Composable Power Management with Energy and Power Budgets per Application

Andrew Nelson
Delft University of Technology
Delft, The Netherlands
a.t.nelson@tudelft.nl

Anca Molnos
Delft University of Technology
Delft, The Netherlands
a.m.molnos@tudelft.nl

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

***Abstract*—Embedded Multiprocessor Systems-on-Chip (MPSoCs) commonly run multiple applications at once. These applications may have different time criticalities, i.e. non real-time, soft real-time, and firm or hard real-time. Application-level composability is used to provide each application with its own virtual platform, such that each application may be developed, verified, and executed independently, given its virtual platform specification. Composability of functional and temporal properties has been demonstrated in previous work.**

In this paper, we extend composability to include power management, where *each application can manage its energy usage independently*. Each application receives an independent energy and/or power budget, which it can manage as it sees fit, with its own application-specific power-management policy. *Time, energy, and power budgets allocated to each application* ensure that its power-management policy cannot cause any interference to the functional, timing, and power behaviours of other applications.

We implement our technique on an existing composable and predictable hardware platform (CompSoC), and extend its Real-Time Operating System (OS) with a power-management infrastructure. Applications use a power-management API to communicate with the OS that implements time, energy, and power budgets. We demonstrate the applicability of our techniques by running several concurrent applications with their own power managers on an FPGA prototype.

I. Introduction

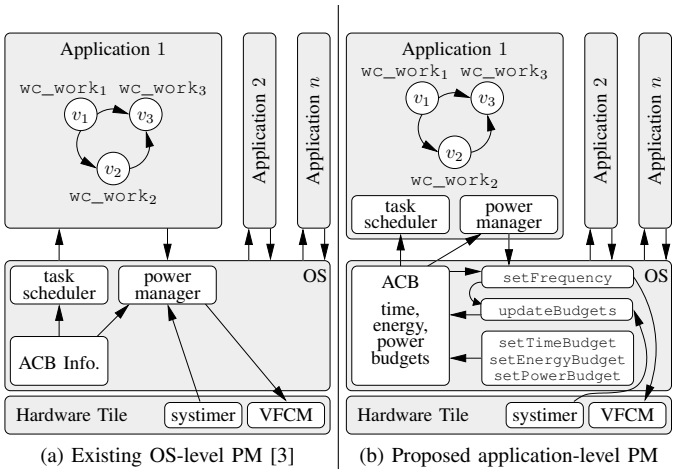
Multiprocessor Systems-on-Chip (MPSoCs) may simultaneously run multiple applications with both real-time and non-real-time requirements. Real-time applications must meet their requirements regardless of other applications that are executing on the same platform. Integration of multiple applications with mixed-time criticality requirements pose a problem as they interfere with each others functional and temporal properties on shared resources. A priori verification of desired properties of such systems is complex since all applications, and the degree and nature of their interference must be known. Independent Development and verification of (real-time) applications is impossible in this case, since detailed knowledge of other (e.g. non real-time, possibly unbounded) applications' resource usage would need to be known. One solution is platform virtualisation through *application-level composability*, where platform resources are shared such that applications execute on their own virtual platform. This enables application-level temporal requirements to be met for the virtual platform, regardless of other applications executing on the MPSoC. This has been shown for functional and temporal properties.

While much work has been done on the composability of applications' functional and temporal properties [1]–[3], power-management policies for such systems are generally made for the entire system, and not per application. In [4] MPSoCs are cited as one of the justifications for virtualisation, but lists energy management as one of its limitations, claiming that power-management is inherently a system-level property, rather than an application-level property.

In this paper we disprove this hypothesis, as we extend the application-level composability approach to include *application-level composable power-management*, with *energy and power budgets per application*. In real terms this adds the benefit of treating individual applications, e.g. on a mobile phone, as if they have their own independent energy source. Even though all the applications share a single battery, or other power source, they maintain completely separate energy and power budgets. This is useful from a user point of view, as illustrated by two common scenarios on a smart phone. *An energy budget* can be reserved for the phone-call application, such that there is guaranteed to be sufficient battery energy to be able to make a five-minute phone call, even when other applications are running, such as (battery-hungry) gaming or wireless video streaming. Similarly, each application can be given a *power budget*, to avoid overheating or drawing more current than is optimal for the battery, or for energy-scavenging systems where energy is unlimited, but power is not.

The major advantage of composable virtual platforms, with independent power budgets, is that designers are able to design their application-specific power-management policy using their knowledge of their (non) real-time application, without affecting the power or temporal profile of other concurrent applications. This enables application designers to optimise their application's power profile, using their virtual platform's specification, independently of the development of other applications for the same platform. When the applications are executed concurrently on the platform, their composition will have no affect on the temporal and power profiles of the other applications, in comparison to executing independently, allowing their independent design and verification.

In this paper we describe how this may be achieved for a predictable and composable hardware platform [5], [6] and Real-Time Operating System (OS) [7] through use of an Application Programming Interface (API), that enables the creation of application-level power-management policies. We provide some example power-management policies that exploit temporal and energy budgeting information, to achieve power reduction through Dynamic Voltage and Frequency



(a) Existing OS-level PM [3] (b) Proposed application-level PM

Figure 1. Power-management (PM) hierarchy

Scaling (DVFS). In order for power-management to be truly composable each application must have its own energy or power budget, so that its power consumption behaviour does not affect other applications. Subsequently each application executes in its own virtual platform and manages its own energy or power budget independently. Meanwhile the OS allocates and enforces time, energy, and power budgets such that concurrent applications do not affect each other.

We describe, in this paper, how independent budgets may be assigned to individual virtual platforms in terms of time in seconds, and either energy in Joules, or power in Watts. These budgets must be enforced while maintaining the application's temporal requirements, which can be best effort, soft real-time, or firm real-time. We explain how or application-level power-management approach is extended to support any mix of best-effort, soft and firm real-time applications.

The rest of this paper is structured as follows. We present related work in the following section. In Section III we describe the background information necessary for Section IV, in which we explain how application-level composable power-management may be achieved. We provide experimental analysis using an FPGA prototype of our system in Section V. We conclude this paper in Section VI.

II. Related Work

Virtualisation of hardware resources for embedded systems has been documented multiple times before. In [8] for instance, it is described how the Xen hypervisor [9] can be extended to achieve a real-time control loop, using virtualisation to achieve isolation between time criticality domains. In [10], [11] it is explained how "temporal isolation" may be achieved through the use of Variable Bandwidth Servers (VBS) [12]. Their technique enables multiple applications to execute simultaneously, providing predictable bounds on latency and throughput, for application execution. The VBS technique has a more liberal interpretation of temporal isolation than what we apply in this paper with our composable approach. In the VBS technique applications may interfere with each others schedules, so long as the overall resource utilisation stays below 100%. In our composable approach, applications are completely temporally isolated, in that they do not affect each others execution by even a single cycle.

In [4] the role of virtualisation in embedded systems is examined. MPSoCs are identified as one of the target use-cases of virtualisation, but energy management is identified as one of the limitations of this approach. One of the main arguments in [4] is that energy, for embedded systems, is a global physical resource, and cannot therefore be traded off against performance.

While, e.g. mobile phones, have a global physical energy resource in the form of a battery, the energy stored in the battery is only a limitation between replenishments, i.e. charging the battery. Trade-offs in performance are therefore possible against replenishment frequency. Embedded systems may also contain energy scavenging components [13] producing unlimited energy but at a limited rate.

In [14] the aforementioned VBS technique is extended to included power-aware behaviour that affects frequency and voltage scaling. This is achieved using off-line and on-line techniques to detect static and dynamic slack, in order to reduce the operating frequency to achieve 100% resource utilisation. The frequency is derived based on the utilisation of the application set, meaning that individual application's cannot affect their own power-management policies, or control the frequency at which they operate.

It is demonstrated in [15] how power management may be achieved for individual voltage/frequency islands through the use of solely on-chip components, enabling fast transition times. In [16] a technique is described that takes advantage of fast transition times between high and low, voltage and frequency states, in order to approximate various frequency levels, and achieve a reduction in energy consumption.

Also in [4], software complexity is listed as a limitation of virtualisation for embedded systems. We address this by enabling composable application-level virtualisation, with composable temporal and energy budgets. As such, while we may not decrease the complexity of development of individual applications, we do decrease the complexity of integrating software from multiple sources on a single platform.

The concept of hierarchical power-management is described in [17], [18]. In [17] a two-tier hierarchical power-management approach is applied to shared hardware resources. One tier is at the system level and the other is at the component-level. Dynamism at the component-level is regulated by power-management policy at the system-level. A two-tier approach is also taken in [18], with the hierarchical power-management regulating a single hardware resource. The bottom hierarchical level contains multiple pre-computed power-management policies. The top hierarchical level selects which lower-level policy to use based on current device state.

In this paper we address the energy management limitation for embedded virtualisation by demonstrating how it may be achieved on a CompSoC [5], [6] hardware platform, running the CompOSe OS [7]. Our power-management system exists at a higher-level than in [17], [18]. Figure 1b illustrates that power management on our platform exists at the application-level, and interacts with application specific budgeting data stored in the RTOS, via an API interface. We demonstrate how applications may use energy budget information to enable power reduction through the use of DVFS.

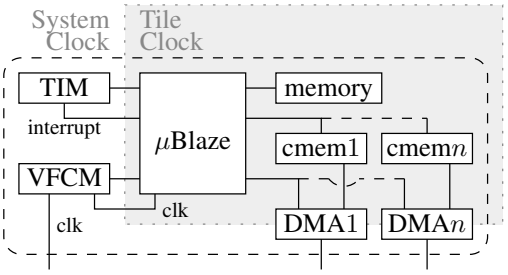


Figure 2. MicroBlaze (μBlaze) based processing tile.

III. Background

In [3] we demonstrated how application-level composable virtualisation may be achieved for functional and temporal properties. For this purpose CompSoC [5], [6] is used consisting of composable and predictable hardware and OS. The tile-based hardware is built around composable and predictable Æthereal NoC [19] and memory controllers [20].

Computational tiles consist of Xilinx μBlaze processors that communicate via the NoC through the use of a DMA [21]. This decouples and parallelises computation and communication. The tile is also equipped with a programmable Timed Interrupt Module (TIM), that sends an interrupt to the μBlaze at a precise programmed time in the future. A Voltage and Frequency Control Module (VFCM) enables programmable tile frequencies. The tile clock is either derived from a system clock (illustrated in Figure 2), or it can be locally generated. In our platform, the clock frequency in the system clock domain is constant at the maximum available frequency. The tile clock frequency is a scaled derivative of the system clock frequency, as programmed using the VFCM. Time in both system and tile clock domains is measured in clock cycles. As such, the progression of time in the tile clock domain is scaled in comparison to the system clock domain. When referring to time we use the following time domains:

- **Wall time** (t): Actual real-world time, measured in seconds.
- **System time** (c_{sys}): Time in the fixed-frequency system clock domain, measured in clock cycles at the system clock frequency (f_{sys}).
- **Tile time** (c_{tile}): Time in the variable-frequency tile clock domain, in clock cycles at the tile clock frequency (f_{tile}).

For real-time applications that interact with the environment, it is essential that the time observed inside a virtual platform can be translated into wall time. The time domains in clock cycles at a particular frequency can be converted to and from wall time using $t = c_{\text{sys}}/f_{\text{sys}}$ or $t = c_{\text{tile}}/f_{\text{tile}}$.

Per-application virtual platforms are enabled using the CompOSe OS [7] in combination with the CompSoC hardware. Applications are scheduled by the CompOSe OS following a TDM schedule. This schedule is regulated by a TIM that sends an interrupt to the processor, as illustrated in Figure 3. Once an interrupt is received by the OS a fixed-duration period of OS time is started, lasting $C_{\text{sys}}^{\text{OS}}$ cycles measured in system time. During this time the context of the previous application is stored and the next application is scheduled following the TDM schedule. The context of the scheduled application is restored, and, for composability [1], [7], the tile clock is gated until exactly $C_{\text{sys}}^{\text{OS}}$ cycles of time has passed.

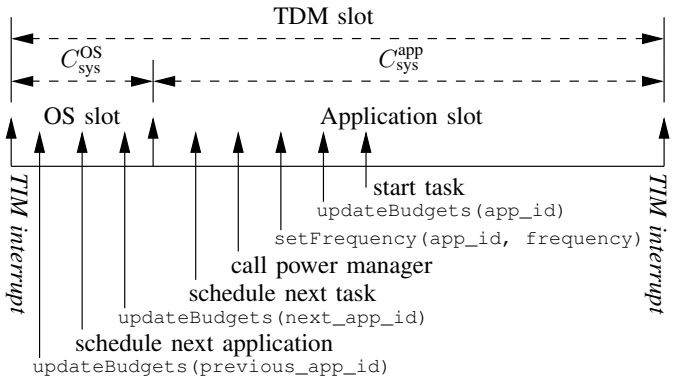


Figure 3. Example timeline for a single TDM table slot.

After the fixed-duration OS time has elapsed, a fixed period of application time begins, lasting $C_{\text{sys}}^{\text{app}}$ cycles, in system time. The application is executed in its virtual platform at a frequency specified by the application, with the application execution's temporal frame of reference being in tile time. The application executes until its current task is finished or until the interrupt arrives from the TIM. If the application finishes before the end of the application slot the tile clock is gated until the interrupt arrives. The interrupt signals another iteration of OS time. The OS time and task time continue to occur alternatively.

In [3] we implement OS-level power-management for real-time data-flow applications. Applications that follow a data-flow graph model like Synchronous Dataflow (SDF) [22], or Cyclo-static SDF (CSDF) [23], are split into tasks that may be represented as vertices in a graph, as illustrated for an H264 decoder in Figure 4. Each vertex v_n in the graph is connected to other vertices by FIFO communication edges. All communication between vertices happens across these edges. The task vertices are annotated with the worst-case work of the task, measured in cycles. The *worst-case work* is the maximum number of clock cycles required to complete the task's execution, independent of the clock frequency at which it is executed. Dynamic execution time information, enables dynamic variations in task execution time, i.e. a task finishing earlier than its worst case, to be translated into lower operating frequencies while maintaining the graph's real-time throughput requirements.

This policy schedules applications and also the application's tasks in the OS. The power-management of the application is also carried out in OS time. The OS time has a fixed duration and as such must bound the time taken to complete all of these things. Even though it is composable, due to the fixed-length OS time, an application with, e.g. complex task scheduling or power-management, will cause a greater fixed OS time overhead for all applications. By moving such application dependent overheads into application time, the OS slot can be shorter, enabling a higher utilisation of the processing core by applications, rather than OS administration.

We will now continue by explaining how power-management can be meaningfully lifted out of OS time and moved into application time, as illustrated in Figure 1.

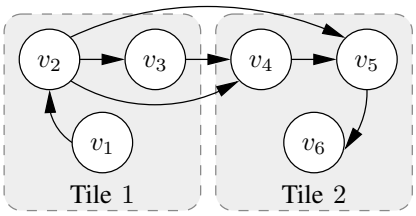


Figure 4. H264 decoder Application task graph, with tile mapping.

IV. Application-Level Power Management

Our previous work has shown how virtualisation may be used to achieve temporal and functional application-level composability. Integrating multiple mixed-time criticality applications for composable systems, can be achieved with relative ease. We propose extending the concept of application-level composability to also include the energy/power domain, thus enabling application designers to develop application specific power-management, while retaining the same ease of integration. To achieve this we propose an application-level power-management system that enables each application to define its own power manager and manage its energy, or power budget independently from other applications. We enable power-management policy to be specified per application, using API commands to interact with energy/power budgeting information that is maintained at the OS-level, as illustrated in Figure 1b. In the following sections we explain how power-management functions may be specified, how energy and power budgets may be specified and maintained per application, and finally how our budgeting technique is extended for a real-time power manager to use with a real-time application.

A. Power Managers

Application-level power-management policies are encapsulated in power-management functions. The user may use one of the provided power-management functions, or specify their own power-management function, to make use of the energy information however they see fit for their needs. We enable configuration of the power-management via an API interface. Application independent power-management functions are assigned to each application using the following interface:

```
setPowerManager(app_id, power_manager)
```

where `power_manager` is a pointer to a power-management function. Some example power-management functions for DVFS consist of:

- `Performance`: Execute the application at the maximum available frequency $f_{\max}$.
- `Conservative`: Execute the application at the maximum frequency permitted, while maintaining the specified power usage constraint.
- `Powersave`: Execute the application at the minimum available frequency $f_{\min}$.

These functions are called by the application, e.g. before scheduling a task, making the appropriate changes to the operating frequency. No restriction is placed on when, during an application slot the application's power-management function may be called. In the case of the `Conservative` power manager, energy budget information is used in the derivation of the appropriate frequency level, and is implemented as follows:

```
Conservative(app_id){
 freq = getMaxFrequency(remaining_energy);
 setFrequency(app_id, freq);
}
```

where `getMaxFrequency` represents an API call to obtain the maximum frequency at which the application can run, considering its remaining energy. This API command is power model specific, and is therefore beyond the scope of this paper.

In order to provide the power-management functions with information to make power-management decisions, we extend the already existing Application Control Block (ACB) information per application in CompOSe, to contain the following extra information:

- `energy_budget`: Energy allocation, in Joules.
- `power_budget`: Power allocation, in Joules per application slot.
- `energy_this_slot`: Current energy consumption this slot, in Joules.
- `remaining_energy`: Current total energy left, in Joules.
- `last_budget_update`: Last time this information was updated, in absolute time, in cycles system time.

The `energy_budget` is the amount of energy that an application has been allocated. Power is the rate of energy. For a power budget, the `power_budget` value is the amount of energy allocated per application slot. Both energy and power budgets may be specified at the same time. The `energy_this_slot` keeps track of the amount of energy consumed during a single application slot. The `remaining_energy` value is initially set equal to the `energy_budget` and is depleted as the application consumes energy by decrementing the `energy_this_slot` value at the end of each application slot. The `remaining_energy` value may also be augmented in the case of a power budget when there is no simultaneous energy budget. In this instance the `remaining_energy` is incremented following each slot by the `power_budget`. It is essential to keep track of when the budget information was last updated, so that the appropriate changes may be applied the next time the budget information is updated. To achieve this, the `last_budget_update` value stores the time, in system time, when the budget information was last updated. ACB budget information is available to the application via API "get" functions, that effectively make the data read-only.

The power management functions use the application's budget information to derive a DVFS operating frequency that is set using the `setFrequency(app_id, frequency)` API call. The `setFrequency` OS-level API command is called by the application-level power-management function to change the operating frequency, and is implemented at as follows:

```
setFrequency(app_id, frequency){
 frequency = power_model_check(frequency);
 if(frequency < fmin){
  frequency = fmin;
```

```
 } else if(frequency > fmax) {
  frequency = fmax;
 }
 updateBudgets(app_id);
 setVFCM_frequency(frequency);
}
```

This function checks against the processor power model, represented by `power_model_check`, to find out if the application has got enough power allocated to sustain the frequency request. If the application has sufficient power, the frequency is changed to the requested level. If the application does not have sufficient power the closest frequency, for which the application has sufficient power, is set. The frequency is also checked to make sure that it is within the available DVFS frequency range `fmin` to `fmax`. The `updateBudgets` API command is called to update the application's budget information, and is described in more detail in Section IV-B. The `setFrequency` API command is not restricted to being called from within a power-management function, and can be called at any time by the application.

While these example power-management policies are relatively straight forward, the user has the ability to craft much more complex power-management functions that take advantage of budgeting information. Regardless of the complexity level, the power management functions execute composably in the application's space, on the application's own time budget. Other application's executing concurrently on the platform are unaffected by the power-management decisions made by the application.

B. Maintaining the Budget Information

The budgeting information on which the power-management functions rely must be initialised and kept up-to-date. At the application's start, the `energy_budget` and `remaining_energy` value are set equal to E_{app}. The budget information is kept up-to-date using the `updateBudgets(app_id)` API command. Budget information is updated before the start, and after the end of each application slot, and whenever the application power consumption changes, e.g. a change in voltage and frequency levels requested by the application. A power model is required to obtain energy values in order to update the budget information, e.g. for our platform that supports DVFS we use a table storing the power that the tile consumes at each (voltage)-frequency level. An implementation of the OS-level API function, looks as follows:

```
updateBudgets(app_id){...
 if(slot_starting){
  last_budget_update = slot_start_time;
  return;
 }
 if(slot_ended){
  elapsed=slot_end_time-last_budget_update;
 } else {
  elapsed = system_time-last_budget_update;
 }
 energy = elapsed*f_to_energy[f_tile];
 energy_this_slot += energy;
 remaining_energy -= energy;
 if(slot_ended && power_budget
    && !energy_budget){
  remaining_energy += power_budget;
 }
 last_budget_update = system_time;
...}
```

Budget updates may occur during application time, e.g. whenever the frequency is changed using `setFrequency` function, or in OS time, initiated by the OS before and after every application slot. If a slot is starting, `slot_starting`, the `last_budget_update` is set to the slot start time, and the function returns without updating any other budget information. This is because the application has not consumed energy or performed work while it has not been scheduled. If the slot has ended, `slot_ended`, and the application has a power budget, `power_budget`, then the `remaining_energy` is incremented by the `energy_budget`, energy per slot amount. The application is not necessarily scheduled in every application slot. As such `elapsed` is the elapsed time, in system time, that the application has been scheduled since the last budget update.

The `energy` that has been consumed since the `last_budget_update` in system time, is calculated by multiplying the `elapsed` system time with the per-cycle of system time energy value. This is obtained from the `f_to_energy` table, that is indexed using the tile frequency between budget updates, `f_tile`. The `remaining_energy` is decremented by the `energy` amount, while `energy_this_slot` is incremented by the same amount.

C. Specifying Energy and Power Budgets

An application may have an energy budget, a power budget, or both. This is specified at the OS-level by an application use-case manager, the details of which are beyond the scope of this paper. The use-case manager assigns budget information to the application via API commands. An energy budget may be specified for an application using the following API call:

```
setEnergyBudget(app_id, energy)
```

where `app_id` is the application's enumerated identifier, and `energy` is a quantity of energy in Joules. This specifies a fixed quantity of energy for the application to consume. The value `energy_budget` is assigned the value of `energy`. The `remaining_energy` is set equal to the `energy_budget`, and will not be replenished. A power budget is specified per application using the following interface:

```
setPowerBudget(app_id, power)
```

where `power` is the requested average power level per application slot, in Watts. This power level is translated into the per application slot `power_budget`, as per the application's allocation in the application TDM scheduling table. For `power` P allocated to an application, the application's slot energy budget E_{app} is calculated as follows:

$$T_{\text{TDM}} = S_{\text{TDM}} \times \frac{C_{\text{sys}}^{\text{OS}} + C_{\text{sys}}^{\text{app}}}{f_{\text{max}}} \tag{1}$$

$$E_{\text{TDM}} = P \times T_{\text{TDM}} \tag{2}$$

$$E_{\text{app}} = \frac{E_{\text{TDM}}}{S_{\text{alloc}}} \tag{3}$$

where in Equation 1, C_{sys}^{OS} and C_{sys}^{app} are the durations of the OS slot and application slot respectively, in cycles in system time. S_{TDM} is the number of TDM slots in the TDM table. T_{TDM} is the period of the TDM table, in seconds. In Equation 2, E_{TDM} is the energy that is consumed at power P in one TDM table period, in Joules. The power level P only applies to the application it was assigned. As such, the entire E_{TDM} energy can be used by the application, during its allocated TDM slots. The application's energy budget E_{app} is therefore obtained by dividing E_{TDM} by the number of slots allocated to the application in the TDM table S_{alloc}, as shown in Equation 3.

D. Real-Time Extension

We continue by extending the application-level power-management of the last section to include support for Dataflow modelled real-time applications, such as the H264 decoder illustrated in Figure 4. Applications are split into tasks that are represented as vertices in the graph. In CompOSe task information is kept in a Task Control Block (TCB) with an enumerated ID, in a similar manner to how the application information is stored in an ACB. Task scheduling is performed on the application's virtual platform, using the application's task scheduler. Temporal accounting is maintained at the task-level using the following information in the TCB's.

- `time_budget`: Duration of time in system time that the task has been allocated for completion.
- `remaining_time_budget`: Current duration of time in system time before the task must be completed.
- `wc_work`: Maximum number of cycles required to complete the task.
- `remaining_work`: Current number of cycles left before the task is complete.

In order to maintain the application throughput requirement, each task must complete its execution within its `time_budget`. For this budget to be feasible it must be greater than, or equal to, the task's `wc_work`. During task execution, the `remaining_work` value tracks the work in cycles a task must still complete before it finishes its current iteration. Before the beginning of each iteration of the task's execution, the task's `remaining_work` is set equal to the task's `wc_work`, and the task's `remaining_time_budget` is incremented by the task's `time_budget`.

We extend the `updateBudgets()` API command, from Section IV-B, to maintain the temporal budget of the currently executing task. As before, budget information is updated at the end of each application slot and whenever a change to the processor's frequency level is requested. The temporal budget information is updated as follows:

```
updateBudgets(app_id){...
 elapsed = system_time-last_budget_update;
 work = (elapsed*f_tile)/f_system;
 remaining_time_budget-= elapsed;
 remaining_work-= work;
...}
```

where `elapsed` is the elapsed system time since the last budget update, `f_tile` is the tile frequency between updates and `f_system` is the system frequency. The elapsed time in system time is decremented from the `remaining_time_budget`. The elapsed time is translated from system time to the quantity of work actually performed in tile time.

Once the task completes its execution, the task's `remaining_work` is set to zero, while any `remaining_time_budget` is left over as slack. We provide an example real-time power-management function, that takes advantage of this slack accumulation, as follows:

- `RT_Powersave`: Execute the application at the minimum possible frequency, while still maintaining the application's real-time throughput requirement.

The `RT_Powersave` power-management function utilises static slack, e.g. from task `time_budget` over-dimensioning, and dynamic slack, e.g. from dynamic variation in task end times. This power-management function is implemented as follows:

```
RT_Powersave(app_id){
 work = remaining_work;
 budget = remaining_time_budget;
 frequency = (work/budget)*fmax;
 setFrequency(app_id, frequency);
}
```

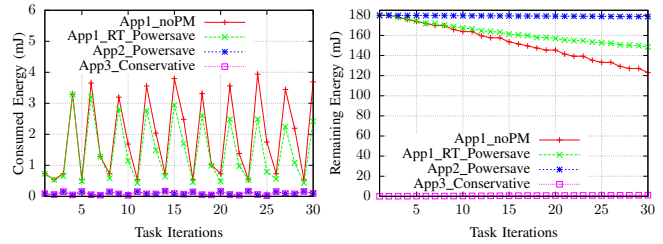
The `remaining_budget` value is only decremented by work that has been done. Any static or dynamic variation in the end time of a task, in comparison to its `time_budget`, results in the accumulation of slack. This enables processor frequency reduction while still maintaining the task's throughput requirement.

Power management functions are called by the application before beginning task computation, and may be called at any time by the application that is executing. As described in Section IV-A, the power manager functions are user specifiable, enabling more complicated policies that take advantage of the available temporal and energy budget information, described in this section. Information such as the `remaining_energy`, `remaining_time_budget` and `remaining_work` enable application designers to create real-time power-management policies, using run-time information.

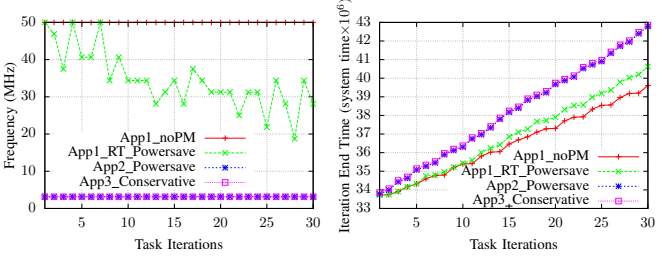
Application-level composable temporal and energy budgeting enables a platform that supports mixed time criticality applications, that utilise their budgeting information for independent power-management without affecting the temporal or power performance of other applications.

V. Experiments

We demonstrate the applicability of our technique through experimentation, on our CompSoC platform prototyped on a Xilinx ml605 FPGA board. For this purpose we use a version of our CompSoC hardware and with two processing tiles as the one illustrated in Figure 2. Each processor has a maximum frequency of 50 MHz and a modelled maximum power consumption of 1.5 Watts. We extend the existing CompOSe OS with temporal and energy budgeting extensions, as described in Section IV. On this platform we exercise a

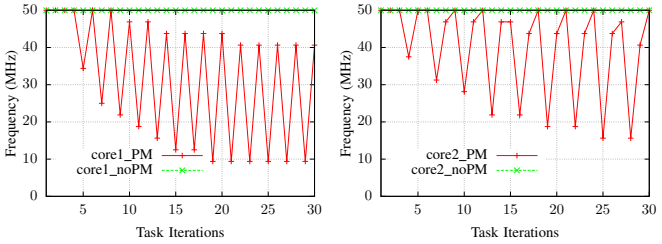


(a) Energy consumption per iteration. (b) Energy budget depletion.

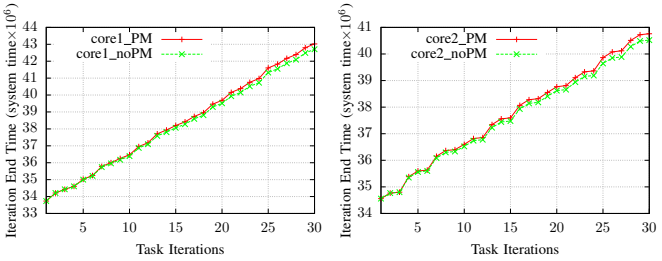


(c) Frequency. (d) Task iteration completion times.

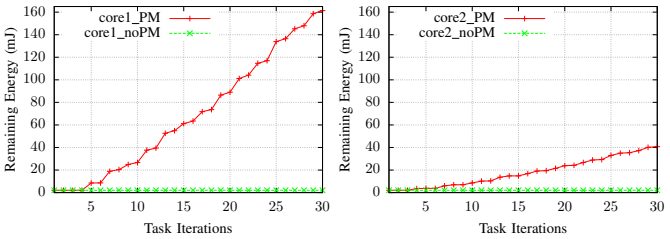
Figure 5. Synthetic application analysis.



(a) Core 1 frequency. (b) Core 2 frequency.



(c) Core 1 task iteration completion. (d) Core 2 task iteration completion.



(e) Core 1 unused power budget. (f) Core 2 unused power budget.

Figure 6. H264 decoder application analysis.

real-time h264 decoder application mapped on two tiles, as illustrated in Figure 4 and a synthetic application consisting of a chain of 5 tasks, with task 1, 3, and 5 mapped on one tile and the rest on the other tile.

To investigate the composable integration of our application-level energy budgeting information, we instantiate the synthetic application 3 times concurrently using different power-management policies for each of the executing applications. Each application is allocated a single slot in the application TDM table, with a length of 3. Applications 1 to 3 are assigned the `RT_Powersave`, `Powersave`, and `Conservative` power managers, respectively. Application 1 and 2 are allocated the same fixed *energy budget*, while application 3 is allocated a *power budget*.

With this configuration, we carry out 4 experiments, measuring application-level energy consumption, budget depletion, frequency level and application graph iteration completion times. All measurements are made per application task iteration completion. To investigate if composable integration is achieved, each experiment is carried out 4 times, once with each of the 3 applications executing alone, and once with the composition of all three applications. For application-level composability to have been achieved, the difference between running an application independently and after integration should always be zero, which is the case for all tested scenarios. However, due to lack of space we do not display all those graphs. The results for the composition of all three applications (App 1, App 2, App 3) and the case in which the synthetic application is executed with no power management policy (App1_noPM) are presented in Figure 5.

For application 1, with the `RT_Powersave` power manager, the energy consumption per iteration, displayed in Figure 5a, fluctuates in concordance with the application's frequency level, shown in Figure 5c. The `RT_Powersave` power manager is targeted at real-time applications by using temporal budgeting information to exploit accumulated slack, in order to reduce the processor frequency, while still meeting the application's throughput requirement. As slack is accumulated a general trend of decreasing frequency level and energy consumption can be seen. This is reflected in application 1's energy budget depletion, as can be seen in Figure 5b. Application 1's energy budget depletes at a continuously lower rate, as the frequency level tends to decrease. Application 1's graph completes at a higher throughput than applications 2 and 3, which is observable in the lower gradient sloping line in Figure 5d. This is as expected, considering application 1's frequency is continuously higher than the other two applications.

The results for application 2, in Figure 5c show that it executed continuously at the minimum frequency, which is in accordance with the `Powersave` power manager that it used. Due to this, application 2's energy consumption per iteration varies very little, as observable in Figure 5a, with any variation being due to dynamic variation in task completion times. Due to the application's continuous usage of the minimum frequency level, its energy consumption is relatively small in comparison to that of application 1. This is reflected in Figure 5b, where both application 1 and 2 started with the same energy budget, application 2 depletes its budget at a lower rate.

In Figure 5d, it can be seen that applications 2 and 3 have a very similar temporal profile. Both applications are identical,

have the same allocation in the application TDM scheduling table, and execute continuously at minimum frequency. Even though application 3 executes continuously at minimum frequency, it does this using the `Conservative` power manager. This power manager uses power budgeting information to calculate the maximum sustainable frequency level that an application can use for execution, given its current budget. A sustainable frequency is one that consumes less power than the budgeted power for the application. As is visible in Figure 5b, application 3 is assigned a relatively low power budget. The `Conservative` power manager calculates that the minimum frequency is the highest sustainable frequency.

For our second investigation we utilise the H264 decoder application. The results of this investigation are displayed in Figure 6. With this configuration, 3 experiments are carried out for each of the 2 cores, measuring frequency level, task iteration completion time and unused power budget. Each of these experiments is carried out twice, once for with the `RT_Powersave` power manager, and once without any power-management. In both cases energy accounting is still maintained. Each core is assigned a power budget that is the exact requirement to sustain the application's execution at maximum frequency.

Figures 6a and 6b show the frequency levels of both cores. It can be seen in the graphs how the `RT_Powersave` power manager scales the H264 decoder's frequency level, in response to dynamic variations in task completions. When the H264 application is executed without any power-management it continuously executes at maximum frequency. Figures 6c and 6d indicate that, as expected, when no power management is applied, the tasks of the H264 decoder complete earlier than when the `RT_Powersave` power manager is applied.

The reduced frequency levels, of the power managed H264, translates into reduced power consumption. Figures 6e and 6f display the accumulated remaining energy that is left from the power budget after each task iteration. The H264 decoder that runs without power-management continuously runs at maximum frequency, and therefore always consumes the power budget it is assigned, as can be seen from the horizontal line it creates in both graphs in Figures 6e and 6f. The H264 decoder with the `RT_Powersave` power-management, runs predominantly below maximum frequency and therefore does not consume all the energy it receives from its power budget. This can be seen as energy accumulating in Figures 6e and 6f. Both cores do not accumulate energy at the same rate due to the variations in task worst-case work and dynamic variations in execution.

In summary, with our experimentation we have verified that composable power-management has been achieved, provided a power-management policy analysis for our example policies, and demonstrated the behaviour of our example real-time power-management policy with an H264 decoder application.

VI. Conclusion

In this paper we extend application-level composability to encapsulate the independent power-management of an application, with energy and/or power budgets per application. Using a composable and predictable hardware platform and OS we demonstrate how this may be achieved for an MPSoC running mixed-time criticality applications. Our technique enables application developers to independently design and verify applications, and application power management policies for a virtual platform specification. When these independently developed applications, and their associated power management, are combined on a single platform, the composition of applications will not affect the application's temporal or power profiles, in comparison to when they were verified independently.

To achieve this we propose an API for setting power management policies, assigning and updating energy and/or power budgets and we describe its implementation for the CompOSe real-time operating system. Furthermore, we present four examples of power managers, one of them targeting the real-time domain. Experimental analysis on an MPSoC prototype on FPGA demonstrates that our technique provides application-level composable power management.

References

[1] B. Akesson *et al.*, "Composability and predictability for independent application development, verification, and execution," in *Multiprocessor System-on-Chip Hardware Design and Tool Integration*. Springer, 2010.

[2] H. Kopetz, *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Kluwer, 1997.

[3] K. Goossens *et al.*, "Composable dynamic voltage and frequency scaling and power management for dataflow applications," in *DSD*, 2010.

[4] G. Heiser, "The role of virtualization in embedded systems," in *IIES*, 2008.

[5] A. Hansson *et al.*, "CoMPSoC: A template for composable and predictable multi-processor system on chips," *TODAES*, vol. 14, no. 1, 2009.

[6] A. Molnos *et al.*, "A composable, energy-managed, real-time MPSoC platform," in *OPTIM*, 2010.

[7] A. Hansson *et al.*, "Design and implementation of an operating system for composable processor sharing," *Microprocessors and Microsystems*, vol. 35, no. 2, 2011.

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

[9] D. Chisnall, *The definitive guide to the xen hypervisor*. Prentice Hall Press, 2007.

[10] S. Craciunas *et al.*, "Programmable temporal isolation in real-time and embedded execution environments," in *IIES*, 2009.

[11] ——, "Everyone virtualizes everything but time," Poster at RTAS'09, 2009.

[12] ——, "Programmable temporal isolation through variable-bandwidth servers," in *SIES*, 2009.

[13] S. Chalasani *et al.*, "A survey of energy harvesting sources for embedded systems," in *Southeastcon*, 2008.

[14] S. Craciunas *et al.*, "Power-aware temporal isolation with variable-bandwidth servers," in *EMSOFT*, 2010.

[15] M. Meijer *et al.*, "On-chip digital power supply control for system-on-chip applications," in *ISLPED*, 2005.

[16] P. Vivet *et al.*, "On line power optimization of data flow multi-core architecture based on vdd-hopping for local dvfs," in *Integrated Circuit and System Design. Power and Timing Modeling, Optimization, and Simulation*. Springer, 2011.

[17] P. Rong *et al.*, "Hierarchical power management with application to scheduling," in *ISLPED*, 2005.

[18] Z. Ren *et al.*, "Hierarchical adaptive dynamic power management," *IEEE Trans. on Comp.*, 2005.

[19] K. Goossens *et al.*, "The Aethereal network on chip after ten years: Goals, evolution, lessons, and future," in *DAC*, 2010.

[20] B. Akesson *et al.*, "Architectures and modeling of predictable memory controllers for improved system integration," in *DATE*, Mar. 2011.

[21] J. Ambrose *et al.*, "Composable local memory organisation for streaming applications on embedded MPSoCs," in *CF*, 2011.

[22] E. Lee *et al.*, "Synchronous data flow," *Proc. of the IEEE*, vol. 75(9).

[23] G. Bilsen *et al.*, "Cyclo-static dataflow," *IEEE Trans. on Sig. Proc.*, vol. 44, no. 2, 1996.