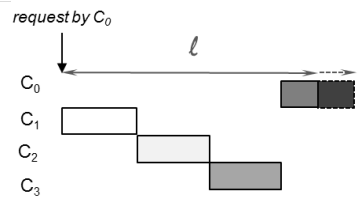


Fig. 2. Worst-case memory latencies

an N-core architecture is computed as:

$$\ell = (N-1) \times t_{RMW} + (t_{mem} \ or \ t_{RMW})$$

In our target architecture, $t_{RMW}$ is 10 cycles (an additional cycle is needed to modify the synchronisation variable for a F&I/F&D instruction) and $t_{mem}$ is 5 cycles for a load and 4 cycles for a store. These values are typical for embedded SDRAM operating with up to 200 MHz and we derived those values with an in-house build FPGA simulator of our multi-core processor [12]. Table II depicts the worst-case latencies for loads, stores, and for the two RMW operations TAS, and F&I/F&D for the augmented memory controller.

Considering worst-case latencies is safe only for processors that are free from timing anomalies [17]. Otherwise, all the possible latency values should be considered.

### B. WCET Analysis of Parallel Applications

In this paper, we focus on data-parallel applications where all threads execute the same code, each on another part of the data. Our target architecture and system software include support to start all the threads simultaneously so that the WCET of the application is the WCET of the longest running thread. Now, the difficulty is to account for the waiting times at any synchronisation point. In [10], we show how these waiting times can be analysed for a small set of synchronisation primitives. In this work, we consider a wider set of primitives and we exploit these results in the context of full applications. In brief, computing the waiting time linked to a lock/semaphore synchronisation function consists in determining the worst-case time during which the synchronisation variable could be held by another thread. This is done by analysing the WCET of all the possible paths from any point where the variable is locked to any point where it is released. As far as barriers are concerned, the longest thread is, by definition, the one that reaches the barrier last. Then this thread will not wait at this point. The approach is further detailed in [11].

To analyse the difference in the worst-case execution of mutex locks respectively binary semaphores or ticket locks we use a dynamically partitioned version of a matrix multiplication (`matmul`). Therefore, the matrix multiplication $A = B \cdot C$ has been partitioned into working units consisting of a scalar multiplication of $A_{ij} = B_i \cdot C_j$. Each working unit is computed by one thread, and getting the next working unit is secured by either a mutex lock or alternatively a binary semaphore. The Integer Fast-Fourier-Transformation (IFFT) application has been parallelised based on an integer version of the iterative radix-2 algorithm, which is working *in place* and stores all samples in an array. In our parallelised version, for $N$ samples the pairwise combination and rearranging in each of the $k = log_2(N)$ stages is done in parallel. Each thread combines independently a pair of samples, and, as in the above version of the `matmul` application, the fetching of the next working unit is secured using a mutex lock, a ticket lock, or respectively a binary semaphore. After each stage, we use a barrier to assure that all threads finished their computation for the current stage before beginning to compute the results in the next stage. The barriers have been implemented either using the F&I barriers, or accordingly the subbarrier implementation.

TABLE II
WORST-CASE LATENCIES (# CYCLES)

| load | store | test-and-set (TAS) | F&I/F&D |
|---|---|---|---|
| 38 | 37 | 43 | 44 |

### C. Results and Discussion

In this section we present WCET estimates for our two parallelised applications, matmul and IFFT. Both have been implemented with three kinds of primitives to guard critical sections: mutex locks (Section III-B1), binary blocking semaphores (Section III-B3) and ticket locks (Section III-B2). In addition, IFFT includes synchronisation barriers and was compiled with barriers implemented using subbarriers and conditionals [15] or F&I instructions (Section III-B4). Results of the WCET analysis are given in Table III. Note that the discussion below refers to WCET estimates only, since determining *real* WCETs is not feasible.

*1) Locks and Semaphores:* As mentioned in Section III-B3, a blocking binary semaphore is very similar to a mutex lock. In our implementation, they differ by the hardware synchronisation primitive they rely on: mutex locks make use of TAS while blocking semaphores are based on F&I/F&D. They mainly differ in the way they implement the list of threads waiting to enter the critical section: the use of F&I/F&D instructions makes it possible to implement hardware-managed FIFO queues that perform noticeably better than software linked lists. Results in Table III show that the worst-case performance of the blocking semaphore is noticeably better than that of the mutex lock: the WCET is improved by 16.2% to 29.4%.

Both mutex locks and blocking semaphores suspend a thread that tries to enter a locked critical region. This solution is favoured in high-performance systems because it reduces the traffic on the bus and thus the number of contentions. However, when computing the WCET of one thread, it is not possible to determine how many other threads will be suspended at any point of the program. So suspension does not help to reduce pessimism on the bus/memory latencies since it must always be assumed that all the threads can be active. The consequence is that instead of improving the WCET, thread suspension tends to degrade it because the code required to suspend a thread is generally longer than the code for busy waiting.

TABLE III
WCET ESTIMATES (# CYCLES)

| | mutex | semaphore | ticket lock |
|---|---|---|---|
| matmul | 1,347,342 | 1,041,525 | 938,312 |
| IFFT (conditional subbarriers) | 233,921 | 196,085 | 183,936 |
| IFFT (F&I barriers) | 156,664 | 110,529 | 102,252 |

So, even if thread suspension contributes to free computing resources and gives opportunities to run NHRT threads, we do not recommend it if tasks are submitted to tight timing constraints and if their WCET must be as low as possible.

As an alternative, we suggest the use of ticket locks that implement a busy waiting strategy with F&I/F&D instructions, as introduced in Section III-B2. Contrarily to the average execution time, the WCET of a thread is not impacted by the other threads' busy waiting because worst-case latencies assume that all threads permanently issue requests to the main memory. Compared to mutex locks or blocking semaphores, the code of acquiring respectively releasing a ticket lock is far shorter because it does not include inserting and suspending (respectively extracting) a thread in (from) a FIFO queue. The result is an improved WCET, as shown in Table III. The gain ranges from 6.2% to 9.9% compared to the implementation with blocking semaphores.

*2) Barriers:* Table III also highlights a considerable WCET improvement when using synchronisation barriers implemented with F&I/F&D instructions instead of primitives on conditions (*wait, broadcast*) and subbarriers, as described in Section III-B4. Again the possibility of implementing FIFO queues in a very efficient way with F&I/F&D instructions makes the difference. Also, the F&I/F&D implementation does not show the problem of threads reentering the barrier while other threads have not left it yet after the previous round, hence subbarriers are not required. The gain in the WCET ranges from 33,0% up to 44,4%.

## V. SUMMARY

We foresee that performance requirements of safety-critical systems will soon motivate the design of parallel applications running on multi-cores. However, this will require predictable hardware and software support, in particular to implement safe and efficient inter-thread synchronisation. Here, we investigate solutions for such a support in the context of HRT applications.

Synchronisation primitives used to implement critical sections must guarantee mutual exclusion and progress for all the threads. To be HRT-capable, they must in addition ensure fairness which is required to compute upper bounds on waiting times. We propose three solutions (mutex locks, blocking binary semaphores and ticket locks) that exhibit such properties. They are implemented using TAS and F&I/F&D primitives processed within the memory controller. We show how F&I/F&D primitives can be used to control a FIFO buffer in a very efficient way. In addition, we consider synchronisation barriers implemented in two variations: one based on conditionals and subbarriers and the other one based on F&I operations.

To compare these primitives, we study two parallelised applications: matmul and IFFT. We consider an HRT-capable multi-core architecture and we estimate the WCET of the applications running on this architecture using the static WCET tool OTAWA. We explicate how worst-case latencies for any access to the main memory should be computed. Experimental results show that the primitives implemented using the F&I/F&D instructions (semaphores and F&I barriers) perform noticeably better than those that use TAS as soon as we consider the WCET. For example, the IFFT application has its WCET improved by 47% with binary semaphores and F&I barriers against mutex locks and conditional-based subbarriers. Additionally, we show that ticket locks based on a busy waiting strategy improve the WCETs further, by 24.7% to 39.9%.

As future work, we plan to introduce a technique in the memory controller to further reduce the WCET overestimation introduced by (slow) synchronisation accesses on faster memory accesses.

## REFERENCES


[1] C. Ballabriga, H. Cassé, C. Rochange, and P. Sainrat, "OTAWA: An Open Toolbox for Adaptive WCET Analysis," in *Software Technologies for Embedded and Ubiquitous Systems*, ser. Lecture Notes in Computer Science, vol. 6399, 2011, pp. 35–46.

[2] M. Monchiero et al., "An Efficient Synchronization Technique for Multiprocessor Systems on-Chip," in *Proc. of MEDEA*, 2005, pp. 33–40.

[3] T. E. Anderson, "The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors," *IEEE Trans. Parallel Distrib. Syst.*, vol. 1, pp. 6–16, January 1990.

[4] J. M. Mellor-Crummey and M. L. Scott, "Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors," *ACM Trans. Comput. Syst.*, vol. 9, pp. 21–65, February 1991.

[5] L. D. Molesky, C. Shen, and G. Zlokapa, "Predictable Synchronization Mechanisms for Multiprocessor Real-Time Systems," *Real-Time Systems*, vol. 2, pp. 163–180, 1990.

[6] M. Paolieri et al., "Hardware Support for WCET Analysis of Hard Real-Time Multicore Systems," in *Proc. 36th Intl' Symposium on Computer Architecture (ISCA09)*, 2009, pp. 57–68.

[7] T. Craig, "Queuing Spin Lock Algorithms to Support Timing Predictability," in *Real-Time Systems Symposium 1993*, Dec. 1993, pp. 148 –157.

[8] M. Schoeberl and P. Puschner, "Is Chip-Multiprocessing the End of Real-Time Scheduling?" in *Proc. of the 9th Int'l Workshop on WCET Analysis (WCET 2009)*, 2009.

[9] A. Gustavsson et al., "Towards WCET Analysis of Multicore Architectures using UPPAAL," in *Proc. of the 10th Int'l Workshop on WCET Analysis (WCET 2010)*, July 2010, pp. 103–113.

[10] J. Wolf et al., "RTOS Support for Parallel Execution of Hard Real-Time Applications on the MERASA Multi-core Processor," *IEEE Int'l Symp. on Object-Oriented Real-Time Dist. Comp. (ISORC)*, pp. 193–201, 2010.

[11] C. Rochange et al., "WCET Analysis of a Parallel 3D Multigrid Solver Executed on the MERASA Multi-Core," in *10th Int'l Workshop on WCET Analysis (WCET 2010)*, July 2010, pp. 92–102.

[12] T. Ungerer et al., "MERASA: Multicore Execution of HRT Applications Supporting Analyzability," *IEEE Micro*, vol. 30, pp. 66–75, 2010.

[13] S. Metzlaff, I. Guliashvili, S. Uhrig, and T. Ungerer, "A Dynamic Instruction Scratchpad Memory for Embedded Processors Managed by Hardware," *24th Int'l Conf. on ARCS*, pp. 122–134, February 2011.

[14] C. P. Kruskal, L. Rudolph, and M. Snir, "Efficient Synchronization of Multiprocessors with Shared Memory," *ACM Trans. Program. Lang. Syst.*, vol. 10, pp. 579–601, October 1988.

[15] R. Marejka, "A Barrier for Threads," *SunOpsis - The Solaris 2.0 Migration Support Centre Newsletter*, vol. Vol. 4, no. 1, November 1994.

[16] R. Wilhelm et al., "The Worst-case Execution Time Problem—Overview of Methods and Survey of Tools," *ACM Trans. on Embedded Computing Systems (TECS)*, vol. 7, no. 3, 2008.

[17] J. Reineke and R. Sen, "Sound and Efficient WCET Analysis in the Presence of Timing Anomalies," in *9th Int'l Workshop on WCET Analysis (WCET 2009)*, 2009.