RTOS Support for Parallel Execution of Hard Real-Time Applications on the MERASA Multi-Core Processor

Julian Wolf*, Mike Gerdes*, Florian Kluge*, Sascha Uhrig*, Jörg Mische*, Stefan Metzlaff*,
Christine Rochange†, Hugues Cassé†, Pascal Sainrat†, and Theo Ungerer*
*University of Augsburg, Germany - {wolf, gerdes, kluge, uhrig, mische, metzlaff, ungerer}@informatik.uni-augsburg.de
†University of Toulouse/IRIT, France - {rochange, casse, sainrat}@irit.fr

Abstract—**Multi-cores are the contemporary solution to satisfy high performance and low energy demands in general and embedded computing domains. However, currently available multi-cores are not feasible to be used in safety-critical environments with hard real-time constraints. Hard real-time tasks running on different cores must be executed in isolation or their interferences must be time-bounded. Thus, new requirements also arise for a real-time operating system (RTOS), in particular if the parallel execution of hard real-time applications should be supported.**

In this paper we focus on the MERASA[1] system software as an RTOS developed on top of the MERASA multi-core processor. The MERASA system software fulfils the requirements for time-bounded execution of parallel hard real-time tasks. In particular we focus on thread control with synchronisation mechanisms, memory management and resource management requirements. Our evaluations show that all system software functions are time-bounded by a worst-case execution time (WCET) analysis.

I. Introduction

In the automotive domain, driver assistance systems like steer-by-wire and brake-by-wire, or safety-relevant systems such as automatic emergency-braking triggered by collision avoidance techniques could evaluate more sensor signals and master more complex situations if higher performance is provided by future control units. Also, motor injection could be optimised to reduce fuel consumption and to minimise emissions by better processor efficiency. Similar advantages arise in other embedded safety-critical domains as e.g. aerospace, power plants, and machinery if a higher performance can be combined with guarantees of timing constraints. However, the high energy requirements of complex general-purpose processors do not satisfy the low-power constraints and the strict cost limitations of common embedded systems. In embedded environments, higher performance will not arise from increasing CPU frequencies, but from augmenting the number of computation units. The most energy and weight efficient way to achieve this is through multi-core processors.

[1]**M**ulti-Core **E**xecution of Hard **R**eal-Time **A**pplications **S**upporting **A**nalysability. This research is funded by the European Community's Seventh Framework Programme under Grant Agreement n° 216415.

Multi-cores provide a better performance per watt ratio than single-core processors. A number of tasks can be scheduled on a single multi-core processor in order to maximise the hardware utilisation, while cost, size, weight and power requirements are reduced compared to a multi-processor solution.

Concerning e.g. the automotive and aerospace domain, most deployed systems are hard real-time systems that require absolute time-predictability. Missing a deadline may cause system failures with catastrophic consequences. Thus, the primary goal is not to optimise the *Average Execution Time*, but to determine the *Worst-Case Execution Time* (WCET) [1] and to provide analysability. A main target of MERASA is necessary to minimise the gap between real and computed WCET, i.e. to compute a tight upper bound of the WCET.

While multi-core processors offer several benefits to embedded systems, their WCET analysis is a much higher challenge than that of single-cores. Due to a dynamic interaction of effects between different threads in multi-core processors, the WCET of a program is quite difficult or no more computable, or the computed WCET significantly differs from the measured execution times. Non-computability of WCET would lead to unsafe systems, overestimation requires to employ more expensive hardware than really necessary.

The MERASA architecture was developed as a WCET-analysable embedded multi-core. Necessarily, cores should execute hard real-time (HRT) threads in isolation and restrict arbitration to shared resources, as e.g. common memory in multi-cores. The hardware must assure a time-bounded access to that shared memory by applying techniques for a real-time capable bus and memory controller. In order to allow mixed application execution of hard and non real-time (NRT) threads each MERASA core supports simultaneous multi-threading (SMT) and provides the possibility to run a hard real-time thread simultaneously in concert with additional non real-time threads.

The MERASA system software provides the fundamental building blocks of a real-time operating system (RTOS) on top of the MERASA multi-core processor like synchronisation functions and memory management facilities for application software. Similar demands as for the processor architecture arise for the RTOS: all functionalities must allow a time-predictable execution of HRT threads by a full isolation of threads. If common resources are accessed, a time-bounded handling must be guaranteed.

In particular, the main contributions of this paper are:

1) We present a WCET-safe system software as a solid and efficient baseline of an RTOS, in order to execute multithreaded hard real-time applications. We focus on the most innovative aspects of the system software with respect to timing analysability and multi-core execution. A detailed description is given in the Technical Report [2].
2) We introduce synchronisation functions and show their efficient implementation on an analysable hard real-time capable multi-core processor to support execution of hard real-time threads in parallel potentially combined with additional non real-time threads.
3) The proposed system software functions are evaluated by a static WCET analysis in order to determine timing boundaries.

This paper is organised as follows: Section II summarises related work in the field of embedded real-time systems. Requirements arising for a multi-core real-time system are described in section III. Subsequently, section IV gives an overview of our MERASA multi-core architecture. In section V we present our proposed system software developed to accomplish the needed requirements. Section VI shows an evaluation of the proposals and section VII concludes the paper.

II. Related Work

In the field of embedded real-time systems, there are different related approaches for predictable architectures.

Schoeberl provides a time-predictable computer architecture for real-time systems that supports WCET analysis, evaluating the concepts by a real-time Java processor, called JOP [3]. For performance enhancements, he proposes a Chip-multiprocessing (CMP) system, featuring time-sliced arbitration of the main memory access in order to enable analysability. As in this CMP approach each application task has its own CPU for execution, task scheduling can be completely avoided. However, only multi-programmed but no multithreaded workloads can be executed. In [4] Schoeberl and Puschner additionally discuss a single-path programming style combined with CMP systems. The single-path approach includes a conversion, which eliminates all input-dependent control flow decisions in order to get straight-line predicated code. In that case, the WCET can be measured in an easy way. On the other hand, this programming paradigm is quite uncommon and restrictive.

To find a trade-off between predictability and processor throughput, Whitham introduced the MCGREP architecture [5], which enhances a sequential processor with dynamically reconfigurable logic. By this, a predictable processor increases performance through the use of application specific microprograms directing the operation of the reconfigurable logic.

Another project reconciling efficiency with predictability is called PREDATOR [6]. Its architectural approach faces analysable caches, a compiler-controlled memory management and simple cores with analysable behaviour. Predictable kernel mechanisms are used to cover the main challenge in the field of OS, i.e. the interference-caused variability in task execution times.

In order to execute multiple threads in parallel supporting a predictable architecture, Edwards and Lee developed the concept of Precision Timed (PRET) Machines [7]. In this approach, a precise timing without a loss of throughput is guaranteed, while also a simplified WCET analysis remains possible. This is enabled by a thread-interleaved pipeline, i.e. each stage of the pipeline feeds from a new thread. However, concerning application side and RTOS the approach is completely different from ours. PRET-C [8] extends the C language with synchronous constructs for expressing logical time, preemption and concurrency. So, a need for any RTOS vanishes, but the challenge for application programmers increases. Especially concerning parallel programming, an RTOS supporting commonly-used standards like POSIX [9] seems to be preferable.

In the field of RTOS, there exists a large number of commercial solutions. Some popular examples are the QNX Neutrino [10] or the ENEA OSE [11], which recently added support for multi-core processors. However, our approach is additionally focused towards timing analysability. Moreover, our solutions work in concert with the MERASA hardware scheduler (see section IV) guaranteeing an isolated execution of HRT threads, so the challenge of our RTOS is focused on the provision of a solid and efficiently working scheduling interface fulfilling real-time constraints.

III. Requirements

Regarding an embedded multi-core environment which enables a real-time execution of applications, it is first necessary to state some minimum requirements that have to be fulfilled and define the basic properties. The details of these requirements are twofold: firstly concerning the system software in order to guarantee a correct execution and to provide an interface for easy programming from application side, secondly affecting the hardware side as a baseline for the time-predictable execution.

A. Functional Requirements for the System Software

In general, the system software provides an interface to access system resources like memory, I/O devices and to manage the execution of tasks. As our objective is the development of a system software for embedded systems with multithreaded and multi-core hardware, we need a combination of different fields of requirements. We first take a look at the concept of embedded real-time systems and then add aspects regarding SMT and multi-core systems.

1) Concept of Embedded Real-Time Systems: Embedded systems are mostly not recognisable as computers, instead they are hidden inside cars, aeroplanes or everyday objects surrounding and helping us in our lives. The characteristical operation of embedded systems is limited by computer memory, processing power and the need for a low power consumption. The services they provide to their users are usually constrained by strict time deadlines. When using a system software in embedded environments, we also have to regard these restrictions. Thus, we can outline the requirements implicated from the field of embedded real-time computing:

- The embedded RTOS must be very time and memory efficient.
- It has to be compact and concentrate on the most necessary functions.
- The timing behaviour of the system software must be known and predictable.
- Predictable thread synchronisation mechanisms and a real-time capable memory management must be supported by the RTOS as well.

2) Demands from SMT and Multi-Core Hardware: Simultaneous multithreading (SMT) is the ability to concurrently run programs divided into subcomponents or threads on a single processor or within a processor core. While the SMT execution is parallel but only applicable when latencies arise within the pipeline, multi-core hardware offers real parallelism. However, both mechanisms promise better utilisation of processors and other system resources. Working with several tasks in parallel, a multithreaded or multi-core hardware can also cause a lot of new potential bugs to be introduced into an application. So we can add as specific requirement that an RTOS for multi-cores has to be aware of the following challenges:

- The RTOS must avoid race conditions and deadlocks caused by timing problems or commonly accessed resources like memory.
- Moreover, the system software has to provide synchronisation mechanisms, which enable an adjustment and communication between different processes running in parallel.

3) Requirements from Application Side: In order to enable an easy-to-handle application development, the interface is needed to be geared towards a common standard. As POSIX [9] is widely used also in the fields of embedded real-time systems (e.g. QNX [10]), it is useful to support functionalities following this standard. Thus, one can provide a familiar handling of parameters, return values and function names, e.g. when using synchronisation mechanisms for parallel applications. It will be easy to port applications developed for other systems using the POSIX interface as well.

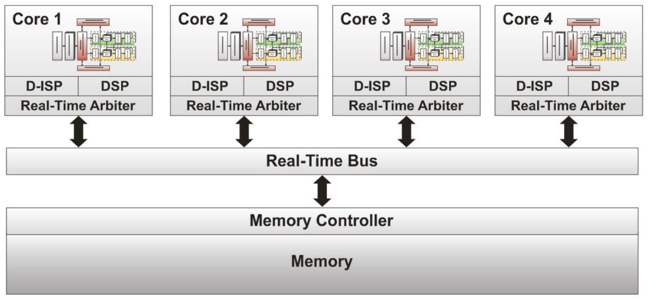


Figure 1. MERASA quad-core processor prototype

B. Requirements for a Multi-Core Hardware

The multi-core hardware on which multithreaded parallel real-time applications are executed needs to fulfil specific real-time requirements. Such an analysable hardware has to ensure that inter-thread interferences on shared ressources (e.g. memory) can be time bounded. Also, the hardware needs to ensure that HRT threads do not interfere with other HRT or NRT threads. Furthermore, multithreaded parallel applications require synchronisation support by the hardware. For real-time applications this synchronisation support must be WCET analysable, too.

IV. Overview of the MERASA multi-core architecture

The MERASA multi-core processor baseline is the single-core CarCore processor [12] [13], developed at University of Augsburg. The CarCore implements the Infineon TriCore Instruction Set Architecture [14]. The cores of the MERASA multi-core processor are in-order SMT-cores allowing one HRT thread per core to run in full isolation. A detailed description of the MERASA multi-core and its implementations as low-level (SystemC) and high-level simulators are available at the project website of the MERASA project (www.merasa.org). In this paper, we focus on a version of the MERASA multi-core processor, as shown in figure 1 and implemented as FPGA prototype. The cores of our multi-core processor features two pipelines, an address and a data pipeline, a first level of hardware scheduling to the thread slots, a real-time aware intra-core arbiter, a data scratchpad (DSP) and a dynamic instruction scratchpad (D-ISP) [15] on core level. Also, a second level of hardware scheduling interfaces with the MERASA RTOS. The real-time aware intra-core arbiters are connected to the real-time bus for accessing the main memory. The real-time bus features different real-time bus policies [16] for dispatching requests to the memory, but in this paper we focus on the TDMA real-time bus policy. The TDMA policy dispatches

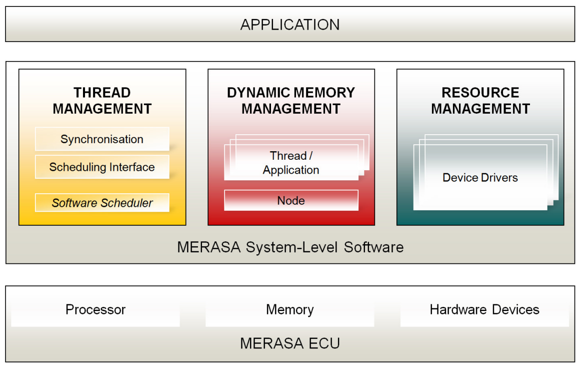


Figure 2. Architecture of the MERASA system software

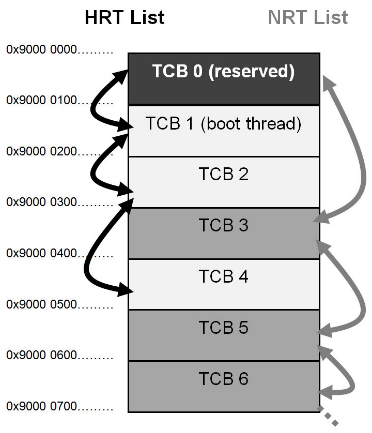


Figure 3. Two linked lists of TCBs, ready for the scheduler

in succeeding bus slots requests of different cores in a round-robin fashion; thus, the waiting time for a bus access is limited by the number of cores. For synchronisation between threads we implemented an atomic read-write in the memory controller, which does not introduce an increase of the overall WCET (see section V). Thus, we are able to execute multi-programmed and multithreaded workloads on our multi-core architecture.

V. Proposed System Software and Hardware Support

The design of the MERASA system software joins several well-known OS techniques. The basic kernel comprises the most important management functionalities following the microkernel principle. Additional functions may run outside this kernel as separate components.

Figure 2 gives an overview of the proposed architecture. The core of the MERASA system software contains three components: the Thread Management, the Dynamic Memory Management and the Resource Management. These three parts run on top of the MERASA hardware proposed in section IV. To give consideration to the real-time requirements the system software uses pre-allocation techniques. In an initial phase, all resources, which may potentially cause non real-time behaviour, are allocated. When the application starts running, real-time behaviour is guaranteed for all resource accesses.

A. Thread Management

A basic goal of the system software is to provide a solid interface for an easy creation, suspension and wake-up of specific threads from application side. The MERASA system software achieves this goal by being compatible to the commonly used POSIX interface. The scheduling interface internally builds a correct and consistent baseline for the hardware by managing hard real-time (HRT) and non real-time (NRT) threads in a time-bounded fashion.

Moreover, it is necessary to fulfil the requirements on thread synchronisation. For this reason, the system software provides commonly used mechanisms like mutex, conditional and barrier variables. All these mechanisms are implemented following the POSIX standard. In the field of synchronisation, the challenges for a HRT execution on a multi-core processor are a time-predictable implementation of synchronisation primitives and a bounding of waiting times. This waiting time is also dependent on coding guidelines for the applications' programmer.

1) Scheduling Interface: The MERASA multi-core processor contains a two-level scheduler, which can distribute threads over the different cores and over the different available thread slots of each core.

The organisation of threads as interface between the system software and the multi-core processor is done as follows. Every HRT and NRT thread has its own *Thread Control Block* (TCB), which is an array of 256 bytes. In this array, the whole thread context, like register values, synchronisation and scheduling information is stored. All TCBs are located in segment `0x90000000` and aligned to 256 byte boundaries. TCB 0 is reserved for special data used by the hardware scheduler, it contains the heads of two double linked lists, one for the HRT threads and one for the NRT threads. Each TCB contains two pointers `sched_next` and `sched_prev`, which are used for building these lists. On thread creation a TCB is put into one of the HRT or NRT lists using these pointers. The position of the TCB within a list indicates the order in which the corresponding threads are scheduled. Figure 3 shows an exemplary structure for such lists within the TCB memory. Concerning the timing behaviour, we benefit from the fact that the number of HRT threads is bounded by the number of cores, allowing only one HRT thread per core (to avoid interferences of HRT threads within a core).

As the progress of thread creation cannot guarantee timing constraints, we decided to introduce an initial phase, where only one boot thread is running on one core. It is possible to create new HRT and NRT threads only during this phase. The system software then fulfils all preparation jobs with

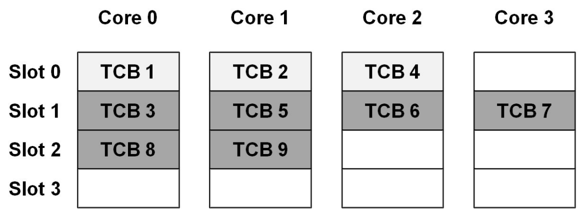


Figure 4. Distribution of the TCBs of figure 3 on a quad-core with four hardware thread slots per core

non real-time behaviour, like memory allocation to guarantee timing correctness during the following execution phase. From application side, it is very important to signal the end of the initialisation phase by calling exactly once the function `finish_thread_init()`. After this call no further creation of threads is possible, all needed threads must have been created during the initialisation phase.

The call to finish the initial phase and to start thread execution internally writes the heads of the HRT and NRT threads to the specified addresses of the reserved TCB 0. This write access is snooped by the second level of the hardware scheduler, which in turn now starts operation. It begins traversing first the HRT list and puts each TCB onto another core, each in thread slot 0. This continues until the list is terminated or the maximum number of cores is reached. There is one special case: as slot 0 of core 0 is the boot thread (the only one active after a reset, the head of the HRT list must always point to TCB 1 (the boot thread) and then to the next HRT thread.

A write to the head of the NRT list, which is also triggered by finishing the initial phase, also invokes the hardware scheduler: it starts traversing the NRT list. In detail it reads the first NRT thread's TCB from the specified memory location and writes it to core 0, thread slot 1. Following `sched_next` it puts the next threads into thread slot 1 of core 1 up to `MAX_CORES`. If the maximum number of cores is reached, it continues with thread slot 2 on each core, and so on. In figure 4, the distribution of the different TCBs over the slots is shown in more detail for the list presented in figure 3.

As already mentioned above, the developed hardware scheduler consists of two levels: a fixed priority (FP) scheduler in the issue stage of the pipeline and the initialisation logic to distribute threads over the slots in all cores. The FP scheduler is quite simple: there are 4 slots per core and the issue logic chooses the thread with the highest priority that is ready. The HRT thread always has the highest priority within one core. The second level of the hardware scheduler is the initialisation logic, which provides an interface to tell the cores where to begin the program execution. It sets the program counters and register values for each slot in every core according to the prepared TCB. In order to achieve higher flexibility of scheduling, it is possible to additionally use a software scheduler.

2) Synchronisation: The second basic part of the Thread Management of the MERASA system software besides the scheduling interface is the provision of synchronisation mechanisms.

As base, a commonly known spinlock mechanism is implemented, which can be used to give only one thread access to a critical region. If another thread tries to access the same region, it performs "busy waiting" until the lock is free. Nevertheless, we can guarantee timing predictability, as the number of potentially waiting threads, i.e. the list of HRT threads is bounded by the number of cores in our architecture. However, the spinlock mechanism is only used in the system software for extremely short critical regions. From architecture side, the implementation of spinlocks is based on the atomic swap instruction (see below for details) enabling good performance and predictable blocking times.

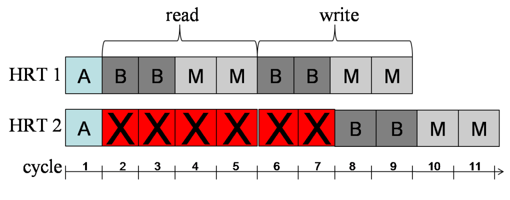
The spinlock methods are not provided to application programmers in order to advise the usage of mutex variables instead. Although the implementation is internally based on spinlocks, they cover only a very short critical region of a few cycles. Whenever a thread has to wait for a lock to become free, it is suspended from execution and inserted to a waiting list connected to the specific mutex variable. As soon as the mutex is unlocked, the suspended threads are reactivated following the waiting list. From a hard real-time point of view, it is essential to avoid common locks for HRT and NRT threads, because a HRT thread may have to wait for freeing a lock held by a NRT and therefore loses its property of being a HRT thread. However, this fact must be included in some coding guidelines to guarantee a correct programming from application side.

In order to give a good solution for thread communication, the thread management of the MERASA system software provides conditional variables. By calling a wait on a conditional variable, a thread is suspended and registered to a waiting list. This progress is very similar to the waiting on a locked mutex variable, as described above. As soon as the signal function is called, the first thread of the waiting list is unsuspended, on broadcast all waiting threads are unsuspended and continue execution.

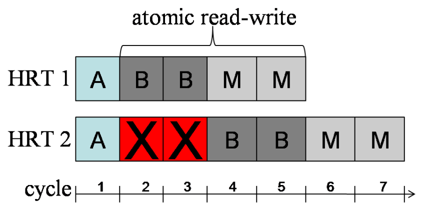
Finally, in order to enable a synchronised start of a specific section of different threads, the thread management provides barriers. In an initialisation function, the application programmer can set the number of threads on which the barrier should wait. Every time a thread calls the wait function, a counter, connected to the initialised barrier is incremented and the thread is suspended from execution. The suspension itself is based on condition variables. As soon as the incremented counter reaches the value which was set on initialisation, all threads in the waiting list get a signal to synchronously continue execution.

3) Hardware Support for Synchronisation: To enable the MERASA multi-core to execute multithreaded, parallel tasks, we introduced an atomic read-write instruction for synchronisation of parallel HRT threads running on different cores. Therefore, the atomic SWAP instruction of the TriCore instruction set is changed, in order to be atomic in a multi-core environment.

The SWAP instruction in the single-core environment consists of a load and a following store, and is executed atomically by blocking other threads from interrupting the load and store sequence. In the MERASA multi-core with a real-time bus connection to memory, this would have introduced additional waiting cycles at the real-time bus for other threads, when a thread issues a swap instruction, and therefore additional latency. This latency would increase the WCET of all load and store instructions.



(a) Locked Real-Time Bus



(b) Atomic Read-Write for Real-Time Bus

Figure 5. Access patterns for two HRT threads accessing the real-time bus.

Figures 5(a) and 5(b) show access patterns for two HRT threads accessing the real-time bus (A), followed by being dispatched to the memory (B) or being stalled and waiting to be dispatched to the real-time bus (X), and the following memory access (M). In the case of thread HRT1 issueing a load, thread HRT2 has to wait to access the bus until thread HRT1 has freed it, and therefore thread HRT2 has to wait two cycles before being dispatched to the real-time bus. Figure 5(a) depicts the increase of waiting time for thread HRT2, if thread HRT1 executes a read-write for synchronisation with locking of the real-time bus. Hence, thread HRT2 has to wait until the sequential read-write is finished and thread HRT1 frees the real-time bus. This leads to an increased WCET of all loads and stores. To avoid the locking of the real-time bus for the sequential execution of the read and write, we introduce an atomic read-write without extra waiting time at the real-time bus for other HRT threads (figure 5(b)). Additional logic is required to assure atomicity in a multi-core environment with shared bus access to memory.

We investigated different possibilities to assure atomicity with focus on adding as little additional latency as possible and put the additional logic for the atomic read-write instruction into the memory controller. This is motivated by the case that the atomic read-write is needed for synchronisation in the MERASA multi-core, and therefore we only need to read and write one bit of data from the memory. Reading a '0' from a specific memory address indicates a lock is free, and reading a '1' indicates that the lock cannot be gained. In the sequential order of a read and write this means a thread reads the data from an address in memory followed by always writing a '1' to that address. This assures that only one thread at a time can hold the lock guarded by that memory address. Also the memory access policy motivates adding this logic in the memory controller. As we access the memory in bursts, the reading and writing of the SWAP can be done in the same time as a usual burst access to memory as we only need to read and write one bit of data. This adds no additional latency for an atomic read-write and thus avoids increasing the WCET of the task. Assuming the burst length is long enough, i.e. 64 bytes, a read from an address and an adjacent write to that same address is possible in less time than a burst needs to access the memory.

Concluding, we introduce less additional latency and a tighter WCET with this implementation of the atomic read-write in the memory controller, as we do not need to lock the real-time bus for other threads, while one thread issues a SWAP instruction. To demonstrate the bounded timing behaviour of our synchronisation mechanisms, we show some details of the execution times of the single methods in section VI.

B. *Memory Management*

In contrast to traditional non real-time memory management systems, it is necessary for our memory management to provide timing guarantees. We introduce a two-layered memory management and use memory pre-allocation. We minimise interferences of several threads among each other and provide higher flexibility for applications at the same time.

On the first layer – the *node* level – large blocks of memory are allocated. This allocation is performed in a mutually exclusive way to keep the state of memory consistent, so here a blocking of threads can occur. But as this is only done in the initialisation phase before threads are started, influences on the real-time behaviour of the system can be neglected. On the *thread* layer the memory management allocates memory to the executed program in the specific thread. This can be done without locking, because the memory is taken from the blocks pre-allocated in the node level exclusively for the thread.

Alongside we can see another advantage of such a two-layered architecture. As it is always necessary to keep the information, which memory block belongs to which thread, a lot of management data is needed by putting them into a linked list including list pointers. In contrast to a conventional (one-layered) allocation scheme, our list pointers need only be added to the large blocks on node level. Thus, some memory can be saved and the management data needed to keep track of the owners is reduced.

Regarding the implementation, dynamic memory management on the node level is currently performed by an allocator based on Lea's allocator [17] (DLAlloc). On the thread level, the user can choose between various implementations of memory allocators. For non real-time applications, efficiency of memory usage can be improved by a best-fit allocator like DLAlloc. This variant is fast and space-conserving but hardly real-time capable. If a real-time application requires the flexibility of dynamic storage allocation, an allocator with bounded execution time can be used. The Two-Level Segregate Fit (TLSF) allocator [18] is a general purpose dynamic memory allocator specifically designed to meet real-time requirements. Using this alternative, the computation of worst-case execution time (WCET) is simplified. Whereas in DLAlloc, the execution time depends on the current state of the allocator and on the previous de- / allocations, TLSF provides a bounded execution time regardless of its former operation at the cost of higher internal fragmentation.

The memory management supports different types of memory. It provides functionalities to define the configuration, i.e. beginning, end, length and the cost for the use. By this means, a high flexibility of memory structures is achieved and timing remains analysable.

C. Resource Management

It is a main task of the system software to enable the usage of computer hardware. As already mentioned, the MERASA system software follows the microkernel principles, i.e. manages only the most essential system resources like processing time and memory. To gain maximum flexibility, hardware devices are accessed through device drivers. These resources are managed by a dedicated *Resource Management* unit.

The implementation of the resource management, containing a driver manager, is a subset of the POSIX standard [9]. It contains the generic `open` / `close` operations as well as functions to access the device (`read` / `write` operations) and for configuration (`ioctl`). Valid configuration values and further parameters depend on the specific device and driver.

In the Resource Management the problem of concurrent use of devices can be reduced to thread synchronization. For this, the Thread Manager already provides solutions. We assume that most operations can be dedicated to the non real-time initialisation phase and therefore have no influences on the timing behaviour of the execution phase. In general, there is no limitation on the number of drivers supported by the resource manager, however, the timing behaviour depends on this quantity. For the use in real-time applications, the number of threads sharing a resource must be limited to guarantee device accesses in bounded time. As the number of devices and threads an application uses is known in advance, constant-access-time handlers can be arranged during the preparation of the application's execution environment. Thus, device accesses can be performed in constant time - of course, depending on their peripherals.

VI. Evaluation

In this section, our objective is to show that the worst-case execution time of the mechanisms described in the previous section is predictable. First, we give a short overview of WCET analysis techniques and we introduce our OTAWA tool used for the experiments. Then we show how the execution times of the system functions can be determined. Finally, we provide the WCETs computed considering our MERASA multi-core architecture.

A. WCET Analysis Techniques

Usual approaches to determine the Worst-Case Execution Time of a task that is subject to strict timing constraints are based on static code analysis techniques. On one side, information about the possible control flows is derived from the source code and/or specified by the user. On the other side, the worst-case execution times of basic blocks, considering any possible execution history, are computed [19] [20]. Finally, the WCET of the whole task is determined, normally using Implicit Path Enumeration [21].

In the MERASA project, static WCET analysis is performed within the OTAWA framework [22]. Specific algorithms have been developed to account for the MERASA multi-core features including an SMT core, instruction and data scratchpads and the real-time bus (as described in section IV). The experimental results provided below were obtained using this toolset, considering a 4-core MERASA architecture with a 1-cycle bus latency and a 4-cycle memory latency.

B. Experimental results

1) Analysis of synchronisation methods: The execution time of a synchronisation function breaks down into:

- a *running* part, T_e, that is the execution time when the thread does not have to wait (for locks, conditions, etc.)
- a *waiting* part, T_w, related either to busy waiting or to the suspension of the thread.

In turn, the waiting time usually breaks down into three elements:

- the first one (T_{w1}) does not depend on the number of concurrent threads, nor on the usage that is made of the function by the user application,
- the second one depends on the number of competing threads (T_{w2}) but *not* on the application
- the third one depends on both the application and the number of competing threads (T_{w3}).

To illustrate this, let us consider the acquisition of a lock. The waiting time includes the time to determine that the lock is not free, the time the lock is held by another thread, and the time to resume after the lock has been released by the other thread. The times to determine that the lock is not free and to resume do not depend on the number of threads nor on the usage of the lock: they are included in T_{w1}. On the other hand, the time the lock can be hold by another thread may be known when the system software is analysed (this is the case for a lock that is private to the system functions) and then it is included in T_{w2}; alternatively, it may remain unknown until the user application is analysed (for a lock used at the application level) and then it is included in T_{w3}. T_{w3} would be the maximum execution time of all the critical sections in the code that are guarded by the lock. In both cases, the time the lock might be hold is to be multiplied by the number of threads that might be competing for it.

If the number of concurrent threads is N (including the thread under analysis, that executes the synchronisation function), the execution time of this function can be written as:

$$T = T_e + T_{w1} + (N - 1) \times (T_{w2} + T_{w3})$$

Table I shows the worst-case execution and waiting times, in cycles, determined for the proposed synchronisation methods running on the MERASA multi-core. Part of the waiting time in `mutex_lock()` depends on how the lock is used in the application (i.e. it depends on the lengths of the critical sections protected with this lock) as well as on the number of threads that may compete for this lock. In the same way, part of the waiting time in `cond_wait()` depends on the time it might take until another thread signals or broadcasts on the condition, which is clearly related to the application. It also depends on the number of threads that can be waiting on the condition. The application-related part of the waiting time will generally be considered differently for methods like `barrier_wait()` and `join()`. When estimating the WCET for a parallel piece of code terminated with a barrier or a join, only the execution of the longest thread would be considered, as the longest thread is the last one to reach the barrier or the join. As a result, the waiting time is zero for this thread.

Table I
WCET OF SYNCHRONISATION METHODS

Function	T_e	T_{w1}	T_{w2}	T_{w3}
mutex_lock()	258	376	351	*depends on application*
mutex_unlock()	288	67	351	0
mutex_trylock()	176	67	351	0
cond_wait()	913	443	702	*depends on application*
cond_signal()	172	0	0	0
barrier_wait()	987	443	1467	-
join()	318	0	0	-

2) Analysis of dynamic memory allocation methods: Thanks to the selection of WCET-aware algorithms, we have found that the WCET of system methods dedicated to dynamic memory allocation is analysable. Table II gives the times that we determined considering the MERASA multi-core. The execution time of `realloc()` depends on the size of the memory block (n, given as a parameter of the function).

Table II
WCET OF MEMORY ALLOCATION METHODS

Function	WCET
malloc()	759
free()	895
realloc()	1671 + 22×n

VII. CONCLUSION

In this paper we proposed a solution for the RTOS support of parallel hard-real time applications on the new MERASA multi-core architecture. First, we identified the problematic demands from analysable real-time environments combined with multi-core techniques - both on the system software and on the supporting hardware platform. In the main part of the paper, we subsequently presented the MERASA system software as multi-core RTOS fulfilling the stated requirements on thread, memory and resource management. In particular, it provides a real-time capable thread scheduling interface to the two-level hardware scheduler of the MERASA multi-core processor as well as time-bounded synchronisation mechanisms for a concurrent execution of threads. Moreover, our multi-core architecture is able to execute hard real-time threads in concert with additional non real-time threads. By ensuring a complete isolation, the hard real-time threads still meet their deadlines, while the processor also provides execution capacity to the non real-time threads.

In our evaluation, we used the static analysis tool OTAWA in order to compute the WCET of the system software mechanisms for synchronisation and memory allocation. As a result, we could break down the methods into different elements and determine upper bounds for each part. By this, we can guarantee and prove a timing correct RTOS support for the execution of parallel hard real-time applications in our multi-core environment.

References

[1] L. Thiele and R. Wilhelm, "Design for time-predictability," in *Design of Systems with Predictable Behaviour*, 2004.

[2] F. Kluge and J. Wolf, "System-Level Software for a Multi-Core MERASA Processor," Institute of Computer Science, University of Augsburg, Tech. Rep. 2009-17, October 2009.

[3] M. Schoeberl, "Time-predictable computer architecture," *EURASIP Journal on Embedded Systems*, vol. 2009, Article ID 758480, p. 17 pages, 2009.

[4] M. Schoeberl, P. Puschner, and R. Kirner, "A single-path chip-multiprocessor system," in *Proceedings of the Seventh IFIP Workshop on Software Technologies for Future Embedded and Ubiquitous Systems (SEUS 2009)*, no. LNCS 5860. Springer, 2009, pp. 47–57.

[5] J. Whitham and N. Audsley, "MCGREP–A Predictable Architecture for Embedded Real-Time Systems," in *RTSS '06: Proceedings of the 27th IEEE International Real-Time Systems Symposium*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 13–24.

[6] "PREDATOR Design for predictability and efficiency - an FP7 project," http://www.predator-project.eu/, visited November 2009.

[7] S. A. Edwards and E. A. Lee, "The case for the precision timed (PRET) machine," in *DAC '07: Proceedings of the 44th annual Design Automation Conference*. New York, NY, USA: ACM, 2007, pp. 264–265.

[8] S. Andalam, P. Roop, A. Girault, and C. Traulsen, "PRET-C: A new language for programming precision timed architectures," Tech. Rep. [Online]. Available: http://hal.archives-ouvertes.fr/inria-00391621/en/

[9] "IEEE Std 1003.1, 2004 Edition. The Open Group Base Specifications Issue 6," 2004.

[10] "QNX Neutrino RTOS," http://www.qnx.com/products/, visited November 2009.

[11] "Enea OSE: Multicore RTOS," http://www.enea.com/, visited November 2009.

[12] S. Uhrig, S. Maier, and T. Ungerer, "Toward a Processor Core for Real-time Capable Autonomic Systems," in *Proceedings of the 5th IEEE International Symposium on Signal Processing and Information Technology*, Dec. 2005, pp. 19–22.

[13] J. Mische, I. Guliashvili, S. Uhrig, and T. Ungerer, "How to Enhance a Superscalar Processor to Provide Hard Real-Time Capable In-Order SMT," in *23rd International Conference on Architecture of Computing Systems (ARCS 2010), Proceedings*, Hannover, Germany, Feb. 2010, pp. 2–14.

[14] *TriCore 1 Architecture Volume 1: Core Architecture V1.3 & V1.3.1*, Infineon Technologies AG, Jan. 2008.

[15] S. Metzlaff, S. Uhrig, J. Mische, and T. Ungerer, "Predictable Dynamic Instruction Scratchpad for Simultaneous Multithreaded Processors," in *MEDEA '08: Proceedings of the 9th workshop on MEmory performance*. New York, NY, USA: ACM, 2008, pp. 38–45.

[16] M. Paolieri, E. Quiñones, F. J. Cazorla, G. Bernat, and M. Valero, "Hardware Support for WCET Analysis of Hard Real-Time Multicore Systems," in *ISCA '09: Proceedings of the 36th annual international symposium on Computer architecture*. New York, NY, USA: ACM, 2009, pp. 57–68.

[17] D. Lea, "A memory allocator," http://g.oswego.edu/, 2000.

[18] M. Masmano, I. Ripoll, A. Crespo, and J. Real, "TLSF: A New Dynamic Memory Allocator for Real-Time Systems," in *ECRTS '04: Proceedings of the 16th Euromicro Conference on Real-Time Systems (ECRTS'04)*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 79–86.

[19] S. Thesing, "Safe and Precise WCET Determination by Abstract Interpretation of Pipeline Models," Ph.D. dissertation, Universität des Saarlandes, 2004.

[20] C. Rochange and P. Sainrat, "A Context-Parameterized Model for Static Analysis of Execution Times," *Transactions on HiPEAC*, vol. 2, pp. 222–241, 2009.

[21] Y.-T. Li and S. Malik, "Performance Analysis of Embedded Software using Implicit Path Enumeration," in *Workshop on Languages, Compilers, and Tools for Real-time Systems*, 1995.

[22] H. Cassé and P. Sainrat, "OTAWA, a framework for experimenting WCET computations," in *3rd European Congress on Embedded Real-Time Software*, 2006.