2013 16th Euromicro Conference on Digital System Design

Run-time slack distribution for real-time data-flow applications on embedded MPSoC

Pavel G. Zaykov, Georgi Kuzmanov
Computer Engineering Lab,
Delft University of Technology,
The Netherlands
{*P.G.Zaykov, G.K.Kuzmanov*}*@tudelft.nl*

Anca M. Molnos
CEA LETI,
Grenoble, France
{*Anca.Molnos*}*@cea.fr*

Kees Goossens
Embedded Systems Group,
Eindhoven University of Technology,
The Netherlands
k.g.w.goossens@tue.nl

***Abstract*—Low energy consumption is crucial for embedded systems, including the ones that employ tiled Multiprocessor Systems-on-Chip (MPSoC). Such systems often execute real-time applications consisting of several tasks synchronized in a data-flow manner and mapped over different MPSoC tiles. Energy can be saved by lowering the processor voltage and frequency, hence extending the application execution over periods of time otherwise left idle, i.e., exploiting slack. In this paper we propose a framework to distribute slack information at run-time, intra- and inter-tile, to enable accurate and conservative slack calculation within each tile. The slack is transferred along with the existing inter-task synchronization and as a result it is distributed across the MPSoC with low overhead. In each tile, we add a hardware block that calculates the slack received during inter-tile communication and a software library to program this hardware. We integrate this framework into an existing MPSoC platform and we prototype an entire system with two tiles on an Xilinx ML605 FPGA board. We demonstrate the effectiveness of our proposal with a simple, conservative, DVFS management policy applied to an H.264 decoder application. The experimental results suggest that our framework reduces the total energy consumption of tiles with 27%, when compared to a state-of-the-art intra-tile approach that uses a similar management policy. Our proposal introduces only a minor software overhead of up to 4% over the application execution time and negligible additional FPGA chip utilization of 0.002%.**

***Keywords*-Inter-tile Slack Framework; Timestamps; MPSoC; RTOS;**

I. Introduction

Nowadays, many modern real-time embedded systems execute computationally intensive, streamming applications. To achieve the desired performance, application designers exploit available concurrency by encapsulating each of the computationally intensive kernels into tasks that may communicate. Many such applications are mapped on a tile-based MPSoC, where tiles are connected through a Network-on-Chip (NoC). At run-time, in each tile, the application tasks are scheduled by a Real-Time Operating System (RTOS).

Real-time embedded systems have to be predictable and often should operate within a limited energy budget. A system is predictable if it is possible to accurately characterize its performance. An important performance metric for streamming applications is throughput, i.e., number of output data items produced per unit of time. Typically throughput is guaranteed by analysing, at design-time, the critical execution paths of the application under worst-case assumptions [1], [2]. At run-time, each task is invoked repeatedly for an undetermined number of iterations. Such systems may have two basic types of slack: static and dynamic [3]. Static slack may be intrinsically present in an application when not all tasks are on the critical paths that limit the application throughput. Dynamic slack is present when the actual case execution time of a task iteration is shorter than its worst case execution time.

In this context, many approaches aim to reduce energy consumption without affecting the application guarantees. A way to save energy is to conservatively lower the processor operating points, e.g., by dynamic voltage-frequency scaling (DVFS) for each application task [4], [5], [3], [6]. However, existing methods cannot observe the entire static and dynamic slack that is present in an application at run-time, leading to energy waste.

In this paper, we address the problem of accurate slack observation in real-time, data-flow applications mapped on MPSoCs. We propose a framework for conservative slack accounting and distribution between tiles. We perform slack accounting with the help of timestamps. Furthermore, the main features of our proposal are, as follows: (1) slack is distributed between tasks, potentially mapped to different tiles, along with the existing inter-task synchronization, (2) slack can be distributed in two directions, i.e., from producer tasks to consumer tasks and vice versa, and (3) static and dynamic slack is addressed.

Our framework consists of a hardware coprocessor and a software library to provide support for the newly introduced coprocessor. Furthermore, we integrate the coprocessor and the software library into an existing MPSoC platform and we prototype the entire system on a Xilinx ML605 FPGA board. We experiment with a data-flow implementation of an H.264 decoder. We compare two variants of our technique

`/13 $26.00 © 2013 IEEE
DOI 10.1109/DSD.2013.15

to an existing intra-tile management technique [3]. The two variants differ in that the first associates slack with a task, as many existing methods, and the second associates it with an application, increasing the potential for DVFS. In each one of the aforementioned cases, we employ a simple, greedy energy management policy that, conservatively utilizes as much slack as possible. Our experiments suggest that the total energy consumption reductions for each one of our two techniques are 23.2% and 26.7% when compared to the existing technique [3]. The software overhead in terms of extra clock cycles varies from 1% to 4% of the application execution time. The hardware overhead is negligible, 0.002% chip utilization from the considered FPGA board (xc6vlx240t).

The remainder of the paper is organized as follows. In Section II, we compare our proposal with the related state-of-the-art. In Section III, we introduce the application and platform models. In Section IV, we present the concepts behind our solution. In Section V, we provide the implementation details. In Section VI, we present an experimental proof of concept of the proposed solution. The paper concludes with Section VII.

II. Related Work

In this section, we discuss approaches in (1) slack management, distribution, and power-aware scheduling for data-flow applications, and (2) timestamps.

Nelson et al. [7] propose to reduce the energy consumption using static slack, when an application is mapped on multiple tiles. The analysis employs a custom design-time tool calculating the Maximum Cycle Mean (MCM) [1] (intuitively, the length of the critical path) of the application graph. The tool finds lower clock frequencies for the tasks that do not belong to the MCM. One of the limitations of this approach is that compile-time tools may not observe all available slack. Furthermore, other compile-time (static) tasks mapping and DVFS schemes for MPSoCs based on the application data-flow graph exist [5], [6], [8]. Compared to these approaches, we propose a run-time technique that has the potential to improve on design-time solutions.

Other approaches [3], [9] propose run-time methods to observe the dynamic slack locally in each MPSoC tile. If a task finishes early, DVFS is conservatively performed for the next executed task of the application. The techniques accurately observe all dynamic slack within a tile, however early completion in one tile may also result in early start in other tiles. This slack cannot be observed here. Compared to these approaches, we distribute the slack information between the tiles which overcomes this limitation. Moreover, we address the static slack as well.

Carta et al. propose to minimize energy consumption in a pipelined MPSoC architecture [4] by using linear and non-linear feedback control schemes. The considered pipeline architecture resembles the execution of a streamming application. Furthermore, Zamora et al. [10] utilize stochastic automata networks for system-level performance/power analysis and trade-offs in designing of multimedia, streaming applications. However these approaches target the soft real-time domain, hence the throughput guarantees are not hard.

Several examples of systems that utilize timestamps to share timing information exists [11]. The first proposes a fully synchronous MPSoC. Timestamps are utilized to synchronize heterogeneous IP blocks which might be operating at various frequencies. In this way real-time guarantees are offered to applications. We use the timestamps in a different context, to compute the slack information that is sent from one tile to another. The approach in [11] assigns timestamps to the arrival of external events in a hard real-time system. This information is used to schedule ready tasks. Similarly, we register the arrival time of the synchronization information. However, unlike existing work, we utilize this information for slack calculation to enable energy management.

In summary, to the best of our knowledge, our proposal to distribute information among tiles together with synchronization, for the purpose of enabling accurate, conservative slack observation in data-flow applications is novel. Moreover, it can augment existing state-of-the-art policies to further save energy.

III. Prerequisites

This section introduces the application and the platform models that are useful to understand the rest of this paper.

A. Application model

A data-flow application, A, consists of a set of tasks that communicate via tokens through first-in-first-out (FIFO) channels of bounded-size. Task execute indefinitely, iteratively, processing tokens from input FIFOs and producing tokens into output FIFOs. As we target the real-time domain, we consider that each task T_a has a worst case execution time, $wcet_a$, known design-time. Each task iteration i, also denoted as $T_{a,i}$, has an actual execution time, $acet_{a,i}$, unknown at design-time. If a task T_a produces tokens that a task T_b consumes, T_a is denoted as the predecessor of T_b and T_b the successor of T_a. A tasks is eligible for execution, or ready, if it has sufficient tokens in the input FIFOs and space in the output FIFOs. The eligibility of a task T_a is given by its state, referred to as $state_a$.

The maximum throughput of an application is given by the Maximum Cycle Mean (MCM) [1], [2] Intuitively, this is the length of the critical path in the task graph. Static throughput analysis utilizes the worst-case execution time of tasks. For simplicity, hereafter the explanations consider single-rate data-flow applications.

In Figure 1.a, we consider a data-flow application with two tasks, a producer and a consumer. In the example

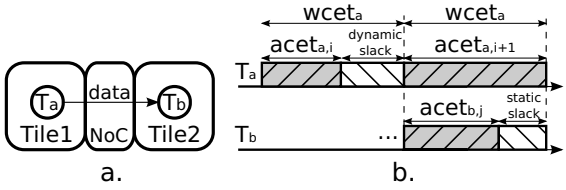


Figure 1. Producer-consumer example: a) considered application; b) static and dynamic slack;

two basic types of slack are distinguished upon: static and dynamic, as graphically presented in Figure 1.b. Intuitively, the static slack occurs because not all task are on the critical path. Static slack is also denoted as the maximum deadline extension in the literature [1], [12]. When the two tasks are executed on different tiles, their iterations may overlap. Consider that $wcet_a > wcet_b$ and there is enough space in the FIFO. We can delay the finish of iteration j of T_b until the finish of iteration $i+1$ of T_a and the throughput of the application will remain the same. Or in other words, in this example T_b has $wcet_a - wcet_b$ static slack. Dynamic slack occurs when the actual case execution time of a task iteration is shorter than its worst case execution time, e.g., $wcet_a - acet_{a,i}$ for T_a in the example in Figure 1.

B. Platform model

The targeted MPSoC consists of a number of processor tiles, hereafter shortly denoted as tiles, communicating via a network-on-chip (NoC). A tile typically comprises of a processor core, a set of memory blocks, and Direct Memory Access (DMA) modules. As we address real-time systems, we assume that the NoC offers guaranteed service, e.g., maximum throughput and minimum latency. We assume that each tile can be scaled independently, i.e., the tiles and the NoC are each in its own clock domain, and a time translation between different clock domains is possible (the clocks are mesochronous or there is a slower, reference clock).

Application tasks may be mapped to different tiles. One tile may be shared between several applications. An RTOS executes on each processor core. Processor scheduling is at two levels, as follows. At the first level, the RTOS allocates fixed time quanta denoted as slots to each application, in a time-division multiplexing (TDM) fashion. Hence the inter-application level scheduler is preemptive. As applications are completely isolated, an application perceives its time as continuous, although in practice it may be preempted. We can consider that each application has a virtual-time consisting of the set of slots allocated to it. By knowing the TDM allocation, one can translate the application virtual-time in the physical-time of the tile and vice versa.

The two time-line translation functions are as follows:

$$\begin{aligned} time_{phy} &= f_{v\_p}(time_{vir}, A, tile), \\ time_{vir} &= f_{p\_v}(time_{phy}, A, tile). \end{aligned} \tag{1}$$

where $time_{phy}$ is the number of cycles in the tile-physical time, $time_{vir}$ is the number of cycles in task-virtual time, (both measured at the maximum MPSoC frequency level), A and *tile* are the considered application and tile, respectively.

At the second level, task are scheduled within an application. Typically, for data-flow applications the intra-application scheduler is non-preemptive, in the sense that the task scheduler is called only after a task iteration has finished. Tasks are scheduled only if they are eligible. Hence, once a task iteration starts it is guaranteed that it finishes without blocking. This means that idle time can occur only between iterations.

Inter-task communication is implemented by memory mapped software FIFOs employing the C-HEAP protocol [13], where each FIFO stores a limited number of data elements (tokens). The number of the available tokens is determined by the values of the read counter (*rc*) and write counter (*wc*) of the FIFO. For each FIFO, the consumer and the producer side are responsible for updating the write counter and the read counter, respectively.

When two communicating tasks are mapped on different tiles, for each token, the NoC travel time is bounded by a worst-case traveling time *wctt*. This time depends on, e.g., the parameters of the NoC, the token size, and it is known at design-time, for each FIFO. As the focus of this paper is tile slack, to simplify the notation, in the rest of this paper we will use *wctt* to denote the worst-case traveling time in general. This notation can be detailed per FIFO and per connection.

IV. Proposed Solution

As mentioned, we distinguish two types of slack, static and dynamic. Although complex, the problem of accurately computing the static slack and saving energy by allocating static frequencies to tasks can be solved at design-time [6]. Nevertheless, it is desirable to have run-time support for the static slack calculation because of several reasons including: (1) the design-time tools may not always observe all static slack and (2) in practice the processor operating points are discrete, hence static scaling may leave a fraction of the slack un-utilized; this slack may accumulate during the application execution and could be used for further scaling and energy saving, if it could be observed. In case of the dynamic slack, early completion of a task may cause an early start other tasks. However current methods do not observe that early start as slack if tasks are mapped on different tiles, hence potential for energy saving is lost. These facts motivate us to address the problem of accurate slack observation.

In the rest of the section, we first outline the concept of our framework for inter-tile slack distribution and second we introduce the equations behind.

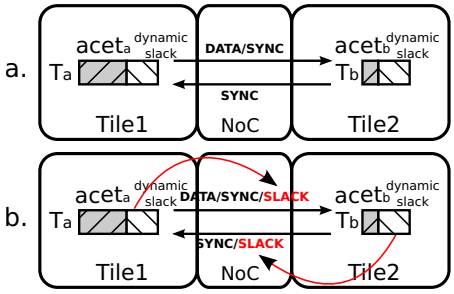


Figure 2. A conceptual model for slack information distribution by: a) Intra-tile technique [3]; b) Our inter-tile technique with dynamic slack;

A. Conceptual solution

In Figure 2, we compare the conceptual models for slack information distribution employing intra-tile technique [3] and our inter-tile technique. We consider the application from Figure 1. In FIFOs, data is transferred from T_a to T_b, while the synchronization information is transferred in both directions. In Figure 2.a, we present a case in when the slack is computed locally, within the tile, e.g., [3]. In Figure 2.b, we present a case when synchronization is augmented with slack information for each one of the communicating tasks. As a result, we can distribute slack in both directions - from producer to consumer task and vice versa.

The distributed slack among the tiles can be static and dynamic. We model static slack as a time reference until which a task has to finish its next iteration. Note, this reference is not the static slack, but rather the information required to calculate it. Furthermore, we model dynamic slack as the duration of time that the current iteration has ran ahead. At run-time, we transfer these two values. If a static slack reference is transferred, the calculation of this type of slack has to be finalized at the recipient tile.

Two ways to represent the slack are possible – either a relative or an absolute value. We choose a relative value representation because we target distributed systems where the global notion of time is often missing. Furthermore, a relative time value can be measured in different time-domains. In the first one, the relative time is measured in the time-domain of the application (application virtual-time). In the second one, the relative value is measured in the time-domain of the tile, taking into account the RTOS overheads and the time when other applications might be running (tile physical-time). Therefore, slack measured in application virtual time-domain in one tile and transferred to another tile can be interpreted wrongly. To avoid this, we transform the slack from the virtual to physical time-domain.

We extend the slack classification, namely, we introduce two new types of slack, *tile* (S_{tile}) and *remote* (S_{remote}), depending on the location of the slack in the system from the perspective of the tile. The S_{tile} is shared among all tasks of an application in a tile. Once it is distributed to another tile, we refer to it as remote on the tile that receives it. Eventually, in the remote tile, S_{tile} is updated with the received S_{remote}. The S_{tile} and S_{remote} contain static and/or dynamic slack. If multiple applications are running on the tile, then each application will have its own S_{tile} and S_{remote}. Slack values are updated at each task iteration start and finish; nevertheless, for brevity and readability we omit the indexes of each iteration.

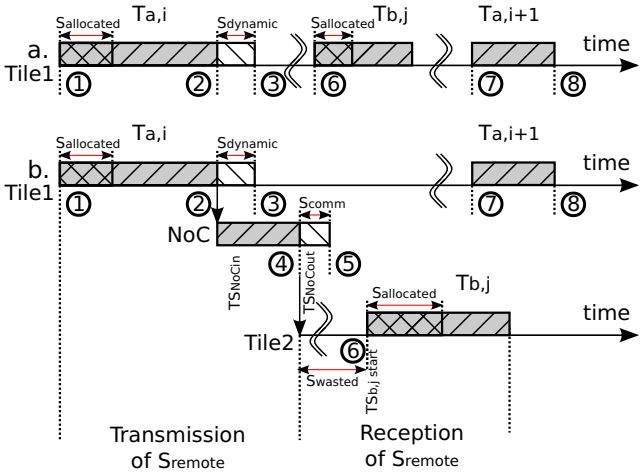


Figure 3. Slack computation, allocation, and distribution for: a) intra-tile task communication; b) inter-tile task communication;

B. Inter-tile slack distribution

For simplicity, we explain our framework by means of the two examples presented in Figure 3. Figure 3 illustrates the moments that are relevant to slack computation, allocation, and distribution during two consecutive task iterations, $T_{a,i}$ and $T_{a,i+1}$. The application is the same as in Figure 1. In Figure 3.a, we present a case when both tasks are mapped on the same tile, while in Figure 3.b, each task is mapped on a different tile. Next, we detail the transmission and reception of S_{remote}.

Transmission of S_{remote}:

The first event that we consider (Figure 3.a and Figure 3.b), is the start of $T_{a,i}$, at instance ①. Note that at this point slack may already exist on the tile and a management policy may have allocated a part of it to $T_{a,i}$. We refer to the allocated slack as $S_{allocated}$.

Second, the $T_{a,i}$ finishes earlier than at worst case, at instance ②. Therefore, the dynamic slack for $T_{a,i}$, in application virtual-time, is computed as:

$$S_{dynamic_{vir}} = wcet_a - acet_{a,i} \quad (2)$$

As a result, at instance ②, the tile slack is updated with the newly computed dynamic slack:

$$S_{tile_{vir}} = S_{tile_{vir}} + S_{dynamic_{vir}} \quad (3)$$

In case that the tasks are mapped on the same tile (Figure 3.a) the execution continues, the consumer task $T_{b,j}$ starts (at instance ⑥) and finishes. Next iteration of the producer task $T_{a,i+1}$ starts and finishes at instances ⑦ and ⑧, respectively. For intra-tile communication, application tasks share the S_{tile} based on dynamic slack only, while remote slack is equal to zero. For example, in Figure 3.a, the $T_{b,j}$ allocates slack based on the dynamic slack of $T_{a,i}$.

In case that tasks are mapped on different tiles, in Figure 3.b, at instance ②, we compute the S_{static} as follows:

$$S_{static} = stdelay_{a,i+1} + \begin{cases} wcet_a & \text{if } state_a = Ready, \\ 0 & \text{otherwise.} \end{cases} \quad (4)$$

where $stdelay_{a,i+1}$ is the starting delay, i.e., the duration between the current moment and the earliest starting time of the next iteration, *i+1*, of T_a. A starting delay may exists, because, in general, the processor may be shared among multiple tasks of the application, hence several other tasks may be scheduled before $T_{a,i+1}$. S_{static} is the latest moment in time (relative to instance ②) until when the predecessors and successors of T_a have to finish their next iterations in order to respect the throughput constraint. If the starting delay cannot be calculated (for example, because the scheduling policy is dynamic) it is conservatively set to zero. If the next candidate for execution is the currently finishing task, then the starting delay is also zero. In case iteration *i+1* of T_a is already eligible for execution ($state_a = Ready$) at the time when the current iteration completes, then if the execution of the $T_{b,j}$ finishes by $wcet_a$ the application throughput is met. If T_b is on the critical path, then the transferred $wcet_a$ to T_b results in zero cycles slack (see Equation 11).

Furthermore, at instance ②, we compute the remote slack as the sum of application slack and static slack, as follows:

$$S_{remote_{vir}} = S_{tile_{vir}} + S_{static_{vir}} \quad (5)$$

As introduced in Section IV-A, we distinguish two types of remote slack:

$$S^{type}_{remote} = \begin{cases} Dyn & \text{if } S_{static_{vir}} = 0, \\ StaDyn & \text{otherwise.} \end{cases} \quad (6)$$

where *Dyn* models that remote slack includes only dynamic slack and *StaDyn* models that remote slack includes static and dynamic.

As a last step, we translate the remote slack value from task-virtual to tile-physical time-domain:

$$S_{remote_{phy}} = < f_{v\_p}(S_{remote_{vir}}, A, tile), S^{type}_{remote} >, \quad (7)$$

where *A* and *tile* are the target application and tile, respectively. Note that the $S_{remote_{phy}}$ also contains S^{type}_{remote}.

Reception of S_{remote}:

After the remote slack is calculated, it is sent to the predecessors and successors of T_a, along with the synchronization information, over the NoC. Because at run-time the actual NoC latency might be smaller that the worst-case, the NoC can generate a type of dynamic slack, which we refer as *communication slack* (S_{comm}). For example, at instance ④, the data arrives to the destination tile earlier that the worst-case which is illustrated by ⑤. We compute the communication slack as follows:

$$S_{comm_{phy}} = wctt - (t_{NoCout} - t_{NoCin}), \quad (8)$$

where t_{NoCout} is the timestamp when $S_{remote_{phy}}$ arrives on the destination tile. The t_{NoCin} is the timestamp when $S_{remote_{phy}}$ initially enters the NoC. t_{NoCout} and t_{NoCin} are measured in the NoC time-domain.

After the data has arrived, the consumer task ($T_{b,j}$) is started, at instance ⑥. We define the time between instances ④ and ⑥ as wasted slack, as follows:

$$S_{wasted_{phy}} = st_{b,j} - t_{NoCout}, \quad (9)$$

where $st_{b,j}$ is the tile physical starting time of iteration j of T_b. For applications with more tasks running on a tile, we may employ the starting time of any of the application tasks (st_A) instead of $st_{b,j}$, if we associate the slack information with application and not with a task. In Section VI, we conduct experimental study for the two cases, i.e., remote slack associated with a task and with an application, that trade accuracy of the computed slack for computation overhead. If the NoC does not share a common source for clock frequency with the tiles, then t_{NoCout} in Equation 9 should be transferred from the NoC physical time-domain to the tile physical time-domain.

In Figure 3.b, at instance ⑥, we calculate, what has left from the remote and communication slacks after the wasted slack is subtracted, i.e., X, as follows:

$$X = S_{remote_{phy}} + S_{comm_{phy}} - S_{wasted_{phy}},$$
$$Y = \begin{cases} X & \text{if } X > 0, \\ 0 & \text{otherwise.} \end{cases} \quad (10)$$

Furthermore, based on the S^{type}_{remote} value, the received remote slack is calculated as follows:

$$Z = \begin{cases} f_{p\_v}(Y - 2 * wctt, A, tile) - wcet_b & \text{if } S^{type}_{remote} = StaDyn, \\ f_{p\_v}(Y, A, tile) & \text{otherwise.} \end{cases}$$

$$S_{remote_{vir}} = \begin{cases} Z & \text{if } Z > 0, \\ 0 & \text{otherwise.} \end{cases} \quad (11)$$

If the S^{type}_{remote} is *StaDyn*, then remote slack contains a non-zero reference for static slack calculation. This means that, to be conservative, the current task iteration should finish until this reference in time. At worst case, finishing would take the communication overhead and $wcet_b$ for the current task iteration. The communication overhead equals to *2*wctt*, because we need to count the overhead of two synchronizations, the

one after $T_{a,i}$ finished and the one after $T_{b,j}$ finishes. Once we obtain the remote slack, we perform the time-domain translation. If the remote slack is equal to zero, it means that all of it is lost.

Next, the available slack for $T_{b,j}$ ($S_{task_{vir}}$) is computed as a max of the remote slack and tile slack:

$$S_{task_{vir}} = \max_{tid=1}^{N\ tasks}(S^{\ tid}_{remote_{vir}}, S_{tile_{vir}}), \tag{12}$$

where $S^{\ tid}_{remote_{vir}}$ is the remote slack for each remote predecessor or successor task (*tid*). The $S_{tile_{vir}}$ is the slack generated by the tasks executed on the local tile. In Equation 12, we achieve accurate and conservative calculation of the slack, even when we apply max instead of min function. To explain such a choice, we consider a case, which multiple tasks are mapped on the same tile and and all of them receive remote slack values. Lets assume, we do not employ Equation 12 while we schedule the task with the highest value of remote slack. Therefore, even if the task iteration finishes with *wcet*, the remote slack of the task is distributed to the other tasks on the tile. As a result, max function in Equation 12 preserves the conservative calculation of the slack.

Existing work typically computes and allocates slack per task, and not per application. However, we differentiate two implementations of Equation 12, depending on the value of *N*, i.e., either the number of remote tasks for $T_{b,j}$ or the total number of remote tasks in the application. In Section VI we investigate the quantitative difference between these two approaches.

Finally, a slack policy (f_{slack}), may be applied to allocate slack for $T_{b,j}$, and potentially scale its operating point:

$$S_{allocated_{vir}} = f_{slack}(S_{task_{vir}}) \tag{13}$$

If not all slack is allocated, the remained tile slack before starting the scheduled task should be updated, as follows:

$$S_{tile_{vir}} = S_{tile_{vir}} - S_{allocated_{vir}} \tag{14}$$

Summarizing, we propose a run-time framework that: (1) accurately computes and distributes static and dynamic slack between tasks mapped on different tiles of an MPSoC, and (2) enables the utilization of any management policy that can allocate slack and scale the tile operating point to reduce energy consumption.

V. System Implementation

In this section we describe the new software and hardware components added to an existing RTOS and MPSoC.

The tiled CompSoC platform [14] is the template for our implementation. Each tile execute the CompOSe RTOS [15]. To implement slack distribution, we augment the CompSoC tile with a Custom Computing Unit (CCU) using Molen-style processor–coprocessor design [16], more specifically, the coprocessor microarchitecture from [17]. We refer to the software part of our framework as Remote Slack (Rem_Slack) library. Respectively, we call our CCU - Rem_Slack CCU.

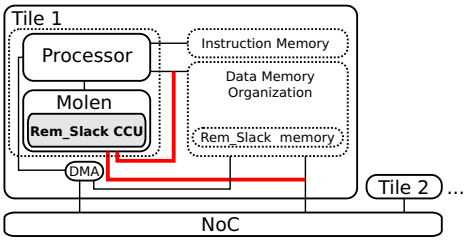


Figure 4. CompSoC processor tile augmented with Molen-style Rem_Slack CCU

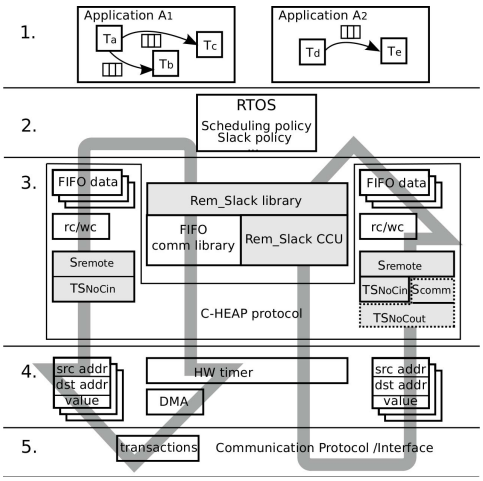


Figure 5. Rem_Slack library and Rem_Slack CCU integration to the CompOSe RTOS

In Figure 4, we present the architecture of a CompSoC tile augmented with Molen-style coprocessor. This CCU accesses the NoC and data memory buses. The Rem_Slack CCU receives the remote slack from the NoC bus and it registers the time of its arrival in t_{NoCin}. The Rem_Slack CCU stores the remote slack, the t_{NoCout}, and the communication slack for each FIFO in its internal memory, which is part of the data memory organization of the tile, hence it is accessible by the tile processor through the data memory bus.

In Figure 5, we present the integration of the the Rem_Slack library and the Rem_Slack CCU to the CompSoC platform. The shaded blocks represent our contribution to the existing platform. The two large arrows illustrate the transmission and reception of FIFO tokens. If a task consumes tokens from a FIFO, then it receives the data and the *wc*, and it is responsible with updating the *rc*. If a task produces tokens into a FIFO, then it receives the *rc* and it sends data, and updates the *wc*. We augment the synchronization information, namely *rc* and *wc*, with two extra fields: remote slack and t_{NoCin}. In this way, remote

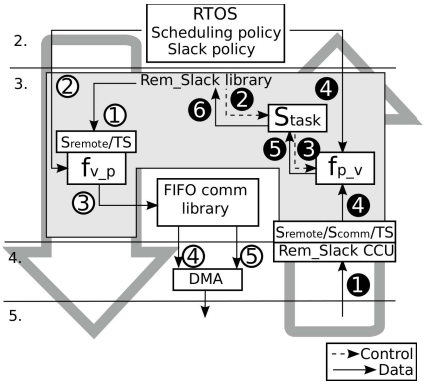


Figure 6. Rem_Slack library and Rem_Slack CCU integration to the CompOSe RTOS: detailed view

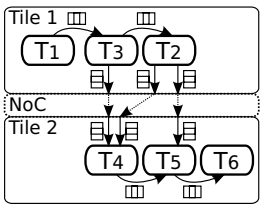
slack is distributed in both directions: from the producer task to the consumer task and vice versa.

Figure 6 details the sequence of steps involved in slack distribution. First, after a task iteration finishes the RTOS invokes the Rem_Slack library to update the $S_{remote_{vir}}$, as represented with ①. Internally, the Rem_Slack library calls the f_{v_p} function that uses the application scheduling information (at RTOS level) to calculate $S_{remote_{phy}}$. This is represented with ② in Figure 6. After that, at instance ③, the RTOS invokes the FIFO communication library to transfer the tokens (if the task is a producer), the synchronization, and the slack to remote tiles. Hence first, at instance ④, the FIFO communication library transfers, via the DMA, to the remote tile, either the DATA+*wc* (producer task), or *rc* (consumer task). At instance ⑤, $S_{remote_{vir}}$ and t_{NoCin} are transferred via the DMA.

On a remote tile, for a given FIFO, at instance ❶, synchronization and slack information is received. The Rem_Slack CCU stores the $S_{remote_{phy}}$, registers its arrival time, t_{NoCout}, and computes the S_{comm}. Later, whenever a task finishes, a new one should be scheduled, for example at instance ❷. At this moment, independently whether instance ❶ occurred, the RTOS invokes the Rem_Slack library to calculate the S_{task} by employing Equation 12 At instance ❸, the Rem_Slack library invokes f_{p_v}. Internally, as labeled with ❹, the f_{p_v} employs the received remote slack and the scheduling policy information to translate the remote slack from the tile physical-time to the application virtual-time. At instances ❺ and ❻, the translated remote slack value is returned back to the Rem_Slack library and the RTOS, respectively. The instances from ❸ to ❺ are repeated *N*-times, as in Equation 12.

VI. Experimental Results

In this section, we first briefly describe the platform and the tools. Then, we present the target application, namely H264, and the experimental setup. In the end of this section



Tile 1	wcet	Tile 2	wcet
T1	114 776	T4	174 067
T2	577 522	T5	232 294
T3	152 698	T6	20 602

Figure 7. H.264 tasks: mapped on CompSoC processor and wcet (clock cycles)

we present the observed clock frequencies of the tiles, consumed energy and overheads in software and hardware.

For our experiments, we employ a dual-tile CompSoC platform. Each tile embeds a Xilinx Microblaze core. The design is synthesized with Xilinx Platform Studio 12.3 and verified on Xilinx Virtex ML605 (xc6vlx240t) evaluation board. Moreover, we consider that each processor can operate in 15 equidistant frequency levels, varying from 50MHz downto 3.1MHz. We consider the same energy model as [7].

We exercise a streaming data-flow H.264/AVC decoder [18], to evaluate our slack distribution technique. In Figure 7, we present the H.264 tasks to tiles mapping and we list the worst-case execution times of tasks. The H.264 application has static slack, due to the difference in the *wcet* of the tasks, and dynamic slack, caused by the variations of the *acet*s of tasks.

We compare our inter-tile slack distribution with intra-tile slack management (Intra-tile slack mngm) [3]. Furthermore, we investigate two variants of our proposal, depending on the computation of S_{task}, i.e., the value of N in Equation 12. In the first case, the remote slack is the maximum value among the remote slack values from the FIFOs linked/connected with the task that is schedule to start next (Inter-tile slack mngm (MAXtask)). In the second case, the remote slack is the maximum value among the remote slack values from the FIFOs of all tasks mapped on the tile (Inter-tile slack mngm (MAXtile)). In both cases, we utilize a simple, greedy slack policy that always allocates all the available slack to the next scheduled task.

In Figure 8 and Figure 9, we present the clock frequency of Tile 1 and Tile 2 for 150 task iterations. In case when no slack management is applied the core operates all the time at the maximum frequency (f_{max}). For Tile 1, the proposed inter-tile slack technique does not always achieve lower frequencies than the intra-tile technique. Compared to the existing intra-tile technique [3], the total frequency level reduction of the tasks for MAXtask is 5.7% and for MAXtile is −0.3%, respectively. There are two main reasons for it: (1) our slack policy employs more slack in a given task iteration and there is not enough slack for the policy to scale down the frequency in the following task iterations; (2) the transferred remote slack value is lower than the Rem_Slack library overhead, therefore, Rem_Slack library introduces

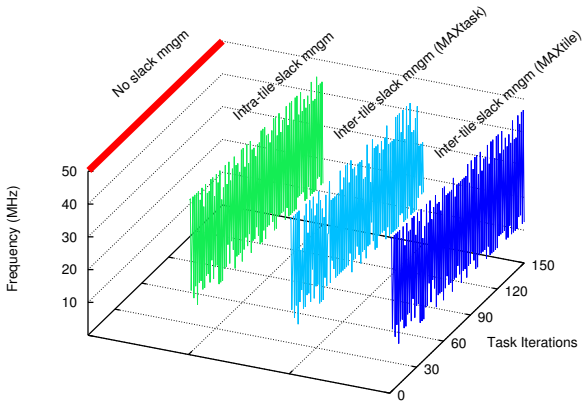


Figure 8. Frequency levels for the H.264 tasks running in Tile 1

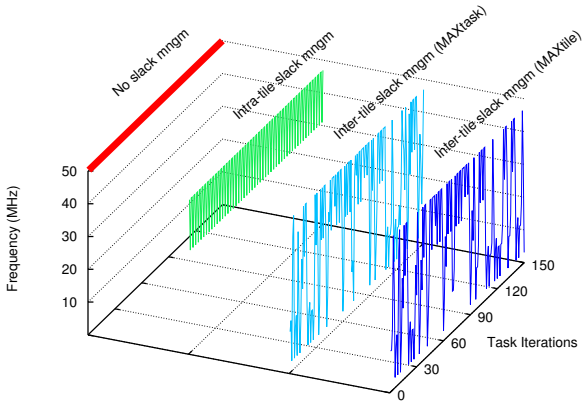


Figure 9. Frequency levels for the H.264 tasks running in Tile 2

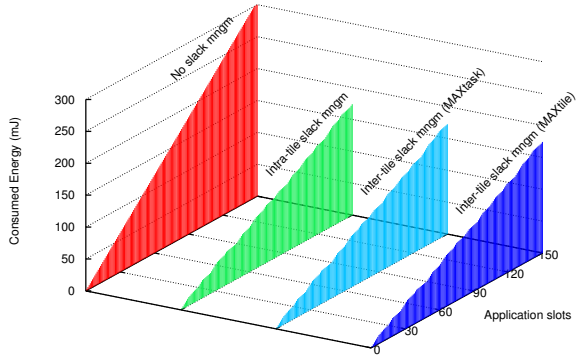


Figure 10. Consumed Energy for the H.264 tasks running in Tile 1

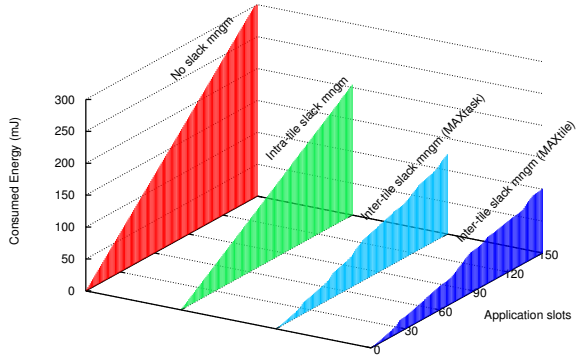


Figure 11. Consumed Energy for the H.264 tasks running in Tile 2

slow-down, instead of speed-up in the application.

For Tile 2, as the results suggest, the MAXtile case finds lower clock frequency than MAXtask. This difference is because of the amount of the S_{wasted} and the task that employs S_{remote}. For example, if the S_{wasted} is equal to zero, then the ratio between the $acet_{T_6}$ and the received remote slack is more than 15 times. In such a way, the task clock frequency can be set to the lowest possible. Compared to [3], the total frequency level reduction for MAXtask is 48.3% and for MAXtile is 56.8%.

In Figure 10 and Figure 11 we present the consumed energy for each one of the previously introduced cases, for Tile 1 and Tile 2, respectively. We run the system for 150 application slots. In Figure 10, the consumed energy for our proposals are almost equal to the intra-tile technique, as it was suggested by the clock frequency of Tile 1 in Figure 8. The total reduction of the consumed energy for MAXtask and MAXtile, compared to [3] is equal to −0.03% and 0.17%, respectively. In Figure 11, we observe that the consumed energy of each of MAXtask and MAXtile is lower than the intra-tile slack management, as also suggested by the frequency levels in Tile 2. During the first few application slots, the consumed energy is equal for the two implementations of our proposal. The reason is that it takes time for the inter-tile communication to be established. In Tile 2, compared to the intra-tile technique, MAXtask and MAXtile reduce energy consumption with 46.5% and 53.3%, respectively. Based on the measured energy reduction, we computed the total reduction over both tiles.

In what follows, we present the software and hardware overhead of the Rem_Slack library and Rem_Slack CCU. The software overhead is in terms of extra clock cycles. The hardware overhead is in terms of chip utilization, i.e., the number of occupied FPGA slices. Depending on the number of remote FIFOs, the software overhead of the REM_Slack library is as follows: for the transmitting of the remote slack varies from 1.5k up to 3.2k clock cycles. As illustrated in Figure 7, the number of remote FIFOs varies between one and two. The software overhead of the slack policy varies from 2.0k up to 2.5k clock cycles. Since, the Rem_Slack library is invoked at the beginning and at the end of a

task iteration, the introduced overhead varies between 1% and 4% compared to the execution time of the H.264 task iterations directly involved in the remote slack transmission and reception.

The chip utilization of the Molen wrapper with a dedicated memory bank of 64 bytes is 585 slices. The overhead of the Rem_Slack CCU is 84 slices. As we expected the chip utilization of the Rem_Slack CCU is negligible, less than 0.002% of the total number of slices in the considered FPGA chip. Furthermore, the hardware overhead of the Rem_Slack CCU does not dependent on the number of applications, tasks, and FIFOs. The only resource which scales with the number of remote FIFOs is the data memory that stores remote slack and timestamps. Although, the Rem_Slack CCU runs continuously, due to its small footprint, we expected its energy consumption to be low as well.

VII. Conclusions

In this paper, we proposed a framework for slack computation, allocation, and distribution that transfers the static and dynamic slack information among the tiles in an MPSoC, executing dataflow applications. We send slack information altogether with the existing inter-task synchronization. Moreover, we transferred the slack in both directions - from the producer to the consumer task and vice versa. We experimented with H.264 decoder and we studied four scenarios: a no-slack management, an inter-tile slack management, and two variants of our inter-tile slack management. As evaluation criteria of our technique, we employed the clock frequency, the consumed energy, and the introduced overhead in software and hardware. The results suggested that our inter-tile technique reduces the total energy consumption of 27% at the cost of minor software overhead of up to 4% and negligible additional FPGA chip utilization of 0.002%.

Acknoledgements

This work was partially funded by projects EU FP7 288248 Flextiles, Catrene CA104 Cobra and CA505 BENEFIC.

References

[1] S. Sriram and S. S. Bhattacharyya, *Embedded Multiprocessors: Scheduling and Synchronization*, 1st ed. New York, USA: Marcel Dekker, Inc., 2000.

[2] A. Dasdan, "Experimental analysis of the fastest optimum cycle ratio and mean algorithms," *ACM TODAES*, vol. 9, no. 4, pp. 385–418, 2004.

[3] A. Molnos and K. Goossens, "Conservative dynamic energy management for real-time dataflow applications mapped on multiple processors," in *Proc. of DSD*, 2009, pp. 409–418.

[4] S. Carta, A. Alimonda, A. Pisano, A. Acquaviva, and L. Benini, "A control theoretic approach to energy-efficient pipelined computation in MPSoCs," *ACM Trans. Embedded Comput. Syst*, vol. 6, no. 4, 2007.

[5] Y. Wang, H. Liu, D. Liu, Z. Qin, Z. Shao, and E. H.-M. Sha, "Overhead-aware energy optimization for real-time streaming applications on multiprocessor System-on-Chip," *ACM TODAES*, vol. 16, no. 2, p. 14, 2011.

[6] P. Huang, O. Moreira, K. Goossens, and A. Molnos, "Throughput-constrained voltage and frequency scaling for real-time heterogeneous multiprocessors," in *Proc. of SAC*. ACM, 2013, pp. 1517–1524.

[7] A. Nelson, O. Moreira, A. Molnos, S. Stuijk, B. T. Nguyen, and K. Goossens, "Power minimisation for real-time dataflow applications," in *Proc. of DSD*, 2011, pp. 117–124.

[8] M. Ruggiero, A. Acquaviva, D. Bertozzi, and L. Benini, "Application-specific power-aware workload allocation for voltage scalable MPSoC platforms," in *Proc. of ICCD*, 2005, pp. 87–93.

[9] R. Jejurikar and R. Gupta, "Dynamic slack reclamation with procrastination scheduling in real-time embedded systems," in *Proc. of DAC*, 2005, pp. 111–116.

[10] N. H. Zamora, X. Hu, and R. Marculescu, "System-level performance/power analysis for platform-based design of multimedia applications," *ACM TODAES*, vol. 12, no. 1, pp. 2:1–2:29, 2007.

[11] D. N. Bui, H. D. Patel, and E. A. Lee, "Deploying hard real-time control software on chip-multiprocessors," in *Proc. of RTSCA*, 2010, pp. 283–292.

[12] O. Moreira, T. Basten, M. Geilen, and S. Stuijk, "Buffer sizing for rate-optimal single-rate data-flow scheduling revisited," *IEEE Transactions on Computers*, vol. 59, no. 2, pp. 188–201, 2010.

[13] A. Nieuwland, J. Kang, O. P. Gangwal, R. Sethuraman, N. Busá, K. Goossens, R. P. Llopis, and P. Lippens, "C-HEAP: A heterogeneous multi-processor architecture template and scalable and flexible protocol for the design of embedded signal processing systems," *Design Automation for Embedded Systems*, vol. 7, no. 3, pp. 233–270, 2002.

[14] A. Hansson, K. Goossens, M. Bekooij, and J. Huisken, "CoMPSoC: A template for composable and predictable multi-processor system on chips," *ACM TODAES*, vol. 14, no. 1, 2009.

[15] A. Molnos, A. Beyranvand Nejad, B. T. Nguyen, S. Cotofana, and K. Goossens, "Decoupled inter- and intra-application scheduling for composable and robust embedded MPSoC platforms," in *Proc. of SCOPES*. ACM, 2012, pp. 13–21.

[16] S. Vassiliadis, S. Wong, G. N. Gaydadjiev, K. Bertels, G. K. Kuzmanov, and E. M. Panainte, "The Molen polymorphic processor," *IEEE Transactions on Computers*, vol. 53, pp. 1363–1375, 2004.

[17] P. G. Zaykov and G. Kuzmanov, "Architectural support for multithreading on reconfigurable hardware," in *ARC*, 2011, pp. 363–374.

[18] B. Nguyen, "Task scheduling methods for composable and predictable MPSoCs," Master's thesis, TUD, October 2010.