Compositional Schedulability Analysis of Multicore Modular Avionic Architectures

Jalil Boudjadar*
Department of Engineering, Aarhus University Denmark.

* Corresponding author. Tel.: 004591962287; email: jalil@eng.au.dk
Manuscript submitted July 14, 2018; accepted August 12, 2018.
doi: 10.17706/jcp.13.10.1202-1215

Abstract: This paper presents a compositional schedulability analysis of multicore modular avionic systems (IMA). It provides a fine grained description of the software architecture and behavior to reduce the over-approximation, and a holistic analysis to check schedulability while considering computation requirements and shared memory interference. The system is structured in terms of subsystems, encapsulating ARINC653 partitions, each of which runs on a processing core. The schedulability analysis is performed for each subsystem individually while accounting for the memory interference that would result if the core under analysis runs effectively alongside with the rest of cores. Thereafter, we introduce an architecting technique to relocate functions between tasks located at the same partition in case of non schedulability, to derive potential schedulable system configurations delivering the same functionality. Schedulability is formally analyzed using Uppaal model checker. Our evaluation results show that our compositional analysis technique consumes up to 95% less than regular analysis, in terms of analysis time and memory space.

Key words: Modular avionic architectures, multicore systems, memory interference, schedulability analysis.

1. Introduction

To achieve separation of concerns, fault containment and efficient resources utilization in avionic systems, modular architectures such as IMA [1] and DIMA[2] have been introduced. Modular avionic systems are built by the integration of different subsystems through an incremental Design and Certification (iD&C) procedure [3]. Multicore platforms are finding their way into the deployment of avionic systems due to their ability to leverage the computing capabilities, reduce the weight of on-board computing equipment and lower the energy consumption. However, due to safety-criticality requirements and the complexity resulting from the architectures of software and execution platforms, analyzing the schedulability of avionic systems is a challenging task [4]-[7].

In software industry, the timing constraints imposed on application programs, represented by tasks, are usually estimated off line using schedulability tests while considering worst case potential parameters such as worst case execution time (WCET) and deadlines as periods. Estimating a precise execution time of tasks is very challenging due to non- determinism and dependency of the tasks execution on the execution platform and environment, so that extra delays can result from caching, memory access conflicts and data transfer via shared networks and buses. What is classical for the schedulability analysis of multicore systems is that the WCETs of tasks need to be estimated first while each task runs in isolation performs

perfect memory access i.e., the memory is immediately available whenever an access request occurs (hits) [6]-[9]. Such an estimation of the WCET at task level is most likely under-approximating. We believe that estimating the execution time at a lower level of the granularity (e.g. function level) results in less under-approximation of the execution time. Thus, the more granular the behavior representation is, the more optimistic the WCET will be. The under-approximation difference between task and function levels could be non comparable and would increase drastically with fine grained description levels.

The interference resulting from shared memories and communication means (buses, networks) is a determinant factor in the schedulability of component-based real-time system. The interference time is in fact related to the number of concurrent components and the bandwidth of shared resources. Accordingly, accurate schedulability techniques require a holistic analysis where both computation and memory interference/ communication have to be considered together [10], [5].

A surge of progress has been achieved in the area of schedulability analysis of multicore systems through an intensive use of model-based settings and formal methods. However, due to the systems size in avionics, given by the integration of huge number of concurrent applications, the use of formal methods can end up in state space explosion. Different techniques have been considered to bypass the state space explosion and provide upper bound guarantees on the schedulability, we cite abstraction-based [11], compositional analysis [12] and incremental analysis [13]. In this paper, we introduce a model- based framework for fine grained modeling and formal schedulability analysis of multicore avionic (IMA-driven) systems with shared memories. The system is structured in terms of subsystems, encapsulating ARINC653 partitions, each of which runs on a processing core. The schedulability analysis is performed for each subsystem individually while accounting for the memory interference that would result if the core under analysis runs effectively alongside with the rest of cores. Our framework does not require the WCET of application tasks to be provided in order to check the schedulability, but rather it assumes that the individual instructions (low level functions) of the application have been identified and associated both an execution time and a memory access pattern. In case of non-schedulability of a subsystem, we provide a technique to re-engineer the faulty subsystem by relocating functions between tasks located within the *same* partition, calculate what will be the resulting load and analyze the underlying schedulability. We restrict the migration of functions between tasks within the same core only to maintain the partition-based design concept and functional architectural constraints of IMA.

The rest of the paper is organized as follows: Section II cites the most relevant related work. Section III describes the IMA architecture and the overall scheduling and analysis methodology introduced in this paper. Section IV describes our formal modeling of modular avionic systems. Section V describes our compositional schedulability analysis and re- location techniques. Section VI discusses the scalability of our compositional analysis compared to regular schedulability analysis. Section VII concludes the paper.

2. Related Work

Analyzing the schedulability of a system application while considering an abstraction of the execution platform leads necessarily to underestimate the system requirements (workload) in terms of resources. This causes certainly serious deficits during deployment where processes run much longer than what they should, due to interference and non-deterministic contention of shared memories and communication means. Many researchers have been paying considerable attention to the schedulability and memory interference of multicore systems [7], [10], [12], [14]-[17].

Boudjadar *et al* [12] introduced a compositional framework for the schedulability analysis of uniprocessor hierarchical scheduling systems. The system components analyzed individually are given with interfaces describing the resource budget (in terms of processor utilization time) that the component under

analysis can acquire, in a non-deterministic way, from its parent level. Similarly, Carnevali *et al* [15] provided a compositional framework for the schedulability analysis of hierarchical scheduling systems, using preemptive time Petri nets, where components at each level are given static resource budgets. However, ignoring the memory interference when analyzing system components [12], [15] leads certainly to inaccurate and under-estimation of the tasks response time. Moreover, when considering shared memory features it is not sufficient to state how many access requests a task can perform but the schedulability is strongly correlated to *when* the requests are issued.

Madsen *et al* [18] studied the impact of mapping software applications to multicore platforms on the system schedulability. The platform model is given by a set of processing elements, each of which consists in a local memory (cache), a scheduler and a processor. The architecture model is given by a static allocation of tasks to cores, making thus the schedulability problem less complicate. However, ignoring the interference resulting from the shared memory may lead to an underestimation of the workload i.e., the delays resulted from local memory (cache) miss are not accumulated in the tasks response time.

Yi *et al* [14] provided a framework to calculate tasks WCET for multicore systems with a shared memory level. The authors use abstract interpretation to analyze the local cache behavior of a taskset running on a given core in order to capture the precise time points when tasks access the shared memory. Compared to that, we consider different access patterns to local cache level and shared DRAM whereas the effective access requests are non-deterministically carried out along the tasks execution. Moreover, we analyze the schedulability of the model formed by a composition of tasks behavior and memory access in a compositional way. Gustavsson *et al* [19] investigated the calculation of WCET for multicore systems. The miss/hit of local and shared cache levels is non-deterministic. Once a local cache miss occurs, the data will be fetched from the shared cache and if it fails again an access to the shared memory is required, which is always assumed to hit. WCET is obtained using a binary search, which could be expensive if the number of programs (tasks) is large and/or the initial WCET candidates are far from the final values.

Madsen *et al* [20] introduced a framework for the modeling and verification of multiprocessor system-on-ships. The platform model is a set of cores, each of which has a local scheduler, without any consideration of memory. In contrast, our work does not assume the application tasks to be assigned with pre-estimated WCETs, rather fine grained description of the concrete behavior of tasks suffices.

In this work, we consider modular avionic systems running on multicore platforms with shared memory interference. Beside to local cache level L1, we consider a shared DRAM memory. The system architecture can be viewed as a 2-level hierarchy where the parent level corresponds to ARINC653 scheduler, whereas the child level is the scheduling algorithm of each individual partition. The schedulability of each core workload is analyzed individually while accounting for the worst case interference time, which abstracts how the rest of the system can effect the core under analysis to access DRAM. In case of non-schedulability, based on counter- examples we relocate functions from the tasks missing deadlines to those having a considerable laxity. This may derive schedulable configurations achieving the same functional mission.

3. Architecture and Methodology

This section outlines our methodology to model and analyze multicore modular avionic systems.

3.1. Conceptual Design

The system architecture we consider is given by a set of independent applications, to achieve separation of concerns [21], representing the different modules in an avionic system. Each application is formed by a set of parallel real-time tasks released periodically. As mentioned earlier, we consider concrete tasks behavior rather than abstractions in terms of worst case execution time (WCET). Basically, a task is a sequence of functions each of which models the execution of a software function. Each function is given by a

sequence of timed actions (statements) representing a subset of the basic commands of a process such as Compute, Input, Output, *etc*. Each statement is then abstracted using its worst case execution time and its access patterns to the existing memories. By access pattern we mean to which shared memory the access hits [10]. We extend the classic static access patterns {Cache, DRAM} [14] with a *Non-deterministic* option where the memory/cache access performed by a timed action can hit in any memory, otherwise if a miss is reported then the access request will be forwarded to the next higher level in the hierarchy.

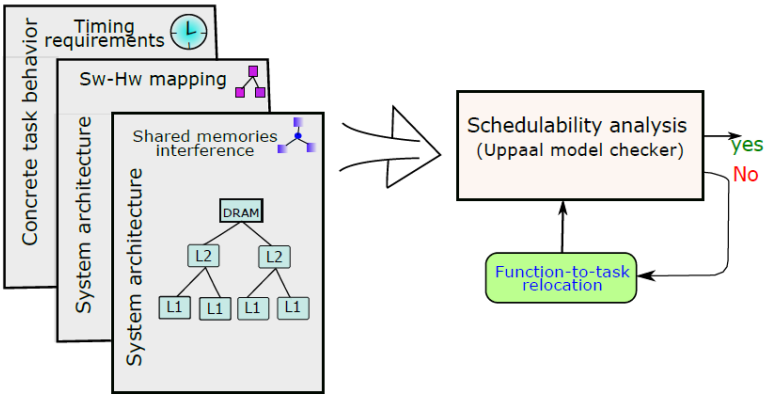

Fig. 1. Overall methodology.

The execution platform is given by a set of processing cores sharing a DRAM memory. Each processing core is equipped with a local cache level L1 and runs a set of applications. Each of the cores runs a set of applications in a static cyclic way according to the scheduling mechanism imposed by ARINC653 [22] where each application is scheduled to run for one or more time slots each major scheduling frame, i.e. the time interval to repeat the schedules execution. The set of applications assigned to the same core is known by *subsystem*. Fig. 1 depicts the inputs, workflow and outcomes of our framework. To be able to perform schedulability analysis, our model-based setting requires as input the concrete tasks behavior, system architecture including both software and platform, software-to-platform mapping and the relative worst case interference time. Using Uppaal model checker, the schedulability is formally and rigorously analyzed. In case a system configuration is not schedulable, based on counter-examples generated by Uppaal, we apply a re-engineering by relocating timed actions between tasks and rerun the analysis in the goal to identify whether a potential configuration leading the system to be schedulable can be derived. To maintain the separation of concerns, functional-architectural constraints [23] and architectural cohesiveness [24], we only consider the relocation of actions belonging to the tasks of the same application, i.e. located at the same core. The relocation technique is in fact performed as a binary search process. The functional architectural constraints impose that the functionality of each architectural unit is maintained if the functions of that unit do not execute within the same real-time task (RTOS time partitioning). The architecture cohesiveness is used to reduce coupling and dependency, where each individual architecture part addresses a single well-defined piece of the problem.

3.2. IMA Architecture and Hierarchical Scheduling

Federated avionic architectures assign software modules and functions to dedicated resources in an exclusive and static way, where each resource can be exploited by *only one* module. Such an architecture favors fault containment due to separation, however it is not efficient in terms of resources utilization. Integrated modular avionics architecture [1] has been introduced as an alternative to improve the utilization of resources by sharing communication and computation resources between the software

applications, thus reducing the design cost, weight and energy consumption compared to federated architectures. The system functions, encapsulated within applications, are gathered in terms of partitions. Namely, a partition is a unit resulting from a functional decomposition of the system, and for which the hardware will be assigned as a whole. A partition may contain processes belonging to different applications. The processes of a given partition are scheduled internally using a local scheduling policy (partition level). Partitions are scheduled by ARINC653, where the processing resources are allocated to partitions according to given well-defined time slotted windows. A partition can then simply be referred to as a logical allocation of a processor to a set of processes. Fig. 2 illustrates a simplified IMA architecture.

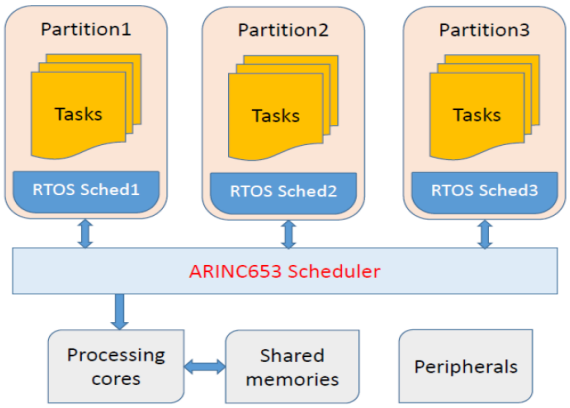

Fig. 2. IMA architecture.

The control flow is handed in from a level to another in the hierarchy as depicted in Fig. 3. The scheduling within IMA architectures can be viewed as a 2-level hierarchy where processing cores are allocated to partitions for given time du- rations (though not necessarily equal) according to ARINC653 predefined static schedules. The cores will be released, leading, thus the partition to be preempted, whenever the specified time quantums expire. Within a partition, a task can request processing resources by issuing a ready event. Similarly, a partition is notified by the termination of a local task via event done. Once a partition is allocated a processing core it schedules its local ready tasks, and potentially preempts running tasks, according to a local scheduling algorithm.

3.3. Interference and Compositional Schedulability Analysis

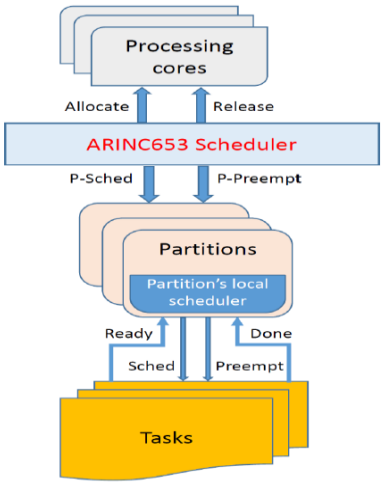

Fig. 3. Scheduling within IMA architecture.

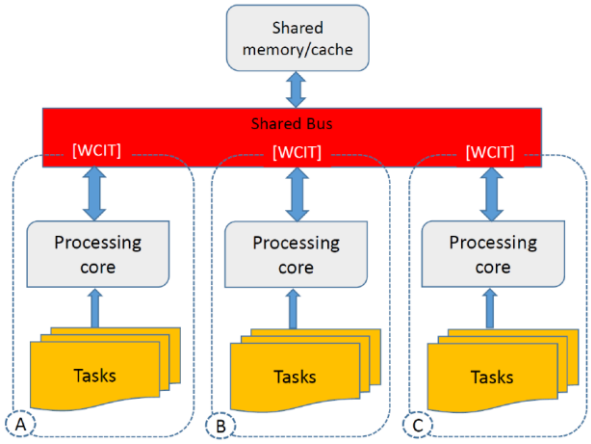

Fig. 4. Compositional schedulability analysis.

To keep pace with the growing deployment and complexity of multicore systems, considerable research interference is one of the strong bottlenecks playing a determinant factor for the deployment of multicore systems. In practice, shared memories are not able to run with the same frequency processing cores, so the number of efforts have been devoted to timing analysis and memory interference [5], [7], [10]. Actually, memory as access requested issued by cores for a time duration is much larger than the memory bandwidth [17]. Hence, processing cores stall to wait for their access requests to be granted. Different techniques have been introduced to analyze and bound memory interference of multicore systems [10], [17], [3], [7]. The worst case interference time (WCIT) can be calculated either online or offline. Online analysis techniques [10], [25] rely on statistical analysis of a system, or though a model of the system, where WCIT is identified to be the largest interference time obtained when the system runs under extreme conditions. In contrast, offline analysis techniques [7], [16] calculate the interference of each access request individually using a mathematical model of the system together with a fine grained internal architecture of the shared memory/caches.

Fig. 4 illustrates our compositional analysis carried out for a 3-cores system having interference at the shared memory bus. The schedulability of each core's load is analyzed separately, assuming that each access request performed by any core lasts for the worst case interference time (WCIT). In such a way, when analyzing an individual core we abstract the rest of the system while taking into account the interference that would result if the core under analysis runs effectively alongside with the other cores and competes for the access to shared memory. Accordingly, the schedulability analysis of the system depicted in Fig. 4 can be performed via three independent and identical analysis processes **A**, **B** and **C**. Each of the analysis processes is parametrized with WCIT. Moreover, the analysis processes can be performed in any order, though potentially in parallel Our framework provides the ability to analyze multicore systems, running any composition of scheduling algorithms, deemed to be non analyzable using regular schedulability tests [26]-[28].

4. Modeling of Multicore Avionic Architectures

This section describes the architecture and behavior of the modular avionic systems we consider for analysis. The overall system architecture is depicted in Fig. 4 where tasks allocated to a given core are encapsulated within partitions. Roughly speaking, a partition represents a software application where the encapsulated tasks exchange data via memory space. The platform is formed by a set of processing cores sharing a DRAM memory. Moreover, each core is equipped with a local cache level L1. At a core level, partitions are scheduled using the ARINC653 policy. To apply the current work, the following assumptions

must be held:

- Tasks are periodic and preemptible.
- Partitions are statically mapped to processing cores.
- Each partition schedules its local tasks using a local scheduling algorithm such as RM, FPS.
- At any cache level, we do not distinguish between cache for data and cache for instructions.
- Relocation of functions can only be performed within the same partition.

Before diving into the technical descriptions, let us introduce the following notations:

- T is a task, and $\boldsymbol{T}$ denotes a set of tasks.
- C is a core, and $\boldsymbol{C}$ denotes the set of all cores.
- P is a partition, and $\boldsymbol{P}$ is a set of partitions.
- f is a function, and $\boldsymbol{F}$ denotes a set of functions.
- a is a timed action, and $\boldsymbol{A}$ denotes a set of timed actions.
- When we refer to a specific element, respectively set, in the aforementioned notations, except $\boldsymbol{C}$, we use indexes e.g., T_i, $\boldsymbol{T_j}$, etc.

4.1. Partitions Specification

A partition P is given by a set of tasks, each of which describes the execution of an individual process, and a local scheduling algorithm. Each task is a sequence of functions representing the basic functions and procedures of a software program. A function in turn is formed by a sequence of timed actions modeling the elementary (atomic) statements of an avionic system such as Compute, Input, Output, etc. Timed actions are not preemptible for CPU use, thus a function can be preempted in-between timed actions only.

Each timed action is assigned an execution time, which is the duration (in terms of clock cycles), to execute on a processing core without including the time to fetch potential data. To perform the computation modeled by a timed action, a data fetching can be required. We consider the following patterns to specify the data access of timed actions:

- NONE: neither cache nor DRAM access is required. This specific option is ideal for particular instructions like *END*, *EOF*, *EXIT*, etc.
- CacheOnly: a data access to L1 is needed and always leads a (successful) cache hit.
- DRAMReq: a data access is required however the L1 cache access always misses, thus data will be fetched from the main DRAM memory.
- NonDeterministic: a data access is required how- ever the cache hit/miss is non-deterministic. Thus, if a cache miss occurs it must be followed by a DRAM access request.

Such patterns are identifiable using AI program/cache analysis [29], e.g. tool PAG [30], and abstract interpretation [14]; and can be stored as a benchmark library for a given platform architecture.

Definition 1 (Timed action): A timed action $a = \langle \pi, \alpha \rangle$ is a single step processing operation given by an execution time π and a data access pattern α.

Accordingly, a function $f = (a_1, .., a_n)$ is given by a sequence of atomic timed actions. A task behavior is formed by a sequence of functions. For the sake of flexibility, we decouple the description of timing constraints and behavior of each task so that the behavior and constraints can be updated independently.

Definition 2 (Task behavior): The behavior β of a task is a transition system $< L, l^0, L_A, \mathrm{F}, \rightarrow >$ specifying the sequence of functions performed by that task where:

- L is a set of locations and $l^0 \in L$ is the initial location,
- $L_A \subseteq L$ is a set of final (acceptance) locations,
- $\mathrm{F} = \{f_1, .., f_n\}$ is the set of functions,
- $\rightarrow \subseteq L/L_A \times \bigcup_i f_i \times L$ is the transition relation from a non-terminal location to another location through the execution of timed actions.

The task behavior is subject to a set of time constraints, which are the classic scheduling parameters such as period, deadline and priority. We constrain the behavior of tasks as described in the following definition.

Definition 3 (Task structure): A task T is given by (p, d, ρ, β) where:

- p is the task period describing the roundness of the task behavior,
- d is the relative task deadline by which the task behavior must complete,
- ρ is the priority level associated to task T ,
- β is the actual task behavior.

We have made our Uppaal task model parametrizable so that it can be instantiated for a large set of tasks. The tasks T of a given partitions P will be scheduled by a real-time scheduling function S given by:

$$S : T \times T \times R_{\geq 0} \rightarrow T$$

where $R_{\geq 0}$ is the time domain. The function S compares two tasks at any time point and identifies the task having priority at that time instant. S is described abstractly in order to be able to model both static and dynamic priority scheduling algorithms. The scheduling policies implemented in our framework are: First-in First-out (FIFO), Fixed Priority Scheduling (FPS) and Rate Monotonic (RM), however other scheduling policies can be implemented in the same way.

Definition 4 (Partitions): A partition $P = <(T_1, .., T_m), S>$ *is* given by a set of tasks and a scheduling policy S.

In order to make our system design flexible and reconfigurable, we do not (statically) specify the identifier of the core to which the partition is assigned but rather it will be given by a mapping during the system instantiation. This feature will in fact be very practical for design space exploration, where different system configurations can easily be created and analyzed.

4.2. Platform Specification

A platform is given by a set of processing elements, a shared DRAM memory and a connector to share the access to DRAM. Each processing element contains a computation resource (core), a local cache memory L1 and the ARINC653 scheduler.

Definition 5 (Processing core): A core C is a transition system $<\{free, occupied\}, free, \emptyset, \rightarrow_C>$ consisting of two locations *free* and *occupied*, initially at location , and a transition relation between both locations with silent labels (internal actions).

Each processing core is equipped with a local cache, abstracted using the effective access time, and shared between a set of partitions according to the adopted ARINC653 schedule. A ARINC653 scheduler of a given core C specifies a cyclic scheduling of the partitions allocated to C. The schedule is given in terms of identical time windows, where a window is a sequence of partitions execution in a given order. Within a time window each partition is scheduled at least one time and can occur many times. Moreover, partitions are not equally served to use processing cores, in a way that each partition runs for a time slot that can be different from the others. Fig. 5 illustrates the scheduling of three partitions. One can see the allocation time slot differs from a partition to another, and from an occurrence to another for the same partition e.g., the first occurrence of Partition1 in the first window W1 lasts for 25 time units whereas the second occurrence with the same window lasts for 30 time units.

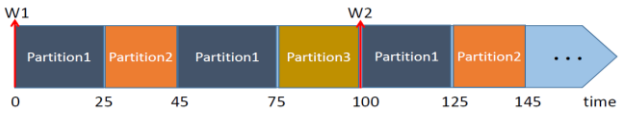

Fig. 5. Example of the ARINC653 scheduling windows.

We perform the ARINC653 scheduling of a core C, running a set of partitions $\boldsymbol{P}$, using a function *Sched* = (*turn, quantum*) given by:

$$turn : R_{\geq 0} \rightarrow \boldsymbol{P}$$

$$quantum : R_{\geq 0} \times \boldsymbol{P} \rightarrow R_{\geq 0}$$

Definition 6 (Processing element): Formally, a processing element $E = (C, \boldsymbol{P}, Sched, h)$ where C is a core processor and *Sched* is the ARINC653 scheduler to arbitrate the partitions $\boldsymbol{P}$ allocated to C. h is the time duration to access local cache L1, regardless of whether the access hits or misses.

In case an access to L1 misses, the underlying core will experience extra delays resulting from the interference to access the shared DRAM. Similarly to cache level L1, we abstract the shared DRAM using the worst case interference time (WCIT). Accordingly, each access request to DRAM will last exactly for WCIT. The access requests to DRAM are scheduled according to a First-in-First-out (FIFO) policy.

5. Compositional Analysis and Relocation

In this section we describe our compositional analysis technique to check the schedulability of each subsystem (processing element and the assigned workload). We also show how to use the feedback from a non-schedulable case, in terms of Uppaal counter-example traces, to relocate functions between the tasks of the faulty partition, so that potential schedulable system configuration can be derived. To be able to perform compositional analysis, we need first to calculate the worst case interference time cores can experience to access the shared DRAM memory.

5.1. WCIT Calculation

The worst case interference time WCIT to access a shared memory is strongly related to the interplay of the following factors:

- The architecture of the execution platform, where the higher the concurrency degree offered by parallel cores is the larger WCIT will be.
- The bandwidth of the shared memory where the slower is the larger WCIT will be.
- The nature of the underlying software functionality, where memory-intensive computations and the frequency of issuing access requests lead certainly to expensive WCIT.

In the literature, different works have been focusing on the calculation and bounding of interference time [5], [10], [7], [14], [16], by considering the internal architecture of DRAM in terms of banks and rows. Throughout this paper, we abstract the internal structure and architecture of DRAM when calculating WCIT, and only consider the real-time behavior of the DRAM connector. This will not impact our analysis technique as WCIT is a parameter of the analysis process we develop. In fact, the more accurate WCIT is the less over-approximating the analysis results will be. In case the internal architecture of DRAM has to be considered, this will not invalidate our analysis technique but rather WCIT will be calculated using appropriate technique accordingly.

Basically, the interference time includes the waiting time to grant an access to DRAM and the effective access duration. WCIT is then the maximum number obtained by any request, regardless of when is issued and which core does it. The effective time duration to access a shared memory DRAM is technically known by *alone-request-service-rate* (ARSR) [17]. ARSR is either provided by the DRAM constructor, or simplify estimated via a naive approach by preventing all cores to access DRAM during a time interval, except one core for which the DRAM request and access can be tracked using registers [31]. Thanks to our fine grained description of the tasks behavior, we can calculate how many access requests each core issues during any time interval. We calculate WCIT for the core to be analyzed using its ARINC653 schedules and the number of requests issued by the cores running in parallel. Let us assume C to be the core under analysis, W to be the size of each ARINC653 scheduling window of C, t is the shortest scheduling slot within W and R to be the maximum number of access requests performed by all cores during W. WCIT of C is calculated as follows:

$$WCIT\,(C) = R * ARSR + \lceil (R * ARSR)/t \rceil * W$$

The reason to consider the shortest time slot t is that the longest interference time most likely happens

when a partition runs for a very short time slot so that the requests issued during the current time slot will be granted in a future schedule of the concerned partition. To sum up, we calculate the interference time of a given core by considering its actual load and the worst case scenario of the rest of the system, in terms of the number of issued DRAM requests.

5.2. Compositional Schedulability Analysis

As introduced earlier, a subsystem is mainly given by a core, a set of partitions and the associated ARINC653 scheduler. Our compositional analysis technique consists in checking the schedulability of each subsystem individually while assuming that each access request to DRAM, performed by the core under analysis, lasts for WCIT. Such an analysis is performed using Uppaal model checker where it explores the state space, formed by all potential executions of the nested tasks, and checks that each task must be at a final (acceptance) location when its deadline is reached. This property must be held for each execution period of each task. The schedulability property is formally expressed using the following CTL query:

$$\forall[]\ not((clock[T_i] > T_i.d) \wedge (loc(T_i) \notin T_i.L_A))$$

where *clock* [T_i] is a clock tracking the runtime of each task T_i within each period, and *loc* is a variable tracking the current location of the task behavior. A clock *clock*[T_i] is reset on the release of each period of task T_i. Accordingly, a system is schedulable if each of its subsys- tems is schedulable. If a subsystem is not schedulable, this does not mean that the underlying system is non schedulable i.e., the system might be schedulable if it is analyzed at once where some of the access requests will not run for the entire WCIT. This classic fact results from the over-approximation related to compositionality.

5.3. Intra-Partition Relocation of Functions

In this section, we perform a partial design space exploration to relocate functions between tasks in case a given subsystem is not schedulable. When analyzing a subsystem Uppaal model checker generates counter-example traces demonstrating the non-schedulability, in case the subsystem is not schedulable. The analysis outcomes can tell us which task has missed a deadline, and how far the faulty task is from reaching a final state (in terms of how many functions left to execute). According to that, one can reduce the workload of such a faulty task by moving some functions from it to the tasks co-located in the same partition. However, to move a function from a task to another one needs to guarantee the execution frequency of the function to be moved. In others words, if a function executes every time interval *x* (which is the period of the holding task), a target task to which a relocation can be studied must have a period smaller or equal to *x*. This guarantees that the functionality associated to the function in question will be delivered at least once every time interval of the same length *x*. Moreover, we restrict the relocation to be intra-partition only to maintain the separation of concerns and fault containment imposed by IMA.

The basic intuition when studying the relocation of function from a faulty task is to target one of the tasks having a high laxity, i.e. the duration between the completion of the effective execution and deadline. A function can be moved if its workload fits in the available laxity of the target task. Hence, the response time of the faulty task will accordingly be shortened upon which the faulty task potentially meets its deadline. The load of a function *f*, belonging to a task *T* running on core *C*, is calculated using function *load*:

$$load(f) = \sum_{a_i \in f} (a_i.\pi + WCIT(C) * \mathcal{Z}(a_i))$$

$$\mathcal{Z}(a) = \begin{cases} 0 & if\ a.\alpha = \texttt{CacheOnly} \\ 1 & otherwise \end{cases}$$

The workloads of both faulty and target tasks are then re- calculated individually by accumulating for

each the workload of the incorporated functions, after relocation. The resulting workload of the faulty task will then be compared against its deadline, no matter of what is the scheduling policy used. If the workload is still exceeding the deadline, then we proceed with the relocation of a second function and so on until the resulting workload is perfectly tighter than the task deadline. On the other hand, if the target task workload reaches the deadline upon the relocation of a function, the former will be moved to another task. In a way that the functions of a faulty task can be spread out to different tasks.

The relocation operation is based on a binary search process where, before moving a function effectively, we perform a low cost analysis to estimate the new workload of both faulty and target tasks. If such an analysis is positive then we proceed to the relocation of the concerned functions to the target tasks and rerun the formal analysis of the underlying subsystem using model checking, without effecting the rest of the system. If the model checker outcome is negative, we can then rely on the new counter examples and start over. Two observations have been noticed when performing the relocation of functions:

- Even though the workload of a faulty task is reduced to be smaller than the deadline, it might happen that the task still misses its deadline. This is in fact related to the scheduling policy used at partition level where a faulty task can be assigned a low priority.

The order of functions after relocation plays a key role in the interference and the overall schedulability, in particular when the functions moved have a memory- intensive behavior. In fact, if a memory-intensive function is placed within a task in a way that it coincides with the execution of non memory-intensive functions running on the parallel cores, this would lead to shorter interference time due to less number of access requests issued in parallel. However, to profit this this observation one needs to know the functional loads of the parallel cores and potential dependencies between the functions.

6. Scalability and Discussion

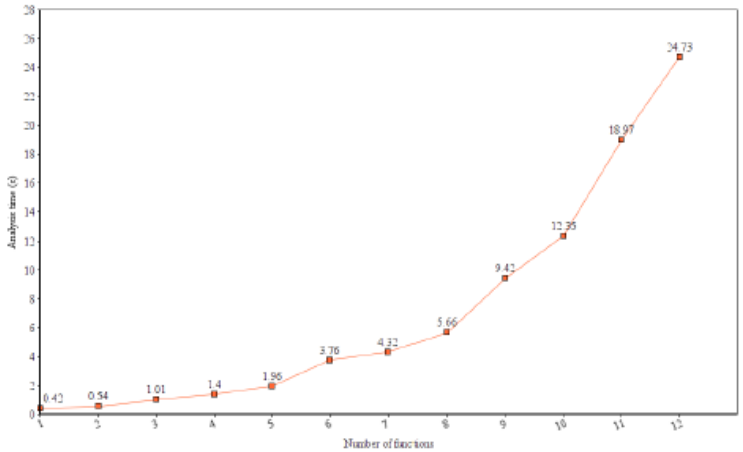

Fig. 6. Analysis time while varying the number of functions per task.

This section studies the scalability of our compositional schedulability analysis technique and compares it with regular schedulability analysis. Given the fine grained description of our system model, we start first by varying the number of ingredients, such as number of tasks and functions, in the system to be analyzed and run the schedulability analysis. Fig. 6 depicts the analysis time consumed by our analysis technique following the number of functions per task. The functions considered in this experiment are single-statement. One can see that the analysis time is linear to the number of functions. The second set of experiments we performed studies the scalability of our compositional analysis technique while varying the total number of tasks in the system to be analyzed. Fig. 7 depicts the resulting analysis time. In fact, the analysis time scales linearly with the number of tasks but with a tiny coefficient.

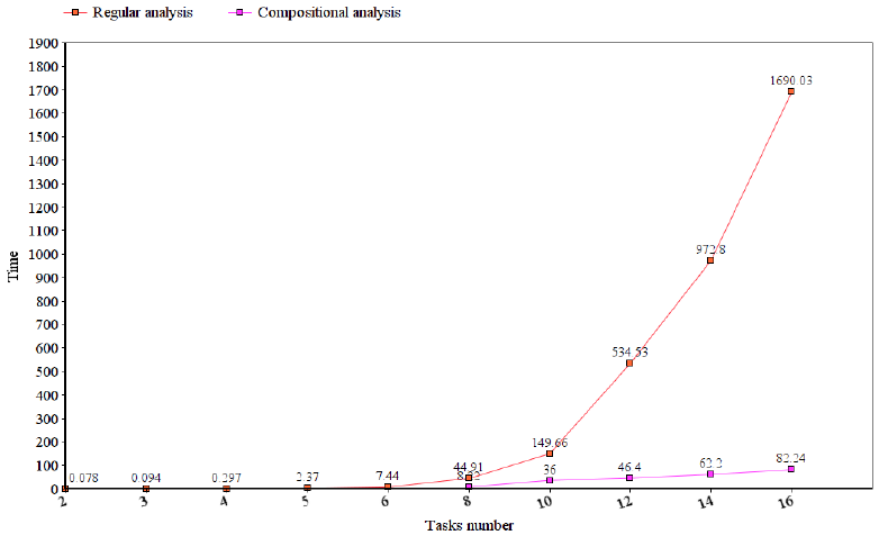

Fig. 7. Analysis time: regular vs compositional analysis.

7. Conclusion

This work introduced a formal setting for the specification and compositional schedulability analysis of modular avionic systems. We defined a fine grained specification ranging from atomic actions to preemptive real-time tasks to describe the system behavior. The system architecture is given by a set of ARINC653 partitions running on a multicore platform. Processing cores compete for the access to a shared DRAM. The schedulability of each core's workload is analyzed separately while assuming the worst case interference time that would result if the core under analysis runs alongside with the rest of existing cores. We provided a relocation technique to enable non- schedulable subsystems to be reconfigured to derive a potential schedulable configuration. The theory introduced in this paper has been mechanized using Uppaal.

References

[1] *Integrated Modular Avionics (IMA) Development Guidance and Certification Considerations*, 2005

[2] Annighöfer, B., & Thielecke, F. (2015). A systems architecting framework for optimal distributed integrated modular avionics architectures. *CEAS Aeronautical Journal, 6(3),* 485-496.

[3] Wilson, A., & Preyssler, T. (2008). Incremental certification and integrated modular avionics. *DASC 2008. IEEE/AIAA 27th,* 1.E.3–1–1.E.3–8.

[4] Löfwenmark, A., & Nadjm-Tehrani, S. (2015). Experience report: Memory accesses for avionic applications and operating systems on a multi-core platform. *ISSRE,* 153-160.

[5] Nowotsch, J., Paulitsch, M., Buhler, D., Theiling, H., Wegener, S., & Schmidt, M. (2014). Multi-core interference-sensitive WCET analysis leveraging runtime resource capacity enforcement. *ECRTS.*

[6] Tan, L. (2009). The worst-case execution time tool challenge. *International Journal on Software Tools for Technology Transfer,* 133-152.

[7] Kim, H., Niz, D., Andersson, B., Klein, M. H., Mutlu, O., & Rajkumar, A. (2014). Bounding memory interference delay in COTS-based multi-core systems. *RTAS'14,* 145-154.

[8] Putrycz, E., Woodside, M., & Wu. X. (2005). Performance techniques for COTS systems. *IEEE Software, 22(4),* 36-44.

[9] Chattopadhyay, S., Kee, C., Roychoudhury, A., Kelter, T., Marwedel, P., & Falk. (2012). A unified wcet analysis framework for multicore platforms. *RTAS'12,* 99-108.

[10] Boudjadar, J., Kim, J. H., & Nadjm-Tehrani, S. (2016). Performance-aware scheduling of multicore time-critical systems. *Proceedings of ACM/IEEE MEMOCODE* (pp. 105-114).

[11] Perathoner, S., Wandeler, E., Thiele, L., Hamann, A., Schliecker, S., Racu, R., Ernst, R., & Gonz´alez Harbour, M. (2009). Influence of different abstractions on the performance analysis of distributed hard real-time systems. *Design Automation for Embedded Systems, 13(1),* 27-49.

[12] Boudjadar, A., David, A., Kim, J. H., Larsen, K. G., Mikucionis, M., Nyman, U., & Skou, A. (2013). Hierarchical scheduling framework based on compositional analysis using uppaal. *FACS 2013.*

[13] Marlowe, T. J., Welch, I. R., Stoyenko, A., & Laplante. P. (1993). Real-time systems. *IEEE Real-Time Syst. Newsl, 9(1-2),* 85-92.

[14] Lv, M., Yi, W., Guan, N., & Yu, G. (2010). Combining abstract interpretation with model checking for timing analysis of multicore software. *Real-Time Systems Symposium,* 339-349.

[15] Carnevali, L., Pinzuti, A., & Vicario, E. (2013). Compositional verification for hierarchical scheduling of real-time systems. IEEE Trans. *Software Eng, 39(5),* 638-657.

[16] Yun, H., Pellizzoni, R., & Valsan, P. K. (2015). Parallelism-aware memory interference delay analysis for COTS multicore systems. *Proceedings of ECRTS*.

[17] Subramanian, L., Seshadri, V., Kim, Y., Jaiyen, B., & Mutlu., O. MISE providing performance predictability and improving fairness in shared main memory systems. *HPCA'13.*

[18] Madsen, J., Hansen, M. R., Knudsen, K. S., Nielsen, J. E., & Brekling, A. W. (2008). System-level verification of multi-core embedded systems using timed-automata. *Proceedings of IFAC.*

[19] Gustavsson, A., Ermedahl, A., Lisper, B., & Pettersson, P. (2010). Towards WCET analysis of multicore architectures using UPPAAL. *Proceedings of the 10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010).*

[20] Brekling, A., Hansen, M. R., & Madsen, J. (2008). Models and formal verification of multiprocessor system-on-chips. *The Journal of Logic and Algebraic Programming, 77,* 1-19.

[21] Panunzio, M., & Vardanega, T. (2014). An architectural approach with separation of concerns to address extra-functional requirements in the development of embedded real-time software systems. *Journal of Systems Architecture, 60(9)*, 770-781.

[22] AEEC Aeronautical Radio Inc. Avionics application software standard interface: Part 1 - required services. *ARINC specification 653.*

[23] Khlif, M., & Shawky, M. (2011). Functional-architectural diagnosability analysis of embedded architecture. *Proceedings of International IEEE Conference on Intelligent Transportation Systems (ITSC),* (pp. 469-476).

[24] Lee, R. Y. (2013). *Software Engineering: A Hands-On Approach.* Springer, 2013

[25] Boywitt, C. D., & Rummel, J. (2012). A diffusion model analysis of task interference effects in prospective memory. *Memory & Cognition, 40(1), 7*0-82.

[26] Regnier, P., Lima, G., Massa, E., Levin, G., & Brandt, S. (2011). Run: Optimal multiprocessor real-time scheduling via reduction to uniprocessor. *Real-Time Systems Symposium,* 104-115.

[27] Levin, G., Funk, S., Sadowski, C., Pye, I., & Brandt, S. Dp-fair: A simple model for understanding optimal multiprocessor scheduling. *Proceedings of the 22nd Euromicro Conference on Real-Time Systems* (pp. 3-13).

[28] Lee, J., Shin, K. G., Shin, I., & Easwaran, A. (2015). Composition of schedulability analyses for real-time multiprocessor systems. *IEEE Transactions on Computers, 64(4)*, 941-954.

[29] Ferdinand, C., & Wilhelm, R. (1999). Efficient and precise cache behavior prediction for real-timesystems. *Real-Time Syst*, *17(2-3),* 131-181.

[30] Alt, M., & Martin, F. (1995). Generation of efficient interprocedural analyzers with PAG. In A. Mycroft (Ed), *Static Analysis,* (pp. 33-50).

[31] Löfwenmark. A., & Nadjm-Tehrani, S. (2016). Understanding shared memory bank access interference

in multi-core avionics. *Proceedings of WCET'16, OpenAccess Series in Informatics (OASIcs).*

Jalil Boudjadar is an assistant professor at the Department of Engineering, Aarhus University Denmark. He received his M.Sc degree in June 2008 from Limoges University, and Ph.D degree in December 2012 from Toulouse University France. His research interests include software architectures, model-based design and formal verification of embedded real-time systems. Boudjadar's research aims to develop advanced architecture description and analysis techniques for embedded real-time systems.