A Software-Based Technique Enabling Composable Hierarchical Preemptive Scheduling for Time-Triggered Applications

Ashkan Beyranvand Nejad*, Anca Molnos†, and Kees Goossens‡
*A.BeyranvandNejad@tudelft.nl (Delft University of Technology, Delft, The Netherlands)
†Anca.Molnos@cea.fr (CEA-Leti, Grenoble, France)
‡K.G.W.Goossens@tue.nl (Eindhoven University of Technology, Eindhoven, The Netherlands)

***Abstract*—Many embedded real-time applications are typically time-triggered and preemptive schedulers are used to execute tasks of such applications. Orthogonally, composable partitioned embedded platforms use preemptive time-division multiplexing mechanism to isolate applications. Existing composable systems that support two-level scheduling are restricted to cooperative intra-application schedulers, and thus cannot execute the time-triggered applications. In this work, we introduce a framework that allows concurrent, composable execution of such applications on temporally-partitioned systems. The framework is composed of an execution platform and a method for timing analysis of applications running on the platform. The platform realizes a software-based timed-interrupt virtualization technique on an existing composable system. Multiple time-triggered applications may run concurrently using different intra-application preemptive scheduling policies, e.g., fixed-priority and rate-monotonic. The analysis method formalizes the available processing time for executing each application on a processor in order to enable schedulability tests for different policies. Finally, these concepts are demonstrated by executing a number of applications, first on an FPGA prototype and second on a Matlab simulation of the platform. The results indicate a composable and concurrent execution of multiple time-triggered applications using our proposed framework. Furthermore, the implementation of the technique has low cost in terms of memory footprint and execution overhead.**

***Index Terms*—Preemptive hierarchical scheduling, Time-triggered applications, Composability, Temporal partitioning, Embedded systems.**

I. Introduction

Nowadays the demand of executing multiple applications concurrently on embedded system-on-chip (SoC) platforms is increasing in various industries such as automotive and consumer electronics. These applications, typically have mixed time-criticality, i.e., each one has either firm, soft, or non real-time requirements. A firm real-time application must never miss a deadline, whereas a soft real-time one may occasionally miss a deadline, and non real-time ones have no timing constraints. A large group of such real-time applications are time-triggered (TT) meaning that they respond to periodic or aperiodic time events. For example, a control application sends appropriate commands to its actuators in response to periodic input data samples from its sensors.

Applications share SoC resources such as processors, interconnect, and memory blocks, in order to reduce cost. For even finer grained resource sharing, each application may consist of a number of concurrent tasks. Resource sharing makes the timing of the applications interdependent, which severely complicates system integration and verification [1] when the applications have mixed criticality. The state-of-the-art is therefore to execute applications in isolation. Temporal resource partitioning [2] and virtualization [3] are two approaches that isolate the execution of applications on a single platform, and consequently simplify their integration and verification. The goal of temporal partitioning is *composability*, which is the property that the (timing) behavior of an application is isolated from that of other applications. Often composability is used to denote that the *worst-case behavior* of an application is unaffected by other applications, enabling a compositional computation of worst-case bounds of an application. In this paper, following [4], [5], [6] we use a stricter definition where the *actual-case behavior* of each application is unaffected by other applications. The starting, finishing, and actual and worst-case execution times of (tasks of) an application are then independent of the other applications, allowing compositional performance verification even for mixed-critical systems. The scope of our work is to execute multiple applications on a shared processor. We assume that all other shared resources, e.g., interconnect [7] and shared memories [8], are arbitrated composably between applications.

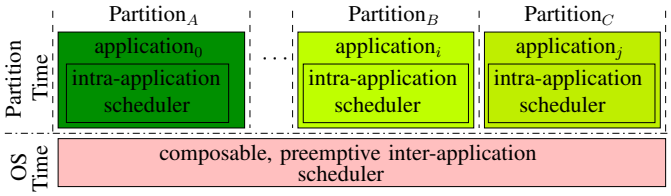


Fig. 1: Two-level scheduling of multiple applications in a composable system.

Schedulers arbitrate the access of the applications and their constituent tasks at each resource such that timing requirements are satisfied. Traditionally, in composable systems, scheduling is implemented at two hierarchical levels, namely inter- and intra-application, as depicted in Figure 1. The inter-application scheduler implements composable temporal partitions each of which is a set of time slots allocated to one application. The intra-application scheduler has to adopt a suitable scheduling policy (i.e., cooperative and preemptive) that fits the application's model of computation. The time-triggered applications must use preemptive schedulers. Thus, a composable system has to support preemptive inter-application scheduling to execute time-triggered applications, while the non-time-triggered or non-real-time applications in other par-

titions may use their own cooperative schedulers.

In existing composable systems, typically, only a privileged real-time operating system (RTOS) implements a preemptive inter-application scheduler to create a temporal partition for each application [9], and intra-application schedulers are co-operative. The reason is that for cooperative schedulers do not need to access critical resources such as timer interrupts that affect the system as a whole, compromising composability rather than just the application. Preemptive schedulers have to access such resources, and therefore, managing this access to not violate composability is crucial. For example, in Figure 1, if the scheduler of the application running in partition A sets an interrupt to occur in partition B's time, it would consequently interfere with the execution of the application running in partition B.

In this paper, we propose a framework that allows concurrent, composable execution of time-triggered applications using preemptive schedulers on temporally-partitioned embedded SoCs. The framework comprises two main parts, (i) the execution platform and (ii) the timing analysis method.

The execution platform is developed on top of an existing composable system [9]. Our contribution here is a software-based technique to virtualize a physical timer interrupt for preemptive inter- and intra-application schedulers. This technique is scalable compared to a hardware solution that provides one hardware timer for each application. We develop a system function and associated data structures on top of an existing light-weight composable RTOS. Whenever inter- or intra-application scheduling is performed, this function programs the timer with the appropriate value for the next timer interrupt. Moreover, this function calls a temporal conversion routine to map the real time value of the next timer interrupt to an actual virtual time in the corresponding partition.

The timing analysis method formalizes the time that is available for executing each application on a processor, and exemplifies how schedulability tests of the applications in the case of preemptive fixed-priority intra-application scheduler are performed.

To demonstrate these concepts, we first execute two real applications on an FPGA prototype of a composable SoC platform [9]. With this prototype, we present the real execution traces of the applications running on a Microblaze processor. The results indicate concurrent and composable execution of multiple time-triggered applications using our proposed framework. Second, we simulate our execution platform in Matlab and analyze how partition parameters affect applications' timing behaviors and schedulability.

The rest of this paper is organized as follows. Section II presents an overview of the related work. Section III provides background information needed by Section IV that presents our approach to implement hierarchical preemptive scheduling. We explain the implementation details of our technique in Section V followed by formalization and discussion on the system and applications timing analysis in Section VI. The case studies are presented in Section VII. Finally, Section VIII concludes this paper.

II. Related Work

A large body of work has been published on scheduling and executing time-triggered real-time applications on embedded systems [10], [11]. To ensure that the timing requirements of applications are met some approaches do not allow sharing resources at inter- and intra-application levels [10], [12], [13]. In this section we discuss existing approaches that include resource sharing and timing constraints.

Existing approaches employ some form of isolation between applications to guarantee that the system meets a certain timing requirements when resources are shared. In this case, interference between the applications invalidates the timing requirements. Isolation mechanisms have been proposed at different levels, from full separation of applications' critical resources [12], to cycle-level isolation in shared (critical) resources. Such approaches can be split in two main categories: those that implement a level of resource partitioning and those that virtualize the shared resources.

Resource partitioning is defined and implemented at different levels in various domains [2]. For example, the ARINC standard is a specification for time and space partitioning in safety-critical avionics real-time operating systems [14]. According to ARINC, time is divided into number of slices denoted as partitions, and every partition is assigned an application. Other examples that employ temporal resource partitioning are presented in [15] and [16]. They execute multiple non-time-triggered applications concurrently, ensuring that they are not affecting each other's worst-case timing behavior. The authors of [15] utilize two-level scheduling, TDM inter-application and cooperative static-order intra-application. The work in [16] analyzes applications individually to enable reasoning about their overall worst-case behavior when executing them on the same platform. These works offer composability of worst-case bounds of applications, which makes them suitable for systems containing only firm-real-time applications. Following paradigm introduced in [9], our definition of composability is more restrictive than other resource partitioning approaches, in that inter-application interference is completely prohibited at cycle-level. Thus the actual-case performance of applications is composable, enabling mixed-criticality applications to be designed, verified, integrated (without re-verification), and executed on the same platform.

Furthermore, virtualization has received a lot of attention recently from the embedded systems community [17]. Typically, in a virtual platform, a hypervisor manages a number of virtual machines (VMs) that run guest (RT)OSs. The scheduler of virtual machines may follow different policies and it is not fully decoupled from the scheduler inside the guest OSs. Specially in case of real-time OSs, the two scheduling levels cooperate with each other to ensure the timing requirements of the time-critical applications are met [18]. In our composable approach, the function of the hypervisor is realized by an RTOS that implements inter-application scheduling, while instead of guest OSs, the applications directly schedule their own tasks. This approach suits embedded systems as it avoids

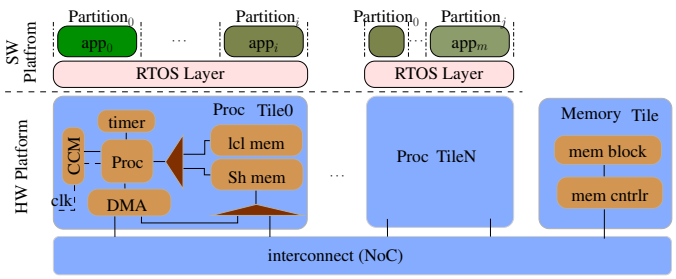


Fig. 2: A template for composable embedded SoC platform.

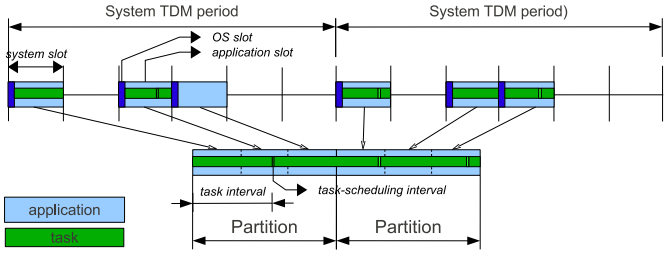


Fig. 3: Temporal partitioning of a processor by a TDM scheduler.

overhead of having guest OSs in terms of execution time and memory footprint, in the cost of losing high-level services that an OS could provide to applications. Moreover, many other VM-based embedded systems exist, especially in the automotive domain [3], [19], [11]. Among all, the approach in [3] is the closest to our technique as it also targets time-triggered applications. It implements two-level scheduling, where in the first level, the hypervisor allocates a single time slot to every virtual machine, and at the second level, inside each virtual machine, a guest RTOS may use a preemptive scheduler. To the best of our knowledge, none of the existing work offer a fully composable platform (by our definition of composability) that executes time-triggered applications.

Moreover, some existing approaches, such as uC-OS/II, multiplex multiple timed events on a single tick generator, i.e., a periodic interrupt timer. They offer software-based virtual timers to time-triggered applications [20], [21]. In these approaches, the granularity of the virtual timed interrupts is restricted to the actual RTOS ticks. Whereas in our approach, we virtualize a single timer interrupt that is used by each application. When it has access to the physical timer that access is exclusive. Every application can set its own timed events independent of the other applications, and the granularity of the events is the same as the system clock ticks. Our RTOS takes care that events of an application are always mapped to its temporal partitions, and never interfere with other partitions (and hence behavior) of other applications.

III. Background

In this section we provide background information on the time-triggered applications and the platform that we consider.

A. Time-Triggered Applications

In a time-triggered application, the tasks are periodically or aperiodically triggered by a timer interrupt. Such application, Γ, is represented by a set of tasks as $\Gamma \equiv \{\tau_i(P_i, L_i, C_i, D_i, R_i)\}$, where P_i is the period in which the task τ_i executes for C_i time-units. The task is ready to execute at its release time which is at L_i time-units from the beginning of its period, and it has to finish its entire work before its relative (to the beginning of the period) deadline D_i time-units. The task's priority R_i is set either statically or dynamically. Smaller values of R_i correspond to higher priority for tasks. To execute a task τ_i for C_i time-units and to finish before its deadline D_i, the following condition must hold true: $(C_i + L_i) \leq D_i \leq P_i$.

In this paper we consider fixed-priority (FP) policy to schedule the tasks inside an application. Such scheduler makes sure that, at any moment in time, the task with highest priority among all the released ones executes. For the sake of simplicity, we consider all the tasks are released at the beginning of their periods, i.e., $L_i = 0$, and the deadline of each task is equal to its period, i.e., $D_i = P_i$.

B. Target Platform

The SoC platform consists of a hardware infrastructure and a software infrastructure. Basically, composability is a property of the platform, and the hardware and the software infrastructures implement this property.

1) Hardware platform: The hardware infrastructure is based on the *CompSOC* template proposed in [9] and consists of a number of processor and memory tiles communicating via a network-on-chip, as depicted in Figure 2. All the shared resources are arbitrated in such a way that they are virtualized to achieve application isolation and therefore system composability. Since the focus of this paper is on the processor scheduling, in this section, we explain only the architecture of the processor tile that is relevant for this purpose.

A processor tile consists of a processor, a timer, a clock control module (CCM), set of local memory blocks, shared memory blocks, and Direct Memory Access (DMA) modules. The timer keeps track of the system time as the reference of the real, physical time. The timer is the only block that can issue an interrupt to the processor. In the existing platform, this interrupt is only used by the inter-application scheduler to implement partitions for every application.

2) Software Platform: The software executing on the platform has two layers, namely the application and the RTOS layer, as in Figure 2. Multiple applications are implemented at the application layer, and the RTOS layer provides essential infrastructure to execute the applications and an application programming interface (API) to access the hardware resources. For a more detailed discussion on the base-structure of the RTOS we refer to [4]. Here we assume that the applications and tasks mapping on the processor tiles are given.

The RTOS realizes cycle-level temporal isolation between the applications by creating temporal partitions and assigns each application to one partition, and consequently make

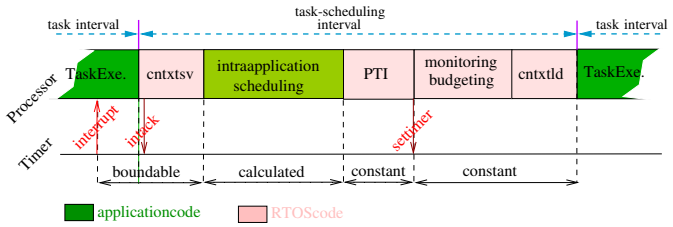


Fig. 4: The general operational time-line of an application in a task-scheduling interval.

the system composable at application level. A partition is a set of time slots, namely *system slot*s, which are fixed time quanta and implemented by a time-division multiplexing (TDM) scheduler, as illustrated in Figure 3. A system slot is split into an operating system (OS) and an application slot. All the slots have to have a fixed duration. The system slots start with timer interrupts issued at fixed intervals. An OS slot always takes a constant time duration, and subsequently the application slots have also constant duration till the next system interrupt. The RTOS code executes in OS slots, and it is the only trusted code that is allowed to access the timer for setting system interrupts.

A partition time is a logically continuous time in which an application's tasks, communication, and the intra-application scheduler execute. A fraction of a partition time in which a task executes, is denoted as *task interval*, and the time between two consecutive task intervals, in which the intra-application scheduler executes, is denoted as *task-scheduling interval*, as illustrated in Figure 3. In a time-triggered application, the task intervals must be implemented by timer interrupts that are set by the intra-application scheduler in every task-scheduling interval.

IV. Preemptive Intra-Application Scheduling

In this section, we propose a technique to virtualize a physical timer for both preemptive inter- and intra-application schedulers. The existing composable platforms [22] are restricted to support such cooperative scheduling. Basically, in such systems, the timer interrupt is only used by the TDM inter-application scheduler. However, to implement preemptive intra-application scheduling, we propose that the RTOS provides the intra-application scheduler with the infrastructure to program the timer interrupts. This infrastructure preserves composability, meaning that the intra-application scheduler is not allowed to alter the timing properties of the application slots, i.e., start and end time of the slots.

Generally five operations are performed in each task-scheduling interval, in order, (i) save the context of the preempted task, (ii) trigger the intra-application scheduler, (iii) program the timer with task interrupt value for the next task interval, (iv) update the monitoring and budgeting information of the current application and task, and (v) load the context of the new task.

Figure 4 shows the task-scheduling and task intervals of a time-triggered application in its application slot. When entering the task-scheduling interval, the processor automatically disables the interrupts till the end of the interval to prevent handling nested interrupts that may cause losing the context of the current task or the scheduler. After a new task is scheduled, the application calls a *program timer interrupt* (PTI) function. This function is implemented as a library in the RTOS to calculate and set the next timer interrupt for the scheduled task in the current partition time. The PTI function safely accesses the system timer and the RTOS timing information values.

There are logically two virtual timers active in each partition. One that keeps track of the current application slot, and the other that implements the task intervals. The first timer issues *system interrupt*s, and the second one issues *task interrupt*s. We virtualize one physical timer interrupt in order to implement these two logical interrupts. This technique is scalable compared to a hardware solution that provides one hardware timer for each application, because all the preemptive intra-application schedulers together with inter-application scheduler use the only hardware timer to implement both task intervals in partitions and system slots. The PTI function distinguishes between the logical interrupts and program the physical timer with the appropriate actual value of the next earliest interrupt.

V. Implementation

This section presents the detailed implementation of virtualizing the system and applications interrupts. The main idea behind this technique is that the PTI function updates all the logically active timer interrupts and program the physical timer with the value of the earliest one. The earliest interrupt may be either a system interrupt, or a task interrupt of the running application. Between these two logical interrupts, precedence is given to system (inter-application scheduler) interrupts, such that they always occur at fixed moments in time and the system composability is not invalidated. The main challenges here are to accurately calculate the moment when the timer interrupts should occur, and to keep track of task-scheduling intervals so that the moment when each application slot starts remains unchanged.

There is one instance of an operating system control block (OCB) per processor. The application control blocks (ACBs) and the task control blocks (TCBs) are created and initialized for each partition by the RTOS before they start executing on the processor. The size of the OS slot and the application slot, denoted as `os_slot` and `app_slot`, respectively, are part of the OCB. Application slots are fixed at run-time for all applications. Slot sizes are parameters of the RTOS instance running on a processor.

The task interrupts of a time-triggered application are either periodic or aperiodic in a partition, hence the interrupts are not perfectly aligned with system interrupts. As presented in Table I, the RTOS control blocks are extended with additional data structures that are required by the PTI function to calculate the time of next interrupts as follows:

- the size of the task interval is given and denoted as `task_interval`.

TABLE I: Additional data structure.

SW Control Blocks	Data Structure
OCB	int os_slot int app_slot int app_slot_left
ACB	int task_scheduling_interval int task_scheduling_interval_left
TCB	int task_interval int task_interval_left

- if a fraction of the scheduled task has already been executed and preempted before, the remaining time of the task interval is calculated and denoted as `task_interval_left`.
- when the current task interval is finished, the remaining time of the application slot is calculated and denoted as `app_slot_left`.
- the duration of the current task-scheduling interval is calculated and denoted as `task_scheduling_interval`, and if a system interrupt is supposed to occur in the middle of the task-scheduling interval, it does not preempt the scheduling operations, however, the duration between the moment of the system interrupt until the end of the task-scheduling interval is denoted as `task_scheduling_interval_left` to be considered at the beginning of the next application slot that belongs to this application.

The duration of task-scheduling interval is variable. However, as illustrated in Figure 4, all the operations in this interval, after intra-application scheduler, are implemented to have constant execution time. The PTI function therefore calculates the current `task_scheduling_interval` by adding the constant execution time to the time that has elapsed from the moment of the last task interrupt. Moreover, the task interval, `task_interval`, for each task is given by the intra-application scheduler to be used for calculating the next task interrupt. The `task_interval` however is in real world time and needs to be converted to a virtual moment inside the partition of the current application. The PTI function calls `conv_rt_to_pt` function that implements this conversion which directly depends on the partition parameters and the availability of the processor to the partition in the given `task_interval`.

Algorithm 1 details the implementation of PTI function using the data structures presented in Table I. The PTI function calculates the new value of `app_slot_left` by deducting the sum of the next `task_interval` and the current `task_scheduling_interval` from the remaining application slot duration. According to the value of `app_slot_left`, three different scenarios for the next interrupt are possible, as illustrated in Figure 5. Following one of these scenarios, as presented in Algorithm 1, the PTI function sets the timer with the proper value for the earliest interrupt in the future.

In the first scenario, after *c_int* interrupt has occurred, the task-scheduling interval can finish its entire operations to schedule a new task and set the next timer interrupt *n_int* at the end of the task interval which is before the periodic system interrupt. This is because the `app_slot_left` is greater than the current `task_scheduling_interval` plus the `task_interval`.

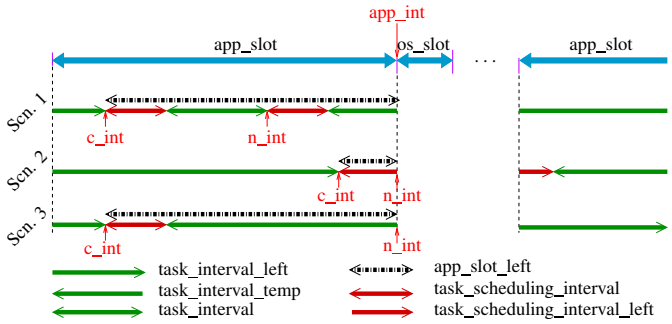


Fig. 5: Three possible scenarios of task interrupt in a partition time.

Algorithm 1: Pseudo code representation of PTI function.

Input: OCB, current scheduled ACB and TCB.
Output: Sets proper timer interrupt value, and updates all the data structures presented in Table I.

```
task_scheduling_interval = cal_task_sched_interval();
task_interval_left_v = conv_rt_to_pt(task_interval_left);
app_slot_left -= (task_scheduling_interval +
task_interval_left_v);
if (app_slot_left > 0) then
    /* Scenario 1                                    */
    task_interval_temp = task_interval_left_v;
    task_interval_left = task_interval;
    task_scheduling_interval_left = 0;
else if (app_slot_left ≤ 0) then
    task_interval_temp = task_interval_left_v + app_slot_left;
    if (task_interval_temp < 0) then
        /* Scenario 2                                */
        task_interval_temp = 0;
        task_scheduling_interval_left = -task_interval_temp;
    else
        /* Scenario 3                                */
        task_scheduling_interval_left = 0;
        task_interval_left = task_interval_left -
        task_interval_temp;
    end
    app_slot_left = app_slot;
end
/* Set the next timer interrupt.                     */
set_timer_interrupt(task_interval_temp);
```

In the second scenario, after *c_int* interrupt has occurred, the task-scheduling interval cannot finish its entire operations before the next interrupt *n_int* which is the periodic system interrupt. This is because the `app_slot_left` is less than the current `task_scheduling_interval`. In this case, we let the task-scheduling operations to be finished, and therefore the task-scheduling interval extends over the OS slot. The OS slot has fixed size and it is designed in such a way that it can absorb the worst case duration of the task-scheduling intervals for all the running applications and its size remains unchanged. The extended time of the interval over the OS slot, or in other words, the

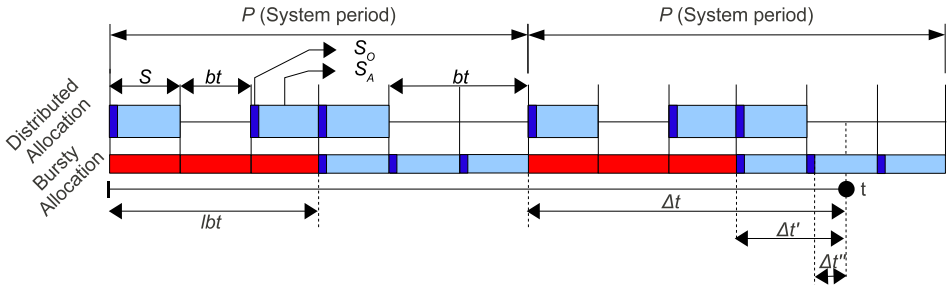


Fig. 6: An example of slots allocation to an application in a temporally-partitioned system, illustrating the cumulative available processing time and the (longest) blocking time of the application.

remaining time of the interval from the current application slot, is calculated in `task_scheduling_interval_left`. This remaining time is implemented at the beginning of the next application slot so that the application does not experience a shorter task-scheduling interval. If *c_int* happens exactly at the moment of the periodic system interrupt, then the entire task-scheduling interval is considered as `task_scheduling_interval_left` and implemented in the next application slot that belongs to this application.

In the third scenario, when the current interrupt *c_int* occurs, the task-scheduling interval can finish its entire operations, but the scheduled task cannot finish its entire execution in the current slot before the next interrupt *n_int* which is the periodic system interrupt. This is because the `app_slot_left` is less than the current `task_scheduling_interval` plus the `task_interval`. The remaining task time, `task_interval_left`, is therefore implemented in the next slot that belongs to this application.

Summary, the proposed software-based technique implements the PTI function that manages the co-existence of system interrupts and task interrupts of time-triggered applications in each partition so that the timing properties of application slots remain unchanged and the system composability is preserved.

VI. Timing Analysis

Timing analysis of a real-time system is based on *response time*, rt, of running applications. The response time of a time-triggered application is defined per task as the duration between the task's release time and the task's finishing time. The task τ_i is schedulable on a system, if and only if $rt_i \leq D_i$ in all situations. This condition states that the response time of a task should be always less than or equal to its relative deadline. When a processor is shared by multiple tasks, the response time of a task may increase if there is any higher priority task that is ready to execute. In the case that the processor is further shared between multiple applications, such as in our platform, the response time of all tasks of an application increase because they are blocked during the slots of other applications. Besides this, the relative deadline of the tasks may be logically moved earlier if the exact moment of a deadline is in the blocking time of the application. In this case, the task should finish before the end of the last allocated slot to the application in order to meet its deadline. As a result, in such systems, the response time analysis takes into account also the allocation of system slots to applications (i.e., partition parameters), and consequently the condition of $(rt \leq D)$ becomes tighter.

A number of different approaches that address these problems exist in the literature. Various analysis methods for so called temporal-partitioning systems are proposed [2], [23], [24], [25]. These methods are not directly applicable to our platform, since in contrary to all the existing systems, in our case, composability implies a fixed operating system slot overhead before every application slot. This motivates us to propose, in this section, a response time analysis of time-triggered applications running on a composable platform. Using this analysis method, application developers may follow two design options:

1) (re-)design (legacy) applications to be schedulable on the composable system.
2) adjust the partition parameters, i.e., slot allocations and slot sizes, so that the available (legacy) applications, e.g., electronic stability control, would be schedulable.

We first formalize the timing properties of the composable platform. As an example, a TDM inter-application schedule with the system period of P time-units and $M = 6$ number of slots is presented in Figure 6. The size of each slot, which consists of an application slot, S_A, plus the operating system slot, S_O, is $S = \frac{P}{M}$ time-units. Application j has a partition with a set of $m_j = 3$ distributed slots (blue blocks) in the system period.

The cumulative processor time that is exclusively available for the execution of application j until a time moment, t, depends directly on the size of its partition. This cumulative time corresponds to sum of all the allocated slots to the application j until time t, e.g., the blue slots in our example. The function that calculates this cumulative time is called *server characteristic* function in [26], *server supply* function in [23], *resource supply bound* function in [27], and *availability* function in [25]. Here, we also use the term of availability function and define it for an application j as follows.

$$A_j(t) = \left\lfloor \frac{t}{P} \right\rfloor \times A_j(P) + A_j(\Delta t) \qquad (1)$$

where, $A_j(P) = m_j \times S_A$ is the cumulative available time for the application j in a system period and, $\Delta t = t - P \times \lfloor \frac{t}{P} \rfloor$ is the time duration left until t in the last system period as depicted in Figure 6. $A_j(\Delta t)$ depends on how the system slots are allocated to the application j. In other words, it depends

on when the application is blocked by the execution of other applications. According to existing theorem [2], when the processor allocation to an application is distributed, schedulability should be tested for every task at all the critical instances, i.e., the time when all the tasks are released at the beginning of a blocking time. Here, we propose a *conservative* approach which considers the longest blocking time (lbt) instance of an allocation. This is the case when all the slots that are not allocated to the application are consecutive at the beginning of the system period, as illustrated in Figure 6. In other words, the longest blocking time instance is when the allocation of slots to an application is bursty. Given such an allocation, the longest blocking time for an application j in a period is calculated with the following equation.

$$lbt_j = P \times \left(1 - \frac{m_j}{M}\right) \tag{2}$$

Therefore, the lower bound on $A_j(\Delta t)$ is defined as $A^l_j(\Delta t)$ and it is calculated as follows.

$$A^l_j(\Delta t) = \begin{cases} A^l_j(\Delta t') & \Delta t > lbt_j \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

where, $\Delta t' = (\Delta t - lbt_j)$, and $A^l_j(\Delta t')$ is calculated as follows.

$$\Delta t'' = (\Delta t' - S \times \left\lfloor \frac{\Delta t'}{S} \right\rfloor), \text{and } S'_A = \left\lfloor \frac{\Delta t'}{S} \right\rfloor \times S_A$$

$$A^l_j(\Delta t') = \begin{cases} S'_A & \Delta t'' \leq S_O \\ S'_A + (\Delta t'' - S_O) & \Delta t'' > S_O \end{cases} \tag{4}$$

Finally, using the longest blocking time of an application, a lower bound on the availability function is given with the following equation.

$$A^l_j(t) = \left\lfloor \frac{t}{P} \right\rfloor \times A_j(P) + A^l_j(\Delta t) \tag{5}$$

This lower bound on the availability function may be used to analyze the timing properties of time-triggered applications and to perform the schedulability analysis for different task scheduling policies. As an example, here, we propose a schedulability analysis for the cases when the preemptive fixed-priority is used by task schedulers.

The traditional response time analysis method of real-time applications using the preemptive fixed-priority scheduling exists for a long time [28]. Following this method, for our platform, the worst-case response time is calculated as follows.

$$wcrt^n_{ji} = C_i + \sum_{k=1}^{i-1}\left(\left\lceil \frac{wcrt^{n-1}_{ji}}{P_k} \right\rceil \times C_k\right) + \left(\left\lceil \frac{wcrt^{n-1}_{ji}}{P_S} \right\rceil \times lbt_j \times \frac{S_A}{S}\right) + \left(\left\lceil \frac{wcrt^{n-1}_{ji}}{S} \right\rceil \times S_o\right) \tag{6}$$

In this equation, intuitively, the worst-case response time of a task is the sum of its workload and the worst-case delay that it may experience during its execution. The first term, C_i, is the task's workload; the second term is the delay due to executing other tasks of the same application with higher priority; the third term is the delay due to the longest blocking times at the beginning of every system period; and the fourth term is the delay that the task experience due to OS slots. The third and fourth terms are the result of temporal partitioning. The equation can be solved iteratively starting with $wcrt^0_{ji} = C_i$ and stopping when $wcrt^n_{ji} = wcrt^{n-1}_{ji}$ [28]. Finally, An application would be schedulable with fixed priority scheduler if and only if, for all its tasks, the condition of $A^l_j(wcrt_{ji}) < A^l_j(D_{ji})$ holds true.

VII. Case Studies

In this section, we present two case studies demonstrating:

1) concurrent, composable execution of multiple time-triggered applications, using an FPGA prototype of the platform.
2) timing behavior of applications with different partition sizes, using a temporal simulation of the platform in Matlab.

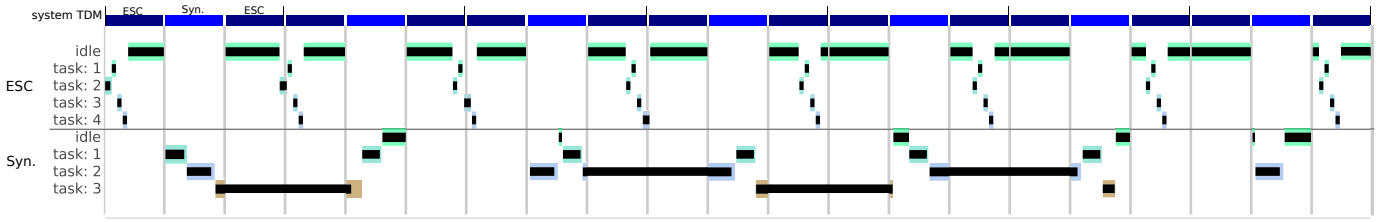
At the end, we report the implementation cost of our technique.

A. FPGA prototype

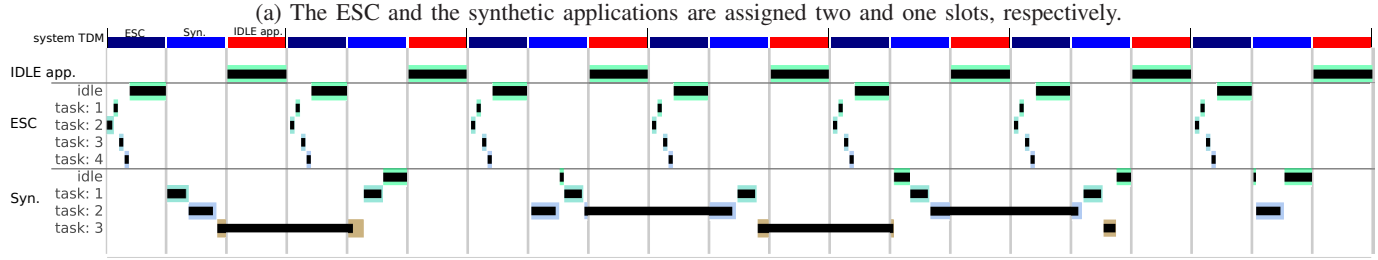
A VHDL implementation of the CompSOC platform running with clock frequency of 120 MHz is prototyped on a Xilinx ML605 FPGA board. We applied our technique to schedule two applications with fixed-priority policy on a processor tile. The first application is a workload of an electronic stability control (ESC) application having four periodic tasks, and the second one is a synthetic application having three periodic tasks.

Figure 7 illustrates the schedule traces of two different runs measured on the FPGA prototype with the inter-application TDM and intra-application fixed-priority scheduling of the applications. The TDM period of the inter-application scheduler (i.e., system TDM) has 3 slots. In the first run, the ESC and the synthetic applications are assigned two and one slots, respectively, as illustrated in Figure 7a. In the second run, each application is assigned one slot, and the last slot is allocated to the *idle* application, as illustrated in Figure 7b. In the figures, the vertical gray bars represent the OS slots, and between each two consecutive bars is an application slot. The colored boxes illustrate the task-intervals, and the black boxes illustrate the duration between the start time and finish time of the tasks.

The tasks of the ESC application are scheduled in its slots fully independent of the synthetic application's tasks. In the run illustrated in Figure 7a, the longest blocking time for the ESC application is one slot and for the synthetic application is two-slot, i.e., the ones that are allocated to the ESC application. While, in Figure 7b, the longest blocking time for the both application is two-slot, namely, the one slot that is allocated to the application itself, plus the slot of the idle application. The workload of ESC tasks is much less than an application slot size and all the tasks could always finish in the same slots that they start in. This is not the case for the tasks of the synthetic application. For example, the first execution of the task 2 starts in the second slot and it finishes



(a) The ESC and the synthetic applications are assigned two and one slots, respectively.



(b) The ESC and the synthetic applications are each assigned one slot.

Fig. 7: Schedule trace of the applications running on an FPGA prototype.

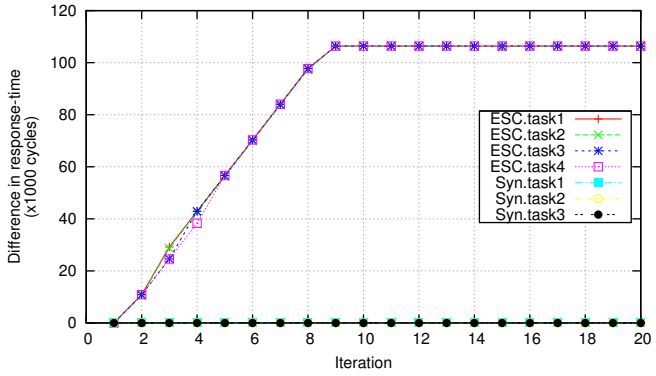


Fig. 8: The difference between response-time of the tasks in two runs, where the processor allocation to ESC application is changed.

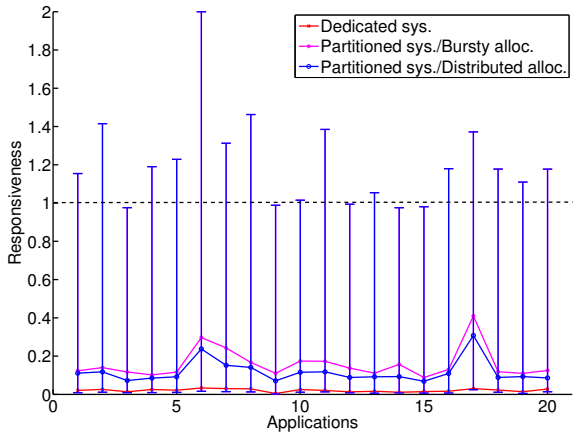


Fig. 9: Average responsiveness of randomly generated applications.

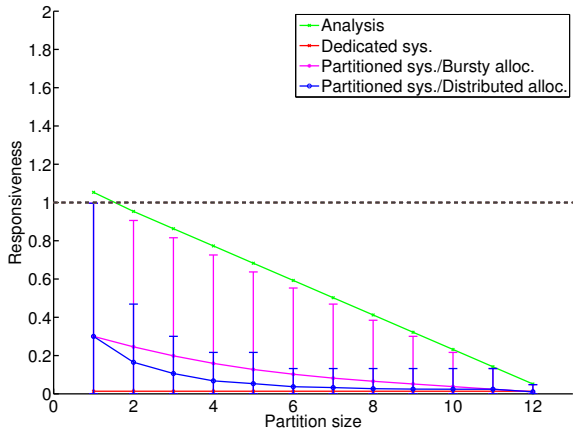
in the fifth slot, and consequently, the response time of the task is more than the case that it runs on a dedicated system.

To illustrate the composability of the system, we compare the response time of the tasks in the two runs, as illustrated in Figure 8. The response times of ESC tasks are increased in the second run because the ESC application is allocated one less slot than the first run. As it is also expected from Eq. 2, the longest blocking time of the ESC application in the second run is greater than in the first run, and therefore, its tasks start executing later (farther to its release time) and subsequently finish later. In spite of the variations in the timing behavior of the ESC tasks in these two runs, the response times of the synthetic tasks are remained unchanged. This observation shows that the timing properties of an application are not affected when the behavior of other applications is modified, and therefore, the applications are independent and the system is composable.

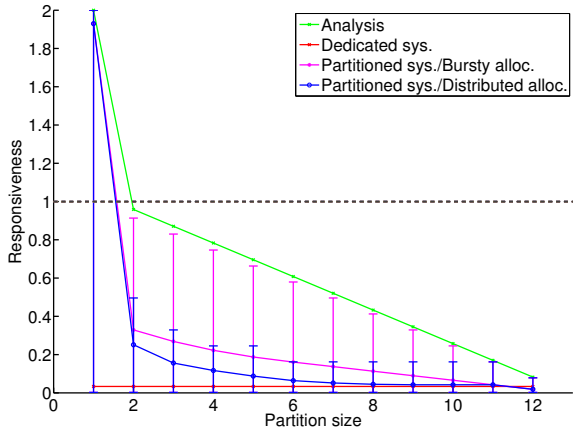
B. Matlab Simulation

Based on the availability function introduced in Eq. 1, an instance of the platform is implemented in Matlab to simulate the timing behavior of a number of applications. For this purpose, 20 applications each consists of five tasks are generated randomly with a modified version (to generate random periodic tasks) of TORSCHE toolbox [29]. Each application is schedulable on a dedicated system in which the application owns the processor exclusively. Here, we consider a system TDM with period of 12 slots. Starting with the minimum partition size of one slot, in each run, we increase the size of the partition with one slot resulting in performing 12 different runs.

To investigate the variations in timing behavior, we define the *responsiveness* of a task as $(\frac{\text{response-time}}{period})$. This gives us a normalized metric to compare the timing behavior of the tasks in our randomly generated applications with different timing properties, i.e, deadline and period. When the responsiveness



(a) Application 3.



(b) Application 6.

Fig. 10: Average responsiveness of two randomly generated applications.

of a task is greater than one, the task has already missed its deadline.

Figure 9 illustrates the responsiveness of the applications in three different cases. The first case demonstrates the average of the worst-case responsiveness of the tasks in each application when executing on a dedicated system. For the other two cases we present the min-max bar and the average of the tasks' responsiveness in each application when executing on a partitioned system in the 12 different runs. In the second case the slot allocations to the partitions are bursty, while in the third case the allocations are as evenly distributed as possible. As expected, the results indicate that the average responsiveness of the tasks for all applications in both partitioned scenarios are always higher than the dedicated system scenario. The responsiveness in bursty allocations is also higher than the distributed allocations, since the applications experience longer bursty blocking time. Finally, we can observe that in this set of applications, 80% of the applications could not meet their timing requirements on the partitioned system regardless of their partition size as the maximum responsiveness of (at least one of) their tasks (is) are greater than one.

Figure 10 illustrates the responsiveness of the tasks of two applications, namely Application 3 & 6, when executing with different partition sizes. According to the Matlab simulation results, Application 3 is schedulable on the partitioned system as the maximum responsiveness of the its tasks are smaller than one in all runs, as illustrated in Figure 10a. While, Application 6 is not schedulable in the run with the partition size of one, as illustrated in Figure 10b. However, using Eq. 6, the results of the worst-case responsiveness analysis of the both applications indicate that none of them are schedulable with the smallest partition size. In Figure 10, the differences between the maximum responsiveness results of bursty allocations obtained from the Matlab simulation and the analysis results are because of not long enough simulation time during which the worst-case blocking situation between the tasks of the application could not occur. Moreover, as expected, by increasing the size of the applications' partitions the average responsiveness of the tasks is decreased.

C. Implementation Cost

The implementation cost consist of run-time overhead and memory footprint. In our experiments, the maximum task-scheduling interval takes 2000 cycles, during which the execution time of PTI function is 200 cycles, and the OS overhead per slot, i.e., OS slot size, is 4600 cycles. These mean that the task-scheduling interval corresponds to less than 17 microseconds, PTI function executes for less than 2 microseconds, and the OS overhead is 38.24 microseconds, running at 120 MHz,. The application slot is set to 100000 cycles which takes 833 microseconds. Practically, these overheads are acceptable, as a significant number of real-time applications typically require response time in the order of milliseconds.

The memory footprint of our light weight RTOS is around 13 KByte, on top up which, our technique only adds a very small implementation overhead of about 300 Byte for the PTI function. The additional data structures cost $(12+(m+n)*8)$ bytes, where m is the number of applications and n is the total number of tasks. In our experiments, this overhead consists of 84 bytes in the local data memory of the tile.

VIII. Conclusions

In this paper we proposed a framework that allows concurrent, composable execution of time-triggered applications. The framework is composed of a software-based technique that enables the platform to implement two-level preemptive inter- and intra-application schedulers, and a timing analysis method to perform schedulability test. These concepts are demonstrated by two case studies: (1) to execute multiple time-triggered applications concurrently, using an FPGA prototype; (2) to simulate a number of randomly generated applications in Matlab to investigate the affect of temporal partitioning on the timing behavior of the applications. The experiments indicate that the applications are independent and the system is composable, where the implementation is low cost.

Acknowledgements

This work was partially funded by projects EU FP7 288008 T-CREST and 288248 Flextiles, Catrene CA104 Cobra and CA505 BENEFIC, CA703 OpenES, and NL STW 10346 NEST.

REFERENCES

[1] H. Kopetz, *Real-Time Systems: Design Principles for Distributed Embedded Applications*, 2011.

[2] A. K. Mok, X. A. Feng, and D. Chen, "Resource partition for real-time systems," in *Proc. of RTAS*, 2001.

[3] T. Kerstan, D. Baldin, and S. Groesbrink, "Full virtualization of real-time systems by temporal partitioning," in *Proc. of OSPERT*, 2010.

[4] A. Hansson, M. Ekerhult, A. Molnos, A. Milutinovic, A. Nelson, J. Ambrose, and K. Goossens, "Design and implementation of an operating system for composable processor sharing," *Microprocessors and Microsystems*, 2011.

[5] A. Molnos, A. B. Nejad, B. T. Nguyen, S. Cotofana, and K. Goossens, "Decoupled inter- and intra-application scheduling for composable and robust embedded MPSoC platforms," in *Proc. of SCOPES*, 2012.

[6] K. Goossens, A. Azevedo, K. Chandrasekar, M. D. Gomony, S. Goossens, M. Koedam, Y. Li, D. Mirzoyan, A. Molnos, A. Beyranvand Nejad, A. Nelson, and S. Sinha, "Virtual execution platforms for mixed-time-criticality systems: The CompSOC architecture and design flow," To appear in ACM SIGBED, 2013.

[7] K. Goossens and A. Hansson, "The Aethereal network on chip after ten years: goals, evolution, lessons, and future," in *Proc. of DAC*, 2010.

[8] B. Akesson and K. Goossens, "Architectures and modeling of predictable memory controllers for improved system integration," in *Proc. of DATE*, 2011.

[9] B. Akesson, A. Molnos, A. Hansson, J. Angelo, and K. Goossens, "Composability and predictability for independent application development, verification, and execution," *Multiprocessor System-on-Chip: Hardware Design and Tool Integration*, 2010.

[10] D. Goswami, M. Lukasiewycz, R. Schneider, and S. Chakraborty, "Time-triggered implementations of mixed-criticality automotive software," in *Proc. of DATE*, 2012.

[11] A. Masrur, S. Drossler, T. Pfeuffer, and S. Chakraborty, "VM-based real-time services for automotive control applications," in *Proc. of RTCSA*, 2010.

[12] H. Kopetz *et al.*, "Compositional design of RT systems: a conceptual basis for specification of linking interfaces," *Proc. of ISORC*, 2003.

[13] A. Wasicek, C. El-Salloum, and H. Kopetz, "A system-on-a-chip platform for mixed-criticality applications," in *Proc. of ISORC*, 2010.

[14] J. L. Tokar, "Space & time partitioning with ARINC 653 and pragma profile," in *Proc. of IRTAW*, 2003.

[15] O. Moreira, F. Valente, and M. Bekooij, "Scheduling multiple independent hard-real-time jobs on a heterogeneous multiprocessor," in *Proc. of EMSOFT*, 2007.

[16] A. Kumar, B. Mesman, B. Theelen, H. Corporaal, and Y. Ha, "Analyzing composability of applications on MPSoC platforms," *JSA*, March 2008.

[17] G. Heiser, "The role of virtualization in embedded systems," in *Proc. of IIES*, 2008, pp. 11–16.

[18] J. Yang, H. Kim, S. Park, C. Hong, and I. Shin, "Implementation of compositional scheduling framework on virtualization," *SIGBED Rev.*, vol. 8, no. 1, Mar. 2011.

[19] Y. Kinebuchi, M. Sugaya, S. Oikawa, and T. Nakajima, "Task grain scheduling for hypervisor-based embedded system," in *Proc. of HPCC*, 2008.

[20] M. M. H. P. van den Heuvel, M. Holenderski, W. Cools, R. J. Bril, and J. J. Lukkien, "Virtual timers in hierarchical real-time systems," in *WiP session of RTSS*, 2009.

[21] M. Holenderski, W. Cools, R. J. Bril, and J. J. Lukkien, "Multiplexing real-time timed events," in *ETFA*, 2009.

[22] A. Beyranvand Nejad, A. Molnos, and K. Goossens, "A unified execution model for data-driven applications on a composable MPSoC," in *Proc. of DSD*, 2011.

[23] X. A. Feng and A. K. Mok, "A model of hierarchical real-time virtual resources," in *Proc. of RTSS*, 2002.

[24] S. Shigero, T. Matsumoto, and H. Kei, "On the schedulability conditions on partial time slots," in *Proc. of RTCSA*, 1999.

[25] L. Almeida and P. Pedreiras, "Scheduling within temporal partitions: response-time analysis and server design," in *Proc. of EMSOFT*, 2004.

[26] G. Lipari and E. Bini, "Resource partitioning among real-time applications," in *Proc. of ECRTS*, 2003.

[27] I. Shin and I. Lee, "Periodic resource model for compositional real-time guarantees," in *Proc. of RTSS*, 2003.

[28] M. Joseph and P. K. Pandya, "Finding response times in a real-time system," *Comput. J.*, vol. 29, no. 5, 1986.

[29] P. Šůcha, M. Kutil, M. Sojka, and Z. Hanzálek, "TORSCHE scheduling toolbox for Matlab," in *Proc. of CACSD*, 2006.