A Composable, Energy-Managed, Real-Time MPSOC Platform

Anca Molnos, Jude Angelo Ambrose,
Andrew Nelson, Radu Stefan, Sorin Cotofana,
Computer Engineering, Delft University of Technology
Email: a.m.molnos@tudelft.nl

Kees Goossens
Eindhoven University of Technology
Email: k.g.w.goossens@tue.nl

***Abstract*—Multi-processors systems on chip (MPSOC) platforms emerged in embedded systems as hardware solutions to support the continuously increasing functionality and performance demands in this domain. Such a platform has to execute a mix of applications with diverse performance and timing constraints, i.e., real-time or non-real-time, thus different application schedulers should co-exist on an MPSOC. Moreover, applications share many MPSOC resources, thus their timing depends on the arbitration at these resources. Arbitration may create inter-application dependencies, e.g., the timing of a low priority application depends on the timing of all higher priority ones. Application inter-dependencies make the functional and timing verification and the integration process harder. This is especially problematic for real-time applications, for which fulfilling the time-related constraints should be guaranteed by construction. Moreover, energy and power management, commonly employed in embedded systems, make this verification even more difficult. Typically, energy and power management involves scaling the resources operating point, which has a direct impact on the resource performance, thus influences the application time behaviour. Finally, a small change in one application leads to the need to re-verify all other applications, incurring a large effort. Composability is a property meant to ease the verification and integration process. A system is composable if the functionality and the timing behaviour of each application is independent of other applications mapped on the same platform. Composability is achieved by utilising arbiters that ensure applications independence. In this paper we present the concepts behind a composable, scalable, energy-managed MPSOC platform, able to support different real-time and non-real time schedulers concurrently, and discuss its advantages and limitations.**

I. Introduction

Embedded systems include ever increasing functionality: (1) applications demand more performance from the underlying hardware, and (2) a growing number of diverse independent applications have to be executed simultaneously. Applications may have *diverse performance demands*, e.g., high throughput for video, low throughput for the user interface, and *diverse timing constraints*, i.e., non-real-time low latency for software tasks on processors, and real-time but latency tolerant for video, and low jitter for audio. Moreover, applications may have variable performance requirements during their execution, and a system may have many modes or use cases in which only a subset of the applications are active.

Multi-processor systems on chip MPSOC, the state-of-the-art hardware platforms for embedded devices [1] often include multiple processors, an advanced interconnect, and a memory hierarchy. The cost of an MPSOC — a crucial aspect for a successful embedded device — is both static, e.g., chip area, bill of material, design effort, and dynamic, e.g., power and energy consumption.

Once the number and type of MPSOC resources has been decided, the static cost has been paid. Often resources (processors, interconnect, memories) are shared between applications. Then the dynamic cost should be minimised by efficiently making use of the resources. For real-time applications the resources are allocated for the worst-case application behaviour. But due to application variability, the worst and the average case behaviour may be quite different, leading to an under-utilisation of the platform. An idle resource that is not switched off wastes energy, which is a dynamic cost. Given that, most embedded MPSOCs hence offer means and methods to *reduce the energy and/or power consumption*.

To reduce the design cost, and be able to easily reuse resources and/or applications inside a product family or between products generations without large costs the platform and the design process should have certain properties [22]. An important one is scalability, which has two aspects. (1) *Scalable design process* means that the cost of designing, implementing, verifying, and integrating an MPSOC platform increases linearly with the number of resources or the number of applications executing on the platform. (2) *Scalable performance* meaning that the performance delivered by the MPSOC platform increases linearly with its (static) cost.

A scalable design process is challenging to achieve because applications interfere when sharing the MPSOC platform. The arbitration at shared resources cause inter-application interference, e.g., the timing of a low priority application depends on the timing of all higher priority ones. Thus, timings of applications are often inter-dependent when they share resources. This is especially problematic when some applications have real-time requirements, such as a minimum throughput or a maximum response time. MPSOC verification is difficult if applications interfere with each other, because the integrated system may have unforeseen timing behaviour, or even errors that were not present in any of the constituent applications when running alone on the platform. The complexity of traditional monolithic functional and temporal verification of the system is exponential in the numbers of

applications, and when an application is added or changed the entire integrated system has to be re-verified. Moreover, often different applications are designed, implemented, and verified in isolation by different parties (e.g., independent software vendors). When the integrated system exhibits unexpected behaviour or errors that none of the parts had, it may be hard to determine the culprit, even leading to legal liability issues. Last but not least, temporal verification is even more difficult in combination with energy management. Energy management typically involves changing the operating points of resources at run time, which influences the performance and timing of applications.

Composability [17] is a system property aimed to alleviate monolithic verification and integration. Here, a system is composable if the functionality and timing of an application is independent of other applications. Composability enables independent application development and incremental MPSOC verification and integration. It is achieved by utilising special arbiters that guarantee independence of applications when sharing resources. Composability also makes the resulting system more robust because there is no interference from other uncharacterised (unknown), failing, or misbehaving applications.

In this paper propose an MPSOC platform that (1) is composable, (2) it is able to execute a mix of real-time and non-real-time applications, each scheduled according to its suitable policy, and (3) supports per-application energy management, again suited for the application type. Our MPSoC consists of a set of processor tiles and a set of memory tiles connected via a Network on Chip. With this platform in mind, we discuss the concepts behind a composable, energy-managed MPSOC platform, its advantages and limitations and we reflect on composability's implications on performance scalability. The rest of the paper is organized as follows. Section II introduces general properties of embedded applications, then Section IV discusses techniques to achieve composability, and implications of composability in MPSOCs. Section VI concludes the paper.

II. Embedded applications

In this section we present the application types that run on an embedded MPSOC, and how they are scheduled and energy managed.

A. Application types

Each application consists of a set of concurrent communicating tasks, implemented in hardware or software. Applications are assumed to be independent. Based on timing constraints one can identify three types of applications: firm real-time, soft real-time, and non real-time, each described below.

Firm real-time (FRT) applications are applications for which missing a deadline causes an unacceptable quality degradation. Examples of such applications are audio decoders and software radios. Mapping FRT applications on a set of resources demands formal analysis (e.g., dataflow analysis [26], real-time calculus [5], combined state-based and functional methods [28], to mention just a few). Such analyses take into account worst-case conditions (e.g worst-case execution time of tasks, worst-case input data arrival) and provide application mapping and resource allocation (e.g., budgets, priorities, memory sizes, etc.) that guarantee that the application deadlines are met at run-time. Resources are allocated for the worst case behaviour of the application.

Soft real-time (SRT) applications are applications for which occasionally missing a deadline causes a quality degradation that is acceptable. Examples of such applications are some video encoders and decoders, where a missed deadline can be mitigated by repeating a previous picture. SRT applications are usually designed using statistical analysis or based on a representative set of inputs. Application mapping and resource allocation aim to ensure that the application meets its deadlines in almost all cases. In this case resources are often allocated to cover the worst case of (all tasks of) the application, rather than the worst case of each individual task. Inside an application, the slack (unused capacity) of a task may be utilized by another task. This lowers the risk of application missing its deadline because an easy task may finish early allowing a hard task to start earlier and still finish in time.

Finally, an embedded system may execute applications with no deadlines whatsoever: non real-time (NRT) applications. Example of such applications are user the interface, file downloads, web browsing, etc. In this case it is important to maximize the overall throughput of the system.

B. Application scheduling

FRT tasks are typically scheduled statically, or else a static schedulability analysis is performed to determine under which workload conditions the dynamically-scheduled system is guaranteed not to miss a deadline. Typically FRT scheduling aims to maximize the resource utilisation while (1) guaranteeing that a set of periodic and aperiodic tasks are schedulable, and (2) fulfilling the application's latency and/or throughput demands. Typical scheduling policies are: (1) based on priorities, e.g., Earliest Deadline First (EDF), Rate Monotonic (RM), Static Priority (SP); (2) based on time sharing, e.g., Round Robin, Time Division Multiplexing (TDM); and (3) a combination of these, e.g., Credit Controlled Static Priority [2].

SRT task scheduling is typically dynamic, to allow the use of slack (unreserved or unused resource capacity) that can only be detected at run time. The same type of schedulers are utilized (i.e., priority, time-sharing, combined) but the scheduler's optimisation criteria are different: minimise average latency (or maximise average throughput) while minimising the number of missed deadlines [19].

Non real-time task scheduling is typically dynamic and time-sharing based. Here the purpose is to maximise the overall throughput of the system, to ensure fairness among applications and no starvation.

In summary, to execute a mix of applications of different types on shared hardware resources, different types of

	FRT applications	SRT applications	NRT applications
requirements	no deadline misses	limited deadline misses	no deadlines
scheduler type	priority-based, time-sharing	priority-based, time-sharing	time-sharing
scheduler optimisation	meet all deadlines, maximise utilisation	meet most deadlines, maximise utilisation	fairness, maximise average performance
energy management	static and proven dynamic slack, conservative	all slack, speculative	all slack, energy vs. performance trade-off

TABLE I
SUMMARY: APPLICATIONS' PROPERTIES & METHODS

schedulers with different optimisation criteria have to co-exist. Note, scheduler co-location is a technique server virtualisation often supports, however in embedded systems it should be implemented at much lower cost.

C. Run-time energy management

A large range of task-level energy management policies exists in the literature, for various application types and scheduling policies [9], [15], [33], [21], [34], [35], [20]. Most of them utilise Dynamic Voltage Frequency Scaling (DVFS) and/or resource shut-down to reduce dynamic (switching) and/or static (leakage) power. But changing the operating point of a resource also changes its speed, and thus affects the performance of an application utilising that resource. To be able to guarantee or achieve a certain performance level, energy consumption is often reduced by exploiting "slack," i.e., idling resources. Two general types of slack can be identified:

- Static slack that can be detected and computed at design time, e.g., originating from tasks that are not on the critical path that determines the application's throughput.
- Dynamic slack that occurs at run-time due to variations in the demands of applications or offered performance of the platform that are not observable at design time. For example, for some input data, tasks may have an execution time shorter than the worst case.

Energy management involves slowing down the resources, which with deadlines constraints, is often similar to real-time scheduling. The similarity is that both must take into account the entire system, i.e., all tasks and resources of an application, and end-to-end deadlines [14]. Hence energy management is frequently performed globally for the entire system, not per application or resource which would be more scalable. In the following we mention only several recent approaches for energy reduction.

For firm real-time applications, energy reduction is typically formulated as a minimisation problem under the constraint that deadlines are never missed. Since this must be conservative with respect to deadlines, static slack can be used. Dynamic slack can only be used after it has been generated; this is called proven slack [23], [24]. The most common approaches are intended for multi-processors executing a set of periodic tasks scheduled (preemptively or not) utilising Earliest Deadline First [9], [15], or for acyclic task-graphs [10], [37].

For soft real-time applications, the energy optimisation problem is different, as the goal is to minimise the energy such that the system delivers a certain average performance level, or that the deadlines are missed only sporadically. Speculations regarding future slack are acceptable and desired, as in this manner more energy can be saved. SRT energy-minimisation methods can be applied at design time, at run time, or both [21], [34], [35].

For non real-time applications deadlines are not defined, thus the energy optimisation problem is mostly tackled as a run-time energy versus performance trade-off [33], or as a energy minimisation problem while maintaining the average performance [20].

Recently energy management methods have evolved towards solutions for variable workloads and platforms. Applications or tasks with variable, different behaviours, and platforms that can adapt/reconfigure are considered. For example, [30] proposes a method to reduce energy consumption in a system executing a mix of periodic, aperiodic, and bursty tasks. [36] targets a system with a unified support for hardware, and operating point configuration methods. Energy-aware, real-time scheduling of tasks with different power characteristics are being looked at in [4] and [6] for a single-processor, and a multi-processor, respectively. The authors of [7] propose energy management for acyclic task graphs with input data variations.

As already mentioned, energy management in embedded systems is often applied globally, on a per system basis. However, the energy optimisation formulations are different for different application types. Hence a policy that minimises consumption for all types of different applications that might co-exist in a system is unlikely to exist, raising the need for a new approach.

D. Summary

Applications that execute on MPSOCs are diverse in terms of their performance demands (GOPS, memory footprint, bandwidth, latency, etc.). Their diverse (non) real-time requirements lead to different scheduling and energy management goals and techniques. Moreover, composability requires that the schedulers and energy managers eliminate any interference between applications. MPSOC platforms must therefore support concurrent FRT, SRT, and NRT applications, each with their own optimised scheduling and energy management. Inter-application scheduling must support composability, i.e., eliminate interference, while intra-application schedulers and energy managers are tailored to the type of the application. Table II-A summarises these findings.

Some initial approaches to support the execution of diverse applications on MPSOC exist. [16] presents a framework for design space exploration supporting heterogeneous scheduling for multimedia applications mapped on MPSOCs. The L4 μ-kernel [32] implements a virtualisation technology, based on a global scheduling policy interleaving priorities of threads from different subsystems/applications. The bandwidth sharing server [18] can schedule a combination of FRT and SRT applications, using EDF inter-application and EDF, RMO, or SPI intra-application. However, there is still a need for platforms and methods supporting (1) different task scheduling policies and (2) different energy management policies, that control not necessarily disjunct parts of the system at the same time. In the following sections we illustrate how our platform offers composable and predictable (real-time) scheduling and energy management at application and task level, hierarchically.

III. MPSOC PLATFORM

A. Architecture model

As a starting point we consider a general multi-processor tiled architecture, shown in Figure 1, as it is likely to offer the desired scalability. It follows the template introduced in [12], and we use a composable interconnect [13] and memory controllers [2], [3]. In the description below, we focus on the architecture and scheduling of processor tiles, since only these are currently energy managed (as described in Section V).

The processor and memory tiles may be heterogeneous; they may contain CPUs, VLIWs, DSPs, etc., or memories of various types, capacities, and speeds. Besides the processor, a processor tile contains a local memory that is also accessible by other processors, to implement point-to-point inter-processor communication, which is scalable and fast.

Like most embedded systems, we configure all tile memories as scratch pads, rather than caches. First, because it is hard to find realistic analytical worst-case execution times for real-time applications when tiles contain caches. Second because they are more expensive than scratch pads. Finally, cache coherence becomes relevant; it tends to be expensive when implemented in hardware, although software implementations are also possible. The impact of coherence must also be characterised for real-time applications. If caches are used, e.g., for NRT applications, then they must be flushed when switching between applications, to ensure composability. Of course, the disadvantage of scratch pads is that data and instructions must fit in the local memories, or they must be dynamically loaded.

B. Application model

The platform executes a mix of FRT, SRT, and NRT applications. Each application consists of a set of communicating tasks; there is no communication between different applications. We assume inter-task communication and synchronisation is explicit, for example via FIFO buffers, as in the C-HEAP protocol [27]. Each task executes infinitely often, consuming its input data (produced by other tasks), performing some computation on it, and then producing its output (to be processed by successor tasks). Figure 4 presents an example with two simple applications sharing the resources of the MPSOC. The tasks of HRT and SRT applications should have a bounded execution time. Dataflow analysis is used to prove that HRT and SRT applications always meet their end-to-end deadlines [26]. For non-real time neither bounded execution times nor formal analysis is needed.

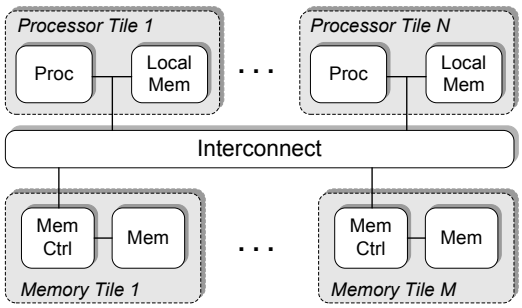

Fig. 1. General tiled-architecture template

IV. COMPOSABILITY

In this section we first describe the general method to achieve composability, which we then detail for the processor tile.

Composability is intended to isolate applications to enable their independent design, implementation, verification, and as a result ease system integration and verification. To achieve a composable system, each resource should be composable.

Eliminating the interference between applications (i.e., enforce zero interference), is most easily achieved by using Time Division Multiplexing (TDM) to create a static schedule of constant-duration time slots on the resource. Each application has a fixed resource budget with fixed starting times, as presented in Figure 2. Switching between applications is fast on some resources, such as SRAMs because different application make use of disjoint parts of the resource (the memory in SRAM case) and hence do not affect each other. Other resources, such as processors and SDRAMs, carry over state from the current uset to the next, and that affects the behaviour and performace of the next user. Examples of such state are cache contents or current memory bank. To avoid interference, the resource state must therefore be reset to a neutral or zero state when switching between applications. This means emptying the cache for a processor to avoid cache pollution, and switching to, e.g., bank 0 with write direction on the bus for an SDRAM [2].

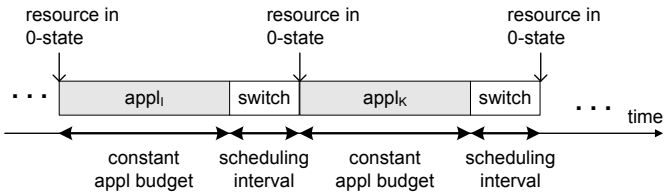

Fig. 2. Composability via zero interference

Preemption of shared resources is a pre-requisite for composability, because an application should not be allowed to monopolise or lock a resource for a duration longer that its budget (either intentionally or erroneously). Alternatively, we would have to rely on the fact that the requests are bounded. This is often not the case: e.g., control loops on a processor, transactions with unbounded and even infinite size. Moreover, this would artificially enforce worst-case sizes or execution times on NRT applications, if they share resources with FRT or SRT applications.

In addition, preemption is crucial because the scheduling interval, i.e., duration of OS between serving two applications, should be bounded and not depend on the applications running on the system. Otherwise the start times, and implicitly the temporal behaviour of an application would depend on the presence or absence of other applications in the system, compromising composability.

A. Processor composability

The processor operating system (OS) schedules different applications and tasks. For composability applications should be given constant-duration time slices (budgets) at regular intervals, to any dependence on other applications. This is achieved with a per-tile *timer* that generates an interrupt when a running task should be swapped out. The OS then determines which application is scheduled next, and then which task of that application. The execution time of the OS may well vary depending on the number of applications and tasks that are active. This is not composable, and hence the OS execution must be made of constant duration too, as described below.

The time to serve the interrupt when an application is running depends on the longest possible processor instruction that is in flight. This takes at most tens of cycles for state-of-the-art processors, except for instructions that involve other resources. In particular, load/store instructions from remote memories (not located in the tile), can take an arbitrary time to complete, depending on the resource allocation in the interconnect and remote memory. For NRT applications this time may be even unbounded. To bound the interrupt latency all load/store operations can be made local by using a Direct remote Memory Access (DMA) engine [25], resulting in the processor tile of Figure 3. In this case, when requiring remote data, the processor initialises a DMA transfer between the local and remote memories, and polls (via a local read) until the DMA finishes the transfer. Thus the processor is interruptible after each polling read. Hence the interrupt latency is kept short, and independent of the resource allocation of the application on other resources. Additionally, explicit DMA requests have to be added; but in most often these can be hidden in standard communication libraries, such as C-HEAP (see Section III-B).

To achieve composability DMAs, like any other resource, have to be composable, if shared. Since they are simple, cheap FSMs, we instantiate a DMA per application (or even per application FIFO) in each tile.

At this point, the maximum number of clock cycles that a time slice of a task on a processor uses has been bounded. As explained above, the number of cycles is not constant because it depends on the last instruction, which is interrupted. For composability, the time slice must be made of a constant duration. This is achieved by clock-gating or halting the processor until the worst-case length of a time slice [25], or idling by polling on a timer [8]. An important detail is that the timer that re-enables the processor should always be on and in the tile's clock domain. The same technique is used to ensure that the time the OS uses to schedule the next application and task is constant.

This scheme can be easily extended to support two-level scheduling: inter- and intra-application. The inter-application scheduler is TDM to ensure composability. The intra-application scheduler should fit the application type (FRT, SRT, NRT). It should execute in a known bounded time, for a bounded OS time slice.

B. Composability and scalability

Composability eliminates the effects of application interference, and thus clearly makes design process scalable.

Moreover, composability is implemented using an independent time-sharing scheduler (TDM) on each processor. This has two major advantages in relation with performance scalability: (1) It is not a centralised scheduler with global queues that could become a bottleneck when increasing the number of applications or processors, and (2) It potentially offers a processor utilisation up to 100%, increasing linearly with the number of applications [11].

The second property is not valid for priority-based schedulers. For example EDF, a common priority-based scheduler able to achieve 100% processor utilisation on a single processor platform does not scale. Its achievable utilisation for m processors = $m/(2m-1)$ [29]. Hierarchical priority-based schedulers have the potential to scale better, but they are complicated and involve global timing analysis and verification.

V. Composable energy management

As described until now, the platform used no dynamic voltage and frequency scaling (DVFS). The performance of all resources, in particular the processors, local memories, and DMAs, were constant per tile. This changes when DVFS is used: changing a frequency entails a change in performance (cycles/second, latency, etc.), and hence a change in actual

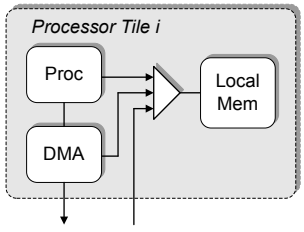

Fig. 3. Basic processor tile with DMA (logical view)

and worst-case execution times of tasks executing on the tiles. If an application can change the frequency of a resource, it can affect the performance of other applications, which is not composable, and real-time applications may even miss their deadlines.

For composable energy management, resources that are not shared between applications (as fisible in Figure 4 can be energy managed in any way the application sees fit. On the other hand, resources that are shared between applications should either have a constant frequency or the scheduling and energy management should be aligned and both be performed by the OS. Examples of the former are the NOC, shared memories on the NOC, etc. that run on their own constant frequencies. For processor tiles, the situation is more complex.

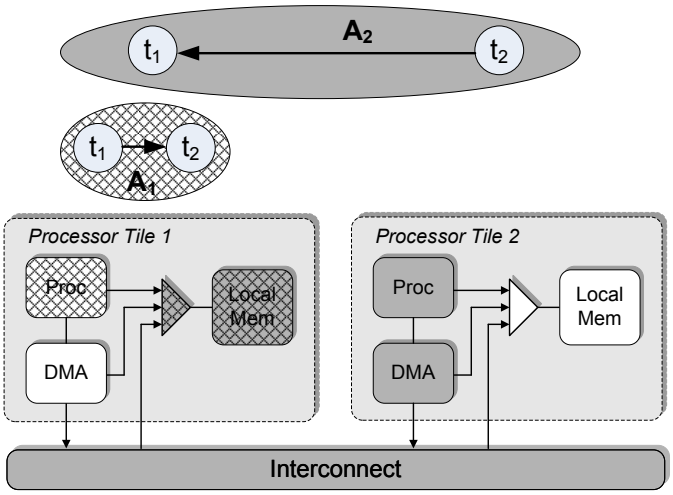

Fig. 4. Example: two application sharing MPSOC resources

Processors are energy-managed at scheduling granularity: the OS schedules an application task, and then decides the frequency it runs at, which depends on the type of the application (FRT, SRT, NRT). Before a task is swapped in the processor switches to the appropriate frequency, as described in Section II-C. The OS always runs at the maximum frequency to minimise its execution time (i.e., overhead). Hence, when the task execution is interrupted, the OS sets the frequency to the maximum, before it schedules and manages power/energy. Changing the frequency make take a variable (but bounded) amount of time; it is added to the variable interrupt time of the application and the variable execution time of the OS. Making use of the tile timers and processor clock gating, both application and OS time slices are therefore of a constant duration (in absolute wall time), even though the number of cycles in them may vary. To minimise the time wasted in frequency switches we use the techniques of [31].

The DMAs in the tile run in parallel with the processor, which apart from making remote load/store instructions interruptable, provides a performance benefit. As a result, a DMA for an application may be active and using the local scratch pad, while another application is running on the processor. The DMA can therefore not operate at the same frequency as the processor, since this would result in inter-application interference. It should either run at the frequency of its application, which entails giving all DMAs their own frequency domain, or better, run at a constant frequency. We chose the latter, and the DMA incorporates a clock domain crossing: it operates at the processor clock at its processor interface, and at the constant tile clock at its NOC interface. (The NOC converts the tile frequency to its internal frequency.) Note that the processor timer, used by the OS, also runs at the tile frequency because it provides a reference wall time, which does not change with the processor frequency.

The reasoning for the DMA also holds for the scratch pad. We use a scratch pad with two asynchronous ports. One port is connected to the processor with a variable frequency, and the other runs at the tile frequency and connects to the DMAs and incoming NOC ports from other tiles. Figure 5 presents the resulting clock domains in a processor tile.

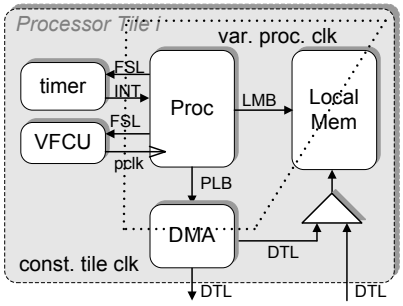

Fig. 5. Composable energy-managed tile

Our energy management is therefore performed per resource per application (or even task), rather than globally. Hence different policies can be used for each application, depending on its type.

VI. Summary

A system is composable if the functionality and timing of an application is independent of other applications. Composability is desired because (1) it enables independent application development and incremental MPSOC verification and integration at design time (scalable design process) and (2) allows diverse application schedulers (FRT, SRT, NRT) and energy managers to co-exist independently at run-time. In this manner both the static and the dynamic costs of designing an MPSOC are reduced. Moreover, composability improves performance scalability.

A platform is composabile if all its resources are composable. We utilised existing composable NOC and SDRAM memory controllers [12]. For the processor composabilily is achieved by strict Time Division Multiplexing with constant task slots, independent of other applications. Moreover, the duration between task slots, needed for the OS to schedule a next application task, should also be constant. The first can be implemented using timer interrupts and halting until the worst case interrupt latency time, and the second by halting the processor up to the OS worst case execution time.

Similar to task scheduling, in a composable system energy management must be performed per application rather than globally. To ensure composable energy management together with the application scheduling, the energy manager execution time should be bounded, as it is included in the OS worst case execution time. Moreover, the VF domains should be defined such that they do not introduce inter-application dependencies. Scaling the operating point of a resource utilised by an application, should not cause scaling the operating point of other resources utilised by other applications.

Although composability brings important advantages it also has its limitations:

1) Applications can not use interrupts since interrupts are utilised by the OS for task switching. Virtualising the interrupts would to alleviate this limitation.
2) Applications can not lock shared resources. If a resource is locked by an application, another application may be denied its budget on that resource.
3) The TDM arbitration couples latency and rate. Hence if an application demands a low response time, it should be allocated many slots. Potentially many of these slots will not be used by this application.
4) In a composable system slack can be utilised within applications, but not between them. This could potentially reduce the resource utilisation.

Conceptually, composability brings numerous advantages. A first implementation of the platform that supports per-application (different) schedulers, and per-application (but identical) energy management was realised on FPGA.

ACKNOWLEDGMENT

The research leading to these results has received funding from the ARTEMIS Joint Undertaking under grant agreement no. 100029 and from the Dutch local authority SenterNovem.

REFERENCES

[1] Nomadik - open multimedia platform for next generation mobile devices.
[2] B. Akesson, A. Hansson, and K. Goossens. Composable resource sharing based on latency-rate servers. In *DSD*, 2009.
[3] B. Akesson, L. Steffens, and K. Goossens. Efficient Service Allocation in Hardware Using Credit-Controlled Static-Priority Arbitration. In *RTCSA*, 2009.
[4] H. Aydin, R. Melhem, and et al. Determining optimal processor speeds for periodic real-time tasks with different power characteristics. In *ECRTS*, 2001.
[5] S. Chakraborty, S. Knzli, and L. Thiele. A general framework for analysing system properties in platform-based embedded system designs. In *DATE*, 2003.
[6] J.-J. Chen and T.-W. Kuo. Multiprocessor energy-efficient scheduling for real-time tasks with different power characteristics. In *ICPP*, 2005.
[7] J. Cong and K. Gururaj. Energy efficient multiprocessor task scheduling under input-dependent variation. In *DATE*, 2009.
[8] M. Ekerhult. COMPOSE, design and implementation of a composable and slack-aware operating system targeting a multi-processor system-on-chip in the signal processing domain. Master's thesis, Lund University, 2008.
[9] S. Funk, J. Goossens, and S. Baruah. Energy minimization techniques for real-time scheduling on multiprocessor platforms, 2001.
[10] M. Goraczko, J. Liu, D. Lymberopoulos, S. Matic, B. Priyantha, and F. Zhao. Energy-optimal software partitioning in heterogeneous multiprocessor embedded systems. In *DAC*, 2008.
[11] K. Goossens, A. Hansson. The Aethereal Network on Chip after Ten Years: Goals, Evolution, Lessons, and Future. In *DAC*, 2010.
[12] A. Hansson, K. Goossens, M. Bekooij, and J. Huisken. CoMPSoC: A template for composable and predictable multi-processor system on chips. *ACM Trans. on Des. Auto. of Elect. Sys.*, 2009.
[13] A. Hansson, M. Subburaman, and K. Goossens. Aelite: A flit-synchronous network on chip with composable and predictable services. In *DATE*, 2009.
[14] G. Heiser. The role of virtualization in embedded systems. In *IIES*, 2008.
[15] H.-R. Hsu, J.-J. Chen, and T.-W. Kuo. Multiprocessor synthesis for periodic hard real-time tasks under a given energy constraint. In *DATE*, 2006.
[16] M. Kim, S. Banerjee, N. Dutt, and N. Venkatasubramanian. Design space exploration of real-time multi-media mpsocs with heterogeneous scheduling policies. In *CODES+ISSS*, 2006.
[17] H. Kopetz. *Real-Time Systems: Design Principles for Distibuted Embedded Applications*. 1997.
[18] G. Lipari and S. K. Baruah. Efficient scheduling of real-time multi-task applications in dynamic systems. In *RTAS*, 2000.
[19] O. Florescu and M. De Hoon and J. Voeten and H. Corporaal Probabilistic Modelling and Evaluation of Soft Real-Time Embedded Systems. In *SAMOS*, 2006.
[20] M. P. M. Ghasemazar, E. Pakbaznia. Minimizing the power consumption of a chip multiprocessor under an average throughput constraint. In *ISQED*, 2010.
[21] P. Malani, P. Mukre, Q. Qiu, and Q. Wu. Adaptive scheduling and voltage scaling for multiprocessor real-time applications with non-deterministic workload. In *DATE*, 2008.
[22] G. Martin. Overview of the mpsoc design challenge. In *DAC*, 2006.
[23] A. Milutinović, K. Goossens, and G. Smit. Impact of power-management granularity on the energy-quality trade-off for soft and hard real-time applications. 2008.
[24] A. Molnos and K. Goossens. Conservative dynamic energy management for real-time dataflow applications mapped on multiple processors. In *DSD*, 2009.
[25] A. Molnos, A. Milutinovic, D. She, and K. Goossens. Composable processor virtualization for embedded systems. In *Workshop on Computer Architecture and Operating System Co-Design*, 2010.
[26] O. Moreira *et al.* Scheduling multiple independent hard-real-time jobs on a heterogeneous multiprocessor. In *EMSOFT*, 2007.
[27] A. Nieuwland, J. Kang, O. P. Gangwal, R. Sethuraman, R. Sethuraman, N. Busa, K. Goossens, and R. P. Llopis. C-Heap: A heterogeneous multi-processor architecture template and scalable and flexible protocol for the design of embedded signal processing systems. 2002.
[28] T. X. L. Phan, S. Chakraborty, P. S. Thiagarajan, and L. Thiele. Composing functional and state-based performance models for analyzing heterogeneous real-time systems. In *RTSS*, 2007.
[29] C. A. Phillips, C. Stein, E. Torng, and J. Wein. Optimal time-critical scheduling via resource augmentation (extended abstract). In *STOC*, 1997.
[30] R. Racu, A. Hamann, R. Ernst, B. Mochocki, and X. S. Hu. Methods for power optimization in distributed embedded systems with real-time requirements. In *CASES*, 2006.
[31] J. Rius, M. Meijer, and J. P. de Gyvez. An activity monitor for power/performance tuning of cmos digital circuits. *J. Low Power Electronics*, 2006.
[32] S. Ruocco. A real-time programmer's tour of general-purpose l4 microkernels. *EURASIP J. Embedded Syst.*, 2008.
[33] C. Xian, Y.-H. Lu, and Z. Li. Energy-aware scheduling for real-time multiprocessor systems with uncertain task execution time. In *DAC*, 2007.
[34] P. Yang, C. Wong, P. Marchal, F. Catthoor, D. Desmet, D. Verkest, and R. Lauwereins. Energy-aware runtime scheduling for embedded-multiprocessor socs. *IEEE Des. Test*, 2001.
[35] W. Yuan and K. Nahrstedt. Energy-efficient soft real-time cpu scheduling for mobile multimedia systems. In *SOSP*, 2003.
[36] G. Zeng, H. Tomiyama, H. Takada, and T. Ishihara. A generalized framework for system-wide energy savings in hard real-time embedded systems. In *EUC (1)*, 2008.
[37] D. Zhu, D. Mosse, and R. Melhem. Power-aware scheduling for and/or graphs in real-time systems. *IEEE Trans. on Parallel and Distributed Systems*, 2004.