Computers and Electrical Engineering 53 (2016) 89–105

Contents lists available at ScienceDirect

Computers and Electrical Engineering

journal homepage: www.elsevier.com/locate/compeleceng

RTOS acceleration in an MPSoC with reconfigurable hardware

Pavel G. Zaykov[a,*], Georgi Kuzmanov[a], Anca Molnos[b], Kees Goossens[c]

[a] Delft University of Technology, CE Lab, Mekelweg 4, 2628 CD Delft, The Netherlands
[b] CEA LETI, Grenoble, France
[c] Eindhoven University of Technology, ES group, Den Dolech 2, 5612 AZ Eindhoven, The Netherlands

ARTICLE INFO

Article history:
Received 10 June 2015
Revised 11 January 2016
Accepted 31 March 2016

Keywords:
Processor–coprocessor execution model
Multiprocessor platform
RTOS
Hardware task-status manager
Thread interrupt state controller
Inter-tile remote slack information
Distribution framework
Molen
FIFO communication

ABSTRACT

In this paper, we address the problem of improving the performance of real-time embedded Multiprocessor System-on-Chip (MPSoC). Such MPSoCs often execute applications composed of multiple tasks. The tasks on each processor are scheduled by a Real-Time Operating System (RTOS) instance. To improve performance, we reduce the Worst Case Execution Time (WCET) of the RTOS by new processor-coprocessor execution models using reconfigurable hardware. Furthermore, we integrate the proposed contributions on an MPSoC platform, where the processor-coprocessor execution models are applied on three reconfigurable coprocessors - Hardware Task-Status Manager (HWTSM), Thread Interrupt State Controller (TISC), and Remote Slack Manager (RSM). As a case study, we investigate the HWTSM, which determines the execution eligibility of tasks. The experimental results suggest overall system improvement up to 13.3% with the help of the HWTSM. Moreover, the TISC can boost further the performance and RSM can significantly reduce the energy consumption.

1. Introduction

Contemporary embedded systems execute an increasing number of applications that demand high performance. Many of these applications belong to the signal processing domain and usually require the execution of complex audio, video, and telecommunication algorithms in real-time. Often, such applications are partitioned into multiple tasks, which are concurrently executed on a Multiprocessor System-on-Chip (MPSoC). Furthermore, efficient task scheduling on each one of the processors of the MPSoC is required. A widely accepted solution for this scheduling problem is to employ a Real-Time Operating System (RTOS) in software.

The temporal behaviour of real-time applications on the MPSoC must be characterizable, i.e., predictable. In turn, this implies that the RTOS should have a worst-case bound on its execution time, i.e., a Worst Case Execution Time (WCET). Hence, to be able to improve overall system performance, the WCET of the RTOS should be as short as possible. Last but not least, existing RTOSs may have extra properties meant to ease application design. Composability, proposed and advocated in several real-time systems [1,2], is one of them. Composability means that the behaviour of an application, including its timing, is independent of the presence or absence of any other application. This property is very important for the temporal

* Corresponding author.
E-mail addresses: p.g.zaykov@tudelft.nl (P.G. Zaykov), g.k.kuzmanov@tudelft.nl (G. Kuzmanov), anca.molnos@cea.fr (A. Molnos), k.g.w.goossens@tue.nl (K. Goossens).

http://dx.doi.org/10.1016/j.compeleceng.2016.03.016
0045-7906/

verification of mixed-time criticality applications that run on the same MPSoC platform [3]. Solutions that improve RTOS performance should preserve composability.

We consider an MPSoC architecture in which each processor executes its own instance of the same RTOS, also known as homogeneous Asymmetric Multiprocessing (AMP). The RTOS typically preempts, schedules, and loads user tasks. The exact steps involved in the RTOS execution, however, are dependent on the application model of computation. One popular model of computation suitable for streaming applications is data-flow [4]. A data-flow application consists of a set of tasks that communicate through First-In-First-Out (FIFO) queues. A FIFO queue has one reading and one writing task. The RTOS schedules only tasks that are eligible for execution, meaning that they have input data to operate on, i.e., input FIFOs are not empty, and space to produce output data, i.e., output FIFOs are not full. In this paper, we investigate two data-flow models and a model of computation similar to POSIX [5].

In this paper, we address the problem of improving the performance of real-time embedded MPSoCs by reducing the WCET of the RTOS. More specifically, we map one of the most time-consuming RTOS services from software, executed on a processor, to a dedicated hardware coprocessor. In the presented case study, this is the service responsible for checking execution eligibility of tasks by using the FIFO-filling information. We denote the custom coprocessor responsible for checking the task eligibility as Hardware Task-Status Manager (HWTSM) [6]. The computed task-status information is employed as an input to the RTOS scheduler. In addition to the HWTSM, we also refer to two other coprocessors reducing the RTOS costs – Thread Interrupt State Controller (TISC) [5] and Remote Slack Manager (RSM) [7]. The TISC is designed to minimize the thread synchronization costs. The RSM reduces overall energy consumption of applications executed over multiple tiles in an MPSoC. To demonstrate the overall benefits of all three coprocessors, we integrate them on a conceptual MPSoC platform.

To decide how to implement performance critical RTOS services in reconfigurable hardware and the HWTSM in particular, we first discuss six processor–coprocessor execution models among which is the processor–coprocessor *parallel non-blocking* execution model. A coprocessor, operating in *parallel non-blocking* execution model, has the following characteristics: (a) it runs continuously and it does not need to be restarted every time the processor needs the coprocessor; (b) the processor can request a result from the coprocessor at any time; (c) independent of its current status, the response time of the coprocessor is always constant. Second, we outline the integration of the investigated processor-coprocessor execution models to common models of computation. Third, we present a conceptual computing platform, which integrates three reconfigurable coprocessors running in various processor-coprocessor execution models.

We implement the HWTSM with the *parallel non-blocking* model, hence its execution overlaps with the software execution of user applications and RTOS services. The response time of the HWTSM is very short and constant, equal to five cycles in our prototype, which leads to a significant reduction of the RTOS cost. Although the HWTSM executes concurrently with the applications, the HWTSM does not influence the applications behaviours. The HWTSM has a constant, application-independent response time and does not introduce additional RTOS execution variability. As a result it preserves the composability of the system. We quantify three potential integrations of the HWTSM to an MPSoC system. The TISC coprocessor operates in *parallel blocking* execution model, while the RSM employs the *parallel non-block* execution model.

Summarizing, the main paper contributions are as follows:

- We provide an overview of a potential integration of processor-coprocessor execution models to two data-driven models of computation and a model of computation similar to POSIX [5];
- We discuss the proposed contributions on a conceptual computing platform, where the processor-coprocessor execution models are applied on three reconfigurable coprocessors;
- We compare in terms of performance and integration effort three implementations of the Hardware Task-Status Manager (HWTSM) [6] to a Multiprocessor System-on-Chip (MPSoC) and detailed RTOS integration;
- We quantify the benefits of the Hardware Task-Status Manager (HWTSM) [6],Thread Interrupt State Controller (TISC) [5], and Remote Slack Manager (RSM) [7] on a conceptual MPSoC platform;

To prove the applicability of the HWTSM, we experiment with two synthetic applications and two real applications, i.e., JPEG and H.264 decoders, respectively. All applications are executed on a CompSoC platform instance [3], implemented on an FPGA. With the synthetic applications, we investigate the RTOS cost reduction by varying the scheduling policies, because they are the ones that directly affect the number of the task-status computations. The experimental results on synthetic benchmarks suggest a reduction in the WCET of the RTOS, compared to a pure software implementation, between 1.1 and 1.8 times for static scheduling policies. For dynamic scheduling policies, this WCET reduction is between 1.1 and 3.0 times. With real applications, the reduction in the WCET of the RTOS with HWTSM is between 1.3 and 1.6 times, for the JPEG and H.264 decoders, respectively. Moreover, we observe that the overall performance gain varies from 2.3% to 7.5%, when the WCET of the RTOS is reduced, respectively.

We conclude the experimental section with an overall result on a conceptual platform, which integrates three reconfigurable coprocessors. We estimate that the overall system improvement is up to 19.6 times with the help of the Thread Interrupt State Controller. The performance can be boosted up to 13.3% more with the help of the Hardware Task Status Manager. Last but not least, the improvement of the system energy consumption can be reduced upto 51.1% over the current state of the art with the help of inter-tile remote slack information distribution framework.

The remainder of the paper is organized as follows. Section 2 discusses the related work. The problem is defined in Section 3. Section 4 presents the proposed solution. Section 5 covers the implementation details. Section 6 reports the experimental results and Section 7 concludes the paper.

2. Related work

In this section, we discuss related projects that employ hardware acceleration for RTOS. Then, we compare our approach with hardware acceleration for data-flow models of computation. We conclude the section, by discussing the available processor–coprocessor execution models.

There are numerous examples of embedded systems, such as [8–10] using an RTOS to co-execute applications in software and in hardware. Although, these approaches target coarse-grained reconfigurable embedded systems, the same ideas can be applied with minor changes to more fabrication technologies. In these approaches, the hardware co-processors implementing parts of the RTOS, are used for synchronization and communication between software and hardware. With the help of these hardware co-processors, the application execution time is reduced without affecting the RTOS execution time. Compared to those approaches, our goal is different: we aim at reducing the worst-case execution time of the RTOS, to accelerate the applications.

Other related approaches [11–13] execute the RTOS scheduling policy entirely in hardware. Compared to them, we leave the scheduling policy in software. In such a way, the system programmer is not restricted to any particular scheduling policy and can employ the most suitable one to fulfil the system specification. Nevertheless, if it is needed to have a configurable hardware scheduler, one might integrate the current proposal with [11].

Some other proposals [14,15] go even further in RTOS acceleration by completely implementing the RTOS in hardware. Due to the fact that RTOS is substituted by a complex Finite State Machine (FSM), the approach is less flexible and limited for future improvements, e.g., substituting scheduling policy or changing the model of computation.

To the best of our knowledge, we are the first to propose a hardware co-processor performing FIFO tracking and computing task-status for data-flow applications. Hereafter, we list the research projects that employ data-flow models of computation and transfer computationally intensive kernels in hardware. The Communication Assists (CA), by Shabbir et al. [16] and by Kyriacou et al. [17], are examples of hardware acceleration in the domain of embedded data-flow systems. In [16], user tasks are executed in non-preemptive mode without employing RTOS, where FIFO communication and FIFO management are entirely integrated in hardware. Moreover, Shabbir et al. [16] do not consider a processor–coprocessor paradigm, i.e., their CA is a stand-alone hardware accelerator. Therefore, the HWTSM might be used to augment their system, if task preemption is supported and RTOS is employed. The proposed CA in [17] provides only a Network on Chip (NoC) abstraction and memory management. It is employed to decouple communication from the computation without affecting the RTOS execution time. Both approaches [16,17] are orthogonal to ours in the sense that they accelerate different parts of the RTOS.

Tumeo et al. [18] propose a custom Direct Memory Access (DMA) controller. Similar to CA [16], the proposed DMA optimizes execution time of user tasks, by reducing the time cost involved in remote memory operations. The proposed HWTSM has a similar execution behaviour to DMAs, in a sense that both are running in parallel to the software. In fact, the DMA controller can be considered to be executed in what we refer as processor–coprocessor *parallel blocking* model, described in Section 4. Furthermore, the DMA signals back the processor by setting a flag and the DMA might have a buffer. The DMA buffer enables temporary storage of multiple DMA invocations. Therefore, their work is different than our new execution model, i.e., processor–coprocessor *parallel non-blocking*. Contrary to the DMAs, the HWTSM does not influence the communication cost, i.e., we are not directly accelerating the applications, but rather the RTOS execution. As a result, both, the DMAs and the HWTSM, could be employed together to improve system performance.

In the domain of execution models for coprocessors, Rupnow et al. [19] introduce three preemption methods for hardware accelerators, executed on reconfigurable logic. Contrary to our new execution model, they assumed that if a software thread is preempted, the associated hardware accelerator should be *blocked, dropped* or *rollbacked*. A *blocked* hardware accelerator is stalled until the corresponding software thread is activated again. The generated output of a *dropped* hardware accelerator is discarded. Finally, a *roll-backed* hardware accelerator is restated from the software when the software thread again becomes active. Compared to our classification of processor–coprocessor, their approach falls into what we refer as a processor–coprocessor *sequential* execution model.

Lange et al. [20] propose an embedded system running real-time Linux. Their system is composed of a single processor combined with multiple hardware accelerators. One of the contributions is an execution model for hardware accelerators. Seidel et al. [21] also proposed an execution model between a processor and coprocessor, which masks and processes interrupts. Both execution models correspond to what we refer to as a processor–coprocessor *parallel blocking* model. Therefore, their work is not related to our contributions, which consider a different processor–coprocessor execution model.

3. Motivation and concurrent execution models

We assume that each application is composed by a set of tasks. The tasks are scheduled by an instance of a RTOS, running on each core in an MPSoC platform. Each task is executed in one or more constant time slots, denoted as application slots. Each application slot is followed by an RTOS slot in which the RTOS stores the current task context, schedules, and loads the next task. Because the system has to be predictable, the execution time of the RTOS needs to have bounds. Moreover, if the system needs to be composable, then the size of the RTOS time slot should be constant and equal or longer than the WCET of the RTOS [3]. Our goal in this paper is to achieve high application performance while preserving the predictability and composability of the system. We improve the application performance by reducing the WCET of the RTOS.

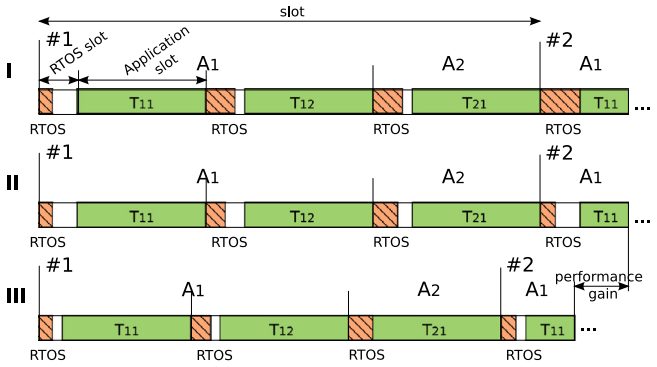

Fig. 1. RTOS & application execution scenarios. I. RTOS in SW; II. RTOS in SW/HW with slack; III. RTOS in SW/HW with performance gain.

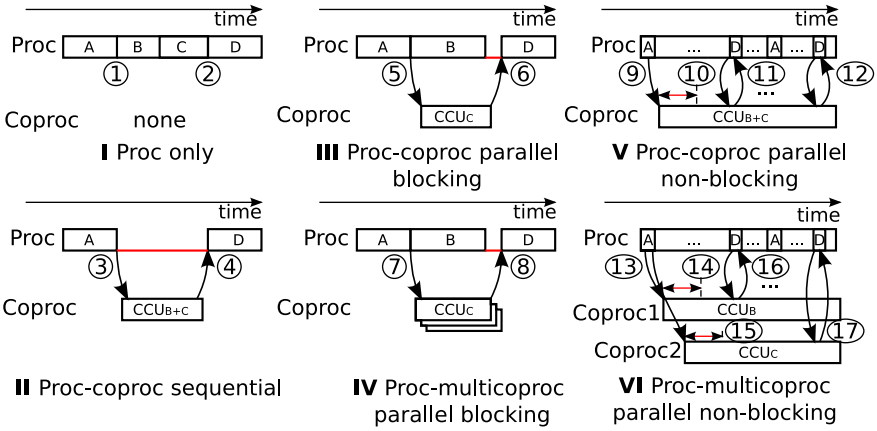

Fig. 2. Processor–coprocessor execution models.

3.1. Motivating example

Fig. 1 provides a motivating example by presenting the execution profile of a predictable and composable real-time embedded system in three scenarios. We assume that an RTOS schedules two applications - A_1 and A_2. Application A_1 comprises two tasks T_{11} and T_{12}. Application A_2 contains only one task - T_{21}. In Fig. 1.I, we present a scenario when user applications and RTOS kernel services are executed on a single processor. The white spaces indicate processor idle periods, denoted as *slack*, which occur when the RTOS finishes earlier than in its WCET. Fig. 1.II illustrates a case when at least one of the RTOS services, which has a highly variable in execution time, is transferred to hardware for acceleration. As a result, the WCET of the RTOS is minimized, which preserves predictability and composability, and improves performance. In Fig. 1.III, the RTOS time slot is made smaller due to the new reduced WCET of the RTOS. As a result of the short RTOS time slot, the overall system performance is improved. Note, that in all three scenarios, the size of the application slots is left unchanged.

We aim at minimizing the WCET of the RTOS. Two main procedures execute during the RTOS time slot: 1. context switching and 2. scheduling. The minimization of the context switching time is already solved in fine-grained simultaneous multithreading architectures, by having a dedicated register file for each one of the hardware contexts [22]. Thus we aim to improve only the RTOS scheduling procedure running on each of the cores of an MPSoC platform. The scheduling procedure is composed of two parts - computing the status of application tasks and the scheduling policy itself. In this paper, we aim to accelerate the computing task-status procedure in hardware, and preserve the flexibility to implement any scheduling policy in software.

3.2. Concurrent execution models

Once the functionality of the coprocessor is identified, the next step is to choose the synchronization model between the processor and the coprocessors as well as the execution model of the coprocessor. We split processor–coprocessor execution models in six categories based on the employed synchronization mechanism, as visible in Fig. 2. For the sake of the example we consider four computationally intensive kernels, represented as A, B, C, and D. In the discussion that follows, we assume that only B and C are accelerated on coprocessors. Below, we describe each processor–coprocessor execution model:

I *Processor only*, as presented in Fig. 2.I, is used as reference. In Fig. 2.I, the *B* starts at instance ① and *C* finishes at instance ②.

II *Processor–coprocessor sequential*, as depicted in Fig. 2.II, is commonly used to accelerate various computation intensive kernels in a coprocessor. To preserve the functional consistency, a software application is blocked after its coprocessor has started. In Fig. 2.II, at instance ③, the coprocessor is executed on a hardware Custom Computing Unit (CCU_{B+C}). When CCU_{B+C} finishes execution, it returns the program control to the processor, illustrated in Fig. 2.II, at instance ④. Depending on the duration of the coprocessor execution and the system requirements, CCU_{B+C} can generate an interrupt or can raise a flag on which the application waits or polls.

III *Processor–coprocessor parallel blocking*, as presented in Fig. 2.III, allows concurrent execution of a processor (software) and a coprocessor (hardware). With the help of hardware synchronization blocks, like those described in [5], the CCU_C is started at instance ⑤ and synchronized with processor at instance ⑥. In Fig. 2. III, the software functionality, denoted as *B*, finishes earlier than CCU_C. To preserve the application consistency, the software remains blocked until CCU_C completes its execution. Alternatively, *B* can finish later than CCU_C.

IV *Processor–multicoprocessor parallel blocking*, as presented in Fig. 2.IV, is an extension of the model in Fig. 2.III, with multiple coprocessors executed in parallel. With the help of processor-multiprocessor parallel blocking, we gain performance from hardware acceleration and parallelism. In Fig. 2.IV, at instance ⑦, the coprocessors are started. At instance ⑧, only after all coprocessors are finished, the software execution is resumed. Cases II and IV in Fig. 2 are in essence the *task-sequential* and *task-parallel* modes in [5], respectively.

V *Processor–coprocessor parallel non-blocking*, as introduced in Fig. 2.V, is one of the contributions in this paper. To our best knowledge, we are the first to apply this execution model in the processor–coprocessor context. Compared to the processor–coprocessor *sequential* and *parallel blocking* models, the software execution in the *parallel non-blocking* model is never blocked during the coprocessor execution. Note that once CCU_{B+C} is started, it finishes only at the request of the processor, i.e., the processor does not need to restart CCU_{B+C} every-time its results are needed. Therefore, in *parallel non-blocking* execution model, the cost to restart the CCU is entirely avoided. In Fig. 2.V, at instance ⑨, after CCU_{B+C} is started from the processor, CCU_{B+C} needs several cycles until the newly computed result is available at instance ⑩ in Fig. 2. Later, at instances ⑪ and ⑫ in Fig. 2.V, the processor reads back the CCU_{B+C} result. The CCU_{B+C} response time remains constant due to two reasons: (a) the coprocessor and the processor execute concurrently and (b) the coprocessor result is conservative. By conservative, we understand that even if the software reads back an old result, the application output is still correct. The only way to terminate CCU_{B+C}, is by an explicit software Application Programming Interface (API) call. Examples of modules capable to be executed in the *parallel non-blocking* model could be potentially any of the services in a real-time system, which tolerates slightly out-of-date information such as status sampling or delaying makable interrupts.

VI *Processor–multicoprocessor parallel non-blocking* in Fig. 2.VI is an extension to case *IV* from Fig. 2. In Fig. 2.VI, we attach multiple coprocessors to the processor, where each coprocessor runs in *parallel non-blocking* execution model. At instance ⑬, both CCUs, CCU_B and CCU_C, are started. The CCUs need several cycles until the newly computed result is ready, marked by instances ⑭ and ⑮. Later, at instances ⑯ and ⑰, the processor fetches the CCUs status.

4. RTOS acceleration: a HWTSM case study

In this section, we outline a potential usage of the processor–coprocessor execution models in various models of computation. We present as a case-study the HWTSM execution profile that follows the new processor-coprocessor execution model. Lastly, we demonstrate the processor-coprocessor execution models though multiple reconfigurable coprocessors integrated on a conceptual computing system.

4.1. Integration of the proposed processor-coprocessor execution models to models of computation

The “killer” performance applications for the contemporary real-time embedded devices such as mobile phones and smart TVs are encoding and decoding of various audio and video formats. Such applications are often referred to as streaming applications [23]. Recently, it becomes a common practice to implement streaming applications with data-driven programming paradigm [24]. In the data-driven programming paradigm, program statements describe the data to be matched and the processing required rather than defining a sequence of steps to be taken [25]. Furthermore, in data-driven programming, a streaming application, written in high-level abstraction language, is presented as a set of autonomous code segments.

In the domain of streaming applications, the real-time embedded system is often required to deliver predefined performance, i.e., to guarantee worst-case execution time (WCET). Therefore, researchers apply different set of restrictions on their data-driven programming models in order to improve the execution time analysis. Two popular data-driven models of computation are Kahn Process Network (KPN) [26] and data-flow [4]. In these models of computation, the autonomous computational code regions are referred to as *processes* (KPN) or *actors* (data-flow). Both entities have clearly defined input and output communication channels. Each communication channel is presented by a First In First Out (FIFO) queue. The synchronization is achieved by exchanging atomic data elements, called *tokens*, passed through the communication channels.

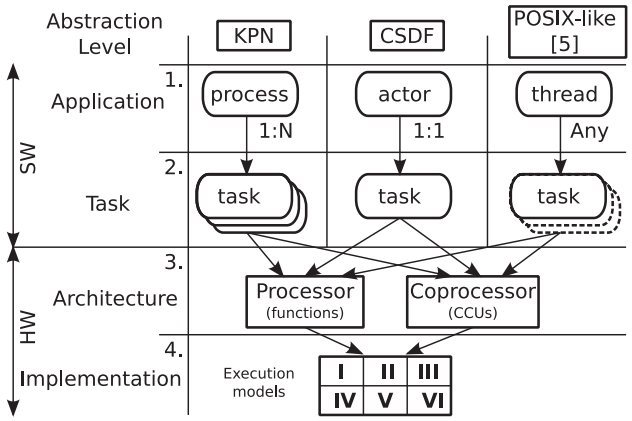

Fig. 3. Processor-coprocessor execution models in KPN, CSDF, and [5] models of computation.

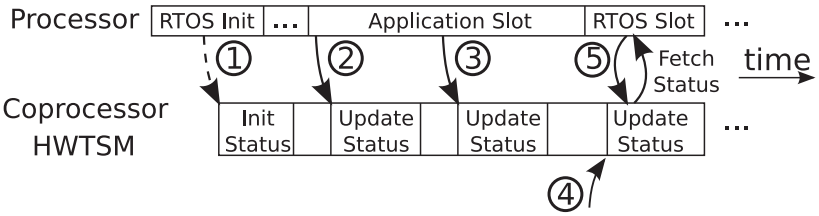

Fig. 4. HWTSM execution profile.

In Fig. 3, we present the mapping of the processor-coprocessor execution models to the Kahn Process Networks (KPN), Cycle-Static Data-Flow (CSDF), and a POSIX-like model of computation [5]. A KPN *process* intersperses computation and synchronization sections, where computation sections are with a variable size. Therefore, a process can have one or multiple *tasks*. The mapping between process and task is 1:N. The basic entity of the CDSF is the *actor*. An *actor* always have read, compute, and write sections, i.e., *actor* has one *task* for a specific input *tokens* and *firing rule* setup. Since an *actor* has one *task*, we define the mapping between an *actor* and a *task* to be 1:1. Zaykov and Kuzmanov in [5] propose that threads are composed of one or multiple tasks. Furthermore, thread communication and synchronization is not limited to FIFO channels only. For example, it could be done through complex data structures using semaphores and mutexes. Moreover, multiple tasks within a thread can be executed in parallel, exploiting the intra-thread parallelism. Therefore, we identify that the mapping between thread and task can either be 1:1 or 1:N.

Independently from the applied level of thread parallelism, a task can be executed on one of the two types of computing resources – either on the processor, or on one of the reconfigurable coprocessors. If a task is executed on the processor, then we refer to it as to a software function. If it is executed on a coprocessor, i.e., an reconfigurable logic, then we refer to it as to a CCU task.

4.2. Proposed execution model for the HWTSM

In Fig. 4, we detail the execution profile of the proposed HWTSM for one processor core. At instance ①, the HWTSM receives the FIFO configuration parameters, during the RTOS Initialization phase. Since the *parallel non-blocking* processor–coprocessor model is employed, tracking and computing the status of tasks is performed in parallel to the software execution. In our example, at instances ② and ③, tasks read/write from/into their input/output FIFOs. This indirectly triggers the calculation of a new task-status by the HWTSM, as indicated by arrows in Fig. 4. The HWTSM computational intervals are marked by *update status*, and are overlapped with the execution of the user tasks. The status update may also be triggered by remote read/write into a FIFO from a task mapped on another core, as exemplified at instance ④. Later, at instance ⑤, the RTOS scheduler fetches the task-status from HWTSM. As a result, HWTSM can be viewed as a high-level status register.

The status returned by the HWTSM leads always to correct application execution, i.e., task-status update is conservative. It is not possible that the RTOS sees a task as eligible whereas in fact the task is ineligible for execution, for the following reasons. Producer and consumer tasks can make each other eligible for execution. For example, when a producer task writes data into a FIFO (the FIFO free space is reduced), a consumer task becomes eligible for execution. Furthermore, when a consumer task reads data from a FIFO, it creates free space for the producer task to write into. When a task is eligible for execution, no other task can change its eligibility. A task can only change its status from eligible to ineligible by consuming or/and producing data. And this can only be done when the task is executed on a processor. Thus the only corner case that

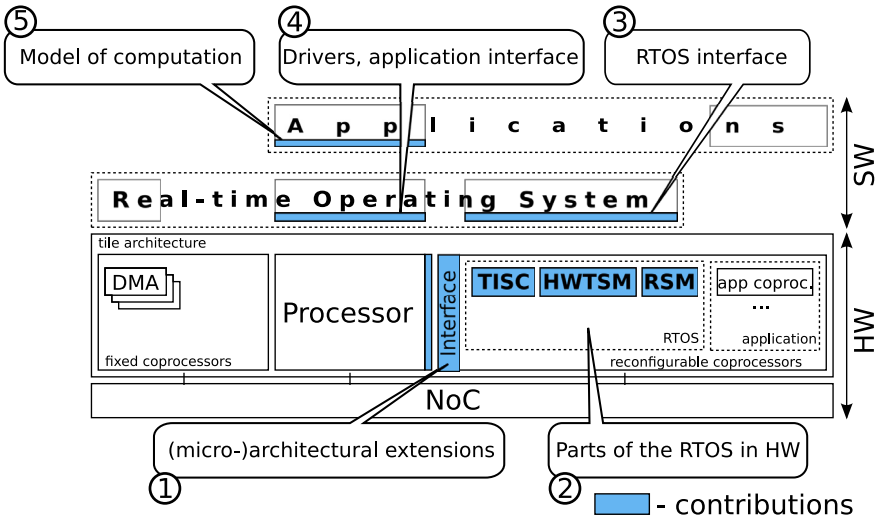

Fig. 5. Conceptual computing system extended with our contributions.

may lead to incorrect execution is when a task was eligible, it changes to ineligible, but the HWTSM does not detect that fast enough. Such a case is application safe, because a different task may be scheduled, and the 'eligible' status will be read in the next RTOS slot.

4.3. Computing system extension with HWTSM

In Fig. 5, we introduce hardware and software of a conceptual computing system extended with our contributions. As an example of the conceptual computing system, we choose a Multiprocessor System-on-Chip (MPSoC). The exemplary MPSoC is composed of tiles, connected through a Network on Chip (NoC). Each tile has a processor (e.g. RISC core), instruction and data memory, and two types of coprocessors - fixed (e.g. DMA controller) and reconfigurable (e.g. HWTSM).

From a software perspective, we consider multiple user applications to be executed on the conceptual MPSoC. Furthermore, we assume that the computing resources in the tile processor can be shared in time among multiple applications. We deliver the temporal management through an instance of a Real-Time Operating System (RTOS).

In Fig. 5, we illustrate our contributions in shaded blocks. We introduce a set of (micro-)architectural extensions (see ①) to support the various processor-coprocessor execution models. As a result of our approach, reconfigurable coprocessors are shared among RTOS services and user applications. Furthermore, we introduce parts of the RTOS in hardware (see ②), i.e, Hardware Task-Status Manager (HWTSM) [6], Thread Interrupt State Controller (TISC) [5], and Remote Slack Manager (RSM) [7], and the corresponding RTOS interface (see ③). The TISC operates in processor–multicoprocessor parallel blocking model, while HWTSM CCU and RSM CCU operate in parallel non-blocking. These RTOS reconfigurable coprocessors are accessible through RTOS drivers and application interface (see ④). At application level, we provide support for processor-coprocessor execution models in various models of computation (see ⑤).

5. Base hardware platform and system implementation

In this section, we describe the baseline tiled-platform, the sequence of steps in data-flow model of computation, and HWTSM implementation details.

5.1. Background - baseline platform

We employ the tiled CompSoC platform [3] as a baseline template for our multiprocessor design. More specifically, we employ the organization of the tiles presented in [27]. Each tile contains one processor core and multiple local memory modules. In Fig. 6, we present a simplified top-view of the CompSoC platform. In this particular implementation, the system is configured with two tiles connected through an dAElite NoC [28]. The local data memory in each of the tiles is organized in three blocks. The first one is *Dmem* which is employed for local data storage only. The second and the third ones, *Cmem.In* and *Cmem.Out* respectively, are dual-port memories, used for inter-tile communication. The *Imem* is used for storing the applications and RTOS executable binaries. All these memories are accessible by the processor [27].

Each user application is partitioned into tasks, following a data-flow [4] application model. The data-flow graph is mapped on an MPSoC, thus multiple tasks might be running and communicating to each other in parallel. The communication between the tasks is performed through FIFO circular-queues that are memory-mapped and implemented in software using the C-HEAP protocol [29]. Reading and writing in a circular FIFO is implemented with a read counter (*rc*) and a

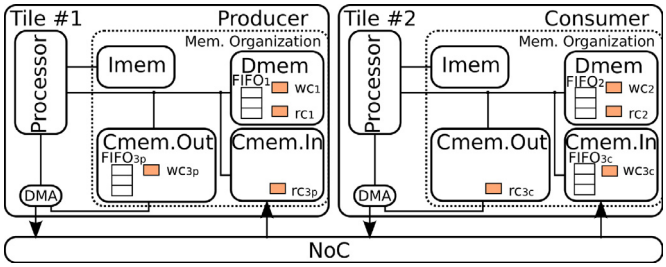

Fig. 6. Baseline CompSoC architecture.

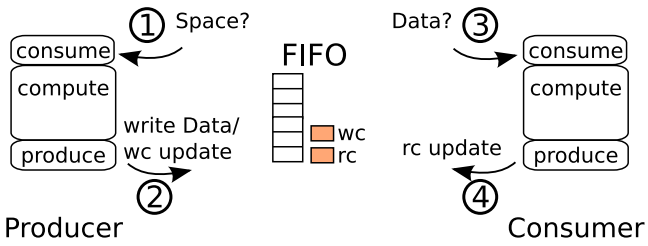

Fig. 7. Producer–consumer implementation of data-flow model of computation.

write counter (*wc*). Thus the amount of data in the queue, hence the task status, is determined by the values of these two counters. The CompSoC platform is designed to be predictable and composable. These characteristics are delivered by the hardware and the light-weighted RTOS called CompOSe [30]. The RTOS provides two-level scheduling, *intra-application* and *inter-application*, on each core.

In Fig. 7, we list the sequence of steps during the communication of two data-flow tasks through a FIFO. A producer is a task, which writes tokens to a FIFO and a consumer is a task, which reads those tokens. Furthermore, the FIFO is implemented by read and write counters, and a circular queue. In Fig. 7 at instance ①, the producer task checks for its firing rules (i.e, whether there is sufficient space in the FIFO) by: queue_size – (*wc*–*rc*) $\geq$ req_space, where req_space is the requested space for the tokens to be written. If the requested space is available, the producer task proceeds to the computation operation. At instance ②, producer task completes its computation and writes token(s) and updates wc to the FIFO. Later, at instance ③, the consumer task checks for its firing rules (i.e., whether there is data available) by: *wc*–*rc* $\geq$ req_size, where req_size is the required size (e.g. number of tokens) for a single task iteration. If the condition is satisfied, the consumer task proceeds to the compute operation. At instance ④, during produce operation, the consumer updates the rc to the FIFO.

In this work, we augment the CompSoC platform with hardware modules using a Molen-style processor–coprocessor design [31]. More specifically, we employ the microarchitecture from [5] employed to solve a different, yet related problem, namely management of multiple threads on reconfigurable hardware. We chose the Molen paradigm, because it provides architectural means to efficiently accommodate any software computationally intensive kernel in hardware. Each kernel might be accelerated by one or multiple Molen-style CCUs. In the current paper, the HWTSM is implemented as a single Molen-style CCU. The CCU is controlled through a fixed set of additional instructions [32]. The data transfers to and from the coprocessor are performed through dedicated exchange registers (XREGs).

5.2. System implementation overview

In Fig. 8, we present a conceptual CompSoC platform extended with one HWTSM per processor tile. The applications executed on the platform in Fig. 8, are those considered in the example of Fig. 1. Application A_1 is mapped on both tiles, while A_2 is mapped on the second tile only. We implement each one of the task-status units, denoted as T_{11}, T_{12}, and T_{21} as separate hardware blocks. These blocks are responsible for computing and updating the status of the assigned software tasks. As a result of this design choice the HWTSM preserves the composability and predictability of the platform, as explained in what follows.

First, the hardware task-status units do not need to exchange information among each other, thus are completely independent. Plus concurrent computation of the task statuses is possible. These tasks may even belong to different applications. Thus, the HWTSM does not create inter-application interference, hence it preserves composability.

Second, the RTOS fetches the task-status by simply reading the registers of the HWTSM. Hence the RTOS perceives the response time of the HWTSM as constant. Moreover, this time is shorter than in a software implementation that would involve a sequential calculation of the available data/space in each FIFO, in turn. Thus the predictability of the RTOS is preserved, and its WCET is reduced when compared to a software solution. Furthermore, the response time of the HWTSM

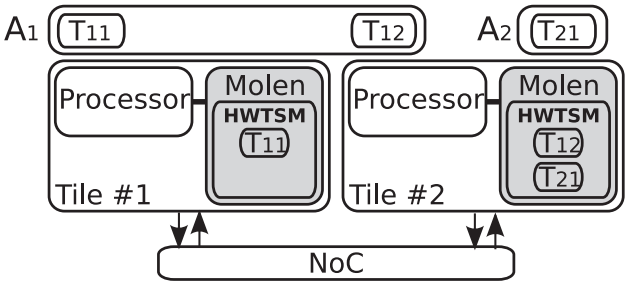

Fig. 8. Conceptual MPSoC with HWTSM.

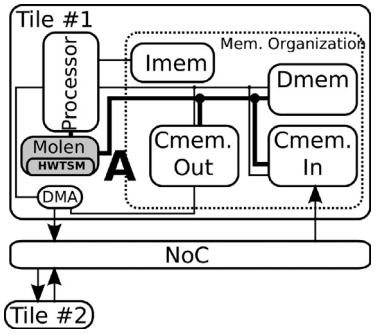

Fig. 9. HWTSM integration to CompSoC – option A.

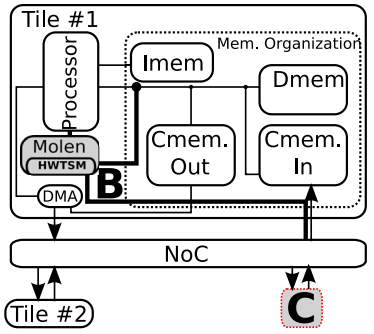

Fig. 10. HWTSM integration to CompSoC – options B, C.

is independent of the task-status and of the number tasks. Therefore, we can scale the number of the hardware task-status units, while the WCET of the RTOS remains constant, which means that the worst-case bounds on the RTOS are lower than in a software implementation. In addition, it is worth to mention that the HWTSM does not require any modifications to the existing NoC.

5.3. Tile microarchitecture modifications

The shaded blocks in Figs. 9 and10 present Molen-style coprocessors attached to the baseline CompSoC multicore platform. Based on the platform characteristics, we distinguish three possible options for the HWTSM integration. In the first option, denoted by A in Fig. 9, the HWTSM is attached through a dedicated memory port to each one of the three data memory types, i.e., HWTSM reads memories without affecting the processor. As a result, the dual-port *Dmem, Cmem.In*, and *Cmem.Out* need to be substituted by three-port memories. We do not consider this option as a viable solution in our design, because multiport memories are expensive in terms of hardware resources. By multiport memories, we denoted those with more than two ports. In the second option, denoted by B in Fig. 10, the HWTSM is attached to the data memory and to the NoC input buses. Such configuration enables the HWTSM to be executed in the *parallel non-blocking* model, which allows it to track FIFO updates (i.e., snoop the communication) at the exact moment they occur on the corresponding buses. Therefore, the integration of the HWTSM in the existing CompSoC platform is accomplished with a minimum number of modifications. The third option, denoted by C in Fig. 10, combines all HWTSMs into one dedicated tile. We ignore option C,

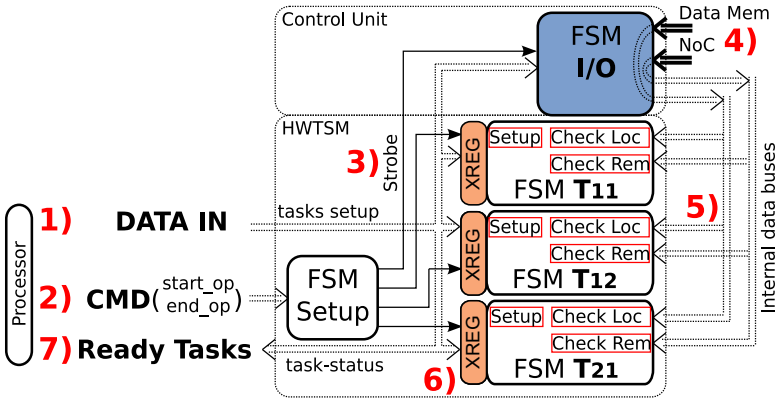

Fig. 11. HWTSM internal organization.

because updating and obtaining the task-status over the NoC causes immense delays amounting to hundreds of cycles. These delays can substantially increase the execution of the RTOS.

Summarizing, we attach the HWTSM to the CompoSoC multicore platform following option B in Fig. 10.

5.4. Hardware task-status manager design

In Fig. 11, we present the internal organization of the HWTSM for the example of Fig. 1. For the operation of the HWTSM, three basic types of FSMs are involved: *FSM T**, *FSM I/O*, and *FSM Setup*. The *FSM T** contains the core functionality of the HWTSM. By *FSM T**, we refer to any of the *FSM* T_{11}, *FSM* T_{12}, and *FSM* T_{21} in Fig. 11. Each of the *FSM T** computes and stores the task-status information of T_{11}, T_{12}, and T_{21}, respectively. The *FSM T** has a tiny internal memory which preserves the *rc*/*wc* memory addresses, the size of the FIFOs and the number of data elements (tokens) per FIFO. The *FSM I/O* translates the external bus protocols, such as NoC interface, to internal buses shared among all Molen-style CCUs. In case of more complex compute architecture, we might have multiple interfaces, where each interface will be handled by a separate *FSM I/O*. We group all *FSM I/O* in one block called control unit. The *FSM setup*, is responsible for starting and terminating the execution of *FSM T**.

At run-time, the following events occur in order, as presented in Fig. 11:

1. A new configuration is transmitted to the *XREGs*;
2. The processor emits the Molen *exec* instruction [32], initiated by the *start_op* signal, indicating that there is a new *CCU* setup available in *XREG*;
3. The *FSM setup* redirects the *start_op* signal to the corresponding *FSM T** or *FSM I/O*. The configuration for the *FSM I/O* contains memory address locations of task's FIFOs. Once each *FSM T** is started, it continuously tracks the changes to all FIFOs connected to a task;
4. The *FSM I/O* snoops on NoC interface and local memory bus and detects write operations to the dedicated *rc*/*wc* memory ranges. The *FSM I/O* is a slave on the bus and it does not affect bus access timings;
5. If there is a write operation to the tracked memory ranges, then the *FSM I/O* generates a strobe signal and transmits the address and data values to the internal bus;
6. The updated task-status by *FSM T** is preserved in *XREG*;
7. During the RTOS time slot, the RTOS task scheduler fetches task-status information from the *XREG*;

5.5. RTOS extensions

In Fig. 12, we present the integration of the HWTSM to the CompOSe RTOS [30]. The HWTSM management is performed through a tiny driver that virtualizes the low-level interface. The following stages occur during the execution of the RTOS. In stage 1, during system initialization, all coprocessors are initialized and user applications are created. After this stage, the processor starts the first application time slot. In stage 2, the context of the scheduled task is loaded. The task reads input data from input FIFOs, performs some computations and writes back the results to the output FIFOs. In case an application is partitioned and distributed on multiple tiles, a DMA module could be used to send data and FIFO information over the NoC to the remote tile memory. When a FIFO is updated, the HWTSM instantly detects the *rc* and *wc* changes and updates task-status. The HWTSM operates identically for intra- and inter-tile communication. If a user task finishes earlier than its slot, the processor goes to an idle state. After an interrupt from the hardware timer is raised, in stage 3, the processor starts the RTOS time slot. The context of the running task is saved. The RTOS scheduler gets task-status information from the HWTSM. Based on it, the RTOS scheduler chooses one of the ready tasks according to its task scheduling policy. Note that stages 2 and 3 are repetitive.

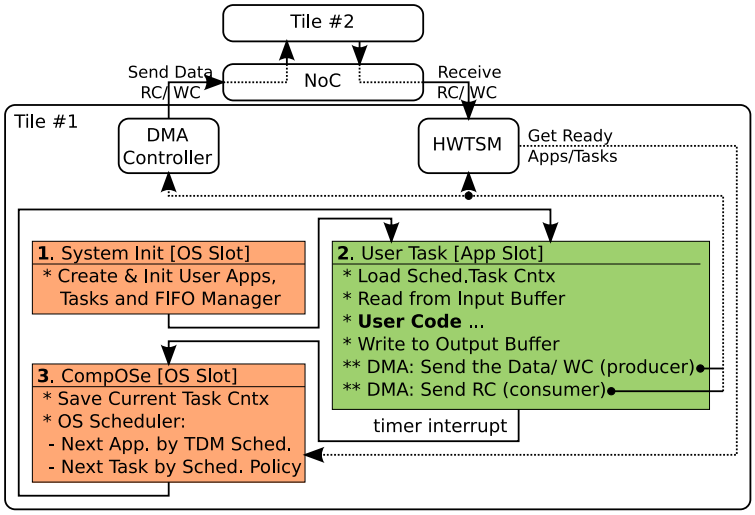

Fig. 12. CompOSe – application and RTOS time slots.

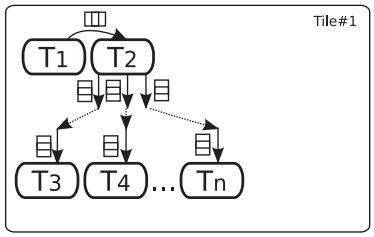

Fig. 13. Synthetic application for the StS policy scenarios.

6. Experimental results

We perform all experiments with a dual-tile CompSoC platform employing Xilinx Microblazes as a processor cores in the tiles. The design is synthesized using Xilinx Platform Studio 12.2 and verified on a Xilinx Virtex 5 ML510 (XC5VFX130T) evaluation board. We exercise with two synthetic and two popular applications in the embedded systems domain – an JPEG decoder and an H.264/AVC decoder. However, our results can be generalized for any arbitrary number of tiles, because the execution time of the RTOS in one processor tile depends only in the number of tasks executed locally on that tile and the number of FIFOs in those tasks.

We develop two synthetic applications to evaluate the HWTSM with two basic types of intra-application scheduling policies: Static Scheduling (StS) and Dynamic Scheduling (DyS). We explicitly investigate different scheduling policy types, because they are the ones which dictate the number of HWTSM calls. We generate our synthetic benchmark by varying the number of FIFOs per task and the total number of tasks, both in the range from 1 to 10. All FIFOs are configured to accommodate up to two tokens. For each one of the synthetic and real applications, we measure the WCET of the RTOS with and without the HWTSM included.

6.1. HWTSM performance with static scheduling policy

In the StS policies, such as Time Division Multiplexing (TDM), the next scheduled task is always the next one in a static table. In case the task is not ready, the corresponding application time slot is left idle. In Fig. 13, we present an inter-task communication pattern of the synthetic application that we use in a combination with StS policy. Let us assume that the currently executed task is T_1 and the next one to be scheduled is T_2. As a result, during the RTOS time slot, the StS policy only checks FIFOs associated with T_2. This why we create the StS scenarios by scaling the number of FIFOs associated with T_2.

The WCET of the RTOS with StS policy without using HWTSM is $WCET_{sts\_sw}$:

$$WCET_{sts\_sw} = T(n_f) + Const_{sched} + Const_{cntx}, \tag{1}$$

where n_f is the number of FIFOs associated with a task. $T(n_f)$ is the time to read the FIFOs *rc* and *wc* and compute the status of the task. The $Const_{sched}$ is the constant time for the scheduling policy. The $Const_{cntx}$ is the context switching time. As typically $T(n_f)$ is linear in n_f, $WCET_{sts\_sw}$ is a linear function, depending only on n_f.

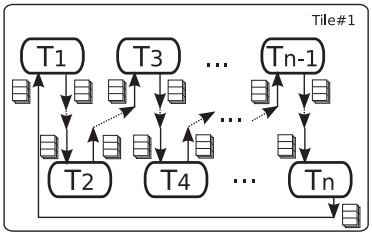

Fig. 14. Synthetic application for the DyS policy scenarios.

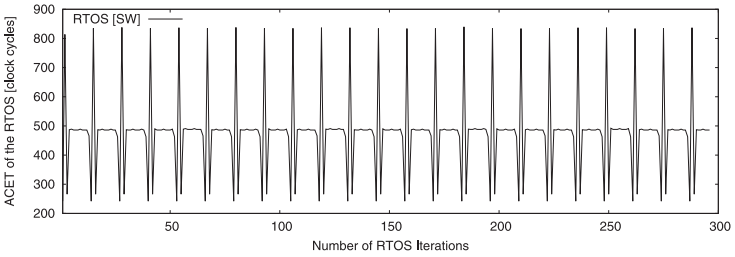

Fig. 15. ACET of the RTOS with StS policy for 10 tasks.

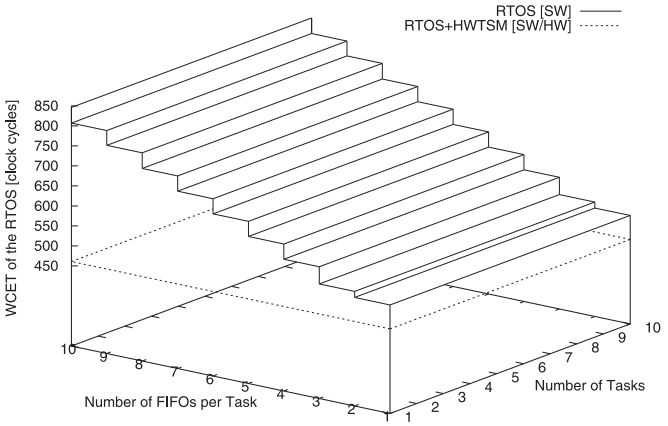

Fig. 16. WCET of the RTOS with StS policy.

The WCET of the RTOS using the HWTSM and the StS policy is $WCET_{sts\_hw}$:

$$WCET_{sts\_hw} = Const'_{sched} + Const_{cntx}, \tag{2}$$

where $Const'_{sched}$ is the constant time necessary to fetch the task-status from the HWTSM. Since the FIFO checking and task-status computing are performed in hardware, overlapped with the execution of the software, then the RTOS execution time is always constant.

In Fig. 15, we present the actual case execution time (ACET) of the pure software RTOS for an application that has 10 tasks and 9 FIFOs in total. The variations in the ACETs of the RTOS are due to the number of FIFOs in the task to be scheduled. A task with low number of FIFOs (such as T_1) requires less time to determine the task status than a task with high number of FIFOs (such as T_2). In Fig. 15, the RTOS actual case execution times mostly varies around 500 clock cycles, because the StS policy suggest to schedule a task with low number of FIFOs, which is not eligible for execution. Therefore, the RTOS StS policy loads the idle task. The WCET of the RTOS is equal to 846 clock cycles for the synthetic application presented in Fig. 13. Our analysis suggest that the WCET of the RTOS is experienced when T_2 is checked for execution eligibility, because T_2 is the task with the highest number of FIFOs in the application. As explained in [3], the composability of the system is ensured by leaving the processor idle for the time difference between the ACET of the RTOS and its WCET. In Fig. 15, the shortest execution time of the RTOS is equal to 266 clock cycles. Therefore, the idle period is equal to 620 clock cycles. Our goal is to reduce the WCET of the RTOS, as well as RTOS execution time variations, such that the idle period is minimized.

In Fig. 16, we present the WCET of the RTOS for the StS policy as a function of the number of tasks and FIFOs. The results are obtained with and without the HWTSM for the synthetic application from Fig. 13. As the analytical model for StS suggests, the variations of the WCET of the RTOS are close to linear, with respect to the number of FIFOs. The

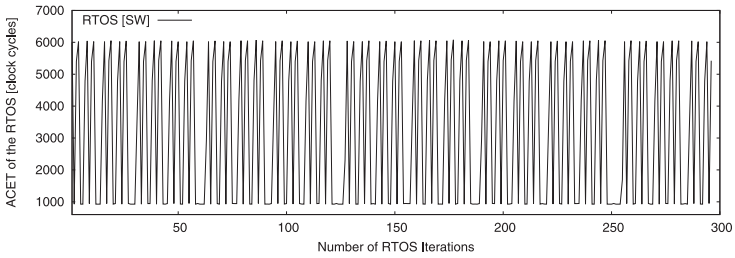

Fig. 17. ACET of the RTOS with DyS for 10 tasks and 10 FIFOs per task.

non-linear behaviour at two FIFOs is caused by the software implementation, because the processing of the input/output FIFOs is different.

As the analytical model for StS suggests, the WCET of the RTOS with HWTSM is always constant, equal to 461 clock cycles. The constant execution time is due to the StS execution profile and the proposed *parallel non-blocking* execution model. Based on the timings, we compute reduction in the WCET of the RTOS as a ratio between the pure software and HWTSM implementations. For StS policy, the results suggest reduction in the WCET of the RTOS between 1.1 and 1.8 times.

6.2. HWTSM performance with dynamic scheduling policy

In Fig. 14, we present the second synthetic application using DyS policy. In our experiments, we experiment the Round-Robin (RR) scheduling algorithm as an example of the DyS policies. Contrary to StS, the DyS policy might check multiple tasks until an eligible for execution one is found. Assuming that there is always at least one task eligible for execution, the WCET of the RTOS for a DyS policy occurs when all tasks belonging to the current application are checked and the next scheduled task is again the currently running one. Therefore, we construct the synthetic application for the DyS policy by varying the number of tasks and their FIFOs.

The WCET of the RTOS with DyS policy for pure software implementation is $WCET_{dys\_sw}$:

$$WCET_{dys\_sw} = T(n_f, m_t) + Tsched(m_t) + Const_{cntx}, \tag{3}$$

where $T(n_f, m_t)$ is the time to read the *rc* and *wc* for each of the FIFOs (n_f) associated with each task (m_t) and compute each task status. The $Tsched(m_t)$ and $Const_{cntx}$ are the scheduling time for DyS policy and the context switching time.

The WCET of the RTOS using HWTSM and the DyS policy is $WCET_{dys\_hw}$:

$$WCET_{dys\_hw} = T'sched_{hw}(m_t) + Const_{cntx}, \tag{4}$$

where $T'sched_{hw}(m_t)$ is time for the employed DyS policy when a HWTSM is used. Since the time for the FIFO management is omitted, the variations of the $WCET_{dys\_hw}$ are only due to $T'sched_{hw}(m_t)$. Therefore, if the DyS policy has linear complexity, the WCET of the RTOS also grows linearly.

In Fig. 17, similarly to the StS, we present the ACET of the RTOS of an application with 10 tasks and 10 FIFOs per task. Consequently, there will be a total of 100 FIFOs in the application. As it is visible in Fig. 17, the ACET varies from 1000 up to 6000 cycles. The ACET variations of the RTOS are due to the number of tasks checked until an eligible for execution one is found.

In Fig. 18, we present the WCET of the RTOS with DyS policy for a synthetic and two real applications. For the DyS policy scenario, we present the WCET of the RTOS as a function of the number of tasks and their FIFOs. The experimental results are obtained for six different scenarios with the number of FIFOs per task equal to 2, 4, 6, 8, and 10. As the analytical study for DyS suggests, the WCET of the RTOS for the pure software implementation grows in two dimensions defined by the number of tasks and FIFOs. For the DyS scenario with HWTSM, denoted as *RTOS+HWTSM* in Fig. 18, the WCET of the RTOS does not change when the number of FIFOs per task is increased. Furthermore, the execution time of the RTOS for *RTOS+HWTSM* is the same throughout all scenarios. It is because the HWTSM computes in parallel the tasks-status, which depends on the number of FIFOs per task. For the synthetic application with DyS policy, the reduction in the RTOS WCET is between 1.1 and 3.0 times.

We follow the implementation approach from [33] to partition and map the JPEG and the H.264 decoders on the baseline CompSoC platform. The application inter-task communication pattern of the JPEG decoder is presented in Fig. 19. We partition the JPEG decoder into three tasks. One of the tasks is mapped on one processor tile and the other two are running on the other processor tile. The application inter-task communication pattern of the H.264 decoder is depicted in Fig. 20. We partition the H.264 decoder into six tasks. Each of the processor tiles executes three of the tasks.

In Fig. 18, we also present the WCET of the RTOS in each one of the tiles for the JPEG and H.264 decoders with and without the HWTSM. Clearly, the WCET of the *RTOS+HWTSM*, illustrated with the solid ladder in Fig. 18, is lower than all cases of software only executions. For the JPEG decoder, the reduction of the WCET of the RTOS with HWTSM is up to 1.3 times. Although the H.264 decoder has an equal number of tasks in each tile, we observe small variations in the measurements, caused by the different number of the input/output FIFOs of the mapped tasks. The reason for the high WCET reduction

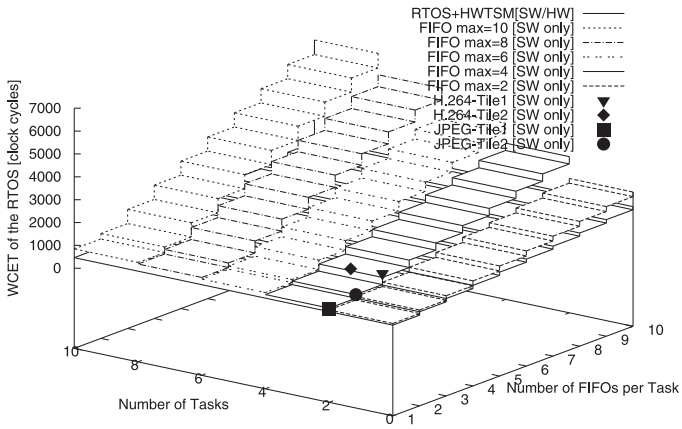

Fig. 18. WCET of the RTOS with DyS policy.

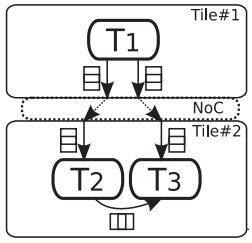

Fig. 19. JPEG decoder.

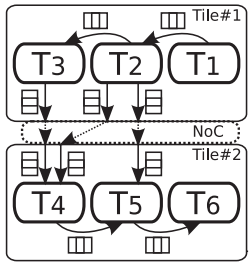

Fig. 20. H.264 decoder.

Table 1
HWTSM system performance improvement.

	RTOS slot : application slot ratio	
	10%: 90%	20%: 80%
StS policy	0.9% – 4.5%	1.8% – 8.9%
DyS policy	0.9% – 6.6%	1.8% – 13.3%
JPEG	2.3%	4.6%
H.264	3.8%	7.5%

of the RTOS even with low number of tasks is due to the *parallel non-blocking* execution model which leads to a constant, short in our case, response time of the HWTSM equal to five cycles.

In summary, the reduction in the WCET of the RTOS with HWTSM is up to 1.6 times.

6.3. HWTSM performance improvement

Table 1 presents the overall system performance when the WCET reduction of the RTOS is employed to speedup the application. We investigate two cases, when the ratio of RTOS to application slot size is 10%:90% and 20%:80%, respectively. As the reduction in the WCET of the RTOS slot varies between 1.1 to 3.0 times and the RTOS slot size is 10% of the total execution, the overall system performance improvement is between 0.9%-6.6%. If the RTOS slot size is 20% of the total

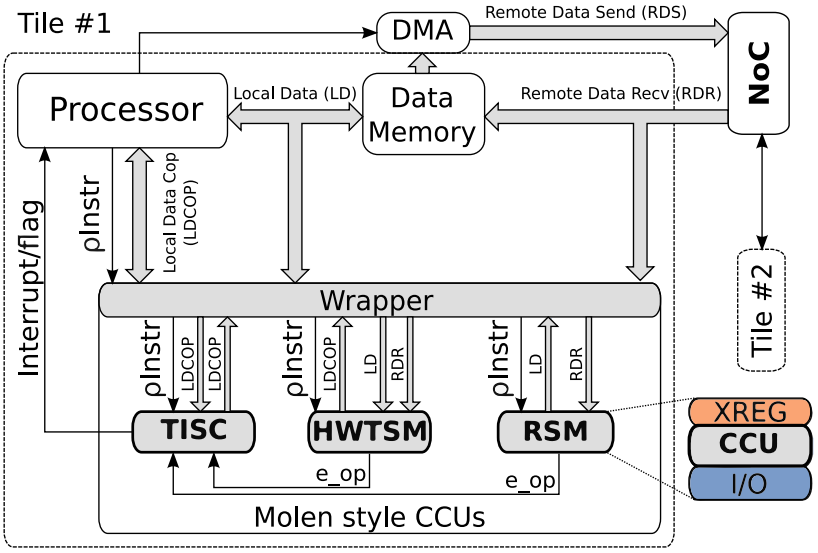

Fig. 21. Conceptual MPSoC extended with three Molen-style CCUs – TISC, HWTSM, and RSM.

execution time, then the overall system performance improvement is between 1.8% and 13.3%. If the number of tasks and FIFOs is further increased, we expect even higher improvement than the listed results.

Once the WCET of the RTOS is reduced, we might consider to: (a) improve the overall performance, (b) increase system responsiveness by increasing the rate of RTOS invocations, (c) a combination of the previous two.

6.4. Extended discussion

In this section, we discuss a conceptual architecture, which illustrates that all three previously discussed reconfigurable coprocessors, namely: TISC [5], HWTSM [6], and RSM [7] can be combined together. As depicted in Fig. 21, we connect the three reconfigurable coprocessors with buses. Local Data Coprocessor (LDCOP) is bus employed for the communication between the processor and reconfigurable coprocessors. Local Data (LD) bus is employed for communication between the processor and the local data memory. Remote Data Send (RDS) and Remote Data Receive (RDR) bus are employed for inter-tile communication, i.e., for receiving data from other tiles through the NoC. ρInstr is the fixed set of additional instructions [32] issued by the processor. Since we run HWTSM and RSM in processor–coprocessor *parallel non-blocking* execution model, their termination signals (denoted as end_op) are employed only during their explicit termination from the software and not during their normal operation.

Based on the available experimental results, we estimate that the improvement in the system speedup can be up to 19.6 times with the help of the TISC [5]. Furthermore, we reduce RTOS cost with the help of the HWTSM, which results in additional application acceleration of up to 13.3% [6]. Last but not least, the improvement of the system energy consumption can be up to 56.7% over current state of the art with the help of inter-tile remote slack information distribution framework [7]. Overall, with the help of three coprocessors, the system performance is improved, the predictability and composability are preserved, all with reduced energy consumption.

7. Conclusions

In this paper, we proposed an implementation of time-consuming RTOS parts in hardware as reconfigurable coprocessors. These coprocessors are executed in a set of processor-coprocessor execution models. We provided an overview of a potential integration of processor–coprocessor execution models to two data-driven models of computation and a model of computation similar to POSIX. Furthermore, we discussed the proposed contributions on a conceptual computing platform, where the processor-coprocessor execution models are applied on three reconfigurable coprocessors – Hardware Task-Status Manager (HWTSM), Thread Interrupt State Controller (TISC), and Remote Slack Manager (RSM). For the HWTSM, we applied processor–coprocessor *parallel non-blocking* execution model, which allows overlapping of the coprocessor operation with the processor operation. As a result, we achieved shorter WCET of the RTOS while preserving the predictability and composability of the original MPSoC. Our proposal is integrated into the existing CompSoC MPSoC platform and this entire system is prototyped on FPGA chip. For the TISC, we employed processor–coprocessor *parallel blocking* execution model, which minimize the thread synchronization costs. The RSM operated in processor–coprocessor *parallel non-blocking* execution model. With the help of the execution model, the RSM distributed the slack information between the tiles in an MPSoC while preserving the predictability and composability. The experimental results are obtained with synthetic and real

applications. Based on the experimental results, we can conclude that implementing time-critical RTOS parts on reconfigurable coprocessors by using the proposed processor-coprocessor execution models can substantially improve the overall system performance. Furthermore, the presented approach preserves system predictability and composability. Finally, there is evidence of a lower energy profile of the MPSoC, which needs to be investigated further.

References

[1] Kopetz H. Real-time systems: design principles for distributed embedded applications. Springer; 2011.
[2] Goossens K, Azevedo A, Chandrasekar K, Gomony MD, Goossens S, Koedam M, et al. Virtual execution platforms for mixed-time-criticality systems: the CompSOC architecture and design flow. ACM SIGBED Rev 2013;10(3):23–34.
[3] Akesson B, Molnos A, Hansson A, Ambrose Angelo J, Goossens K. Composability and predictability for independent application development, verification, and execution. Multiprocessor system-on-chip – hardware design and tool integration. chap 2 Circuits and Systems; 2010.
[4] Lee EA, Parks T. Dataflow process networks. Proc IEEE 1995;83(5):773–801.
[5] Zaykov PG, Kuzmanov GK. Multithreading on reconfigurable hardware: an architectural approach. Microprocess Microsyst (MICPRO) 2012;36(8):695–704.
[6] Zaykov PG, Kuzmanov G, Molnos A, Goossens K. Hardware task-status manager for an RTOS with FIFO communication. In: Proceedings of the int'l conference on ReConFigurable computing and FPGAs (ReConFig); 2014. p. 1–8.
[7] Zaykov PG, Kuzmanov GK, Molnos AM, Goossens KGW. Run-time slack distribution for real-time data-flow applications on embedded MPSoC. In: Proceedings of the Euromicro conference on digital system design (DSD); 2013. p. 39–47.
[8] Lübbers E, Platzner M. ReconOS: an RTOS supporting hard- and software threads. In: Proceedings of the int'l conference on field-programmable logic and applications (FPL); 2007. p. 441–6.
[9] Chandra S, Regazzoni F, Lajolo M. Hardware/software partitioning of operating systems: a behavioral synthesis approach. In: Proceedings of the int'l conference on great lakes symposium on VLSI (GLSVLSI); 2006. p. 324–9.
[10] Nordström S. Hardware support for real-time systems - an overview. Tech. Rep.; 2004.
[11] Kuacharoen P, Shalan MA, III VJM. A configurable hardware scheduler for real-time systems. In: Proceedings of the int'l conference on engineering of reconfigurable systems and algorithms (ERSA); 2003. p. 96–101.
[12] Kohout P, Ganesh B, Jacob B. Hardware support for real-time operating systems. In: Proceedings of the int'l conference on hardware/software codesign and system synthesis (CODES+ISSS); 2003. p. 45–51.
[13] Burgio P, Tagliavini G, Conti F, Marongiu A, Benini L. Tightly-coupled hardware support to dynamic parallelism acceleration in embedded shared memory clusters. In: Proceedings of the int'l design, automation and test in Europe conference and exhibition (DATE); 2014. p. 1–6.
[14] Nakano T, Utama A, Itabashi M, Shiomi A, Imai M. Hardware implementation of a real-time operating system. In: 12th TRON project international symposium; 1995. p. 34–42.
[15] Ong SE, Lee SC, Ali N, Hussin F. SEOS: hardware implementation of real-time operating system for adaptability. In: Proceedings of the int'l symposium on computing and networking (CANDAR); 2013. p. 612–16.
[16] Shabbir A, Kumar A, Stuijk S, Mesman B, Corporaal H. CA-MPSoC: an automated design flow for predictable multi-processor architectures for multiple applications. J Syst Arch (JSA) 2010;56(7):265–77.
[17] Kyriacou C, Evripidou P. Communication assist for data driven multithreading. In: Advances in informatics. In: LNCS, 2563; 2003. p. 351–67. Berlin, Heidelberg
[18] Tumeo A, Monchiero M, Palermo G, Ferrandi F, Sciuto D. Lightweight DMA management mechanisms for multiprocessors on FPGA. In: Proceedings of the int'l conference on application-specific systems, architectures and processors (ASAP); 2008. p. 275–80.
[19] Rupnow K, Fu W, Compton K. Block, drop or roll(back): alternative preemption methods for RH multi-tasking. In: Proceedings of the int'l conference on field-programmable custom computing machines (FCCM); 2009. p. 63–70.
[20] Lange H, Koch A. Architectures and execution models for hardware/software compilation and their system-level realization. IEEE Trans Comput 2010;59:1363–77.
[21] Seidel H, Robelly P, Herhold P, Fettweis G. A low power soft core for multi-standard video applications. In: Proceedings of the int'l global signal processing expo and conference (GlobalSIP); 2005. p. 1–8.
[22] Levy HM, Lo JL, Stamm RL, Eggers SJ, Emer JS, Tullsen DM. Simultaneous multithreading: a foundation for next-generation processors. IEEE Micro 1997;17:12–18.
[23] Thies W, Karczmarek M, Amarasinghe SP. Streamit: a language for streaming applications. In: Proceedings of the int'l conference on compiler construction (CC); 2002. p. 179–96.
[24] Bic L, Lee C. A data-driven model for a subset of logic programming. ACM Trans Program Lang Syst 1987;9:618–45.
[25] Nejad AB, Molnos A, Goossens K. A unified execution model for data-driven applications on a composable MPSoC. J Syst Arch - Embedded Syst Design 2013;59(10-C):1032–46.
[26] Kahn G. The semantics of a simple language for parallel programming. In: Proceedings of the int'l federation for information processing (IFIP) congress; 1974. p. 471–5.
[27] Ambrose J, Molnos A, Nelson A, Goossens K, Cotofana S, Juurlink B. Composable local memory organisation for streaming applications on embedded MPSoCs. In: Proceedings of the int'l conference on computing frontiers (CF); 2011. 23:1–23:2.
[28] Stefan R, Molnos A, Goossens K. dAElite: a TDM NoC supporting QoS, multicast, and fast connection set-up. IEEE Trans Comput 2014;63(3):583–94.
[29] Nieuwland A, Kang J, Gangwal OP, Sethuraman R, Busá N, Goossens K, et al. C-HEAP: a heterogeneous multi-processor architecture template and scalable and flexible protocol for the design of embedded signal processing systems. Design Autom Embedded Systems 2002;7(3):233–70.
[30] Hansson A, Ekerhult M, Molnos A, Milutinovic A, Nelson A, Ambrose J, et al. Design and implementation of an operating system for composable processor sharing. Microprocess Microsyst (MICPRO) 2011;35:246–60.
[31] Vassiliadis S, Wong S, Cotofana SD. The MOLEN $\mu\rho$-coded processor. In: Proceedings of the int'l conference on field programmable logic and applications (FPL); 2001. p. 275–85.
[32] Vassiliadis S, Wong S, Gaydadjiev GN, Bertels K, Kuzmanov GK, Panainte EM. The Molen polymorphic processor. IEEE Trans Comput 2004;53:1363–75.
[33] Nguyen B. Task scheduling methods for composable and predictable MPSoCs. TUD; 2010. Master's thesis.

Pavel G. Zaykov received his M.Sc. at the Department of Computer Systems and Technologies at the Technical University of Sofia, Bulgaria, in 2007, and Ph.D. at Computer Engineering laboratory at Delft University of Technology, The Netherlands, in 2014. Since 2012, he is an R&D scientist at Honeywell International, Czech Republic. Pavel's research interests include multithreading architectures, reconfigurable computing, embedded system design, real-time operating systems, and multiprocessor-systems-on-chip.

Georgi Kuzmanov was born in Sofia, Bulgaria. He received the M.Sc. degree in computer systems from the Technical University of Sofia, Bulgaria, in 1998, and the Ph.D. degree in computer engineering from Delft University of Technology (TU Delft), Delft, The Netherlands, in 2004. Between 1998 and 2000, he was an R&D engineer with "Info MicroSystems" Ltd., Sofia. After a short service as a postdoc, he served as a full-time assistant professor at TU Delft between 2006 and 2011. Between 2011 and 2014, Dr. Kuzmanov took position of a program officer at ARTEMIS-JU, Brussels and since 2014 he has been a programme officer at ECSEL-JU, Brussels. He is currently affiliated with the Computer Engineering Laboratory, TU Delft. His research interests include recongurable computing, media processing, computer arithmetic, computer architecture and organization, scientific computing, and embedded systems.

Anca Molnos received the M.Sc. degree in computer science from the "Politehnica" University of Bucharest, Romania in 2011 and the Ph.D. degree in computer engineering from the Delft University of Technology, Netherlands, in 2009. Between 2006 and 2009 she worked at NXP Semiconductors, Netherlands, on low-power multi-processors and parallel, distributed real-time systems. The next 3 years she was a postdoc researcher at Delft University of Technology investigating composable, predictable multi-core embedded platforms and embedded multi-core resource management for low-power and quality of service. In 2013 she joined CEA-Leti where she is currently active on variability management and run-time adaptivity for many-core platforms where guaranteed performance, energy consumption, and heating are crucial issues.

Kees Goossens received his Ph.D. in Computer Science from the University of Edinburgh in 1993 on hardware verification using embeddings of formal semantics of hardware description languages in proof systems. He worked for Philips/NXP Research from 1995 to 2010 on networks on chip for consumer electronics, where real-time performance, predictability, and costs are major constraints. He was part-time professor at Delft university from 2007 to 2010, and is now full professor at the Eindhoven university of technology, where his research focusses on composable (virtualised), predictable (real-time), low-power embedded systems, supporting multiple models of computation. He published 3 books, 100+ papers, and 24 patents.