Decoupled inter- and intra-application scheduling for composable and robust embedded MPSoC platforms

Anca Molnos
Technical University of Delft
the Netherlands
a.m.molnos@tudelft.nl

Ashkan Beyranvand Nejad
Technical University of Delft
the Netherlands

Ba Thang Nguyen
DEK Technologies
Ho Chi Minh City, Vietnam

Sorin Cotofana
Technical University of Delft
the Netherlands

Kees Goossens
Technical University of
Eindhoven, the Netherlands

ABSTRACT

Systems-on-Chip (SoCs) typically implement complex applications, each consisting of multiple tasks. Several applications share the SoC cores, to reduce cost. Applications have mixed time-criticality, i.e., real-time or not, and are typically developed together with their schedulers, by different parties. Composability, i.e., complete functional and temporal isolation between applications, is an SoC property required to enable fast integration and verification of applications. To achieve composability, an Operating System (OS) allocates processor time in quanta of constant duration. The OS executes first the application scheduler, then the corresponding task scheduler, to determine which task runs next. As the OS should be a trusted code base, both inter- and intra-application schedulers should be thoroughly analysed and verified. This is required anyway for real-time intra-application schedulers. But for non-real-time applications, a costly effort is required to achieve the desired confidence level in their intra-application schedulers. In this paper we propose a light-weight, real-time OS implementation that overcomes these limitations. It separates the two arbitration levels, and requires only the inter-application scheduler to run in OS time. The intra-application scheduler runs in user time, and is therefore not trusted code. This approach allows each application to execute its own specialised task scheduler. We evaluated the practical implications of our proposal on an SoC modelled in FPGA, running an H264 and a JPEG decoder and we found that composability is preserved and performance is improved with up to 37%.

Categories and Subject Descriptors

C.3 [**Special-purpose and application-based systems**]: Real-time and embedded systems; D.4.1 [**Operating systems**]: Process Management—*Multitasking*

SCOPES '12, May 15-16, 2012, St. Goar, Germany

Keywords

Composability, SoC, RTOS

1. INTRODUCTION

State-of-the-art Systems on a Chip (SoC) often execute complex embedded applications, many of them from the signal-processing, streaming domain. Applications may have *diverse performance demands*, e.g, high throughput for video, low latency for the user interface. Furthermore, they have various time constraints, i.e., some are real-time, e.g., audio processing, while some are not, e.g, web browsing. Some real-time requirements are firm, meaning that a deadline miss results into an unacceptable output quality loss. Other requirements are soft, which means that occasional deadline misses can be tolerated. Real-time applications require *temporal verification* to ensure that deadlines are met when running on the SoC. Furthermore, the applications may be developed by different parties, such as different teams within a company, or independent software vendors.

Applications consist of a set of tasks. To reduce the cost of an SoC, the tasks of these applications *share resources*, such as processors, memories, and interconnect. Hence, multiple tasks, potentially belonging to different applications, may demand a processor at a given moment in time. The processor arbitration is conventionally performed by a Real-Time Operating System (RTOS), and it has two parts, namely inter-application scheduling, henceforth called application scheduling, and intra-application scheduling, called task scheduling in the remainder. The RTOS' code should be trusted, to make sure that the stringent timing requirements of firm real-time applications are met. Hence the RTOS should be thoroughly analysed and verified to make sure it will always operate correctly, and cause no timing violation.

The inter-application interference at shared resources is a major hurdle for complex SoC design. The temporal behaviour of applications is inter-dependent, hence system verification and integration are circular processes that have to be repeated whenever an application is slightly changed, resulting in a large effort. Furthermore, resource sharing decreases system robustness, as a misbehaving application may monopolise resources, potentially leading to timing violation or erroneous behaviour of other applications.

Existing research proposes *timing isolation*, to reduce the complexity of timing verification. Here two types of ap-

proaches exist: ones that isolate applications at the level of timing bounds [1–3], and others that isolate applications completely, at cycle-level [4, 5], The latter approaches are called *composable*. In a composable system, the application starting time (defined as the moment when it begins to process its input data) and the application response time (defined as the duration between starting to process the input data and finishing to produce the output) are independent of other applications. This increases robustness, and facilitates independent application debug, besides offering timing isolation.

The composable approaches employ a processor sharing model in which a number of constant duration time quanta, called ticks or slots, are allocated to tasks. In these approaches, as well as in conventional RTOSes, the RTOS kernel typically performs both the application and the task scheduling, hence two main problems remain. Firstly, all task schedulers have to be thoroughly verified together with the RTOS, to gain the confidence that they behave according to the specification. Whereas this is not an issue for firm real-time schedulers, it represents a limitation for the soft and non real-time ones, which are typically designed to optimise the application performance and not necessarily to have tight temporal bounds. Hence application developers should simply not use some schedulers, or invest effort in thoroughly verifying and certifying them. The latter requires design procedures, methods, and tools specific to firm real-time and may incur investments that many application developers may not afford to make. Secondly, if all scheduling decisions are taken exclusively by the RTOS, between each consecutive quantum, if a task finishes early in a quantum, the left time, called internal slack, is wasted.

In this paper we propose a light-weight RTOS implementation that solves the abovementioned two problems by scheduling the applications in a composable manner, following a Time Division Multiplexing (TDM) policy, and allowing each application to schedule their tasks according to a policy that the application designer sees most fit. The system has all the benefits of composability, and our scheme allows designers to choose from a larger variety of task schedulers, and potentially achieve higher processor utilisation by not wasting the internal slack. Moreover, the RTOS designer is relieved from the burden of integration, strict verification, and characterisation of code from other parties. When an application, or its scheduler, malfunctions, other parts of the system are left unaffected. Furthermore, in case parts exhibit anomalous behaviour, the functional and temporal isolation makes it easy to determine the source of errors, facilitating debug.

The current limitation to our method is that the schedulers in application-time have no access to timers and interrupts, hence they cannot preempt tasks. The interrupt access must be reserved exclusively to the RTOS to implement the inter-application TDM using timers, such that system remains composable and robust. Notice that our method does not rule out the preemptive task scheduling in OS-time. A subset of the applications may defer task scheduling to the RTOS, and another subset executes it in the application time.

We investigate the composability and the performance of our proposal on a dual-processor SoC, implemented on an FPGA, using a workload consisting of two applications, an H264 decoder and a JPEG decoder. The experiments indicate that OS-time and application-time schedulers may be utilised simultaneously by different applications, and the platform remains composable. Moreover, up to 37% increase in performance is achieved when the task scheduler is executed in application-time as opposed to OS-time.

The paper is organised as follows: Section 2 discusses related work and Section 3 introduces the background for the rest of this paper. Section 4 presents the details of the proposed approach and it is followed by a presentation of our experiments in Section 5. Section 6 concludes the paper.

2. RELATED WORK

The problem of meeting timing constraints and ensuring robustness of mixed time-criticality applications that share the resources of an SoC is recognised as being complex [4, 6–8]. All existing approaches to reduce this complexity employ some form of isolation between applications. We identify four types of inter-application isolation, as follows. The first type of isolation is at the level of memory, I/O, and access privileges. This type of isolation ensures that applications do not, intentionally or otherwise, corrupt each-others data, or monopolise the resources. The second type of isolation is at the level of resource management. Resource management optimises platform utilisation, such that a required level of performance is achieved, potentially at a low energy consumption. The resource management policy suitable to an application depends on the performance and timing constraints of that application. We denote resource management as decoupled when each application is dealt with internally by its own management policy and the inter- and intra-application management decisions are taken at independent points in time. Decoupled resource management has the potential to achieve high platform utilisation, because it is able to immediately respond to the needs of each application. The third type of isolation is at the level of temporal bounds. This type of isolation ensures that, when the temporal bounds of an application are calculated in isolation, the results of this calculation hold true when the application shares a platform with other applications. The fourth type of isolation is at the level of cycles. We denote a system in which applications are completely isolated and cannot affect each other's temporal behaviour, at cycle level, as *composable*. Composability can be achieved if applications are not sharing resources [4], however we do not consider this approach due to its high costs and hence we do not discuss it further. To ensure composability we employ the techniques introduced in [5].

Research in application isolation takes place in several domains, namely, virtualisation for general purpose platforms, virtualisation for embedded systems, real-time scheduling and real-time OSes. Existing approaches achieve various levels of isolation, as briefly discussed in what follows.

Virtualisation solutions, such as Xen [9], VMWare [10], and VirtualBox [11], enable robust coexistence of several OSes on the same hardware platform. These solutions are natively designed to ensure memory, I/O, and privilege separation and to decouple scheduling, and are not concerned with timing. Recently real-time extensions, such as RTXen [12], are proposed. Such extensions rely exclusively on priorities to offer timing guarantees. In this case, the temporal bounds of applications are not truly independent, as the temporal bounds of lower priority applications depend on the behaviour of higher priority applications [13].

The examples above come from general purpose processors domain, and are prohibitively expensive for embedded systems. Nevertheless, light-weighted forms of virtualisation [6, 8, 14, 15] emerge also in embedded systems. Lower cost is achieved by sacrificing some of the virtualisation features. Similarly with the general purpose virtualisation solutions which target real-time, the embedded ones also rely mostly on priorities. A popular example is RT Linux [16] with uses priorities to orchestrate the simultaneous execution of real-time and average-case optimised tasks. Similarly, μC/OS-II [17] relies on preemptive, priority-based scheduling. These may offer temporal guarantees to some applications, however does not ensure temporal isolation. Exception makes the L4 real-time kernel [7] that relies on reservations.

Real-time schedulers [1–3, 18, 19] and real-time OSes [20–22] that offer temporal isolation utilise various forms of resource reservation. The temporal bounds of the application depend only on the parameters of the reservation (e.g., budgets) and not on other applications. The real-time schedulers do not implement memory, I/O, and privilege separation, and perform coupled application and task scheduling, even if the schedulers are hierarchical. The work in [23] and [24] goes one step further, and demonstrate cycle-level isolation, i.e., composability. Also in these approaches, inter- and intra- application scheduling is coupled.

While all these approaches bring interesting ideas on how to meet timing constraints and ensure robustness, we address application isolation from a different angle. Instead of simplifying existing virtualisation techniques to offer real-time guarantees, or extending real-time schedulers with timing isolation, we start from a baseline platform that is composable by construction. On this platform we add features (isolation mechanisms) one by one, making sure that the SoC remains composable. In this paper we present a light-weight RTOS that decouples application and task scheduling. In the future we will address memory and I/O isolation.

3. BACKGROUND

In this section we present the software applications, the architecture template, and finally we highlight the existing processor composability mechanisms relevant to application execution. In Section 4 we build up upon these mechanisms and present the proposed light-weight RTOS.

3.1 Software application model

In this section we introduce the real-time and non real-time application models.

The model considered for firm and soft real-time applications is a streaming one, as many such applications in the embedded domain are dedicated to signal-processing. A streaming application consists of a set of tasks which communicate data tokens through first-in first-out (FIFO) buffers, for instance following the C-HEAP protocol [25]. Such applications can be modelled using data-flow. The literature offers many variants of data-flow models, e.g., single-rate data-flow, cyclo-static data-flow, and variable-rate data-flow. Some of these models are amenable to formal performance analysis [26], e.g., single-rate data-flow and cyclo-static data-flow, and hence can be used for real-time applications. Nevertheless, our RTOS offers native support to execute applications modelled in any data-flow variant. Each application task corresponds to a data-flow actor, and a FIFO is modelled as a pair of opposing edges. Such a task iterates infinitely, in each iteration consuming, i.e., reading, its input tokens, executing what we denote as the task function, then producing, i.e., writing, its output tokens, which in turn may be processed by other tasks. Each FIFO is attached to one producer task and one consumer task, which block if the FIFO is full, or empty, respectively. A firing rule defines the number of tokens produced and consumed in a task iteration. A task is scheduled only if eligible, meaning that the firing rule of the task is met. If a task needs to carry data values from one iteration to another, this should be implemented explicitly with a self FIFO, because data-flow actors are state-less. Note that state here refers to values of variables, and not to the task state, as defined in Section 4.3. Furthermore, we assume tasks communicate only with tasks of the same application and that the task to processor binding is static.

Non real-time applications may also be implemented under the same programming model as real-time applications. However, non real-time applications do not have to be analysable, thus any other model implementable over a shared memory multi-processor is also suitable, within the platform limits, as presented below. Thus application can be expressed using sequential C code, Message Passing Interface, etc. Our software stack offers a basic Application Programmer Interface (API), for inter-task and inter-processor communication. These primitives can be utilised to implement higher level programming libraries.

3.2 SoC architecture model

We consider a composable, tiled MPSoC built along the principles presented in [5]. The MPSoC consists of a set of processor tiles and a set of memory tiles, that communicate via a connection-oriented Network on Chip (NoC). A processor tile comprises of a processor, local memory blocks (lcl mem), and Direct Memory Access (DMA) engines. A DMA engine transfers data from/into a local shared memory (Sh mem) block to/from a remote memory via the global NoC, as shown in Figure 1, hence it improves performance by overlapping communication and computation.

To achieve a composable system, each shared resource of the SoC should be composable [24]. The limitation of composability is that no shared resource can be locked by an application for an indefinite duration. The arbiters in the platform ensure that. Hence any programming library that does not rely on locking resources indefinitely can be implemented on this architecture.

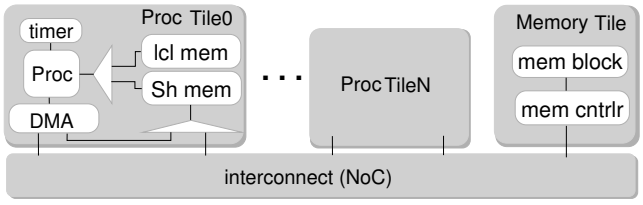

Figure 1: Tiled architecture template

We assume that the instructions and data of each task executing on a processor tile reside in local memory of that tile, otherwise explicit DMA calls should be included in the task code. The FIFO buffers may be mapped in the producer's or the consumer's local memory, or in a memory tile.

In the following section we discuss the composable processor particularities relevant for inter- and intra- application scheduling.

3.3 Composable processor tile

Processor composability is achieved by strict TDM allocation of constant duration (application) slots [23] that have periodical, fixed start times. The time in which an application executes, i.e., within a slot, is called application-time. The constant-duration slots are implemented with timer interrupts and preemptive scheduling. The time it takes to give the control to the interrupt service routine is dependent on the application instructions present the processor pipeline when the interrupt arrives. Thus all tasks should be interruptible in bounded time. For some processors, e.g., the MicroBlaze, the instructions that potentially have an unbounded interrupt delay are remote blocking memory accesses, hence these processors should not execute such accesses. Instead, the processor programs a DMA to perform remote accesses, and then it polls on a local memory location in case it needs to determine if a read access has completed.

To ensure independent application starting times, the time in between two application slots, in which the interrupt is served and the RTOS executes, should be application-independent. This time between two consecutive application slots is called OS-time. Here two operations may take an application-dependent time. The first is the jump to the interrupt service routine just before the RTOS is invoked. The second is the application scheduling and, potentially also the task scheduling. To guarantee composability, the OS time is forced to appear as these operations always take their worst case execution time. This is implemented by halting or keeping the processor busy up to its worst case execution time [23].

As the application timing have to be independent, if a task finishes earlier within an slot, the left time cannot be used to execute another application. To use this left time for another task of the same application, the task scheduling has to be triggered within the application slot, as proposed in the next section.

4. COMPOSABLE RTOS WITH DECOUPLED SCHEDULING

In this section we present our RTOS that implements application scheduling in OS-time and task scheduling in either application- or OS-time.

Each processor tile executes its own, independent, RTOS instance. We implement the application-time task scheduling starting with the baseline processor tile introduced in Section 3.3. The main components of the proposed RTOS are presented in Figure 2. The RTOS main responsibilities are to schedule applications and to offer scheduling and communication interfaces. Note that the RTOS can also schedule tasks using a set of native, verified task schedulers. Moreover, each scheduler takes tile-local decisions, and it is not synchronised with schedulers on other tiles, which ensures complete decoupling from other processors and memories. Furthermore, the RTOS maintains the data structures necessary to implement applications, tasks, and FIFOs.

To ensure robustness we prohibit, by construction, that applications freely access RTOS's data structures, hence the RTOS provides an application wrapper and an API described

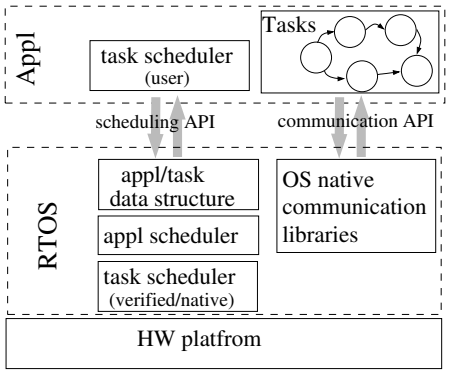

Figure 2: Hardware and software stack

in the next sections. Note that calling an API function does not require a context switch, as conventional system calls in many existing OSes. Inside these functions the interrupts are disabled during the actual updates of RTOS data-structures to ensure mutually exclusive access to these shared data and thus preserve their coherent state. To reduce the worst case time to jump to the interrupt service routine, and to limit the application switching penalty, the API functions are optimised to keep the interval with interrupts disabled to the bare minimum. Moreover, for robustness, the RTOS does not offer an API to disable interrupts. If a task scheduler is faulty and, for example, never returns, the current application blocks, however the timer interrupts occur regularly, and the RTOS schedules other applications.

In the following we start by presenting the RTOS and application initialisation procedure, and continue by describing the RTOS functionality.

4.1 Initialisation

The initialisation of the RTOS data structures that support the execution of application is performed at system start-up and it follows the steps presented in Figure 3.

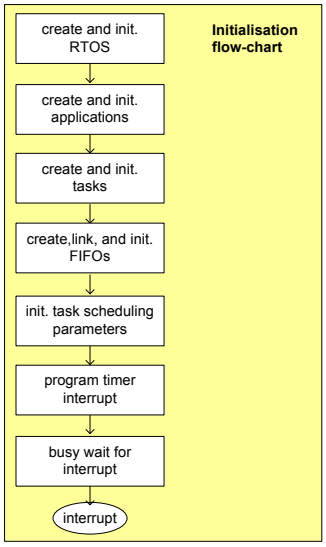

Figure 3: Initialisation steps.

As mentioned before, one RTOS instance executes on each

processor tile. In the first initialisation step, the *OS control block (OCB)* data structure, which corresponds to the RTOS instance, is created and initialised. The OCB stores, among others, the parameters of the TDM application scheduler and the parameters, e.g., the address, of each DMA. In the next step, the data structures corresponding to applications and tasks, i.e. *application control block (ACB)* and *task control block (TCB)*, respectively, are created and initialised. Among others, the ACB specifies whether an application is real-time or not, and whether the task scheduling is performed in application- or OS-time. The TCB is linked to two user-defined functions that specify the task computation and its firing rules. After initialisation, tasks are in an idle state. The FIFOs are created by *FIFO control blocks (FCBs)* with static space allocation in local or remote memories. Every FCB is linked to a producer and a consumer TCB. The FIFOs are initialised by injecting a number of initial tokens that are provided by the user application.

The task scheduling arguments are created and initialised with values provided by the user application. These arguments may include any information necessary to the user-defined task scheduler, e.g., a list of previously executed tasks and the tasks order. The address of this memory space is passed to the user defined task scheduler that can read and update these arguments during its execution.

Finally, the interrupt timer is programmed and the system waits for the first interrupt signal to start the RTOS and application execution, which is explained in the next subsection.

4.2 Decoupled application and task scheduling

Applications are executed following a functional loop that involves the main steps presented in Figure 4. Each of these steps belongs to the RTOS kernel, the application wrapper, or the user application itself, as indicated in this figure. The RTOS kernel executes in OS-time. The application wrapper runs in application-time, and it is a sequence of instructions that enables the execution of the user application. The user application comprises the computation function and the firing rule update function of each of the application tasks, and the task scheduler. The kernel and the application wrapper are trusted code, provided, verified, and analysed by the RTOS developers, whereas the user code does not need to be thoroughly verified. Moreover, Figure 4 explicitly marks the steps that can be interrupted, i.e., in which a task can be preempted, and the potential states of a task during each step. The latter are detailed in the next section. As a proof of concept, we implement a Round-Robin (RR) task scheduling policy, both on OS- and application-time and an TDM task scheduling policy in OS-time, using the templates described below. In what follows we discuss the execution steps corresponding to the RTOS kernel and the two application classes we address, namely real-time and non real-time.

4.2.1 RTOS kernel

In the OS-time, the RTOS kernel starts by saving the context of the currently running task on the stack of this task. After that, the next application to be executed is determined. If the task scheduling is performed in OS-time one of the native RTOS task schedulers are called, to determine the next tasks to run. Otherwise, the next task to run is the one that was preempted when the application was previously interrupted. The next step is to re-program the

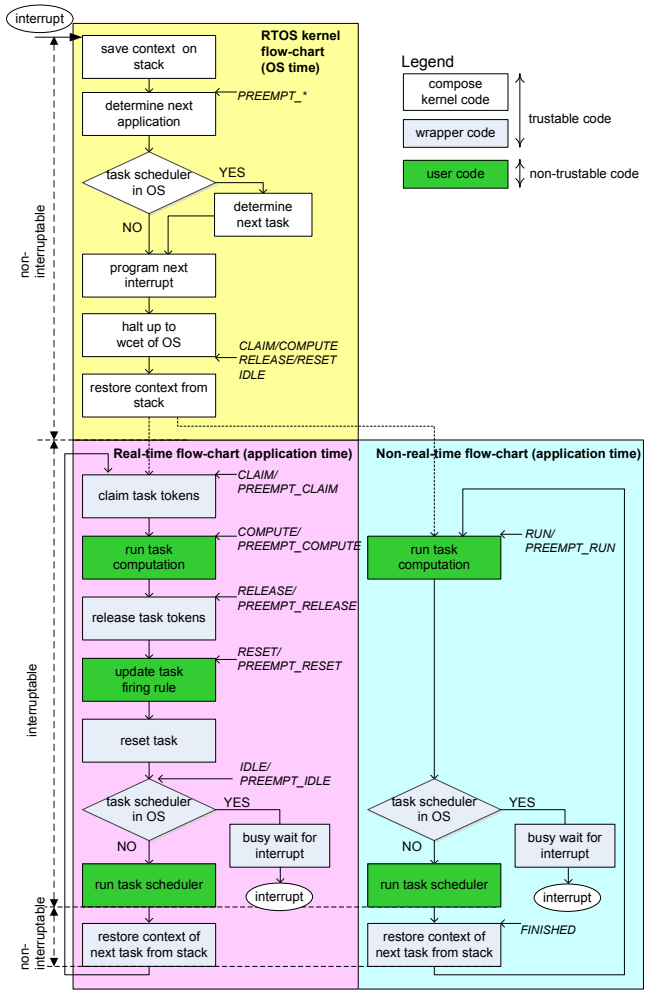

Figure 4: RTOS and application steps.

timer interrupt. This is followed by a processor halt up to the worst case execution time of the kernel, to ensure composability, as mentioned in Section 3.3. Finally the context of the next running task is restored.

4.2.2 Real-time applications

The execution of a real-time application follows the data-flow model, hence the corresponding functional loop matches this model. Our RTOS provides a wrapper to implement data-flow applications, thus such applications can be easily developed and executed on the platform. Similar to a data-flow actor, a task is fired, i.e., scheduled, only if it is eligible, meaning that it has all its input data and output space available. Once a data-flow actor fires, it should end its iteration without blocking for other resources like DMAs. To ensure this, the wrapper makes sure that the input data of a task is stored locally, in the memory of the processor tile that runs that task. Thus, the first step in executing a streaming task is to claim tokens. This step copies the input data from a remote memory, to the local communication memory, if these data were not already there, i.e., they were stored at the consumer processor tile, as described in Section 3.1. Following that, in the run task computation step, the user-provided task function is called. This function has the following interface:

`rt_task_func(input_data, output_data, fr_param)`

where the `input_data` and `output_data` are the local memory addresses of the input data and output storage space, and the `fr_param` is explained below. After the task computation is finished, the tokens produced are released, meaning that, if necessary, the output data is sent to a remote memory location, either at the processor tile that runs the consumer task, or in a separate memory tile.

The next step is to update the firing rule for the next task iteration. This is performed via a user-defined function with the following interface:

`update_firing_rule(fr_param)`

where the `fr_param` points to a user-defined data structure that stores all the information necessary to update the firing rules for the next task iteration. This function is used to implement, for example, the cyclo-static data-flow model which has cyclically changing firing rules. Furthermore, these parameters can be set during the task computation, such that data dependent firing rules are supported, to implement the variable-rate data-flow model.

After updating the firing rules, the task is reset to its original starting point. In this manner the application wrapper implements infinite, iterative task execution, i.e., next time when the task is scheduled it starts with claiming tokens, etc. This implies resetting all the task registers.

If the task scheduler is executed in the OS-time, the rest of the application slot is left idle, while busy waiting for a timer interrupt. Otherwise, the user-provided task scheduler is called. This scheduler has the following interface:

`int task_scheduler(scheduler_param)`.

where the `scheduler_param` points to a user-defined data structure that stores all the information necessary to schedule tasks. The scheduler function returns the identifier of the next task, or, if no eligible task is found, the one of the idle task. To determine task eligibility the RTOS offers the function `check_firing_rules(task_id)`, as a part of scheduling API. Another function provided by the scheduling API, to facilitate the implementation of simple task schedulers, is `get_task_id()`. The last step is to restore the context of the next scheduled task. Hierarchical context switches are not supported, hence restoring the context of a task is executed with the interrupts disabled, as visible in Figure 4.

4.2.3 Non real-time applications

The execution of a non real-time application is simpler and consist of two steps, namely the task function call, and, when appropriate, the task scheduler call. The non real-time task scheduler has the same interface as the one in real-time applications. In case of non real-time applications the task function has as arguments its input and output FIFOs and the DMA engines that the task may use, as follows:

`nrt_task_func(in_fifo_ids, out_fifo_ids, dma_ids)`

Note that preemptive scheduling is not currently supported in application-time, hence, in non real-time applications, a task that is once started will run to completion before any other task may execute on the processor. To avoid deadlock situations in which one task cannot progress because it is waiting for data from other tasks that can never be swapped in the processor, whenever a task is blocked in communication, the task scheduler is called. This means that each of the communication APIs checks for data and space availability, and in case the communication cannot proceed the user-provided task scheduler is invoked.

4.3 Task states

In this section we present the task states and transitions between states, first for real-time application and then for non real-time ones.

The states of tasks of a real-time application are tailored to match the data-flow iterative execution. The corresponding states and transitions are presented in Figure 5. A task starts by being `IDLE`, and, when it is scheduled, it goes through the `CLAIM`, `COMPUTE`, and `RELEASE` states, in this order. After the task finishes releasing its tokens, its iteration is in fact completed. However its firing rules have to be updated and the task has to be reset before it can start its next iteration. The firing rule update and task reset are performed in the `RESET` state. Each of the states above have their `PREEMPT_` counterpart, for the case when the task is interrupted. In application-time scheduling, after updating the firing rules the task becomes `IDLE` and task scheduler is executed. A timer interrupt may occur during the execution of the task scheduler, hence, the `IDLE` state has also a `PREEMPT_IDLE` pair.

In OS-time scheduling, when a task is preempted in one of the `PREEMPT_CLAIM`, `PREEMPT_COMPUTE`, `PREEMPT_RELEASE`, and `PREEMPT_RESET` states, it can be re-scheduled immediately, as it is eligible for execution. If a task is in the `PREEMPT_IDLE` state it means that the task is between two iterations, hence its firing rules should be checked before scheduling.

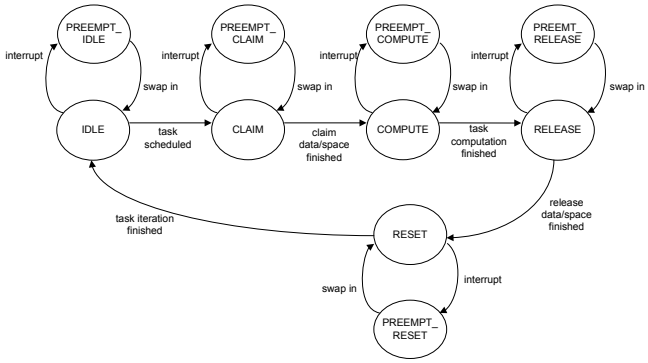

Figure 5: Task states for a real-time application.

The states of tasks of non real-time applications are presented in Figure 6. Initially, a task is in the `IDLE` state, then it is scheduled for execution and switches to `RUN`, and finally it becomes `FINISHED` when its execution ends. A task in the `RUN` state can be preempted when a timer interrupt occurs, and it changes to the `PREEMPT_RUN` state.

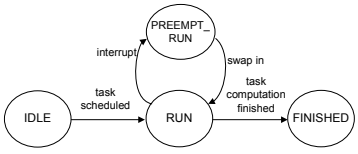

Figure 6: Task states for a non real-time application.

4.4 Communication API

The RTOS provides two categories of communication API, namely, FIFO communication API, and general, remote communication API, to prevent applications from stealing eachother's communication resources, namely the DMAs. The light-weight FIFO APIs offer native support for streaming applications and the general communication APIs offer support for various programming models and libraries.

The first API set consists of the following functions:

`readFIFO(infifo_id, buffer, ntokens)`.

`writeFIFO(outfifo_id, buffer, ntokens)`.

The later API set consists of the following functions:

`readSHM(buff_local, buff_remote, size, dma_id)`.

`writeSHM(buff_remote, buff_local, size, dma_id)`.

The arguments of these functions are self-explanatory.

The `readFIFO` and `writeFIFO` calls may initiate DMA transfers, depending on the location of the FIFO buffer, e.g., no DMA transfer occurs if the the producer and consumer tasks are on the same processor tile and the FIFO is hence local, and a DMA transfer occurs at `writeFIFO` if the the producer and consumer tasks are on different processor tiles and the FIFO buffer is mapped in the local memory of the consumer's tile. When required, the DMA transfers are initiated in the FIFO API calls, transparently to the tasks of the application.

The `readSHM` and `writeSHM` calls are self explanatory. They initiate a memory copy from, or to, a remote memory, located in another processor tile or in a dedicated memory tile, utilising one of the DMA identifiers that the task received as argument. We would like to underline that all RTOS APIs, hence also the communication ones, are interruptible in bounded time, so scheduling other application cannot be delayed indefinitely.

5. EXPERIMENTAL RESULTS

We experiment with the proposed RTOS on a composable platform comprising two processor tiles, each equipped with a MicroBlaze core, and one memory tile, connected via a network on chip, implemented on a Virtex 6 Xilinx FPGA. Each of the MicroBlazes cores executes an instance of the RTOS. We would like to emphasise that the RTOS overhead is small. Its execution time is smaller that 800 cycles, and its memory footprint is 13 KBytes, out of which 4 KBytes represent the FIFO APIs. This footprint is not far from what other light-weighted RTOSes, e.g, TinyOS have.

In the experiments we use two applications: a H264 video decoder and a JPEG decoder. Each of these applications consists of a set of communicating tasks. To provide a proof of concept, we execute the H264 and JPEG decoder concurrently, and both processor tiles are shared by both applications. As mentioned, the inter-application scheduling policy is TDM, and we implement a Round-Robin (RR) task scheduling policy, both on OS- and application-time. In OS-time a TDM task scheduler is also available, hence in total we have three task schedulers. In this section we verify whether the composability is ensured on the implemented platform, and we investigate the performance of the application-time task scheduler.

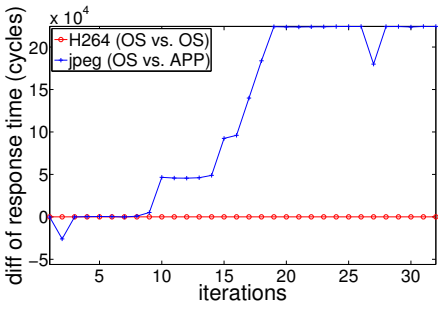

Figure 7: Response time differences; H264 composable.

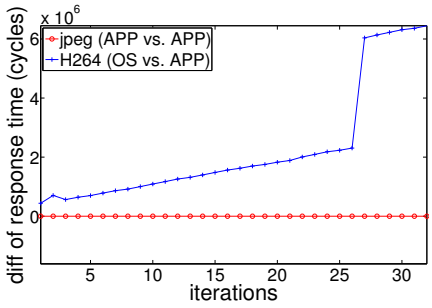

Figure 8: Response time differences; JPEG composable.

5.1 Composability

We tested and validated the RTOS functionality in many scenarios. Here we present two scenarios when an application changes its task scheduling from OS-time to application-time, while the other application's task scheduling policy is unchanged. If the system is composable, the starting and response time of the unchanged application should be identical in the two scenarios.

In each scenario, we performed two executions. In the first scenario, the H264 is scheduled in the OS-time while the JPEG task scheduler is changed from OS-time to application-time in the two executions. Likewise, in the second scenario the JPEG is scheduled in the application time and H264 task scheduler is changed. Figures 7 and 8 presents the difference in response time between the two executions in the first and second scenario, respectively. This difference is presented per iteration, and each application is scheduled following a RR policy. The illustrated graphs in Figure 7 shows that the response time difference between the H264 application executions is zero and therefore it is unaffected by the change in task scheduling strategy of the other application. Similar results are shown in Figure 8 for the JPEG application. We observed the same behaviour for starting times, i.e., unchanged regardless of the task scheduling strategy of other application. This indicates a composable co–existence of application- and OS-time task schedulers for different applications executing concurrently on the platform.

5.2 Performance

Figures 9 and 10 present the H264 and JPEG finishing time per iteration, as a function of the duration of the slot, for the three task schedulers introduced in the begin-

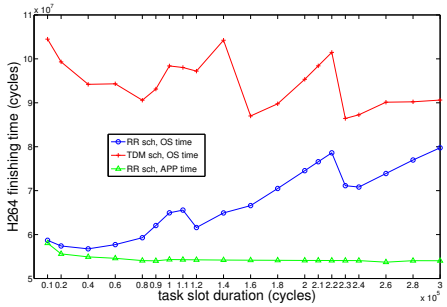

Figure 9: H.264 finishing times

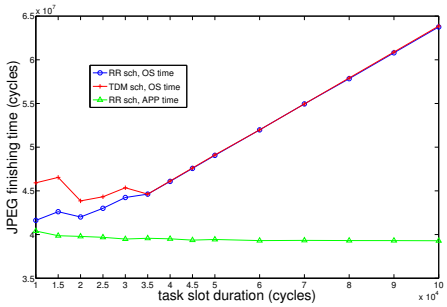

Figure 10: JPEG finishing times

ning of this section. We ensure that applications start at a fixed point in time, thus smaller finishing times indicate better performance. The slots are varied starting with 10000 and ending with 300000 and 100000 cycles for the H264 and JPEG, respectively. We stop the slot duration investigation when no significant difference in performance is observed for the application-time task scheduling.

The experiments suggest that in the OS-time case, the performance generally decreases with the increase of the slot duration, for both the JPEG and the H264 applications. For small slots, the performance slightly improves up to a certain point after which the tasks iterations start to fully fit inside of a slot. For larger slots, the performance decreases due to increased internal slack waste. This is not the case for application-time scheduling where the internal slack is eliminated, thus the performance increases, and saturates at the point when tasks iterations entirely fit within a slot.

Moreover, we found that TDM yields the worst performance because it is non work-conserving. Application-time scheduling always yields better performance than the OS-time scheduling with the same policy. Over the investigated task slots range, the finishing times in case of application-time scheduling decrease up to 33%, and to 37% when compared to the OS-time scheduling, for the the H264 and JPEG, respectively. Further increase in slot duration would decrease performance in OS-time due to increased internal slack waste, whereas the performance in application-time would remain constant, growing the gap between the two. The experiments also indicate that the overhead due to extra context switches in application-time scheduling does not impact the overall performance. In these experiments we enforced the same RTOS duration, thus the application-time scheduling performance would further improve if we would exclude the task scheduling time from the RTOS.

6. CONCLUSION

This paper proposes a light-weight RTOS implementation for composable, robust execution of applications with mixed time-criticality on the same SoC platform. The top scheduling level, inter-application, is composable, and it is performed in the OS-time; the second level, intra-application, may follow whatever cooperative policy the application designer sees fit, and it is performed in application-time. This RTOS has two contributions when compared to prior composable RTOSes: (i) it achieves a higher processor utilisation by not wasting any user-time; (ii) it shifts the responsibility of intra-application scheduling from the RTOS designer to the application designer, who typically has the required knowledge, without compromising on system robustness. The RTOS offers native support for streaming applications with iterative tasks that communicate through FIFO buffers, and basic general support for inter-processor communication, such that other types of applications can be executed. Furthermore we demonstrate this RTOS on an SoC on FPGA with two large applications, an H264 decoder and a JPEG decoder, and three different task scheduling policies. The experiments indicate that application-time task scheduling delivers up to 37% higher performance when compared to OS-time task scheduling, and the OS-time and application-time scheduling strategies may be utilised concomitantly by different applications, and the platform is composable.

7. REFERENCES

[1] I. Shin *et al.*, "Periodic resource model for compositional real-time guarantees," in *In Proc. of RTSS*, 2003.

[2] Z. Deng *et al.*, "Scheduling real-time applications in an open environment," in *In Proc. of RTSS*, 1997.

[3] G. Lipari *et al.*, "A methodology for designing hierarchical scheduling systems," *J. Emb. Comp.*, 2005.

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

[5] B. Akesson, A. Molnos, A. Hansson, J. Angelo, and K. Goossens, "Composability and Predictability for Independent Application Development, Verification, and Execution," in *Multiprocessor System-on-Chip: Hardware Design and Tool Integration*, ser. Circuits & Systems. Springer Verlag, November 2010, pp. 25–56.

[6] A. K. Mok and A. X. Feng, "Real-time virtual resource: A timely abstraction for embedded systems," in *In Proc. of EMSOFT*, 2002.

[7] H. Hartig *et al.*, "Ten years of research on L4-based real-time systems," in *In Proc. of the 8th Real-Time Linux Workshop*, 2006.

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

[9] Xen. [Online]. Available: http://www.xen.org

[10] VMWare. [Online]. Available: http://www.vmware.com

[11] VirtualBox. [Online]. Available: http://www.virtualbox.org

[12] S. Xi, J. Wilson, C. Lu, and C. Gill, "RT-Xen: Real-time virtualization based on fixed-priority hierarchical scheduling," Washington University in San Louis, Tech. Rep., 2010.

[13] C. Mercer *et al.*, "Temporal protection in real-time operating systems," in *In Proc. of RTOSS*, May 1994.

[14] A. Masrur *et al.*, "VM-based real-time services for automotive control applications," in *In Proc. of RTCSA*, 2010.

[15] Virtuallogix. [Online]. Available: http://www.virtuallogix.com

[16] RT Linux. [Online]. Available: https://rt.wiki.kernel.org/

[17] μC/OS-II. [Online]. Available: http://micrium.com

[18] S. Saewong *et al.*, "Analysis of hierarhical fixed-priority scheduling," in *In Proc. of RTS*, 2002.

[19] R. J. Bril, "Towards pragmatic solutions for two-level hierarchical scheduling: A basic approach for independent applications," Technische Universiteit Eindhoven, The Netherlands, CS-report 07/19, 2007.

[20] J. Sales, *Symbian OS Internals*. John Wiley & Sons, 2005.

[21] M. Behnam *et al.*, "Towards hierarchical scheduling on top of VxWorks," in *In Proc. of OSPERT*, 2008.

[22] K. Lakshmanan *et al.*, "Distributed resource kernels: OS support for end-to-end resource isolation," *In Proc RTAS*, 2008.

[23] A. Hansson *et al.*, "Design and implementation of an operating system for composable processor sharing," in *Microprocessors and Microsystems*. Elsevier, 2010.

[24] ——, "CoMPSoC: A template for composable and predictable multi-processor system on chips," *In ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 1, 2009.

[25] A. Nieuwland *et al.*, "C-HEAP: A heterogeneous multi-processor architecture template and scalable and flexible protocol for the design of embedded signal processing systems," in *Design Automation for Embedded Systems*. Kluwer, 2002.

[26] S. Sriram *et al.*, *Embedded Multiprocessors: Scheduling and Synchronization*. CRC Press, 2000.